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, septiembre 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

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

# ÍNDICE DE CONTENIDOS

[RESUMEN 3](#_Toc455920991)

[ABSTRACT 4](#_Toc455920992)

[ÍNDICE DE CONTENIDOS 5](#_Toc455920993)

[ÍNDICE DE TABLAS 8](#_Toc455920994)

[ÍNDICE DE ILUSTRACIONES 8](#_Toc455920995)

[INTRODUCCIÓN 9](#_Toc455920996)

[CAPÍTULO I 10](#_Toc455920997)

[1. Planteamiento y justificación del problema 10](#_Toc455920998)

[2. Objetivos de la investigación 11](#_Toc455920999)

[2.1. Objetivo general 11](#_Toc455921000)

[2.2. Objetivos específicos 11](#_Toc455921001)

[3. Alcance 12](#_Toc455921002)

[CAPÍTULO II 13](#_Toc455921003)

[1. Antecedentes 13](#_Toc455921004)

[2. Bases teóricas 14](#_Toc455921005)

[2.1. Compilador 14](#_Toc455921006)

[2.1.1. Estructura básica de un compilador 14](#_Toc455921007)

[2.2. Analizador sintáctico ALL(\*) 16](#_Toc455921008)

[2.3. ANTLR 16](#_Toc455921009)

[2.4. Grafo 16](#_Toc455921010)

[2.5. Lista de adyacencia 17](#_Toc455921011)

[2.7. Grado de un vértice 17](#_Toc455921012)

[CAPÍTULO III 19](#_Toc455921013)

[1. Metodología utilizada 19](#_Toc455921014)

[CAPÍTULO IV 22](#_Toc455921015)

[1. Descripción 22](#_Toc455921016)

[2. Grafos compactos 22](#_Toc455921017)

[3. Elementos léxicos 23](#_Toc455921018)

[3.1. Comentarios: 23](#_Toc455921019)

[3.2. Identificadores: 23](#_Toc455921020)

[3.3. Palabras reservadas: 23](#_Toc455921021)

[3.4. Operadores y delimitadores 23](#_Toc455921022)

[3.5. Literales enteros 24](#_Toc455921023)

[3.6. Literales flotantes 24](#_Toc455921024)

[3.7. Literales de cadena de caracteres 24](#_Toc455921025)

[4. Tipos 24](#_Toc455921026)

[5. Estado de error 26](#_Toc455921027)

[6. Variables 26](#_Toc455921028)

[7. Expresiones 27](#_Toc455921029)

[8. Conversiones de tipo 30](#_Toc455921030)

[9. Funciones 31](#_Toc455921031)

[10. Sentencias 32](#_Toc455921032)

[10.1. Asignaciones 32](#_Toc455921033)

[10.2. Sentencias de retorno 33](#_Toc455921034)

[10.3. Sentencia de control 34](#_Toc455921035)

[10.4. Sentencia de impresión. 34](#_Toc455921036)

[10.5. Sentencia de aserción 34](#_Toc455921037)

[10.6. Bloque if 35](#_Toc455921038)

[10.7. Bloque while 36](#_Toc455921039)

[10.8. Bloque for. 37](#_Toc455921040)

[11. Programas 37](#_Toc455921041)

[12. Librería estándar 38](#_Toc455921042)

[12.1. Control de errores 38](#_Toc455921043)

[12.2. Lectura por pantalla 38](#_Toc455921044)

[12.3. Tamaño de un grafo 38](#_Toc455921045)

[12.4. Existencia de nodos en un grafo 38](#_Toc455921046)

[12.5. Agregar nodos a un grafo 39](#_Toc455921047)

[12.6. Agregar aristas a un grafo 39](#_Toc455921048)

[12.7. Eliminar aristas de un grafo 39](#_Toc455921049)

[12.8. Existencia de aristas en un grafo 39](#_Toc455921050)

[12.9. Eliminar bucles de un grafo 40](#_Toc455921051)

[12.10. Caminos más cortos 40](#_Toc455921052)

[12.11. Clausura de transitividad 40](#_Toc455921053)

[CAPÍTULO V 41](#_Toc455921054)

[1. Opciones de línea de comandos 41](#_Toc455921055)

[2. Arquitectura del compilador 41](#_Toc455921056)

[2.1. Análisis léxico 41](#_Toc455921057)

[2.2. Análisis sintáctico 42](#_Toc455921058)

[2.3. Chequeo de globales 42](#_Toc455921059)

[2.4. Chequeo estructural 42](#_Toc455921060)

[2.5. Verificación de tipos 42](#_Toc455921061)

[2.6. Pre-generación 42](#_Toc455921062)

[2.7. Generación 42](#_Toc455921063)

[3. Diagrama de clases del compilador 43](#_Toc455921064)

[4. Ejemplo de uso del compilador 44](#_Toc455921065)

