REPÚBLICA BOLIVARIANA DE VENEZUELA

UNIVERSIDAD DEL ZULIA

FACULTAD EXPERIMENTAL DE CIENCIAS

DIVISIÓN DE PROGRAMAS ESPECIALES

DEPARTAMENTO DE COMPUTACIÓN

**LENGUAJE DE PROGRAMACIÓN PARA GRAFOS COMPACTOS**

Proyecto de trabajo especial de grado presentado como requisito para optar al título de Licenciado en Computación

Autor: Br. Simón Rafael Oroño Nava

Tutor: MSc. Gerardo Pirela

Maracaibo, junio del 2016

**LENGUAJE DE PROGRAMACIÓN PARA GRAFOS COMPACTOS**

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

Simón Rafael Oroño Nava

C.I. No.: 23.270.382

Teléfono: 04264601667

Urb. Altos del Sol Amado, 1era etapa, Casa #160

Correo electrónico: simonorono@protonmail.com

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

MSc. Gerardo Pirela

C.I. No.: 12.404.565

Correo electrónico: gepirela@fec.luz.edu.ve

**TABLA DE CONTENIDOS**

[RESUMEN 4](#_Toc449803915)

[ABSTRACT 5](#_Toc449803916)

[CAPÍTULO I 6](#_Toc449803917)

[1. Planteamiento y justificación del problema 6](#_Toc449803918)

[2. Objetivos de la investigación 7](#_Toc449803919)

[2.1. Objetivo general 7](#_Toc449803920)

[2.2. Objetivos específicos 7](#_Toc449803921)

[3. Alcance 8](#_Toc449803922)

[CAPÍTULO II 9](#_Toc449803923)

[1. Antecedentes 9](#_Toc449803924)

[2. Bases teóricas 10](#_Toc449803925)

[2.1. Compilador 10](#_Toc449803926)

[2.1.1. Estructura básica de un compilador 10](#_Toc449803927)

[2.2. Analizador sintáctico ALL(\*) 12](#_Toc449803928)

[2.3. ANTLR 12](#_Toc449803929)

[2.4. Grafo 12](#_Toc449803930)

[2.5. Lista de adyacencia 13](#_Toc449803931)

[2.7. Grado de un vértice 13](#_Toc449803932)

[CAPÍTULO III 14](#_Toc449803933)

[1. Metodología utilizada 14](#_Toc449803934)

[CAPÍTULO 4 17](#_Toc449803935)

[1. Descripción 17](#_Toc449803936)

[2. Grafos compactos 17](#_Toc449803937)

[3. Estructura gramatical 17](#_Toc449803938)

[4. Tipos 20](#_Toc449803939)

[ÍNDICE DE REFERENCIAS 22](#_Toc449803940)

Simón Rafael Oroño Nava. **Lenguaje de programación para grafos compactos.** Proyecto de Trabajo Especial de Grado. Universidad del Zulia. Facultad Experimental de Ciencias. División de Programas Especiales. Licenciatura en Computación. Maracaibo. Venezuela. 2016. 12pp.

# RESUMEN

El estudio de redes se ve limitado en la actualidad porque las herramientas desarrolladas para este fin consumen demasiados recursos computacionales. Esta investigación propone como solución a este problema el desarrollo de un lenguaje de programación para grafos compactos. Al finalizar el proyecto se espera tener la especificación de un lenguaje de programación para grafos compactos y un compilador que genere ejecutables nativos a partir de código fuente de este lenguaje. Se empleará la metodología de Programación Extrema para el desarrollo.

**Palabras Clave:** lenguaje de programación, grafos, compilador.

**Dirección electrónica:** simonorono@protonmail.com

Simón Rafael Oroño Nava. **Programming language for compact graphs.** Proyecto de Trabajo Especial de Grado. Universidad del Zulia. Facultad Experimental de Ciencias. División de Programas Especiales. Licenciatura en Computación. Maracaibo. Venezuela. 2016. 12pp.

# ABSTRACT

The study of networks is currently limited because the tools developed for this purpose consume too many computer resources. This research proposes as a solution to this problem to develop a programming language for compact graphs. Upon completion of the project it is expected to have the specification of a programming language for compact graphs and a compiler that generates native executables from source code in this language. The Extreme Programming methodology will be used for development.

**Keywords:** Programming Language, graphs, compiler.

**E-mail:** simonorono@protonmail.com

# CAPÍTULO I

**EL PROBLEMA**

## Planteamiento y justificación del problema

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 grandes (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 propuso 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.

