GOLD 3

Un lenguaje de programación imperativo para la manipulación de grafos y otras estructuras de datos

Autor

Alejandro Sotelo Arévalo
UNIVERSIDAD DE LOS ANDES

Asesora

Silvia Takahashi Rodríguez, Ph.D.
UNIVERSIDAD DE LOS ANDES

TESIS DE MAESTRÍA EN INGENIERÍA
DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y COMPUTACIÓN

FACULTAD DE INGENIERÍA
UNIVERSIDAD DE LOS ANDES
BOGOTÁ D.C., COLOMBIA
ENERO 27 DE 2012

⁂ ⁂ ⁂

A las personas que más quiero en este mundo:

mi madre Gladys y mi novia Alexandra.

⁂ ⁂ ⁂

Abstract

Para disminuir el esfuerzo en la programación de algoritmos sobre grafos y otras estructuras de datos avanzadas es necesario contar con un lenguaje de propósito específico que se preocupe por mejorar la legibilidad de los programas y por acelerar el proceso de desarrollo. Este lenguaje debe mezclar las virtudes del pseudocódigo con las de un lenguaje de alto nivel como *Java* o *C++* para que pueda ser fácilmente entendido por un matemático, por un científico o por un ingeniero. Además, el lenguaje debe ser fácilmente interpretado por las máquinas y debe poder competir con la expresividad de los lenguajes de propósito general.

GOLD (*Graph Oriented Language Domain*) satisface este objetivo, siendo un lenguaje de propósito específico imperativo lo bastante cercano al lenguaje utilizado en el texto *Introduction to Algorithms* de Thomas Cormen et al. [1] como para ser considerado una especie de pseudocódigo y lo bastante cercano al lenguaje *Java* como para poder utilizar la potencia de su librería estándar y del entorno de programación *Eclipse*.

Índice general

Lista de figuras

Lista de tablas

Lista de códigos

Parte I

Preliminares

Capítulo 1

Introducción

El uso de grafos como herramienta de modelaje es bastante frecuente en muchos campos de la ingeniería. Asimismo, se ha escrito mucho en relación con los algoritmos para manipular estas estructuras de datos. En la mayoría de los casos se requiere que quienes desarrollan estas aplicaciones usen un lenguaje de propósito general o herramientas limitadas para la definición de estas estructuras por medio de interfaces gráficas. Por otro lado, últimamente se han realizado diversas investigaciones en el desarrollo de lenguajes de propósito específico con miras a acercar más el lenguaje al usuario final, que no necesariamente es un experto programador. El desarrollo de un nuevo lenguaje no es de ninguna forma una tarea fácil pues no comprende únicamente su sintaxis sino también aspectos semánticos y pragmáticos. Este documento describe *GOLD* (*Graph Oriented Language Domain* por sus siglas en inglés), un lenguaje de programación de propósito específico diseñado para facilitar la escritura de algoritmos sobre estructuras de datos avanzadas como árboles, grafos y autómatas a través de una sintaxis muy cercana al pseudocódigo, basada en la notación matemática estándar que se usa en los libros de texto para manipular números, expresiones booleanas, conjuntos, secuencias y otros dominios de interés.

Este capítulo enuncia el contexto, la motivación y el propósito del proyecto *GOLD* en su versión 3, exponiendo las razones que justificaron su nacimiento y formulando su objetivo general.

1.1. Resumen

Para disminuir el esfuerzo en la programación de algoritmos sobre grafos y otras estructuras de datos avanzadas es necesario contar con un lenguaje de propósito específico que se preocupe por mejorar la legibilidad de los programas y por acelerar el proceso de desarrollo. Este lenguaje debe mezclar las virtudes del pseudocódigo con las de un lenguaje de alto nivel como *Java* o *C++* para que pueda ser fácilmente entendido por un matemático, por un científico o por un ingeniero. Además, el lenguaje debe ser fácilmente interpretado por las máquinas y debe poder competir con la expresividad de los lenguajes de propósito general.

GOLD (*Graph Oriented Language Domain*) satisface este objetivo, siendo un lenguaje de propósito específico imperativo lo bastante cercano al lenguaje utilizado en el texto *Introduction to Algorithms* de Thomas Cormen et al. [1] como para ser considerado una especie de pseudocódigo y lo bastante cercano al lenguaje *Java* como para poder utilizar la potencia de su librería estándar y del entorno de programación *Eclipse*.

1.2. Contexto

El principal precursor del proyecto *GOLD* lo constituye la tesis de maestría *CSet: un lenguaje para composición de conjuntos* [2], desarrollada por Víctor Hugo Cárdenas en el año 2008 bajo la dirección de Silvia Takahashi del grupo de investigación *TICSw* (Tecnologías de la Información y Construcción de Software) del Departamento de Ingeniería

de Sistemas y Computación de la Universidad de los Andes. *CSet* es un lenguaje de propósito específico que permite componer conjuntos de datos a través de una implementación extensible que provee ‘‘una gran flexibilidad en la definición de los tipos de datos’’ [2].

CSet buscaba convertirse en el punto de partida de otros proyectos que necesitaran un lenguaje especializado en la definición de conjuntos, en particular aquellos relacionados con el modelado de problemas en investigación de operaciones y con el desarrollo de lenguajes para la descripción y simulación en grafos. Teniendo en cuenta este precedente, el proyecto *GOLD* (*Graph Oriented Language Domain* por sus siglas en inglés) nació en el año 2009 como un lenguaje de propósito específico diseñado para expresar matemáticamente grafos en términos de conjuntos, desarrollado por Luis Miguel Pérez en la tesis de pregrado *GOLD: un lenguaje orientado a grafos y conjuntos* [3] bajo la asesoría de Silvia Takahashi del Departamento de Ingeniería de Sistemas y Computación de la Universidad de los Andes y de Andrés Medaglia del Departamento de Ingeniería Industrial de la misma universidad, en la línea de investigación de Modelado de Propósito Específico (*Domain Specific Modeling*) perteneciente al área de Métodos Formales (*MF*) del grupo de investigación de Tecnologías de la Información y Construcción de Software (*TICSw*) del Departamento de Ingeniería de Sistemas y Computación (*DISC*) de la Universidad de los Andes.

La tesis de Luis Miguel Pérez ofrecía un lenguaje descriptivo básico para definir grafos a través de la descripción matemática de su conjunto de nodos y de su conjunto de arcos, rescatando la definición formal que se encuentra en los libros de texto. En el año 2010 Diana Mabel Díaz, con su tesis de maestría *GOLD+: lenguaje de programación para la manipulación de grafos: extensión de un lenguaje descriptivo a un lenguaje de programación* [4] dirigida por Silvia Takahashi, extendió el lenguaje descriptivo diseñado por Luis Miguel Pérez para permitir la manipulación algorítmica de grafos con un lenguaje de programación sencillo basado en un conjunto limitado de instrucciones de control. Aunque *GOLD+* permitía describir grafos y realizar determinadas operaciones básicas entre éstos, no era un lenguaje lo suficientemente potente para implementar efectivamente algoritmos clásicos como el algoritmo de Dijkstra o para desarrollar grandes proyectos de software que requieren una manipulación exhaustiva de grafos.

Para solucionar las limitaciones de *GOLD+* se diseñó un lenguaje de programación completamente nuevo que permitiera la manipulación de grafos en grandes proyectos de software. Siguiendo la línea de producción de los desarrollos previos, primero con la tesis Luis Miguel Pérez (en adelante, *GOLD 1*) y después con la tesis de Diana Mabel Díaz (en adelante, *GOLD 2*), se continuó con la evolución del lenguaje en el presente proyecto de maestría, denominado *GOLD 3: un lenguaje de programación imperativo para la manipulación de grafos y otras estructuras de datos* (en adelante, *GOLD 3*). La última versión de *GOLD*, desarrollada en esta tesis, es un lenguaje de programación imperativo que puede utilizarse en cualquier proyecto *Java* bajo el entorno *Eclipse* para la escritura de algoritmos de alto nivel sobre una colección ilimitada de estructuras de datos (definidas en términos de conjuntos, bolsas y listas) con un lenguaje muy cercano al pseudocódigo descrito en el texto *Introduction to Algorithms* de Thomas Cormen et al. [1]. Permitiendo en el lenguaje el uso de la librería estándar de *Java*, de cualquier librería externa y de cualquier clase que implemente el usuario, *GOLD* puede utilizarse para simplificar enormemente el desarrollo de aplicaciones que usen intensivamente estructuras de datos basadas en colecciones de datos. Al combinarse con un lenguaje de propósito general como *Java*, la expresividad del lenguaje puede llegar a límites insospechados, facilitando actividades como el desarrollo de interfaces gráficas y la manipulación de archivos.

1.3. Motivación

Los grafos son estructuras de datos compuestas por un conjunto de vértices V y por un conjunto de arcos $E \subseteq V \times V$ que conectan vértices entre sí. Esta definición, aunque parezca simplista, define un objeto matemático increíblemente versátil que permite modelar relaciones, redes y jerarquías, ayudando a resolver problemas en una infinidad de disciplinas, en particular aquellas relacionadas con el mundo de la ingeniería.

La teoría de grafos se encarga de estudiar las propiedades de los grafos y la algorítmica subyacente para solucionar problemas típicos que ocurren a menudo en las ciencias de la computación y en la ingeniería industrial, como

problemas de redes de flujo, problemas de transporte, problemas de conectividad y problemas de costo mínimo, sólo por mencionar algunos ejemplos. Para manipular grafos y otras estructuras de datos avanzadas es suficiente contar con un lenguaje de propósito general (como *Java* o *C++*) que suministre a los desarrolladores librerías que implementen las estructuras de datos e instrucciones de control para manejarlas. Sin embargo, el principal inconveniente de usar un lenguaje de propósito general es que la algorítmica puede dificultarse a tal grado que, para implementar ciertos algoritmos clásicos de los libros de texto (escritos breve y claramente en pseudocódigo), es necesario dominar el lenguaje y usar instrucciones complicadas que aumentan el tiempo de desarrollo y ofuscan el código fuente escrito. Por ejemplo el algoritmo de Kruskal [1] para resolver el problema del árbol de expansión mínimo es estudiado en la literatura a través de un pseudocódigo como el mostrado a continuación:

Código 1.1. *Pseudocódigo del algoritmo de Kruskal [1].*

```
1  MST-KRUSKAL(G,w)
2    A←∅
3    for each vertex v∈V[G] do
4      MAKE-SET(v)
5    sort the edges of E into nondecreasing order by weight w
6    for each edge (u,v)∈E, taken in nondecreasing order by weight do
7      if FIND-SET(u)≠FIND-SET(v) then
8        A←A∪{(u,v)}
9        UNION(u,v)
10   return A
```

El propósito de cualquier pseudocódigo debería ser el de suministrar una descripción compacta, clara y entendible de cómo opera un determinado algoritmo, pensando en que va a ser leído por un ser humano y no por una máquina. Por esta razón deben ser diseñados en beneficio de la legibilidad, permitiendo el uso de frases en lenguaje natural, de estructuras de datos básicas y de símbolos matemáticos estándar. En particular, el algoritmo 1.1 usa una estructura especializada para administrar conjuntos disyuntos mediante las operaciones `MAKE-SET`, `FIND-SET` y `UNION` [1], usa ciclos *for-each* para iterar colecciones, usa operadores matemáticos como la pertenencia ($\in$) y la unión de conjuntos ($\cup$), usa símbolos matemáticos para denotar constantes como el conjunto vacío ($\varnothing$), y hace referencia a una instrucción informal (*sort the edges of E ...*) para describir una operación de alto nivel que no se puede expresar con instrucciones del lenguaje.

Código 1.2. *Algoritmo de Kruskal implementado en Java.*

```
1  public class Edge {
2    public int source;
3    public int target;
4    public double cost;
5  }
6  public class DisjointSet { // Véase Introduction to Algorithms (Cormen et al.)
7    private DisjointSet parent=this;
8    private int rank=0;
9    public static void UNION(DisjointSet x, DisjointSet y) {
10     x=FIND_SET(x); y=FIND_SET(y);
11     if (x.rank>y.rank) y.parent=x;
12     else if (x.rank<y.rank) x.parent=y;
13     else if (x!=y) {y.parent=x; x.rank++;}
14   }
15   public static DisjointSet FIND_SET(DisjointSet x) {
16     return x.parent==x?x:(x.parent=FIND_SET(x.parent));
17   }
18 }
19 import java.util.*;
20 public class Kruskal { // Véase Introduction to Algorithms (Cormen et al.)
```

```
21   public static List<Edge> MST_KRUSKAL(List<Edge>[] adjacencyList) {
22     List<Edge> edges=new ArrayList<Edge>();
23     for (List<Edge> list:adjacencyList) edges.addAll(list);
24     Collections.sort(edges,new Comparator<Edge>() {
25       public int compare(Edge edge1, Edge edge2) {
26         int c1=((Double)(edge1.cost)).compareTo((Double)(edge2.cost));
27         int c2=edge1.source-edge2.source;
28         int c3=edge1.target-edge2.target;
29         return c1!=0?c1:(c2!=0?c2:c3);
30       }
31     });
32     DisjointSet forest[]=new DisjointSet[adjacencyList.length];
33     for (int i=0; i<forest.length; i++) forest[i]=new DisjointSet();
34     List<Edge> A=new ArrayList<Edge>();
35     for (Edge edge:edges) {
36       int u=edge.source,v=edge.target;
37       if (DisjointSet.FIND_SET(forest[u])!=DisjointSet.FIND_SET(forest[v])) {
38         A.add(edge);
39         DisjointSet.UNION(forest[u],forest[v]);
40       }
41     }
42     return A;
43   }
44 }
```

Si únicamente contamos con la librería estándar de *Java* o el *STL* (*Standard Template Library*) de *C++*, la labor de implementar el algoritmo de Kruskal no sería fácil ni inmediata. Sólo para comenzar, deberíamos implementar una clase para representar los arcos, codificar una estructura de datos para administrar conjuntos disyuntos [1] y transformar los símbolos matemáticos en llamados a procedimiento. Además, si realizamos la traducción sin tener cuidado, es posible que afectemos dramáticamente la esencia del algoritmo. Específicamente, podríamos degradar la complejidad temporal del proceso si seleccionamos una estructura de datos inadecuada para almacenar los arcos, si usamos un algoritmo ineficiente para ordenarlos de menor a mayor según costo o si administramos los conjuntos disyuntos con una estructura de datos ineficiente. Por esta razón, usar un lenguaje de propósito general en el desarrollo de un problema difícil de grafos podría terminar siendo un trabajo largo y tedioso, dado que en su mayoría no están diseñados para tal fin.

1.4. Justificación

Usar un lenguaje de propósito general para implementar algoritmos sobre conjuntos, grafos y otras estructuras de datos puede llegar a ser complicado. Por fortuna existen muchas librerías y lenguajes que se especializan en la manipulación y visualización de grafos, dando una luz de esperanza a quienes desean modelar y resolver computacionalmente problemas en términos de grafos, pues dotan al programador de herramientas especialmente diseñadas para tal fin. Sin embargo, las librerías disponibles en la actualidad suelen ser un conjunto de clases que enriquecen el *API* estándar de *Java* o el *STL* (*Standard Template Library*) de *C++* con implementaciones típicas de determinadas estructuras de datos avanzadas que buscan facilitar la algorítmica sobre los grafos, y los lenguajes de propósito específico que se consiguen hoy en día suelen ser complicados de aprender, de usar y de integrar con otras herramientas en grandes proyectos de software, y adicionalmente suelen no tener la misma expresividad que la de un lenguaje de alto nivel como *Java* o *C++*.

Para disminuir el esfuerzo en la programación de algoritmos sobre grafos y otras estructuras de datos avanzadas es necesario contar con un lenguaje de propósito específico que se preocupe por mejorar la legibilidad de los programas y por acelerar el proceso de desarrollo. Este lenguaje debe mezclar las virtudes del pseudocódigo con las de un

lenguaje de alto nivel como *Java* o *C++* para que pueda ser fácilmente entendido por un matemático, por un científico o por un ingeniero, ser fácilmente interpretado por una máquina, y aprovechar la expresividad de algún lenguaje de propósito general.

Cada vez que en este documento se mencione el término *pseudocódigo*, se estará haciendo alusión a los programas que se pueden escribir con el lenguaje trabajado en el libro *Introduction to Algorithms* de Thomas Cormen et al. [1], que fue diseñado para ser compacto, claro y entendible. La última versión del lenguaje, denominada *GOLD* versión 3 (*Graph Oriented Language Domain* por sus siglas en inglés), es un lenguaje de propósito específico imperativo lo bastante cercano al lenguaje utilizado en la referencia [1] para ser considerado una especie de pseudocódigo y lo bastante cercano al lenguaje *Java* para poder utilizar la potencia de su librería estándar, de su compilador, de su máquina virtual, y de *Eclipse*, uno de sus entornos de programación más reconocidos. Dado que el lenguaje está diseñado para que desde los programas escritos en *GOLD* se pueda usar cualquier clase de *Java* y viceversa, sería factible tener un proyecto de software que mezcle clases implementadas en *Java* con procesos implementados en *GOLD*, aunque también el proyecto podría estar completamente implementado en *Java* o completamente implementado en *GOLD*.

Pese a que *GOLD 3* está fundamentado en los lenguajes *GOLD* de Luis Miguel Pérez [3] y *GOLD+* de Diana Mabel Díaz [4], que fueron desarrollados en la Universidad de los Andes, el nuevo lenguaje comparte pocos aspectos de diseño e implementación con sus dos antecesores, tanto en sintaxis como en semántica y en tecnología usada; sin embargo, el objetivo general sigue siendo el mismo: diseñar un lenguaje de propósito específico que facilite a los programadores la labor de describir y manipular grafos. Las principales diferencias de *GOLD 3* con respecto a sus dos antecesores es que la nueva sintaxis evoca a los pseudocódigos que se estudian en libros como el de Cormen et al. [1], es orientado a objetos, está completamente integrado al ambiente de desarrollo *Eclipse*, se puede usar transparentemente cualquier clase implementada en *Java* (ya sea de la librería estándar, de alguna librería externa, o implementada por el usuario), cuenta con una amplia variedad de estructuras de datos (listas, pilas, colas, conjuntos, bolsas, asociaciones llave-valor, árboles, grafos y autómatas, entre otras), y suministra una diversa gama de implementaciones típicas para cada una de las estructuras de datos provistas.

El principal hecho que justifica este proyecto de tesis es que en la actualidad no existe un producto como el que se plantea, convirtiendo a *GOLD* en un lenguaje potente y novedoso que presenta la siguiente ambivalencia:

- actúa como un lenguaje de propósito específico que permite la escritura de algoritmos para la manipulación de grafos y otras estructuras de datos con una sintaxis amigable al programador, muy similar al pseudocódigo trabajado en el texto *Introduction to Algorithms* [1] de Thomas Cormen et al.; y
- actúa como un lenguaje de propósito general que permite la invocación de métodos y la utilización de clases de la librería estándar de *Java*, de cualquier librería externa y de cualquier clase implementada por el usuario.

1.5. Descripción del problema

Estudiantes, profesionales e investigadores de múltiples carreras relacionadas con ramas de la ciencia como las ingenierías (particularmente la ingeniería industrial y la ingeniería de sistemas), las matemáticas y las ciencias de la computación, tienen la necesidad de plantear enunciados, de modelar situaciones y de resolver problemas que requieren mencionar estructuras de datos especializadas e implementar algoritmos que las manipulen. Por ejemplo, muchos problemas pueden ser definidos en términos de grafos (como el problema de la ruta más corta, el problema del árbol de expansión mínimo, el problema del agente viajero y otros problemas de optimización) y muchas soluciones son casos particulares de algún algoritmo clásico de teoría de grafos (como el algoritmo de Dijkstra o el algoritmo de Kruskal).

Es necesario que exista un lenguaje de propósito específico para manipular grafos y otras estructuras de datos de una manera cómoda, a través de código fuente que sea compacto, claro y legible. Sin embargo, en la actualidad

no existen herramientas que permitan programar algoritmos clásicos sobre grafos como el algoritmo de Kruskal usando un lenguaje cercano al ser humano como el utilizado en los pseudocódigos de Cormen et al. [1], que actúe simultáneamente como un lenguaje de propósito específico que facilite la escritura de pseudoalgoritmos sobre estructuras de datos especializadas, y como un lenguaje de propósito general que rescate la gran expresividad que tienen lenguajes de alto nivel reconocidos como *Java* y *C++*.

El problema de investigación se puede plantear a través de la siguiente pregunta: ¿es posible diseñar un lenguaje de programación de propósito específico en el que se puedan implementar algoritmos que manipulen grafos y otras estructuras de datos con una sintaxis cercana al pseudocódigo que permita aprovechar la expresividad de un lenguaje de propósito general como *Java* o *C++*?

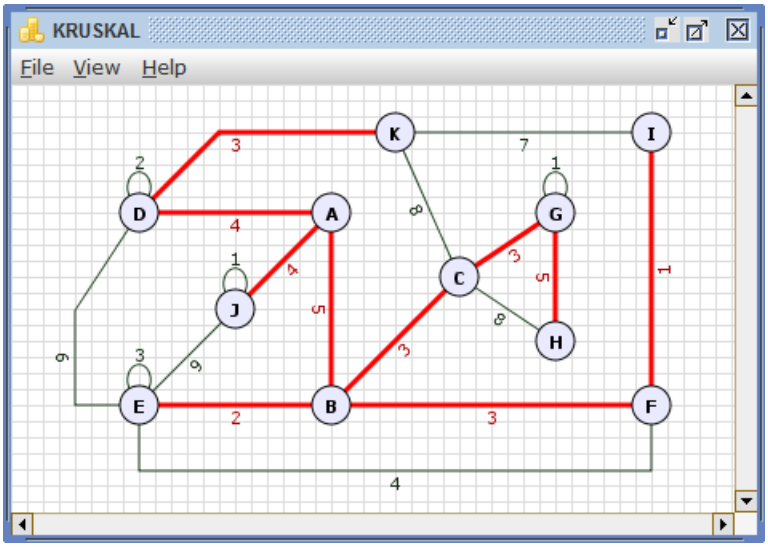
Código 1.3. *Algoritmo de Kruskal implementado en GOLD 3.*

```
1  package kernel
2  import gold.**
3  function kruskal(G:IGraph) begin // Kruskal's algorithm
4    V,E,F:=G.getVertices(),G.getEdges(),GForestDisjointSets()
5    for each v∈V do
6      F.makeSet(v)
7    end
8    A:=∅
9    E:=GCollections.sort(GArrayList(E))
10   for each ⟨u,v⟩∈E do
11     if F.findSet(u)≠F.findSet(v) then
12       A:=A∪{⟨u,v⟩}
13       F.union(u,v)
14     end
15   end
16   return A
17 end
```

Código 1.4. *Aplicación del algoritmo de Kruskal sobre un grafo aleatorio en GOLD 3.*

```
1  @SuppressWarnings("types")
2  package gui
3  import java.awt.*
4  import gold.structures.graph.*
5  import gold.visualization.graph.*
6  import kernel.Kruskal
7  var random:java.util.Random(0)
8  procedure main(args:String[]) begin // Kruskal test
9    var G:GUndirectedGraph('A'..'K')
10   for each a in 'A'..'K' do
11     for each b in a..'K' do
12       if random.nextDouble()<0.3 then
13         G.addEdge(a,b,random.nextInt(10))
14       end
15     end
16   end
17   A:=Kruskal.kruskal(G)
18   B:=A∪{⟨y,x⟩|⟨x,y⟩∈A}
19   print "KRUSKAL:\n  Edges:",A,", Cost:",(Σx,y|⟨x,y⟩∈A:G.getCost(x,y))
20   frame:=GGraphFrame.show(G)
21   frame.getPainter().getEdgeDrawPaintTransformer().setAll(B,Color.RED)
22   frame.getPainter().getEdgeStrokeTransformer().setAll(B,new BasicStroke(3.0f))
23   frame.setTitle("KRUSKAL")
24 end
```

Figura 1.1. *Ventana gráfica desplegada después de ejecutar el programa 1.4.*



1.6. Objetivos

El objetivo general del proyecto *GOLD* consiste en diseñar un lenguaje de propósito específico similar al usado en los pseudocódigos del texto *Introduction to Algorithms* [1], junto con un entorno de desarrollo integrado (*IDE*: *integrated development environment*) que facilite a los desarrolladores la labor de describir y manipular grafos.

El lenguaje debe permitir la programación de algoritmos sobre grafos y otras estructuras de datos avanzadas, aprovechando toda la potencia de un lenguaje de propósito general orientado a objetos como *Java* para rescatar la gran expresividad que ya se tiene ganada con su aplicación, y aprovechando el estilo de escritura de los pseudocódigos para fomentar la programación de código fuente compacto, claro y entendible. De esta manera, los matemáticos, ingenieros e investigadores contarían con una herramienta computacional fácil de usar, con la que podrían programar algoritmos sobre una variada colección de estructuras de datos (incluyendo los grafos) para resolver una gran cantidad de problemas. La versión *GOLD 3* tiene los siguientes objetivos específicos:

- Facilitar la codificación de algoritmos sobre estructuras de datos como:
  - tuplas (*tuples*) y listas (*lists*);
  - pilas (*stacks*), colas (*queues*) y bicolas (*deques*);
  - conjuntos (*sets*) y bolsas (*bags*);
  - montones (*heaps*);
  - asociaciones llave-valor (*maps*);
  - árboles (árboles binarios, árboles ordenados, árboles enearios, *Tries*, *Quadtrees*, etc);
  - grafos (grafos dirigidos, grafos no dirigidos, hipergrafos); y
  - autómatas (autómatas finitos determinísticos, autómatas finitos no determinísticos).
- Dotar al programador de un lenguaje cercano al pseudocódigo donde pueda operar objetos a través de la notación matemática estándar estudiada en dominios de interés como:
  - lógica proposicional;
  - lógica de predicados;
  - teoría de secuencias;
  - teoría de conjuntos;

- teoría de enteros;
- teoría de grafos;
- teoría de lenguajes; y
- matemáticas discretas.

1.7. ¿Cómo leer este documento?

A grandes rasgos, este documento presenta los antecedentes, requerimientos, desarrollo y resultados de la versión 3 del proyecto *GOLD*, visto como un lenguaje de programación imperativo para la manipulación de grafos y otras estructuras de datos. Para facilitar la lectura de los temas, el contenido del documento se organizó en cuatro partes, que se dividen en diez capítulos y un anexo:

Parte I. *Preliminares*: cubre los prerrequisitos utilizados como insumo durante el desarrollo del proyecto.

Capítulo 1. *Introducción*: describe el propósito del proyecto, justifica su nacimiento y formula su objetivo general.

Capítulo 2. *Antecedentes*: describe brevemente la historia de los proyectos de la Universidad de los Andes que precedieron al presente: *CSet* [2], *GOLD 1* [3] y *GOLD 2* [4] (*GOLD*+).

Capítulo 3. *Estado del arte*: enumera varios lenguajes, librerías y aplicativos de escritorio que existen en la actualidad para describir, manipular e implementar algoritmos sobre grafos.

Capítulo 4. *Marco teórico*: estudia las teorías necesarias para fundamentar el trabajo desarrollado y para abordar con propiedad los temas tratados en este documento, incluyendo tópicos generales sobre lenguajes de propósito general, lenguajes de propósito específico y estructuras de datos.

Parte II. *Propuesta de solución*: enuncia los requerimientos y trata todos los asuntos relacionados con el desarrollo de la solución al problema, relatando los aspectos técnicos y los lineamientos que guiaron la implementación del lenguaje y de su entorno de programación asociado.

Capítulo 5. *Requerimientos*: enuncia los requerimientos funcionales y no funcionales que guiaron el desarrollo del proyecto.

Capítulo 6. *Herramientas*: realiza un inventario de todas las tecnologías y librerías que se usaron durante la implementación del producto de software.

Capítulo 7. *Diseño*: expone todas las decisiones de diseño que influenciaron la etapa de desarrollo.

Capítulo 8. *Implementación*: describe los aspectos más relevantes relacionados con el desarrollo del producto que satisface los requerimientos propuestos.

Parte III. *Resultados*: analiza los resultados obtenidos, resaltando la potencia y expresividad del lenguaje.

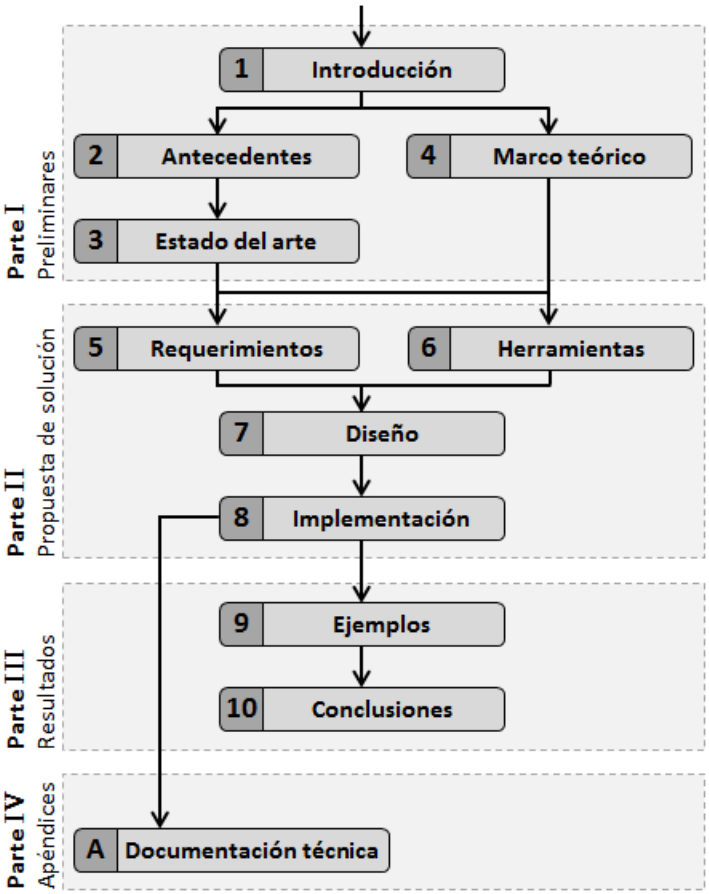
Capítulo 9. *Ejemplos*: presenta algunos programas de ejemplo codificados en el lenguaje propuesto, haciendo énfasis en el uso de las distintas instrucciones de control y de la librería suministrada.

Capítulo 10. *Conclusiones*: presenta las conclusiones del proyecto describiendo el trabajo realizado y declarando como trabajo futuro los aspectos que quedaron por mejorar, los requerimientos deseables que no se incluyeron y las funcionalidades que podrían implementarse en las siguientes versiones del proyecto buscando la evolución del lenguaje y de su entorno de programación.

Parte IV. *Apéndices*: anexo que incluye la documentación técnica de la herramienta.

Apéndice A. *Documentación técnica*: contiene documentación técnica del producto como las producciones de la gramática en notación *BNF* extendida [5], la definición del lenguaje en notación *Xtext* [6], algunos diagramas de diseño en notación *UML* (*Unified Modeling Language*), y algunas instrucciones para generar e instalar el *plug-in* en *Eclipse* [7].

Figura 1.2. *Estructura del documento de tesis exhibiendo las relaciones de dependencia entre sus capítulos.*



A continuación se enumeran cinco niveles de detalle recomendados para la lectura del documento, pasando gradualmente desde una lectura superficial hasta una lectura completa y profunda:

1. Para conocer las generalidades del proyecto y tener una noción del trabajo realizado, lea completamente los capítulos §1 (*Introducción*) y §10 (*Conclusiones*), y el resumen de cada uno de los capítulos que se encuentra justo al principio de éstos.

2. El nivel 1 más la lectura completa de los capítulos §2 (*Antecedentes*) y §3 (*Estado del arte*) para conocer el contexto y el estado del arte que enmarcan el desarrollo del proyecto.

3. El nivel 2 más la lectura completa de los capítulos §4 (*Marco teórico*) y §5 (*Requerimientos*) para conocer los fundamentos teóricos y los requerimientos básicos que guiaron el desarrollo del proyecto.

4. El nivel 3 más la lectura completa de los capítulos §6 (*Herramientas*), §7 (*Diseño*) y §8 (*Implementación*) para conocer los detalles particulares al diseño e implementación del producto.

5. El nivel 4 más la lectura completa del capítulo §9 (*Ejemplos*) y del anexo A (*Documentación técnica*) para estudiar algunos ejemplos de utilización del lenguaje y conocer los pormenores técnicos de la herramienta.

En todo caso, se recomienda una lectura secuencial del documento en el orden en que aparecen los capítulos.

Capítulo 2

Antecedentes

Este capítulo describe brevemente las características de los proyectos de la Universidad de los Andes que precedieron al presente trabajo de tesis: *CSet: un lenguaje para composición de conjuntos* [2] de Víctor Hugo Cárdenas (2008), *GOLD: un lenguaje orientado a grafos y conjuntos* [3] de Luis Miguel Pérez (2009), y *GOLD+: lenguaje de programación para la manipulación de grafos: extensión de un lenguaje descriptivo a un lenguaje de programación* [4] de Diana Mabel Díaz (2010).

2.1. *CSet*

La tesis de maestría *CSet: un lenguaje para composición de conjuntos* [2] fue desarrollada por Víctor Hugo Cárdenas en el año 2008 bajo la asesoría de Silvia Takahashi del grupo de investigación de Tecnologías de la Información y Construcción de Software (*TICSw*) del Departamento de Ingeniería de Sistemas y Computación de la Universidad de los Andes. *CSet* es un lenguaje de propósito específico que permite componer conjuntos de datos, proveyendo una herramienta flexible para describir conjuntos en general y realizar operaciones sobre éstos, basándose en un estilo de programación declarativo [2]. El proyecto *CSet* se convirtió en el punto de partida de otros trabajos dentro del grupo de investigación, que requerían el uso de un lenguaje especializado en la definición de conjuntos de datos, concretamente aquellos relacionados con el modelado de problemas de optimización en investigación de operaciones y con el desarrollo de lenguajes para la descripción y simulación en grafos [2].

A pesar de que existe una amplia notación usada en matemáticas para describir y para operar conjuntos, la sintaxis de *CSet* se ‘‘orientó a utilizarla en la medida de lo posible, considerando las limitaciones de un teclado de computador en la escritura de algunos símbolos matemáticos’’ [2]. Esta decisión obligó a que el lenguaje *CSet* estuviese definido sobre un alfabeto que únicamente utilizaba los caracteres pertenecientes a las categorías Latín Básico (*Basic Latin*: `0x0020-0x007E`) y Suplemento Latin-1 (*Latin-1 Supplement*: `0x00A0-0x00FF`) del estándar de codificación de caracteres *Unicode*, que reúnen la mayoría de los símbolos que comúnmente se encuentran en los teclados de computador occidentales. Al no usar la gama completa de caracteres proporcionada por el estándar *Unicode*, muchos símbolos matemáticos tuvieron que ser reemplazados por códigos foráneos para el usuario (e.g., `notin` en lugar de $\notin$, `+` en lugar de $\cup$ y `*` en lugar de $\cap$).

CSet fue implementado usando el *framework* suministrado por el proyecto *openArchitectureWare* (*oAW*), que en el año 2009 pasó a formar parte del *Eclipse Modeling Project* (*EMF*), cuyo objetivo es ‘‘la evolución y promoción de tecnologías de desarrollo basado en modelos dentro de la comunidad *Eclipse*, ofreciendo un conjunto unificado de *frameworks* de modelado, herramientas e implementaciones estándar’’ [8]. El proyecto *openArchitectureWare* administró las primeras versiones de *Xtext* [6] (véase la sección §6.1.3) y todas sus tecnologías asociadas para facilitar la implementación de lenguajes de propósito específico con una arquitectura basada en modelos. *CSet* es distribuido a través de un *plug-in* para el ambiente de programación *Eclipse* incluyendo funcionalidades como el

resaltado de la sintaxis, el auto-completado de código y la validación de la sintaxis [2], implementadas con varias herramientas que formaban parte integral de *openArchitectureWare* como [2]:

- *Xtext* para generar automáticamente el metamodelo a partir de la gramática del lenguaje en notación *BNF* extendida [5].
- *Xtend* para definir las operaciones responsables de manipular el metamodelo generado por *Xtext*.
- *Check* para definir las validaciones relacionadas con la semántica del lenguaje.
- *Xpand* para transformar los programas codificados en *CSet* a código *Java* que posteriormente podía ser ejecutado en *Eclipse*.

Lo anterior convierte a *CSet* en un lenguaje compilado donde los programas son sometidos a un proceso de traducción que los transforma en archivos codificados en *Java*. Su gramática, definida mediante *Xtext*, incluye los siguientes elementos [2]:

- *Conjuntos base*. Representan conjuntos de números enteros o de cadenas de texto, descritos por enumeración (listando explícitamente cada uno de sus elementos).
- *Conjuntos derivados*. Representan conjuntos de tuplas, descritos por comprensión a través de condiciones que especifican el rango de cada una de las variables ligadas.
- *Operaciones simples*. Permiten la aplicación de las siguientes operaciones entre conjuntos: pertenencia (`in`), anti-pertenencia (`notin`), unión (`+`), intersección (`*`) y diferencia (`-`).
- *Operaciones de comparación*. Permiten la aplicación de las siguientes operaciones de comparación entre números enteros: menor que (`<`), menor o igual que (`<=`), mayor que (`>`), mayor o igual que (`>=`), igual a (`=`), distinto de (`<>`).
- *Funciones numéricas*. Permiten la aplicación de funciones estadísticas básicas para calcular el máximo (`max`), el mínimo (`min`), la suma (`sum`) y el promedio (`avg`) de los elementos de un conjunto de números enteros †[1].
- *Funciones de salida*. Permiten la exportación en archivos *CSV* (*comma-separated values*) y la impresión en consola de los elementos de un conjunto. Los conjuntos exportados a archivos con formato *CSV* se pueden importar con una instrucción especial provista para los conjuntos base.

Código 2.1. *Programa de ejemplo escrito en CSet [2].*

```
1  set A = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 }
2  set B = { "Bogota", "Santa Marta", "Cali" }
3  set C = { 2,3,4,5 }
4  set D from "data/productos.csv" get("ProductID", "ProductName")
5  set SIMP = { (i,j,k) : i in B, j in D, k in A }
6  set CONJ = { (i,j): i in A, j in B, i < avg(A), j <> "Cali" }
7  set UNION = { (i) : i in B + D }
8  set INTER = { (i) : i in A * C }
9  set DIFF1 = { (i) : i in A - C }
10 set CRUZ = { (i,j) : i in A, j in B }
11 set F = { (f) : f <= 3, f in C }
12 set G = { (i) : i in A, i >= 5, i < 11 }
13 set K = { (k) : k in A, k < sum(C) }
14
```

[1] En el documento de tesis de *CSet* [2] estas funciones son catalogadas erróneamente dentro de las operaciones de comparación.

```
15 print D
16 print SIMP
17 print CONJ
18 print F
19 print G
20 print H
21 print K
22 print UNION
23 save B format=CSV "data/outB.csv" headers ( "Columna1" )
24 save CRUZ2 format=CSV "data/cruz.csv"
```

2.2. *GOLD 1*

Teniendo como precedente el desarrollo de *CSet* [2], el proyecto *GOLD* (*Graph Oriented Language Domain* por sus siglas en inglés) surgió en el año 2009 como un lenguaje de propósito específico diseñado para expresar matemáticamente grafos en términos de conjuntos, implementado por Luis Miguel Pérez en la tesis de pregrado *GOLD: un lenguaje orientado a grafos y conjuntos* [3] bajo la asesoría de Silvia Takahashi del Departamento de Ingeniería de Sistemas y Computación de la Universidad de los Andes y de Andrés Medaglia del Departamento de Ingeniería Industrial de la misma universidad, en la línea de investigación de Modelado de Propósito Específico (*Domain Specific Modeling*) perteneciente al área de Métodos Formales (*MF*) del grupo de investigación de Tecnologías de la Información y Construcción de Software (*TICSw*) del Departamento de Ingeniería de Sistemas y Computación (*DISC*) de la Universidad de los Andes. De esta forma, *GOLD* se constituyó como uno de los primeros proyectos en incorporar los frutos obtenidos en *CSet*, aprovechando la expresividad ofrecida por este lenguaje para definir los conjuntos de vértices y de arcos correspondientes a los grafos.

La versión de *GOLD* implementada por Luis Miguel Pérez, identificada en este documento como *GOLD 1*, es un lenguaje de propósito específico para definir grafos en términos matemáticos a través de la descripción formal de su conjunto de vértices y de su conjunto de arcos [3], tanto por comprensión (describiendo las propiedades que cumplen sus elementos) como por extensión (enumerando explícitamente sus elementos). *GOLD 1* pretendía atacar las falencias presentes en muchas de las herramientas disponibles en la época para la manipulación de grafos, que usualmente no proveían un mecanismo para definir los vértices y los arcos de un grafo por comprensión (describiendo matemáticamente las propiedades que cumplen sus elementos), no dejando otro camino que hacerlo manualmente por extensión (enumerando sus elementos), lo que involucra un trabajo tedioso y repetitivo de adición de vértices y de arcos que puede resultar inviable si el grafo posee cientos de nodos.

El analizador léxico y sintáctico de *GOLD 1* fue desarrollado en *JavaCC* [9] (*Java Compiler Compiler*), que es una herramienta capaz de generar automáticamente clases codificadas en *Java* que implementan el compilador de un lenguaje de propósito específico a partir de su sintaxis en notación *BNF* extendida [5]. A diferencia de *CSet*, el lenguaje *GOLD 1* es interpretado y no compilado puesto que todas sus instrucciones son interpretadas por una máquina especializada que está construida en *Java*.

A grandes rasgos, *GOLD 1* tiene las siguientes características:

- ofrece diversos tipos de datos, incluyendo los enteros (`int`), los reales (`real`), los booleanos (`boolean`), las cadenas de caracteres (`String`), las tuplas (sin identificador asociado para su tipo de datos), los conjuntos (`Set`), y los grafos (`Graph`);
- permite la definición de conjuntos por comprensión (describiendo las propiedades que cumplen sus elementos) y por extensión (enumerando explícitamente sus elementos), con la restricción de que todos los elementos del conjunto deben ser del mismo tipo;
- provee las siguientes operaciones sobre números: suma (`+`), resta (`-`), multiplicación (`*`), división (`/`);

- provee las siguientes operaciones sobre valores booleanos: conjunción (`&&`), disyunción (`||`), implicación (`->`), consecuencia (`<-`), equivalencia (`==`), negación (`!`);
- permite las siguientes operaciones de comparación sobre números: menor que (`<`), menor o igual que (`<=`), mayor que (`>`), mayor o igual que (`>=`), igual a (`=`), distinto de (`!=`);
- provee las siguientes operaciones sobre conjuntos: unión (`U`), intersección (`&`), diferencia (`\`), cardinalidad (`#`), determinar si un conjunto es vacío (`isEmpty`);
- provee las siguientes operaciones sobre grafos: obtener los predecesores de un nodo (`getIn`), obtener los sucesores de un nodo (`getOut`);
- permite la declaración de variables y de funciones aritméticas que actúan como macros;
- permite la definición de *atributos* [3] para hacer posible la asignación de valores numéricos a los elementos de un conjunto, simulando el comportamiento de una asociación llave-valor (*map*);
- permite la exportación de grafos en formato *XGMML* [10]; y
- permite la visualización de grafos a través de *GIDEN* [11].

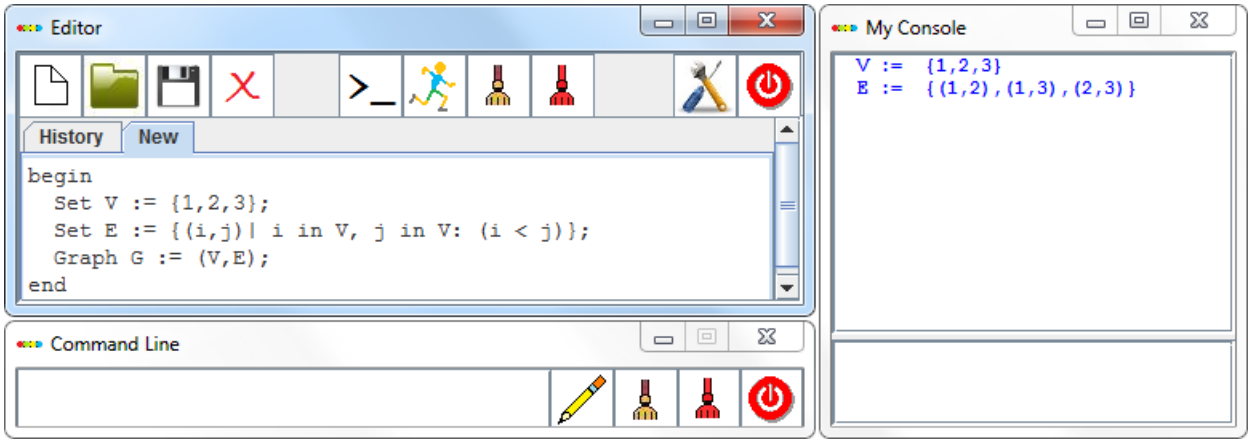
Para describir conjuntos por comprensión en *GOLD 1* se debe suministrar una lista de variables ligadas (*dummies*), un cuerpo que define la forma de sus miembros y un rango que provee las condiciones que cumplen sus elementos, mediante una notación basada en la del proyecto *CSet* [2] y en la del texto *A Logical Approach to Discrete Math* [12] de David Gries y Fred Schneider.

Código 2.2. *Programa de ejemplo escrito en GOLD 1 [3].*

```
1 begin
2   Set N := {i | 0 < i <= 8};
3   Set A := {(i,j)| i in N, j in N: (i != j)};
4   Graph G := (N,A);
5   Set X := getIn(2);
6   Set Y := getOut(2);
7 end
```

Por último, *GOLD 1* ofrece un ambiente de desarrollo para la escritura de programas que está compuesto por un editor de texto, una ventana de línea de comandos y una consola de mensajes.

Figura 2.1. *IDE de GOLD 1 [3].*



2.3. *GOLD 2* (*GOLD+*)

Aunque la tesis de Luis Miguel Pérez ofrecía un lenguaje básico para definir grafos a través de la descripción matemática de su conjunto de nodos y de su conjunto de arcos, hacía falta un lenguaje de programación para manipularlos. En el año 2010 Diana Mabel Díaz, con su tesis de maestría *GOLD+: lenguaje de programación para la manipulación de grafos: extensión de un lenguaje descriptivo a un lenguaje de programación* [4] dirigida por Silvia Takahashi e identificada en este documento como *GOLD 2*, extendió *GOLD 1* para permitir la manipulación algorítmica de grafos con un lenguaje de programación sencillo basado en un conjunto limitado de instrucciones de control. A pesar de que *GOLD 2* permite describir grafos y realizar determinadas operaciones básicas entre éstos, no es un lenguaje diseñado para la manipulación exhaustiva de grafos en grandes proyectos de software. Reutilizando el entorno de programación de *GOLD 1*, el desarrollo de *GOLD 2* se limitó a:

- analizar superficialmente el estado del arte en relación a los productos existentes para definir y manipular grafos;
- añadir el tipo de datos `list` para representar listas y la constante `infi` para representar el infinito positivo;
- extender la sintaxis y la semántica del lenguaje descriptivo *GOLD 1* para transformarlo en un lenguaje de programación incipiente; y
- adaptar la máquina desarrollada en *GOLD 1* para permitir la interpretación de programas escritos siguiendo la nueva sintaxis, de acuerdo con la semántica establecida.

La gramática del lenguaje *GOLD 1* fue enriquecida con las siguientes instrucciones imperativas, donde las palabras reservadas no son sensibles a las mayúsculas (*case insensitive*):

- `print(`<*text*>`);`
  imprime en la consola de mensajes la cadena de texto <*text*>.
- `AddAttr` <*attribute*> `in Nodes` <*graph*>`;`
  asigna el atributo <*attribute*> a todos los nodos del grafo <*graph*>.
- `AddAttr` <*attribute*> `in Edges` <*graph*>`;`
  asigna el atributo <*attribute*> a todos los arcos del grafo <*graph*>.
- `AddAttr` <*attribute*> `like` <*value*> `forAll` <*graph*> `Nodes;`
  asigna el atributo <*attribute*>, con valores de la forma <*value*>, a todos los nodos del grafo <*graph*>.
- `AddAttr` <*attribute*> `like` <*value*> `forAll` <*graph*> `Edges;`
  asigna el atributo <*attribute*>, con valores de la forma <*value*>, a todos los arcos del grafo <*graph*>.
- `getAttr` <*attribute*> `of` <*graph*> <*node*>`;`
  informa el valor del atributo <*attribute*> del nodo <*node*> perteneciente al grafo <*graph*>.
- `SetNodes` <*name*> `:=` <*graph*> `Nodes;`
  declara una variable con nombre <*name*> y tipo `SetNodes`, que tiene como valor el conjunto de nodos del grafo <*graph*>.
- `delete` <*node*> `of` <*set*>`;`
  elimina el nodo <*node*> del conjunto <*set*>, suponiendo que es de tipo `SetNodes`.

- `SetInitial <number> in <graph>;`
  designa al nodo con identificador <*number*> como el nodo inicial del grafo <*graph*> [†2].

- `Node <name> := node <number> of <graph>;`
  declara una variable con nombre <*name*> y tipo `Node`, que actúa como un apuntador al nodo con identificador <*number*> perteneciente al grafo <*graph*>.

- `SetNodes <name> := nodesOut <node> of <graph>;`
  declara una variable con nombre <*name*> y tipo `SetNodes`, que tiene como valor el conjunto de nodos que son sucesores del nodo <*node*> en el grafo <*graph*>.

- `Node <name> := [<number>, <values>];`
  declara una variable con nombre <*name*> y tipo `Node`, que tiene como valor un nuevo nodo con identificador <*number*> y con atributos <*values*>.

- <*type*> `function <name>(<parameters>) begin <instructions> end`
  declara una función con nombre <*name*>, tipo de retorno <*type*> y parámetros <*parameters*>, cuya implementación está dada a través del subprograma <*instructions*>.

- `foreach <graph> begin <instructions> end`
  ejecuta las instrucciones <*instructions*> para cada uno de los nodos del grafo <*graph*>, a través de una instrucción repetitiva *for-each* cuya variable de iteración es de tipo `Node` y siempre se debe mencionar a través del nombre `item`.

- `if <guard> then <instructions> end`
  ejecuta las instrucciones <*instructions*> si se cumple la condición booleana <*guard*>, a través de una instrucción condicional *if-then* que carece de cláusulas *else* y *elseif*.

De este modo, se puede considerar que *GOLD 2* es un lenguaje de programación de propósito específico para la descripción y la manipulación de grafos que permite la programación de algoritmos sobre dicha estructura de datos a través de una sintaxis complicada, difícil de usar, de aprender y de recordar, que incluye asignaciones, condicionales, ciclos y llamados a procedimiento. Pese a que *GOLD 2* permite describir conjuntos y grafos relativamente complejos, es evidente que los desarrolladores deben tener en cuenta una gran cantidad de consideraciones no triviales para lograr escribir el código fuente de sus algoritmos. Algunos de los aspectos de *GOLD 2* que restringen su expresividad y complican la escritura de programas son los siguientes (como se puede constatar en el algoritmo de ejemplo 2.3):

- No permite el uso de caracteres especiales del estándar *Unicode*, obligando a que algunos operadores matemáticos tengan símbolos inadecuados que no corresponden con la notación matemática habitual. Por ejemplo, los siguientes operadores tienen símbolos que no son naturales para los usuarios que están acostumbrados a usar la notación tradicional como la manejada en textos como *A Logical Approach to Discrete Math* [12]:

Tabla 2.1. *Símbolos foráneos definidos en GOLD 2 para algunos operadores binarios.*

Operador	Símbolo foráneo	Símbolo habitual
Implicación booleana	`->`	$\Rightarrow$
Pertenencia de conjuntos	`in`	$\in$
Intersección de conjuntos	`&`	$\cap$
Unión de conjuntos	`U`	$\cup$

[2] En *GOLD 2* se aduce que esta instrucción se necesita en el lenguaje porque ‘‘muchos problemas de grafos, principalmente los de ruteo, requieren que se asigne un nodo como nodo inicial o nodo de partida’’ [4].