[5. Pruebas 44](#_Toc455921066)

[CONCLUSIÓN 45](#_Toc455921067)

[RECOMENDACIONES 46](#_Toc455921068)

[ÍNDICE DE REFERENCIAS 47](#_Toc455921069)

# ÍNDICE DE TABLAS

Tabla de tipos del lenguaje *CG* 24

Tabla de precedencia y asociatividad de operadores del lenguaje *GC* 28

Tabla de conversión de tipos del lenguaje *GC* 30

# ÍNDICE DE ILUSTRACIONES

Arquitectura del compilador *cgc* 41

Diagrama de clases del compilador *cgc* 43

# INTRODUCCIÓN

Las redes son una de las mejores formas de representar fenómenos naturales, es por ello que su estudio ha experimentado un auge desde la década de los 90; es por ello que muchas herramientas se han desarrollado para la manipulación, descripción y hasta dibujado de redes.

El siguiente trabajo se basa en el diseño de un lenguaje de programación imperativo para la manipulación de grafos. Este trabajo está estructurado de la siguiente manera:

Capítulo I – El problema: se plantean los problemas que tienen las herramientas actuales y que dan lugar a la necesidad de una nueva herramienta.

Capítulo II – Marco teórico: se exponen antecedentes significativos a esta investigación y además se definen conceptos claves de la teoría grafos y de desarrollo de lenguajes de programación y compiladores.

Capítulo III – Marco metodólogo: en este capítulo se define la metodología con la que se desarrollará tanto la especificación del lenguaje como un compilador para el mismo.

Capítulo IV – El lenguaje de programación *Compact Graphs* (*CG*): este capítulo consiste en la especificación de un nuevo lenguaje de programación, orientado al manejo de una nueva representación de grafos (grafos compactos), estricto y de tipado estático.

Capítulo V – El compilador *cgc*: se discute la arquitectura e implementación de un compilador para el lenguaje de programación *CG*.

# 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 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 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*. Los *tokens* tienen la siguiente estructura:

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

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.

# 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 etapas y fases: una etapa de verificación y otra de generación, ambas separadas en tantas fases como sean necesarias de forma que cada una sea lo más simple posible.
* Cualquier funcionalidad necesaria que vuelva más complejo al lenguaje o al compilador debe implementarse en forma de función de la librearía estándar.
* 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 IV

**EL LENGUAJE DE PROGRAMACIÓN *COMPACT GRAPHS***

## Descripción

El lenguaje de programación *Compact Graphs* (de ahora en adelante CG) es un lenguaje fuertemente tipado, con chequeo estático de tipos y está orientado al manejo de grafos compactos; en esta sección se presenta su especificación completa e independiente de cualquier implementación. Se muestran fragmentos de gramática libre de contexto en la notación aceptada por ANTLR. Una regla gramatical definida en una sección no se repetirá en caso de ser usada en otra.

En algunas secciones se presentan fragmentos de código fuente que deben ser considerados como pruebas a un compilador de *CG*. Si el compilador puede compilar y ejecutar estas pruebas, entonces se podrá aseverar que ese compilador es una implementación completa de *CG*. Estas pruebas se considerarán como pasadas si y solo sí durante su ejecución el programa no imprime nada en la salida estándar, siendo la única excepción, la prueba de la sentencia de impresión.

## 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 de los bits 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.

## Elementos léxicos

## Comentarios:

Los comentarios en CG sirven como forma de documentar código fuente y son ignorados por el compilador. Hay dos tipos de comentarios:

* Comentarios de línea: empiezan por la secuencia de caracteres // y se extienden hasta el final de la línea.
* Comentarios generales: empiezan por la secuencia de caracteres /\* y se extienden hasta la primera ocurrencia de la secuencia de caracteres \*/.

Un comentario de bloque es interpretado como espacio en blanco si se encuentra entre otros elementos léxicos.

## Identificadores:

Un identificador es un nombre que se le da a una variable o a una función. Se representa como una secuencia de una o más letras o dígitos. Un identificador debe comenzar por una letra. Los identificadores tiene la siguiente forma:

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

fragment DecimalDigit: [0-9];

Identifier: Letter (Letter | DecimalDigit)\*;

## Palabras reservadas:

Las siguientes palabras están reservadas y no pueden ser usadas como identificadores:

assert bool break continue digraph elif else false float for func

graph if int print return string true var void while

## Operadores y delimitadores

Las siguientes secuencias de caracteres son usadas como operadores y delimitadores:

+ - \* / % , ; = == != ! && || ( ) { } [ ] && || > <

>= <= += -= /= \*= &&= ||=

## Literales enteros

Un literal entero es una secuencia de dígitos decimales que representa un número entero. En notación de gramática de ANTLR:

fragment DecimalLit: DecimalDigit+;

IntLit: DecimalLit;

## Literales flotantes

Un literal flotante es una secuencia de dígitos decimales que representa un número flotante. Puede tener parte entera, decimal y exponente. Un literal de flotante está definido de la siguiente manera:

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

FloatLit: DecimalLit '.' DecimalLit Exponent?

| DecimalLit Exponent

| '.' DecimalLit Exponent?