## Objetivos de la investigación

## Objetivo general

Desarrollar un lenguaje de programación para grafos compactos.

## Objetivos específicos

* 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.
* Realizar pruebas funcionales del compilador.

## Alcance

Este trabajo consiste en desarrollar un lenguaje de programación con soporte para grafos en su sistema de tipos y en su librería estándar. Permitirá la manipulación de grafos no pesados, dirigidos y no dirigidos.

El compilador de este lenguaje de programación generará ejecutables nativos para la máquina virtual de Java (JVM), y serán compatibles con la versión 8 y posteriores de la misma.

Un programa sin método de entrada se considerará errado puesto que el compilador solo generará ejecutables, no librerías.

# CAPÍTULO II

**MARCO TEÓRICO**

## Antecedentes

Para realizar esta investigación se revisaron diversos proyectos en el área de lenguajes de programación y herramientas de recorridos de grafos, con el fin de agregar sus características más relevantes al lenguaje de programación que resultará de esta investigación.

Uno de los lenguajes de programación consultados fue Java, un lenguaje fuertemente tipado y con chequeo estático de tipos (Goslin y col. 2015). Diseñado por James Gosling y desarrollado por Sun Microsystems, cuya primera versión (Java 1.0) fue publicada en mayo del 1995 y su última versión mayor (Java 8) en marzo del 2014. Actualmente es desarrollado por Oracle Corporation. El código fuente de Java es compilado a un formato binario conocido como “class” que luego es interpretado por la Java Virtual Machine (Lindholm y col., 2015).

Otro lenguaje de programación con chequeo estático de tipos es Go, desarrollado por Google Inc. y diseñado por Robert Griesemer, Rob Pike y Ken Thompson. Las reglas de chequeo estático de tipos de Go no permiten conversiones implícitas de tipos (golang.org, 2016), a diferencia de Java.

El lenguaje a diseñar emulará el sistema de tipos de Java, con excepción de que contará con un solo tipo de número entero y un solo tipo de número flotante, y tendrá reglas de chequeo estático tan rigurosas como Go.

Como ejemplo de lenguaje de programación orientado puramente a grafos se encontró el lenguaje llamado GP (*Graph Programs*), un lenguaje de programación para la resolución de problemas que involucren grafos; se basa en reglas (declarativo) y estas se ejecutan de forma no determinista (Steinert, 2007). Para ejecutar programas escritos en GP se definió la York Abstract Machine (YAM), una máquina abstracta basada en pilas que interpreta instrucciones escritas en código YAM (Manning y Plump, 2011). El código GP debe ser previamente traducido o compilado a código YAM. Actualmente no existe ningún compilador o traductor para este lenguaje, tampoco una implementación de la máquina abstracta YAM.

A diferencia de GP, el lenguaje a desarrollar será imperativo, de ejecución determinista, y será ejecutado sobre la JVM.

## Bases teóricas

## 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

El analizador 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.

## Analizador sintáctico ALL(\*)

Los analizadores sintácticos ALL (*Adaptative LL*) son una especialización de los analizadores sintácticos LL. Los analizadores LL se suspenden en cada punto de decisión (no-terminal) hasta que su mecanismo de predicción haya elegido la producción apropiada a expandir, en cambio, los analizadores ALL ejecutan en cada punto de decisión tantos sub-analizadores en paralelo como producciones posibles y estos terminan tan pronto se detecta que no pueden cotejar la entrada restante; si al finalizar de cotejar la entrada, más de un sub-analizador está vivo para un no-terminal, se reporta un error por ambigüedad (Parr y col., 2014).

## 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).

## 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 ).

* 1. 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).

* 1. Distribución de grados

# CAPÍTULO III

**MARCO METODOLÓGICO**

## 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.

Uno de los aspectos más sorprendente de la Programación Extrema es la simplicidad de sus reglas, lo que lo convierte en un rompecabezas de pequeñas piezas que individualmente no tienen mucho sentido, sino acopladas unas con las otras (Wells, 2009). Estás reglas se aplican a cada una de las fases, y fueron adaptadas para este proyecto de la siguiente manera:

* Planificación

El proyecto debe dividirse en iteraciones y se deben planificar estas para poder tener una versión parcial o completa del producto al finalizarlas, por lo que, tanto el lenguaje como el compilador, fueron construidos de forma incremental; agregando características y funcionalidad cuando se cumplían una de dos condiciones:

* + La característica y funcionalidad en la que se estaba trabajando anteriormente se completó.
  + La característica o funcionalidad nueva era requerida por otra que ya estaba en desarrollo.
* Diseño

El diseño debe ser simple y debe ser reestructurado (refactorizado [sic]) donde sea y cuando sea con la finalidad de volverlo más simple de ser posible, por esto, para el desarrollo de este proyecto se siguieron estas reglas:

* El lenguaje debe ser lo más pequeño posible, sin agregar características que vuelvan innecesariamente complicado tanto al lenguaje como al compilador de este.
* El compilador estará compuesto por fases y sub-fases: una fase de verificación y otra de generación, ambas separadas en tantas sub-fases como sean necesarias de forma que cada una sea lo más simple posible.
* Implementación y pruebas

Durante la fase de implementación el código debe ser probado de forma intensiva, y puesto que las pruebas deben ser escritas antes que el código, se adoptó la medida de escribir fragmentos de código fuente (tanto correctos, como incorrectos) para probar la consistencia entre las reglas sintácticas y semánticas del lenguaje con la implementación del compilador.

# CAPÍTULO 4

**EL LENGUAJE DE PROGRAMACIÓN COMPACT GRAPHS**

## Descripción

El lenguaje de programación Compact Graphs (CG) es un lenguaje fuertemente tipado, con chequeo estático de tipos, se ejecuta sobre la Java Virtual Machine y está orientado al manejo de grafos compactos.

## 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.

## Estructura gramatical

La estructura gramatical del lenguaje está dada por la siguiente gramática escrita para ANTLR:

grammar CG;

ADD : '+' ;

ADD\_ASSIGN : '+=' ;

AND : '&&' ;

AND\_ASSIGN : '&&=' ;

BOOL : 'bool' ;

BREAK : 'break' ;

COMMA : ',' ;

CONTINUE : 'continue' ;

DIGRAPH : 'digraph' ;

DIV : '/' ;

DIV\_ASSIGN : '/=' ;

ELIF : 'elif' ;

ELSE : 'else' ;

EQUAL : '=' ;

EQUAL\_EQUAL : '==' ;

FLOAT : 'float' ;

FOR : 'for' ;

FUNC : 'func' ;

GE : '>=' ;

GRAPH : 'graph' ;

GT : '>' ;

IF : 'if' ;

INT : 'int' ;

LBRACE : '{' ;

LE : '<=' ;

LPAREN : '(' ;

LT : '<' ;

ML\_COMMENT : '/\*' (.)\*? '\*/' -> skip ;

MOD : '%' ;

MOD\_ASSIGN : '%=' ;

MUL : '\*' ;

MUL\_ASSIGN : '\*=' ;

NOT : '!' ;

NOT\_EQUAL : '!=' ;

OR : '||' ;

OR\_ASSIGN : '||=' ;

RBRACE : '}' ;

RETURN : 'return' ;

RPAREN : ')' ;

SEMI : ';' ;

STRING : 'string' ;

SUB : '-' ;

SUB\_ASSIGN : '-=' ;

VAR : 'var' ;

VOID : 'void' ;

WHILE : 'while' ;

LBRACK : '[' ;

RBRACK : ']' ;

WS: [ \t\r\n] -> skip;

BoolLit: 'true' | 'false';

fragment Letter: [a-zA-Z\_];

fragment DecimalDigit: [0-9];

Identifier: Letter (Letter | DecimalDigit)\*;

fragment DecimalLit: DecimalDigit+;

IntLit: DecimalLit;

fragment Exponent: [eE] [+-]? DecimalLit;

FloatLit: DecimalLit '.' DecimalLit Exponent?

| DecimalLit Exponent

| '.' DecimalLit Exponent?