- No permite la definición recursiva de funciones, lo que resulta nocivo para la implementación de determinados algoritmos sobre grafos como el recorrido por profundidad (*DFS*: *Depth First Search*), que se describe normalmente a través de un procedimiento recurrente. Las funciones son tratadas como macros que se interpretan justo en el momento en el que son invocadas, la sintaxis del lenguaje no cuenta con una instrucción específica para retornar el valor calculado por una función y la máquina no permite el paso de parámetros por referencia, exigiendo que todos los argumentos de las funciones sean pasados por valor.

- No permite la escritura de ciclos *while*, *do-while*, *repeat-until* ni *for* porque la única instrucción repetitiva proporcionada por el lenguaje es el *for-each*, impidiendo la escritura de instrucciones repetitivas que utilicen contadores numéricos. Incluso, la sintaxis prohíbe la escritura de bucles *while* que ejecutan un conjunto de instrucciones mientras se cumpla cierta condición booleana. Tampoco se permite la escritura de ciclos *for-each* que iteran sobre cualquier colección de datos, puesto que el lenguaje obliga a que la estructura recorrida siempre sea el conjunto de nodos de un grafo.

- No facilita la escritura de ciclos anidados. Para hacer mención al elemento iterado en cualquier ciclo *for-each* se debe usar la palabra reservada `item`, lo que entorpece enormemente la programación de instrucciones repetitivas anidadas porque se tendría que copiar la variable iterada (`item`) en una variable auxiliar para poderla referir dentro de un ciclo interno. En todo caso, no sería muy útil anidar ciclos en el lenguaje dado que la única instrucción repetitiva provista (*for-each*) sólo permite recorrer los nodos de un grafo.

- No permite la escritura de condicionales *if-then-else* ni permite cláusulas *elseif*. Tampoco cuenta con expresiones condicionales (*B?E:F*) ni con instrucciones *switch*. La única instrucción de selección que suministra el lenguaje es el condicional *if-then* que ejecuta una secuencia de instrucciones si se cumple una determinada guarda booleana.

- No permite declarar variables sin tipo, puesto que el lenguaje es fuertemente tipado. Esto hace que la sintaxis sea lejana a los pseudocódigos donde se acostumbra no declarar explícitamente el tipo de todas las variables que se utilizan. Además, no se ofrece ningún mecanismo para mencionar atributos o para invocar métodos sobre los objetos construidos, dependiendo de su tipo.

- No suministra instrucciones fáciles de recordar para la manipulación de objetos en los algoritmos implementados, sumado al hecho de que los símbolos de los operadores no respetan la notación matemática estándar. Por ejemplo, la instrucción

  ```
  AddAttr complete like 0 forAll G Nodes;
  ```

  no tiene un propósito claro y es difícil de recordar.

- No permite el uso de estructuras de datos avanzadas adicionales a las que proporciona. Las únicas estructuras de datos ofrecidas por *GOLD 2* son las listas, las bolsas, los conjuntos y los grafos (por supuesto), lo que afecta enormemente la eficiencia de los algoritmos escritos. Es claro que para poder programar procedimientos eficientes sobre grafos es necesario contar con distintas implementaciones de estructuras de datos avanzadas como las pilas, las colas, los montones (*heaps*) y las asociaciones llave-valor (*maps*), entre otras. Por ejemplo el algoritmo de Dijkstra aplicado sobre un grafo disperso obtiene una mejor complejidad temporal usando una cola de prioridad implementada con un *Fibonacci Heap* [1], que usando un árbol ordenado implementado con árboles Rojinegros (o peor aún, un simple arreglo).

- No provee un mecanismo sencillo para definir la función de costos de un grafo, obligando a que ésta se suministre a través de una lista de valores numéricos cuyos elementos son enumerados en el mismo orden en el que se definieron los arcos del grafo. De ninguna manera se permite definir la función de costos de un grafo a través de una fórmula matemática que establezca el costo exacto asociado a cada uno de sus arcos.

- El lenguaje no se puede extender fácilmente para la manipulación de otras estructuras de datos avanzadas como los árboles binarios, los árboles enearios (incluyendo los *Tries* y los *Quadtrees*), los autómatas finitos, los autómatas de pila, y las máquinas de Turing, puesto que su sintaxis se limita a la programación sobre grafos.
- No permite usar clases implementadas en *Java* ni permite que desde *Java* se puedan usar funciones implementadas en *GOLD 2*. En muchas ocasiones resulta útil embeber o invocar código nativo de un lenguaje de alto nivel dentro del código correspondiente al lenguaje de propósito específico. Por ejemplo, *JavaCC* [9] permite describir compiladores mezclando código específico del lenguaje con código escrito en *Java*, *PHP* permite diseñar páginas *WEB* embebiendo código *HTML*, y otros lenguajes (como *Groovy*) [13] permiten la invocación de rutinas implementadas en lenguajes de propósito general como *Java* y *C++*.
- No permite la invocación de funciones declaradas en otros archivos, lo que afecta de forma dramática el desarrollo de software porque impide la reutilización de código escrito anteriormente, ya sea en *Java* o en *GOLD 2*. Sin duda alguna es esencial que el lenguaje cuente con un mecanismo que permita utilizar rutinas implementadas en algún lenguaje de propósito general y funciones *GOLD* implementadas en otros archivos. Al permitir el uso de clases codificadas en *Java*, el lenguaje podría ganar bastante expresividad sin tener que alterar demasiado su gramática.
- No permite animar gráficamente la operación de los algoritmos en tiempo de ejecución. Aunque *GOLD 2* provee una instrucción para desplegar un grafo de forma gráfica usando la librería *GIDEN* [11] y otra instrucción para imprimir el estado de todas las variables declaradas, no es posible depurar gráficamente paso a paso la ejecución de cualquier algoritmo, visualizar grafos con una herramienta distinta a *GIDEN* ni implementar algoritmos diferentes para su dibujado. Además, no permite al usuario especificar en sus programas los atributos visuales de los nodos y arcos de los grafos dibujados a través de *GIDEN*, y mucho menos permite alterarlos en tiempo de ejecución.
- No provee un entorno de desarrollo con funcionalidades como la coloración de la sintaxis (*syntax highlighting*), el indentamiento automático del código fuente (*code formatting*), el emparejamiento de paréntesis (*bracket matching*), la validación de la sintaxis resaltando los errores de compilación en tiempo de desarrollo (*code validation*), el despliegue de ayudas de contenido (*content assist*) y el completado automático de código (*code completion*).

El único algoritmo típico de teoría de grafos que se presentó en la tesis *GOLD 2* fue el algoritmo de Dijkstra. A continuación se muestra la versión distribuida en las fuentes del proyecto *GOLD 2* [†3]:

Código 2.3. *Implementación del algoritmo de Dijkstra en GOLD 2 [4].*

```
1  begin
2    // "Example graph"
3    Set N := {1,2,3,4,5};
4    Set E := {|(1,2),(1,3),(1,4),(1,5),(3,2),(3,4),(4,2),(5,4)|};
5    Graph G := (N,E);
6    list edgeCost := {10,100,30,50,5,50,20,10};
7    AddAttr edgeCost in G Edges;
8    // "Temporal variables"
9    real isNodeComplete := 0.0;
10   real distance2CurrentNode := 0.0;
11   real saveDistance2CurrentNode := 0.0;
12   real possibleDistance2CurrentNode := 0.0;
13   real realCost := 0.0;
14   Node current := node 1 of G;
```

[3] Una versión distinta, presentada en la sección §3.4.1, fue la que se incluyó en el documento de tesis de *GOLD 2*.

```
15   Edge currentEdge := [(current,current)];
16   // "Algorithm Parameters"
17   setinitial 1 in G;
18   list distance := {0,infi,infi,infi,infi};
19   AddAttr distance in G Nodes;
20   AddAttr complete like 0 forAll G Nodes;
21   SetNodes unresolved copy G Nodes;
22   SetNodes adjacent := nodesOut current of G;
23   void MyDijkstra() begin
24     foreach unresolved begin
25       // "elimina del conjunto unresolved todos los elementos con la marca complete en 1"
26       foreach unresolved begin
27         isNodeComplete := getAttr complete of G item;
28         if (isNodeComplete = 1) then
29           delete item of unresolved;
30         end
31       end
32       adjacent := nodesOut item of G;
33       current copy item;
34       foreach adjacent begin
35         // "elimina del conjunto unresolved todos los elementos con la marca complete en 1"
36         distance2CurrentNode := getAttr distance of G current;
37         saveDistance2CurrentNode := getAttr distance of G item;
38         currentEdge := edge (current, item) of G;
39         realCost := getAttr edgeCost of G currentEdge;
40         possibleDistance2CurrentNode := realCost + distance2CurrentNode;
41         if (possibleDistance2CurrentNode < saveDistance2CurrentNode) then
42           assign possibleDistance2CurrentNode in distance of item;
43         end
44       end
45       assign 1.0 in complete of item;
46     end
47   end
48   print(Graph G nodesAtt distance);
49   MyDijkstra();
50   print(Graph G nodesAtt distance);
51 end
```

Se puede ver que el código exhibido es complicado de entender (usa una gran cantidad de instrucciones difíciles de recordar), no es eficiente (su complejidad es $O(n^2)$ donde n es la cantidad de nodos del grafo) y no está estructurado dentro de una función que pueda ser invocada sobre cualquier grafo y cualquier nodo inicial (aunque esté declarada la rutina `MyDijkstra`, que no tiene parámetros). Por todas las razones expuestas anteriormente es indispensable rediseñar por completo el lenguaje para permitir la escritura de algoritmos sobre grafos de una manera cómoda, como los pseudoalgoritmos implementados en el libro *Introduction to Algorithms* de Thomas Cormen et al. [1].

Capítulo 3

Estado del arte

Existen diversas librerías y lenguajes de propósito específico que permiten implementar algoritmos sobre grafos mediante instrucciones de control que no son fáciles de recordar y operaciones complicadas que obstaculizan la programación y dificultan el desarrollo de software que necesita el uso de estructuras de datos especializadas. Matemáticos, científicos e ingenieros suelen utilizar diferentes productos de software cuando necesitan resolver problemas sobre grafos, entre los que se pueden mencionar el problema de la ruta más corta, el problema del árbol de expansión mínimo, el problema del agente viajero, problemas de conectividad en redes, problemas de coloreo de nodos y problemas de flujo en redes. Por un lado, los programadores frecuentan el uso de algún lenguaje de propósito general enriquecido con una librería externa o la aplicación de algún lenguaje de propósito específico limitado. Por otro lado, los ingenieros industriales y matemáticos están acostumbrados a usar programas de escritorio especializados en visualizar y manejar grafos como *GIDEN* [11] y *Graphviz* [14], y en resolver problemas genéricos de optimización como *GAMS*, *MOSEK*, *Xpress* y *Solver*, que pueden llegar a administrar grafos si se configuran adecuadamente las variables, las restricciones y las funciones objetivo.

Este capítulo enumera varios lenguajes, librerías y aplicativos de escritorio que existen en la actualidad para describir, manipular e implementar algoritmos sobre grafos. Como revisión del estado del arte se estudiaron algunas herramientas concebidas exclusivamente para describir grafos o para programar algoritmos sobre grafos.

3.1. Lenguajes para describir grafos

Existen muchos lenguajes de propósito específico especializados en describir grafos, dado su conjunto de vértices y su conjunto de arcos, que han sido diseñados para suplir alguno de los siguientes objetivos:

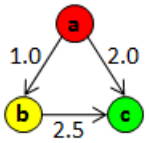
- proveer un formato de intercambio de información estructurada orientada a grafos que permita el flujo de información entre diferentes componentes o sistemas de software; y
- proveer un mecanismo para la visualización de grafos en dos dimensiones a través de la configuración de los atributos gráficos de sus vértices y de sus arcos.

En cualquiera de las situaciones, estos lenguajes poseen las siguientes falencias:

- no permiten la definición de grafos a través de expresiones matemáticas que describan por comprensión su conjunto de vértices y su conjunto de arcos;
- no permiten la manipulación algorítmica de grafos puesto que no son lenguajes de programación que incluyen instrucciones de control diseñadas para tal fin; y
- no permiten configurar más atributos gráficos que los predefinidos por el lenguaje.

Claramente, sin un mecanismo que permita definir los vértices y los arcos de un grafo por comprensión (describiendo matemáticamente las propiedades que cumplen sus elementos) entonces no queda otro remedio que hacerlo manualmente por extensión (enumerando sus elementos), lo que involucra un trabajo tedioso y repetitivo de adición de vértices y de arcos que puede resultar inviable si el grafo tiene cientos de nodos. El único de los lenguajes descriptivos analizados en el marco teórico que permite describir formalmente los grafos a través de expresiones matemáticas es *GOLD 1*, uno de los precursores del proyecto *GOLD 3*.

Figura 3.1. *Grafo de ejemplo para ilustrar el uso de algunos lenguajes de descripción de grafos.*



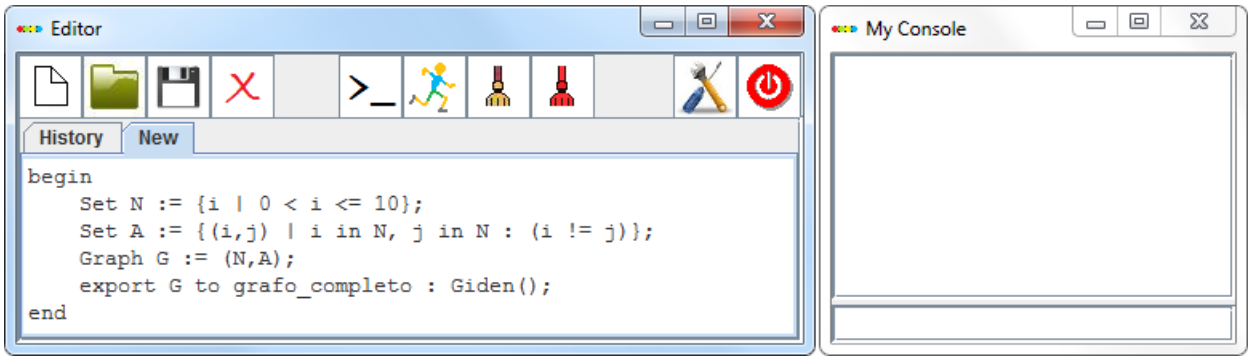
Código 3.1. *Definición del grafo de ejemplo 3.1 como se debería hacer en GOLD 3.*

```
graph := GDirectedGraph({'a','b','c'})
graph.addEdge('a','b',1.0)
graph.addEdge('b','c',2.5)
graph.addEdge('a','c',2.0)
```

3.1.1. *GOLD 1*

La primera versión de *GOLD*, bautizada en este documento como *GOLD 1*, es el producto de la tesis de pregrado *GOLD: un lenguaje orientado a grafos y conjuntos* [3] de Luis Miguel Pérez (véase la sección §2.2), que está basada en la tesis de maestría *CSet: un lenguaje para composición de conjuntos* [2] de Víctor Hugo Cárdenas. *GOLD 1* es un lenguaje de propósito específico diseñado para describir matemáticamente grafos definiendo formalmente su conjunto de vértices y su conjunto de arcos, convirtiéndose así en un lenguaje descriptivo de grafos que no ofrece posibilidad de manipularlos algorítmicamente.

Figura 3.2. *IDE de GOLD 1 [3], ilustrando la definición de un grafo de diez nodos.*



Código 3.2. *Definición del grafo de ejemplo 3.1 en GOLD 1.*

```
begin
  Set N := {'a','b','c'};
  Set A := {('a','b'),('b','c'),('a','c')};
  Graph G := (N,A);
end
```

Además de permitir la definición de grafos, *GOLD 1* da la opción de exportarlos y visualizarlos a través de la librería *GIDEN* [11]. Como desventajas de *GOLD 1* se puede mencionar que no provee instrucciones de control para manipular grafos, que no usa caracteres *Unicode* especiales para representar los símbolos de los operadores matemáticos, que cuenta con una cantidad limitada de tipos de datos, y que la función de costo de los grafos se debe definir arco por arco mediante un archivo de texto que se debe cargar con una instrucción especial.

3.1.2. *GML*

GML [15] (*Graph Modelling Language*) es un formato de archivos portable, extensible y flexible desarrollado en la Universidad de Passau para el intercambio de grafos entre diferentes programas computacionales. *GML* tiene una sintaxis simple que está diseñada para describir grafos adjuntando cualquier tipo de información sobre sus nodos y sobre sus arcos [15], incluso estructuras de datos. Aunque *GML* directamente no ofrece ningún mecanismo para visualizar ni para manipular grafos, es el formato de archivos estándar usado en varios sistemas incluyendo *Graphlet* (sin soporte en la actualidad), que fue desarrollado en la misma Universidad como una herramienta basada en *GTL* [16] para la implementación de editores y de algoritmos de visualización de grafos [15].

Código 3.3. *Definición del grafo de ejemplo 3.1 en GML.*

```
graph [
  comment "G"
  directed 1
  IsPlanar 1
  node [
    id 1
    label "a"
  ]
  node [
    id 2
    label "b"
  ]
  node [
    id 3
    label "c"
  ]
  edge [
    source 1
    target 2
    label "1.0"
  ]
  edge [
    source 2
    target 3
    label "2.5"
  ]
  edge [
    source 1
    target 3
    label "2.0"
  ]
]
```

3.1.3. *Graphviz DOT*

Graphviz [14] (*Graph Visualization Software*) es un conjunto de herramientas orientadas hacia la visualización de grafos, que fueron creadas en los laboratorios de investigación de *AT&T*. *Graphviz* incluye un lenguaje de propósito

específico llamado *DOT*, diseñado para describir grafos a partir de instrucciones en texto plano que permiten especificar determinadas propiedades visuales de los nodos y de los arcos, como su forma, su color y su tamaño. Aunque *Graphviz* es una herramienta versátil para configurar las características visuales de los grafos, no permite la codificación de programas que los manipulen.

Código 3.4. *Definición del grafo de ejemplo 3.1 en DOT.*

```
digraph G {
  a [label="a" color=red];
  b [label="b" color=yellow];
  c [label="c" color=green];
  a -> b [label="1.0"];
  b -> c [label="2.5"];
  a -> c [label="2.0"];
}
```

3.1.4. Dialectos *XML*: *GraphML*, *GXL*, *DGML* y *XGMML*

GraphML [17] es un formato de archivo especializado en la descripción de grafos cuya sintaxis está basada en el lenguaje de marcas extensible *XML*. Concretamente, *GraphML* es un dialecto *XML* que permite definir las propiedades estructurales de un grafo (incluyendo grafos dirigidos, grafos no dirigidos e hipergrafos), con la posibilidad de añadir información adicional que puede describir representaciones gráficas, referencias a datos externos o datos de atributos específicos a la aplicación [17].

GXL [18] (*Graph eXchange Language*), *DGML* [19] (*Directed Graph Markup Language*) y *XGMML* [10] (*eXtensible Graph Markup and Modeling Language*) son otros tres dialectos *XML* especializados en la descripción de grafos, que fueron concebidos como formatos de intercambio estándar para transferir grafos entre distintos sistemas y para la persistencia de grafos en aplicaciones que los manipulen. El más reciente es *DGML*, que fue creado por *Microsoft Corporation* para representar grafos dirigidos, permitiendo al usuario etiquetar los nodos y los arcos con cualquier tipo de información [19].

La sintaxis de los cuatro lenguajes está descrita a través de archivos *DTD* (*Document Type Definition*) que definen cada uno de los elementos y atributos que se pueden mencionar. Aunque son idóneos para describir por enumeración los vértices y los arcos de los grafos, ninguno de los cuatro dialectos ofrece un visualizador ni un lenguaje de programación para su manipulación.

Código 3.5. *Definición del grafo de ejemplo 3.1 en GraphML [17].*

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
         http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <key id="d0" for="node" attr.name="color" attr.type="string"/>
  <key id="d1" for="edge" attr.name="weight" attr.type="double"/>
  <graph id="G" edgedefault="directed">
    <node id="a"><data key="d0">red</data></node>
    <node id="b"><data key="d0">yellow</data></node>
    <node id="c"><data key="d0">green</data></node>
    <edge id="e1" source="a" target="b"><data key="d1">1.0</data></edge>
    <edge id="e2" source="b" target="c"><data key="d1">2.5</data></edge>
    <edge id="e3" source="a" target="c"><data key="d1">2.0</data></edge>
  </graph>
</graphml>
```

Código 3.6. *Definición del grafo de ejemplo 3.1 en DGML [19].*

```
<?xml version="1.0" encoding="UTF-8"?>
<DirectedGraph Title="G" xmlns="http://schemas.microsoft.com/vs/2009/dgml">
  <Nodes>
    <Node Id="a" Label="a" Background="#FFFF0000"/>
    <Node Id="b" Label="b" Background="#FFFFFF00"/>
    <Node Id="c" Label="c" Background="#FF00FF00"/>
  </Nodes>
  <Links>
    <Link Source="a" Target="b" Cost="1.0"/>
    <Link Source="b" Target="c" Cost="2.5"/>
    <Link Source="a" Target="c" Cost="2.0"/>
  </Links>
  <Properties>
    <Property Id="Label" Label="Label" DataType="String"/>
    <Property Id="Background" Label="Background" DataType="Brush"/>
    <Property Id="Cost" DataType="String"/>
  </Properties>
</DirectedGraph>
```

3.2. Aplicaciones de escritorio para manipular grafos

Hoy en día hay varias herramientas computacionales diseñadas para la creación y manipulación de grafos a través de interfaces gráficas que le permiten al usuario editar las propiedades conceptuales y visuales de los vértices y de los arcos del grafo, para luego gráficamente simular la ejecución de algún proceso previamente implementado en la herramienta. Aunque estos productos son fáciles de usar para personas con precarias bases en programación, presentan los siguientes inconvenientes:

- no permiten la definición de grafos a través de expresiones matemáticas que describan por comprensión su conjunto de vértices y su conjunto de arcos; y
- no permiten la manipulación algorítmica de grafos puesto que no son lenguajes de programación que incluyen instrucciones de control diseñadas para tal fin.

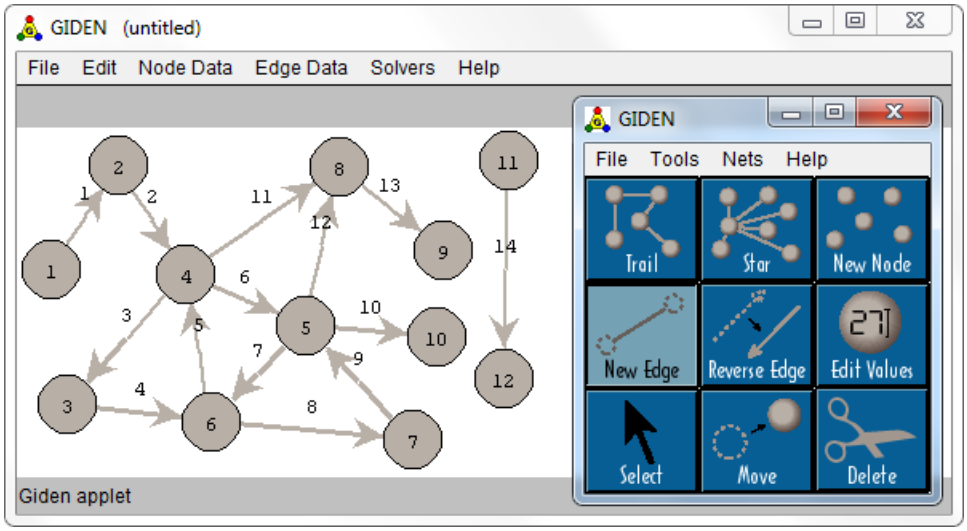
Las aplicaciones de escritorio para manipular grafos normalmente no permiten que el usuario pueda implementar nuevos algoritmos para poderlos ejecutar o simular gráficamente y, si incluyen un lenguaje para programarlos, este es fuertemente limitado por la herramienta. Además, estas aplicaciones dificultan la creación y modificación de grafos con cientos de nodos porque el proceso de edición es enteramente manual.

3.2.1. *GIDEN*

GIDEN [11] (*Graphical Implementation Development Environment for Networks*) es una herramienta desarrollada en la *Northwestern University* que opera como un entorno gráfico interactivo diseñado para facilitar la manipulación de grafos y la visualización paso a paso de la ejecución de algoritmos que solucionan determinados problemas de optimización sobre redes. *GIDEN* ofrece una interfaz gráfica interactiva con la que se pueden construir grafos, permitiendo ejecutar sobre éstos animaciones de determinados algoritmos capaces de solucionar el problema del árbol de expansión mínimo, el problema de la ruta más corta, el problema del flujo máximo o el problema de flujo máximo de costo mínimo [11]. *GIDEN* no es práctico para manipular grafos con cientos o miles de nodos porque todos deben ser suministrados por el usuario a través de la interfaz gráfica, no permite animar procesos que solucionen problemas distintos a los cuatro mencionados anteriormente, y mucho menos permite el desarrollo de nuevos algoritmos [†1].

[1] Parafraseo del análisis llevado a cabo en la tesis de Diana Mabel Díaz [4] sobre el aplicativo *GIDEN*.

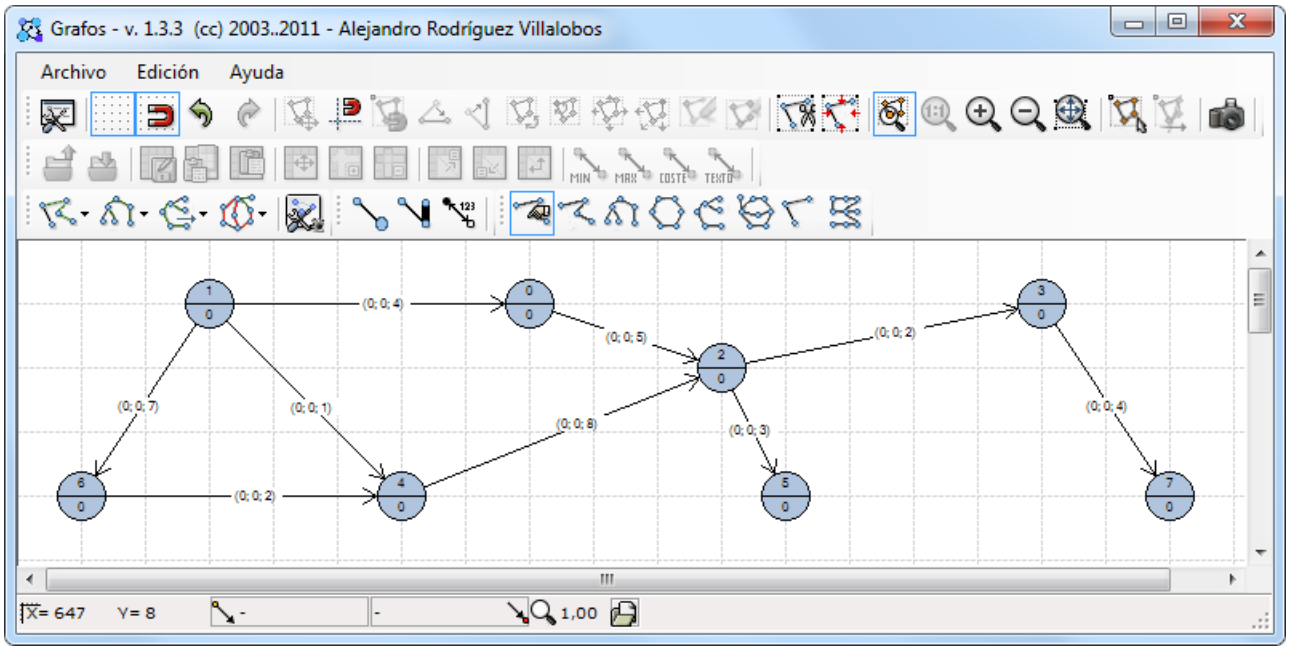
Figura 3.3. *Interfaz gráfica del aplicativo GIDEN [11].*



3.2.2. *Grafos*

Grafos [20] es un aplicativo de la *Universitat Politècnica de Catalunya* orientado a la enseñanza de la teoría de grafos, que permite el diseño de redes, la solución de ciertos problemas típicos sobre grafos y el análisis de los resultados obtenidos [20]. *Grafos* provee una interfaz gráfica para la edición de grafos a través de un formulario interactivo, y permite la ejecución animada de determinados algoritmos sobre diferentes problemas típicos, incluyendo el algoritmo de Dijkstra, el algoritmo de Bellman-Ford, el algoritmo de Floyd-Warshall, el algoritmo de Kruskal, el algoritmo de Prim y el algoritmo de Ford-Fulkerson [20]. Ninguno de los algoritmos provistos se puede editar y el usuario no puede programar sus propios procedimientos.

Figura 3.4. *Interfaz gráfica del aplicativo Grafos [20].*



3.3. Frameworks y librerías sobre grafos

Existe una colección de *frameworks* y librerías que ofrecen herramientas valiosas que facultan a los desarrolladores de software para que puedan manipular grafos y otras estructuras de datos avanzadas sobre un lenguaje de programación de propósito general como *Java* o *C++*. Estas herramientas permiten la creación de grafos con miles de nodos para su posterior manipulación mediante algoritmos implementados usando las instrucciones de control del lenguaje de propósito general y las estructuras de datos provistas por la misma librería, presentando las siguientes desventajas:

- no permiten la escritura de código fuente compacto, claro y legible (dependiendo del lenguaje);
- requieren que el programador domine con madurez el lenguaje de propósito general;
- requieren que el programador conozca con detalle las operaciones que la librería brinda para administrar cada estructura de datos; y
- dificultan y ralentizan la implementación de algoritmos que manipulan las estructuras de datos, incluso aquellos que son descritos a través de un pseudocódigo sencillo como el del algoritmo de Dijkstra [1].

Cada librería suministra un conjunto de clases que enriquecen el *API* estándar de *Java* o el *STL* de *C++* con implementaciones típicas de determinadas estructuras de datos que facilitan la algorítmica sobre los grafos. Aunque la infraestructura del lenguaje *GOLD 3* ofrece una librería propia que implementa una gran colección de estructuras de datos (véase la sección §5.1.1), también permite el uso de la librería estándar de *Java*, de cualquier librería externa y de cualquier clase que implemente el usuario, fomentando así la reutilización de código y el aprovechamiento de las bondades de la sintaxis del lenguaje para operar con mayor comodidad alguna librería existente. En otras palabras, el usuario estaría en capacidad de implementar en *GOLD 3* algoritmos que manipulen cualquier estructura de datos provista en alguna librería externa diseñada para *Java*. De hecho, el *framework* de *GOLD 3* incluye:

- la librería *JUNG* [21], usada como herramienta para la visualización de grafos;
- algunas clases de la librería *JGraphT* [22], que implementan montones (*heaps*) con *Fibonacci heaps* [1]; y
- algunas clases pertenecientes a la implementaciones de referencia de Cormen et al. [23], que implementan montones (*heaps*) con *Binomial heaps* [1].

Entonces, cualquier librería *Java* que ofrezca implementaciones a determinadas estructuras de datos puede estar sujeta a ser manipulada a través del lenguaje *GOLD 3*, convirtiéndose así en *aliados* más que en *rivales*.

3.3.1. *GTL*

GTL [16] (*Graph Template Library*) es una librería que suministra un conjunto de clases y algoritmos diseñados para manipular grafos. Existe una versión para *Java* que extiende el *API* estándar y una versión para *C++* que extiende el *STL* (*Standard Template Library*). Para implementar exitosamente algoritmos complejos sobre grafos usando *GTL*, los desarrolladores necesitarían gastar tiempo valioso aprendiendo a usar las funciones de la nueva librería, como se evidencia en los siguientes fragmentos de código.

Código 3.7. *Fragmento de la implementación del algoritmo de Dijkstra en Java usando GTL [16].*

```
1 public int run() {
2   distanceMap=new HashMapDouble();
3   NodeIterator nit=g.getNodeIterator();
4   while (nit.hasNext()) distanceMap.put(nit.next(),Double.POSITIVE_INFINITY);
5   distanceMap.put(source,0.0);
6   Map colourMap=new HashMap();
7   Heap greySet=new HeapTree(new Comparator() {
```

```
8     public int compare(Object o1, Object o2){
9       double d1=distanceMap.get(o1),d2=distanceMap.get(o2);
10      return d1<d2?-1:(d1==d2?0:1);
11    }
12  });
13  greySet.add(source);
14  colourMap.put(source,new Integer(GREY));
15  Node v=null;
16  while (!greySet.isEmpty()&&(!targetOnly||!target.equals(v))) {
17    v=(Node)greySet.deleteMin();
18    colourMap.put(v,new Integer(WHITE));
19    EdgeIterator it=v.getAdjEdgesIterator();
20    while (it.hasNext()) {
21      Edge edgeVW=it.next();
22      Node w=v.getOpposite(edgeVW);
23      if (colourMap.get(w)==null) {
24        greySet.add(w);
25        colourMap.put(w,new Integer(GREY));
26        distanceMap.put(w,distanceMap.get(v)+costMap.get(edgeVW));
27      }
28      else if (colourMap.get(w).equals(new Integer(GREY))) {
29        double distW=distanceMap.get(w),distV=distanceMap.get(v),costVW=costMap.get(edgeVW);
30        if (distW>distV+costVW){
31          greySet.remove(w);
32          distanceMap.put(w,distV+costVW);
33          greySet.add(w);
34        }
35      }
36    }
37  }
38  return GTL_OK;
39 }
```

Código 3.8. *Fragmento de la implementación del algoritmo de Dijkstra en C++ usando GTL [16].*

```
1  int dijkstra::run(graph& G) {
2    init(G);
3    less_dist prd(&dist,&mark);
4    bin_heap<node,less_dist> node_heap(prd,G.number_of_nodes());
5    mark[s]=grey;
6    dist[s]=0.0;
7    node_heap.push(s);
8    while (!node_heap.is_empty()) {
9      node cur_node=node_heap.top();
10     node_heap.pop();
11     mark[cur_node]=white;
12     if (cur_node==t) return GTL_OK;
13     node::adj_edges_iterator adj_edge_it;
14     node::adj_edges_iterator adj_edges_end=cur_node.adj_edges_end();
15     for (adj_edge_it=cur_node.adj_edges_begin();adj_edge_it!=adj_edges_end;++adj_edge_it) {
16       node op_node=(*adj_edge_it).opposite(cur_node);
17       if (mark[op_node]==black) {
18         mark[op_node]=grey;
19         dist[op_node]=dist[cur_node]+weight[*adj_edge_it];
20         node_heap.push(op_node);
21         if (preds_set) pred[op_node]=*adj_edge_it;
22       }
23       else if (mark[op_node]==grey) {
```

```
24          if (dist[op_node]>dist[cur_node]+weight[*adj_edge_it]) {
25            dist[op_node]=dist[cur_node]+weight[*adj_edge_it];
26            node_heap.changeKey(op_node);
27            if (preds_set) pred[op_node]=*adj_edge_it;
28          }
29        }
30      }
31    }
32    return GTL_OK;
33  }
```

Indudablemente, el algoritmo de Dijkstra escrito en *GTL* está lejos de ser evidente para quien tiene pocos conocimientos sobre los detalles internos de la librería y sobre *Java* o *C++*. Sin importar la pericia del programador, cualquier procedimiento sobre grafos terminaría implementándose en *Java* o en *C++*, lo que no es muy transparente para los programadores, sobre todo aquellos que están traduciendo pseudocódigo.

3.3.2. *Gravisto*

Gravisto [24] (*Graph Visualization Toolkit*) es un *framework* para la implementación de editores gráficos y de algoritmos de visualización sobre grafos. La herramienta está codificada en *Java*, está orientada a objetos, provee una estructura de datos fácil de extender para representar grafos y se puede instalar en el ambiente de desarrollo *Eclipse* como un *plug-in*. *Gravisto* es adecuado para implementar algoritmos de visualización sobre grafos puesto que está diseñado para tal fin; sin embargo, el *framework* no provee ninguna facilidad especial para implementar algoritmos clásicos sobre grafos ni para depurar estos algoritmos de forma animada. Además, como la única estructura de datos suministrada por la librería son los grafos, si el usuario deseara usar estructuras de datos avanzadas debería implementarlas por su propia cuenta o importarlas desde alguna otra librería externa. Finalmente, como *Gravisto* es una librería que extiende el *API* de *Java*, todo el código fuente que desarrolle el usuario tendría que ser código *Java*, que no es pseudocódigo por naturaleza.

Código 3.9. *Ejemplo de un algoritmo de visualización implementado en Java usando Gravisto [24].*

```
1  public class GuideExampleAlgorithm extends AbstractAlgorithm {
2    private IntegerParameter nodesParam;
3    private Bundle bundle=Bundle.getBundle(getClass());
4    public GuideExampleAlgorithm() {
5      nodesParam=new IntegerParameter(5,bundle.getString("parameter.nodes_cnt.name"),
6        bundle.getString("parameter.nodes_cnt.description"),0,50,0,Integer.MAX_VALUE);
7    }
8    protected Parameter<?>[] getAlgorithmParameters() {
9      return new Parameter[]{nodesParam};
10   }
11   public void check() throws PreconditionException {
12     PreconditionException errors=new PreconditionException();
13     if (nodesParam.intValue()<0) errors.add(bundle.getString("precondition.nodes_ge_zero"));
14     if (graph==null) errors.add(bundle.getString("precondition.graph_null"));
15     if (!errors.isEmpty()) throw errors;
16   }
17   public void execute() {
18     int n=nodesParam.getInteger().intValue();
19     Node[] nodes=new Node[n];
20     graph.getListenerManager().transactionStarted(this); // start a transaction
21     for (int i=0; i<n; ++i) { // generate nodes and assign coordinates to them
22       nodes[i]=graph.addNode();
23       CoordinateAttribute ca=(CoordinateAttribute)nodes[i].getAttribute(
24           GraphicAttributeConstants.GRAPHICS+Attribute.SEPARATOR+
```

```
25            GraphicAttributeConstants.COORDINATE);
26        ca.setCoordinate(new Point2D.Double(100+(i*100),100));
27      }
28      for (int i=1; i<n; ++i) graph.addEdge(nodes[i-1],nodes[i],true); // add edges
29      graph.getListenerManager().transactionFinished(this); // stop a transaction
30      graph.setDirected(true,true); // add arrows to edges
31    }
32    public String getName() {
33      return bundle.getString("name");
34    }
35  }
```

3.3.3. *FGL*

FGL [25] (*A Functional Graph Library*) es una librería desarrollada por Martin Erwin de la Universidad Estatal de Oregon para la manipulación de grafos a través de una colección de operaciones que se pueden usar en lenguajes funcionales como *ML* y *Haskell*.

Código 3.10. *Búsqueda por profundidad (Depth First Search) implementada en FGL [26].*

```
1  dfs :: [Node] -> Graph a b -> [Node]
2  dfs [] g = []
3  dfs vs Empty = []
4  dfs (v:vs) (c &^v g) = v : dfs (suc c ++ vs) g
5  dfs (v:vs) g = dfs vs g
```

Código 3.11. *Algoritmo de Dijkstra implementado en FGL [26].*

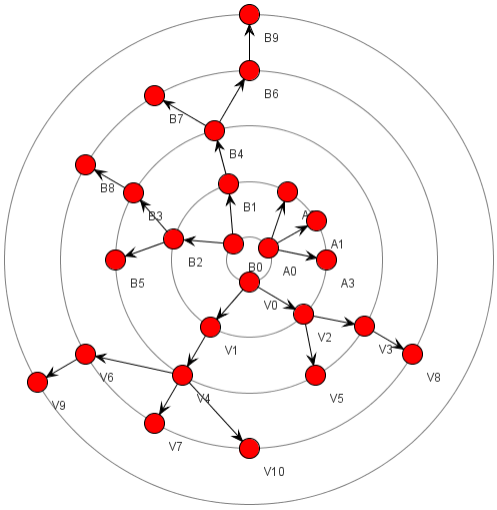
```
1  type Lnode a = (Node, a)
2  type Lpath a = [Lnode a]
3  type LRTree a = [Lpath a]
4  instance Eq a => Eq (Lpath a) where ((_,x):_) == ((_,y):_) = x == y
5  instance Ord a => Ord (Lpath a) where ((_,x):_) < ((_,y):_) = x < y
6  getPath Node -> LRTree a -> Path
7  getPath = reverse . map fst . first (\((w,_):_) -> w == v)
8  sssp :: Real b => Node -> Node -> Graph a b -> Path
9  sssp s t = getPath t . dijkstra (unitHeap [(s,0)])
10 expand :: Real b =>
11           b -> LPath b -> Context a b -> [Heap(LPath b)]
12 expand d p (_,_,_,s) = map(\(l,v) -> unitHeap((v,l+d):p)) s
13 dijkstra :: Real b =>
14             Heap(LPath b) -> Graph a b -> LRTree b
15 dijkstra h g
16    | isEmptyHeap h || isEmpty g = []
17 dijkstra (p@((v,d):_) << h) (c &^v g) =
18          p:dijkstra (mergeAll (h:expand d p c)) g
19 dijkstra (_ << h) g = dijkstra h g
```

3.3.4. *JUNG*

JUNG [21] (*Java Universal Network/Graph Framework*) es una librería *Java* de código abierto que ofrece un *API* ‘‘común y extensible para el modelado, análisis y visualización de datos que pueden ser representados a través de grafos o redes’’ [21]. La librería ‘‘incluye implementaciones de algunos algoritmos de teoría de grafos, minería de datos y análisis de redes sociales’’ [21], y ‘‘está diseñada para manipular una variedad de representaciones de

entidades y sus relaciones, como grafos dirigidos y no dirigidos, grafos multi-modales, grafos con arcos paralelos, e hipergrafos'' [21]. La librería *JUNG* se está utilizando en *GOLD 3* para dibujar árboles binarios, árboles enearios y grafos (véase la sección §6.2.1).

Figura 3.5. *Bosque (conjunto de árboles) dibujado con JUNG [21].*



3.3.5. *JGraphT*

JGraphT [22] es una librería *Java* con licencia *GPL* que ''provee algoritmos y objetos de la teoría de grafos'' [22], permitiendo ''la manipulación de varios tipos de grafo, incluyendo grafos dirigidos y no dirigidos, grafos simples, hipergrafos y pseudografos'' [22]. *JGraphT* implementa varias estructuras de datos especializadas en la manipulación de grafos y suministra un módulo de visualización de grafos que usa *JGraph* [27].

Código 3.12. *Algoritmo de Kruskal implementado en Java, incluido en la librería JGraphT [22].*

```
1  package org.jgrapht.alg;
2  import java.util.*;
3  import org.jgrapht.*;
4  import org.jgrapht.alg.util.*;
5  public class KruskalMinimumSpanningTree<V,E> {
6    private double spanningTreeCost;
7    private Set<E> edgeList;
8    public KruskalMinimumSpanningTree(final Graph<V,E> graph) {
9      UnionFind<V> forest=new UnionFind<V>(graph.vertexSet());
10     ArrayList<E> allEdges=new ArrayList<E>(graph.edgeSet());
11     Collections.sort(allEdges,new Comparator<E>() {
12       public int compare(E edge1, E edge2) {
13         return Double.valueOf(graph.getEdgeWeight(edge1))
14                .compareTo(graph.getEdgeWeight(edge2));
15       }
16     });
17     spanningTreeCost=0;
18     edgeList=new HashSet<E>();
19     for (E edge:allEdges) {
20       V source=graph.getEdgeSource(edge);
21       V target=graph.getEdgeTarget(edge);
22       if (forest.find(source).equals(forest.find(target))) continue;
23       forest.union(source,target);
24       edgeList.add(edge);
```

```
25        spanningTreeCost+=graph.getEdgeWeight(edge);
26      }
27    }
28    public Set<E> getEdgeSet() {
29      return edgeList;
30    }
31    public double getSpanningTreeCost() {
32      return spanningTreeCost;
33    }
34  }
```

Aunque *JGraphT* permite la escritura de código fuente legible y eficiente, hay que recordar que sigue siendo código escrito en *Java* que está lejos de parecerse al pseudocódigo que se maneja en libros como el de Thomas Cormen et al. [1]. La librería *JGraphT* se está utilizando en *GOLD 3* para proveer implementaciones a algunas estructuras de datos especializadas (véase la sección §6.2.2).

3.3.6. Implementaciones de referencia de Cormen et al.

El disco compacto distribuido con el texto guía *Introduction to Algorithms* de Thomas Cormen et al. [1] contiene una librería *Java*, publicada bajo el paquete `com.mhhe.clrs2e` [23] [†2], que provee implementaciones de referencia para la mayoría de las estructuras de datos y algoritmos presentados en el libro.

Código 3.13. *Algoritmo de Kruskal implementado en Java, incluido en el paquete com.mhhe.clrs2e [23].*

```
1  package com.mhhe.clrs2e;
2  import java.util.Iterator;
3  public class Kruskal implements MST {
4    public WeightedAdjacencyListGraph computeMST(WeightedAdjacencyListGraph g) {
5      WeightedAdjacencyListGraph a=(WeightedAdjacencyListGraph)g.useSameVertices();
6      DisjointSetUnion forest=new DisjointSetForest();
7      Object handle[]=new Object[a.getCardV()];
8      Iterator vertexIter=g.vertexIterator();
9      while (vertexIter.hasNext()) {
10       Vertex v=(Vertex)vertexIter.next();
11       handle[v.getIndex()]=forest.makeSet(v);
12     }
13     WeightedEdge[] edge=new WeightedEdge[g.getCardE()];
14     int i=0;
15     vertexIter=g.vertexIterator();
16     while (vertexIter.hasNext()) {
17       Vertex u=(Vertex)vertexIter.next();
18       WeightedEdgeIterator edgeIter=g.weightedEdgeIterator(u);
19       while (edgeIter.hasNext()) {
20         Vertex v=(Vertex)edgeIter.next();
21         if (u.getIndex()<v.getIndex()) {
22           double w=edgeIter.getWeight();
23           edge[i++]=new WeightedEdge(u,v,w);
24         }
25       }
26     }
27     MaxHeap heap=new MaxHeap();
28     (heap.makeSorter()).sort(edge);
29     for (i=0; i<edge.length; i++) {
30       Object uHandle=handle[edge[i].u.getIndex()];
```

[2] El nombre clave `mhhe` abrevia *McGraw-Hill Higher Education* y el nombre clave `clrs2e` abrevia los apellidos de los autores del libro *Introduction to Algorithms* [1] (Cormen, Leiserson, Rivest y Stein) con su número de edición (2 ed).