;

## Literales de cadena de caracteres

Un literal de cadena de caracteres es una secuencia de caracteres delimitada por comillas dobles y que representa una yuxtaposición de caracteres. Puede contener caracteres de escape. En notación de gramática:

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

fragment Char: ~[\\'"];

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

## Tipos

El lenguaje de programacion CG soporta los siguientes tipos:

| Palabra reservada | Valor por defecto | Rango |
| --- | --- | --- |
| int | 0 | -2147483648 - 2147483647 |
| float | 0.0 | -3.4028235e38 - 3.4028235e38 |
| bool | false | false - true |
| void |  |  |
| string | “” |  |
| graph | graph [0] |  |
| digraph | digraph [0] |  |

Tabla de tipos del lenguaje *CG*. Fuente: propia, 2016.

En la gramática, los tipos están definidos de la siguiente forma:

type: 'int'

| 'float'

| 'string'

| 'void'

| 'bool'

| 'graph'

| 'digraph'

;

El tipo de dato int engloba al conjunto enteros con signo de 32 bits representables en complemento a 2.

void representa la ausencia de tipo de dato y solo aplica a funciones.

El tipo de dato float engloba todos los números en coma flotante de precisión simple compatible con el estándar IEEE 742 y almacenado en 32 bits.

El tipo de dato bool solo tiene dos valores posibles: true y false para verdadero y falso, respectivamente.

El tipo de dato string representa todas las cadenas de caracteres posibles. El tamaño máximo de una cadena de caracteres dependerá de la implementación al igual que la configuración regional (idioma).

Para representar grafos se emplean los tipos de dato digraph y graph que representan grafos compactos dirigidos y no dirigidos, respectivamente. En un literal grafo se especifica el tamaño de este y, opcionalmente, las aristas que contiene. En notación de gramática para ANTLR:

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

graphLit: ('graph'|'digraph') '[' expr ']' ('{' (edge (',' edge)\*)? '}')?;

Los nodos de un grafo están etiquetados por un número entero, por lo tanto, el máximo número teórico de nodos que puede contener un grafo es de 2147483647.

## Estado de error

Los programas escritos en *CG* mantienen un estado de error. Este estado de error puede ser consultado en cualquier momento durante la ejecución. Los errores posibles dependen de la implementación y representan errores en tiempo de ejecución, sin embargo, existen operaciones y funciones del lenguaje que, en caso de incurrir en error, requieren que la implementación actualice el estado de error.

Las operaciones que modifican el estado de error reinician el mismo y luego lo modifican si y solo si incurren en error. El estado de error no puede ser modificable por el programador, solo por operaciones que puedan incurrir en error.

Al comienzo de un programa el estado de error debe ser la ausencia de este. Como prueba:

|  |
| --- |
| func pruebaError() {  assert !error();  } |

## Variables

Una variable es el nombre que se le da a un espacio en memoria para almacenar un valor; el conjunto de valores que puede almacenar dicho espacio en memoria depende del tipo de la variable. Una variable puede ser global o local. Las variables locales solo son visibles desde el ámbito en el que son declaradas.

Las variables locales son visibles solo desde el ámbito en el que son declaradas y pueden ser inicializadas durante su declaración. En notación de gramática de ANTLR:

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

Las variables son identificadas únicamente por su nombre, por lo que declarar dos variables con el mismo identificador no está permitido. Las variables no pueden ser ocultadas por otra declaración en un ámbito inferior.

El tipo puede ser omitido y este será inferido del tipo de la expresión de inicialización. Si el tipo es omitido y la inicialización está errada o es omitida no se puede inferir el tipo de dato de la variable, por lo tanto, no está permitido.

Si se omite la inicialización la variable tomará el valor por defecto correspondiente a su tipo. Declarar una variable del tipo void no está permitido.

Las variables globales se declaran fuera de las funciones, son de ámbito global y estas son visibles y accesibles durante la ejecución de todo el programa. La declaración de variable global en notación de gramática de ANTLR:

glExpr: IntLit | FloatLit | BoolLit | StringLit ;

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

El tipo de dato en la declaración de variables globales es obligatorio. El valor inicial de las variables globales se establece durante el proceso de inicio del programa. Las variables globales solo pueden inicializarse con literales de tipo int, float, bool y string. De omitirse la inicialización en la declaración, una variable global tomará el valor por defecto correspondiente a su tipo de dato.