;

fragment Escape: '\\' [tbnr"'\\];

fragment Char: ~[\\'"];

StringLit: '"' (Escape | Char)\* '"';

type: 'int'

| 'float'

| 'string'

| 'void'

| 'bool'

| 'graph'

| 'digraph'

;

edge: '[' source=StringLit ',' target=StringLit ']';

graphLit: gtype=('graph'|'digraph') '{' (edge (',' edge)\*)? '}';

arg: Identifier type;

argList: (arg (',' arg)\*)?;

atom: IntLit #Integer

| FloatLit #Float

| BoolLit #Boolean

| StringLit #String

| Identifier #VarName

| graphLit #Graph

| funcCall #FunctionCall

| type '(' expr ')' #Cast

;

expr: atom #Atomic

| <assoc=right> op=('-'|'+'|'!') expr #Unary

| '(' expr ')' #Assoc

| expr op=('\*'|'/'|'%') expr #MulDivMod

| expr op=('+'|'-') expr #AddSub

| expr op=('>'|'<'|'>='|'<=') expr #Comparison

| expr op=('=='|'!=') expr #Equality

| expr '&&' expr #LogicAnd

| expr '||' expr #LogicOr

;

exprList: (expr (',' expr)\*)? ;

glExpr: IntLit | FloatLit | BoolLit | StringLit ;

glVarDec: 'var' Identifier type ('=' glExpr)? ;

varDec: 'var' Identifier type ('=' expr)? ;

funcDef: 'func' Identifier '(' argList ')' type? '{' stmt\* '}' ;

funcCall: Identifier '(' exprList ')' ;

assignment: expr op=('='|'+='|'-='|'\*='|'/='|'%='|'&&='|'||=') expr ;

ifc: 'if' '(' expr ')' '{' stmt\* '}' elifc\* elsec? ;

elifc: 'elif' '(' expr ')' '{' stmt\* '}' ;

elsec: 'else' '{' stmt\* '}' ;

forc: 'for' '(' initial=assignment ';' cond=expr ';' mod=assignment ')' '{' loopStmt\* '}' ;

whilec: 'while' '(' expr ')' '{' loopStmt\* '}' ;

controlStmt: wr=('continue'|'break') ';' ;

returnStmt: 'return' expr? ;

compoundStmt: ifc

| forc

| whilec

;

simpleStmt: varDec

| assignment

| expr

| returnStmt

;

stmt: simpleStmt ';'

| compoundStmt

;

loopStmt: (stmt | controlStmt) ;

init: (funcDef | glVarDec ';')+;

## Tipos

CG cuenta con los siguientes tipos:

|  |  |  |
| --- | --- | --- |
| Palabra clave | Valor por defecto | Descripción |
| int | 0 | Entero sin signo de 32 bits |
| float | 0.0 | Número de punto flotante con 32 bits de precisión |
| string | “” | Cadena de caracteres |
| void | --- | Sin tipo |
| bool | false | Valor booleano |
| graph | graph { } | Grafo no dirigido vacío |
| digraph | digraph { } | Grafo dirigido vacío |

Fuente: Oroño, 2016

# ÍNDICE DE REFERENCIAS

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

golang.org (2015): **The Go Programming Language Specification.** [Disponible en] https://golang.org/ref/spec. Consultado marzo 2016

Goslin, J., Joy, B., Steele, G., Bracha, G. y Buckley, A. (2015) **The Java® Language Specification. Java SE 8 Edition.** [Disponible en] http://docs.oracle.com/javase/specs/jls/se8/html/index.html. Consultado marzo 2016

Lindholm, T., Yellin, F., Bracha, G. y Buckley, A. (2015) **The Java® Virtual Machine Specification. Java SE 8 Edition.** [Disponible en] http://docs.oracle.com/javase/specs/jvms/se8/html/index.html. Consultado marzo 2016

Manning, G. y Plump, D. (2008). **The York Abstract Machine.** [Disponible en] https://www.cs.york.ac.uk/plasma/publications/pdf/ManningPlumpGT-VMT.06.pdf. Consultado marzo 2016

Marsh, J. (2014): **An Overview of Extreme Programming.** [Disponible en] https://blog.udemy.com/extreme-programming/. Consultado enero 2016.

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

Parr, T. (2012): **The Definitive ANTLR 4 Reference.** Primera edición. Texas, Estados Unidos. The Pragmatic Bookshelf. 322pp.

Parr, T., Harwell, S. y Fisher, K. (2014): **Adaptive LL(\*) Parsing: The Power of Dynamic Analysis.** [Disponible en] www.antlr.org/papers/allstar-techreport.pdf. Consultado marzo 2016

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.

Steinert, S. (2009): **The Graph Programming Language GP.** [Disponible en] <https://www.cs.york.ac.uk/ftpdir/reports/2007/YCST/15/YCST-2007-15.pdf>. Consultado marzo 2016

Wells, Don. (2009): **Extreme Programming.** [Disponible en] http://www.extremeprogramming.org/