```
31        Object vHandle=handle[edge[i].v.getIndex()];
32        if (forest.findSet(uHandle)!=forest.findSet(vHandle)) {
33          a.addEdge(edge[i].u,edge[i].v,edge[i].w);
34          forest.union(uHandle,vHandle);
35        }
36      }
37      return a;
38    }
39    private static class WeightedEdge implements Comparable {
40      public Vertex u;
41      public Vertex v;
42      public double w;
43      public WeightedEdge(Vertex a, Vertex b, double weight) {
44        u=a;
45        v=b;
46        w=weight;
47      }
48      public int compareTo(Object o) {
49        WeightedEdge e=(WeightedEdge)o;
50        return w<e.w?-1:(w==e.w?0:1);
51      }
52      public String toString() {
53        return "("+u.getName()+","+v.getName()+","+w+")";
54      }
55    }
56  }
```

Los algoritmos implementados en el paquete `com.mhhe.clrs2e` están escritos en *Java* usando instrucciones particulares a este lenguaje de propósito general, siguiendo un estilo muy distinto al usado en los pseudocódigos exhibidos en el texto guía *Introduction to Algorithms* [1]. Esto es entendible puesto que el lenguaje *Java* no está diseñado para escribir pseudocódigos como los estudiados en la referencia [1]. La librería se está utilizando en *GOLD 3* para proveer implementaciones a algunas estructuras de datos especializadas (véase la sección §6.2.3).

3.4. Lenguajes para implementar algoritmos sobre grafos

A lo largo de la historia se han diseñado varios lenguajes de propósito específico especializados en la implementación de algoritmos sobre grafos, presentando algunos de los siguientes inconvenientes:

- no pueden integrarse transparentemente con librerías externas;
- no están enfocados en la legibilidad del código fuente escrito;
- no pueden embeberse fácilmente en grandes proyectos de software; y
- no tienen la misma expresividad que la de un lenguaje de propósito general como *Java* o *C++*.

3.4.1. *GOLD 2* (*GOLD+*)

La segunda versión de *GOLD*, nombrada *GOLD+* por su creador y bautizada en este documento como *GOLD 2*, es el producto de la tesis de maestría *GOLD+: lenguaje de programación para la manipulación de grafos: extensión de un lenguaje descriptivo a un lenguaje de programación* [4] de Diana Mabel Díaz (véase la sección §2.3), que está basada en la primera versión implementada por Víctor Hugo Cárdenas [2]. *GOLD 2* es un lenguaje de propósito específico que extiende el diseñado en *GOLD 1* para permitir la manipulación algorítmica de grafos con un lenguaje de programación sencillo basado en un conjunto limitado de instrucciones de control. Aunque *GOLD 2* permite

describir grafos y realizar determinadas operaciones básicas entre éstos, no es un lenguaje lo suficientemente potente para implementar efectivamente algoritmos clásicos como el de Dijkstra. El único algoritmo clásico en teoría de grafos que se incluye como ejemplo en el documento de tesis del lenguaje *GOLD+* es precisamente el algoritmo de Dijkstra, presentado a continuación:

Código 3.14. *Supuesto algoritmo de Dijkstra, implementado en GOLD+ [4].*

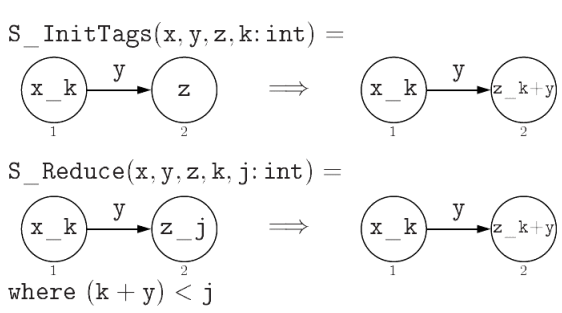
```
1  begin
2    Set N := {1,2,3,4,5};
3    Set E := {|(1,4),(1,5),(1,3),(1,2),(5,4),(4,2),(3,4),(3,5)|};
4    Graph G := (N,E);
5    Set nodeCost := {7,9,44,67};
6    list edgeCost := {10,100,30,50,5,50,20,10};
7    AddAttr edgeCost in G Edges;
8    setinitial 1 in G;
9    list distance := {0,infi,infi,infi,infi};
10   AddAttr distance in G Nodes;
11   Print(G);
12   Node actual := node 1 of G;
13   AddAttr complete like 0 forAll G Nodes;
14   SetNodes unresolved := G Nodes;
15   real b;
16   foreach unresolved begin
17     b := getAttr complete of G actual;
18     if 1 equals 1 then
19       delete item of unresolved;
20     end
21   end
22 end
```

Es evidente que el algoritmo implementado no es claro y que no funciona. Además, no está escrito como un procedimiento que se pueda invocar sobre cualquier grafo con costos y sobre cualquier nodo inicial.

3.4.2. *GP*

GP [28] (*Graph Programs*) ‘‘es un lenguaje de programación no determinístico basado en reglas de transformación diseñado para resolver problemas de grafos con un alto nivel de abstracción, liberando a los programadores del manejo de estructuras de datos de bajo nivel’’ [28]. El lenguaje *GP* fue creado por investigadores de la Universidad de York, quienes han escrito varios artículos estudiando formalmente su semántica operacional y su implementación prototipo ([29], [28]). Actualmente se está investigando sobre su semántica axiomática para plantear reglas de inferencia con las que sea posible verificar la corrección parcial de los programas escritos en el lenguaje [28].

Figura 3.6. *Implementación del algoritmo de Dijkstra en GP [28].*



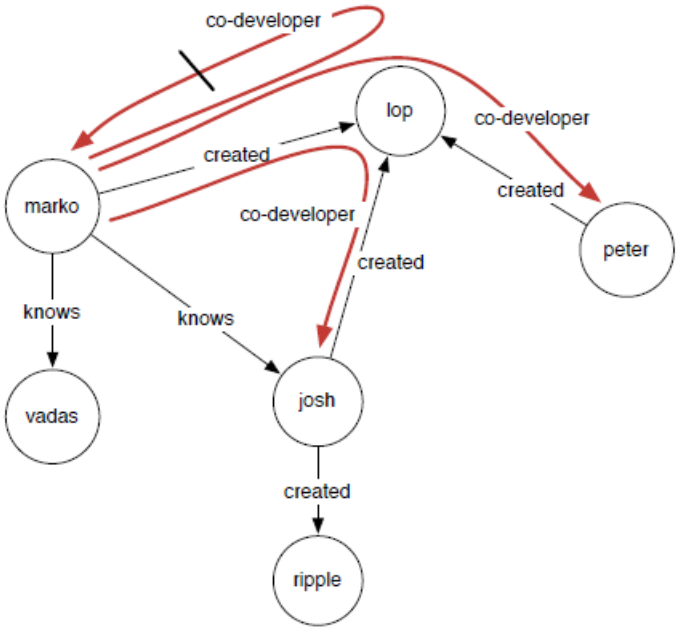
El lenguaje *GP*, analizado como objeto de estudio, es sumamente interesante pues está basado en reglas formales, pero su uso en grandes proyectos de software está limitado por el hecho de que no existe ningún mecanismo de integración con un lenguaje de propósito general como *Java* o *C++*. Además, *GP* puede ser complicado de usar para aquellos desarrolladores acostumbrados a la programación imperativa orientada a objetos puesto que no está estructurado a través de instrucciones imperativas clásicas como las asignaciones, los condicionales y los ciclos.

3.4.3. *Gremlin*

Gremlin [30] es un lenguaje de programación de propósito específico orientado hacia la consulta, análisis y manipulación de grafos [30], diseñado por Marko A. Rodríguez en el Laboratorio Nacional de Los Álamos para facilitar recorridos en grafos mediante la sintaxis provista por el lenguaje *XPath* [31] (*XML Path Language*) y la infraestructura provista por el lenguaje dinámico *Groovy* [13]. *Gremlin* puede embeberse dentro de aplicaciones *Java* con el *API* de *scripting* brindado desde *JDK 6* ([32], [33]) y puede trabajar con determinadas bases de datos y *frameworks* sobre grafos a través de la colección de interfaces e implementaciones *Blueprints* [30], que provee un mecanismo similar a *JDBC* para administrar bases de datos creadas bajo el *property graph data model* [30].

Gremlin fue diseñado para el análisis y manipulación de cierto tipo de grafos dirigidos, denominados en la literatura de *Gremlin* como *property graphs* [30], cuyos vértices tienen un identificador único, un conjunto de arcos de salida, un conjunto de arcos de entrada y una colección de propiedades definidas en una asociación llave-valor [30], y cuyos arcos tienen un identificador único, un vértice origen, un vértice destino, una etiqueta que denota el tipo de relación entre sus dos vértices y una colección de propiedades definidas en una asociación llave-valor [30].

Figura 3.7. *Grafo de ejemplo trabajado en el programa 3.15 [30].*



Código 3.15. *Un recorrido simple en Gremlin para obtener los co-desarrolladores de Marko A. Rodríguez [30].*

```
1 gremlin> ./@name
2 ==>marko
3 gremlin> ./outE[@label='created']/inV/inE[@label='created']/outV[g:except($_)]/@name
4 ==>josh
5 ==>peter
```

Aunque *Gremling* es adecuado para manipular grafos a través de recorridos, no está diseñado para la implementación de cualquier tipo de algoritmo general sobre grafos, como el algoritmo de Dijkstra.

3.4.4. *GRAAL*

GRAAL [34] (*GRAph Algorithmic Language*) es un lenguaje de programación propuesto en los años setenta como una extensión de *ALGOL 60* para la descripción e implementación de algoritmos sobre grafos usando los conjuntos como piedra angular. Aunque *GRAAL* fue inventado mucho tiempo antes que la mayoría de lenguajes de programación que conocemos hoy en día, muchas razones que motivaron su creación son las mismas que justificaron el nacimiento de *GOLD*, como lo muestra el siguiente fragmento del artículo *GRAAL: On a programming language for graph algorithms* [34] escrito por los creadores de *GRAAL* en 1972:

> *''For the implementation of a graph algorithm on a computer, standard algorithmic languages, such as FORTRAN or ALGOL, are, in general, rather unsuitable. In fact, they are neither well-adapted to expressing basic graph operations, nor to describing and manipulating most of the data structures upon which these operations are defined. Although list processing languages provide for a more appropriate data structure, they tend to hide the graph theoretical nature of the algorithms besides leading to slow execution and large demands for storage. This points to the need for the development of special-purpose languages which facilitate the programming as well as the publication of graph algorithms. [...] In this article we propose such a language -- named GRAAL (GRAph Algorithmic Language) -- for use in the solution of graph problems of the type primarily arising in applications. These problems involve a wide variety of graphs of different types and complexity; and one of the objectives in the design of GRAAL was to allow for this range of possibilities with as little degradation as possible in the efficient implementation and execution of an algorithm designed for a specific type of problem. Our second objective relates to the need for a language which facilitates the design and communication of graph algorithms independent of the computer. In line with this, we aimed at ensuring a concise and clear description of such algorithms in terms of data objects and operations natural to graph theory, that is, without introducing too many instructions required mainly by programming considerations.'' [34]*

Código 3.16. *Un programa implementado en GRAAL para encontrar un árbol de expansión [34].*

```
1  procedure spantree(G,u,Tree);
2  graph G, Tree; set u;
3  comment This procedure generates a directed spanning tree with root u
4          for the connected component of G containing the node u;
5  begin set S, T, w, x, y, a;
6    assign(Tree,u); S:=u; T:=cob(G,u);
7    while T≠empty do
8    begin for all a in T do
9      begin w:=bd(G,A); y:=w∼S;
10       if y≠empty then
11       begin S:=S∪y; x:=w∼y; assign(Tree,x−y to a) end
12     end;
13     T:=cob(G,S)
14   end
15 end
```

GRAAL permite la implementación de algoritmos sobre grafos usando operadores de la teoría de conjuntos y funciones especializadas suministradas por el lenguaje, dando énfasis a la legibilidad del código fuente desarrollado. A pesar de que años después de la creación de *GRAAL* se propuso una versión basada en *Fortran* llamada *FGRAAL* (*Fortran extended GRAph Algorithmic Language*) y se creó independientemente un paquete de procedimientos llamado *GRAAP* [35] (*GRaph Algorithmic Applications Package*) para dotar a *ALGOL 68* de operaciones especializadas sobre teoría de grafos, todos estos instrumentos cayeron en desuso para dar paso a lenguajes de programación más modernos, incluyendo los orientados a objetos. De cierta manera, *GOLD* está retomando los objetivos de *GRAAL* en un mundo diferente donde existen nuevas herramientas y lenguajes de propósito general avanzados como *Java* y *C++*.

Capítulo 4

Marco teórico

En este capítulo se exponen los fundamentos teóricos en los que está basado el presente documento de tesis, incluyendo algunos tópicos generales sobre lenguajes de propósito general y lenguajes de propósito específico, suponiendo que el lector ya tiene conocimientos básicos sobre estructuras de datos y sus implementaciones típicas. Para quienes no cuentan con bases fuertes sobre estructuras de datos, se recomienda una lectura somera de algunas de las siguientes referencias, enumeradas en orden de importancia:

1. *Introduction to Algorithms* [1] de Thomas H. Cormen et al.;
2. *Data Structures and Algorithms in Java* [36] de Adam Drozdek;
3. *Data Structures & Algorithms in JAVA* [37] de Robert Lafore;
4. *Diseño y manejo de estructuras de datos en C* [38] de Jorge Villalobos; y
5. *Lenguajes, gramáticas y autómatas: Curso básico* [39] de Rafel Cases y Lluís Màrquez.

Para estudiar el tema de grafos se sugiere inspeccionar el texto de Cormen [1] y para el de autómatas se recomienda revisar el libro de Cases y Màrquez [39].

4.1. Lenguajes de propósito general

Los *lenguajes de programación de propósito general* (*GPL* por sus siglas en inglés: *general-purpose programming languages*) son lenguajes de programación diseñados para resolver problemas sobre una gran cantidad de dominios, al contrario de los *lenguajes de programación de propósito específico* (*DSL* por sus siglas en inglés: *domain-specific programming languages*), que limitan su aplicación sobre algún dominio particular. Cada lenguaje de programación se define a través de su sintaxis (forma) y su semántica (significado) [40], que en muchas ocasiones están orientadas para permitir la implementación de estructuras de datos y la descripción de algoritmos que las manipulan.

Los conceptos tratados en esta sección están completamente basados (directa o indirectamente) en las referencias:

1. *Programming languages : design and implementation* [41] de Terrence W. Pratt y Marvin V. Zelkowitz;
2. *Programming Language Pragmatics* [42] de Michael L. Scott; y
3. *Programming Language Design Concepts* [40] de David A. Watt.

4.1.1. Introducción

Un buen lenguaje de programación, al igual que una buena notación matemática, debe ayudar al desarrollador a formular y comunicar sus ideas claramente [40], satisfaciendo los siguientes requerimientos fundamentales [40]:

- *Un lenguaje de programación debe ser universal*. Todo problema que pueda ser resuelto por un algoritmo computacional debe tener una solución que pueda ser programada en el lenguaje. Cualquier lenguaje en el que se puedan definir funciones recursivas es universal.
- *Un lenguaje de programación debería ser intuitivo*. Los problemas que forman parte de su área de aplicación deben poderse resolver de una forma razonable y natural para el desarrollador.
- *Un lenguaje de programación debería ser implementable en un computador*. Debe ser posible ejecutar computacionalmente cualquier programa bien escrito en el lenguaje. La notación matemática en su completa generalidad y el lenguaje natural no son clasificados como lenguajes de programación puesto que no son implementables.
- *Un lenguaje de programación debería poder contar un una implementación razonablemente eficiente*. En la práctica, los programadores esperan que sus programas sean casi tan eficientes como sus correspondientes programas en *lenguaje de máquina* (tal vez por un factor constante), dependiendo de la arquitectura del computador y de las características del lenguaje.

Es indispensable entender los conceptos teóricos inherentes a los lenguajes de programación para explotarlos con una mayor efectividad y así poder construir programas confiables y mantenibles [40]. Asimismo, es importante aprender a decidir cuál lenguaje de programación es más apropiado para resolver un determinado problema [40]. Sin importar su origen, todo lenguaje de programación está definido a través de tres aspectos fundamentales [40]:

- *Sintaxis*. Se preocupa de la forma de los programas (cómo las expresiones, comandos, declaraciones y otras sentencias pueden ser organizadas para constituir un programa bien formado), influenciando la manera en la que los programas son escritos por el programador, leídos por otros desarrolladores y analizados por un computador.
- *Semántica*. Se preocupa del significado de los programas (cómo se espera que se comporte un programa bien formado cuando es ejecutado sobre las máquinas), determinando la manera en la que los programas son armados por el programador, entendidos por otros desarrolladores e interpretados por un computador [†1].
- *Pragmática*. Se preocupa de la forma en la que el lenguaje pretende ser usado en la práctica, influenciando como se espera que los programadores diseñen e implementen programas en un entorno real.

Si son estudiados con respecto al nivel de abstracción que ofrecen a los desarrolladores, los lenguajes de programación pueden dividirse en dos grandes categorías: los *lenguajes de alto nivel* que son relativamente independientes de las máquinas en las que sus programas son ejecutados [40], y los *lenguajes de bajo nivel* (también llamados *lenguajes de máquina*) que permiten manipular con libertad los componentes internos de la arquitectura de las máquinas. Los lenguajes de alto nivel son implementados transformando programas en *lenguaje de máquina* (*compilándolos*), ejecutando directamente cada una de sus instrucciones (*interpretándolos*) o mediante alguna combinación de ambas [40]. Los *procesadores de lenguaje* (*language processors*), también conocidos como *entornos de desarrollo integrado* (*IDE* por sus siglas en inglés: *integrated development environment*), son sistemas especializados en el procesamiento de programas escritos en el lenguaje, que incluyen compiladores, intérpretes y herramientas auxiliares como editores de código fuente y depuradores [40]. Por otro lado, el *núcleo del lenguaje* comprende únicamente su compilador, que está conformado por el analizador léxico, el analizador sintáctico y el analizador semántico.

[1] La descripción dada corresponde con su *semántica operacional*. En la sección §4.1.3 se enuncian otras formas de semántica igualmente importantes, como la *semántica axiomática* y la *semántica denotacional*.

4.1.1.1. Paradigmas de programación

Las distintas maneras en las que un lenguaje de programación teje los conceptos básicos individuales influyen radicalmente en el estilo de programación subyacente, denominado *paradigma* [40]:

1. *Programación imperativa* (*imperative programming*). Se caracteriza por el uso de comandos y procedimientos que actualizan el estado de las variables declaradas. Los programas son vistos como secuencias de instrucciones de control que describen detalladamente la operación de un determinado algoritmo.

2. *Programación orientada a objetos* (*object-oriented programming*). Se caracteriza por el uso de *clases* que representan la abstracción de una familia de objetos de la realidad con propiedades (*atributos*) y comportamiento (*métodos*) similares, y de *objetos* que representan instancias de clases cuyos atributos tienen valores específicos. A través de mecanismos de *composición*, *agregación* y *herencia* se pueden modelar diversas relaciones entre conceptos, fomentando la reutilización de código dentro de las distintas clases.

3. *Programación concurrente* (*concurrent programming*). Se caracteriza por el uso de procesos concurrentes y de abstracciones de control que permiten declararlos. Los *programas concurrentes* son capaces de llevar a cabo más de una operación al mismo tiempo mediante la ejecución simultánea de varios hilos de ejecución (*threads*). En contraparte, los *programas secuenciales* imperativos se componen de comandos que se ejecutan en secuencia uno después de otro a medida que van terminando de operar, prohibiendo que más de una acción sea ejecutada al mismo tiempo.

4. *Programación funcional (*functional programming*)*. Se caracteriza por el uso de funciones sobre tipos de datos como listas y árboles, prescindiendo de las variables y asignaciones propias de la programación imperativa. Algunos lenguajes funcionales tratan las funciones como valores ordinarios que pueden ser pasados como parámetro o ser retornados como resultado de otras funciones. Además, se suelen incorporar avanzados sistemas de tipado (*type systems*) para permitir la escritura de funciones polimórficas que operen sobre parámetros de una diversa variedad de tipos.

5. *Programación lógica* (*logic programming*). Se caracteriza por el uso de relaciones, actuando como un subconjunto de la lógica de predicados. Los programas escritos bajo este paradigma son capaces de inferir relaciones entre valores en lugar de calcular un valor de salida dados unos valores de entrada.

6. *Lenguajes de scripting* (*scripting languages*). Se caracterizan por la declaración de pequeñas rutinas de alto nivel (*scripts*) que son escritas muy rápidamente para acoplar subsistemas codificados en otros lenguajes. Cada *script* almacena una secuencia habitual de comandos que puede invocarse en el momento que se necesite.

Existen diversos lenguajes diseñados para programar como *Fortran*, *ALGOL*, *Lisp*, *COBOL*, *BASIC*, *Pascal*, *Prolog*, *C*, *Scheme*, *C++*, *Perl*, *Haskell*, *Python*, *Visual Basic*, *Ruby*, *PHP*, *Java* y *C#* (entre otros), cada uno con sus seguidores y contradictores. Muchos de los lenguajes de programación siguen el paradigma de la *programación imperativa*, en la que los programas son tratados como secuencias de instrucciones que alteran el estado de las variables mediante el uso de comandos y procedimientos, describiendo paso o paso cómo se realiza una determinada tarea (i.e., *cómo se hace*). Sin embargo, existen otros lenguajes de programación, que siguen el paradigma de la *programación declarativa*, en la que los programas son expresados mediante la declaración de reglas lógicas que describen el problema sin exhibir explícitamente las instrucciones que componen el algoritmo que lo soluciona (i.e., *qué se hace*). El desarrollo histórico de los lenguajes de programación puede consultarse en la referencia *Programming Language Design Concepts* [40], donde varios lenguajes representativos son organizados en las líneas de tiempo de las figuras 4.1 y 4.2. Por otro lado, la figura 4.3 resume las fechas y la ascendencia de algunos de los lenguajes de programación más importantes [40].

Figura 4.1. *Línea de tiempo con la fecha de aparición de algunos lenguajes de programación imperativos.*

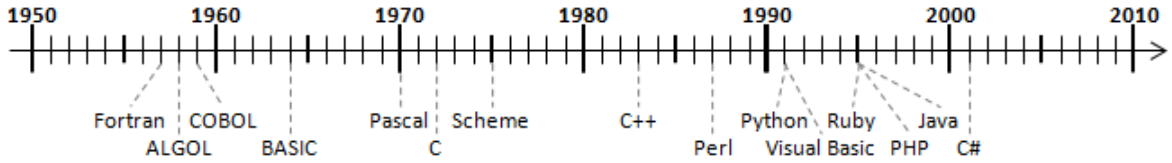


Figura 4.2. *Línea de tiempo con la fecha de aparición de algunos lenguajes de programación declarativos.*

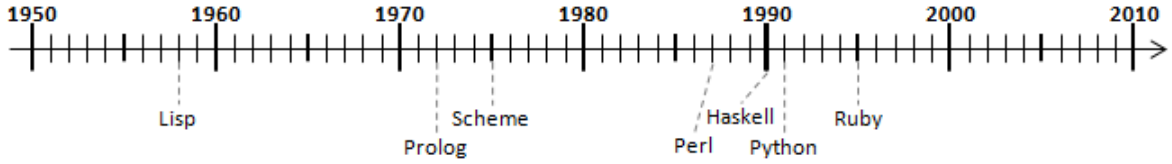
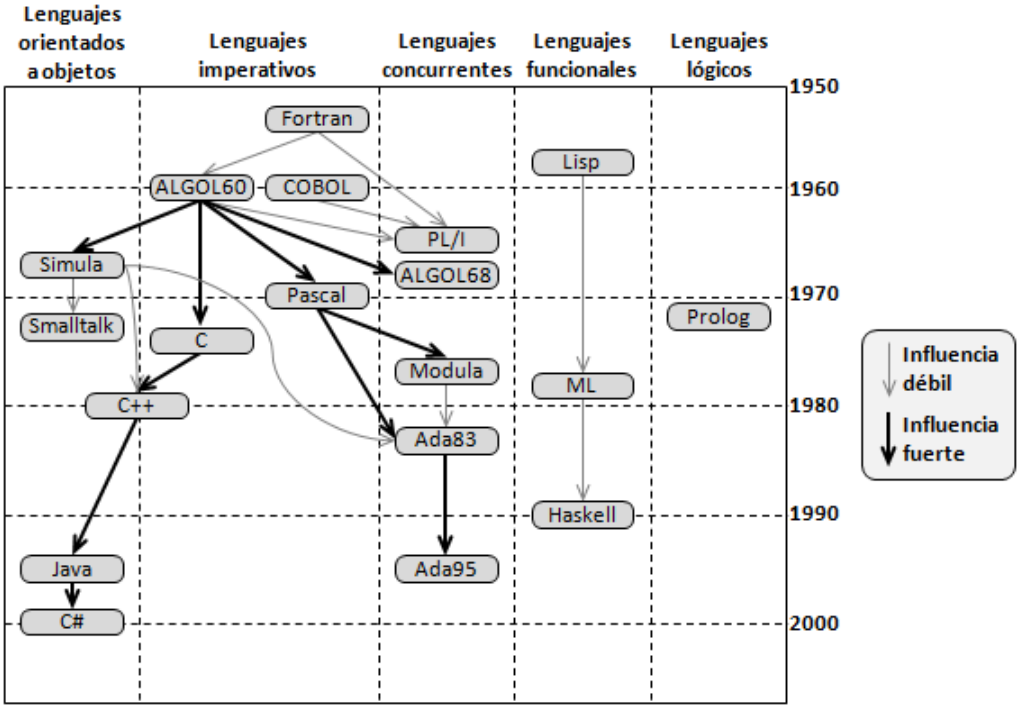


Figura 4.3. *Linaje de algunos de los lenguajes de programación más importantes [40].*



4.1.1.2. Criterios para evaluar un lenguaje de programación

De acuerdo con Pratt y Zelkowitz, algunos criterios que debe satisfacer un buen lenguaje de programación son [41]:

1. *Claridad, simplicidad y unidad* (*Clarity, simplicity and unity*). El lenguaje debe proveer un conjunto de conceptos claro, simple y unificado que puedan ser usados como primitivas al momento de desarrollar algoritmos. Adicionalmente, la sintaxis del lenguaje debe favorecer la legibilidad, facilitando la escritura de algoritmos de tal forma que en el futuro sean fáciles de entender, de probar y de modificar. Según Abelson y Sussman, ‘‘*programs must be written for people to read, and only incidentally for machines to execute*’’ [43].

2. *Ortogonalidad* (*Orthogonality*). Determinadas características del lenguaje deben poderse combinar en todas las formas posibles, donde cada combinación tenga cierto significado.

3. *Naturalidad para la aplicación* (*Naturalness for the application*). La sintaxis del lenguaje debe permitir que la estructura de los programas refleje la estructura lógica subyacente de los algoritmos que implementan. Adicionalmente, el lenguaje debe suministrar estructuras de datos, operaciones e instrucciones de control apropiadas para el tipo de problemas que pueden resolverse con éste.

4. *Apoyo para la abstracción* (*Support for abstraction*). El lenguaje debe permitir la definición de nuevas estructuras de datos especificando sus atributos e implementando sus operaciones usando las características primitivas brindadas por el lenguaje, de tal forma que el desarrollador pueda usarlas en otras partes del programa conociendo únicamente sus propiedades abstractas, sin preocuparse por los detalles de implementación.

5. *Facilidad para la verificación de programas* (*Ease of program verification*). El lenguaje debe facilitar la verificación formal o informal de que los programas desempeñan correctamente su función mediante técnicas como:

   - *Formal verification*. Demostrar formalmente que los programas son correctos con respecto a su especificación a través de teoremas de corrección que definen su semántica axiomática.
   - *Model checking*. Demostrar que los programas son correctos con respecto a su especificación probando automáticamente todos los posibles estados que pueden tener los parámetros de entrada y verificando que los resultados satisfagan la postcondición.
   - *Desk checking*. Revisar manual y exhaustivamente el código fuente de los programas para garantizar que no tienen errores y que su semántica operacional coincida con la lógica del algoritmo implementado.
   - *Program testing*. Probar automáticamente los programas ejecutándolos con un conjunto de casos de entrada que satisfacen la precondición y revisando que las salidas cumplen la postcondición.

6. *Entorno de programación* (*Programming environment*). El lenguaje debe contar con un entorno de desarrollo integrado (*IDE* por sus siglas en inglés: *integrated development environment*) que facilite la implementación de programas en el lenguaje, proveyendo una implementación confiable, eficiente y bien documentada.

7. *Portabilidad de los programas* (*Portability of programs*). Los programas desarrollados en una máquina deben poderse ejecutar en cualquier otra máquina que tenga instalada la infraestructura mínima que demanda el lenguaje. El comportamiento de la ejecución no debe verse alterado por factores que dependen de las máquinas como su sistema operativo, su hardware o su software instalado.

8. *Costo de uso* (*Cost of use*). Las siguientes medidas de costo se pueden usar para evaluar un lenguaje de programación:

   - *Costo de ejecución* (*Cost of program execution*). El código ejecutable traducido no debe presentar sobrecargas considerables de tiempo adicionales a las inherentes al tiempo de procesamiento empleado por las instrucciones codificadas por el desarrollador.
   - *Costo de compilación* (*Cost of program translation*). Se deben proveer mecanismos eficientes que realicen el proceso de compilación sabiendo que puede ser invocado frecuentemente durante la etapa de desarrollo y depuración de un programa.
   - *Costo de creación, pruebas y uso* (*Cost of program creation, testing and use*). Los usuarios que usen el lenguaje deberían tardar menos resolviendo sus problemas (en comparación con aquellos que no lo usen), contabilizando el tiempo que les toma diseñar los programas, implementarlos, ejecutarlos, revisarlos, probarlos y usarlos.
   - *Costo de mantenimiento* (*Cost of program maintenance*). Los programas implementados en el lenguaje deben ser fáciles de mantener, tratando de reducir el costo total de su ciclo de vida. El mantenimiento incluye la reparación de errores descubiertos después de que el programa es puesto en uso, cambios necesarios por una actualización en el hardware o en el sistema operativo subyacente, y extensiones y mejoras que se necesiten para satisfacer nuevos requerimientos.

De acuerdo con Watt, algunos criterios técnicos y económicos que deben ser considerados cuando se esté evaluando el uso de un lenguaje de programación para un determinado fin son [40]:

1. *Escalabilidad* (*Scale*). El lenguaje debe permitir el desarrollo de proyectos de gran escala, permitiendo que los programas sean construidos desde unidades de compilación que han sido codificadas y probadas separadamente, tal vez por programadores distintos.

2. *Modularidad* (*Modularity*). El lenguaje debe permitir el desarrollo de proyectos en donde se pueda descomponer el código fuente en módulos con funciones claramente distinguibles, a través de la organización del código fuente en proyectos, paquetes y clases.

3. *Reusabilidad* (*Reusability*). El lenguaje debe permitir la reutilización de módulos a través de paquetes y librerías que encapsulen código fuente que ya haya sido probado.

4. *Portabilidad* (*Portability*). El lenguaje debe garantizar que el código fuente sea portable entre todas las plataformas para las que fue diseñado, operando uniformemente en los distintos sistemas operativos.

5. *Nivel* (*Level*). El lenguaje debe fomentar la programación en términos de abstracciones de alto nivel cercanas a su dominio de aplicación.

6. *Confiabilidad* (*Reliability*). El lenguaje debe estar diseñado de tal forma que los errores de programación puedan ser detectados y eliminados tan rápido como sea posible.

7. *Eficiencia* (*Efficiency*). El lenguaje debe contar con un compilador eficiente que traduzca el código fuente a su forma ejecutable cada vez que sea necesario, y los programas que se codifiquen con éste deben tener un bajo costo de ejecución.

8. *Legibilidad* (*Readability*). El lenguaje debe fomentar la escritura de código fuente que sea legible.

9. *Modelaje de datos* (*Data modeling*). El lenguaje debe suministrar tipos y operaciones asociadas que sean adecuadas para representar entidades en su dominio de aplicación. Si el lenguaje carece de los tipos necesarios, debe permitir a los programadores definir nuevos tipos y operaciones.

10. *Modelaje de procesos* (*Process modeling*). El lenguaje debe suministrar instrucciones de control que sean adecuadas para modelar el comportamiento de entidades en su dominio de aplicación.

11. *Disponibilidad de compiladores y de herramientas* (*Availability of compilers and tools*). El lenguaje debe contar con un compilador de buena calidad que valide la sintaxis del lenguaje, genere código ejecutable correcto y eficiente, desarrolle validaciones en tiempo de ejecución que atrapen errores no detectados en tiempo de compilación, y reporte todos los errores clara y precisamente. Además, el lenguaje debe tener un entorno de programación completo y agradable.

12. *Familiaridad* (*Familiarity*). El lenguaje debe ser de alguna manera familiar a los programadores, así no lo hayan usado antes.

4.1.2. Sintaxis

La *sintaxis* [†2] de un lenguaje de programación se preocupa por la forma de los programas [40], estableciendo una serie de reglas que indican cómo escribir programas *bien formados* (*well-formed*) en el lenguaje [41]. Particularmente,

[2] Según la Real Academia Española, la *sintaxis* es la *parte de la gramática que enseña a coordinar y unir las palabras para formar las oraciones y expresar conceptos*, o el *conjunto de reglas que definen las secuencias correctas de los elementos de un lenguaje de programación.*

define la manera en la que expresiones, comandos, declaraciones y otras sentencias pueden ser organizados para codificar programas, influenciando la forma en la que son escritos por las personas, leídos por otros desarrolladores y analizados por un computador [40].

La definición formal de la sintaxis de un lenguaje de programación usualmente se conoce con el nombre de *gramática* [41]. Las *gramáticas independientes del contexto* son notaciones para describir *lenguajes independientes del contexto* [39] y, a grandes rasgos, describen las reglas de producción de cadenas de un determinado lenguaje. Formalmente, una *gramática independiente del contexto* es una tupla de la forma $\langle N, T, S, P \rangle$ donde:

- N es un conjunto finito de *símbolos no terminales* $(N \neq \varnothing)$;
- T es un conjunto finito de *símbolos terminales* $(T \neq \varnothing)$;
- S es el *símbolo distinguido* $(S \in N)$;
- P es un conjunto de *reglas de producción* (*producciones*) que especifican las secuencias de símbolos terminales y no terminales (*tokens*) que forman construcciones admisibles en el lenguaje que está siendo definido [41]. Cada regla de producción es de la forma $\rho \rightarrow \delta$ donde $\rho \in N$ y $\delta \in (N \cup T)^*$.

Existen varias notaciones usadas para describir formalmente la gramática de los lenguajes de programación. Una de las más importantes es la notación *BNF* [44] (*Backus-Naur Form*) para definir gramáticas independientes del contexto, que suele usarse para especificar la sintaxis de lenguajes de programación, formatos de archivo, plantillas de documentos, conjuntos de instrucciones y protocolos de comunicación [44]. Una de las extensiones más ampliamente usadas del formalismo *BNF* es la notación *EBNF* [5] (*Extended Backus-Naur Form*), que tiene por lo menos dos variantes reconocidas:

- La variante definida por el estándar *ISO/IEC* 14977 [45], que está basada en la notación sugerida por Niklaus Wirth en el año de 1977 [45].
- La variante definida por la *W3C* [46] para describir el lenguaje *XML*, que está basada en la notación acostumbrada para construir expresiones regulares.

4.1.2.1. Criterios sintácticos generales

El principal objetivo de la sintaxis es proveer una notación para la comunicación de información entre el programador y el procesador del lenguaje [41]. Sin embargo, los detalles de diseño de la sintaxis deben ser escogidos a la luz de criterios secundarios que no necesariamente están relacionados con este objetivo, como los siguientes [41]:

- *Facilidad de lectura* (*Readability*). Un programa es legible si la estructura subyacente del algoritmo y de los datos representados por el programa son evidentes después de una inspección de su código fuente, ojalá siendo entendido sin necesidad de revisar documentación suministrada por separado.
- *Facilidad de escritura* (*Writeability*). El diseño de reglas sintácticas concisas y uniformes mejoran considerablemente la facilidad con la que un programa es escrito, en detrimento de la legibilidad. Las convenciones sintácticas implícitas que permiten que algunas declaraciones y operaciones queden sin especificar hacen que los programas queden más cortos y más fáciles de escribir, pero más difíciles de leer.
- *Facilidad de verificación* (*Ease of verifiability*). El proceso de construir programas correctos con respecto a su especificación es extremadamente difícil y requiere de técnicas matemáticas sofisticadas que posibiliten la verificación formal de su código fuente.
- *Facilidad de traducción* (*Ease of translation*). Los programas escritos en el lenguaje pueden ser fáciles de traducir a su forma ejecutable, lo que simplificaría la programación del compilador que procesa su código fuente.

- *Ausencia de ambigüedad* (*Lack of ambiguity*). La definición de un lenguaje debe proveer un significado único para cada construcción sintáctica que un programador pueda escribir. Se deben evitar las construcciones ambiguas pues permitirían dos o más interpretaciones distintas.

4.1.2.2. Elementos sintácticos de un lenguaje

El estilo general de la sintaxis de un lenguaje imperativo se configura a través de la escogencia de varios elementos sintácticos como los siguientes [41]:

- *Codificación de caracteres* (*Character set*). Define el estándar que establece el juego de caracteres que pueden ser usados dentro de los programas.

- *Identificadores* (*Identifiers*). La sintaxis básica para identificadores que consiste de una cadena de texto compuesta por letras y dígitos comenzando con una letra es ampliamente utilizada, aunque existen algunas variaciones que incluyen algunos caracteres para mejorar la legibilidad.

- *Símbolos de operador* (*Operator symbols*). Las operaciones primitivas lógico-aritméticas pueden ser representadas completamente con caracteres especiales (e.g., $+$ y $-$), o con identificadores alfanuméricos (e.g., `PLUS` y `TIMES`). Algunos lenguajes combinan ambos, utilizando cadenas de texto compuestas por caracteres especiales para algunos operadores e identificadores alfanuméricos para otros.

- *Palabras clave* (*Keywords*). Una *palabra clave* (*keyword*) es un identificador usado como una parte fija de la sintaxis de una instrucción (e.g., `if`, `do` y `while`). Muchos comandos suelen comenzar con una palabra clave sugestiva que tenga alguna relación con su descripción.

- *Palabras reservadas* (*Reserved words*). Una *palabra reservada* (*reserved word*) es una palabra clave que no puede ser declarada ni usada por los programadores como un identificador. El uso de palabras reservadas facilita el análisis sintáctico de un lenguaje y la detección de errores de compilación.

- *Palabras irrelevantes* (*Noise words*). Una palabra irrelevante (*noise word*) es una palabra opcional que es insertada en las sentencias para mejorar su legibilidad.

- *Comentarios* (*Comments*). La inclusión de comentarios en un programa es una parte importante de su documentación. Algunos lenguajes de programación permiten la inserción de comentarios delimitándolos a través de marcadores especiales (e.g., `/*` $\cdots$ `*/`), o encerrándolos entre una secuencia especial de caracteres (e.g., `//` $\cdots$) y el siguiente cambio de línea.

- *Blancos* (*Blanks (spaces)*). Normalmente, caracteres ocultos como los espacios, las tabulaciones, los retornos de carro y los cambios de línea no tienen ninguna semántica, salvo dentro de los literales que corresponden a cadenas de texto o caracteres. No obstante, los espacios en blanco suelen ser importantes para separar parejas de identificadores que de estar pegados representarían una entidad distinta.

- *Delimitadores y paréntesis* (*Delimiters and brackets*). Un *delimitador* (*delimiter*) es un elemento sintáctico usado para marcar el comienzo o el fin de alguna unidad sintáctica como una sentencia o una expresión (e.g., `,` y `:`). Por otro lado, los *paréntesis* (*brackets*) son delimitadores que vienen de a parejas (e.g., `(`$\cdots$`)` y `begin` $\cdots$ `end`). Aunque los delimitadores pueden ser usados para mejorar la legibilidad o simplificar el análisis sintáctico, frecuentemente sirven para eliminar ambigüedades definiendo explícitamente los límites de una construcción sintáctica particular.

- *Expresiones* (*Expressions*). Las expresiones son aplicaciones de función que acceden a los datos de un programa retornando algún valor, constituyendo los elementos sintácticos básicos usados para construir las sentencias de un programa.

- *Sentencias* (*Statements*). Las sentencias son el componente sintáctico más destacado en los lenguajes imperativos pues representan sus instrucciones o comandos. Las *sentencias simples* no contienen sentencias embebidas, mientras que las *sentencias anidadas* están compuestas por otras sentencias más sencillas.

4.1.3. Semántica

La sintaxis por sí sola no es suficiente para definir un lenguaje porque no brinda un mecanismo que dote a los programas de un significado que indique cómo entenderlos. La *semántica* [†3] de un lenguaje de programación se preocupa por el significado de los programas, determinando la manera en la que son armados por el programador, entendidos por otros desarrolladores e interpretados por un computador [40]. Típicamente, la semántica de un lenguaje describe cómo se espera que se comporte un programa bien formado cuando es ejecutado sobre una máquina [40] (i.e., la *semántica operacional*), pero también puede ofrecerse otro tipo de significado más formal que sea independiente de la arquitectura de los computadores (e.g., la *semántica axiomática* y la *semántica denotacional*). Usualmente, la semántica operacional se define sobre una *máquina abstracta*, que es un modelo teórico que abstrae el software o el hardware de los computadores [47].

4.1.3.1. Semántica axiomática

La *semántica axiomática* establece el significado de los programas mediante axiomas formales que describen bajo qué condiciones lógicas éstos son correctos con respecto a su especificación. Tales axiomas enuncian *teoremas de corrección* que se aplican para demostrar formalmente que un determinado programa es correcto con respecto a su especificación, usando como herramientas el cálculo proposicional y el cálculo de predicados.

La *verificación de algoritmos* se encarga del estudio de las técnicas necesarias para la demostración formal de la corrección de programas. Informalmente, se dice que un programa S es *correcto* con respecto a la precondición Q y a la postcondición R si y sólo si para todo estado que satisface Q, después de ejecutar el programa S, éste termina (en tiempo finito) en un estado que satisface R.

4.1.3.2. Semántica denotacional

La *semántica denotacional* establece el significado de los programas mediante un modelo formal [41] que define *funciones denotacionales* para asignar a cada elemento sintáctico del lenguaje un objeto específico perteneciente a cierto dominio de valores. Por ejemplo, usando el cálculo lambda, se puede definir formalmente el efecto que tienen los programas sobre un intérprete de alto nivel que actúa en un computador virtual [41].

Al igual que la semántica axiomática, la semántica denotacional ofrece un mecanismo basado en fundamentos matemáticos que es independiente de la máquina donde se ejecuten los algoritmos.

4.1.3.3. Semántica operacional

La *semántica operacional* establece el significado de los programas imperativos describiendo su operación en términos de cómo éstos se ejecutan en una máquina abstracta instrucción tras instrucción, transformando el estado de las variables durante su ejecución. La semántica de cada instrucción del lenguaje se puede definir informalmente a través de diagramas de flujo o más formalmente mediante funciones denotacionales expresadas en términos de transformaciones sobre el estado de una computación [41].

[3] Según la Real Academia Española, la *semántica* es el *estudio del significado de los signos lingüísticos y de sus combinaciones*.

4.1.4. Compilación

El proceso de traducción de un programa desde su sintaxis original hasta su forma final (posiblemente código ejecutable) es de vital importancia en la implementación de todo lenguaje de programación [41]. Los lenguajes pueden ser implementados a través de un proceso de *interpretación* que ejecuta secuencialmente las instrucciones de un programa sobre una máquina abstracta desarrollada en algún otro lenguaje de alto nivel. Sin embargo, es común que los programas sean sometidos a un proceso de *compilación* que traduzca los programas en *código ejecutable* [41], que puede estar escrito en *lenguaje de máquina* (que es directamente interpretado por el hardware del computador) o en algún otro lenguaje intermediario usado como anfitrión.

El proceso de traducción de un programa puede ser dividido lógicamente en dos grandes etapas [41]:

1. el *análisis* de su código fuente (*source code*); y
2. la *síntesis* de su código ejecutable (*executable code*), denominado también *código objeto* (*object code*).

En muchos traductores estas etapas lógicas no se encuentran claramente separadas sino que están mezcladas de tal forma que el análisis y la síntesis se alternan entre sí a medida que se procesan las instrucciones del programa [41].

4.1.4.1. Análisis del código fuente

La etapa de *análisis* recibe como entrada una secuencia de caracteres representando el código fuente de un programa y entrega como resultado una secuencia de bytes representando una versión preliminar del código ejecutable correspondiente (i.e., el código objeto). El *análisis* involucra la ejecución de los siguientes componentes [41]:

- *Analizador léxico* (*Lexer*). Es el componente que se responsabiliza de transformar la secuencia de caracteres que representa el código fuente de un programa en una secuencia de elementos sintácticos denominados *tokens*. Luego del análisis léxico (*lexing*) cada uno de los *tokens* generados debe corresponder unívocamente con algún símbolo terminal definido en la gramática del lenguaje.
- *Analizador sintáctico* (*Parser*). Es el componente que se responsabiliza de transformar la secuencia de *tokens* generada por el analizador léxico (*lexer*) en un árbol de derivación cuya estructura está determinada por las reglas de producción de la gramática del lenguaje. Luego del análisis sintáctico (*parsing*), se puede alimentar un *modelo semántico* que represente los elementos sintácticos.
- *Analizador semántico* (*Semantic analyzer*). Es el componente que se responsabiliza de procesar las estructuras sintácticas reconocidas por el *parser* navegando el modelo semántico para desarrollar una traducción preliminar a código ejecutable. Comúnmente, el analizador semántico se divide en rutinas especializadas en manejar cada uno de los diferentes tipos de *token*.

Es común que la síntesis del código ejecutable se realice durante el análisis semántico, produciendo de una vez el código objeto definitivo. De la misma forma, el análisis sintáctico usualmente se alterna con el análisis semántico comunicándose elementos sintácticos a través de una pila [41].

4.1.4.2. Síntesis del código ejecutable

La etapa de *síntesis* recibe como entrada la versión preliminar del código ejecutable generado por el analizador semántico y entrega como resultado la versión definitiva del código objeto, listo para ser ejecutado. La *síntesis* puede incluir los siguientes subprocesos [41]:

- *Optimización* (*Optimization*). El analizador semántico normalmente produce como salida una especie de *código intermedio* que debe ser procesado para generar código ejecutable susceptible de ser optimizado mediante el uso de técnicas sofisticadas.

- *Generación de código* (*Code generation*). Después de que el programa traducido ha sido optimizado, éste debe ser convertido en código objeto escrito en el lenguaje que se haya escogido para la salida del proceso de traducción (que puede ser *lenguaje ensamblador*, *lenguaje de máquina* o cualquier otro lenguaje).

4.1.5. Conceptos básicos

Esta sección enumera una serie de conceptos básicos fundamentales para el estudio, diseño e implementación de los lenguajes de propósito general.

4.1.5.1. Valores y tipos

Un *valor* es una entidad que puede ser manipulada por un programa [40]. Los valores pueden ser evaluados, almacenados, pasados como argumento, retornados como el resultado de funciones, etcétera [40]. Por ejemplo, *Java* ofrece valores booleanos, números enteros, números reales, arreglos y objetos, donde los tres primeros corresponden a *valores primitivos* y los dos últimos corresponden a *valores compuestos* [40].

Un *tipo* es un conjunto de valores equipado con operaciones que se comportan de manera uniforme sobre éstos [40]. Siempre que se tenga un valor v de tipo T se puede usar la notación $v \in T$, y siempre que se diga que una expresión E sea de tipo T se está afirmando que el resultado de evaluar E en cualquier estado siempre es un valor de tipo T [40].

Un *valor primitivo* es un valor que no puede ser descompuesto en valores más simples y un *tipo primitivo* es un tipo cuyos valores son primitivos [40]. Por otro lado, un *valor compuesto* (o *estructura de datos*) es un valor que se compone de otros valores más simples y un *tipo compuesto* es un tipo cuyos valores son compuestos [40]. Por ejemplo, *Java* cuenta con ocho tipos primitivos (véase la tabla 8.9) y con una cantidad ilimitada de tipos compuestos que pueden ser definidos a través de *clases*, cuyos valores compuestos se denominan *objetos*.

Algunos tipos compuestos importantes son [40]:

- *Arreglos*. Un *arreglo* es una secuencia indexada de elementos, cuyo rango de índices suele ser el subconjunto de los números naturales que va desde cero hasta el tamaño del arreglo menos uno.
- *Clases*. Un *objeto* de una *clase* puede ser visto como un valor compuesto por dos elementos: una tupla de componentes que almacena los valores de sus atributos y una etiqueta que identifica la clase de la que es instancia.
- *Cadenas de texto*. Una *cadena de texto* (*string*) es una secuencia finita de caracteres. En algunos lenguajes de programación como *Java* las cadenas de texto son tratadas como objetos que tienen como atributo un arreglo de caracteres.
- *Tipos recursivos*. Un *tipo recursivo* es un tipo definido en términos de sí mismo.

Los tipos recursivos poseen valores que están compuestos por otros valores del mismo tipo [40]. Declarando tipos compuestos y tipos recursivos se puede definir una gran variedad de *estructuras de datos* que permiten almacenar y organizar adecuadamente los datos para facilitar su administración y manipulación, como las siguientes:

- *Tuplas* (*tuples*). Una *n-tupla* (*tuple*) es una secuencia finita y ordenada de n valores del mismo tipo donde importa el orden en el que se encuentran sus elementos y n es un número natural constante. Se prohíbe la inserción y eliminación de valores (por ende, su tamaño es fijo).
- *Listas* (*lists*). Una *lista* (*list*) es una secuencia finita y ordenada de valores del mismo tipo donde importa el orden en el que se encuentran sus elementos. Se permite la inserción y eliminación de valores en cualquier posición (por ende, su tamaño es variable).

- *Conjuntos* (*sets*). Un *conjunto* (*set*) es una colección finita de valores del mismo tipo donde no importa el orden en el que se encuentran sus elementos y no hay repeticiones. A diferencia de una lista, en un conjunto los valores aparecen a lo sumo una vez y no tienen asociada una posición.

- *Bolsas* (*bags*). Una *bolsa* (*bag*), también llamada *multiconjunto* (*multiset*), es una colección finita de valores del mismo tipo donde no importa el orden en el que se encuentran sus elementos pero sí importan las repeticiones. A diferencia de un conjunto, en una bolsa los valores pueden aparecer cualquier cantidad de veces.

- *Pilas* (*stacks*). Una *pila* (*stack*) es una lista que únicamente permite inserciones y eliminaciones sobre uno de sus extremos (llamado *tope*).

- *Colas* (*queues*). Una *cola* (*queue*) es una lista que únicamente permite inserciones sobre un extremo (llamado *cola*) y eliminaciones sobre el otro extremo (llamado *cabeza*).

- *Bicolas* (*deques*). Una *bicola* (*deque*) es una lista que únicamente permite inserciones y eliminaciones sobre ambos extremos (llamados *principio* y *fin*).

- *Montones* (*heaps*). Un *montón* (*heap*), también llamado *montículo*, es una estructura de datos arbórea (típicamente un árbol binario) donde todo nodo distinto de su raíz tiene un valor menor o igual que el valor de su padre [1], dada una cierta relación de orden preestablecida.

- *Conjuntos disyuntos* (*disjoint-sets*). Una *estructura de datos de conjuntos disyuntos* (*disjoint-set data structure*) administra un agrupamiento de elementos particionados en una colección de subconjuntos disyuntos [48], que permite unir subconjuntos diferentes y determinar si dos elementos pertenecen al mismo subconjunto [1].

- *Árboles binarios* (*binary trees*). Un *árbol binario* (*binary tree*) es una estructura de datos jerárquica en la que cada nodo tiene máximo dos hijos. Se puede definir recursivamente diciendo que un *árbol binario* es un *árbol vacío* o una tupla compuesta por un valor (llamado *raíz*) y por dos árboles binarios (llamados *subárbol izquierdo* y *subárbol derecho*).

- *Árboles enearios* (*n-ary trees*). Un *árbol eneario* (*n-ary tree*) es una estructura de datos jerárquica en la que cada nodo puede tener cualquier cantidad finita de hijos. Se puede definir recursivamente diciendo que un *árbol eneario* es un *árbol vacío* o una tupla compuesta por un valor (llamado *raíz*) y por cualquier cantidad finita de árboles enearios (llamados *subárboles*).

- *Asociaciones llave-valor* (*maps*). Una *asociación llave-valor* (*map*), también llamada *arreglo asociativo* (*associative array*), *mapa* (*map*) o *diccionario* (*dictionary*), es una estructura de datos compuesta por un conjunto finito de llaves únicas y por una colección de valores, donde cada llave tiene asociado exactamente un valor (aunque es posible que llaves distintas tengan asociado el mismo valor). Sirve para modelar funciones discretas cuyo dominio es el conjunto de llaves y cuya imagen es el conjunto de valores.

- *Asociaciones llave-valores* (*multimaps*). Una *asociación llave-valores* (*multimap*) es una asociación llave-valor donde a cada llave se le puede asociar uno o más valores (i.e., una lista de valores).

- *Grafos* (*graphs*). Un *grafo* (*graph*) es una estructura de datos compuesta por un conjunto de nodos V y por un conjunto de arcos E que conectan pares de nodos entre sí ($E \subseteq V \times V$).

- *Multigrafos* (*multigraphs*). Un *multigrafo* (*multigraph*) es una estructura de datos compuesta por un conjunto de nodos V y por una bolsa de arcos E que conectan pares de nodos entre sí ($E : V \times V \to \mathbb{N}$).

- *Hipergrafos* (*hypergraphs*). Un *hipergrafo* (*hypergraph*) es una estructura de datos compuesta por un conjunto de nodos V y por un conjunto de hiper-arcos E que agrupan conjuntos de nodos entre sí ($E \subseteq \wp(V) \backslash \{\varnothing\}$).

- *Autómatas finitos determinísticos* (*deterministic finite automata*). Un *autómata finito determinístico* (*deterministic finite automaton*) es una 5-tupla $\langle Q, \Sigma, q_I, F, \delta \rangle$ donde Q es un conjunto finito de *estados* ($Q \neq \varnothing$), Σ es un conjunto finito de símbolos llamado *alfabeto* ($\Sigma \neq \varnothing$), q_I es el *estado inicial* ($q_I \in Q$), F es el conjunto de *estados finales* ($F \subseteq Q$), y δ es la *función de transición de estados* ($\delta : Q \times \Sigma \to Q$).

- *Autómatas finitos no determinísticos* (*nondeterministic finite automata*). Un *autómata finito no determinístico* (*nondeterministic finite automaton*) es una 5-tupla $\langle Q, \Sigma, q_I, F, \Delta \rangle$ donde Q es un conjunto finito de *estados* ($Q \neq \varnothing$), Σ es un conjunto finito de símbolos llamado *alfabeto* ($\Sigma \neq \varnothing$), q_I es el *estado inicial* ($q_I \in Q$), F es el conjunto de *estados finales* ($F \subseteq Q$), y Δ es la *función de transición de estados* ($\Delta : Q \times \Sigma \to \wp(Q)$).

- *Autómatas con respuesta* (*deterministic finite transducers*). Un *autómata con respuesta* en las transiciones y/o en los estados (*deterministic finite transducer*) es una 7-tupla $\langle Q, \Sigma, \Sigma', q_I, \delta, g, h \rangle$ donde Q es un conjunto finito de *estados* ($Q \neq \varnothing$), Σ es un conjunto finito de símbolos llamado *alfabeto de entrada* ($\Sigma \neq \varnothing$), Σ' es un conjunto finito de símbolos llamado *alfabeto de salida* ($\Sigma' \neq \varnothing$), q_I es el *estado inicial* ($q_I \in Q$), δ es la *función de transición de estados* ($\delta : Q \times \Sigma \to Q$), g es la *función de salida en los estados* ($g : Q \to (\Sigma')^*$), y h es la *función de salida en las transiciones* ($h : Q \times \Sigma \to (\Sigma')^*$).

- *Autómatas de pila* (*pushdown automata*). Un *autómata de pila* (*pushdown automaton*) es una 6-tupla $\langle Q, \Sigma, \Gamma, q_I, F, \Delta \rangle$ donde Q es un conjunto finito de *estados* ($Q \neq \varnothing$), Σ es un conjunto finito de símbolos llamado *alfabeto de entrada* ($\Sigma \neq \varnothing$), Γ es un conjunto finito de símbolos llamado *alfabeto de pila*, q_I es el *estado inicial* ($q_I \in Q$), F es el conjunto de *estados finales* ($F \subseteq Q$), y Δ es la *relación de transición de estados* ($\Delta \subseteq (Q \times \Gamma^* \times \Sigma^*) \times (Q \times \Gamma^*)$).

El *sistema de tipado* (*type system*) de un lenguaje de programación se responsabiliza de agrupar valores en tipos [40]. Antes de realizar cualquier operación, los tipos de sus operandos deben ser revisados para detectar un posible *error de tipos*, que podría conllevar una excepción en tiempo de ejecución [40]. En un lenguaje *estáticamente tipado* (*statically typed*) cada variable y cada expresión tiene un tipo fijo que es explícitamente declarado por el programador o automáticamente inferido por el compilador, permitiendo que el tipo de los operandos pueda ser revisado en tiempo de compilación [40]. En contraparte, en un lenguaje *dinámicamente tipado* (*dynamically typed*) los valores tienen tipos fijos pero las variables y expresiones no, obligando a que el tipo de los operandos deba ser revisado en tiempo de ejecución justo después de que éstos son evaluados, pues cada vez que un operando es calculado podría arrojar un valor de un tipo distinto [40].

4.1.5.2. Expresiones y evaluación

Una *expresión* es una construcción que puede ser *evaluada* para entregar un valor [40]. Las expresiones pueden ser formadas de varias maneras [40] (los ejemplos del numeral 5 están dados en el lenguaje matemático introducido en el libro de Gries y Schneider [12], y el resto de ejemplos están dados en el lenguaje *Java*):

1. *Literales* (*Literals*). Un *literal* es una expresión que denota un valor fijo de algún tipo (e.g., `17`, `3.141592`, `false`, `'A'`, `"Hello World"`).

2. *Accesos a variable* (*Variable accesses*). Un *acceso a variable* es una referencia a una variable previamente declarada, entregando el valor actual de tal variable.

3. *Construcciones* (*Constructions*). Una *construcción* es una expresión que crea un valor compuesto a partir de los valores que lo conforman. Bajo el paradigma de la programación orientada a objetos, una *construcción* es un llamado a una operación llamada *constructor*, que crea un nuevo objeto de una clase en particular (e.g., `new int[]{51,33,21,84}`, `new int[]{{93,81},34,{{}},{{71},12}}`, `new double[]{3.1,9.4,0.9}`, `new java.util.Date(System.currentTimeMillis())`).

4. *Llamados a función* (*Function calls*). Un *llamado a función* (*function call*) calcula el resultado de aplicar una función (o método) sobre algunos argumentos (e.g., `Math.random()`, `Math.abs(-7)`, `Math.min(7.3,2.1)`). La aplicación de un operador puede ser tratada como un llamado a función (e.g., `-5`, `4.3+2.5`, `!true`, `true&&false`).

5. *Expresiones condicionales* (*Conditional expressions*). Una *expresión condicional* calcula un valor que depende de una condición. Dados `B` una expresión booleana y `E`,`F` dos expresiones del mismo tipo, en algunos lenguajes de programación se ofrecen expresiones condicionales de la forma `B?E:F` que entregan el valor de la expresión `E` si la guarda `B` es verdadera, o el valor de la expresión `F` de lo contrario (e.g., `Math.random()<0.5?51:92`).

6. *Expresiones iterativas* (*Iterative expressions*). Una *expresión iterativa* es una expresión que realiza un cálculo sobre una serie de valores (típicamente los componentes de un arreglo o de una lista), entregando algún resultado. La descripción de conjuntos por comprensión y las cuantificaciones pueden ser consideradas como expresiones iterativas (e.g., `{x|0<=x<=5}`, `(∀x|x<=5:x*2<=10)`).

Las *reglas de precedencia* entre operadores son convenciones preestablecidas que reducen la necesidad de uso de paréntesis para facilitar la escritura y manipulación de las expresiones. Estas reglas deben ser tenidas en cuenta al momento de evaluar cualquier expresión. Por ejemplo, sabiendo que la multiplicación y la división tienen mayor precedencia que la suma y la resta, se tiene que $5+3*8<64/2-2$ abrevia la expresión $(5+(3*8))<((64/2)-2)$ en vez de expresiones como $((5+3)*8)<(64/(2-2))$ y $5+3*(8<64)/2-2$.

Similarmente, las reglas de *asociatividad* (*associativity*) de los operadores también influyen en la evaluación de las expresiones:

- *Asociatividad por la izquierda*. Un operador binario $\star$ es *asociativo por la izquierda* si $a\star b\star c$ denota la expresión $(a\star b)\star c$ para todos los valores de a, b y c. Por ejemplo, la sustracción $(-)$ es asociativa por la izquierda porque $a-b-c$ denota $(a-b)-c$ pero no $a-(b-c)$.

- *Asociatividad por la derecha*. Un operador binario $\star$ es *asociativo por la derecha* si $a\star b\star c$ denota la expresión $a\star(b\star c)$ para todos los valores de a, b y c. Por ejemplo, la implicación $(\Rightarrow)$ es asociativa por la derecha porque $a\Rightarrow b\Rightarrow c$ denota $a\Rightarrow(b\Rightarrow c)$ pero no $(a\Rightarrow b)\Rightarrow c$.

- *Asociatividad*. Un operador binario $\star$ es *asociativo* si $((a\star b)\star c)=(a\star(b\star c))$ para todos los valores de a, b y c. Por ejemplo, la adición $(+)$ es asociativa porque $(a+b)+c=a+(b+c)$.

Para reducir aún más el uso de paréntesis se suelen definir algunas reglas especiales de *asociatividad mutua* (*mutual associativity*):

- *Asociatividad mutua por la izquierda*. Dos operadores binarios $\star$ y $\bullet$ son *mutuamente asociativos por la izquierda* si $a\star b\bullet c$ denota la expresión $(a\star b)\bullet c$, y $a\bullet b\star c$ denota la expresión $(a\bullet b)\star c$ para todos los valores de a, b y c. Por ejemplo, la adición $(+)$ y la sustracción $(-)$ son mutuamente asociativas por la izquierda porque $a+b-c$ denota $(a+b)-c$, y $a-b+c$ denota $(a-b)+c$.

- *Asociatividad mutua por la derecha*. Dos operadores binarios $\star$ y $\bullet$ son *mutuamente asociativos por la derecha* si $a\star b\bullet c$ denota la expresión $a\star(b\bullet c)$, y $a\bullet b\star c$ denota la expresión $a\bullet(b\star c)$ para todos los valores de a, b y c. Por ejemplo, la implicación $(\Rightarrow)$ y la anti-implicación $(\not\Rightarrow)$ son mutuamente asociativas por la derecha porque $a\Rightarrow b\not\Rightarrow c$ denota $a\Rightarrow(b\not\Rightarrow c)$, y $a\not\Rightarrow b\Rightarrow c$ denota $a\not\Rightarrow(b\Rightarrow c)$.

- *Asociatividad mutua*. Dos operadores binarios $\star$ y $\bullet$ son *mutuamente asociativos* si ambos son conmutativos y $((a\star b)\bullet c)=(a\star(b\bullet c))$ para todos los valores de a, b y c. Por ejemplo, la inequivalencia $(\not\equiv)$ y la equivalencia $(\equiv)$ son mutuamente asociativas porque ambas operaciones son conmutativas y $(a\not\equiv b)\equiv c$ da el mismo valor que $a\not\equiv(b\equiv c)$ sin importar el valor de las variables proposicionales a, b y c.

Además, un operador puede ser *conjuncional* (*conjunctional*), de acuerdo con la terminología del libro de Gries y Schneider [12]:

- *Conjuncionalidad*. Un operador binario $\star$ es *conjuncional* si $a \star b \star c$ abrevia la expresión $(a \star b) \wedge (b \star c)$ para todos los valores de a, b y c. Por ejemplo, el menor que ($<$) es conjuncional porque $a < b < c$ abrevia $(a < b) \wedge (b < c)$.

- *Conjuncionalidad mutua*. Dos operadores binarios $\star$ y $\bullet$ son *mutuamente conjuncionales* si ambos son conjuncionales, $a \star b \bullet c$ abrevia la expresión $(a \star b) \wedge (b \bullet c)$, y $a \bullet b \star c$ abrevia la expresión $(a \bullet b) \wedge (b \star c)$ para todos los valores de a, b y c. Por ejemplo, la igualdad ($=$) y el menor que ($<$) son mutuamente conjuncionales porque ambos son conjuncionales, $a = b < c$ abrevia $(a = b) \wedge (b < c)$, y $a < b = c$ abrevia $(a < b) \wedge (b = c)$.

Las expresiones booleanas se pueden evaluar por *cortocircuito* (*short-circuit evaluation*) omitiendo la evaluación de la segunda mitad de una operación cuando el valor total de la expresión puede ser determinado a partir de la evaluación de la primera mitad [42]. Por ejemplo, siendo P y Q dos expresiones booleanas, se puede evadir la evaluación de la expresión Q en las siguientes situaciones:

- cuando se tiene la conjunción $P \wedge Q$ sabiendo que la expresión P es falsa (el resultado es falso);

- cuando se tiene la disyunción $P \vee Q$ sabiendo que la expresión P es verdadera (el resultado es verdadero);

- cuando se tiene la implicación $P \Rightarrow Q$ sabiendo que la expresión P es falsa (el resultado es verdadero); y

- cuando se tiene la anti-consecuencia $P \not\Leftarrow Q$ sabiendo que la expresión P es verdadera (el resultado es falso).

4.1.5.3. Variables y almacenamiento

En los lenguajes de programación imperativos, una *variable* es un contenedor de un valor, que puede ser inspeccionado y actualizado cada vez que se desee [40].

Un depósito (*store*) es una colección de celdas de almacenamiento (*storage cells*), donde cada celda tiene una dirección única (*address*) y un estado actual (*current status*) que indica si está asignada (*allocated*) o si no está asignada (*unallocated*) [40]. Las celdas de almacenamiento que han sido asignadas tienen un contenido actual (*current content*) que puede ser un *valor almacenable* (*storable value*) o el valor *indefinido* (*undefined*) [40]. En términos de este modelo de almacenamiento, cada variable puede ser vista como un contenedor que consiste de una o más celdas de almacenamiento asignadas [40].

Una *variable simple* es una variable que puede contener un valor almacenable y que ocupa una sola celda de almacenamiento [40]. Por otro lado, una *variable compuesta* es una variable de un tipo compuesto que ocupa un grupo de celdas de almacenamiento ubicadas en direcciones contiguas [40]. De esta manera, el valor de las variables termina siendo almacenado en las celdas de almacenamiento que tenga asignadas.

Cuando un programa ejecuta una operación que asigna un determinado valor a una variable del mismo tipo, lo que suceda depende del lenguaje [40]:

- *Copia por valor*. La asignación copia todos los componentes del valor en los componentes correspondientes de la variable.

- *Copia por referencia*. La asignación hace que la variable termine conteniendo un apuntador (o referencia) al valor.

Las asignaciones en *C* y *C++* actúan copiando por valor, aunque se puede lograr el efecto de copiar por referencia usando apuntadores explícitamente [40]. Por otro lado, *Java* adopta la copia por valor para los valores primitivos y

la copia por referencia para los objetos, aunque se puede lograr el efecto de copiar por valor usando el método `clone` de los objetos [40].

Toda variable es creada (o registrada en el depósito) en un determinado momento y posteriormente destruida (o eliminada del depósito) cuando ya no se necesite más [40]. El *tiempo de vida* (*lifetime*) de una variable es el intervalo transcurrido entre su creación y su destrucción [40]. Una variable sólo necesita ocupar celdas de almacenamiento del depósito durante su tiempo de vida, que son asignadas cuando la variable es creada, y posteriormente liberadas cuando la variable es destruida [40]. Después de la destrucción de una variable, las celdas de almacenamiento que fueron liberadas pueden ser reasignadas a variables que sean creadas subsecuentemente [40].

Un *bloque* (*block*) es una parte del programa que puede incluir declaraciones locales, como el cuerpo de los procedimientos [40]. En lenguajes como *Java*, *C* y *C++* los bloques son los fragmentos de código que se encuentran delimitados entre llaves ($\{\cdots\}$). Una *activación* (*activation*) de un bloque es un intervalo de tiempo durante el cual está siendo ejecutado [40]. En particular, una activación de un procedimiento sería el intervalo de tiempo que transcurre entre su invocación y su retorno [40]. Como durante la ejecución de un programa un determinado bloque puede ser activado durante muchas ocasiones, las variables locales que éste declare serían creadas y destruidas una y otra vez [40].

Una *variable global* (*global variable*) es una variable cuyo tiempo de vida es todo el tiempo de ejecución del programa: la variable se crea cuando el programa comienza su ejecución y se destruye cuando éste termina [40]. En contraparte, una *variable local* es una variable cuyo tiempo de vida es una activación del bloque que contiene su declaración: la variable se crea cuando el flujo de ejecución entre al bloque y se destruye cuando éste salga del bloque [40]. En otras palabras, las variables globales son declaradas para ser usadas por todas partes del programa mientras que las variables locales son declaradas dentro de un bloque específico para ser usadas únicamente dentro de ese bloque [40]. Observe que una variable local tendría varios ciclos de vida si el bloque que la declara es activado varias veces, sin poder retener su valor contenido entre activaciones consecutivas [40]. En los lenguajes orientados a objetos, las variables globales se conocen como *variables estáticas* (*static variables*) o *variables de clase* (*class variables*) [40].

Un *apuntador* (*pointer*) es una referencia a una variable particular y un *apuntador nulo* (*null pointer*) se usa para denotar un apuntador que no referencia ninguna variable [40]. En términos del modelo de almacenamiento, un apuntador es esencialmente la dirección que tiene la variable referenciada dentro del depósito [40]. En *Java* existe un proceso especial denominado *recolector de basura* (*garbage collector*) que se ejecuta automáticamente de vez en cuando para liberar las celdas de almacenamiento ocupadas por aquellos objetos que se han dejado de referenciar desde el programa.

4.1.5.4. Comandos y control de flujo

En los lenguajes de programación imperativos, un *comando* (*command*) es una instrucción que puede ser ejecutada para actualizar el valor de las variables [40]. Los *comandos primitivos* son comandos simples que no están formados por otros comandos, mientras que los *comandos compuestos* son comandos que pueden ser descompuestos en otros comandos más sencillos [40]. Los comandos pueden ser construidos de muchas maneras [40]:

- *Instrucciones vacías* (*Skips*). Una *instrucción vacía* (*skip*) es una instrucción que no tiene ningún efecto sobre el valor de las variables.

- *Llamados a procedimiento* (*Procedure calls*). Un *llamado a procedimiento* (*procedure call*), típicamente escrito en la forma $P(E_1, E_2, \ldots, E_n)$ donde P es el nombre de un procedimiento y $E_1, E_2, \ldots, E_n$ son expresiones, es una instrucción que logra su efecto invocando un procedimiento (o método) P sobre los argumentos $E_1, E_2, \ldots, E_n$ (después de ser evaluados).

- *Asignaciones* (*Assignments*). Una *asignación* (*assignment*), típicamente escrita en la forma $v = E$ (o $v := E$) donde v es un acceso a variable y E es una expresión, es una instrucción que asigna a la variable v el valor de una expresión E (después de ser evaluada).

- *Asignaciones múltiples* (*Multiple assignments*). Una *asignación múltiple* (*multiple assignment*), típicamente escrita en la forma $v_1 = v_2 = \cdots = v_n = E$, es una asignación que causa que el valor de la expresión sea asignado a múltiples variables.

- *Asignaciones simultáneas* (*Simultaneous assignments*). Una *asignación simultánea* (*simultaneous assignment*), típicamente escrita en la forma $v_1, v_2, \ldots, v_n := E_1, E_2, \ldots, E_n$, es una asignación que evalúa simultáneamente el valor de cada una de las expresiones del lado derecho para asignarle tales valores a las variables correspondientes en el lado izquierdo (v_i queda con el valor de E_i para cada i de 1 a n).

- *Comandos secuenciales* (*Sequential commands*). Un *comando secuencial* (*sequential command*) es una instrucción que ejecuta en secuencia dos o más comandos más simples justo en el orden en el que aparecen (este orden es relevante). Un comando secuencial puede ser escrito en la forma $C_1; C_2; \cdots; C_n$, indicando que primero se ejecuta el comando C_1, luego el comando C_2 y así sucesivamente hasta el comando C_n.

- *Comandos colaterales* (*Collateral commands*). Un *comando colateral* (*collateral command*) es una instrucción que ejecuta en cualquier orden dos o más comandos más simples (el orden es irrelevante). Un comando colateral puede ser escrito en la forma $C_1, C_2, \ldots, C_n$, indicando que las respectivas instrucciones pueden ser ejecutadas en cualquier orden.

- *Comandos condicionales* (*Conditional commands*). Un *comando condicional* (*conditional command*) es una instrucción compuesta por dos o más subcomandos más simples donde exactamente uno de estos es escogido para ser ejecutado, dependiendo de una o más condiciones. Los comandos condicionales comprenden sentencias populares como *if-then*, *if-then-else* y *switch*.

- *Comandos iterativos* (*Iterative commands*). Un *comando iterativo* (*iterative command*), comúnmente conocido como *ciclo* o *bucle* (*loop*), es una instrucción que está compuesta por un subcomando (llamado *cuerpo*) que es ejecutado repetitivamente. Cada ejecución del cuerpo del bucle se denomina *iteración* (*iteration*). Los comandos iterativos comprenden sentencias populares como *while*, *do-while*, *repeat-until*, *for* y *for-each*.

Una ejecución es *determinística* (*deterministic*) si la secuencia de pasos que desarrolla es completamente predecible, y es *no determinística* (*nondeterministic*) de lo contrario [40]. Por otro lado, una ejecución es *efectivamente determinística* (*effectively deterministic*) si su salida es predecible aunque la secuencia de pasos que desarrolle sea impredecible [40].

Los comandos mencionados exhiben un control de flujo (*control flow*) con una única entrada (*single-entry*) y una única salida (*single-exit*), que es adecuado para la mayoría de propósitos prácticos [40]. Sin embargo, algunos lenguajes imperativos proveen mecanismos para alterar el control de flujo de tal forma que pueda tener múltiples salidas (*multi-exit*) [40]. Estos mecanismos se denominan *secuenciadores* (*sequencers*), que son construcciones capaces de transferir el control a algún otro punto del programa, denominado *destino* (*destination*) [40]:

- *Saltos* (*Jumps*). Son secuenciadores que transfieren el control a un punto específico del programa (e.g., `goto`).

- *Escapes* (*Escapes*). Son secuenciadores que terminan inmediatamente la ejecución del comando o procedimiento sobre el que se encuentran (e.g., `break`, `continue`, `return`).

- *Excepciones* (*Exceptions*). Son secuenciadores que pueden ser usados para reportar situaciones anormales que impiden que el programa pueda continuar con su ejecución (e.g., `throw`). Las excepciones se lanzan (*throw*) y pueden ser posteriormente atrapadas (*catch*) para su tratamiento, mediante instrucciones de la forma *try-catch*.

4.1.5.5. Ataduras (*bindings*) y alcance (*scope*)

Todo lenguaje de programación permite escribir declaraciones que atan (*bind*) identificadores a entidades como valores, variables y procedimientos [40]. El hecho de atar un identificador a una entidad en una declaración, para luego usar ese identificador con el fin de denotar la entidad en muchos otros lugares dependiendo de su alcance (*scope*), ayuda a que el programa sea fácil de modificar porque si la entidad necesita cambiarse entonces sólo su declaración debe ser modificada, no los lugares del código fuente donde es usada [40].

Una *atadura* (*binding*) es una asociación fija entre un identificador y una entidad como un valor, una variable o un procedimiento [40]. Un *entorno* (*environment*) o *espacio de nombres* (*namespace*) es un conjunto de ataduras [40]. Cada declaración produce una o más ataduras sobre el espacio de nombres, enlazando el identificador declarado con la entidad construida [40]. Cada expresión (o comando) se debe interpretar en un espacio de nombres particular, y todos los identificadores usados en la expresión (o comando) deben tener ataduras en tal espacio de nombres [40].

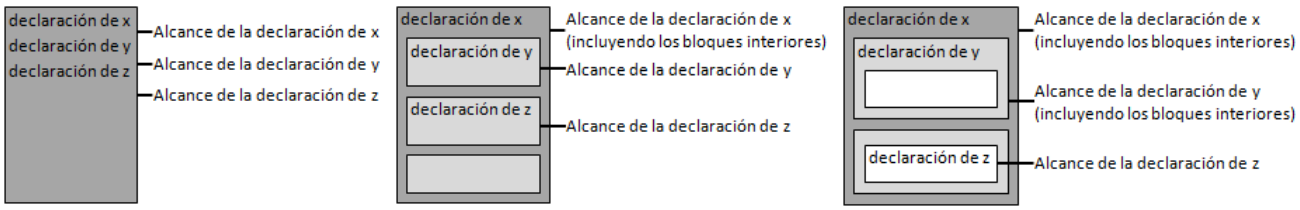
El *alcance* (*scope*) de una declaración es la región del código fuente sobre la que es efectiva y el *alcance* de una atadura es la región del código fuente sobre la que aplica [40]. Un *bloque* (*block*) es una parte del programa que delimita el alcance de cualquier declaración que sea efectuada dentro de éste [40]. Por ejemplo, los bloques de un programa *Java* son los fragmentos de código que están encerrados entre llaves ($\{\cdots\}$), los cuerpos de métodos, las declaraciones de clase y el programa como un todo [40].

La *estructura de bloques* (*block structure*) describe la forma en la que están distribuidos los bloques de un programa, influenciando el alcance de las declaraciones [40]:

- En un lenguaje con *estructura de bloques monolítica* (*monolithic block structure*) el único bloque es todo el programa.
- En un lenguaje con *estructura de bloques plana* (*flat block structure*) los programas son particionados en varios bloques que no se traslapan, teniendo un bloque global que alberga todo el programa.
- En un lenguaje con *estructura de bloques anidados* (*nested block structure*) los bloques pueden estar anidados unos dentro de otros envolviéndose entre sí.

Figura 4.4. *Distintos tipos de estructura de bloques en los lenguajes de programación [40].*

(a) *Estructura monolítica.* **(b)** *Estructura plana.* **(c)** *Estructura anidada.*



Para definir el entorno de un procedimiento cuando es invocado hay dos posibilidades [40]:

- Un lenguaje tiene *alcance estático* (*statically scoped*) si el cuerpo de un procedimiento es ejecutado en el espacio de nombres propio a la definición del procedimiento. El alcance de cada declaración se puede decidir en tiempo de compilación y sería el bloque más pequeño que la contiene, excluyendo el código fuente previo a la declaración.
- Un lenguaje tiene *alcance dinámico* (*dynamically scoped*) si el cuerpo de un procedimiento es ejecutado en el espacio de nombres de la instrucción que realizó el llamado a procedimiento. El alcance de cada declaración se debe decidir en tiempo de ejecución, dependiendo del control de flujo del programa.

Un *bloque de comando* (*block command*) es un comando que contiene declaraciones locales y subcomandos, de tal forma que las ataduras producidas por sus declaraciones pueden ser usadas únicamente para ejecutar sus subcomandos [40]. Los subcomandos de un bloque son ejecutados en un entorno que hereda las ataduras del entorno externo, adicionando las ataduras producidas por sus propias declaraciones locales [40].

Un *bloque de expresión* (*block expression*) es una expresión que contiene declaraciones locales y subexpresiones, de tal forma que las ataduras producidas por sus declaraciones pueden ser usadas únicamente para evaluar sus subexpresiones [40]. Las subexpresiones de una expresión son evaluadas en un entorno que hereda las ataduras del entorno externo, adicionando las ataduras producidas por sus propias declaraciones locales [40].

Una *declaración* (*declaration*) es una construcción para producir ataduras de identificadores a entidades [40]. Cada lenguaje de programación permite de una u otra manera la declaración de tipos, constantes, variables, procedimientos, clases y paquetes [40].

4.1.5.6. Procedimientos y funciones

Un *procedimiento* (*procedure*) es una entidad que abstrae un cálculo [40]. En particular, una *función* (*function*) es un procedimiento que abstrae una expresión a ser evaluada y un *procedimiento propio* (*proper procedure*) es un procedimiento que abstrae un comando a ser ejecutado [40]. En el paradigma de la programación orientada a objetos, una función sería un método con retorno (i.e., con retorno de tipo distinto de `void`) y un procedimiento propio sería un método sin retorno (i.e., con retorno de tipo `void`).

Cuando una función es *invocada* o *llamada* (*called*), la expresión que abstrae es evaluada y su valor es entregado como *resultado* (*result*) [40]. Por otro lado, cuando un procedimiento propio es *invocado* o *llamado* (*called*), el comando que abstrae es ejecutado causando eventualmente una actualización en las variables cuyo alcance incluye el procedimiento [40]. A pesar de que una función abstrae una expresión que debe ser evaluada, en los lenguajes imperativos cada función suele verse sintácticamente como un comando que calcula el valor de la expresión correspondiente, entregándolo como resultado mediante la ejecución de un *retorno* (*return*) [40].

Una función tiene un *cuerpo* (*body*) que es una expresión (o en su defecto, un comando que la calcula y retorna su resultado), y un *llamado a función* (*function call*) es una expresión que entrega como resultado la evaluación del cuerpo de la función [40]. Similarmente, un procedimiento propio tiene un *cuerpo* (*body*) que es un comando, y un *llamado a procedimiento* (*procedure call*) es un comando que actualiza variables ejecutando el cuerpo del procedimiento propio [40].

Un *argumento* (*argument*) es un valor (u otra entidad) que es pasado a un procedimiento y un *parámetro* (*parameter*) es un identificador a través del cual un procedimiento puede tener acceso a un argumento [40]. Cuando un procedimiento es invocado, cada parámetro es asociado con su argumento correspondiente usando alguno de los siguientes mecanismos de paso de parámetros [40]:

- *Paso por valor* (*call by value*). El *paso de parámetros por valor* (*copy parameter mechanism*) ata el parámetro a una variable local que contiene una copia del argumento. Por ejemplo, en *Java* los valores de tipo primitivo son pasados por valor.
- *Paso por referencia* (*call by reference*). El *paso de parámetros por referencia* (*reference parameter mechanism*) ata el parámetro a una variable local que contiene un apuntador al argumento mismo. Por ejemplo, en *Java* los objetos son pasados por referencia.

4.1.5.7. Módulos y paquetes

Un *módulo* (*program unit*) es cualquier parte del programa a la que se le ha asignado nombre, que puede ser diseñada e implementada con relativa independencia [40]. El *API* (*Application Programming Interface*) de un

módulo contiene la información que los programadores necesitan saber para poder usar el módulo con pericia [40]. En particular, el *API* debería describir para cada procedimiento su identificador, sus parámetros y el tipo de su resultado, junto con una especificación de su comportamiento observable [40].

Un lenguaje de programación cuyos módulos sean procedimientos es apropiado únicamente para la construcción de programas a pequeña escala [40]. Para la construcción de programas a gran escala, el lenguaje debería proveer módulos como paquetes y clases [40]. Un *paquete* (*package*) es un grupo de varios componentes declarados para un propósito común [40] y una *clase* (*class*) es la abstracción de una familia de objetos de la realidad con propiedades (*atributos*) y comportamiento (*métodos*) similares.

4.2. Lenguajes de propósito específico

Los *lenguajes de programación de propósito específico* (*DSL* por sus siglas en inglés: *domain-specific programming languages*) son lenguajes de programación cuya aplicación está limitada a algún dominio particular, al contrario de los *lenguajes de programación de propósito general* (*GPL* por sus siglas en inglés: *general-purpose programming languages*), que sirven para resolver problemas sobre una gran cantidad de dominios.

Los conceptos tratados en esta sección están completamente basados (directa o indirectamente) en las referencias:

1. *Domain-Specific Languages* [49] de Martin Fowler y Rebecca Parsons;

2. *Notable design patterns for domain-specific languages* [50] de Diomidis Spinellis; y

3. *When and how to develop domain-specific languages* [51] de Marjan Mernik, Jan Heering y Anthony M. Sloane.

4.2.1. Definición

Un *lenguaje de propósito específico* (*DSL* por sus siglas en inglés: *domain-specific language*) es un lenguaje de programación de expresividad limitada concentrado sobre un dominio particular [49]. Hay cuatro elementos clave en esta definición [49]:

- *Lenguaje de programación*. Un lenguaje de propósito específico debe tener una estructura diseñada para facilitar que los humanos lo entiendan y para permitir su implementación computacional.

- *Naturaleza del lenguaje*. Un lenguaje de propósito específico debe tener un sentido de fluidez cuya expresividad dependa de la forma en la que se compongan expresiones individuales.

- *Expresividad limitada*. Un lenguaje de propósito específico debe proveer una pequeña cantidad de características para respaldar su dominio de aplicación, no montones de características para resolver problemas en muchos dominios.

- *Enfoque de dominio*. Un lenguaje de propósito específico debe ser un lenguaje limitado que se concentre únicamente sobre un dominio pequeño, lo que hace que valga la pena.

En otras palabras, un lenguaje de propósito específico debe servir para atacar un dominio particular de un sistema, no para construir todo el sistema. Los *DSLs* ofrecen ganancias en expresividad y facilidad de uso comparados con los *GPLs*, en su dominio de aplicación [51].

4.2.2. Categorías

Los *DSLs* se pueden dividir en tres categorías principales [49]:

- *DSLs externos*. Un *DSL externo* es un lenguaje cuya sintaxis está separada del lenguaje de programación principal con el que está trabajando.

- *DSLs internos*. Un *DSL interno* es un lenguaje cuya sintaxis denota una forma particular de usar un lenguaje de propósito general.

- *Language workbenches*. Un *language workbench* es un *IDE* especializado para definir y construir *DSLs*, con el que se puede personalizar un entorno gráfico de edición para el *DSL*.

4.2.3. Tópicos generales (Fowler)

A continuación se enumeran algunos tópicos generales descritos por Fowler [49], relevantes para el diseño de *GOLD* como lenguaje de propósito específico:

1. *Modelo semántico* (*Semantic model*). Es una representación de lo que el *DSL* describe, que puede ser un modelo de objetos en memoria principal donde cada clase denota un elemento sintáctico distinto del lenguaje. En la práctica, el modelo semántico es la estructura que el *DSL* puebla una vez es procesado.

2. *Traducción basada en delimitadores* (*Delimiter-directed translation*). Es un proceso de traducción que recibe la entrada correspondiente al código fuente y la divide en fragmentos más pequeños según cierto carácter que actúa como delimitador, que comúnmente es el cambio de línea. Luego, cada fragmento es procesado para ser traducido.

3. *Traducción basada en sintaxis* (*Syntax-directed translation*). Es un proceso de traducción que recibe la entrada correspondiente al código fuente y la analiza léxico-sintácticamente para generar un árbol de sintaxis de acuerdo con las reglas de una gramática que describe cómo cada elemento del lenguaje se puede dividir en subelementos. Luego, el árbol de sintaxis es procesado para realizar la traducción.

4. *Notación BNF* (*Backus-Naur Form*). Es una forma de describir gramáticas para definir la sintaxis de un lenguaje de programación, guiando la traducción basada en sintaxis.

5. *Tabla de elementos léxicos* (*Regex table lexer*). Es una tabla que define cada uno de los elementos léxicos (*tokens*) del lenguaje a través de expresiones regulares, guiando la implementación del analizador léxico (*scanner* o *lexer*).

6. *Generador de analizadores sintácticos* (*Parser generator*). Es un proceso que genera automáticamente la implementación del analizador sintáctico (*parser*) de un lenguaje a partir de un archivo de texto que describe su gramática usando algún formalismo.

7. *Expresiones anidadas* (*Nested operator expression*). Son expresiones que pueden contener recursivamente la misma forma de expresión. Su análisis sintáctico depende de las reglas de precedencia entre los operadores.

8. *Separadores de cambios de línea* (*Newline separators*). Es un rasgo común de algunos lenguajes de programación, donde los cambios de línea marcan el fin de cada instrucción.

4.2.4. Patrones de diseño (Spinellis)

A continuación se enumeran algunos patrones descritos por Spinellis [50], relevantes para el diseño e implementación de *GOLD* como lenguaje de propósito específico:

1. *Piggyback*. Usa las capacidades de un lenguaje existente, que actúa como *anfitrión* de un nuevo *DSL*. En este caso, el *DSL* puede ser implementado como un lenguaje compilado, donde el código fuente escrito en el *DSL* es transformado en código escrito completamente en el lenguaje anfitrión, aprovechando así todos los elementos lingüísticos provistos por este último, incluyendo (por ejemplo) el manejo de expresiones, variables y procedimientos. Este patrón puede ser usado cuando el *DSL* comparte elementos en común con algún otro lenguaje, típicamente uno de propósito general. El proceso de traducción es relativamente sencillo y puede ser implementado usando el patrón *Source-to-source transformation*.

2. *Language extension*. Añade nuevas características y elementos sintácticos sobre el núcleo de un lenguaje existente para que pueda satisfacer una nueva necesidad. De esta forma, un nuevo *DSL* puede ser diseñado e implementado como una extensión del lenguaje *base*. Este patrón difiere del patrón *Piggyback* en cuanto a los roles desempeñados por los dos lenguajes: el patrón *Piggyback* usa un lenguaje existente como un medio para implementar un nuevo *DSL*, mientras que el patrón de extensión es usado cuando un lenguaje existente es extendido dentro de su propio marco sintáctico y semántico para formar un nuevo *DSL*.

3. *Language specialization*. Elimina determinadas características de un lenguaje existente para limitar sus capacidades. De esta forma, un nuevo *DSL* puede ser diseñado e implementado como un subconjunto del lenguaje *base*. Esto pues, en algunos casos, la potencia de un lenguaje existente puede impedir su adopción para un propósito especializado. El diseño del nuevo *DSL* involucra la eliminación de las características sintácticas o semánticas que no se desean del lenguaje base.

4. *Source-to-source transformation*. Permite una implementación eficiente del traductor de un *DSL* a través de un proceso que transforma código fuente escrito en el *DSL* en código escrito en un lenguaje existente. Las herramientas disponibles para el lenguaje existente son usadas para compilar o interpretar el código generado por el proceso de transformación, aprovechando sus características y su infraestructura.

Parte II

Propuesta de solución

Capítulo 5

Requerimientos

El objetivo principal del proyecto *GOLD 3* es el diseño e implementación de un lenguaje de programación de propósito específico que facilite a los desarrolladores la escritura de algoritmos sobre estructuras de datos avanzadas como árboles, grafos y autómatas a través de una sintaxis muy cercana al pseudocódigo, basada en la notación matemática estándar que se usa en los libros de texto para manipular números, expresiones booleanas, conjuntos, secuencias y otros dominios de interés. Para lograr este objetivo se establecieron varios requerimientos, inspirados en su mayoría en los pseudocódigos trabajados en el texto *Introduction to Algorithms* [1], en especial el correspondiente al algoritmo de Dijkstra para resolver el problema de la ruta más corta en grafos.

En este capítulo se enuncian los requerimientos básicos que guiaron el desarrollo de *GOLD 3*, clasificados bajo los criterios consignados en las siguientes referencias:

- El estándar internacional *ISO/IEC* 9126 [52], que establece algunos lineamientos generales para evaluar la calidad del software.
- El texto *Programming languages : design and implementation* de Pratt y Zelkowitz [41], que presenta algunos atributos que debe tener un buen lenguaje de programación.
- El texto *Programming Language Design Concepts* de Watt [40], que enumera algunos criterios técnicos y económicos que deben ser considerados cuando se evalúa el uso de un lenguaje de programación.

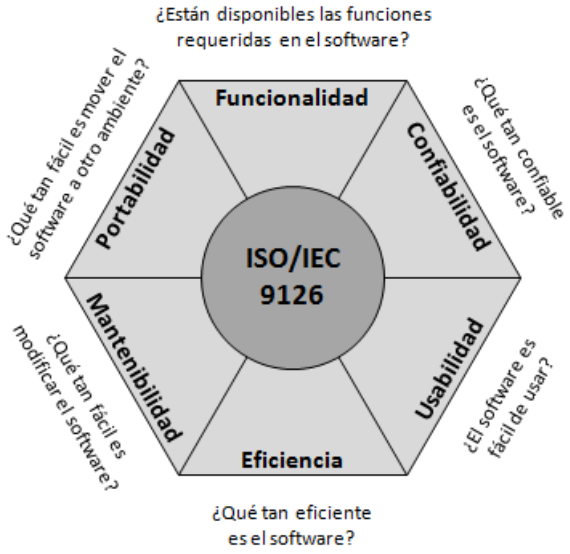
Código 5.1. *Algoritmo de Dijkstra implementado en GOLD 3.*