Un compilador debe pasar la siguiente prueba de inferencia de tipos:

|  |
| --- |
| func pruebaInferencia() {  var a = 4.6;  var b = 5;  var c = "Hello";  var d = graph [1];  /\* Los tipos de literales son inferidos por el compilador  por lo que probar igualdad de variables con tipo inferido  no debe resultar en error de compilación \*/  assert a == 4.6;  assert b == 5;  assert c == "Hello";  assert d == graph [1];  } |

## Expresiones

Una expresión es una secuencia de uno o más operandos y operadores. Las expresiones pueden ser usadas para calcular un valor o producir un efecto secundario. En *CG* las expresiones son consideradas como sentencias.Los operandos de una expresión pueden ser valores atómicos u otras expresiones.

Las expresiones están definidas en la gramática de la siguiente manera:

funcCall: Identifier '(' exprList ')' ;

atom: IntLit

| FloatLit

| BoolLit

| StringLit

| Identifier

| graphLit

| funcCall

| type '(' expr ')'

;

expr: atom

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

| '(' expr ')'

| expr ('\*'|'/'|'%') expr

| expr ('+'|'-') expr

| expr ('>'|'<'|'>='|'<=') expr

| expr ('=='|'!=') expr

| expr '&&' expr

| expr '||' expr

;

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

Un valor atómico es cualquier literal, resultado de llamada a función o resultado de una conversión de tipos.

El tipo de los operandos debe ser el mismo para operadores binarios, sin excepción. En caso de ser diferentes, el compilador debe reportar un error de incompatibilidad de tipos.

El orden de evaluación de expresiones obedece a la siguiente tabla de precedencia y asociatividad:

| Precedencia | Operadores | Asociatividad |
| --- | --- | --- |
| 1 | ! - + (Unarios) | Derecha |
| 2 | / \* % | Izquierda |
| 3 | + - |
| 4 | > < >= <= |
| 5 | == != |
| 6 | && |
| 7 | || |

Tabla de precedencia y asociatividad de operadores del lenguaje GC. Fuente: propia, 2016

Para los valores de tipo numérico (int y float) los operadores +, -, \*, /, % corresponden a la suma, resta, multiplicación, división y modulo, respectivamente; los operadores ==, !=, >, <, >=, <= se utilizan para comparar si un valor es igual, diferente, mayor, menor, mayor o igual y menor o igual que otro valor, respectivamente y producen como resultado valores de tipo bool.

El operador + con operandos del tipo string realiza una concatenación del valor al lado izquierdo del operador con el operando del lado derecho. También se puede probar la igualdad o desigualdad de cadenas de caracteres con los operadores == y !=, respectivamente.

El operador &&, cuando se aplica a operandos del tipo bool, dará como resultado true si y solo si ambos operandos tienen el valor true, si no, su resultado será false. El operador || también aplica a operandos del tipo bool y su resultado será true si y solo si al menos uno de los operandos tiene el valor true, si no, será false. La igualdad o desigualdad de valores el tipo bool también puede ser probada con los operadores == y !=, respectivamente. El operador unario ! invierte el valor de un operando del tipo bool.

Para los valores con tipo de dato graph o digraph el operador unario ! arroja un nuevo grafo del mismo tipo que el operando con el estado de sus aristas invertidos: si entre dos nodos no existe una conexión, se creará y si la conexión existe, se eliminará.

El operador binario && aplicado a operandos de tipo graph o digraph del mismo tipo produce como resultado un nuevo valor del mismo tipo de sus operandos y que contiene solo los vértices comunes entre ambos operandos. Si los operandos tienen distintos tamaños, se actualizará el estado de error y el resultado será un grafo vacío del mismo tipo que los operandos.

El operador binario || aplicado a operandos de tipo graph o digraph del mismo tipo produce como resultado un nuevo valor del mismo tipo de sus operandos y que contiene todos los vértices que están en cualquiera de los operandos. Si los operandos tienen distintos tamaños, se actualizará el estado de error y el resultado será un grafo vacío del mismo tipo que los operandos.

También puede probarse la igualdad o desigualdad de grafos con los operadores == y !=, respectivamente.

En esta sección se definieron pruebas para probar la asociatividad y precedencia de operaciones,

|  |
| --- |
| func pruebaExpresiones() {  var a = 3 + 4 + 5 \* 6 / 6 - 3;  var b = "He" + "llo";  var c = 2.5 + 9.6 / 2.0;  var d = digraph[2] {[0, 1]};  var e = !(1 == 2) && false || true;    assert a == 9;  assert (a % 2) == (7 % 6);  assert b == "Hello";  assert c == 7.3;  assert (d && !digraph[2]) == d;  assert (d || digraph[2]) == d;  assert e;  } |

## Conversiones de tipo

El lenguaje *CG* tiene soporte para convertir valores de un tipo de dato a otro. Las conversiones soportadas pueden visualizarse en esta tabla:

| Tipo origen | Tipo destino |
| --- | --- |
| int | float string |
| float | int string |
| string | int float bool |
| bool | string |

Tabla de conversión de tipos del lenguaje *GC*. Fuente: propia, 2016.