```
1  function dijkstra(G:IGraph,s) begin
2    V:=G.getVertices()
3    d,π:=GHashTableMap(|V|),GHashTableMap(|V|)
4    for each v∈V do
5      d[v],π[v]:=∞,NIL
6    end
7    d[s]:=0
8    Q:=GFibonacciHeap(d)
9    while Q≠∅ ∧ Q.minimumKey()≠∞ do
10     u:=Q.extractMinimum()
11     for each v∈G.getSuccessors(u) do
12       if v∈Q ∧ d[v]>d[u]+G.getCost(u,v) then
13         d[v],π[v]:=d[u]+G.getCost(u,v),u
14         Q.decreaseKey(v,d[v])
15       end
16     end
17   end
18   return ⟨d,π⟩
19 end
```

5.1. Criterios de calidad de acuerdo con el estándar *ISO/IEC* 9126

La clasificación de los criterios de esta sección están basados en el estándar internacional *ISO/IEC* 9126 [52], que establece algunos lineamientos generales para evaluar la calidad del software, orientando los objetivos del proyecto.

Figura 5.1. *Criterios establecidos por el estándar ISO/IEC 9126 [52].*



5.1.1. Funcionalidad (*Functionality*)

El lenguaje debe permitir la programación de algoritmos sobre grafos y otras estructuras de datos avanzadas en grandes proyectos de software, aprovechando la expresividad de un lenguaje de propósito general orientado a objetos como *Java*, y beneficiándose del estilo de escritura de los pseudocódigos para fomentar la programación de código fuente compacto, claro y entendible.

Específicamente, el lenguaje de programación a diseñar en el proyecto debe satisfacer las siguientes funcionalidades y requisitos, vitales para lograr los objetivos propuestos en la sección §1.6:

1. Debe tener una sintaxis similar a la usada en los pseudocódigos descritos en el texto *Introduction to Algorithms* [1] de Thomas Cormen et al. y debe proveer instrucciones fáciles de recordar, buscando mejorar la legibilidad de los programas implementados en el lenguaje.

2. Debe permitir la definición formal de secuencias, conjuntos, bolsas, grafos y autómatas mediante expresiones matemáticas. En particular:

   - Las secuencias deben poderse definir por extensión (enumerando sus elementos).
   - Los conjuntos y las bolsas deben poderse definir tanto por comprensión (describiendo las propiedades que cumplen sus elementos) como por extensión (enumerando sus elementos).
   - Los grafos dirigidos y no dirigidos deben poderse definir describiendo su conjunto de vértices, su conjunto de arcos y su función de costo.
   - Los autómatas finitos determinísticos y no determinísticos deben poderse definir describiendo su conjunto de estados, su alfabeto, su estado inicial, sus estados finales y su función de transición.

   Adicionalmente, debe existir un mecanismo sencillo para definir la función de costo de los grafos y la función de transición de los autómatas.

3. Debe usar simbología matemática estándar fácil de recordar, permitiendo la escritura de expresiones lógicas y aritméticas a través de constantes matemáticas, conectivos booleanos, operadores y cuantificaciones. Todos los símbolos de constante y de operador deben coincidir con los que se usan comúnmente en los textos de matemáticas. Por otro lado, todas las cuantificaciones deben poderse escribir en una notación similar a la introducida en el texto *A Logical Approach To Discrete Math* de David Gries y Fred Schneider [12], sobre operadores conmutativos y asociativos como los siguientes:

Tabla 5.1. *Cuantificadores que debe suministrar GOLD 3.*

Operador	**Cuantificación**	**Símbolo**
Suma	Sumatoria	Σ
Multiplicación	Multiplicatoria	Π
Conjunción	Cuantificación universal	$\forall$ (para todo)
Disyunción	Cuantificación existencial	$\exists$ (existe)
Máximo		$\uparrow$
Mínimo		$\downarrow$
Unión		$\cup$
Intersección		$\cap$

4. Debe permitir la declaración, manipulación e implementación de algoritmos sobre algunas de las estructuras de datos más básicas, entre éstas las siguientes:

Tabla 5.2. *Estructuras de datos e implementaciones que debe suministrar GOLD 3.*

Estructura de datos	**Implementaciones a proveer**
Tupla (*tuple*)	Tuplas vacías, *singletons*, pares ordenados, *n*-tuplas.
Lista (*list*/*sequence*)	Vectores dinámicos, Listas doblemente encadenadas.
Conjunto (*set*)	Árboles Rojinegros, Árboles *AVL*, Tablas de *Hashing*, *Skip Lists*.
Bolsa (*bag*/*multiset*)	Árboles Rojinegros, Árboles *AVL*, Tablas de *Hashing*, *Skip Lists*.
Pila (*stack*)	Vectores dinámicos, Listas doblemente encadenadas.
Cola (*queue*)	Vectores dinámicos circulares, Listas doblemente encadenadas.
Bicola (*deque*)	Vectores dinámicos circulares, Listas doblemente encadenadas.
Montón (*heap*)	*Binary heaps*, *Binomial heaps*, *Fibonacci heaps*, Árboles Rojinegros.
Conjuntos disyuntos (*disjoint-set data structure*)	*Disjoint-set forests* [1], Listas encadenadas.
Árbol binario (*binary tree*)	Árboles sencillamente encadenados.
Árbol eneario (*n-ary tree*)	Vectores dinámicos, *Quadtrees*, *Tries*.
Asociación llave-valor (*map*)	Árboles Rojinegros, Árboles *AVL*, Tablas de *Hashing*, *Skip Lists*.
Asociación llave-valores (*multimap*)	Árboles Rojinegros, Árboles *AVL*, Tablas de *Hashing*, *Skip Lists*.
Grafo dirigido (*directed graph*)	Listas de adyacencia, Matrices de adyacencia, Representaciones implícitas.
Grafo no dirigido (*undirected graph*)	Listas de adyacencia, Matrices de adyacencia, Representaciones implícitas.
Autómata determinista (*deterministic automaton*)	Representaciones explícitas (Tablas de *Hashing*), Representaciones implícitas.
Autómata no determinista (*nondeterministic automaton*)	Representaciones explícitas (Tablas de *Hashing*), Representaciones implícitas.

5. Debe permitir la manipulación de valores pertenecientes a conjuntos como los valores booleanos ($\mathbb{B}$), los números naturales ($\mathbb{N}$), los números enteros ($\mathbb{Z}$), los números racionales ($\mathbb{Q}$), los números reales ($\mathbb{R}$) y los números complejos ($\mathbb{C}$), a través de tipos primitivos de datos y de librerías de alto rendimiento para el desarrollo de operaciones aritméticas de precisión arbitraria, como *Apfloat* [53].

Tabla 5.3. *Tipos primitivos de datos que debe suministrar GOLD 3.*

Tipo primitivo	Símbolo
Caracteres	
Booleanos	$\mathbb{B}$
Naturales	$\mathbb{N}$
Enteros	$\mathbb{Z}$
Racionales	$\mathbb{Q}$
Reales	$\mathbb{R}$
Complejos	$\mathbb{C}$

6. Debe permitir la aplicación de las siguientes operaciones [†1] sobre las estructuras de datos y sobre los tipos primitivos de datos provistos, donde los símbolos de los operadores son los usados en la notación matemática estándar que se estudia en los libros de texto para manipular números, expresiones booleanas, conjuntos, secuencias y objetos relacionados con otros dominios de interés:

Tabla 5.4. *Operadores que debe suministrar GOLD 3.*

Nombre	Contexto	Símbolo(s)
GRUPO 1: OPERADORES BINARIOS MUTUAMENTE ASOCIATIVOS		
Equivalencia (*Equivalence*) / Si y sólo si (*If and only if*)	Booleanos	$\equiv$, $\Leftrightarrow$, *eqv*
Inequivalencia (*Inequivalence*) / O exclusivo (*Exclusive or*)	Booleanos	$\not\equiv$, $\oplus$, *xor*
GRUPO 2: OPERADORES BINARIOS MUTUAMENTE ASOCIATIVOS POR LA DERECHA		
Implicación (*Implication*)	Booleanos	$\Rightarrow$
Anti-implicación (*Anti-implication*)	Booleanos	$\not\Rightarrow$
GRUPO 3: OPERADORES BINARIOS MUTUAMENTE ASOCIATIVOS POR LA IZQUIERDA		
Consecuencia (*Consequence*)	Booleanos	$\Leftarrow$
Anti-consecuencia (*Anti-consequence*)	Booleanos	$\not\Leftarrow$
GRUPO 4: OPERADORES BINARIOS ASOCIATIVOS		
Disyunción (*Disjunction*)	Booleanos	$\vee$, *or*, $\vert\vert$
Conjunción (*Conjunction*)	Booleanos	$\wedge$, *and*, &&
GRUPO 5: OPERADORES BINARIOS MUTUAMENTE CONJUNCIONALES		
Igualdad (*Equality*)		$=$, $==$
Diferente de (*Inequality*)		$\neq$, $!=$, $<>$
Menor que (*Less than*)	Números	$<$
Menor o igual que (*Less than or equal to*)	Números	$\leq$, $<=$
Mayor que (*Greater than*)	Números	$>$
Mayor o igual que (*Greater than or equal to*)	Números	$\geq$, $>=$
Divisibilidad (*Divisibility*)	Números enteros	$\vert$
Anti-divisibilidad (*Anti-divisibility*)	Números enteros	$\nmid$
Pertenencia (*Membership*)	Conjuntos, bolsas	$\in$, *in*
Anti-pertenencia (*Anti-membership*)	Conjuntos, bolsas	$\notin$
GRUPO 6: OPERADORES BINARIOS MUTUAMENTE CONJUNCIONALES		
Colecciones disyuntas (*Disjoint operator*)	Conjuntos, bolsas	$\bowtie$
Subconjunto (*Subset*)	Conjuntos, bolsas	$\subseteq$
No subconjunto (*Not subset*)	Conjuntos, bolsas	$\not\subseteq$
Superconjunto (*Superset*)	Conjuntos, bolsas	$\supseteq$
No superconjunto (*Not superset*)	Conjuntos, bolsas	$\not\supseteq$
Subconjunto propio (*Proper subset*)	Conjuntos, bolsas	$\subset$, $\subsetneq$
No subconjunto propio (*Not proper subset*)	Conjuntos, bolsas	$\not\subset$
Superconjunto propio (*Proper superset*)	Conjuntos, bolsas	$\supset$, $\supsetneq$
No superconjunto propio (*Not proper superset*)	Conjuntos, bolsas	$\not\supset$
GRUPO 7: OPERADORES BINARIOS ASOCIATIVOS POR LA IZQUIERDA		

[1] Los operadores están agrupados según su asociatividad y los grupos están ordenados de menor a mayor según su precedencia (véase la sección §4.1.5.2 del marco teórico). El único operador que no aparece en la literatura es $\bowtie$, que sirve para determinar si dos colecciones son disyuntas o no: dadas A y B dos colecciones, $A \bowtie B \Leftrightarrow ((A \cap B) = \varnothing)$.

Prepend [12]	Secuencias	$\triangleleft$
Concatenación (*Concatenation*)	Secuencias	$\hat{}$
GRUPO 8: OPERADORES BINARIOS ASOCIATIVOS POR LA DERECHA		
Append [12]	Secuencias	$\triangleright$
GRUPO 9: OPERADORES BINARIOS NO ASOCIATIVOS		
Número de ocurrencias (*Number of occurrences*)	Bolsas, secuencias	#
GRUPO 10: OPERADORES BINARIOS NO ASOCIATIVOS		
Rango de intervalo (*Interval range*)	Números	..
GRUPO 11: OPERADORES BINARIOS ASOCIATIVOS		
Máximo (*Maximum*)	Números	$\uparrow$
Mínimo (*Minimum*)	Números	$\downarrow$
GRUPO 12: OPERADORES BINARIOS MUTUAMENTE ASOCIATIVOS POR LA IZQUIERDA		
Adición (*Addition*)	Números	$+$
Sustracción (*Subtraction*)	Números	$-$
GRUPO 13: OPERADORES BINARIOS MUTUAMENTE ASOCIATIVOS POR LA IZQUIERDA		
Multiplicación (*Multiplication*)	Números	$*$, $\cdot$
División (*Division*)	Números	$/$
Residuo entero (*Integer residue*) / Módulo (*Module*)	Números enteros	%, *mod*
División entera (*Integer division*) / Cociente (*Quotient*)	Números enteros	$\div$, *div*
Máximo común divisor (*Greatest common divisor*)	Números enteros	*gcd*
Mínimo común múltiplo (*Least common multiple*)	Números enteros	*lcm*
GRUPO 14: OPERADORES BINARIOS ASOCIATIVOS POR LA IZQUIERDA		
Unión (*Union*)	Conjuntos, bolsas	$\cup$
Intersección (*Intersection*)	Conjuntos, bolsas	$\cap$
Diferencia (*Difference*)	Conjuntos, bolsas	$\setminus$
Diferencia simétrica (*Symmetric difference*)	Conjuntos, bolsas	$\triangle$
GRUPO 15: OPERADORES BINARIOS ASOCIATIVOS POR LA IZQUIERDA		
Potenciación (*Exponentiation*)	Números	$\hat{}$
Potenciación cartesiana (*Cartesian power*)	Conjuntos	$\hat{}$
GRUPO 16: OPERADORES BINARIOS ASOCIATIVOS		
Producto cartesiano (*Cartesian product*)	Conjuntos	$\times$
GRUPO 17: OPERADORES UNARIOS PREFIJOS		
Más unario (*Unary plus sign*)	Números	$+$
Menos unario (*Unary minus sign*)	Números	$-$
Negación (*Negation*)	Booleanos	$\neg$, *not*, !
Cardinalidad (*Cardinality*)	Conjuntos, bolsas, secuencias	#
Complemento (*Complement*)	Conjuntos	$\sim$
Conjunto potencia (*Power set*)	Conjuntos	$\wp$
GRUPO 18: OPERADORES UNARIOS POSFIJOS		
Factorial (*Factorial*)	Números	!
GRUPO 19: PARÉNTESIS (BRACKETS)		
Cardinalidad (*Cardinality*)	Conjuntos	$\lvert\cdot\rvert$
Valor absoluto (*Absolute value*)	Números	$\lvert\cdot\rvert$
Piso (*Floor*)	Números	$\lfloor\cdot\rfloor$
Techo (*Ceiling*)	Números	$\lceil\cdot\rceil$

7. Debe permitir la utilización de cualquier clase implementada en *Java* a través de declaración de variables, invocación de constructores, consulta de atributos e invocación de métodos. Los tipos primitivos y las clases deben poderse referenciar en *GOLD* sin importar su origen:

   - Tipos primitivos del lenguaje de programación *Java*: `boolean` (valores booleanos), `char` (caracteres *Unicode*), `byte` (enteros de 8 bits), `short` (enteros de 16 bits), `int` (enteros de 32 bits), `long` (enteros de 64 bits), `float` (flotantes de 32 bits) y `double` (flotantes de 64 bits).
   - Clases pertenecientes a la librería estándar de *Java*, que se encuentran documentadas en su *API* (*Application Programming Interface*).

- Clases pertenecientes a librerías externas empaquetadas en archivos *JAR* o distribuidas en archivos compilados con extensión *.class*.
- Clases pertenecientes al usuario o a otras personas, cuya implementación esté disponible en código fuente en archivos con extensión *.java*.
- Clases pertenecientes a la librería suministrada por *GOLD*, que contiene las implementaciones a las estructuras de datos provistas, y rutinas útiles para la administración y visualización de objetos creados en *GOLD*.

Con respecto a la copia de valores, los correspondientes a los tipos primitivos deben manejarse por valor y las instancias de las clases deben manejarse por referencia. Además, el usuario debe tener la posibilidad de decidir si desea declarar explícitamente el tipo de las variables que necesita usar en su programa, obligando a que *GOLD* deba comportarse como un lenguaje estáticamente tipado o como un lenguaje dinámicamente tipado dependiendo de la circunstancia. En caso de que el usuario opte por no declarar el tipo de una variable, se puede entender por defecto que tiene un tipo general, como `java.lang.Object`.

8. Debe permitir la declaración de *funciones* capaces de retornar un valor resultado y de *procedimientos* sin retorno, con las siguientes consideraciones:

   - Deben poder ser implementados iterativa o recursivamente.
   - Deben poder ser invocados desde programas codificados en el lenguaje *GOLD* o en el lenguaje *Java*.
   - El paso de parámetros se debe definir como lo establecido en el lenguaje de programación *Java*: las variables de tipo primitivo (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`) deben pasarse por valor y las instancias de las clases (i.e., los objetos de tipo no básico) deben pasarse por referencia.

9. Debe suministrar comandos fáciles de escribir y de recordar, incluyendo los siguientes:

   - Asignaciones.
   - Intercambios (*swaps*).
   - Condicionales (*if-then*, *if-then-else*).
   - Ciclos (*while*, *do-while*, *repeat-until*, *for*, *for-each*).
   - Llamados a función y a procedimiento, tanto los implementados en el lenguaje *GOLD* como los implementados en el lenguaje *Java*.

   Adicionalmente, el lenguaje debe ofrecer las siguientes instrucciones para facilitar la programación:

   - Declaración de variables globales (comunes a todas las funciones y procedimientos) o locales (propias de una función o procedimiento).
   - Aserciones para realizar depuración de código.
   - Lanzamiento de excepciones para notificar fallas en el flujo normal de ejecución.
   - Retorno de valores para informar el valor calculado por una función.
   - Secuenciadores para alterar el flujo normal de ejecución: *break* para terminar la ejecución de un ciclo, *continue* para seguir con la siguiente iteración del ciclo sin ejecutar el código subsiguiente y *finalize* para terminar la ejecución de un procedimiento.

10. Debe ofrecer un entorno de programación completo y maduro con las siguientes virtudes y capacidades:

    - Coloración de la sintaxis (*syntax highlighting*).
    - Indentamiento automático del código fuente (*code formatting*).
    - Resaltado de los errores de compilación en tiempo de desarrollo, dando al usuario descripciones detalladas de cada error.
    - Facilidades para insertar símbolos matemáticos especiales a través de una tabla de caracteres *Unicode*.

- Facilidades para auto-completar instrucciones en el código fuente.
- Facilidades para la ejecución y depuración de los programas desarrollados.
- Funcionalidades inherentes a los editores de texto tradicionales: abrir, guardar, copiar/cortar/pegar, buscar/reemplazar, hacer/deshacer, etcétera.

11. Debe proporcionar mecanismos para que los desarrolladores puedan animar gráficamente y paso a paso la operación de los programas en tiempo de ejecución. Se deben poder configurar los atributos visuales de los nodos y de los arcos de los grafos, así como el algoritmo utilizado para ubicar los nodos en la superficie de dibujo (*layout algorithm* [54]).

12. Como mínimo, debe permitir la implementación de los siguientes algoritmos sobre grafos [55]:

    - Ordenamiento topológico (*Topological Sort*).
    - Coloreo de grafos bipartitos (*Bipartite Matching*).
    - Recorridos sobre grafos: Recorrido por anchura (*BFS*: *Breadth First Search*), Recorrido por profundidad (*DFS*: *Depth First Search*).
    - Ciclos eulerianos (*Eulerian Circuits*): Algoritmo de Fleury, Algoritmo de Hierholzer.
    - Ciclos hamiltonianos (*Hamiltonian Circuits*): Algoritmo de backtracking $O(n!)$, Algoritmo de programación dinámica $O\left(n^2 \cdot 2^n\right)$.
    - Problema de la ruta más corta (*Shortest Path*): Algoritmo de Dijkstra, Algoritmo de Bellman-Ford, Algoritmo de Johnson, Algoritmo de Floyd-Warshall.
    - Problema del árbol de expansión mínimo (*MST*: *Minimum Spanning Tree*): Algoritmo de Prim-Jarník, Algoritmo de Kruskal, Algoritmo de Borůvka, Algoritmo *reverse-delete*.
    - Problema del clique maximal (*Maximal Clique Problem*): Algoritmo de Bron-Kerbosch.
    - Problema del agente viajero (*TSP*: *Travelling Salesman Problem*): Algoritmo de backtracking $O(n!)$, Algoritmo de programación dinámica $O\left(n^2 \cdot 2^n\right)$.
    - Componentes fuertemente conexas (*Strongly Connected Components*): Algoritmo de Tarjan, Algoritmo de Kosaraju, Algoritmo de Cheriyan-Mehlhorn/Gabow.
    - Redes de flujo (*Flow Networks*): Algoritmo de Ford-Fulkerson, Algoritmo de Edmonds-Karp.

Los requerimientos enumerados anteriormente persiguen el desarrollo de un lenguaje potente, expresivo, sencillo y fácil de usar, que satisfaría las siguientes características del estándar *ISO/IEC* 9126 [52], clasificadas bajo el criterio de funcionalidad:

- *Idoneidad* (*Suitability*). El lenguaje debe facilitar la programación de algoritmos sobre grafos y otras estructuras de datos. Todas las funcionalidades y requisitos expuestos deben satisfacerse para que el producto sea idóneo para este fin.
- *Precisión* (*Accuracy*). Los programas escritos en el lenguaje deben tener los efectos esperados y deben arrojar los resultados correctos. Para mostrar el cumplimiento de este hecho, es necesario describir formalmente la semántica operacional del lenguaje a través de la función de traducción utilizada para compilar a código ejecutable los programas escritos en *GOLD*. De ser posible, sería ideal describir también la semántica axiomática del lenguaje mediante teoremas de corrección que hagan posible la verificación de algoritmos implementados en *GOLD*.
- *Interoperabilidad* (*Interoperability*). El lenguaje debe ser capaz de interactuar con algún lenguaje de programación de propósito general. Concretamente, se requiere que las funciones y procedimientos escritos en *GOLD* se puedan invocar desde *Java*, y que todas las clases de *Java* se puedan utilizar en los programas escritos en *GOLD*.

5.1.2. Confiabilidad (*Reliability*)

El producto debe satisfacer las siguientes características del estándar *ISO/IEC* 9126 [52], clasificadas bajo el criterio de confiabilidad:

- *Madurez* (*Maturity*). La ejecución de programas escritos en *GOLD* no debe presentar errores causados por fallas en el proceso de traducción o por errores de programación en la librería suministrada. Asimismo, el entorno de programación de *GOLD* debe estar libre de fallos que entorpezcan el trabajo de desarrollo.
- *Tolerancia a fallas* (*Fault tolerance*). El entorno de programación de *GOLD* debe poder seguir operando con un rendimiento adecuado después de cualquier eventual falla de alguno de sus componentes internos. Adicionalmente, el sistema debe lanzar errores adecuados ante cualquier acción del usuario que no esté permitida.
- *Capacidad de recuperación* (*Recoverability*). En caso de alguna falla, el entorno de programación de *GOLD* debería tener la capacidad de restablecerse en un tiempo prudencial sin ver afectado dramáticamente su rendimiento y sin dañar o perder ningún dato del usuario, ya sea código fuente o cualquier otro tipo de información.

5.1.3. Usabilidad (*Usability*)

El producto debe satisfacer las siguientes características del estándar *ISO/IEC* 9126 [52], clasificadas bajo el criterio de usabilidad:

- *Comprensibilidad* (*Understandability*). El entorno de programación debe ser intuitivo de usar y debe rescatar paradigmas comunes a los ambientes de desarrollo de grandes lenguajes de programación, como *Eclipse*, *Netbeans* y *Dev-C++*.
- *Facilidad de aprendizaje* (*Learnability*). La sintaxis del lenguaje debe ser similar al pseudocódigo del texto *Introduction to Algorithms* [1] y debe proveer instrucciones fáciles de recordar, fomentando la programación de código fuente compacto, claro y legible. Aunque el lenguaje debe ser fácil de aprender, indudablemente debe existir un manual de usuario sencillo que apoye la labor de aprendizaje de la herramienta. Incluso un manual de usuario incipiente podría ser eficaz para ayudar a usar el lenguaje.
- *Atractivo* (*Attractiveness*). El sistema debe incitar a los usuarios a que aprovechen sus bondades, debe generar cierto tipo de dependencia para que lo sigan usando con periodicidad en sus proyectos de software y debe ser lo suficientemente agradable para que puedan recomendarlo a sus colegas.

5.1.4. Eficiencia (*Efficiency*)

El producto debe satisfacer las siguientes características del estándar *ISO/IEC* 9126 [52], clasificadas bajo el criterio de eficiencia:

- *Comportamiento en el tiempo* (*Time behaviour*). Se deben proveer mecanismos eficientes que realicen automáticamente la validación semántica y el resaltado de la sintaxis a medida que el usuario vaya digitando el código fuente, y que realicen el proceso de compilación cada vez que el usuario guarde un archivo implementado en el lenguaje. Adicionalmente, el código ejecutable traducido no debe presentar sobrecargas considerables de tiempo adicionales a las inherentes al tiempo de procesamiento empleado por las instrucciones codificadas por el desarrollador.
- *Comportamiento de recursos* (*Resource behavior*). En tiempo de ejecución, el consumo de espacio y de recursos de red dependería de las características de los programas implementados por el usuario en el lenguaje. En todo caso, el entorno de programación debe consumir una cantidad regular de memoria principal, necesaria para la sostenibilidad de los servicios ofrecidos al usuario.

5.1.5. Mantenibilidad (*Maintainability*)

El producto debe satisfacer las siguientes características del estándar *ISO/IEC* 9126 [52], clasificadas bajo el criterio de mantenibilidad:

- *Facilidad de análisis* (*Analyzability*). La arquitectura de la infraestructura que da soporte al lenguaje debe estar diseñada y organizada de tal manera que permita el rápido diagnóstico de la causa de alguna falla o deficiencia presentada, mediante una fácil identificación del componente que debe ser corregido o modificado.
- *Facilidad de cambio* (*Changeability*). La implementación del producto debe aplicar buenas prácticas de desarrollo de software como la utilización de patrones de diseño, buscando facilitar la modificación, corrección o mejora de cualquiera de sus componentes, ya sea para enriquecer alguna funcionalidad o para reparar fallas. Debe existir documentación técnica adecuada que apoye la labor de mantener y evolucionar el producto.
- *Facilidad de pruebas* (*Testability*). Debe proveerse un conjunto de pruebas unitarias implementadas a través de la librería *JUnit* [56], que puedan ser ejecutadas automáticamente en lote para detectar fallos o inconsistencias luego de cualquier modificación que sufra el software.
- *Estabilidad* (*Stability*). Las pruebas automáticas que se diseñen deben reducir considerablemente el riesgo de inyectar fallas en el producto luego de cualquier modificación.
- *Extensibilidad* (*Extensibility*). Debe ser relativamente intuitivo adicionar nuevas características al lenguaje.

5.1.6. Portabilidad (*Portability*)

El producto debe satisfacer las siguientes características del estándar *ISO/IEC* 9126 [52], clasificadas bajo el criterio de portabilidad:

- *Adaptabilidad* (*Adaptability*). El producto debe operar uniformemente en los sistemas operativos *Windows*, *Linux*, *Mac OS* y *Solaris* en procesadores de 32 y de 64 bits.
- *Facilidad de instalación* (*Installability*). El producto debe ser fácil de instalar en todos los sistemas operativos para los que fue diseñado y no debe necesitar que el usuario realice acciones inusuales o extraordinarias para garantizar un correcto funcionamiento. Debe existir un manual de instalación que especifique los requisitos mínimos de hardware y de software, y que describa claramente el procedimiento para instalar el producto.

Para facilitar la portabilidad es aconsejable que la aplicación sea distribuida como un *plug-in* de *Eclipse* [7], puesto que este entorno de programación ya se encuentra disponible en los sistemas operativos enumerados. Además se facilitaría el cumplimiento de muchos de los requerimientos descritos anteriormente, en especial aquellos relacionados con el ambiente de desarrollo.

5.2. Criterios de calidad de acuerdo con Pratt y Zelkowitz

La clasificación de los criterios de esta sección están basados en los consignados en el texto *Programming languages : design and implementation* de Pratt y Zelkowitz [41], que presenta algunos atributos que debe tener un buen lenguaje de programación. Aplicando estos criterios al proyecto, obtenemos los siguientes requerimientos:

1. *Claridad, simplicidad y unidad* (*Clarity, simplicity and unity*). El lenguaje debe proveer un conjunto de conceptos claro, simple y unificado que puedan ser usados como primitivas al momento de desarrollar algoritmos [41]. Adicionalmente, la sintaxis del lenguaje debe favorecer la legibilidad, facilitando la escritura de algoritmos de tal forma que en el futuro sean fáciles de entender, de probar y de modificar [41].

2. *Ortogonalidad* (*Orthogonality*). Debe ser posible combinar varias características del lenguaje en todas las formas posibles, donde cada combinación tenga cierto significado [41]. Concretamente:

- las expresiones y las instrucciones deben ser ortogonales: las expresiones (que están formadas por constantes, variables, operaciones, cuantificaciones, conjuntos (por enumeración o por comprensión), bolsas (por enumeración o por comprensión), secuencias, arreglos, aplicaciones de función y llamados a procedimiento) deben poderse combinar de todas las maneras concebibles y deben poderse usar dentro de cualquier instrucción que pueda referir una expresión (las asignaciones, las inicializaciones de variable, los retornos de las funciones, las guardas de los condicionales y de los ciclos, los argumentos que son pasados a las funciones y los procedimientos, etcétera); y
- la invocación de miembros de clase y las expresiones deben ser ortogonales: debe ser posible invocar atributos y métodos sobre cualquier expresión, dependiendo de la clase *Java* que represente el tipo de la expresión.

En caso de que alguna combinación de características genere una inconsistencia de tipos, se debe lanzar un error de ejecución. En particular, las siguientes acciones deben resultar en una excepción:

- la evaluación de una expresión no booleana en la guarda de un condicional o de un ciclo;
- la invocación de un atributo o de un método sobre una expresión que llame un método sin retorno (i.e., con retorno *void*);
- la asignación a una variable de un valor cuyo tipo no coincida con el de la variable; y
- la invocación de una función o de un procedimiento con un argumento cuyo tipo no coincida con el declarado en sus parámetros.

3. *Naturalidad para la aplicación* (*Naturalness for the application*). La sintaxis del lenguaje debe permitir que los programas puedan reflejar la estructura lógica subyacente de los algoritmos que implementan [41]. Adicionalmente, el lenguaje debe suministrar estructuras de datos, operaciones e instrucciones de control apropiadas [41] para la manipulación de grafos y otras estructuras de datos avanzadas. La sintaxis del lenguaje debe facilitar la escritura de expresiones aritméticas y lógicas a través de la notación que suele trabajarse en los libros de texto, para la definición de objetos como conjuntos, bolsas, secuencias, grafos y autómatas.

4. *Apoyo para la abstracción* (*Support for abstraction*). El lenguaje debe permitir la definición de nuevas estructuras de datos especificando sus atributos e implementando sus operaciones usando las características primitivas brindadas por el lenguaje, de tal forma que el desarrollador pueda usarlas en otras partes del programa conociendo únicamente sus propiedades, sin preocuparse por los detalles de implementación [41].

5. *Facilidad para la verificación de programas* (*Ease of program verification*). El lenguaje debe facilitar la verificación formal o informal de que los programas desempeñan correctamente su función mediante técnicas como [41]:

   - *Formal verification*. Demostrar formalmente que los programas son correctos con respecto a su especificación a través de teoremas de corrección que definen su semántica axiomática.
   - *Model checking*. Demostrar que los programas son correctos con respecto a su especificación probando automáticamente todos los posibles estados que pueden tener los parámetros de entrada y verificando que los resultados satisfagan la postcondición.
   - *Desk checking*. Revisar manual y exhaustivamente el código fuente de los programas para garantizar que no tienen errores y que su semántica operacional coincida con la lógica del algoritmo implementado.
   - *Program testing*. Probar automáticamente los programas ejecutándolos con un conjunto de casos de entrada que satisfacen la precondición y revisando que las salidas cumplen la postcondición.

6. *Entorno de programación* (*Programming environment*). Debe existir un entorno de desarrollo integrado (*IDE*: *integrated development environment*) que facilite la implementación de programas en el lenguaje [41] y que provea:

- una implementación confiable, eficiente y bien documentada del lenguaje [41];
- un editor especializado que coloree la sintaxis del lenguaje, indente automáticamente el código fuente, resalte los errores de compilación y permita la inserción de símbolos matemáticos especiales a través de una tabla de caracteres *Unicode*;
- una librería que implemente las estructuras de datos fundamentales que ofrece el lenguaje;
- herramientas para ejecutar, depurar y probar con comodidad los programas desarrollados; y
- funcionalidades comunes a los editores de texto como abrir, guardar, copiar/cortar/pegar, buscar/reemplazar, hacer/deshacer, etcétera.

7. *Portabilidad de los programas* (*Portability of programs*). Los programas desarrollados en una máquina deben poderse ejecutar en cualquier otra máquina que tenga instalada la infraestructura mínima que demanda el lenguaje. El comportamiento de la ejecución no debe verse alterado por factores que dependen de las máquinas como su sistema operativo, su hardware o su software instalado.

8. *Costo de uso* (*Cost of use*). Pratt y Zelkowitz [41] enumeran las siguientes medidas de costo para evaluar un lenguaje de programación:

   - *Costo de ejecución* (*Cost of program execution*). El código ejecutable traducido no debe presentar sobrecargas considerables de tiempo adicionales a las inherentes al tiempo de procesamiento empleado por las instrucciones codificadas por el desarrollador.
   - *Costo de compilación* (*Cost of program translation*). Se deben proveer mecanismos eficientes que realicen automáticamente la validación semántica y el resaltado de la sintaxis a medida que el usuario vaya digitando el código fuente, y que realicen el proceso de compilación cada vez que el usuario guarde un archivo implementado en el lenguaje.
   - *Costo de creación, pruebas y uso* (*Cost of program creation, testing and use*). Los usuarios que usen el lenguaje deberían esforzarse menos y ahorrar más tiempo que los usuarios que no lo usen para resolver un mismo problema sobre grafos o sobre alguna estructura de datos avanzada, contabilizando el tiempo que les toma diseñar los programas, implementarlos, ejecutarlos, revisarlos, probarlos y usarlos.
   - *Costo de mantenimiento* (*Cost of program maintenance*). Los programas implementados en el lenguaje deben ser fáciles de mantener, tratando de reducir el costo total de su ciclo de vida. ‘‘El mantenimiento incluye la reparación de errores descubiertos después de que el programa es puesto en uso, cambios necesarios por una actualización en el hardware o en el sistema operativo subyacente, y extensiones y mejoras que se necesiten para satisfacer nuevos requerimientos’’ [41].

5.3. Criterios de calidad de acuerdo con Watt

La clasificación de los criterios de esta sección están basados en los consignados en el texto *Programming Language Design Concepts* de Watt [40], que enumera algunos criterios técnicos y económicos que deben ser considerados cuando se esté evaluando el uso de un lenguaje de programación para un determinado fin. Aplicando estos criterios al proyecto, obtenemos los siguientes requerimientos (algunos tomados textualmente del libro de Watt [40]):

1. *Escalabilidad* (*Scale*). El lenguaje debe permitir el desarrollo de proyectos de gran escala, permitiendo que los programas sean construidos desde unidades de compilación que han sido codificadas y probadas separadamente, tal vez por programadores distintos [40].

2. *Modularidad* (*Modularity*). El lenguaje debe permitir el desarrollo de proyectos en donde se pueda descomponer el código fuente en módulos con funciones claramente distinguibles, a través de la organización del código fuente en proyectos, paquetes y clases [40].

3. *Reusabilidad* (*Reusability*). El lenguaje debe permitir la reutilización de módulos a través de librerías que empaqueten código fuente que ya haya sido probado, reuniendo implementaciones escritas en el lenguaje con un mecanismo similar al provisto por el formato *JAR*.

4. *Portabilidad* (*Portability*). El lenguaje debe garantizar que el código fuente sea portable entre todas las plataformas para las que fue diseñado, operando uniformemente en los distintos sistemas operativos.

5. *Nivel* (*Level*). El lenguaje debe fomentar la programación en términos de abstracciones de alto nivel [40] que usen expresiones matemáticas y manipulen estructuras de datos avanzadas como los grafos.

6. *Confiabilidad* (*Reliability*). El lenguaje debe estar diseñado de tal forma que los errores de programación puedan ser detectados y eliminados tan rápido como sea posible [40].

7. *Eficiencia* (*Efficiency*). El lenguaje debe contar con un compilador eficiente que traduzca el código fuente a su forma ejecutable cada vez que sea necesario, y debe tener un bajo costo de ejecución (véase la sección §5.2).

8. *Legibilidad* (*Readability*). El lenguaje debe fomentar la escritura de código fuente que sea legible, siguiendo una sintaxis similar a los pseudocódigos del libro *Introduction to Algorithms* de Cormen et al. [1].

9. *Modelaje de datos* (*Data modeling*). El lenguaje debe suministrar tipos y operaciones asociadas que sean adecuadas [40] para describir y manipular estructuras de datos avanzadas como los grafos. Además, el lenguaje debe permitir que el desarrollador pueda definir e implementar sus propias estructuras de datos que luego puede manipular en el lenguaje.

10. *Modelaje de procesos* (*Process modeling*). El lenguaje debe suministrar instrucciones de control apropiadas para manipular estructuras de datos avanzadas como los grafos.

11. *Disponibilidad de compiladores y de herramientas* (*Availability of compilers and tools*). El lenguaje debe contar con un compilador que traduzca el código fuente a su forma ejecutable y debe contar con un entorno de programación agradable (véase la sección §5.2). ‘‘Un compilador de buena calidad debe validar la sintaxis del lenguaje, debe generar código ejecutable correcto y eficiente, debe generar validaciones en tiempo de ejecución que atrapen errores no detectados en tiempo de compilación, y debe reportar todos los errores clara y precisamente’’ [40].

12. *Familiaridad* (*Familiarity*). El lenguaje debe ser de alguna manera familiar a los programadores, así no lo hayan usado antes. Para lograr esto, se recomienda que la sintaxis del lenguaje evoque características de los pseudocódigos del libro *Introduction to Algorithms* de Cormen et al. [1] y del lenguaje de programación *Java*.

Capítulo 6

Herramientas

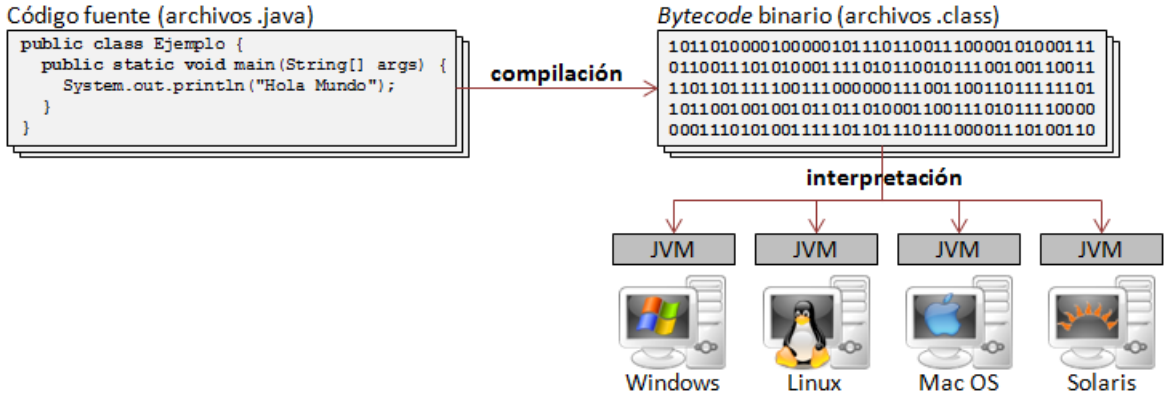
En este capítulo se realiza un breve inventario de todas las tecnologías y librerías externas que se usaron durante el diseño y la implementación de la infraestructura del lenguaje *GOLD 3*, explicando las razones que motivaron la escogencia de cada una de éstas. De ahora en adelante, el nombre *GOLD* será usado para referirse exclusivamente al lenguaje *GOLD 3*, mientras que los nombres *GOLD 1* [3] y *GOLD 2* [4] se reservarán para las dos versiones anteriores.

6.1. Tecnologías

6.1.1. *Java*

La infraestructura del lenguaje *GOLD* está implementada completamente en el lenguaje *Java*, desarrollado en los años noventa por *Sun Microsystems*, que en 2010 pasó a ser propiedad de *Oracle Corporation*. *Java* es un lenguaje de programación de propósito general orientado a objetos ampliamente usado en la actualidad tanto en entornos académicos como empresariales, cuya característica más representativa es su gran portabilidad, permitiendo que los programas escritos en *Java* puedan ser ejecutados en cualquiera de los sistemas operativos más ampliamente utilizados (*Windows*, *Linux*, *Mac OS* y *Solaris*) sin tener que modificar el código fuente. Aunque el lenguaje *Java* tiene una sintaxis similar a la de *C* y *C++*, se diferencia principalmente de estos últimos en que restringe el uso de ciertos mecanismos de bajo nivel como el manejo explícito de apuntadores y la administración manifiesta de memoria principal, provee un recolector de basura que facilita la liberación automática del espacio en memoria ocupado por objetos que se dejaron de referenciar, y no permite la herencia múltiple.

Figura 6.1. *Proceso de compilación e interpretación de programas en Java.*



Las aplicaciones implementadas en *Java* son sometidas a un proceso de compilación que transforma el código

fuente (editado en archivos con extensión `.java`) en un tipo de código intermedio llamado *bytecode* (reunido en archivos binarios con extensión `.class`), que posteriormente es interpretado por la *Máquina Virtual de Java* (*JVM*: *Java Virtual Machine*). *Java* suministra varias versiones de la *JVM* que pueden ser instaladas en diversos sistemas operativos (*Windows*, *Linux*, *Mac OS* y *Solaris*) con distintas arquitecturas de procesador (32 bits y 64 bits), cuenta con una librería estándar de clases bastante amplia que implementa su *API* (*Application Programming Interface*), dispone de un sinnúmero de librerías externas que extienden el *API* con funcionalidades especializadas, y posee una vasta cantidad de documentación disponible en recursos bibliográficos y en la red.

La versión específica de *Java* que se está explotando es *Java SE 6* (*Java Standard Edition 6*), distribuida a través de *JDK 6* (*Java Development Kit 6*). *Java* fue el lenguaje escogido para la implementación de la infraestructura del lenguaje *GOLD* incluyendo su compilador y su entorno de desarrollo, por las siguientes razones:

- facilita la reutilización de código pues las versiones anteriores de *GOLD* fueron implementadas en *Java*;
- facilita el desarrollo de algunos procesos vitales como la visualización de grafos pues cuenta con una cantidad enorme de librerías; y
- facilita la implementación de la infraestructura del lenguaje *GOLD* pues cuenta con tecnologías avanzadas como *Eclipse* [7] y *Xtext* [6].

Por otro lado, se escogió *Java* como lenguaje de propósito general para traducir los programas escritos en *GOLD*, ya que:

- facilita la familiarización durante el proceso de aprendizaje en *GOLD*, pues *Java* es uno de los lenguajes de propósito general más populares en la actualidad;
- potencia la expresividad de *GOLD*, pues el usuario estaría en capacidad de usar cualquier función del *API* de *Java*, de alguna librería externa, o de alguna clase que implemente por su propia cuenta; y
- resuelve de manera directa algunos de los requerimientos expuestos en la sección §5.3, como la escalabilidad, la modularidad, la reusabilidad, la portabilidad, la familiaridad y la disponibilidad de compiladores y de herramientas.

6.1.2. *Eclipse*

Eclipse IDE [7] (*Eclipse Integrated Development Environment*) es un entorno de desarrollo libre y de código abierto empleado en la implementación de grandes proyectos de software en lenguajes como *Java*, que ofrece una gran cantidad de funcionalidades para facilitar la labor de programación. *Eclipse 3.7.1* (versión identificada con el nombre clave *Indigo*) es el ambiente de programación que se utilizó para implementar la infraestructura que da soporte al lenguaje *GOLD* y es la herramienta que deben utilizar los usuarios que deseen crear proyectos que incluyan código fuente escrito en *GOLD*.

Figura 6.2. *Logotipo de Eclipse [7].*

GOLD es distribuido como un *plug-in* que se debe instalar dentro de *Eclipse* para permitir la codificación y ejecución de programas escritos en *GOLD* aprovechando todas las funcionalidades inherentes a *Eclipse*. Usando *Eclipse SDK*

(*Eclipse Software Development Kit*) a través de la biblioteca de clases provista por *Eclipse JDT* [7] (*Eclipse Java Development Tools*) se efectúa la completa integración de *GOLD* con *Java* para que los desarrolladores puedan mezclar de forma transparente código fuente implementado en *Java* con código fuente implementado en *GOLD*, pudiendo invocar funciones *Java* desde archivos *GOLD* y funciones *GOLD* desde archivos *Java*.

6.1.3. *Xtext*

Xtext [6] es un *framework* de código abierto para el desarrollo de lenguajes de programación (ya sean lenguajes de propósito específico o lenguajes de propósito general), capaz de crear automáticamente el analizador léxico y sintáctico del lenguaje, un metamodelo de clases para representar los elementos sintácticos del lenguaje mediante una estructura arbórea denominada *Abstract Syntax Tree* (*AST*), y un entorno de desarrollo sofisticado [6] basado en *Eclipse* que incluye un editor de texto especializado y funcionalidades como la coloración de la sintaxis (*syntax highlighting*), el indentamiento automático del código fuente (*code formatting*), el emparejamiento de paréntesis (*bracket matching*), la validación de la sintaxis resaltando los errores de compilación en tiempo de desarrollo (*code validation*), el despliegue de ayudas de contenido (*content assist*), el completado automático de código (*code completion*) y la navegación sobre el esquema que describe la estructura semántica de un programa (*outline view*). Con la ayuda de *Xtext* se puede implementar toda la infraestructura relacionada con un nuevo lenguaje de programación, configurando los diferentes aspectos del lenguaje a través de los mecanismos ofrecidos por *Xtext* [6]. Aunque los módulos generados automáticamente por *Xtext* estén diseñados para trabajar en conjunto con *Eclipse*, ‘‘los componentes del lenguaje relacionados con su compilador son independientes de *Eclipse* y pueden ser usados en cualquier ambiente *Java*’’.

Figura 6.3. *Logotipo de Xtext [6].*

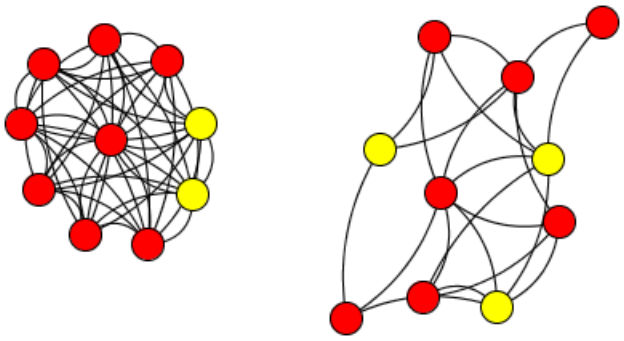
Xtext (versión 2.2.1) es el *framework* con el que se desarrolló la infraestructura relacionada con el lenguaje de programación *GOLD*, apoyando de forma automática la creación del *plug-in* de *Eclipse* que reúne el analizador léxico y sintáctico del lenguaje, el metamodelo *AST* que actúa como modelo semántico [49], el analizador semántico que traduce código *GOLD* a código *Java*, y el entorno de desarrollo dotado con las características enumeradas en el párrafo anterior y con todas las funcionalidades ya suministradas por *Eclipse*. De esta manera se estarían atacando muchos de los requerimientos no funcionales impuestos sobre el lenguaje en el capítulo §5. Por último, falta decir que el analizador sintáctico de *GOLD* es generado automáticamente por *Xtext* usando *ANTLR* [57] (*ANother Tool for Language Recognition*), que ya se encuentra integrado dentro de *Xtext*.

6.2. Librerías externas

6.2.1. *JUNG*

JUNG 2 [21] (*Java Universal Network/Graph Framework*, versión 2.0.1) es una librería *Java* (previamente descrita en la sección §3.3.4) que ofrece un *API* extensible para el modelado, análisis y visualización de grafos. Contiene un potente visualizador altamente configurable con el que se pueden dibujar grafos, permitiendo cambiar el estilo de los nodos y de los arcos, así como el algoritmo utilizado para ubicar los nodos dentro de la superficie de dibujo. La librería provee funciones para alterar el aspecto de los grafos en tiempo de ejecución, lo que facilita la depuración y la animación paso a paso de cualquier algoritmo sobre grafos de forma interactiva.

Figura 6.4. *Algunos grafos de ejemplo dibujados con JUNG [21].*

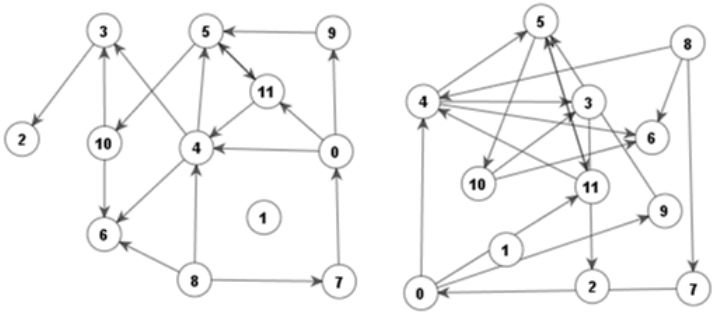


Uno de los aspectos más interesantes de la herramienta es que proporciona varios algoritmos de ubicación de nodos en la superficie de dibujo (*graph layout algorithms*) para renderizar [†1] los grafos de manera que se perciban agradables a la vista, incluyendo los siguientes:

- *Circle Layout*. Ubica los nodos del grafo espaciándolos uniformemente alrededor de un círculo.
- *KK Layout*. Ubica los nodos del grafo usando el algoritmo de Tomihisa Kamada y Satoru Kawai [58].
- *FR Layout*. Ubica los nodos del grafo usando el algoritmo de Thomas Fruchterman y Edward Reingold [59].
- *Spring Layout*. Ubica los nodos del grafo a través de un algoritmo que aplica determinados principios de la física para estabilizar los nodos en sus posiciones finales. Los arcos son tratados como resortes que intentan acercar sus nodos adyacentes y los vértices son tratados como cargas eléctricas que intentan repelerse. Ejecutando una simulación física en donde se utiliza intensivamente la ley de Hooke para los resortes y la ley de Coulomb para las cargas eléctricas, los nodos terminan ubicándose en sus posiciones definitivas después de cierta cantidad de iteraciones.

Figura 6.5. *Un grafo de ejemplo dibujado con dos algoritmos distintos para ubicar sus nodos.*

(a) *Los nodos del grafo se ubicaron con el algoritmo de Kamada-Kawai.* **(b)** *Los nodos del grafo se ubicaron deliberadamente al azar.*



JUNG permite la configuración de los parámetros que gobiernan el comportamiento de todos los algoritmos de *layout* proporcionados; por ejemplo, admite la parametrización de las constantes que afectan la fuerza con la que se encogen los resortes y con la que se repelen las cargas eléctricas en el algoritmo *Spring Layout*. Tanto *Spring Layout* como *FR Layout* son algoritmos basados en los principios básicos físicos de las fuerzas (véase la referencia

[1] El verbo *renderizar* es un extranjerismo introducido a la jerga informática para referirse al proceso de construcción de la representación gráfica de una imagen partiendo de su representación matemática.

[54] para mayor información), que son aprovechados para renderizar grafos dispersos buscando que luzcan de forma agradable, minimizando el número de intersecciones entre los arcos. Si los grafos se dibujaran ubicando aleatoriamente sus nodos entonces se terminarían cruzando los arcos indiscriminadamente, lo que dificultaría en gran medida la lectura del grafo.

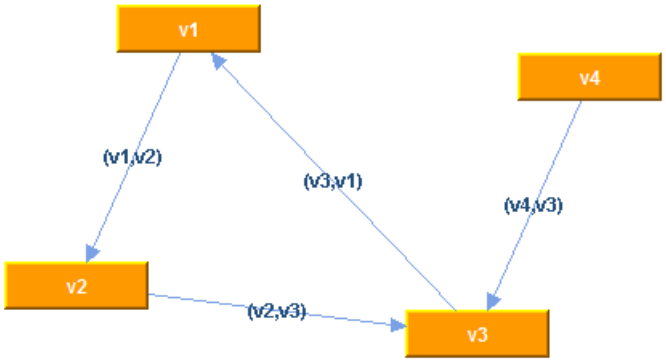
Además, *JUNG* provee estructuras de datos versátiles para representar grafos dirigidos, grafos no dirigidos e hipergrafos, y suministra implementaciones de referencia para algunos problemas típicos sobre grafos, incluyendo el problema de la ruta más corta. Pese a que *JUNG* es una excelente librería para quienes trabajan sobre *Java*, posee el mismo problema de todas las librerías de grafos que intentan enriquecer un lenguaje de propósito general: el código que implemente el usuario sigue siendo código escrito en el lenguaje de alto nivel, en este caso *Java*.

JUNG se está usando en *GOLD* para cumplir con todos los requerimientos relacionados con la visualización de grafos, proporcionando al desarrollador un mecanismo ideal para que pueda renderizar sus grafos y animar gráficamente la operación de sus algoritmos. A través de *JUNG* el programador puede configurar fácilmente los atributos visuales de los nodos y de los arcos de los grafos, así como el algoritmo utilizado para ubicar sus vértices en la superficie de dibujo. Además, *GOLD* provee adaptadores y funciones especiales para transformar grafos representados en *GOLD* en grafos representados con las estructuras de datos provistas por *JUNG*, y viceversa.

6.2.2. *JGraphT*

JGraphT [22] (versión 0.8.2) es una librería *Java* (previamente descrita en la sección §3.3.5) que suministra una colección de clases y de algoritmos diseñados para trabajar sobre teoría de grafos, permitiendo visualizar distintos tipos de grafo mediante la librería *JGraph* [27].

Figura 6.6. *Un grafo de ejemplo visualizado en JGraphT [22] a través de JGraph [27].*



Según sus creadores, la librería es fuertemente tipada (*type-safe*) y respeta el esquema de genericidad de *Java* [22]. Se distribuye bajo los términos de la licencia *LGPL*, su código fuente está disponible para la descarga, tiene documentación clara y completa en formato *Javadoc*, y es fácil de usar y de extender. *JGraphT* incluye implementaciones de referencia claras y eficientes que resuelven algunos problemas clásicos sobre grafos como el problema de la ruta más corta (mediante el algoritmo de Dijkstra, el algoritmo de Bellman-Ford y el algoritmo de Floyd-Warshall), el problema del árbol de expansión mínimo (mediante el algoritmo de Kruskal), el problema del flujo máximo (mediante algoritmo de Edmonds-Karp), el ordenamiento topológico, la clausura transitiva, y la detección de ciclos eulerianos y hamiltonianos.

Además de proveer clases para representar grafos dirigidos, grafos no dirigidos e hipergrafos, *JGraphT* brinda implementaciones útiles de algunas estructuras de datos avanzadas como:

- *Disjoint-set forests* [1], que implementa conjuntos disyuntos (*disjoint-set data structure*).
- *Fibonacci heaps* [1], que implementa montones (*heaps*).

JGraphT es de las librerías más potentes que están disponibles en la actualidad para manipular grafos. La biblioteca de estructuras de datos provista por *GOLD* importa las clases `org.jgrapht.util.FibonacciHeap` y `org.jgrapht.util.FibonacciHeapNode` de *JGraphT* para dar soporte a la implementación de montones (*heaps*) a través de *Fibonacci heaps* [1]. A pesar de que ninguna otra clase de *JGraphT* se utiliza en *GOLD*, el usuario podría importar la librería para explotarla en *GOLD* según su conveniencia.

6.2.3. Implementaciones de referencia de Cormen et al.

El disco compacto distribuido con el texto guía *Introduction to Algorithms* de Thomas Cormen et al. [1] contiene una librería *Java* (previamente descrita en la sección §3.3.6) que suministra implementaciones de referencia para la mayoría de las estructuras de datos y algoritmos presentados en la segunda edición del mencionado libro. La biblioteca de estructuras de datos provista por *GOLD* importa las clases `BinomialHeap`, `DynamicSetElement`, `KeyUpdateException` y `MergeableHeap` del paquete com.mhhe.clrs2e para dar soporte a la implementación de montones (*heaps*) a través de *Binomial heaps* [1]. A pesar de que ninguna otra clase del paquete `com.mhhe.clrs2e` se utiliza en *GOLD*, el usuario podría importar la librería para aprovecharla en *GOLD*.

6.2.4. *Apfloat*

Apfloat [53] (versión 1.6.2) es una librería *Java* de alto rendimiento que provee tipos de datos para la representación de números, y rutinas diseñadas para el desarrollo de operaciones aritméticas de precisión arbitraria sobre éstos. *Apfloat* se usa en *GOLD* para enriquecer el lenguaje con tipos de datos sofisticados y algoritmos veloces para la computación de operaciones que involucran números arbitrariamente grandes. De esta manera se busca fomentar la manipulación de valores pertenecientes a conjuntos típicos como los naturales ($\mathbb{N}$), los enteros ($\mathbb{Z}$), los racionales ($\mathbb{Q}$), los reales ($\mathbb{R}$) y los complejos ($\mathbb{C}$) sin restricciones artificiales como las impuestas por el número de bits usados en la representación interna de los tipos primitivos de *Java* y sin problemas incómodos como el desbordamiento (*overflow*) o la propagación de inestabilidad numérica causada por falta de precisión. Se prefirió usar *Apfloat* en vez de las clases `java.math.BigInteger` y `java.math.BigDecimal` de *Java* porque *Apfloat* implementa algoritmos más eficientes para el desarrollo de las operaciones y cuenta con muchas más funciones aritméticas que no están presentes en el *API* estándar de *Java* como por ejemplo el cálculo de la raíz cuadrada de un número flotante de precisión arbitraria con una cantidad ingente de dígitos decimales.

Tabla 6.1. *Clases provistas por Apfloat [53] para representar números de precisión arbitraria.*

Conjunto	Símbolo	Clase
Naturales	$\mathbb{N}$	`org.apfloat.Apint`
Enteros	$\mathbb{Z}$	`org.apfloat.Apint`
Racionales	$\mathbb{Q}$	`org.apfloat.Aprational`
Reales	$\mathbb{R}$	`org.apfloat.Apfloat`
Complejos	$\mathbb{C}$	`org.apfloat.Apcomplex`

Capítulo 7

Diseño

En este capítulo se presenta el diseño de la sintaxis y de la semántica del lenguaje *GOLD* (i.e., *GOLD 3*) teniendo como base el marco teórico estudiado en el capítulo §4 y como inspiración los requerimientos enumerados en el capítulo §5, exponiendo todas las decisiones que influenciaron la etapa de desarrollo. Se recomienda revisar con detalle el capítulo §4 antes de leer este capítulo pues a lo largo del diseño se hace alusión a términos especializados que fueron definidos previamente en el marco teórico.

GOLD (*Graph Oriented Language Domain* por sus siglas en inglés) es un lenguaje de programación imperativo que puede ser estudiado como un lenguaje de propósito general (*GPL*) diseñado para facilitar la escritura de rutinas que utilizan intensivamente objetos matemáticos expresados en la notación acostumbrada en los libros de texto, y a la vez como un lenguaje de propósito específico (*DSL*) diseñado para facilitar la escritura de algoritmos sobre estructuras de datos avanzadas como árboles, grafos y autómatas a través de una sintaxis muy cercana al pseudocódigo trabajado en la referencia *Introduction to Algorithms* de Thomas Cormen et al. [1]. Esta ambivalencia que suena contradictoria a primera vista, porque en teoría un *DSL* no puede ser un *GPL* y viceversa, termina potenciando el lenguaje al permitir que sea usado como un lenguaje de propósito general para resolver problemas en una diversidad de dominios, o como un lenguaje de propósito específico para resolver problemas concentrados en el dominio de los grafos.

7.1. Pragmática

En esta sección se describen algunos aspectos generales del lenguaje *GOLD* relacionados con su pragmática, usando la terminología y conceptos básicos descritos en el marco teórico (véase el capítulo §4).

7.1.1. Clasificación

Es iluso pretender diseñar un *DSL* limitado que se concentre únicamente en el dominio de la algorítmica sobre grafos, simplemente porque para implementar algoritmos eficaces y eficientes sobre éstos se necesita poder escribir comandos de alto nivel como condicionales y ciclos, poder declarar variables y procedimientos, y tener la posibilidad de usar estructuras de datos como listas, pilas, colas y asociaciones llave-valor. Si no se provee alguna de las características mencionadas, el lenguaje no sería lo suficientemente expresivo, impidiendo la programación de algoritmos clásicos sobre grafos. Por lo tanto, *GOLD* debe ser concebido como un *GPL* que sirva para resolver problemas sobre cualquier dominio de aplicación, pudiendo actuar como un *DSL* sobre el dominio de los grafos si se restringe su uso a tal estructura de datos.

Según Fowler [49], la frontera que separa los *DSLs* de los *GPLs* es borrosa, dificultando la clasificación de algunos lenguajes dentro de una de las dos categorías exclusivamente. Analizado como un lenguaje que sirve para cualquier propósito concebible, *GOLD* sería sin duda alguna un *GPL* al igual que *Java*, *C* y *C++* (entre otros), pudiendo ser diseñado a la luz de la teoría existente para los lenguajes de propósito general. Sin embargo, ‘‘un uso particular de

un lenguaje puede ponerlo en uno u otro lado de la frontera de los *DSLs*'' [49]. Por lo tanto, si se restringe *GOLD* para que sólo sea usado en la definición y manipulación de grafos (y tal vez otras estructuras de datos específicas), entonces puede ser considerado como un *DSL* en todo el sentido de la palabra. En todo caso, aún si *GOLD* no fuera clasificado como un *DSL*, sería entonces un *GPL* que si es utilizado de una forma particular (concretamente para resolver problemas sobre grafos y otras estructuras de datos relacionadas), daría nacimiento a un *DSL interno* [49] cuyo dominio particular serían los grafos. Sabiendo que *GOLD* es un *GPL*, no vale la pena discutir si también es o no un *DSL*: simplemente, *GOLD* será tratado como un *GPL* o como un *DSL* dependiendo de si va a ser usado como un lenguaje multipropósito o como un lenguaje concentrado en el dominio de los grafos, respectivamente. Analizar *GOLD* de ambas maneras enriquecería su estudio en vez de complicarlo, dejando a un lado el dilema propuesto por Fowler [49] al intentar establecer un límite claro entre los *DSLs* y los *GPLs*:

> *''"Domain-specific language" is a useful term and concept, but one that has very blurred boundaries. Some things are clearly DSLs, but others can be argued one way or the other. The term has also been around for a while and, like most things in software, has never had a very firm definition. [...] Many people use a literal definition of DSL as a language for a specific domain. But literal definitions are often incorrect: We don't call coins "compact disks" even though they are disks that rather more compact than those disks that we do apply the term to. [...] One common indicator of a DSL is that it isn't Turing-complete. DSLs usually avoid the regular imperative control structures (conditions and loops), don't have variables, and can't define subroutines. It's at this point where many people will disagree with me, using the literal definition of a DSL to argue that [some] languages [...] should be counted as a DSL. The reason I put a strong emphasis on limited expressiveness is that it is what makes the distinction between DSLs and general-purpose languages useful. The limited expressiveness gives DSLs different characteristics, both in using them and in implementing them. This leads to a different way of thinking about DSLs compared to general-purpose languages. If this boundary isn't fuzzy enough, let's consider XSLT. XSLT's domain focus is that of transforming XML documents, but it has all the features one might expect in a regular programming language. In this case, I think the way it is used matters more than the language itself. If XSLT is being used to transform XML, then I would call it a DSL. However, if it's being used to solve the eight queens problem, I would call it a general-purpose language. A particular usage of a language can put it on either side of the DSL line.'' [49]*

Si es visto como un *GPL* diseñado para resolver problemas sobre una gran cantidad de dominios, *GOLD* debe ayudar a los desarrolladores a formular y comunicar sus ideas claramente [40], y debe ser universal, intuitivo, e implementable en un computador [40]. Bajo este punto de vista, *GOLD* es un lenguaje de programación imperativo de alto nivel y de propósito general, que debe ser definido en términos de su sintaxis, su semántica y su pragmática. Al ser un lenguaje diseñado bajo el paradigma de la programación imperativa, los programas escritos en *GOLD* serían secuencias de instrucciones de control que describen la operación de un determinado algoritmo [40].

Por otro lado, si es visto como un *DSL* diseñado para definir y manipular grafos, *GOLD* debe ser un lenguaje de programación con expresividad limitada que esté enfocado exclusivamente sobre el dominio de los grafos. Bajo este punto de vista, *GOLD* es un lenguaje de propósito específico externo [49] cuya sintaxis está separada de cualquier otro lenguaje de programación, especialmente *Java*.

Si es usado como *GPL*, *GOLD* facilitaría la escritura de rutinas que utilizan intensivamente objetos matemáticos expresados en la notación acostumbrada en los libros de texto. En contraparte, si es usado como *DSL*, *GOLD* facilitaría la escritura de algoritmos sobre estructuras de datos avanzadas como árboles, grafos y autómatas a través de una sintaxis muy cercana al pseudocódigo trabajado en la referencia *Introduction to Algorithms* de Thomas Cormen et al. [1].

7.1.2. Procesamiento

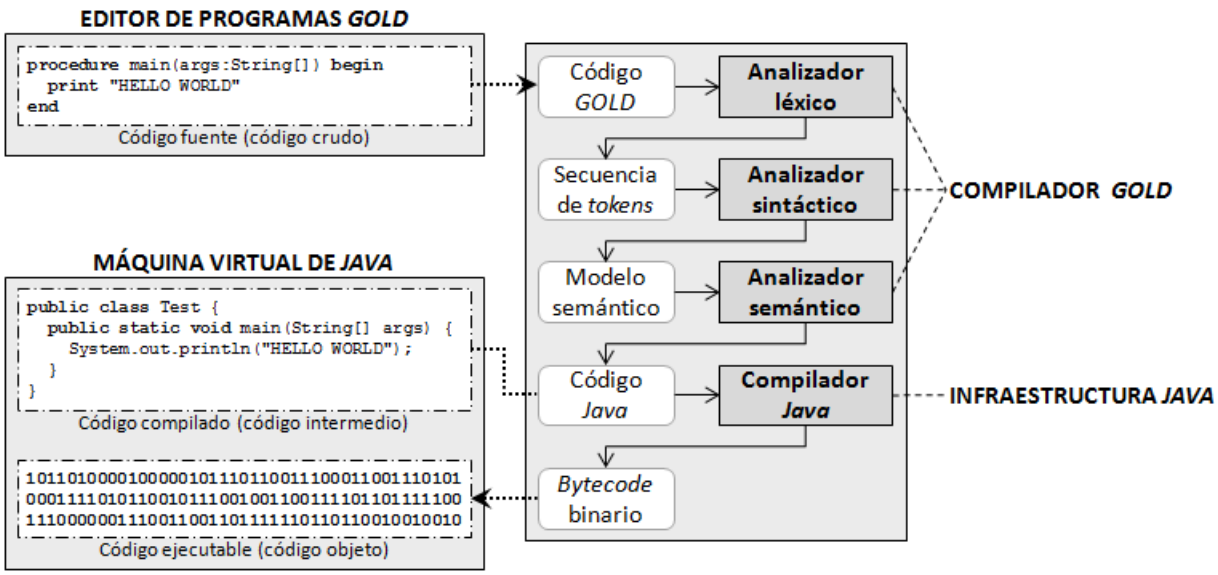
Al contrario de su predecesor (*GOLD+*), el lenguaje de programación *GOLD* es sometido a un proceso de *compilación* y no de *interpretación*. Siguiendo el patrón *syntax-directed translation* de Fowler [49], se realiza un proceso de compilación que traduce los programas escritos en *GOLD* en código fuente codificado en *Java* (véase la sección §6.1.1). Lo anterior hace posible que desde cualquier programa *GOLD* se puedan usar librerías externas como el *API* estándar de *Java*, permitiendo la manipulación de una multitud de estructuras de datos. Así pues, *GOLD* puede ser integrado en grandes proyectos de software donde se necesite manipular exhaustivamente objetos matemáticos formales como los grafos y otras estructuras de datos.

Un mecanismo completamente distinto de procesamiento fue llevado a cabo en *GOLD+*, la anterior versión de *GOLD* que fue diseñada por Diana Mabel Díaz en su tesis de maestría titulada *GOLD+: lenguaje de programación para la manipulación de grafos: extensión de un lenguaje descriptivo a un lenguaje de programación* [4] (véase la sección §2.3). En *GOLD+* los programas son sometidos a un proceso de interpretación que ejecuta cada instrucción en una máquina abstracta restringida, lo que limita enormemente la potencia del lenguaje. Cambiando el proceso de interpretación por uno de compilación usando *Java* como lenguaje *anfitrión*, se puede aprovechar toda su infraestructura, incluyendo su librería estándar, su compilador, su máquina virtual, y uno de sus entornos de desarrollo más reconocidos: *Eclipse* [7] (véase la sección §6.1.2). Al descartar el proceso de interpretación seguido en *GOLD+*, reemplazándolo por un proceso de compilación, se ahorra una gran cantidad de trabajo (al no tener que implementar la máquina abstracta) y a la vez se eleva el potencial del lenguaje (al heredar todas las capacidades del lenguaje anfitrión).

El compilador de *GOLD* está conformado por varios componentes importantes que son responsables de realizar las etapas de análisis y síntesis del lenguaje [41] (véase la figura 7.1):

- *Analizador léxico* (*Lexer*). Transforma el código fuente de un programa *GOLD* en una secuencia de *tokens*.
- *Analizador sintáctico* (*Parser*). Procesa la secuencia de *tokens* generada por el analizador léxico para construir un árbol de derivación cuya estructura imita las reglas de producción de la gramática *BNF*, que termina siendo utilizado para poblar el modelo semántico.
- *Analizador semántico* (*Semantic analyzer*). Recorre el modelo semántico para realizar la traducción definitiva a código *Java*, que es convertido a *bytecode* binario por el compilador estándar de *Java* (`javac`), que posteriormente puede ser ejecutado en la *Máquina Virtual de Java* (*JVM*: *Java Virtual Machine*).

Figura 7.1. *Proceso de compilación de programas escritos en GOLD 3.*



Al someter los programas *GOLD* a un proceso de compilación que los transforma en archivos *Java*, se faculta a los programadores para que mezclen en sus proyectos código *GOLD* con código *Java*, enriqueciendo la aplicación de ambos lenguajes puesto que cada uno se beneficiaría de las capacidades del otro. A grandes rasgos, *GOLD* explotaría las librerías disponibles para *Java* (en especial su *API* estándar) y los conceptos de la programación orientada a objetos inherentes a *Java*; por otro lado, *Java* aprovecharía la versatilidad de *GOLD* para expresar y manipular objetos matemáticos de diversos dominios con una sintaxis cercana al pseudocódigo.

GOLD es un *DSL* externo [49] porque su sintaxis, descrita más adelante en la sección §7.2, está separada del lenguaje de programación principal con el que se está trabajando, que es precisamente *Java*. Dado que la sintaxis de *GOLD* define un *GPL* completamente distinto a cualquier lenguaje de programación conocido, entonces no es subconjunto de ningún otro lenguaje existente, aunque se esté usando *Java* como el lenguaje anfitrión en el que se expresa la salida del proceso de compilación y se permita usar clases *Java* en los comandos de *GOLD*. Así pues, *GOLD* no puede ser considerado un *DSL* interno porque no es una forma particular de usar *Java*.

El diseño propuesto para *GOLD* constituye literalmente una aplicación de los patrones *Piggyback* y *Source-to-source transformation* de Spinellis [50], donde el lenguaje anfitrión es *Java*. No aplica el patrón *Language extension* [50] puesto que *GOLD* no extiende la sintaxis ni la semántica de ningún lenguaje conocido. Tampoco aplica el patrón *Language specialization* [50] porque *GOLD* no es subconjunto de ningún lenguaje existente.

7.2. Sintaxis

La sintaxis del lenguaje de programación *GOLD* se describe en la sección §A.1.1 a través de una gramática independiente del contexto escrita en la notación *EBNF* [5], usando la variante definida por la *W3C* [46]. Para cada elemento sintáctico del lenguaje se provee la regla que lo define, una minuciosa descripción semántica indicando la forma en la que opera, y algunos ejemplos que ilustran su utilización. Más adelante, en la sección §7.3 se presenta con detalle la semántica de los principales componentes sintácticos de *GOLD*.

El diseño de la sintaxis de *GOLD* está basado en la notación de los pseudocódigos trabajados en la segunda edición del libro *Introduction to Algorithms* de Thomas Cormen et al. [1], pues es un lenguaje imperativo de alto nivel cercano al ser humano que permite la descripción de algoritmos computacionales de una forma sencilla, dirigida a mejorar la legibilidad y la facilidad de escritura de los programas. La notación del libro de Cormen et al. [1] es idónea para satisfacer los objetivos propuestos en este proyecto porque está ideada para simplificar la escritura de procedimientos estructurados en términos de objetos matemáticos que están expresados a través del formalismo comúnmente utilizado en dominios como el cálculo proposicional, el cálculo de predicados, la teoría de conjuntos y las matemáticas discretas. Además, se sabe que la notación es lo suficientemente expresiva como para permitir la manipulación de una diversidad de estructuras de datos, porque precisamente en el mencionado libro se utiliza intensivamente para desarrollar la algorítmica sobre éstas.

El lenguaje subyacente a los pseudocódigos trabajados en el texto de Cormen et al. [1] (en adelante, abreviado *lenguaje de Cormen et al.*) se adaptó minuciosamente con el objetivo de facilitar la implementación del procesador del lenguaje (especialmente su compilador y su *IDE*), promover su inmersión dentro de la infraestructura de *Java*, permitir la utilización de clases *Java* en sus instrucciones, y acercar su definición al paradigma de la programación orientada a objetos. El nuevo lenguaje obtenido después de aplicar estas modificaciones es el que se denominó *GOLD*. De acuerdo con la terminología de Spinellis [50], se puede decir que *GOLD* es una extensión del lenguaje de Cormen et al., diseñado a través de la aplicación del patrón *Language extension*. Sin embargo, usando el sentido estricto de la definición, se tendría que *GOLD* no es una extensión del lenguaje de Cormen et al. porque algunos elementos sintácticos fueron alterados en beneficio de la legibilidad, de la ortogonalidad [41] y de la uniformidad, eliminando aquellos aspectos informales característicos de los pseudocódigos. Por ejemplo, se abolió el uso de instrucciones dadas en lenguaje natural, la omisión intencional de detalles esenciales, el uso de notación matemática inadecuada, y la indentación de código fuente para indicar implícitamente la estructura de bloques [40] del programa. De esta forma, se logró diseñar un lenguaje potente pensado para que tanto humanos como máquinas puedan leerlo fácilmente, satisfaciendo la mayoría de los criterios sintácticos generales establecidos por Pratt y Zelkowitz [41]: facilidad de lectura (*readability*), facilidad de escritura (*writeability*), facilidad de traducción (*ease of translation*) y ausencia de ambigüedad (*lack of ambiguity*). El único criterio que no fue mencionado es el de facilidad de verificación (*ease of verifiability*), que será discutido en la sección §7.3.2.

Vale la pena advertir que la notación de *GOLD* está basada en la segunda edición del libro *Introduction to Algorithms* [1] (2001), no en la tercera [55] (2009). Esto pues la sintaxis de la tercera edición se aleja de las bondades del pseudocódigo para intentar parecerse a algunos lenguajes modernos de programación orientados a objetos como *Java* y *C++*, distanciándose ligeramente de la notación matemática tradicional. Algunas diferencias sintácticas de la tercera edición con respecto a la segunda son:

- Las asignaciones se escriben en la forma '`x=E`', no en la forma '`x←E`'.
- El operador de igualdad es el símbolo ==, no el símbolo =.
- Los intercambios (*swaps*) se escriben en la forma '`swap x with y`', no en la forma '`exchange x↔y`'.
- Muchas palabras irrelevantes (*noise words*) fueron eliminadas, perturbando la legibilidad (e.g., '`if B S`' en lugar de '`if B then S`' y '`while B S`' en lugar de '`while B do S`').

Seguramente los programadores estarían más cómodos con la notación de la tercera edición mientras que los matemáticos estarían más cómodos con la de la segunda. Para complacer a ambos tipos de usuario, *GOLD* proporciona reglas de producción redundantes para poder denotar un mismo comando (o una misma expresión) de varias formas distintas, facilitando la escritura de programas sin penalizar la diversidad de estilos de codificación.

7.2.1. Elementos léxicos

Para permitir el uso de símbolos matemáticos sofisticados se escogió como codificación de caracteres el estándar *UTF-8* (*8-bit UCS Transformation Format*), que es capaz de representar todos los símbolos pertenecientes al conjunto de caracteres *Unicode*, a diferencia del formato *ISO-8859-1*, que únicamente incluye letras del alfabeto latino y algunos símbolos especiales. De esta manera, el alfabeto del lenguaje estaría definido como el conjunto de 65536 caracteres *Unicode*, codificados con el formato *UTF-8*. Todos los símbolos especiales utilizados en el lenguaje *GOLD* están descritos en la sección §A.5.1 de la documentación técnica.

Los diferentes elementos léxicos (*tokens*) de *GOLD* corresponden unívocamente con los símbolos terminales de su gramática y pueden ser clasificados en las siguientes categorías (de acuerdo con Pratt y Zelkowitz [41]):

- *Identificadores* (*Identifiers*). Son cadenas de caracteres de longitud arbitraria que sirven para identificar cualquier entidad de un programa *GOLD*, incluyendo tipos, variables y procedimientos. Concretamente, un identificador en *GOLD* debe comenzar con una letra (del alfabeto latino o griego) o con un guión bajo (*underscore* (_)), que puede estar seguida por cero o más letras (del alfabeto latino o griego), guiones bajos (*underscores* (_)), dígitos numéricos (0, 1, …, 9), subíndices numéricos ($_0$, $_1$, …, $_9$) o símbolos prima (*prime symbol* ($'$)). Además, un identificador en *GOLD* puede comenzar por el signo pesos (`$`) para escaparlo (*to escape*) cuando haya conflicto con alguna palabra reservada definida con el mismo nombre, haciendo posible la declaración e invocación de procedimientos y variables cuyo nombre coincida con alguna palabra reservada (e.g., el identificador `$print` escapa la palabra reservada `print`).
- *Símbolos de operador* (*Operator symbols*). Son símbolos para representar operaciones lógico-aritméticas, que deben estar compuestos por uno o más caracteres *Unicode*. Los operadores (unarios y binarios) en *GOLD* son denotados con secuencias de símbolos especiales (e.g., `&&`, ∧, `||`, ∨, `!`, ¬, `==`, ≡) o con identificadores alfanuméricos (e.g., `and`, `or`, `not`, `eqv`).
- *Palabras reservadas* (*Reserved words*). Son cadenas de texto alfanuméricas usadas como partes fijas de la sintaxis de las distintas instrucciones, que no pueden ser declaradas ni usadas por los programadores como identificadores. Todas las palabras clave (*keywords*) de *GOLD* son también palabras reservadas (*reserved words*), y viceversa (por definición). Muchas de las palabras reservadas de *GOLD* denotan funciones (e.g., `sin`) y operaciones (e.g., `and`). En orden alfabético, las palabras reservadas de *GOLD* son las siguientes:

abort, abs, acos, acosh, and, as, asin, asinh, assert, atan, atanh, begin, boolean, break, by, byte, call, case, cbrt, char, continue, cos, cosh, default, div, do, double, downto, each, else, elseif, end, eqv, error, exchange, exp, false, FALSE, finalize, float, for, function, gcd, if, import, in, include, int, lcm, ln, log, long, max, min, mod, new, nil, NIL, not, null, NULL, or, package, pow, print, procedure, repeat, return, root, short, sin, sinh, skip, sqrt, SuppressWarnings, swap, switch, tan, tanh, then, throw, to, true, TRUE, until, using, var, void, while, whilst, with, y xor.

- *Comentarios* (*Comments*). Son cadenas de texto que sirven para la inserción de comentarios y de documentación técnica en los programas, sin formar parte de la semántica de sus instrucciones. En *GOLD* se permite la inserción de comentarios delimitándolos entre las cadenas `/*` y `*/`, o encerrándolos entre dos *slashes* (`//`) y el siguiente cambio de línea. Para satisfacer la notación de la segunda edición del libro de Cormen et al. [1], también se permite encerrar los comentarios entre el símbolo ⧐ y el siguiente cambio de línea (no se usa ▷ porque ya denota una operación: el *append* [12]).

- *Blancos* (*Blanks (spaces)*). Son caracteres ocultos como los espacios (‘ ’), las tabulaciones (‘`\t`’), los retornos de carro (‘`\r`’) y los cambios de línea (‘`\n`’), que no tienen ninguna semántica en *GOLD*, salvo dentro de los literales que corresponden a cadenas de texto o caracteres.

- *Delimitadores* (*Delimiters*). Son caracteres usados en *GOLD* para ‘‘marcar el comienzo o el fin de alguna unidad sintáctica como una sentencia o una expresión’’ [41] (e.g., `,`, `:`, `|`).

- *Paréntesis* (*Brackets*). Son delimitadores que vienen de a parejas, donde el primero de éstos se denomina *paréntesis de apertura* y el segundo se denomina *paréntesis de cierre*. Los paréntesis que se pueden usar en *GOLD* están descritos en la tabla A.12 y son: $(\cdots)$ (paréntesis circular), $[\cdots]$ (corchetes para acceder posiciones de un arreglo), $[\![\cdots]\!]$ (corchetes blancos para expresar arreglos), $\langle \cdots \rangle$ (paréntesis angular para expresar secuencias), $\{ \cdots \}$ (llaves para expresar conjuntos), $\{\!| \cdots |\!\}$ (llaves blancas para expresar bolsas), $\lfloor \cdots \rfloor$ (piso), $\lceil \cdots \rceil$ (techo), $| \cdots |$ (valor absoluto / cardinalidad de una colección).

El lenguaje es sensible a las mayúsculas (*case sensitive*), haciendo que la escritura de caracteres en mayúsculas o minúsculas sea relevante en los identificadores y palabras reservadas. En la definición de la gramática de *GOLD* en formato *EBNF* (véase la sección §A.1.1), todos los símbolos terminales están identificados con nombres escritos en mayúsculas (e.g., `ML_COMMENT`) y todos los símbolos no terminales están identificados con nombres que alternan mayúsculas con minúsculas (e.g., `GoldProgram`). Algunos símbolos terminales representan conjuntos de caracteres especiales del estándar *Unicode* y el resto representan elementos léxicos del lenguaje.

Tabla 7.1. *Símbolos terminales de la gramática de GOLD 3.*

Nombre	Descripción
`ALL`	Todos los caracteres *Unicode* en el rango `0x0000-0xFFFF`.
`LF`	Cambio de línea (*new line*): ‘`\n`’ (`0x000A`).
`CR`	Retorno de carro (*carriage return*): ‘`\r`’ (`0x000D`).
`TAB`	Tabulación (*tab*): ‘`\t`’ (`0x0009`).
`LETTER`	Las letras del alfabeto latino, tanto las mayúsculas del rango *Unicode* `0x0041-0x005A` (‘`A`’, ‘`B`’, ‘`C`’, …, ‘`Z`’) como las minúsculas del rango *Unicode* `0x0061-0x007A` (‘`a`’, ‘`b`’, ‘`c`’, …, ‘`z`’). Además, incluye las letras del alfabeto griego, tanto las mayúsculas del rango *Unicode* `0x0391-0x03A9` (‘`A`’, ‘`B`’, ‘Γ’, …, ‘Ω’) como las minúsculas del rango *Unicode* `0x03B1-0x03C9` (‘α’, ‘β’, ‘γ’, …, ‘ω’).
`DIGIT`	Dígitos numéricos del rango *Unicode* `0x0030-0x0039` (‘`0`’, ‘`1`’, ‘`2`’, …, ‘`9`’).
`SUBINDEX`	Subíndices numéricos (*numerical subscripts*) del rango *Unicode* `0x2080-0x2089` (‘$_0$’, ‘$_1$’, ‘$_2$’, …, ‘$_9$’).
`HEX_DIGIT`	Dígitos hexadecimales de los rangos *Unicode* `0x0030-0x0039` (‘`0`’, ‘`1`’, ‘`2`’, …, ‘`9`’), `0x0041-0x0046` (‘`A`’, ‘`B`’, ‘`C`’, ‘`D`’, ‘`E`’, ‘`F`’), y `0x0061-0x0066` (‘`a`’, ‘`b`’, ‘`c`’, ‘`d`’, ‘`e`’, ‘`f`’).
`HEX_CODE`	Códigos hexadecimales para representar caracteres *Unicode* en la forma `uXXXX`, donde `XXXX` son cuatro dígitos hexadecimales (`HEX_DIGIT`).

`ID`	Identificadores (*identifiers*) construidos de acuerdo con la regla descrita anteriormente: comienzan con una letra (`LETTER`) o con un guión bajo ('_'), y sigue una secuencia posiblemente vacía compuesta por letras (`LETTER`), guiones bajos ('_'), dígitos numéricos (`DIGIT`), subíndices numéricos (`SUBINDEX`) o símbolos prima ('). Para escapar un identificador se puede anteponer el signo pesos (`$`).
`QN`	Nombres calificados (*qualified names*) compuestos por uno o más identificadores (`ID`) separados por puntos (e.g., `Integer`, y `java.util.LinkedList`), que pueden estar seguidos por una arroba ('`@`') y uno o más identificadores (`ID`) separados por arrobas (e.g., `java.util.Map@Entry`, y `package.Foo@Goo@Hoo`). El signo arroba ('`@`') sirve para mencionar *clases anidadas* [60] (*nested classes*) declaradas en *Java*.
`CONSTANT`	Constantes básicas, incluyendo los valores booleanos (`TRUE`, `true`, `FALSE`, `false`), el apuntador nulo (`NIL`, `nil`, `NULL`, `null`), el conjunto vacío ($\varnothing$), el conjunto universal ($\mathbf{U}$), la secuencia vacía (ε), la cadena de texto vacía (λ) y el infinito positivo (∞). Las constantes matemáticas están listadas en la tabla A.3.
`NUMBER`	Números enteros y números de punto flotante. Al valor numérico se le puede poner al final una letra para interpretarlo como un valor de tipo primitivo de *Java* (véase la tabla 8.9), de acuerdo con la siguiente convención: '`L`' o '`l`' para `long`, '`I`' o '`i`' para `int`, '`S`' o '`s`' para `short`, '`B`' o '`b`' para `byte`, '`D`' o '`d`' para `double`, '`F`' o '`f`' para `float`, y '`C`' o '`c`' para `char`. Adicionalmente, se puede usar la notación científica o exponencial adjuntando como sufijo la letra '`E`' (mayúscula o minúscula) concatenada con el exponente (e.g., `2.957E-5` denota el número `0.00002957`, y `2.957E5` denota el número `295700`).
`STRING`	Literales conformados por cadenas de texto de longitud arbitraria encerradas entre comillas dobles ('`"`'). Las secuencias de escape que se pueden usar dentro de las cadenas de texto están enumeradas en la tabla A.13.
`CHARACTER`	Literales conformados por un solo carácter encerrado entre comillas sencillas ('`'`'). Las secuencias de escape que se pueden usar para denotar caracteres especiales están enumeradas en la tabla A.13.
`JAVA_CODE`	Código fuente escrito en *Java*, delimitado entre las cadenas `/?` y `?/`.
`ML_COMMENT`	Comentarios multilínea (*multi-line comments*) delimitados entre las cadenas `/*` y `*/`.
`SL_COMMENT`	Comentarios de una sola línea (*single-line comments*) encerrados entre dos *slashes* (`//`) y el siguiente cambio de línea, o entre el símbolo ⧐ y el siguiente cambio de línea.
`WS`	Blancos (*whitespaces*), incluyendo caracteres ocultos como espacios ('` `'), tabulaciones ('`\t`'), retornos de carro ('`\r`') y cambios de línea ('`\n`').

Los símbolos terminales pueden definirse formalmente a través de expresiones regulares usando el patrón *regex table lexer* de Fowler [49]. Sin embargo, pueden describirse más cómodamente usando la notación *EBNF* [5] de la *W3C* [46] (véase la sección §A.1.1).

7.2.2. Tipos

Los tipos primitivos de *GOLD* se dividen en dos categorías:

1. *Conjuntos matemáticos básicos*. Comprende el tipo booleano ($\mathbb{B}$), los números naturales ($\mathbb{N}$), los números enteros ($\mathbb{Z}$), los números racionales ($\mathbb{Q}$), los números reales ($\mathbb{R}$) y los números complejos ($\mathbb{C}$).

2. *Tipos primitivos heredados de Java*. Comprende los ocho tipos primitivos de *Java*: `boolean` (valores booleanos), `char` (caracteres *Unicode*), `byte` (enteros de 8 bits), `short` (enteros de 16 bits), `int` (enteros de 32 bits), `long` (enteros de 64 bits), `float` (flotantes de 32 bits) y `double` (flotantes de 64 bits).

Cada uno de los tipos primitivos de *GOLD* es un conjunto de valores que está equipado con determinadas operaciones lógico-aritméticas, descritas en la tabla 5.4. Los conjuntos matemáticos básicos proveen tipos infinitos que deben ser implementados (en su mayoría) con números de precisión arbitraria, mientras que los tipos primitivos heredados de *Java* están limitados por el número de bits usados en su representación (véase la tabla 8.9).

Los tipos compuestos de *GOLD* engloban todas las clases *Java*, por ejemplo:

- clases pertenecientes a la librería estándar de *Java*, que se encuentran documentadas en su *API* (*Application Programming Interface*);

- clases pertenecientes a librerías externas empaquetadas en archivos *JAR* o distribuidas en archivos compilados con extensión *.class*;
- clases pertenecientes al usuario o a otras personas, cuya implementación esté disponible en código fuente en archivos con extensión *.java*; y
- clases pertenecientes a la librería de *GOLD*, que debe implementar las estructuras de datos que se van a proporcionar.

De esta manera, cualquier clase *Java* puede actuar como un tipo compuesto de *GOLD* cuyo identificador es un nombre calificado (*qualified name*) correspondiente al símbolo terminal `QN`, sabiendo que el espacio de nombres de un programa *GOLD* puede ser afectado por sentencias *import* (véase la sección §7.2.7). Adicionalmente, al igual que en *Java*, si `T` es un tipo cualquiera, entonces `T[]` es un tipo compuesto que denota un arreglo de elementos de tipo T. Todo lo anterior pone en evidencia que, para declarar nuevos tipos compuestos en *GOLD*, basta construir nuevas clases en *Java*. El potencial de crear nuevas clases, sumado a las clases ya existentes en el *API* estándar de *Java* y otras librerías, hace que *GOLD* pueda alimentarse de tipos compuestos especiales como arreglos, clases, cadenas de texto, tipos recursivos, y estructuras de datos en general.

Por otro lado, hay que tener en consideración que, cuando en *GOLD* se declara una variable, ésta se inicializa automáticamente con el *valor por defecto* asociado a su tipo: cero (0) para los tipos primitivos numéricos (i.e., $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, `byte`, `short`, `int`, `long`, `float` y `double`), falso ($false$) para los tipos primitivos booleanos (i.e., $\mathbb{B}$ y `boolean`), el carácter nulo (‘`\0`’) para el tipo `char`, y el apuntador nulo (`null`) para los tipos compuestos.

Tabla 7.2. *Ejemplos de tipos primitivos y tipos compuestos en GOLD 3.*

Tipo	Descripción
`char[]`	Arreglos de caracteres.
`java.lang.String`	Cadenas de texto.
`String`	Denota la clase `java.lang.String` porque por defecto todo programa *GOLD* importa las clases pertenecientes al paquete `java.lang`.
`int`	Valores enteros de 32 bits.
`int[]`	Arreglos de elementos de tipo `int`.
`int[][]`	Matrices bidimensionales de elementos de tipo `int` (o en su defecto, arreglos de arreglos de elementos de tipo `int`).
`int[][][]`	Matrices tridimensionales de elementos de tipo `int` (o en su defecto, arreglos de arreglos de arreglos de elementos de tipo `int`).
$\mathbb{Z}$	Números enteros de precisión arbitraria.
$\mathbb{Z}$`[][]`	Matrices bidimensionales de elementos de tipo $\mathbb{Z}$ (o en su defecto, arreglos de arreglos de elementos de tipo $\mathbb{Z}$).
`java.util.TreeSet`	Árboles Rojinegros.
`java.util.TreeSet[][]`	Matrices bidimensionales de elementos de tipo `java.util.TreeSet` (o en su defecto, arreglos de arreglos de elementos de tipo `java.util.TreeSet`).
`TreeSet`	Denota la clase `java.util.TreeSet`, suponiendo que el paquete `java.util` es importado por el programa principal y que no hay conflicto de nombres.
`TreeSet[][]`	Matrices bidimensionales de elementos de tipo `java.util.TreeSet`, suponiendo que el paquete `java.util` es importado por el programa principal y que no hay conflicto de nombres (o en su defecto, arreglos de arreglos de elementos de tipo `java.util.TreeSet`).

GOLD está diseñado como un *lenguaje dinámicamente tipado* en el que no se obliga al usuario a indicar explícitamente el tipo de cada una de las variables que declara, haciendo que el tipo de las expresiones pueda variar en tiempo de ejecución [40]. Sin embargo, si el usuario decide indicar el tipo de todas las variables que va a declarar, *GOLD* termina comportándose como un *lenguaje estáticamente tipado*, donde todas las expresiones tienen un tipo que puede ser inferido en tiempo de compilación [40]. Cada una de las variables que el usuario declare sin tipo será

tratada por *GOLD* como una variable de tipo `java.lang.Object`, que es superclase de todas las clases de *Java* (i.e., toda clase hereda de `Object` por defecto). Por último, para que el comportamiento descrito sea consistente con los tipos primitivos de *Java*, se aplicará su mecanismo de *autoboxing* [61] fielmente.

Así pues, *GOLD* se comporta como un lenguaje estática o dinámicamente tipado dependiendo de si el usuario declara o no el tipo de todas sus variables, respectivamente. Todo posible error de tipos que sea detectado sobre una variable declarada sin tipo será reportado como una *advertencia de compilación* y, en contraparte, todo error de tipos que sea detectado sobre una variable declarada con tipo será reportado como un *error de compilación*. Si el usuario lo desea, puede deshabilitar estas advertencias como se describe en la sección §7.2.7.

La principal ventaja de que *GOLD* sea un lenguaje dinámicamente tipado es que se permite la declaración de procedimientos y funciones que actúen sobre una gran variedad de tipos. Por ejemplo, el procedimiento definido en el código 7.1 recibe un parámetro *A* que puede ser una lista, un arreglo de elementos de tipo primitivo o un arreglo de elementos de tipo compuesto. En contraste, en *Java* hubiese sido necesario declarar una multitud de métodos con distintas signaturas pero con el mismo cuerpo, uno por cada posible tipo que pudiera tener el parámetro *A*. En *Java*, este problema es evidente en clases como `java.util.Arrays`, donde la mayoría de sus métodos (e.g., `sort`) están replicados para manejar tipos compuestos (clases) y cada tipo primitivo por aparte.

Código 7.1. *Bubble-sort implementado en GOLD, sin declarar ninguna variable.*