Las conversiones desde valores de tipo string pueden resultar en el valor por defecto del tipo destino y actualizar el estado de error en caso de darse un error de formato, por ejemplo, tratar de transformar letras en números.

Se definió la siguiente prueba para la conversión de tipos:

|  |
| --- |
| func pruebaConversion() {  var a int = int(3.4) + int("4") + 5;  var b = float(3) + 2.5 - float(int("1"));  var c = bool("true") || bool("false");  var d = string(1) + string(3.42) + string(true);  var e = int("hello"); // Error de formato  assert a == 12;  assert b == 4.5;  assert c == true;  assert d == "13.42true";  assert e == 0;  assert error();  } |

## Funciones

Las funciones son bloques de sentencias que, juntas, ejecutan una tarea. Las funciones son usadas para organizar código o simplemente para evitar repetición del mismo. En la gramática de *CG*, una función está definida de la siguiente forma:

arg: Identifier type;

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

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

Las funciones son identificadas únicamente por su nombre, por lo que declarar dos funciones con el mismo identificador no está permitido. Si en la declaración de la función el tipo es omitido se debe asumir que el tipo de retorno de la función es void.

Los argumentos de una función son considerados variables locales de la misma, por lo que declarar una variable con el mismo identificador de un argumento no está permitido.

Para toda función con tipo de retorno distinto a void el compilador debe garantizar que la función retorna un valor. Que al menos una de las sentencias en la función retorne un valor es garantía suficiente de que la función retorna.

Todas las pruebas están definidas en forma de función, por lo tanto, superar las pruebas es suficiente para demostrar el soporte del compilador a las funciones.

## Sentencias

Las sentencias son las instrucciones que controlan la ejecución de los programas. Estas solo pueden usarse dentro de funciones. Las sentencias pueden ser simples o compuestas, y están en la gramática de la siguiente manera:

compoundStmt: ifc

| forc

| whilec

;

simpleStmt: varDec

| assignment

| expr

| returnStmt

| controlStmt

| printStmt

| assertStmt

;

stmt: simpleStmt ';'

| compoundStmt

;

Toda sentencia simple terminan con el símbolo ;. Las sentencias compuestas están delimitadas por los símbolos { y }.

## Asignaciones

Una asignación es una operación que consiste en darle al elemento del lado izquierdo del operador el valor del elemento al lado derecho del operador. Las asignaciones tienen la siguiente forma:

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

Los operadores de asignación con la forma a operador= b son equivalentes a la operación a = a operador b.

El elemento del lado izquierdo de la asignación debe ser una expresión asignable, es decir, un identificador de variable, de no serlo, el compilador debe reportar como error que el lado izquierdo al operador de asignación no es asignable.

Si el tipo del lado izquierdo del operador de asignación no coincide con el tipo del lado derecho, el compilador debe reportar incompatibilidad de tipos durante la asignación.

Se definió como prueba de asignaciones a la siguiente función:

|  |
| --- |
| func pruebaAsignacion() {  var a int;  var b float;  var c string;  a = 354;  b = 3.14159265;  c = "Hello";  assert a == 354;  assert b == 3.14159265;  assert c == "Hello";  } |

## Sentencias de retorno

Una sentencia de retorno marca el final de una función. Las sentencias de retorno están definidas de la siguiente forma:

returnStmt: 'return' expr? ;

El tipo de dato de la expresión debe coincidir con el tipo de dato de la función. Si el tipo de la función en la que se encuentra una sentencia de retorno es void y la expresión está presente, el compilador debe reportar un error; lo mismo ocurre en el caso contrario, si la expresión no está presente y el tipo de retorno de la función es distinto a void. En funciones de tipo void, una sentencia de retorno sin expresión marca el fin de la función.

La prueba de las sentencias de retorno es la siguiente:

|  |
| --- |
| func retorno() int {  return 42;  }  func pruebaRetorno() {  assert retorno() == 42;  } |

## Sentencia de control

Existen dos tipos de sentencias de control: continue y break y sirven para controlar la ejecución dentro de un bucle. En la gramática están definidas como:

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

La sentencia continue obliga al programa a pasar a la siguiente iteración de un bucle, mientras que la sentencia break obliga al programa a abandonar la ejecución de un bucle.