```
1 procedure bubbleSort(A) begin
2   for i:=0 to |A|-1 do
3     for j:=|A|-1 downto i+1 do
4       if A[j]<A[j-1] then
5         A[j],A[j-1]:=A[j-1],A[j]
6       end
7     end
8   end
9 end
```

7.2.3. Expresiones

GOLD provee las siguientes formas de expresión, ordenadas de acuerdo con la clasificación de Watt [40] estudiada en la sección §4.1.5.2:

1. *Literales* (*Literals*). Son expresiones que denotan valores fijos de algún tipo:

   - *Constantes* (*Constant literals*). Denotan algunas constantes básicas de acuerdo con la sintaxis del símbolo terminal `CONSTANT`, incluyendo los valores booleanos de tipo `boolean` (`TRUE`, `true`, `FALSE`, `false`), el apuntador nulo (`NIL`, `nil`, `NULL`, `null`), la colección vacía ($\varnothing$), el conjunto universal (**U**), la secuencia vacía (ε), la cadena de texto vacía (λ) y el infinito positivo (∞).
   - *Números* (*Number literals*). Denotan valores pertenecientes a los tipos primitivos de *Java*, exceptuando `boolean` (i.e., `long`, `int`, `short`, `byte`, `double`, `double`, `float` y `char`). Al valor numérico, que debe estar escrito en base decimal, se le puede concatenar como sufijo una letra para indicar a qué tipo primitivo de *Java* pertenece, de acuerdo con la nomenclatura consignada en la tabla 7.3. En caso de que el número no tenga ninguna letra como sufijo, denotará por defecto un valor de tipo `double` si tiene punto flotante (e.g., `17.34`), o de tipo `int` de lo contrario (e.g., `17`). Para denotar caracteres se puede usar el literal `#`c (o `#`C), donde # es un número en base decimal con el código *Unicode* del carácter representado (e.g., `98`c corresponde a la letra 'b', y `0`c representa el carácter nulo). Además, se puede usar la notación científica, que fue descrita en la definición del símbolo terminal `NUMBER`.

- *Cadenas de texto* (*String literals*). Denotan instancias de la clase `java.lang.String` de *Java*, de acuerdo con la sintaxis del símbolo terminal `STRING` (e.g., `"Hello World\r\nHELLO WORLD"`). Para concatenar cadenas se puede usar con libertad el operador $+$, como en *Java*.
- *Caracteres* (*Character literals*). Denotan caracteres correspondientes al tipo primitivo `char`, de acuerdo con la sintaxis del símbolo terminal `CHARACTER` (e.g., `'A'`, `'\n'`, `'\u2200'`).

2. *Accesos* (*Accesses*). Son referencias a entidades previamente declaradas, a través del uso de identificadores (`ID`) o nombres calificados (`QN`):

   - *Accesos a variable* (*Variable access expressions*). Son referencias a variables previamente declaradas en *GOLD*, que cuando son evaluadas entregan el valor actual de la variable referida.
   - *Accesos a arreglo* (*Array access expressions*). Representan accesos a los componentes de un arreglo *Java* a través de índices enteros no negativos, donde la numeración de sus posiciones inicia desde cero. Para acceder a un arreglo se sigue la sintaxis $array[E_1][E_2]\cdots[E_n]$, donde *array* es el arreglo a acceder y $E_1, E_2, \ldots, E_n$ son los subíndices que denotan la posición del arreglo que desea ser accedida (e.g., `x[5]`, `x[3][8]`). Cada subíndice debe ser un número de tipo primitivo (o de tipo `java.lang.Number`) que automáticamente es convertido por *GOLD* en un valor entero de tipo `int`. Después de acceder a una posición de un arreglo se obtiene el valor actual de la componente indicada por los subíndices. De manera similar, se puede obtener un subarreglo siguiendo la sintaxis $array[E_1..E_2]$, donde *array* es el arreglo a acceder, E_1 es el límite inferior (inclusive) y E_2 es el límite superior (inclusive).
   - *Accesos a estructura* (*Structure access expressions*). Es una potente extensión de los accesos a arreglo, que permite usar la misma notación para acceder a una posición de una lista (*list* / *sequence*) o a una entrada de una asociación llave-valor (*map* / *associative array* / *dictionary*), como se hace en lenguajes como *Python* y *C#* a través de la clase `Dictionary`. Para acceder a una lista, los subíndices deben seguir siendo números (e.g., `x[5]`), mientras que para acceder a una asociación llave-valor, los subíndices pueden ser de cualquier tipo (e.g., `x["Hello World"]`). De hecho, este mecanismo se puede aplicar a cualquier instancia de la interfaz `java.lang.Iterable` de *Java*, enumerando sus elementos en el orden en que son entregados por su iterador. De manera similar, se puede obtener una sublista siguiendo la sintaxis $list[E_1..E_2]$, donde *list* es la lista a acceder, E_1 es el límite inferior (inclusive) y E_2 es el límite superior (inclusive).
   - *Accesos a atributo* (*Field access expressions*). Representan accesos a atributos de una clase *Java*, ya sean estáticos o no. Para acceder a un *atributo de instancia* se sigue la sintaxis $object.field$, donde *object* es el objeto invocado y $field$ es el nombre del atributo a acceder (e.g., `x.color`). Por otro lado, para acceder a un *atributo estático* o *atributo de clase*, se sigue la sintaxis $Class.field$, donde *Class* es el nombre calificado de la clase invocada y $field$ es el nombre del atributo estático a acceder (e.g., `java.lang.Math.PI`). Al evaluar un atributo se obtiene su valor actual, que depende del estado del objeto invocado (si es un atributo de instancia).
   - *Accesos a tipo* (*Type access expressions*). Representan accesos a instancias de la clase `java.lang.Class<T>`. Para acceder al objeto *Java* que denota un determinado tipo se sigue la sintaxis *Class*`.class`, donde *Class* es el nombre calificado de la clase (o el nombre del tipo primitivo) a acceder (e.g., `java.lang.String.class`, `String.class`, `String[].class`, `int.class`, `int[].class`, `int[][].class`, $\mathbb{Z}$`.class`). Asimismo, para acceder al objeto *Java* que representa la clase correspondiente a un programa *GOLD*, se usa la palabra clave `class`. Este mecanismo permite la utilización del *API* de reflexión de *Java* (*Java Reflection API*) [62] mediante la invocación de métodos de la clase `java.lang.Class<T>` (e.g., `Math.class.getDeclaredMethods()`, `int[][][].class.getComponentType()`, `class.getResourceAsStream("dir/file")`).
   - *Accesos a procedimiento* (*Procedure access expressions*). Son referencias a procedimientos (tanto procedimientos propios como funciones) previamente declarados en *GOLD*, que cuando son evaluadas entregan un apuntador al procedimiento. De esta manera, las referencias a los procedimientos se pueden tratar como

valores cualesquiera, permitiendo (por ejemplo) el paso de -referencias a- procedimientos como parámetros, el retorno de -referencias a- procedimientos, y el almacenamiento de -referencias a- procedimientos dentro de variables. Lo anterior facilitaría la definición de operaciones entre funciones numéricas como la composición, la derivación y la integración, acercando *GOLD* al paradigma de la programación funcional porque ‘‘las funciones se tratarían como valores ordinarios que pueden ser pasados como parámetro o ser retornados como resultado de otras funciones’’ [40].

3. *Construcciones* (*Constructions*). Son expresiones que crean valores compuestos a partir de subexpresiones más sencillas:

   - *Invocaciones a constructores de clase* (*Class constructor calls*). Representan llamados a métodos constructores de una clase *Java*, con el fin de crear nuevos objetos. Para invocar un *método constructor* se sigue la sintaxis $Class(E_1, E_2, \ldots, E_n)$, donde *Class* es el nombre calificado de la clase que se desea instanciar (o el identificador de un tipo primitivo), y $E_1, E_2, \ldots, E_n$ son los argumentos (e.g., `java.util.String("Hello")`, `String("Hello")`, `int(5)`, `ℚ("17/14")`, `ℂ("(5,31)")`). Se puede usar la palabra reservada `new` para mejorar la legibilidad, permitiendo la escritura de expresiones de la forma `new` $Class(E_1, E_2, \ldots, E_n)$ (e.g., `new String("Hello")`).
   - *Invocaciones a constructores de arreglo* (*Array constructor calls*). Representan expresiones que crean nuevos arreglos. Para crear un arreglo sin contenido se sigue la sintaxis $Class[E_1][E_2]\cdots[E_n]$, donde *Class* es el nombre calificado de la clase de los componentes del arreglo que se desea crear (o el identificador de un tipo primitivo), y $E_1, E_2, \ldots, E_n$ son expresiones enteras no negativas que indican el tamaño de cada una de las dimensiones del nuevo arreglo (e.g., `int[5]`, `ℤ[7][9]`, `String[91][7][12]`). Por otro lado, para crear un arreglo con contenido se sigue la sintaxis $Class[][]\cdots[]⟦E_1, E_2, \ldots, E_n⟧$, donde *Class* se define como antes y $E_1, E_2, \ldots, E_n$ es una secuencia de expresiones que describe el contenido del arreglo (e.g., `int[]⟦57,32,84⟧`, `int[]⟦⟦91,74⟧,⟦⟧,⟦8,35⟧,null⟧`). Se puede usar la palabra reservada `new` para mejorar la legibilidad, permitiendo la escritura de expresiones de la forma `new` $Class[E_1][E_2]\cdots[E_n]$ (e.g., `new int[5]`) y `new` $Class[][]\cdots[]⟦E_1, E_2, \ldots, E_n⟧$ (e.g., `new int[]⟦57,32,84⟧`). Las posiciones de los arreglos sin contenido son inicializadas con el valor por defecto asociado al tipo (véase la sección §7.2.2).
   - *Colecciones descritas por enumeración* (*Enumerations*). Representan colecciones (arreglos, listas, conjuntos y bolsas) que son definidas por extensión, listando explícitamente cada uno de los elementos que contienen. No hay que olvidar que en las listas importa el orden y las repeticiones, que en los conjuntos no importa el orden ni las repeticiones, y que en las bolsas no importa el orden pero si las repeticiones.
     - Los *arreglos por enumeración* (*array enumerations*), que son de tipo `java.lang.Object[]`, son arreglos dados en la forma $⟦E_1, E_2, \ldots, E_n⟧$, donde $E_1, E_2, \ldots, E_n$ son sus componentes (e.g., `⟦⟧`, `⟦94,31,30,17,17⟧`, `⟦⟦31,18,99⟧,⟦11,24,51⟧,⟦38,70,30⟧⟧`, y `⟦⟦⟦10,25⟧,⟦81,81⟧⟧,⟦⟦10,25⟧,⟦81,81⟧⟧⟧`).
     - Las *listas por enumeración* (*list enumerations*) son listas dadas en la forma $⟨E_1, E_2, \ldots, E_n⟩$, donde $E_1, E_2, \ldots, E_n$ son sus elementos (e.g., `⟨⟩`, `⟨9,3,3,1,4⟩`, y `⟨91,⟨80⟩,⟨⟨35⟩⟩⟩`).
     - Los *conjuntos por enumeración* (*set enumerations*) son conjuntos dados en la forma $\{E_1, E_2, \ldots, E_n\}$, donde $E_1, E_2, \ldots, E_n$ son sus miembros (e.g., `{}`, `{9,3,1,4}`, y `{9,{},{8,74},{71,{{74,29}}}}`).
     - Las *bolsas por enumeración* (*bag enumerations*) son bolsas dadas en la forma $\{\!|E_1, E_2, \ldots, E_n|\!\}$, donde $E_1, E_2, \ldots, E_n$ son sus miembros (e.g., `{||}`, `{|9,3,3,1,4|}`, y `{|{|3,3,1|},{|3,3,1|}|}`).

4. *Llamados a función* (*Function calls*). Son invocaciones que calculan el resultado de aplicar una función, método u operador, sobre ciertos argumentos dados como parámetro:

   - *Llamados a procedimiento propio* (*Proper procedure calls*). Representan invocaciones a procedimientos propios declarados en *GOLD*, siguiendo la sintaxis $f(E_1, E_2, \ldots, E_n)$, donde f es el nombre del procedimiento propio invocado (o una referencia a un procedimiento propio a través de un acceso a variable), y

$E_1, E_2, \ldots, E_n$ son los argumentos (e.g., `bubbleSort(new int[]{5,3})`). El resultado obtenido después de la invocación de un procedimiento propio es un apuntador nulo (`null`).

- *Aplicaciones de función* (*Function applications*). Representan invocaciones a funciones declaradas en *GOLD*, siguiendo la sintaxis $f(E_1, E_2, \ldots, E_n)$, donde f es el nombre de la función invocada (o una referencia a una función a través de un acceso a variable), y $E_1, E_2, \ldots, E_n$ son los argumentos (e.g., `fib(5)`). El resultado obtenido después de la invocación de una función es el valor que entregue como retorno. Hay veintidós funciones predefinidas en *GOLD*, que pueden ser invocadas en cualquier momento (evaluando previamente cada una de las expresiones dadas como argumento, que deben entregar valores numéricos):
  - `max`$(E_1, E_2, \ldots, E_n)$ para calcular el máximo de la lista no vacía de valores $E_1, E_2, \ldots, E_n$;
  - `min`$(E_1, E_2, \ldots, E_n)$ para calcular el mínimo de la lista no vacía de valores $E_1, E_2, \ldots, E_n$;
  - `gcd`(E, F) para calcular el máximo común divisor de E y F;
  - `lcm`(E, F) para calcular el mínimo común múltiplo de E y F;
  - `abs`(E) para calcular el valor absoluto de E;
  - `pow`(E, F) para calcular el valor de E elevado a la F (i.e., E^F);
  - `sqrt`(E) para calcular la raíz cuadrada del valor E (i.e., $\sqrt{E}$);
  - `cbrt`(E) para calcular la raíz cúbica del valor E (i.e., $\sqrt[3]{E}$);
  - `root`(E, F) para calcular la raíz F-ésima del valor E (i.e., $\sqrt[F]{E}$);
  - `ln`(E) para calcular el logaritmo natural del valor E (i.e., $ln(E)$);
  - `log`(E, F) para calcular el logaritmo en base F del valor E (i.e., $log_F(E)$);
  - `exp`(E) para calcular la función exponencial aplicada sobre el valor E (i.e., e^E);
  - `sin`(E) para calcular el seno del valor E (i.e., $sin(E)$), donde E está dado en radianes;
  - `cos`(E) para calcular el coseno del valor E (i.e., $cos(E)$), donde E está dado en radianes;
  - `tan`(E) para calcular la tangente del valor E (i.e., $tan(E)$), donde E está dado en radianes;
  - `sinh`(E) para calcular el seno hiperbólico del valor E (i.e., $sinh(E)$);
  - `cosh`(E) para calcular el coseno hiperbólico del valor E (i.e., $cosh(E)$);
  - `tanh`(E) para calcular la tangente hiperbólica del valor E (i.e., $tanh(E)$);
  - `asin`(E) para calcular el arcoseno (en radianes) del valor E (i.e., $asin(E)$);
  - `acos`(E) para calcular el arcocoseno (en radianes) del valor E (i.e., $acos(E)$);
  - `atan`(E) para calcular la arcotangente (en radianes) del valor E (i.e., $atan(E)$);
  - `asinh`(E) para calcular el arcoseno hiperbólico del valor E (i.e., $asinh(E)$);
  - `acosh`(E) para calcular el arcocoseno hiperbólico del valor E (i.e., $acosh(E)$); y
  - `atanh`(E) para calcular la arcotangente hiperbólica del valor E (i.e., $atanh(E)$).
- *Invocaciones a método* (*Method calls*). Representan llamados a métodos de una clase *Java*, ya sean estáticos o no. Para invocar un *método de instancia* se sigue la sintaxis $object.method(E_1, E_2, \ldots, E_n)$, donde *object* es el objeto invocado, *method* es el nombre del método a invocar, y $E_1, E_2, \ldots, E_n$ son los argumentos (e.g., `x.toString()`). Por otro lado, para invocar un *método estático* o *método de clase*, se sigue la sintaxis $Class.method(E_1, E_2, \ldots, E_n)$, donde *Class* es el nombre calificado de la clase invocada, *method* es el nombre del método estático a invocar, y $E_1, E_2, \ldots, E_n$ son los argumentos (e.g., `java.lang.Math.max(5,8)`). En caso de que el método invocado no tenga retorno (i.e., tenga retorno de tipo `void`), el resultado obtenido después de su invocación es un apuntador nulo (`null`).
- *Aplicaciones de operador* (*Operator applications*). Todos los operadores unarios y binarios enumerados en la tabla 5.4 pueden ser utilizados para realizar operaciones sobre números, valores booleanos, conjuntos, bolsas y secuencias en *GOLD*, siguiendo el patrón *nested operator expression* de Fowler [49] (e.g., `5+3*4`, `p⇒q≡true`). Las reglas de precedencia, asociatividad, y conjuncionalidad definidas en dicha tabla son

respetadas de acuerdo con la teoría expuesta en la sección §4.1.5.2, incluyendo las distintas formas de mencionar cada operador (e.g., $\vee$, `or` y `||` para la disyunción; $\wedge$, `and` y `&&` para la conjunción; $=$ y $==$ para la igualdad). Algunas operaciones booleanas se evalúan por cortocircuito (dejando de calcular su segundo operando en casos específicos), como la conjunción ($\wedge$), la disyunción ($\vee$), la implicación ($\Rightarrow$) y la anti-consecuencia ($\not\Leftarrow$); el resto de operaciones deben evaluar todas sus subexpresiones para poder calcular el valor resultante. Por otro lado, el operador con símbolo $+$ no sólo sirve para sumar números; también puede usarse para concatenar cadenas de texto de tipo `java.lang.String` (e.g., `"Hola"+" "+"Mundo"` entrega la cadena `"Hola Mundo"`), como en *Java* [†1]. Finalmente, el operador de igualdad en *GOLD* ($=$) sirve para determinar si dos objetos son iguales o no, cuyo equivalente en *Java* sería el método `equals` (no hace lo mismo que el operador de comparación en *Java* ($==$), que en general sirve para determinar si dos apuntadores referencian al mismo objeto).

5. *Expresiones condicionales* (*Conditional expressions*). Son expresiones que calculan un valor que depende de una condición. Para escribir una expresión condicional se sigue la sintaxis $B?E:F$, donde B es una expresión booleana llamada *guarda* y E,F son dos expresiones de cualquier tipo. Al evaluar la expresión $B?E:F$ en un estado determinado, se entrega el valor de la expresión E si la guarda B se cumple, o el valor de la expresión F de lo contrario (e.g., `A&&B?53:80`, `A?53:(B?80:97)`).

6. *Expresiones iterativas* (*Iterative expressions*). Son expresiones que realizan cálculos iterativos sobre sus subexpresiones, actuando como bloques de expresión [40] cuyas declaraciones locales corresponden a entidades denominadas *variables cuantificadas*, *variables ligadas* o *dummies*. Este tipo de expresiones comprende tanto colecciones descritas por compresión como cuantificaciones [†2]:

   - *Colecciones descritas por comprensión* (*Comprehensions*). Representan colecciones (listas, conjuntos y bolsas) que son definidas por comprensión, describiendo matemáticamente las propiedades que cumplen sus elementos. No hay que olvidar que en las listas importa el orden y las repeticiones, que en los conjuntos no importa el orden ni las repeticiones, y que en las bolsas no importa el orden pero si las repeticiones.
     - Las *listas por comprensión* (*list comprehensions*), inspiradas en *Haskell* [63], se escriben en la forma $\langle Body \mid Range \rangle$, donde *Range* es el rango y *Body* es el cuerpo que define la forma de sus elementos (e.g., $\langle$`abs(x)`$\mid$`-3≤x≤3,[x≠0]`$\rangle$ denota $\langle 3,2,1,1,2,3\rangle$, y $\langle\langle$`x,y`$\mid$`0≤x≤3,x≤y≤3,[x+y≠5]`$\rangle$ denota $\langle\langle 0,0\rangle,\langle 0,1\rangle,\langle 0,2\rangle,\langle 0,3\rangle,\langle 1,1\rangle,\langle 1,2\rangle,\langle 1,3\rangle,\langle 2,2\rangle,\langle 3,3\rangle\rangle$).
     - Los *conjuntos por comprensión* (*set comprehensions*), inspirados en la notación *set-builder* [64], se escriben en la forma $\{Body \mid Range\}$, donde *Range* es el rango y *Body* es el cuerpo que define la forma de sus miembros (e.g., $\{$`abs(x)`$\mid$`-3≤x≤3,[x≠0]`$\}$ denota $\{1,2,3\}$, y $\{$`x+y`$\mid$`0≤x≤3,x≤y≤3,[¬(x=1∧y=1)]`$\rangle$ denota $\{0+0,0+1,0+2,0+3,1+2,1+3,2+2,2+3,3+3\} = \{0,1,2,3,3,4,4,5,6\} = \{0,1,2,3,4,5,6\}$).
     - Las *bolsas por comprensión* (*bag comprehensions*), inspiradas en la notación *set-builder* [64], se escriben en la forma $\{\!|Body \mid Range|\!\}$, donde *Range* es el rango y *Body* es el cuerpo que define la forma de sus

[1] Mejor aún, si uno de los argumentos del operador con símbolo $+$ es un objeto de tipo `java.lang.String`, entonces se da como resultado la cadena de texto producto de concatenar las representaciones textuales de sus dos operandos, luego de invocar el método `toString()` sobre cada uno de éstos (e.g., `"Hola"+5` entrega la cadena `"Hola5"`, `5+"Hola"` entrega la cadena `"5Hola"`). Para evitar cualquier ambigüedad, se define que el operador $+$ sobre números y el operador $+$ sobre cadenas de texto son mutuamente asociativos por la izquierda (e.g., `5.1+3.2+"Hola"` es la cadena `"8.3Hola"`, `"Hola"+5.1+3.2` es la cadena `"Hola5.13.2"`).

[2] Es necesario tener en cuenta varios conceptos básicos comunes a todas las expresiones iterativas de *GOLD*: los *cuerpos*, que son expresiones de cualquier tipo; y los *rangos*, que son listas de *fragmentos* (de la forma $G_1, G_2, \ldots, G_n$) donde cada fragmento (G_i) es una condición booleana encerrada entre corchetes (de la forma $[E]$, donde E es una expresión) o una declaración de una variable ligada (de la forma $x{=}E$, $x{\in}E$, $x{\subseteq}E$, $x{\subset}E$, $E{\leq}x{\leq}F$, $E{\leq}x{<}F$, $E{<}x{\leq}F$, o $E{<}x{<}F$, donde E y F son expresiones, y x es un identificador). Los fragmentos de un rango se procesan en orden de aparición, donde cada declaración de variable ligada se traduce en un comando iterativo (concretamente, en un ciclo *for-each* que normalmente itera sobre valores enteros) y cada condición booleana se traduce en un comando condicional (concretamente, en un condicional *if-then*). Para mayor información sobre la sintaxis de las colecciones descritas por compresión y de las cuantificaciones, véase la notación en formato *EBNF* disponible en la sección §A.1.1.

miembros (e.g., {|`abs(x)|-3≤x≤3,[x≠0]`|} denota $\{\!|1,1,2,2,3,3|\!\}$, y {|`x+y|0≤x≤3,x≤y≤3,[x≠1∨y≠1]`|} denota $\{\!|0+0,0+1,0+2,0+3,1+2,1+3,2+2,2+3,3+3|\!\} = \{\!|0,1,2,3,3,4,4,5,6|\!\}$).

- *Cuantificaciones* (*Quantifications*). Representan cuantificaciones inspiradas en la notación del libro *A Logical Approach to Discrete Math* de Gries y Schneider [12][65], que se escriben en la forma $(\star \textit{Dummies} \mid \textit{Body} : \textit{Range})$, donde *Dummies* es una lista no vacía de variables ligadas (distintas y separadas por comas), *Range* es el rango, *Body* es el cuerpo, y $\star$ es un operador binario asociativo y conmutativo denominado *cuantificador* (e.g., `(Σx|1<x<5:x)` denota la suma $2+3+4=9$, `(Σx|1<x<5:x^2)` denota la suma $2^2+3^2+4^2=29$, `(Σk|1≤k≤n:k)` denota la suma $1+2+\cdots+n = n\cdot(n+1)/2$, `(Σi|1≤i<n,[n%i=0]:i)` denota la suma de los divisores propios positivos de n, `(Σi|1≤i<100,[i%2=0],[i%3=0]:i)` denota la suma de todos los números entre 1 y 99 que son múltiplos de 2 y de 3, y `(Σi,j|1≤i<100,i≤j<100:i+j)` denota la doble sumatoria `(Σi|1≤i<100:(Σj|i≤j<100:i+j))`). Concretamente, $(\star x_1, x_2, \ldots, x_n \mid R : P)$ es una expresión que denota la aplicación del operador $\star$ sobre los valores de la forma P para todas las posibles combinaciones de valores de las variables ligadas $x_1, x_2, \ldots, x_n$ que satisfacen el rango R [12][65]. Las variables ligadas $x_1, x_2, \ldots, x_n$ deben coincidir con las declaradas en el rango R; de lo contrario, se producirá un error de compilación. Se pueden usar ocho posibles operadores como cuantificador:
  - *sumatoria* Σ (e.g., `(Σi|1≤i<n,[n%i=0]:i)=n` es una expresión que es verdadera si y sólo si `n` es perfecto);
  - *multiplicatoria* Π (e.g., `(Πi|1≤i≤n:i)` denota el factorial de n);
  - *mínimo* $\downarrow$ (e.g., `(↓i|1≤i≤8:i^2)` es $1^2 \downarrow 2^2 \downarrow 3^2 \downarrow 4^2 \downarrow 5^2 \downarrow 6^2 \downarrow 7^2 \downarrow 8^2 = 1^2 = 1$);
  - *máximo* $\uparrow$ (e.g., `(↑i|1≤i≤8:i^2)` es $1^2 \uparrow 2^2 \uparrow 3^2 \uparrow 4^2 \uparrow 5^2 \uparrow 6^2 \uparrow 7^2 \uparrow 8^2 = 8^2 = 64$);
  - *intersección de conjuntos* $\cap$ (e.g., `(∩i|1≤i≤8:(0..i))` es `(0..1)={0,1}`);
  - *unión de conjuntos* $\cup$ (e.g., `(∪i|1≤i≤8:(0..i))` es `(0..8)={0,1,2,3,4,5,6,7,8}`);
  - *para todo* o *cuantificador universal* $\forall$ (e.g., `(∀x|1≤x≤5:x^2<25)` es falso porque $\neg(5^2 < 25)$); y
  - *existe* o *cuantificador existencial* $\exists$ (e.g., `(∀i|2≤i<n:n%i=0)` denota la expresión `¬(∃i|2≤i<n:n%i≠0)` que es verdadera si y sólo si n es un número primo).

7. *Otras expresiones*. Son expresiones que no están catalogadas dentro de la clasificación de Watt [40]:

   - *Expresiones parentizadas* (*Parenthesized expressions*). Son expresiones escritas entre paréntesis redondos (i.e., expresiones escritas en la forma (E), donde E es una expresión), que son símbolos técnicos que se pueden utilizar para componer aplicaciones de operadores de menor precedencia como subexpresiones de otros de mayor precedencia (e.g., los paréntesis en la expresión `(5+3)*4` son necesarios porque la adición ($+$) tiene menor precedencia que la multiplicación ($*$)), o para facilitar la lectura de las expresiones (e.g., `p⇒q⇒r` es más claro si se escribe como `p⇒(q⇒r)`, sabiendo que la implicación ($\Rightarrow$) es un operador asociativo por la derecha). En toda expresión se pueden eliminar los paréntesis que resulten innecesarios de acuerdo con las reglas de precedencia enunciadas en la tabla 5.4. También se pueden usar los paréntesis (*brackets*) descritos en la tabla A.12.
   - *Conversión de tipos* (*Casts*). Son aplicaciones del operador de conversión de tipos (*cast*), que permiten transformar una expresión a un determinado tipo. Para convertir una expresión E a un tipo T se sigue la sintaxis $(E : T)$, o $(E$ `as` $T)$ (e.g., `(5 as int)`, `(5:int)`, `(5:Integer)`, `(5:ℤ)`, `(5 as double)`, `⟦9,3,3,1,4⟧:double[]`). Cada vez que se evalúe la expresión E, el valor resultado será convertido automáticamente al tipo T, con la siguiente restricción: para que un valor de tipo U pueda ser convertido en uno de tipo T, debe cumplirse que U y T sean tipos primitivos tales que $U \subseteq T$, o que U y T sean clases tales que U sea subclase de T. Sin embargo, *GOLD* debilita la condición impuesta sobre los tipos primitivos para que sea posible realizar conversiones entre cualesquiera dos tipos numéricos ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, `byte`, `short`, `int`, `long`, `float` y `double`), posiblemente conllevando pérdida de información. También se dan facilidades para convertir cadenas de texto de tipo `String` en valores pertenecientes a cualquiera de los tipos primitivos de *GOLD*, y viceversa.

Tabla 7.3. *Convenciones de GOLD 3 para denotar valores de los tipos primitivos de Java, excepto* `boolean`.

Tipo primitivo	Sufijo	Ejemplos
`char`	'`C`' o '`c`	`65c` es la letra '`A`', `8704C` es el símbolo '∀'.
`byte`	'`B`' o '`b`'	`52b`, `127B`.
`short`	'`S`' o '`s`'	`931s`, `32767S`.
`int`	'`I`' o '`i`'	`12i`, `2147483647I`.
`long`	'`L`' o '`l`'	`93871l`, `9223372036854775807L`.
`float`	'`F`' o '`f`'	`0.03014f`, `3.014E-2f`, `157573.49230F`.
`double`	'`D`' o '`d`'	`0.00002957d`, `2.957E-5d`, `0.10939162670D`.

7.2.4. Variables

Para declarar una variable en *GOLD* puede usarse cualquiera de las siguientes sentencias *var*:

- La sentencia

  `var` x

  declara una *variable atipada* (i.e., con tipo `java.lang.Object`) con identificador x (e.g., `var a`).

- La sentencia

  `var` $x : T$

  declara una *variable tipada* con identificador x y tipo T (e.g., `var a:String`, `var b:int`, `var c:int[]`, `var d:int[][]`, `var e:`$\mathbb{Z}$, `var f:java.util.LinkedList`).

- La sentencia

  `var` $x : Class(E_1, E_2, \ldots, E_n)$

  declara una variable con identificador x, inicializada (explícitamente) con un nuevo objeto creado por la invocación del constructor de clase $Class(E_1, E_2, \ldots, E_n)$ (e.g., `var a:String("Hello World")`, `var b:int(5)`, `var c:java.util.LinkedList()`, `var d:`$\mathbb{Q}$`("17/14")`).

- La sentencia

  `var` $x : Class[E_1][E_2]\cdots[E_n]$

  declara una variable con identificador x, inicializada (explícitamente) con un nuevo arreglo sin contenido creado por la invocación del constructor de arreglo $Class[E_1][E_2]\cdots[E_n]$ (e.g., `var a:String[5][3]`, `var b:int[8][(Math.random()*10 as int)]`, `var c:Float[0]`, `var d:`$\mathbb{Z}$`[3][8][19]`).

Cada variable que se declare en *GOLD* es inicializada automáticamente con el valor por defecto asociado a su tipo (véase la sección §7.2.2), excepto si fue inicializada explícitamente con un objeto o con un arreglo. Adicionalmente, en una misma sentencia *var* pueden declararse varias variables distintas en la forma `var` $x_1, x_2, \ldots, x_n$, donde éstas son inicializadas según su orden de aparición (e.g., `var a,b,c`, `var a:int,b:double,c:String`, `var a:int,b,c:int`, `var a:int(4),b:int(a+3),c:int(a-b)`).

Como *GOLD* es un lenguaje dinámicamente tipado, el usuario no está obligado a indicar explícitamente el tipo de cada una de las variables que declara, permitiendo así la existencia de variables atipadas. De hecho, se podría no declarar una variable explícitamente porque, como se verá más adelante, las asignaciones en *GOLD* son capaces de declarar implícitamente las variables asignadas, si previamente no fueron declaradas.

GOLD adopta la siguiente convención al momento de asignar valores a las variables: los valores de tipo primitivo de *GOLD* se copian *por valor* [40] y el resto de valores (i.e., los objetos) se copian *por referencia* [40]. De esta manera, el comportamiento sería coherente con el de *Java*, donde el efecto de copiar por valor sobre los objetos se puede simular invocando el método `clone` [40].

Al igual que en *Java*, los objetos en *GOLD* se manejan con apuntadores, y existe el concepto de *apuntador nulo* (*null pointer*), que es denotado por las palabras reservadas `NIL`, `nil`, `NULL` y `null`. Además, aprovechando que los programas *GOLD* son traducidos en clases *Java*, la liberación de memoria principal durante la ejecución de un programa *GOLD* termina siendo responsabilidad del recolector de basura (*garbage collector*) de *Java*.

7.2.5. Comandos

GOLD provee las siguientes formas de comando, que están clasificadas con base en la teoría de Watt [40] que fue estudiada en la sección §4.1.5.2:

1. *Instrucción vacía*. La instrucción

   ```
   skip
   ```

   es un comando que no realiza ninguna operación, dejando intacto el valor de todas las variables. La cadena vacía también representa la misma instrucción, permitiendo así que un bloque de comandos pueda estar vacío.

2. *Declaración de variables*. Cualquiera de las sentencias *var* enunciadas en la sección §7.2.4 puede ser usada como una instrucción de *GOLD* para declarar variables (e.g., `var a`, `var a,b,c`, `var a:int,b,c:int`, `var a:int(4),b:int(a+3),c:int(a-b)`, `var a:double[5],b:int[a.length],c:long[a.length+b.length]`).

3. *Llamado a procedimiento*. La instrucción

   `call` E

   evalúa la expresión E, que puede ser un llamado a procedimiento propio, una aplicación de función, o una invocación a método, según lo definido en la sección §7.2.3 (e.g., `call (x[5]).foo().goo().hoo()`). Se dice que una expresión tiene *efectos secundarios* si su evaluación involucra la alteración del valor de alguna variable [40]. Como E puede ser cualquier expresión, sólo aquellas con efectos secundarios pueden tener sentido dentro de una instrucción de la forma `call` E (e.g. `call java.util.Arrays.sort(A)`). La palabra reservada `call` es una *palabra irrelevante* [40] que puede ser omitida para simplificar los llamados a procedimiento en *GOLD* (e.g., `(x[5]).foo().goo().hoo()`, `java.util.Arrays.sort(A)`).

4. *Asignaciones*. Son instrucciones que asignan valores a variables:

   - *Asignación simple*. La instrucción

     $$x := E$$

     asigna a la variable x el valor de la expresión E, después de ser evaluada (e.g., `x:=5`, `x:=Math.random()`). Se deben tener en cuenta las siguientes consideraciones:

     - para el operador de asignación, se puede usar $\leftarrow$ o $=$ en vez del símbolo $:=$ (e.g., `x`$\leftarrow$`5`, `x=5`, `x:=5`);
     - x debe ser un acceso a variable (e.g., `a:=false`), un acceso a arreglo (e.g., `a[5]:=false`), o un acceso a estructura (`a["Hello World"]:=false`), según lo definido en la sección §7.2.3;
     - x no puede ser el acceso a un atributo (i.e., *GOLD* no puede alterar directamente el valor de un atributo de una clase *Java*, lo que debería hacerse a través de un método modificador (*setter*) diseñado para tal fin);

- si x es un acceso a una variable que aún no ha sido declarada, entonces se declara implícitamente una variable atipada con identificador x y tipo `java.lang.Object` antes de realizar la asignación;
- si el valor de la expresión E no puede ser convertido a través de un *cast* en un valor perteneciente al tipo de la variable x, entonces se produce un error de ejecución;
- no se proveen asignaciones múltiples, que son de la forma $x_1 = x_2 = \cdots = x_n = E$ (con $n \geq 2$), donde el valor de la expresión E es asignado a las variables $x_1, x_2, \ldots, x_n$;
- ninguna asignación en *GOLD* retorna el valor asignado; y
- ninguna asignación en *GOLD* puede usarse como expresión o subexpresión.

Al permitir la declaración implícita de variables a través de asignaciones, se reduce la necesidad de escribir sentencias *var*. Este mecanismo es un *azúcar sintáctico* [†3] que facilita enormemente la escritura de programas en *GOLD*, pero que tiene el defecto de que no permite la especificación del tipo de la variable asignada. Para solucionar este último inconveniente, se diseñó como *sal sintáctica* la instrucción $x : T := E$, que primero declara una variable con identificador x y tipo T, luego evalúa la expresión E, y finalmente le asigna a la variable x el valor que entregó E (e.g., `x:int:=5` es una instrucción difícil de leer que abrevia la aplicación de las instrucciones `var x:int` y `x:=5`). Sin embargo, si la expresión E en la asignación $x := E$ es una invocación a un constructor (de clase o de arreglo), entonces el tipo de x se infiere automáticamente.

- *Asignación simultánea*. La instrucción

  $$x_1, x_2, \ldots, x_n := E_1, E_2, \ldots, E_n$$

  primero evalúa las expresiones $E_1, E_2, \ldots, E_n$, y luego asigna simultáneamente a las variables $x_1, x_2, \ldots, x_n$ los valores de las expresiones $E_1, E_2, \ldots, E_n$, respectivamente (e.g., `x,y:=y,x`, `x,y←y,x`, `x,y=y,x`, `x,y,z:=z+x,y+3-x,x+y+z`, `a[3],a[5]:=a[5],a[3]`, `a:int,b:int:=0,1`). En otras palabras, a la variable x_1 se le asigna el valor de la expresión E_1, a la variable x_2 se le asigna el valor de la expresión E_2, ..., y a la variable x_n se le asigna el valor de la expresión E_n, donde todas las expresiones son evaluadas antes de efectuar cualquiera de las asignaciones [12]. Las asignaciones simultáneas permiten realizar el intercambio de valores de dos variables x_1,x_2 a través de instrucciones de la forma $x_1, x_2 := x_2, x_1$. Además de las consideraciones enunciadas para las asignaciones simples, se deben tener en cuenta las siguientes:
  - todas las variables en la lista $x_1, x_2, \ldots, x_n$ deben ser distintas; y
  - la lista $x_1, x_2, \ldots, x_n$ debe tener por lo menos dos elementos.
- *Intercambio (swap)*. La instrucción

  `swap` x `with` y

  intercambia el valor de las variables x y y (e.g., `swap a with b`). Se deben tener en cuenta las siguientes consideraciones:
  - se puede usar el símbolo ↔ en vez de la palabra reservada `with` (e.g., `swap a↔b`);
  - se puede usar la palabra reservada `exchange` en vez de la palabra reservada `swap` (e.g., `exchange a with b`, `exchange a↔b`);
  - x debe ser un acceso a variable (e.g., `swap a↔b`), un acceso a arreglo (e.g., `swap a[5]↔b`), o un acceso a estructura (`swap a["str"]↔b`);
  - y debe ser un acceso a variable (e.g., `swap a↔b`), un acceso a arreglo (e.g., `swap a↔b[5]`), o un acceso a estructura (`swap a↔b["str"]`);

[3] El término *azúcar sintáctico* (*syntactic sugar*) se refiere a las reglas sintácticas diseñadas para facilitar la lectura o escritura de las sentencias [66]. En contraparte, el término *sal sintáctica* (*syntactic salt*) se refiere a las reglas sintácticas diseñadas para dificultar la lectura o escritura de las sentencias [66]. ''*[The] syntactic salt [...] indicates a feature designed to make it harder to write bad code*'' [66].

- ni x ni y pueden ser accesos a atributos de clases *Java*;
- si x o y involucran el acceso a una variable aún no declarada, se produce un error de compilación; y
- si el tipo de x no coincide con el de y, se produce un error de ejecución.

5. *Comandos secuenciales*. La instrucción

   $S_1\ S_2\ \cdots\ S_n$

   ejecuta en secuencia los comandos S_1, S_2, ..., y S_n, justo en ese orden, donde $n \geq 2$. Ningún carácter en especial es usado en *GOLD* para separar los comandos entre sí, ni siquiera el punto y coma (‘`;`’) o el cambio de línea (‘`\n`’). Lo anterior implica que no se implementa el patrón *newline separators* ni el patrón *delimiter-directed translation* de Fowler [49]. Es posible que entre comandos consecutivos se necesiten blancos (*blanks*) para no ocasionar errores de sintaxis (e.g., si se eliminara el segundo espacio en el comando secuencial `var x,y x,y:=5,3`, se tendría un error de sintaxis: `var x,yx,y:=5,3`).

6. *Comandos condicionales*. Son instrucciones compuestas por subcomandos más simples donde a lo sumo uno de éstos es escogido para ser ejecutado, dependiendo de ciertas condiciones específicas [40]:

   - *Instrucción if-then* e *Instrucción if-then-else*. La instrucción

     ```
     if B1 then S1
     elseif B2 then S2
     ...
     elseif Bn then Sn
     else Sn+1
     end
     ```

     es un comando condicional *if-then-else* [†4] que opera de la siguiente manera:

     - si la guarda B_1 se cumple, entonces se ejecuta el comando S_1;
     - de lo contrario, si la guarda B_2 se cumple, entonces se ejecuta el comando S_2;
     - $\cdots$;
     - de lo contrario, si la guarda B_n se cumple, entonces se ejecuta el comando S_n;
     - de lo contrario, entonces se ejecuta el comando S_{n+1}.

     Se deben tener en cuenta las siguientes consideraciones:

     - $S_1, S_2, \ldots, S_n, S_{n+1}$ son comandos denominados *cuerpos*;
     - $B_1, B_2, \ldots, B_n$ son expresiones booleanas que representan condiciones denominadas *guardas*;
     - la sentencia `if` B_1 `then` S_1 es la *cláusula if*, cuyo cuerpo es el comando S_1;
     - cada sentencia `elseif` B_i `then` S_i es una *cláusula elseif* $(2 \leq i \leq n)$, cuyo cuerpo es el comando S_i;
     - la sentencia `else` S_{n+1} es la *cláusula else*, cuyo cuerpo es el comando S_{n+1};
     - la cláusula *if* es obligatoria, las cláusulas *else-if* son opcionales, y la cláusula *else* es opcional;
     - si alguna guarda no entrega un valor booleano después de ser evaluada, se produce un error de ejecución;
     - si únicamente se encuentra presente la cláusula *if*, entonces la instrucción se denomina *if-then*; y
     - si la cláusula *else* no está presente y ninguna de las guardas $B_1, B_2, \ldots, B_n$ se cumple, entonces no se realiza ninguna operación (en particular, no se termina la ejecución anormalmente).

[4] La palabra clave `elseif` se escribe de seguido, sin espacios. Si por accidente se pone `else if` en vez de `elseif`, se pueden producir errores de compilación que no tienen justificación aparente, debido al déficit de sentencias `end`.

- *Instrucción switch*. La instrucción

```
switch E begin
  case E1 : S1
  case E2 : S2
  ...
  case En : Sn
  default : Sn+1
end
```

es un comando condicional *switch* que opera de la siguiente manera [†5]:

- se evalúa la expresión E, que entrega un valor que es almacenado en una nueva variable auxiliar x;
- si la expresión $x = E_1$ es verdadera, entonces se ejecuta el comando S_1;
- de lo contrario, si la expresión $x = E_2$ es verdadera, entonces se ejecuta el comando S_2;
- $\cdots$;
- de lo contrario, si la expresión $x = E_n$ es verdadera, entonces se ejecuta el comando S_n;
- de lo contrario, entonces se ejecuta el comando S_{n+1}.

Se deben tener en cuenta las siguientes consideraciones:

- $S_1, S_2, \ldots, S_n, S_{n+1}$ son comandos denominados *cuerpos*;
- E es una expresión de cualquier tipo, denominada *expresión de control* [†6];
- $E_1, E_2, \ldots, E_n$ son expresiones de cualquier tipo, denominadas *casos*;
- cada sentencia `case` $E_i : S_i$ es una *cláusula case* ($1 \leq i \leq n$), cuyo cuerpo es el comando S_i;
- la sentencia `default` $: S_{n+1}$ es la *cláusula default*, cuyo cuerpo es el comando S_{n+1};
- debe haber por lo menos una cláusula *case*, y la cláusula *default* es opcional; y
- si la cláusula *default* no está presente y ninguna de las cláusulas *case* ejecuta su cuerpo, entonces no se realiza ninguna operación (en particular, no se termina la ejecución anormalmente).

7. *Comandos iterativos*. Son instrucciones (denominadas *ciclos* o *bucles*) que ejecutan repetitivamente un subcomando llamado *cuerpo* [40]:

- *Instrucción while*. La instrucción