Se definió una prueba para las sentencias de control:

|  |
| --- |
| func pruebaControl() {  var i int;  var r int;  for (i = 0; i < 10; i += 1) {  if (i > 7) { break; }  else if (i % 2 == 0) { r += 2; continue; }  else { continue; }  print "No debe imprimirse";  }  assert r == 12;  } |

## Sentencia de impresión.

La sentencia de impresión imprime datos a la salida estándar. Recibe una expresión como operando y esta puede ser de cualquier tipo de dato. En la gramática está definida de la siguiente forma:

printStmt: 'print' expr;

La prueba de la sentencia de impresión es el famoso “Hola, mundo”:

|  |
| --- |
| func pruebaImpresion() {  print "Hola, mundo. ";  } |

## Sentencia de aserción

Una sentencia de aserción es usada para comprobar que una condición siempre es cierta. Están definidas en la gramática como:

assertStmt: 'assert' expr;

La sentencia de aserción está compuesta por una expresión; esta expresión debe ser del tipo bool, si no, debe reportarse como error de compilación. Si el resultado de la expresión es false el programa debe terminar, reportando la línea de código fuente dónde se encuentra la aserción que ha fallada. Si el resultado es true el programa continuará con su ejecución normal.

Todas las pruebas están en función de la sentencia de aserción.

## Bloque if

El bloque if es una sentencia compuesta por otras sentencias y está formado por un bloque if, cero o más bloques elif y un bloque opcional else. En la gramática está definido de la siguiente manera:

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

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

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

Las expresiones de los bloques if y elif deben ser del tipo bool. Al ejecutar un bloque if, su expresión es evaluada; las sentencias de este bloque solo se ejecutaran si el resultado es true, si no, se pasa al siguiente bloque elif (en caso de existir) y su expresión es evaluada de la misma forma que la del bloque if. Si no se cumple ninguna de las condiciones de los bloques if o elif se ejecutan las sentencias del bloque else (en caso de existir).

Se considera que un bloque if retorna un valor si y solo si cumple las siguientes condiciones:

* Al menos una de las sentencias dentro del bloque if retorna.
* Tiene un bloque else y al menos una de las sentencias de este retorna.
* Para todo bloque elif que exista, al menos una de las sentencias de cada uno retorna.

El bloque if es la única sentencia compuesta que se puede garantizar que retorna un valor.

|  |
| --- |
| func pruebaIf() {  var r = 0;  var i = 16;    if (i % 15 == 0) { r = 3; }  elif(i % 5 == 0) { r = 2; }  else { r = 1; }  assert r == 1;  } |

## Bloque while

El bloque while es una sentencia compuesta y a su vez un bucle que se ejecuta siempre y cuando su condición (expresión que debe ser del tipo bool) se evalúe a true. En notación de gramática de ANTLR:

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

Al comienzo de un bloque while se evalúa la condición y las sentencias se ejecutan solo si esta evalúa a true. Las sentencias del bloque siempre y cuando la condición evalúe a true. Le ejecución del bloque while puede controlarse con las sentencias de control.

Al no poder garantizar en tiempo de compilación que un bloque while se ejecuta, este no se puede utilizar para garantizar que una función retorne.

La prueba del bloque while es una función que calcula la sumatoria de todos los números del 1 al 100:

|  |
| --- |
| func pruebaWhile() {  var r = 0;  var i = 1;  while (i <= 100) {  r += i;  i += 1;  }  assert r == 5050;  } |

## Bloque for.

El bloque for es una sentencia compuesta y a su vez bucle que se define con una inicialización (asignación), una condición de parada (expresión del tipo bool) y un modificador (asignación). Está definido en la gramática de la siguiente manera:

forc: 'for' '(' assignment ';' expr ';' assignment ')' '{' stmt\* '}' ;

Al comienzo de la ejecución de un bloque for se evalúa la inicialización y luego la condición; si la condición evalúa en true se continua con la ejecución de las sentencias. Al finalizar la ejecución de las sentencias se ejecuta el modificador y luego la condición. El bucle se repetirá hasta que la condición evalúe en false. Su ejecución se puede controlar también con las sentencias de control. Luego de cada sentencia continue se evalúa el modificador.

Al no poder garantizar en tiempo de compilación que un bloque for se ejecuta, este no se puede utilizar para garantizar que una función retorne.

La prueba del bloque for es una función que calcula el factorial del número 8:

|  |
| --- |
| func pruebaFor() {  var r = 1;  var i = 1;  for (i = 1; i <= 8; i += 1) {  r \*= i;  }  assert r == 40320;  } |

## Programas

En CG, un programa está formado por declaraciones de variables globales y funciones. Como requisito, se debe definir al menos una función de nombre “main”, con tipo de retorno void y que no recibe parámetros; está será el punto de entrada del programa. El programa valido más pequeño posible es el siguiente:

|  |
| --- |
| func main() {  } |

## Librería estándar

Una implementación del lenguaje de programación *CG* debe contener las funciones explicadas en esta sección. Las funciones se clasifican por funcionalidad y se representan de la siguiente manera:

nombre\_de\_función(argumentos) tipo\_retorno

Las funciones que apliquen a grafos y sus propiedades tienen dos versiones: una para grafos dirigidos y otra para grafos no dirigidos.

## Control de errores

error() bool : Devuelve true si el estado de error es distinto a “No hay error”; false en caso contrario.

serror() string : Devuelve el mensaje de error en forma de cadena de texto.

perror() void : Imprime el mensaje de error por salida estándar.

## Lectura por pantalla

read() string : Lee una línea de texto por pantalla y la devuelve como valor de tipo string. Si ocurre un error durante la lectura, debe retornarse la cadena vacía y actualizarse el estado de error.

## Tamaño de un grafo

g\_size(g graph) int

dg\_size(g digraph) int

Devuelve el número de nodos de un grafo. Recibe como parámetro el grafo cuyo tamaño se desea conocer.

## Existencia de nodos en un grafo

g\_contains\_node(g graph, n int) bool

dg\_contains\_node(g digraph, n int) bool

Sirven para probar la existencia de un nodo en un grafo. Recibe como parámetros un grafo g y un índice n del nodo cuya existencia se quiere probar.

## Agregar nodos a un grafo

g\_add\_nodes(g graph, n int) graph

dg\_add\_nodes(g digraph, n int) digraph

Recibe por parámetros un grafo g y un número de nodos n. El resultado de esta función es un nuevo grafo con una cantidad de nodos igual a la suma de los nodos de g más n. El nuevo grafo conservará los mismos arcos o aristas que g.

## Agregar aristas a un grafo

g\_add\_edge(g graph, a int, b int) void

dg\_add\_edge(g graph, a int, b int) void

Recibe por parámetros un grafo g y dos índices de nodo, a y b. Modifica el grafo g y agrega a este una arista desde el nodo a hasta el nodo b. Si la arista ya existe, la función no tiene efecto.

## Eliminar aristas de un grafo

g\_remove\_edge(g graph, a int, b int) void

dg\_remove\_edge(g graph, a int, b int) void

Recibe por parámetros un grafo g y dos índices de nodo, a y b. Modifica el grafo g y elimina de este una arista desde el nodo a hasta el nodo b. Si la arista no existe, la función no tiene efecto. Si alguno de los nodos no existe, la función no tiene efecto y se actualiza el estado de error.

## Existencia de aristas en un grafo

g\_contains\_edge(g graph, a int, b int) bool

dg\_contains\_edge(g graph, a int, b int) bool

Recibe por parámetros un grafo g y dos índices de nodo, a y b. Devuelve true si en el grafo g existe un arista desde a hasta b, si no, false. Si alguno de los nodos de un grafo no existe, devuelve false.

## Eliminar bucles de un grafo

g\_remove\_loops(g graph) void

dg\_remove\_loops(g digraph) void

Recibe como parámetro un grafo g y modifica este eliminando todos los bucles que contenga.

## Caminos más cortos

g\_shortest\_path(g graph, n int) graph

dg\_shortest\_path(g digraph, n int) digraph

Recibe como parámetro un grafo g y el índice n de un nodo. Devuelve un grafo nuevo con los caminos más cortos desde el nodo de índice n hasta los demás nodos alcanzables desde él. Si el nodo no existe, se devuelve un grafo vacío y se actualiza el estado de error.

## Clausura de transitividad

g\_transitivity(g graph) graph

dg\_transitivity(g digraph) digraph

Recibe como parámetro un grafo g y calcula la clausura de transitividad de este. Por cada nodo del grafo se creara un arista hasta otro nodo del grafo si y solo si el nodo es alcanzable desde .

# CAPÍTULO V

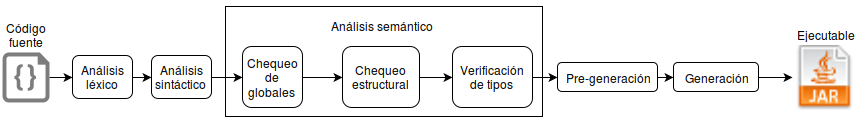
**EL COMPILADOR *cgc***

Como parte de este trabajo se diseñó e implementó un compilador para el lenguaje *CG* llamado *cgc.* Este recibe como parámetro de línea de comando la ubicación de un archivo de código fuente y crea ejecutables nativos para la máquina virtual de Java (JVM).

## Opciones de línea de comandos

* -c -check: el código fuente solo se verificará en busca de errores, no se generará ejecutable.
* -d -debug: imprimirá mensajes de depuración a la salida estándar durante el proceso de compilación.
* -h -help: muestra un mensaje de ayuda describiendo las opciones de línea de comandos y el uso del programa.
* -o -output: especifica el nombre del ejecutable a producir.

## Arquitectura del compilador



Arquitectura del compilador *cgc*. Fuente: propia, 2016.

El compilador cgc está dividido en siete etapas, estas transforman el código fuente hasta convertirlo en un ejecutable.

## Análisis léxico

Es la primera etapa de compilación, en ella el código fuente se separa en *tokens* que luego son enviados a la siguiente etapa. Esta etapa se genera a partir de la gramática de ANTLR.