```
while B do
  S
end
```

es un comando iterativo *while* que opera de la siguiente manera: mientras se cumpla la guarda B, ejecute el cuerpo S. Concretamente:

(1) si la guarda B se cumple, entonces:
  (1.1) se ejecuta el comando S; y
  (1.2) se regresa al paso 1.
(2) de lo contrario, se termina la ejecución del ciclo.

[5] Se debe recordar que el operador de igualdad en *GOLD* (=) sirve para determinar si dos objetos son iguales o no, cuyo equivalente en *Java* sería el método `equals`. Por otro lado, el operador de comparación en *Java* (==) en general sirve para determinar si dos apuntadores referencian al mismo objeto.

[6] En las sentencias *switch* de *Java 7* se deben usar instrucciones `break`, y únicamente se permiten expresiones de tipo `byte`, `Byte`, `short`, `Short`, `char`, `Character`, `int`, `Integer`, `String` (cadenas de texto), y `Enum` (tipos enumerados) [67]. En contraparte, las sentencias *switch* de *GOLD* no necesitan el uso de instrucciones `break`, y permiten expresiones de cualquier tipo.

Se deben tener en cuenta las siguientes consideraciones [†7]:

- S es un comando denominado *cuerpo*;
- B es una expresión booleana que representa una condición denominada *guarda*; y
- si alguna evaluación de la guarda no entrega un valor booleano, se produce un error de ejecución.

- *Instrucción do-while*. La instrucción

```
do
  S
whilst B
```

es un comando iterativo *do-while* [†8] que opera de la siguiente manera: ejecute el cuerpo S mientras se cumpla la guarda B. Concretamente:

(1) se ejecuta el comando S;

(2) si la guarda B se cumple, entonces se regresa al paso 1.

(3) de lo contrario, se termina la ejecución del ciclo.

Se deben tener en cuenta las mismas consideraciones que para la instrucción *while*.

- *Instrucción repeat-until*. La instrucción

```
repeat
  S
until B
```

es un comando iterativo *repeat-until* que opera de la siguiente manera: ejecute el cuerpo S hasta que se cumpla la guarda B. Concretamente:

(1) se ejecuta el comando S;

(2) si la guarda B no se cumple, entonces se regresa al paso 1.

(3) de lo contrario, se termina la ejecución del ciclo.

Se deben tener en cuenta las mismas consideraciones que para la instrucción *while*.

- *Instrucción for-each*. La instrucción

```
for each x:T ∈ E do
  S
end
```

es un comando iterativo *for-each* que opera de la siguiente manera: para todo elemento de la colección E, ejecute el cuerpo S. Concretamente:

(1) se declara una variable auxiliar con identificador x y tipo T, cuyo alcance sea el cuerpo del ciclo;

(2) por cada elemento que pertenezca a la colección E:

(2.1) se ejecuta el comando S, donde la variable x denota el elemento actualmente visitado.

Se deben tener en cuenta las siguientes consideraciones:

- S es un comando denominado *cuerpo*;
- E es una expresión denominada *colección iterada*;

[7] La instrucción `for(`*Inic*`;`*B*`;`*Incr*`){`*S*`}` de *Java* y *C++* puede simularse en *GOLD* con el comando *Inic* `while(`*B*`) do` *S Incr* `end`.

[8] La palabra reservada `while` fue reemplazada por `whilst`, pues el uso de la primera complica el análisis sintáctico de comandos de la forma ... `do` S_1 `while` B `do` S_2 ..., entrando en conflicto con la instrucción *while*. El término *whilst* es un término arcaico que se usa en el Reino Unido y en Australia como sinónimo de *while* [68].

- x es una variable usada para recorrer los elementos de la colección E, denominada *variable de iteración*;
- $:T$ es una sentencia opcional que declara el tipo T de la variable de iteración x;
- si algún elemento de la colección E no se puede convertir al tipo T, se produce un error de ejecución;
- en lugar del operador $\in$, puede escribirse la palabra reservada `in`;
- si no se especifica la sentencia $:T$, el tipo de la variable de iteración x sería `java.lang.Object`;
- si ya existía una variable declarada con el identificador x, entonces se produce un error de compilación;
- no se debe modificar la colección E mientras esté siendo recorrida, porque esto podría acarrear un error de ejecución; y
- la evaluación de la expresión E debe entregar un arreglo, o un objeto de tipo `java.lang.Iterable`, `java.util.Enumeration` o `java.lang.CharSequence` (de lo contrario, se produce un error de ejecución).

- *Instrucción for*. La instrucción

```
for x:=E1 to E2 by E3 do
  S
end
```

es un comando iterativo *for* que opera de la siguiente manera:

(1) se ejecuta la asignación simple $x:=E_1$;

(2) mientras la condición $x \leq E_2$ se cumpla:

(2.1) se ejecuta el comando S;

(2.2) se ejecuta la asignación simple $x:=x+E_3$.

Se deben tener en cuenta las siguientes consideraciones:

- S es un comando denominado *cuerpo*;
- $x:=E_1$ es una asignación simple denominada *inicialización*, sujeta a las condiciones establecidas anteriormente para todas las asignaciones;
- E_2 es el *límite superior* que restringe los valores que puede tomar la variable x;
- E_3 es el *incremento* que indica la cantidad en la que *aumenta* el valor de la variable x en cada iteración;
- la sentencia `by` E_3 es opcional, y se denomina *cláusula step*;
- si no se especifica la cláusula *step*, entonces E_3 toma el valor 1 (por defecto);
- si la variable x no es declarada antes del ciclo y su tipo no es especificado explícitamente en la asignación $x:=E_1$, entonces se le asigna por defecto el tipo `int`; y
- el alcance de la variable x está definido por la semántica operacional de la asignación simple $x:=E_1$: si x es una variable declarada antes del ciclo, entonces su alcance viene dado por su propia declaración (e.g., `var x` ... `for x:=`E_1 ...); de lo contrario, si x es una variable que no fue declarada antes del ciclo, entonces la asignación $x:=E_1$ declara implícitamente una nueva variable x cuyo alcance es únicamente el cuerpo del *for* (por ende, la variable x no podría ser usada por fuera del ciclo), inicializada con el valor entregado por la expresión E_1 (e.g., `for x:=`E_1 ...).

- *Instrucción for-downto*. La instrucción

```
for x:=E1 downto E2 by E3 do
  S
end
```

es un comando iterativo *for-downto* que opera de la siguiente manera:

(1) se ejecuta la asignación simple $x:=E_1$;

(2) mientras la condición $x \geq E_2$ se cumpla:

(2.1) se ejecuta el comando S;

(2.2) se ejecuta la asignación simple $x := x - E_3$.

Se deben tener en cuenta las mismas consideraciones que para la instrucción *for*, cambiando *límite superior* por *límite inferior*, *incremento* por *decremento*, y *aumenta* por *disminuye*.

8. *Otras instrucciones*. Son comandos que no están catalogados dentro de la clasificación de Watt [40]:

   - *Impresión en consola*. La instrucción

     `print` $E_1, E_2, \ldots, E_n$

     imprime en la consola del sistema una línea con el mensaje de texto dado por la lista no vacía de expresiones $E_1, E_2, \ldots, E_n$ (e.g., `print "El valor de x es ",x,"."`). Concretamente, el mensaje de texto que se produce es el resultado de concatenar las representaciones textuales de cada una de las expresiones de la lista $E_1, E_2, \ldots, E_n$, luego de invocar el método `toString()` sobre cada una de éstas. En particular, si todas las expresiones de la lista son cadenas de texto de tipo `java.lang.String`, el mensaje de texto impreso en consola es la concatenación de éstas. El método `System.out.println` de *Java* satisface el mismo objetivo.

   - *Código Java*. La sentencia

     `/?` S `?/`

     embebe textualmente el código fuente S escrito en *Java*, dentro del código fuente *GOLD*. El analizador semántico del compilador de *GOLD* concatena S dentro del código *Java* traducido, justo en el punto en el que fue embebido. Este mecanismo permite embeber *código nativo* escrito en *Java* dentro de programas *GOLD*, así como se puede embeber código nativo escrito en *Lenguaje Ensamblador* (*Assembler*) dentro de programas *C++*. Lo anterior hace posible la simulación de instruciones no provistas por *GOLD*, como los *try-catch* y los bloques *synchronized*.

Los comandos anteriormente mencionados exhiben un control de flujo con una única entrada y una única salida [40]. Para que el control de flujo pueda tener múltiples salidas, *GOLD* provee los siguientes secuenciadores, definidos con el apoyo de la teoría del libro de Watt [40]:

1. *Escapes*. Son secuenciadores que terminan inmediatamente la ejecución del comando o procedimiento sobre el que se encuentran [40]:

   - *Ruptura de ciclo*. La instrucción

     `break`

     termina la ejecución del ciclo cuyo bloque de comandos es el más pequeño que la incluye. Al contrario de *Java*, la instrucción `break` de *GOLD* no sirve para terminar la ejecución de un comando *switch*.

   - *Ruptura de iteración*. La instrucción

     `continue`

     termina la ejecución de la iteración actual del cuerpo del ciclo cuyo bloque de comandos es el más pequeño que la incluye. En otras palabras, la instrucción `continue` de *GOLD* sirve para continuar con la siguiente iteración de un ciclo, dejando de ejecutar el código posterior a su aparición dentro del cuerpo de éste.

- *Terminación de procedimiento propio*. La instrucción

  `finalize`

  termina la ejecución del cuerpo de un procedimiento propio, transfiendo el control de flujo de regreso al comando que realizó la invocación.

- *Instrucción de retorno*. La instrucción

  `return` E

  termina la ejecución del cuerpo de una función, retornando el valor obtenido como resultado de evaluar la expresión E, y transfiendo el control de flujo de regreso al comando que realizó la invocación. Múltiples valores pueden ser retornados si E representa una lista por enumeración, de la forma $\langle x_1, x_2, \ldots, x_n \rangle$.

2. *Excepciones*. Son secuenciadores que pueden ser usados para reportar situaciones anormales que impiden que el programa pueda continuar con su ejecución [40] [†9]:

   - *Lanzamiento de excepción*. La instrucción

     `throw` E

     termina abruptamente la ejecución de un programa *GOLD*, lanzando como excepción el resultado de evaluar la expresión E (e.g., `throw new IllegalArgumentException("El valor "+x+" no satisface la condición.")`). Si después de evaluar la expresión E no se obtuvo un valor de tipo `java.lang.Throwable`, entonces se produce un error en tiempo de ejecución a través del lanzamiento de una excepción de tipo `java.lang.ClassCastException`.

   - *Lanzamiento de error de ejecución*. La instrucción

     `error` $E_1, E_2, \ldots, E_n$

     termina abruptamente la ejecución de un programa *GOLD*, lanzando como excepción una instancia de la clase `java.lang.RuntimeException`, cuyo mensaje de error está dado por la lista no vacía de expresiones $E_1, E_2, \ldots, E_n$ (e.g., `error "El valor ",x," no satisface la condición."`). Concretamente, el mensaje de error que se produce es el resultado de concatenar las representaciones textuales de cada una de las expresiones de la lista $E_1, E_2, \ldots, E_n$, luego de invocar el método `toString()` sobre cada una de éstas. En particular, si todas las expresiones de la lista son cadenas de texto de tipo `java.lang.String`, el mensaje de error entregado es la concatenación de éstas.

   - *Terminación anormal*. La instrucción

     `abort`

     termina abruptamente la ejecución de un programa *GOLD*, lanzando como excepción una instancia de la clase `java.lang.RuntimeException`, cuyo mensaje de error es la cadena de texto `"Execution terminated abnormally"`.

[9] En *GOLD* no se suministra una instrucción *try-catch* que permita atrapar las excepciones lanzadas. Por ende, las excepciones terminan propagándose a través de las invocaciones, hasta que sean atrapadas por una instrucción *try-catch* escrita en *Java*, o hasta que sean lanzadas por un método *main* escrito en *Java* o por un procedimiento *main* escrito en *GOLD* (véase la sección §7.2.7). En éste último caso, se imprime en la consola del sistema la traza que describe la excepción lanzada, causando la terminación abrupta de la ejecución del programa.

- *Aserción*. La instrucción

  `assert` E

  es una sentencia que permite probar si la condición booleana E se cumple o no (e.g., `assert Math.PI>3`), tal como se hace en *Java* con las aserciones [69]. Después de evaluar la expresión E: (1) si no se obtuvo un valor de tipo `bool`, se lanza una excepción de tipo `java.lang.ClassCastException`; (2) si se obtuvo el valor `false`, se lanza una excepción de tipo `java.lang.AssertionError`, informando que la condición no se satisfizo; y (3) si se obtuvo el valor `true`, no se realiza ninguna operación, continuando así con la ejecución normal del programa. De esta manera, las aserciones permiten depurar la ejecución de un programa sin tener que lanzar excepciones explícitamente [†10].

Código 7.2. *Traza de ejemplo de una excepción lanzada por una instrucción* `abort` *en GOLD.*

```
1 Exception in thread "main" java.lang.RuntimeException: Execution terminated abnormally
2   at Example.hoo(Example.java:41)
3   at Example.goo(Example.java:31)
4   at Example.foo(Example.java:21)
5   at Example.main(Example.java:12)
```

Todos los comandos de *GOLD* son determinísticos, pues desarrollan una secuencia de pasos que es completamente predecible [40], siempre y cuando la evaluación de las expresiones también lo sea. La aclaración sobre la evaluación de las expresiones es necesaria porque, como en *GOLD* se pueden usar clases *Java* (e.g., `java.util.Random`) o invocar rutinas *Java* (e.g., `Math.random()`), es posible que el control de flujo de un programa *GOLD* termine siendo pseudo-imprecedible. No es realmente impredecible porque la generación de números al azar en *Java* es *pseudo-aleatoria* [70], ya que en la actualidad es realizada con procesos determinísticos sobre máquinas con componentes determinísticos. Por lo tanto, la aleatoriedad aparente que *Java* provee, es en realidad una pseudo-aleatoriedad determinística, lo que apoya la idea de que todos los comandos de *GOLD* son determinísticos.

7.2.6. Procedimientos

En *GOLD* se pueden declarar procedimientos de varias formas:

- *Procedimientos propios*. Son procedimientos que abstraen un comando a ser ejecutado [40] (posiblemente un comando secuencial), escritos siguiendo la sintaxis

  `procedure` *Name*(*Parameters*)`:void begin`
  Command
  `end`

  donde:

  - *Name* es un identificador que define el *nombre* del procedimiento propio.
  - *Parameters* es una lista (posiblemente vacía) que define los *parámetros* del procedimiento propio, escritos en la forma $P_1, P_2, \ldots, P_n$, donde cada parámetro P_i se declara usando cualquiera de las siguientes sentencias:
    - La sentencia x declara un *parámetro atipado* con identificador x y tipo `java.lang.Object` (e.g., `a`).
    - La sentencia x`:`T declara un *parámetro tipado* con identificador x y tipo T (e.g., `a:String`, `b:int[][]`).

[10] Por defecto, las aserciones en *GOLD* se encuentran habilitadas, y en *Java* deshabilitadas. Para activar las aserciones se debe poner como argumento de la *Máquina Virtual de Java* la opción `-enableassertions`, y para desactivarlas, la opción `-disableassertions`.

- :`void` es una sentencia opcional que indica que el procedimiento propio no tiene retorno (i.e., no entrega un resultado como respuesta).
- *Command* es un comando (posiblemente uno secuencial) que define el *cuerpo* del procedimiento propio.

En *GOLD*, los procedimientos propios también son denominados *procedimientos*, si no hay lugar a confusión con las funciones. Se deben tener en cuenta las siguientes consideraciones que aplican a todos los procedimientos propios en *GOLD*:

- en el cuerpo de todo procedimiento propio se pueden usar instrucciones `finalize` para terminar inmediatamente su ejecución, y transferir el control de flujo de regreso a la instrucción que realizó la invocación;
- si se termina la ejecución de un procedimiento propio sin invocar alguna instrucción `finalize`, se transfiere el control de flujo de regreso a la instrucción que realizó la invocación;
- cuando se realiza el proceso de compilación a *Java*, cada procedimiento propio es traducido en un método estático público que tiene retorno de tipo `java.lang.Object`, entregando siempre un apuntador nulo (`null`) como respuesta (excepto el procedimiento propio *main*, que es descrito más adelante); y
- si se invoca accidentalmente un procedimiento propio (o un método con retorno de tipo `void`) en un llamado a función para evaluar una expresión determinada, el resultado obtenido después de la invocación es un apuntador nulo (`null`).

- *Funciones*. Son procedimientos que abstraen una expresión a ser evaluada [40], escritos siguiendo la sintaxis

```
function Name(Parameters):ReturnType begin
  Command
end
```

donde:

- *Name* es un identificador que define el *nombre* de la función.
- *Parameters* es una lista que define los *parámetros* de la función, con la misma sintaxis descrita en los procedimientos propios.
- :*ReturnType* es una sentencia opcional que define el tipo de retorno *ReturnType* de la función (i.e., el tipo de las expresiones que la función entrega como resultado). Si no se especifica el tipo de retorno, se establece por defecto que es de tipo `java.lang.Object`.
- *Command* es un comando (posiblemente uno secuencial) que define el *cuerpo* de la función, cuyo propósito es calcular el valor de la expresión abstraída, entregándolo como resultado mediante una instrucción `return`.

Se deben tener en cuenta las siguientes consideraciones que aplican a todas las funciones en *GOLD*:

- la palabra reservada `function` es una *palabra irrelevante* [40] (i.e., una palabra opcional cuyo propósito es mejorar la legibilidad de la declaración);
- en el cuerpo de toda función se pueden usar instrucciones `return` para terminar inmediatamente su ejecución, entregar el valor calculado como resultado, y transferir el control de flujo de regreso a la instrucción que realizó la invocación;
- si se termina la ejecución de una función sin invocar alguna instrucción `return`, se entrega como resultado un apuntador nulo (`null`), y se transfiere el control de flujo de regreso a la instrucción que realizó la invocación;
- una función puede retornar varios valores a través de colecciones descritas por enumeración o por comprensión (e.g., varios elementos enumerados en una lista); y

- cuando se realiza el proceso de compilación a *Java*, cada función es traducida en un método estático público cuyo tipo de retorno es el que haya sido declarado en la función.

- *Macros*. Son funciones escritas siguiendo la sintaxis

  `function` *Name*(*Parameters*):*ReturnType* := *Expression*

  para abreviar la declaración

  ```
  function Name(Parameters):ReturnType begin
    return Expression
  end
  ```

  Además de las consideraciones descritas para las funciones, se deben tener en cuenta las siguientes para las macros en *GOLD*:

  - se puede prescindir del uso de la palabra reservada `function`, pues es una *palabra irrelevante*; y
  - para el operador de definición de macro se puede usar el símbolo = en vez de :=.

  Los anteriores hechos permiten declarar macros en la forma *Name*(*Parameters*):=*Expression*, o incluso en la forma *Name*(*Parameters*)=*Expression*. Vale la pena advertir que las macros en *GOLD* no actúan como las macros `#define` de *C++* (sustituciones de texto), sino como procedimientos que retornan el resultado de evaluar una expresión.

Código 7.3. *Merge-sort [1] implementado en GOLD.*

```
1  procedure merge(A,B,p,q,r) begin
2    // Merge A[p..q] and A[q+1..r] into B[p..r]
3    i,j,k:=p,q+1,p
4    while i≤q or j≤r do
5      if j>r or (i≤q and A[i]≤A[j]) then
6        B[k],i:=A[i],i+1
7      else
8        B[k],j:=A[j],j+1
9      end
10     k:=k+1
11   end
12   // Copy B[p..r] into A[p..r]
13   for k:=p to r do
14     A[k]:=B[k]
15   end
16 end
17 procedure mergeSort(A,B,p,r) begin
18   if p<r then
19     q:=⌊(p+r)/2⌋
20     mergeSort(A,B,p,q)    // Sort the first half
21     mergeSort(A,B,q+1,r)  // Sort the second half
22     merge(A,B,p,q,r)      // Merge the two sorted halves
23   end
24 end
25 procedure mergeSort(A) begin
26   mergeSort(A,GToolkit.clone(A),0,|A|-1)
27 end
```

Código 7.4. *Función de Fibonacci implementada recursivamente en GOLD, con números de precisión arbitraria.*

```
function fib(n) begin
  if n=0 then
    return ℤ(0)
  elseif n=1 then
    return ℤ(1)
  else
    return fib(n-1)+fib(n-2)
  end
end
```

Código 7.5. *Macro GOLD que implementa recursivamente la función de Fibonacci, usando el tipo* `long`*.*

```
fib(n) := n=0?0L:(n=1?1L:fib(n-1)+fib(n-2))
```

Código 7.6. *Función de Fibonacci implementada iterativamente en GOLD, con números de precisión arbitraria.*

```
function fib(n) begin
  var a:ℤ(0),b:ℤ(1)
  for i:=1 to n do
    a,b:=b,a+b
  end
  return a
end
```

GOLD adopta el siguiente mecanismo de paso de parámetros a los procedimientos: los valores de tipo primitivo se pasan *por valor* [40] y el resto de valores (i.e., los objetos) se pasan *por referencia* [40]. Además, si un procedimiento reasigna el valor de un parámetro, el cambio no es visto por el procedimiento que efectuó la invocación [55]. De esta manera, el comportamiento de *GOLD* en relación al paso de parámetros es el mismo que el de *Java*.

La *signatura* de un procedimiento describe su nombre, la cantidad de parámetros que recibe, y el tipo de cada uno de éstos justo en el orden en el que fueron declarados (e.g., `merge(Object,Object,Object,Object,Object)`, `mergeSort(Object,Object,Object,Object)`, `mergeSort(Object)`, `main(String[])`, `foo()`, `foo(int)`, `foo(double)`, `foo(int[])`, `goo(int,String)`, `goo(String,int)`). Nótese que en la signatura de un procedimiento no importa el nombre de sus parámetros, pero sí el tipo de cada uno y el orden en el que fueron declarados. En *GOLD* se pueden declarar procedimientos con el mismo nombre pero con diferente signatura (i.e., con parámetros distintos), fomentando así el *polimorfismo ad-hoc* [71].

Todo programa *GOLD* que vaya a ser ejecutado como una aplicación debe tener declarado un procedimiento propio denominado *main*, cuya signatura sea `main(String[])` (i.e., un procedimiento con nombre `main`, que no tenga retorno, y que reciba un solo parámetro (de tipo `String[]`)). Cuando el procedimiento propio *main* finaliza su ejecución (posiblemente debido a la acción de una instrucción `finalize`), también termina la ejecución del programa, sin transferir el control de flujo de regreso a la instrucción que realizó la invocación (porque esto no tiene sentido). Durante el proceso de compilación a *Java*, el procedimiento propio *main* de un programa *GOLD* es traducido en un método estático público con retorno de tipo `void` y con la signatura `main(String[])` (i.e., en un método *main* escrito en *Java* con el encabezado `public static void main(String[] args)`, donde *args* es el nombre del único parámetro del procedimiento propio *main* en *GOLD*).

Para facilitar el manejo de tuplas, *GOLD* suministra las siguientes abreviaciones, que actúan como azúcar sintáctico:

1. *Tuplas como variables ligadas*. Para fomentar el uso de tuplas en comprensiones y cuantificaciones, se diseñaron las siguientes convenciones para describir los fragmentos que componen los rangos:

- El fragmento $\langle x_1,x_2,\ldots,x_n\rangle{=}E$ abrevia los fragmentos $x{=}E$, $x_1{=}x[0]$, $x_2{=}x[1]$, $\ldots$, $x_n{=}x[n{-}1]$, donde x es una variable auxiliar, y $n\geq 1$ (e.g., `(Σa,b|⟨a,b⟩=⟨3,5⟩:a*b)` denota `(Σx,a,b|x=⟨3,5⟩,a=x[0],b=x[1]:a*b)` = `(Σx|x=⟨3,5⟩:x[0]*x[1])` = `(Σa,b|a=3,b=5:a*b)` = `3*5` = `15`).
- El fragmento $\langle x_1,x_2,\ldots,x_n\rangle{\in}E$ abrevia los fragmentos $x{\in}E$, $x_1{=}x[0]$, $x_2{=}x[1]$, $\ldots$, $x_n{=}x[n{-}1]$, donde x es una variable auxiliar, y $n\geq 1$ (e.g., `(Σa,b|⟨a,b⟩∈(1..4)×(6..7):a*b)` denota `(Σx,a,b|x∈(1..4)×(6..7),a=x[0],b=x[1]:a*b)` = `(Σx|x∈(1..4)×(6..7):x[0]*x[1])` = `1*6+1*7+2*6+2*7+3*6+3*7+4*6+4*7` = `130`).

2. *Tuplas como variables de iteración*. Para simplificar los recorridos sobre colecciones compuestas por tuplas, se estableció la siguiente nomenclatura (donde x es una variable auxiliar, y $n\geq 1$):

   - La variable de iteración de una instrucción *for-each* puede ser de la forma $\langle x_1,x_2,\ldots,x_n\rangle$. Concretamente,

     ```
     for each ⟨x1,x2,...,xn⟩ ∈ E do
       S
     end
     ```

     es una abreviación de

     ```
     for each x ∈ E do
       x1,x2,...,xn := x[0],x[1],...,x[n−1]
       S
     end
     ```

3. *Tuplas como parámetros de procedimiento*. Para permitir la declaración de parámetros en forma de tuplas, se proporcionaron los siguientes mecanismos (donde x es una variable auxiliar, y $n\geq 1$):

   - Para procedimientos propios,

     ```
     procedure Name(...,⟨x1,x2,...,xn⟩,...):void begin
       Command
     end
     ```

     es una abreviación de

     ```
     procedure Name(...,x,...):void begin
       x1,x2,...,xn := x[0],x[1],...,x[n−1]
       Command
     end
     ```

   - Para funciones,

     ```
     function Name(...,⟨x1,x2,...,xn⟩,...):ReturnType begin
       Command
     end
     ```

     es una abreviación de

     ```
     function Name(...,x,...):ReturnType begin
       x1,x2,...,xn := x[0],x[1],...,x[n−1]
       Command
     end
     ```

- Para macros,

  `function` $Name(\ldots,\langle x_1,x_2,\ldots,x_n\rangle,\ldots):ReturnType := Expression$

  es una abreviación de