## Análisis sintáctico

Recibe los *tokens* resultantes del análisis léxico y verifica que el orden de estos cumpla con la estructura gramatical del lenguaje. El resultado de esta etapa es un árbol de sintaxis abstracto (AST).

## Chequeo de globales

Durante esta etapa se recorre el AST resultante del análisis sintáctico y se meten en la tabla de símbolos las variables globales y funciones (con sus respectivos parámetros). La re-declaraciones y re-definiciones se reportan como errores de compilación.

## Chequeo estructural

Durante esta etapa se vuelve a recorrer el AST y se verifican que todas las sentencias de control (continue y break) se encuentran dentro de bloques for o while. También se verifica que todas las funciones con tipo de retorno distinto de void retornen un valor.

## Verificación de tipos

Durante esta etapa se realizan múltiples verificaciones para asegurar la consistencia del programa con respecto al estricto sistema de tipos de *CG*. Se verifican los tipos de todas las operaciones y expresiones así como la consistencia en las llamadas a funciones, sentencias de retorno y las expresiones de los bloques if, for y while.

## Pre-generación

En esta etapa se asigna un índice a cada variable declarada dentro de las funciones. Este paso es necesario pues la JVM requiere esta información para verificar los ejecutables. Desde esta etapa se asume que el programa no tiene errores.

## Generación

Durante esta etapa se crean los ejecutables del programa.

## Diagrama de clases del compilador



Diagrama de clases del compilador *cgc*. Fuente: propia, 2016

El lenguaje de programación utilizado fue Kotlin; a diferencia de Java, Kotlin soporta programación imperativa y funcional en forma de funciones de primer orden. Las funciones y clases usadas están organizadas en paquetes.

La clase *Compiler* es la encargada de preparar e iniciar el proceso de compilación. Esta contiene una instancia de la clase *State*, que a su vez contiene a la tabla de símbolo, los parámetros de compilación, índices de variables y errores. Los errores de compilación se programaron en forma de excepciones y estas se encuentran en el paquete *exception*.

En el paquete *internal* se encuentran las clases y funciones generadas por ANTLR, el compilador usa estas y las contenidas en el paquete *ast* para crear al árbol de sintaxis abstracto que luego usaran los objetos del paquete *phase*. En el paquete *type* se encuentran funciones para trabajar con los tipos de datos disponibles en *CG*, la tabla de operaciones unarias y binarias, y la tabla de posibles conversiones de tipo.

En el paquete *symbol* se encuentran las clases y funciones para trabajar con la tabla de símbolos.

En los paquetes *phase.generation* y *phase.globals* se encuentran funciones de ayuda utilizadas por las fases de generación y chequeo de globales, respectivamente.

## Ejemplo de uso del compilador

Si se tiene el siguiente archivo de código fuente ejemplo.grp:

|  |
| --- |
| /\*  \* Hola mundo  \*/  func main() {  print “Hola, mundo.”  } |

Puede compilarse y ejecutarse de la siguiente forma:

|  |
| --- |
| $ cgc ejemplo.grp  $ java –jar ejemplo.jar  Hola, mundo. |

## Pruebas

Para demostrar la consistencia con la especificación del lenguaje se desarrolló un programa que contiene todas las funciones de prueba con la siguiente función “main”:

|  |
| --- |
| func main() {  pruebaError();  pruebaInferencia();  pruebaExpresiones();  pruebaConversion();  pruebaAsignacion();  pruebaRetorno();  pruebaControl();  pruebaImpresion();  pruebaIf();  pruebaWhile();  pruebaFor();  } |

Cuya ejecución solo imprime la cadena “Hola, mundo.”:

|  |
| --- |
| $ cgc prueba.grp  $ java –jar prueba.jar  Hola, mundo. |

# CONCLUSIÓN

La revisión documental fue clave para determinar carencias en los antecedentes a esta investigación, de esta forma, dichas carencias no afectaron con el desarrollo de esta investigación.

Como resultado se obtuvo un lenguaje de programación llamado *Compact Graphs* (*CG*) que es imperativo, determinista, estricto, de tipado estático y que está orientado a la manipulación de grafos compactos; la formalización de esta forma de representar grafos fue clave en el propósito de hacer que el lenguaje consumiera la menor cantidad de recursos computacionales posible.

También se desarrolló un compilador para el lenguaje *CG*: una herramienta de línea de comandos llamada *cgc* y que produce ejecutables nativos para la máquina virtual de Java (JVM).

# RECOMENDACIONES

Al ser una herramienta orientada a la manipulación de grafos, el lenguaje de programación CG es ideal para la enseñanza de estos en el ámbito académico, por lo que se recomienda incluir en el programa de estudio.

La representación de grafos compactos aplica solo a grafos no pesados, por lo que se propone, para trabajos futuros, crear una representación que permita operar sobre grafos pesados usando pocos recursos computacionales y se incluya a este lenguaje.

# 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/