  ```
  function Name(...,x,...):ReturnType begin
    x1,x2,...,xn := x[0],x[1],...,x[n-1]
    return Expression
  end
  ```

7.2.7. Programas

Un programa *GOLD* está representado por el código fuente de un archivo con extensión `.gold` que se encuentra ubicado dentro del directorio `src` de un proyecto del usuario, y tiene la estructura

Annotations
Package
Imports
StaticVariables
StaticProcedures
EmbeddedJavaBlocks

donde:

1. *Annotations* es una lista (posiblemente vacía) de *anotaciones* de la forma:

   - `@SuppressWarnings`$(Code)$, donde *Code* es una cadena de texto que identifica las advertencias de compilación que desean ocultarse; o
   - `@SuppressWarnings`$(\{Code_1,Code_2,\ldots,Code_n\})$, donde $Code_1,Code_2,\ldots,Code_n$ son las cadenas de texto que identifican las advertencias de compilación que desean ocultarse.

   Existen varias cadenas de texto que pueden incluirse dentro de las sentencias *SuppressWarnings* para ocultar determinadas advertencias de compilación sobre el código fuente de un programa *GOLD*:

   - `"types"` para ocultar las advertencias de compilación relacionadas con el sistema de tipado estático;
   - `"imports"` para ocultar las advertencias de compilación relacionadas con la importación de paquetes;
   - `"main"` para ocultar las advertencias de compilación que ocurren cuando se declaran procedimientos con nombre `main` cuya signatura no tiene la forma `main(String[])`; y
   - `"errors:access"` para ocultar los errores de compilación que ocurren cuando se acceden variables sin declarar implícita o explícitamente, permitiendo el uso de variables declaradas en código *Java* embebido.

2. *Package* es una sentencia opcional de la forma `package` *directory*, donde *directory* es el nombre calificado correspondiente al subdirectorio donde se encuentra el programa dentro de la carpeta `src` del proyecto del usuario (e.g., `org.kernel.util`), reemplazando *slashes* (‘/’) y *backslashes* (‘\’) por puntos (‘.’). Una sentencia *package* sirve para declarar explícitamente el paquete donde se encuentra un determinado programa *GOLD*, ayudando así a organizar los archivos del usuario de una manera modular. El *nombre calificado completo* (*fully qualified name*) de un programa *GOLD* es el nombre de su paquete y el nombre de su archivo, separados por un punto (e.g., `org.kernel.util.Foo`); o simplemente el nombre de su archivo (e.g., `Foo`), si se encuentra ubicado en la raíz del directorio `src` (en cuyo caso, no se debe declarar el paquete). En realidad, el usuario no está obligado a declarar los paquetes de sus programas *GOLD*, porque éstos se infieren

automáticamente a partir de la ubicación que tienen sus archivos dentro del proyecto. De hecho, es preferible que no se especifique el paquete de los programas *GOLD*, puesto que esto facilitaría moverlos entre distintos directorios sin tener que cambiar manualmente la declaración de su paquete, como sí ocurre con las clases *Java*. Además, si la ubicación del archivo no coincide con el paquete declarado, se produce un error de compilación.

3. *Imports* es una lista (posiblemente vacía) de sentencias *import* de la forma:

   - *Importación de clases*. `import` *Class*, donde *Class* es el nombre calificado completo (*fully qualified name*) de la clase *Java* o del programa *GOLD* que se desea importar (e.g., `import java.util.LinkedList`). En este caso, después de importar la clase, en el programa *GOLD* se puede hacer mención a ésta a través de su nombre simple (e.g., `LinkedList`).
   - *Importación de paquetes*. `import` *Package*`.*`, donde *Package* es el nombre calificado (*qualified name*) del paquete que se desea importar (e.g., `import java.util.*`). En este caso, después de importar el paquete, en el programa *GOLD* se puede hacer mención a cualquiera de sus clases a través de su nombre simple (e.g., `ArrayList`, `LinkedList`, `TreeMap`, `TreeSet`).
   - *Súper-importación de paquetes*. `import` *Prefix*`.**`, donde *Prefix* es el nombre calificado (*qualified name*) que indica el prefijo de los paquetes que se desean importar (e.g., `import java.**`). En este caso, después de importar la colección de paquetes con el prefijo dado, en el programa *GOLD* se puede hacer mención a cualquier clase que esté dentro de cualquier paquete que tenga como prefijo la cadena *Prefix* seguida de un punto (i.e., cualquier clase que pertenezca al paquete con nombre *Prefix* o a algún subpaquete de éste, recursivamente), a través de su nombre simple (e.g., `JFrame`, `LinkedList`, `Pattern`).

   En vez de la palabra reservada `import` (tomada de *Java*), también puede usarse `include` (como en *C++*) o `using` (como en *C#*). La súper-importación de paquetes es una característica exclusiva de *GOLD*, que simplifica la labor de importar decenas o cientos de paquetes mediante una sola sentencia (e.g., `import java.**` importa todas las clases de *Java* cuyo nombre calificado comience por `java.`) [†11]. Por otro lado, la importación de paquetes y clases se comporta de la misma manera que en *Java*:

   - si se importa una entidad que no existe en el *classpath* de la aplicación, se produce un error de compilación;
   - independientemente de los paquetes importados en el programa, toda clase puede accederse sin ambigüedad mediante su nombre calificado completo (e.g., `java.util.LinkedList`);
   - la repetición de una sentencia *import* previamente declarada produce una advertencia de compilación que en *GOLD* puede ocultarse a través de la anotación `@SuppressWarnings("imports")`;
   - en caso de que existan clases con el mismo nombre que pertenezcan a dos paquetes distintos que hayan sido importados en el mismo programa, hay dos mecanismos para resolver la ambigüedad: importar la clase explícitamente y usar su nombre simple; o, mencionar la clase a través de su nombre calificado completo; y
   - si se importan dos clases distintas que tengan el mismo nombre simple, se genera un error de compilación producto de una *colisión de nombres* (e.g., `import java.awt.List` colisiona con `import java.util.List`).

   De esta manera, todas las clases implementadas en *Java* se pueden mencionar en los programas escritos en *GOLD*. Por defecto, hay tres paquetes que se importan automáticamente en todo programa *GOLD*:

   - `java.lang`, que es el paquete que se importa implícitamente en toda clase *Java*;

[11] En *Java* es muy frecuente ver programas con una multitud de sentencias *import*, que en *GOLD* podrían resumirse en unas pocas súper-declaraciones de paquetes. Por ejemplo, trabajando con *Java 6*, la súper-importación `import java.util.**` abrevia la importación de diez paquetes individuales: `java.util`, `java.util.concurrent`, `java.util.concurrent.atomic`, `java.util.concurrent.locks`, `java.util.jar`, `java.util.logging`, `java.util.prefs`, `java.util.regex`, `java.util.spi` y `java.util.zip`. Más dramáticamente, las súper-importaciones `java.**` y `javax.**` comprenden en conjunto 167 paquetes individuales y alrededor de 3000 clases.

- `org.apfloat`, que es el paquete correspondiente a la librería *Apfloat* [53] (véase la sección §6.2.4), que provee números de precisión arbitraria para implementar los tipos primitivos numéricos particulares a *GOLD* ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$ y $\mathbb{C}$); y
- `gold.util`, que es un paquete que contiene clases que implementan rutinas generales para apoyar el proceso de ejecución de programas *GOLD* (véase la sección §A.6.2.20).

4. *StaticVariables* es una lista (posiblemente vacía) de sentencias *var*, que sirven para efectuar las declaraciones de las variables *globales* o *estáticas* del programa.

5. *StaticProcedures* es una lista (posiblemente vacía) de declaraciones de procedimiento, incluyendo procedimientos propios, funciones y macros, que siempre son *globales* o *estáticos*. Si se pretende que el programa sea ejecutado como una aplicación, debe tener declarado un procedimiento propio con la signatura `main(String[])` (i.e., un procedimiento sin retorno con nombre `main`, que reciba un solo parámetro de tipo `String[]`).

6. *EmbeddedJavaBlocks* es una lista (posiblemente vacía) de sentencias de la forma `/?` *S* `?/`, donde *S* es código fuente escrito en *Java*, representando un *bloque de código Java embebido*. De manera similar a lo descrito en la sección §7.2.5, este mecanismo permite embeber código *Java* dentro de programas *GOLD* para declarar procedimientos adicionales, variables globales especiales y *clases anidadas* [60] (*nested classes*), sin necesidad de tener que hacerlo en una clase *Java* por separado.

Las variables globales, los procedimientos estáticos y los bloques de código *Java* embebido pueden aparecer de forma mezclada en un programa *GOLD*, permitiendo que sean definidos en cualquier orden (e.g., una declaración de variable global puede aparecer entre dos declaraciones de procedimiento). Vale la pena anotar que, cada programa *GOLD* es traducido a una clase *Java*, donde las variables globales son convertidas en atributos estáticos públicos, los procedimientos en métodos estáticos públicos, y los bloques de código *Java* embebido son pasados textualmente justo en el lugar donde fueron insertados, relativo al resto de declaraciones. Esto permite que los procedimientos escritos en *GOLD* puedan ser invocados desde *Java* o desde otros programas implementados en *GOLD*.

Además, en *GOLD* no hay ningún carácter que oficie como separador de sentencias, de instrucciones o de declaraciones (de variables o procedimientos), ni siquiera el cambio de línea (aunque así parezca en los ejemplos presentados). En particular, esto implica que no se implementa el patrón *newline separators* de Fowler [49], haciendo posible que un programa *GOLD* pueda estar contenido completamente en una sola línea de código.

El símbolo distinguido de la gramática de *GOLD* es el símbolo no terminal `GoldProgram` (véase la sección §A.1.1), que define los programas que pueden ser escritos en *GOLD*.

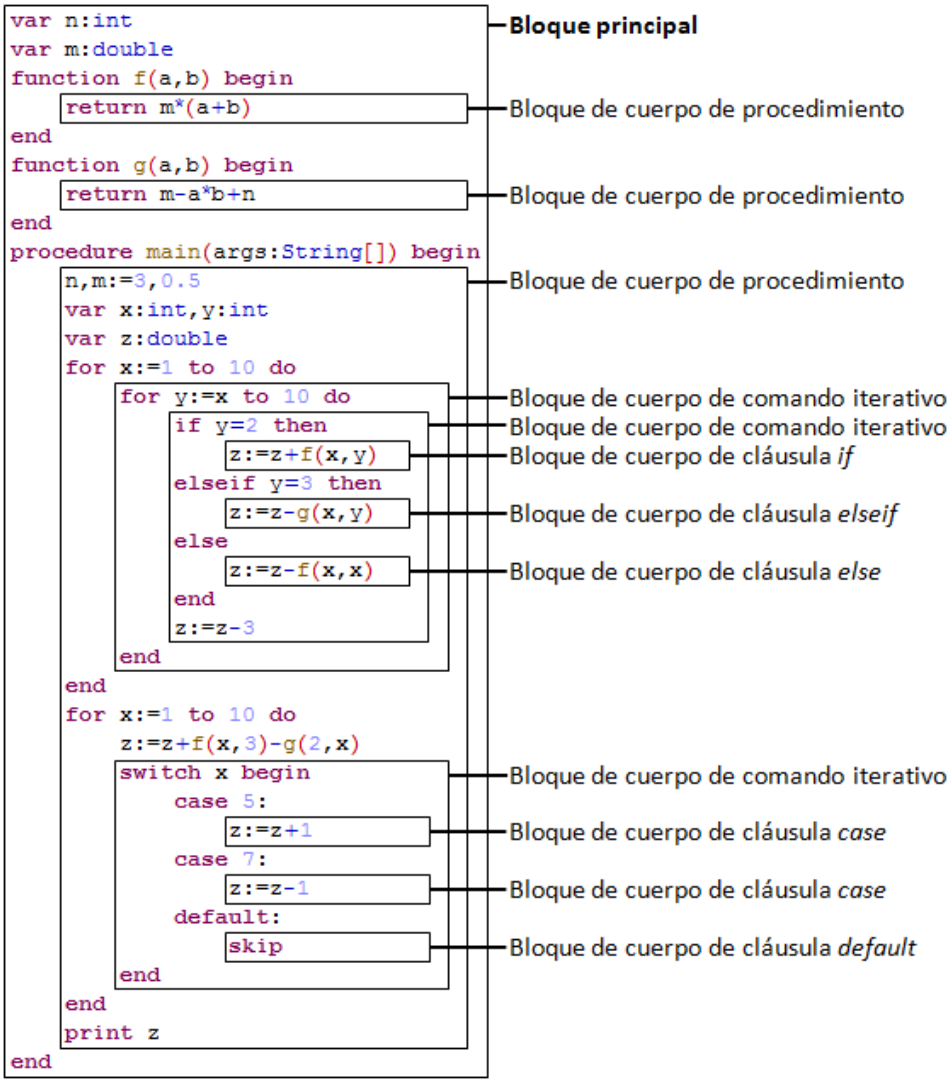
7.2.8. Alcance (*scope*)

GOLD tiene una *estructura de bloques anidados* (*nested block structure*) como la de *C* (véase la figura 7.2), donde ‘‘los cuerpos de los procedimientos no se pueden traslapar, pero los bloques de comando pueden estar libremente anidados dentro de los cuerpos de los procedimientos’’ [40]. Un *bloque* en *GOLD* es:

- todo un programa *GOLD* (denominado *bloque principal*);
- el cuerpo de un procedimiento (ya sea un procedimiento propio o una función);
- el cuerpo de un comando iterativo (*while*, *do-while*, *repeat-until*, *for-each*, *for* o *for-downto*);
- el cuerpo de una cláusula *if*, *elseif* o *else* de un comando condicional *if-then-else*; o
- el cuerpo de una cláusula *case* o *default* de un comando condicional *switch*.

De esta forma, cada bloque en *GOLD* es una parte del programa que puede incluir declaraciones locales [40], donde el bloque más externo (que es el que corresponde a todo el programa) se denomina *bloque principal*. Las variables se pueden declarar en cualquier bloque, mientras que los procedimientos únicamente se pueden declarar en el bloque principal. Toda variable que sea declarada en el bloque principal sería una *variable global* cuyo tiempo de vida es todo el tiempo de ejecución del programa [40], y en contraparte, toda variable que sea declarada en un bloque que no sea el principal sería una *variable local* cuyo tiempo de vida es una activación del bloque que contiene su declaración [40]. Por ende, las variables globales son declaradas para ser usadas en cualquier parte del programa, y las variables locales son declaradas para ser usadas únicamente dentro de su bloque [40].

Figura 7.2. *Estructura de bloques en GOLD 3.*



El alcance de cualquier declaración de variable local en *GOLD* es el bloque más pequeño donde se encuentra, incluyendo todos sus bloques anidados, pero excluyendo el código fuente anterior a la declaración. Por otro lado, el alcance de cualquier declaración de variable global en *GOLD* es todo el programa, excepto las declaraciones de variables globales previas (permitiendo que las variables globales puedan ser usadas en todos los procedimientos, pero evitando que puedan ser usadas para inicializar variables globales declaradas anteriormente, mediante invocaciones a constructor (e.g., `var x:Integer(y+2) var y:Integer(5)` no es permitido)). Finalmente, el alcance de cualquier declaración de procedimiento en *GOLD* es todo el programa donde se encuentra, incluyendo el código fuente anterior

a la declaración (permitiendo que los procedimientos puedan ser invocados dentro del cuerpo de cualquier otro, o en la inicialización de cualquier variable global). En todo caso, cada vez que una variable global sea accedida antes de que sea inicializada (lo que puede suceder si se inicializa una variable global invocando un procedimiento estático que acceda a una variable global declarada más adelante), se obtiene un apuntador nulo.

Además, como cada programa *GOLD* termina traduciéndose en una clase *Java* donde los procedimientos en *GOLD* son convertidos en métodos estáticos públicos *Java*, entonces el alcance de una declaración de procedimiento en *GOLD* terminaría siendo todo el proyecto del usuario. Lo anterior permite que las rutinas implementadas en todo programa *GOLD* se puedan invocar desde cualquier clase *Java* o desde cualquier otro programa *GOLD*. Asimismo, el alcance de una declaración de variable global en *GOLD* también es todo el proyecto del usuario, puesto que las variables globales en *GOLD* son convertidas en atributos estáticos públicos *Java*. Hay que enfatizar que las variables globales y procedimientos estáticos pueden ser accedidos en cualquier procedimiento, sin importar si aparece antes o después que las respectivas declaraciones.

Por ejemplo, véase el programa 7.7, que ilustra una función para calcular la desviación estándar de un conjunto de datos numéricos. El alcance de las variables n, u y μ es el fragmento de código que se encuentra entre las líneas 3 y 13 (ambos límites inclusive), y el alcance de las variables s y σ son las líneas 8, 9, 10, 11, 12 y 13. De manera similar, el alcance de la variable x declarada en el primer ciclo es la línea 5, y el alcance de la variable x declarada en el segundo ciclo es la línea 10. Por último, el alcance del parámetro *data* es todo el cuerpo de la función, que está entre las líneas 2 y 13 (ambos límites inclusive). El programa 7.7 fue diseñado (a propósito) para ilustrar el alcance de las variables en *GOLD*; versiones más cortas del procedimiento se exhiben en los ejemplos subsiguientes.

Código 7.7. *Cálculo de la desviación estándar de un conjunto de datos, declarando variables explícitamente.*

```
1  function standardDeviation(data) begin
2    var n,u,μ
3    n,u:=|data|,0
4    for each x∈data do
5      u:=u+x
6    end
7    var s,σ
8    s,μ:=0,u/n
9    for each x∈data do
10     s:=s+(x-μ)^2
11   end
12   σ:=(s/n)^0.5
13   return σ
14 end
```

Código 7.8. *Cálculo de la desviación estándar de un conjunto de datos, declarando variables implícitamente.*

```
1  function standardDeviation(data) begin
2    n,u:=|data|,0
3    for each x∈data do
4      u:=u+x
5    end
6    s,μ:=0,u/n
7    for each x∈data do
8      s:=s+(x-μ)^2
9    end
10   σ:=(s/n)^0.5
11   return σ
12 end
```

Código 7.9. *Cálculo de la desviación estándar de un conjunto de datos, usando cuantificaciones.*

```
function standardDeviation(data) begin
  n := |data|
  μ := (Σx|x∈data:x)/n
  σ := sqrt((Σx|x∈data:(x-μ)^2)/n)
  return σ
end
```

Código 7.10. *Macro ineficiente que calcula la desviación estándar de un conjunto de datos.*

```
standardDeviation(data) := sqrt((Σx|x∈data:(x-(Σy|y∈data:y)/|data|)^2)/|data|)
```

GOLD es un lenguaje que tiene *alcance estático* (*statically scoped*) [40], puesto que los procedimientos son ejecutados en un espacio de nombres propio a su definición, que no se ve alterado por el espacio de nombres del comando que realizó la invocación. De esta manera, el alcance de cada declaración se puede decidir en tiempo de compilación [40], como se describió anteriormente.

Las expresiones iterativas en *GOLD* (i.e., las colecciones descritas por comprensión y las cuantificaciones) actúan como *bloques de expresión* (*block expressions*) [40], cuyas declaraciones locales corresponden a las *variables ligadas* (*dummies*) [12], y cuyas subexpresiones corresponden a su rango y cuerpo. Esto implica que el alcance de cada variable ligada es únicamente la expresión iterativa que la declara, ya sea una comprensión o una cuantificación. Por otro lado, las cláusulas de los comandos condicionales (i.e., las cláusulas *if*, *elseif*, *else*, *case* y *default*), los comandos iterativos (i.e., *while*, *do-while*, *repeat-until*, *for-each*, *for* y *for-downto*) y los procedimientos (i.e., procedimientos propios y funciones) en *GOLD* actúan como *bloques de comando* (*block commands*) [40], cuyas declaraciones locales corresponden a las variables declaradas dentro de su propio bloque, y cuyos subcomandos corresponden a las instrucciones que conforman su cuerpo.

Con respecto al espacio de nombres, se tienen las siguientes condiciones en *GOLD*:

- No pueden existir dos procedimientos declarados con la misma signatura. Si se declara un procedimiento con la misma signatura de un procedimiento previamente declarado, se lanza un error de compilación.
- El procedimiento propio *main* debería tener la signatura `main(String[])`. De lo contrario, se lanza una advertencia de compilación.
- No pueden existir dos variables declaradas con el mismo identificador cuyos alcances se intersequen. Si se declara una variable con el mismo identificador de una variable previamente declarada en su mismo bloque o en algún bloque externo, se lanza un error de compilación.
- No pueden existir dos variables ligadas declaradas con el mismo identificador cuyos alcances se intersequen. Si se declara una variable ligada dentro de una expresión iterativa que declare un *dummy* con el mismo identificador, se lanza un error de compilación.
- No puede existir una variable declarada con el mismo nombre de un procedimiento y viceversa. De lo contrario, se lanza un error de compilación.

7.3. Semántica

En la sección §7.2 se describió detalladamente la semántica operacional del lenguaje *GOLD*, a medida que iba definiéndose su sintaxis. En esta sección se comentan algunos detalles adicionales sobre la semántica de los programas escritos en *GOLD*.

7.3.1. Semántica Denotacional

El analizador semántico, que forma parte del compilador de *GOLD*, es el componente responsable de traducir programas *GOLD* en código fuente escrito en *Java*, que se almacena en archivos con extensión `.java`. Por otro lado, el compilador estándar de *Java* (`javac`) convierte código fuente *Java* en un tipo de código intermedio denominado *bytecode*, que se aloja en archivos binarios con extensión `.class`. Posteriormente, el *bytecode* puede ser interpretado por la *Máquina Virtual de Java* (*JVM*: *Java Virtual Machine*), que es la que termina dándole cierta semántica a los programas *GOLD*.

Sean *GOLD*, *Java* y *Bytecode* tres conjuntos, donde:

- *GOLD* es el conjunto que contiene las secuencias de caracteres que representan programas bien formados, de acuerdo con la sintaxis de *GOLD* en formato *EBNF* descrita en la sección §A.1.1;
- *Java* es el conjunto que contiene las secuencias de caracteres que representan código fuente *Java* (implementando una sola clase pública), según la sintaxis específica a *Java 6*; y
- *Bytecode* es el conjunto que contiene las secuencias de bytes que representan *bytecode* binario que puede ser interpretado por la *JVM*.

El analizador semántico de *GOLD* puede verse como una función denotacional $f : GOLD \rightarrow Java$, tal que $f(q)$ representa la implementación de la clase *Java* que corresponde al resultado de efectuar el proceso de traducción sobre el programa *GOLD* p. De manera similar, compilador estándar de *Java* puede verse como una función denotacional $g : Java \rightarrow Bytecode$, tal que $g(q)$ es el *bytecode* binario correspondiente al código fuente *Java* dado en q. Si componemos ambas funciones, obtenemos una función $h = g \circ f : GOLD \rightarrow Bytecode$ que recibe programas *GOLD* y entrega *bytecode* binario. Dado un programa *GOLD* q, se tiene que $h(q) = g(f(q))$ es el *bytecode* binario que simula el programa q, que puede ser directamente ejecutado sobre la *JVM*.

De esta manera, la función h dotaría a *GOLD* de una semántica denotacional sobre una máquina abstracta muy compleja: la *Máquina Virtual de Java*. Claramente, existen muchas otras formas de definir una semántica denotacional para *GOLD*. Por ejemplo, se podría definir una función denotacional para traducir programas *GOLD* en programas escritos en el *Lenguaje de Comandos Guardados*.

El *Lenguaje de Comandos Guardados* (abreviado *GCL* por sus siglas en inglés: *Guarded Command Language*) fue inventado por Edsger Dijkstra para facilitar el estudio formal de los algoritmos a través de un conjunto reducido de instrucciones [72]. Se usará la sintaxis de *GCL* descrita en el libro *Programming: The Derivation of Algorithms* de Anne Kaldewaij [72] para definir el codominio de la nueva función denotacional.

Sea *GCL* el conjunto que contiene las secuencias de caracteres que representan programas bien formados en el lenguaje *GCL*, siguiendo la sintaxis de Kaldewaij [72], y sea $\xi : Gold \rightarrow GCL$ una función denotacional cuyo dominio son los programas escritos en *GOLD* y cuyo codominio son los programas escritos en *GCL*. No se puede pretender definir ξ sin restringir la sintaxis de *GOLD*, porque *GOLD* permite la invocación de rutinas implementadas en *Java* y el uso de secuenciadores (e.g., `break`, `continue`), que complican bastante el control de flujo. Entonces, se trabajará sobre un subconjunto de *GOLD* denominado *GOLD*$'$, que únicamente comprende instrucciones vacías, terminaciones anormales, asignaciones, intercambios, comandos secuenciales, comandos condicionales y comandos iterativos.

Fórmula 1. *Traducción de la instrucción vacía, de GOLD′ a GCL.*

$$\xi(\texttt{skip}) \quad = \quad \textbf{skip}$$

Fórmula 2. *Traducción de la terminación anormal, de GOLD′ a GCL.*

$$\xi(\texttt{abort}) \quad = \quad \textbf{abort}$$

Fórmula 3. *Traducción de la asignación simple, de GOLD′ a GCL.*

$$\xi(x := E) \quad = \quad x := E$$

Fórmula 4. *Traducción de la asignación simultánea, de GOLD′ a GCL.*

$$\xi(x_1, x_2, \ldots, x_n := E_1, E_2, \ldots, E_n) \quad = \quad x_1, x_2, \ldots, x_n := E_1, E_2, \ldots, E_n$$

Fórmula 5. *Traducción del intercambio, de GOLD′ a GCL.*

$$\xi(\texttt{swap } x \texttt{ with } y) \quad = \quad x, y := y, x$$

Fórmula 6. *Traducción del comando secuencial, de GOLD′ a GCL.*

$$\xi(S_1\ S_2\ \cdots\ S_n) \quad = \quad S_1; S_2; \cdots; S_n$$

Fórmula 7. *Traducción de la instrucción condicional if-then, de GOLD′ a GCL.*

$$\xi\left(\begin{array}{l}\texttt{if } B \texttt{ then } S \\ \texttt{end}\end{array}\right) \quad = \quad \begin{array}{l}\textbf{if } B \ \rightarrow\ S \\ [\,]\ \neg B \ \rightarrow\ \textbf{skip} \\ \textbf{fi}\end{array}$$

Fórmula 8. *Traducción de la instrucción condicional if-then-else, de GOLD′ a GCL.*

$$\xi\left(\begin{array}{l}\texttt{if } B_1 \texttt{ then } S_1 \\ \texttt{elseif } B_2 \texttt{ then } S_2 \\ \cdots \\ \texttt{elseif } B_n \texttt{ then } S_n \\ \texttt{else } S_{n+1} \\ \texttt{end}\end{array}\right) \quad = \quad \begin{array}{ll}\textbf{if } B_1 & \rightarrow\ S_1 \\ [\,]\ \neg B_1 \wedge B_2 & \rightarrow\ S_2 \\ \cdots & \\ [\,]\ \neg B_1 \wedge \neg B_2 \wedge \cdots \wedge \neg B_{n-1} \wedge B_n & \rightarrow\ S_n \\ [\,]\ \neg B_1 \wedge \neg B_2 \wedge \cdots \wedge \neg B_{n-1} \wedge \neg B_n & \rightarrow\ \textbf{skip} \\ \textbf{fi} & \end{array}$$

Fórmula 9. *Traducción de la instrucción condicional switch, de GOLD′ a GCL.*

$$\xi\left(\begin{array}{l}\texttt{switch } E \texttt{ begin}\\ \quad\texttt{case } E_1 : S_1\\ \quad\texttt{case } E_2 : S_2\\ \quad\cdots\\ \quad\texttt{case } E_n : S_n\\ \quad\texttt{default} : S_{n+1}\\ \texttt{end}\end{array}\right) = \begin{array}{ll} x := E; & \\ \textbf{if} & x = E_1 \rightarrow S_1\\ [] & x \neq E_1 \wedge x = E_2 \rightarrow S_2\\ \cdots & \\ [] & x \neq E_1 \wedge x \neq E_2 \wedge \cdots \wedge x = E_n \rightarrow S_n\\ [] & x \neq E_1 \wedge x \neq E_2 \wedge \cdots \wedge x \neq E_n \rightarrow S_{n+1}\\ \textbf{fi} & \end{array}$$

Fórmula 10. *Traducción de la instrucción repetitiva while, de GOLD′ a GCL.*

$$\xi\left(\begin{array}{l}\texttt{while } B \texttt{ do}\\ \quad S\\ \texttt{end}\end{array}\right) = \begin{array}{l}\textbf{do } B \rightarrow S\\ \textbf{od}\end{array}$$

Fórmula 11. *Traducción de la instrucción repetitiva do-while, de GOLD′ a GCL.*

$$\xi\left(\begin{array}{l}\texttt{do}\\ \quad S\\ \texttt{whilst } B\end{array}\right) = \begin{array}{l} S;\\ \textbf{do } B \rightarrow S\\ \textbf{od}\end{array}$$

Fórmula 12. *Traducción de la instrucción repetitiva repeat-until, de GOLD′ a GCL.*

$$\xi\left(\begin{array}{l}\texttt{repeat}\\ \quad S\\ \texttt{until } B\end{array}\right) = \begin{array}{l} S;\\ \textbf{do } \neg B \rightarrow S\\ \textbf{od}\end{array}$$

Fórmula 13. *Traducción de la instrucción repetitiva for-each sobre una colección, de GOLD′ a GCL.*

$$\xi\left(\begin{array}{l}\texttt{for each } x \in \{v_1, v_2, \ldots, v_n\} \texttt{ do}\\ \quad S\\ \texttt{end}\end{array}\right) = \begin{array}{ll} i := 1; & \\ \textbf{do } i \leq n \rightarrow & x := v_i;\\ & S;\\ & i := i + 1\\ \textbf{od} & \end{array}$$

Fórmula 14. *Traducción de la instrucción repetitiva for sobre una colección, de GOLD′ a GCL.*

$$\xi\left(\begin{array}{l}\texttt{for } x := E_1 \texttt{ to } E_2 \texttt{ by } E_3 \texttt{ do}\\ \quad S\\ \texttt{end}\end{array}\right) = \begin{array}{ll} x := E_1; & \\ \textbf{do } x \leq E_2 \rightarrow & S;\\ & x := x + E_3\\ \textbf{od} & \end{array}$$

Fórmula 15. *Traducción de la instrucción repetitiva for-downto sobre una colección, de GOLD′ a GCL.*

$$\xi\left(\begin{array}{l}\texttt{for } x := E_1 \texttt{ downto } E_2 \texttt{ by } E_3 \texttt{ do}\\ \quad S\\ \texttt{end}\end{array}\right) \quad = \quad \begin{array}{l} x := E_1;\\ \textbf{do } x \geq E_2 \rightarrow S;\\ \qquad\qquad\quad x := x - E_3\\ \textbf{od}\end{array}$$

7.3.2. Semántica Axiomática

Usando la función denotacional ξ definida en la sección §7.3.1 y la teoría de verificación de algoritmos estudiada en libro *Programming: The Derivation of Algorithms* de Anne Kaldewaij [72], se pueden enunciar teoremas de corrección de programas para las instrucciones del lenguaje *GOLD′*, dotando a *GOLD* de una semántica axiomática básica.

Definir una semántica axiomática para *GOLD* es mucho más complicado porque incluye llamados a procedimiento, invocación de rutinas implementadas en *Java* y secuenciadores que complican el control de flujo de los programas.

7.3.3. Semántica Operacional

En la sección §7.2 se expuso detalladamente la semántica operacional de *GOLD* mientras se iba describiendo la sintaxis de cada una de sus sentencias. Usando la función denotacional ξ definida en la sección §7.3.1 se pueden diseñar diagramas de flujo para ilustrar la semántica operacional de las distintas instrucciones de *GOLD′*, así como se hace en el libro *Verificación y Desarrollo de Programas* de Rodrigo Cardoso [73] para el lenguaje *GCL*.

Capítulo 8

Implementación

En este capítulo se describen los principales aspectos relacionados con la implementación de la infraestructura del lenguaje *GOLD 3*, incluyendo su *entorno de desarrollo integrado* (*IDE* por sus siglas en inglés: *integrated development environment*), su compilador (conformado por el analizador léxico, el analizador sintáctico y el analizador semántico), y la librería especializada que apoya el desarrollo de programas a través de múltiples implementaciones de las estructuras de datos más importantes así como los componentes gráficos para manipularlas y visualizarlas. La aplicación fue implementada en el lenguaje de programación *Java* como un *plug-in* de *Eclipse* [7] bajo el *framework Xtext* [6] para así permitir la codificación y ejecución de programas escritos en *GOLD* aprovechando las funcionalidades inherentes a *Eclipse*, las características brindadas por *Xtext* y la potencia del *API* estándar de *Java*. La combinación de herramientas *Java-Eclipse-Xtext* permitió construir una infraestructura completa, idónea para satisfacer exitosamente los requerimientos impuestos (véase el capítulo §5) de acuerdo con los lineamientos definidos en el diseño (véase el capítulo §7).

El código fuente del producto se encuentra distribuido en el directorio `/Sources` bajo los siguientes proyectos *Xtext*:

1. `org.gold.dsl`: contiene la implementación del núcleo del lenguaje y de los aspectos no visuales del *IDE*.
2. `org.gold.dsl.lib`: contiene los empaquetados *JAR* de las librerías *JUNG* [21] y *Apfloat* [53].
3. `org.gold.dsl.tests`: eventualmente alojará las pruebas que se vayan a realizar sobre el núcleo del lenguaje.
4. `org.gold.dsl.ui`: contiene la implementación de los aspectos visuales del *IDE*.

La implementación del lenguaje *GOLD* se divide en tres grandes componentes que serán descritos por separado: su *IDE*, su núcleo y su librería de clases.

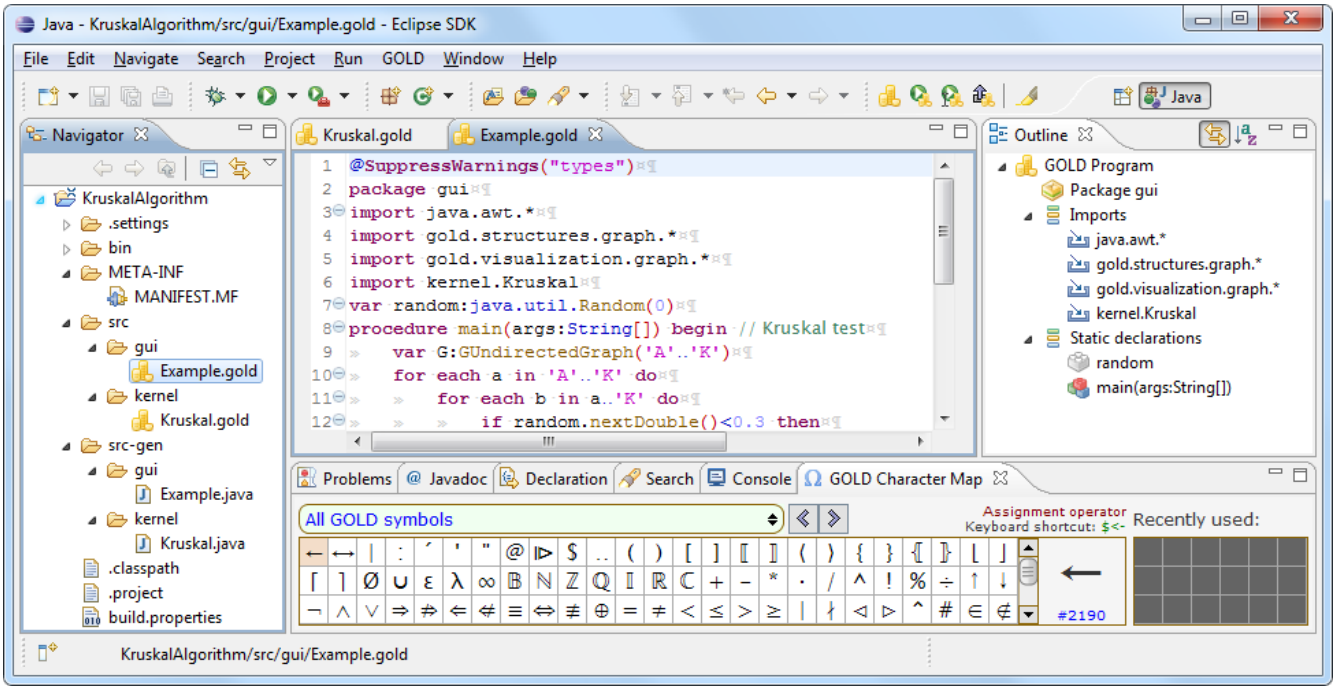
8.1. Entorno de desarrollo integrado (*IDE*)

Un componente esencial de un lenguaje de programación es su *IDE* (*integrated development environment*), que debe ser completo, maduro, portable, intuitivo y fácil de usar. Claramente, la calidad de un lenguaje se puede degradar profundamente si no cuenta con una herramienta adecuada que facilite la implementación, ejecución, depuración, mantenimiento y distribución de los programas escritos en su sintaxis. Como de nada sirve tener un lenguaje potente que no provea mecanismos para agilizar el proceso de desarrollo, fue necesario gastar un gran porcentaje del esfuerzo en diseñar una interfaz gráfica de usuario (*GUI*: *Graphical User Interface*) comparable con los entornos de desarrollo de grandes lenguajes de programación como *Java* y *C++*.

Para facilitar el cumplimiento de la mayoría de los requerimientos relacionados con el ambiente de programación se implementó la aplicación como un *plug-in* de *Eclipse* [7] desarrollado en la plataforma provista por el *framework*

Xtext 2.2.1 [6]. De esta manera se reutilizó una gran cantidad de componentes ya existentes, reduciendo considerablemente el trabajo que debía realizarse. Específicamente, *Xtext* apoyó la programación del *plug-in* generando automáticamente el analizador léxico y sintáctico del lenguaje, un metamodelo de clases para representar los elementos sintácticos del modelo semántico [49] del lenguaje mediante una estructura arbórea denominada *Abstract Syntax Tree* (*AST*), y el cascarón básico de un entorno de desarrollo sofisticado [6] basado en *Eclipse* que incluye un editor de texto especializado y funcionalidades como el resaltado de la sintaxis (*syntax highlighting*), el indentamiento automático del código fuente (*code formatting*), el emparejamiento de paréntesis (*bracket matching*), la validación de la sintaxis resaltando los errores de compilación en tiempo de desarrollo (*code validation*), el despliegue de ayudas de contenido (*content assist*), el completado automático de código (*code completion*) y la navegación sobre el modelo que describe la estructura semántica de un programa (*outline view*), entre otros. Además, dado que *GOLD* termina siendo un *plug-in* más de *Eclipse*, se pueden rescatar todas las funcionalidades suministradas por este ambiente de desarrollo y se puede aprovechar el uso de otros *plug-ins*.

Figura 8.1. *IDE de GOLD 3, embebido dentro de Eclipse.*



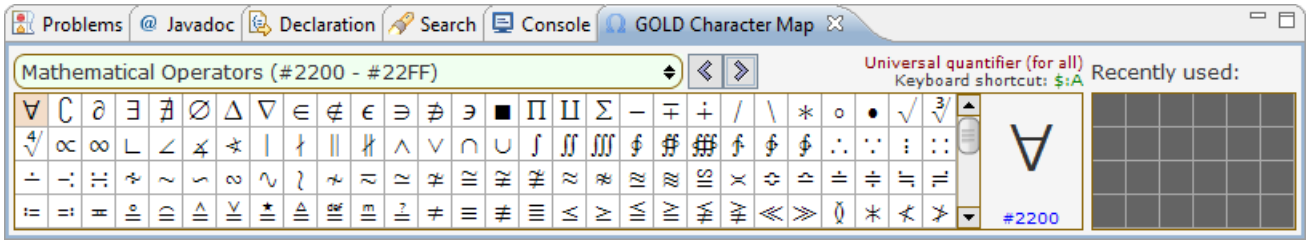
Aunque con el uso de *Xtext* se logró ahorrar una gran cantidad de trabajo, el código fuente generado automáticamente por *Xtext* no fue suficiente para resolver todos los detalles técnicos inherentes al *IDE*. Por lo tanto, para terminar la implementación de la infraestructura fue necesario implementar algunos componentes especializados y configurar exhaustivamente los diferentes aspectos del ambiente de desarrollo a través de los mecanismos ofrecidos por *Xtext*. Otro módulo que debió ser implementado fue el compilador que traduce código *GOLD* a código *Java*, que será descrito más adelante en la sección §8.2.

8.1.1. Tipografía (*font*)

Para la codificación de programas en *GOLD* es necesario disponer de un conjunto de tipografías adecuadas que incluyan símbolos comúnmente utilizados en dominios especializados como el cálculo proposicional, el cálculo de predicados, la teoría de números y la teoría de conjuntos. Sin estas tipografías los usuarios tendrían que recordar códigos alfanuméricos no intuitivos para cada uno de los símbolos de las constantes y de los operadores, sufriendo confusiones y retrasos al momento de usar el editor de texto porque no estarían en capacidad de expresarse en la

notación matemática estándar a la que pueden estar acostumbrados. Por ejemplo, el conjunto $\sim((A \cup B) \cap (C \triangle D)) \setminus \varnothing$ es más fácil de recordar en la notación tradicional que usando convenciones alfanuméricas foráneas con cadenas del estilo ‘‘not((*A* union *B*) inter (*C* simdiff *D*)) minus empty’’ o del estilo ‘‘!((*A* U *B*)&(*C* $ *D*))−0’’.

Figura 8.2. *Mapa de caracteres de GOLD 3, desplegando la categoría de operadores matemáticos.*



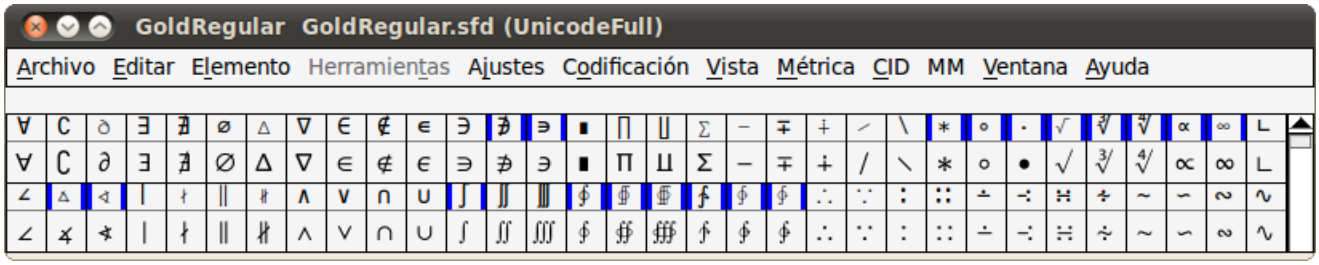
La totalidad de los símbolos matemáticos del alfabeto de la gramática de *GOLD* están codificados con el estándar *Unicode* y están incluidos en los tipos de letra [†1] *DejaVu Sans* [74], *Lucida Sans Regular* [75] y *STIX General* [76], cuya manipulación se simplificó después de mezclarlos en un solo tipo de letra, denominado *Gold Regular*.

Tabla 8.1. *Tipos de letra TrueType y OpenType que conforman la tipografía Gold Regular.*

Nombre	Formato	Versión	Fecha
DejaVu Sans	*TrueType*	2.33	2011/02/27
Lucida Sans Regular	*TrueType*		2010/09/29
STIX General	*OpenType*	1.0.0	2010/12/29

Para mezclar los tipos de letra *DejaVu Sans*, *Lucida Sans Regular* y *STIX General* se usó *FontForge* [77], que es un editor de tipos de letra libre distribuido bajo la licencia *BSD*, que provee un lenguaje de *scripting* [78] para automatizar los procesos de creación de nuevas tipografías aún si están basadas en tipos de letra ya existentes.

Figura 8.3. *Porción del tipo de letra Gold Regular, visualizada a través del editor gráfico de FontForge [77].*



Mediante una rutina implementada en el lenguaje de *scripting* de *FontForge* [78] se mezclaron los tres tipos de letra de forma controlada, escogiendo selectivamente qué caracteres [†2] se incluían de cada uno de éstos. En el directorio `/Data/Fonts` se encuentra el código fuente de la rutina y los tipos de letra mencionados.

Tabla 8.2. *Contenido del directorio* `/Data/Fonts` *de GOLD 3.*

Archivo	Descripción
`DejaVuSans.ttf`	Tipo de letra *DejaVu Sans* en formato *TrueType* (extensión `.ttf`).
`DejaVuSans.licence`	Licencia de uso del tipo de letra *DejaVu Sans*.
`LucidaSansRegular.ttf`	Tipo de letra *Lucida Sans Regular* en formato *TrueType* (extensión `.ttf`).

[1] El término *tipo de letra* hace referencia al anglicismo *fuente*, que viene de la palabra inglesa *font*.

[2] Según la Real Academia Española, un *carácter* (cuyo plural es *caracteres*, sin tilde) es una *señal o marca que se imprime, pinta o esculpe en algo* o un *signo de escritura o de imprenta*.

`LucidaSansRegular.licence`	Licencia de uso del tipo de letra *Lucida Sans Regular*, heredada de *JRE*.
`STIXGeneral.otf`	Tipo de letra *STIX General* en formato *OpenType* (extensión `.otf`).
`STIXGeneral.license`	Licencia de uso del tipo de letra *STIX General*.
`GoldRegular.pe`	Rutina escrita en *FontForge* que genera el tipo de letra *Gold Regular*.
`GoldRegular.sh`	*Script bash* para ejecutar el archivo `GoldRegular.pe` en sistemas *Linux*.
`GoldRegular.output`	Salida generada por *FontForge* luego de ejecutar `GoldRegular.sh`.
`GoldRegular.sfd`	Tipo de letra *Gold Regular* generado automáticamente por *FontForge*, en formato *Spline Font Database* (extensión `.sfd`).
`GoldRegular.ttf`	Tipo de letra *Gold Regular* generado automáticamente por *FontForge*, en formato *TrueType Font* (extensión `.ttf`).

Para ejecutar el *script* `GoldRegular.pe` basta con instalar el paquete `fontforge` en un sistema *Linux* mediante algún administrador de paquetes y correr el comando `GoldRegular.sh`. El tipo de letra generado tiene el nombre *Gold Regular*, se encuentra en formato *TrueType* y es compatible con sistemas *Windows* y *Unix*.

Código 8.1. *Script FontForge que genera el tipo de letra Gold Regular.*

```
# Abrir el tipo de letra "DejaVu Sans":
Open("DejaVuSans.ttf",1);
# Eliminar caracteres no deseados:
Select(0u0100,0u22FF,0u2B00,0uFFFF);
Clear();
# Mezclar con el tipo de letra "Lucida Sans Regular":
MergeFonts("LucidaSansRegular.ttf",1);
# Copiar el carácter 120128 a la posición 0x2148 (tipo irracional):
Select(120128); Copy(); Select(0u2148); Paste();
# Copiar el carácter 120121 a la posición 0x212C (tipo booleano):
Select(120121); Copy(); Select(0u212C); Paste();
# Copiar el carácter 0x004F a la posición 0x2375 (big-oh):
Select(0u004F); Copy(); Select(0u2375); Paste();
# Copiar el carácter 0x006F a la posición 0x2376 (small-oh):
Select(0u006F); Copy(); Select(0u2376); Paste();
# Copiar el carácter 0x03A9 a la posición 0x2377 (big-omega):
Select(0u03A9); Copy(); Select(0u2377); Paste();
# Copiar el carácter 0x03C9 a la posición 0x2378 (small-omega):
Select(0u03C9); Copy(); Select(0u2378); Paste();
# Copiar el carácter 0x0398 a la posición 0x2379 (big-theta):
Select(0u0398); Copy(); Select(0u2379); Paste();
# Poner todos los símbolos de complejidad en cursiva:
Select(0u2375,0u2379);
Skew(20);
# Eliminar los caracteres más allá del rango de 16 bits:
Select(65536,1114944);
Clear();
# Eliminar todos los caracteres que no estén en los rangos
# 0x0000-0x052F, 0x1D00-0x23FF y 0x2460-0x2BFF:
Select(0u0530,0u1CFF,0u2400,0u245F,0u2C00,0uFFFF);
Clear();
# Asignar el nombre al nuevo tipo de letra:
SetFontNames("GoldRegular","GoldRegular","GoldRegular","Regular");
# Exportar el nuevo tipo de letra al archivo "GoldRegular.sfd":
Save("GoldRegular.sfd");
# Exportar el nuevo tipo de letra al archivo "GoldRegular.ttf":
# (la opción 128 genera tablas compatibles en Apple y Microsoft)
Generate("GoldRegular.ttf");
Generate("GoldRegular.otf");
```

Para cubrir una mayor cantidad de símbolos del estándar *Unicode*, contar con varios estilos de letra, y tener la posibilidad de presentar caracteres monoespaciados, el *IDE* de *GOLD* utiliza intensivamente los tipos de letra *Lucida Sans Unicode*, *Dialog* [†3], *Gold Regular* y *Courier New*. La clase `org.gold.dsl.ui.util.GoldDSLFont` representa una enumeración de todas las tipografías mencionadas, que se explotan para desplegar programas escritos en *GOLD* y para presentar cualquier texto a través de la interfaz gráfica.

Figura 8.4. *Tipos de letra utilizados por el IDE de GOLD 3.*

(a) *Lucida Sans Unicode.*

Hello World! 01234ABCDEabcdeαβγδε(){}□□□□∪∩∆⊆⊄⊇⊉℘∀∃Σ∏ℬℕℤℚ□ℝℂ□□□□□□□

(b) *Dialog.*

Hello World! 01234ABCDEabcdeαβγδε(){}□□□□∪∩∆⊆⊄⊇⊉℘∀∃Σ∏ℬℕℤℚ□ℝℂ❁☎☀あば漢字

(c) *Gold Regular.*

Hello World! 01234ABCDEabcdeαβγδε(){}⟨⟩⦃⦄∪∩∆⊆⊄⊇⊉℘∀∃Σ∏𝔹ℕℤℚ𝕀ℝℂ❁☎☀□□□□

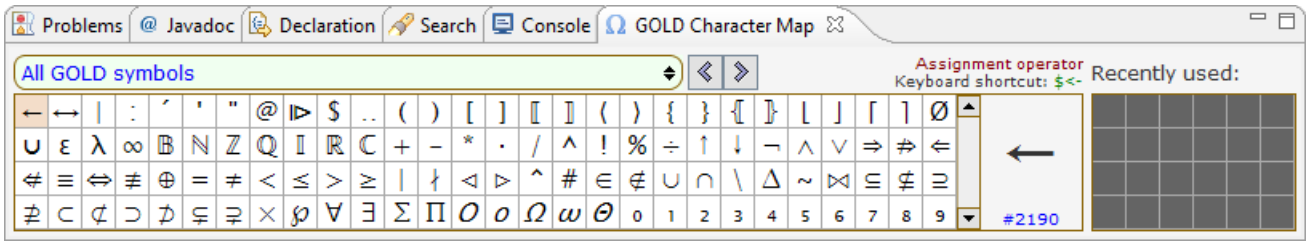
(d) *Courier New.*

Hello World! 01234ABCDEabcdeαβγδε(){}□□□□□∩∆□□□□□□□∑∏□□□□□□□□□□□□□□

8.1.2. Mapa de caracteres (*character map*)

La sintaxis del lenguaje *GOLD* permite la escritura de algoritmos usando una gran cantidad de operadores matemáticos para manipular expresiones booleanas, números, conjuntos, bolsas y secuencias.

Figura 8.5. *Mapa de caracteres de GOLD 3, desplegando la categoría de símbolos predeterminada.*



Para facilitar la edición de programas y promover el uso de la notación matemática estándar, se implementó un mapa de caracteres (véase la figura 8.5) que reúne un sinnúmero de símbolos especiales codificados en el estándar *Unicode*, que se encuentran organizados en varias categorías que son desplegadas a través de tablas de símbolos. Cada uno de los caracteres puede ser insertado en el editor de texto haciendo doble clic en su celda correspondiente de la tabla de símbolos, o escribiendo con el teclado un determinado atajo alfanumérico (*keyboard shortcut*). Lo anterior evita tener que desplazar el puntero del ratón hasta la ubicación exacta que ocupa su respectivo glifo, acelerando así la escritura de algoritmos.

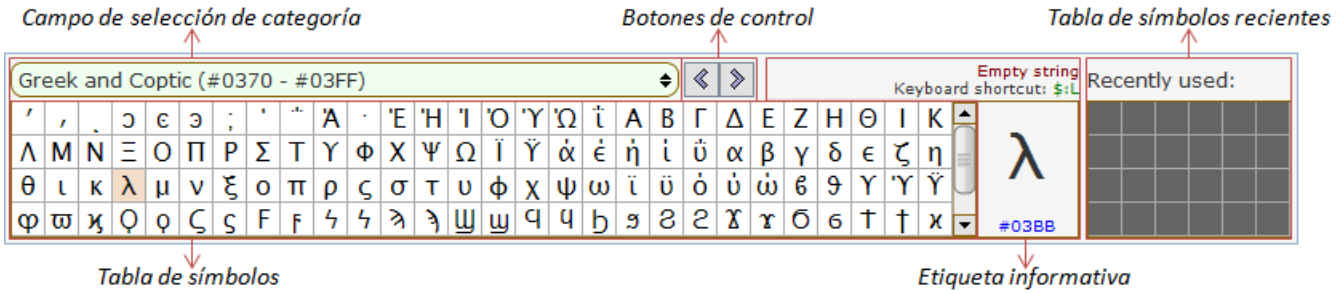
La interfaz gráfica del mapa de caracteres de *GOLD* está basada en la diseñada en el sistema *LOGS2005* [79], dividiéndose en varios subcomponentes:

- un campo de selección para escoger la categoría de caracteres;
- dos botones de control para seleccionar la anterior y la siguiente categoría de la lista mostrada por el campo de selección;

[3] *Dialog* es el tipo de letra predeterminado en *Java*, que es construido por la *Máquina Virtual de Java* mezclando varias fuentes físicas instaladas en el sistema operativo subyacente.

- una tabla de símbolos que despliega los caracteres pertenecientes a la categoría actualmente seleccionada;
- una etiqueta de texto que informa el nombre del carácter seleccionado de la tabla de símbolos, indica con color verde la cadena de texto que puede ser escrita para transcribir su símbolo al editor de texto, magnifica su glifo al 250 % y exhibe con color azul su código *Unicode* en base hexadecimal precedido por el signo '#'; y
- un tabla con los veinticuatro símbolos más recientemente usados, ordenados del más al menos reciente.

Figura 8.6. *Componentes gráficos que hacen parte del mapa de caracteres de GOLD 3.*



El juego de caracteres del lenguaje *GOLD* comprende ciento doce símbolos especiales que están clasificados en once *categorías principales* [†4].

Tabla 8.3. *Categorías principales brindadas por el mapa de caracteres de GOLD 3.*

Nombre	Descripción
All GOLD symbols	Reúne todos los símbolos del lenguaje *GOLD*, exceptuando los dígitos numéricos, las letras del alfabeto latino y las letras del alfabeto griego.
Technical symbols	Reúne símbolos técnicos en general.
Constants	Reúne símbolos que representan constantes matemáticas.
Basic types	Reúne símbolos que representan conjuntos matemáticos básicos.
Arithmetic operators	Reúne símbolos que denotan operadores aritméticos.
Boolean operators	Reúne símbolos que denotan operadores booleanos.
Comparison operators	Reúne símbolos que denotan operadores de comparación entre valores numéricos.
Collection operators	Reúne símbolos que denotan operadores sobre conjuntos, bolsas y secuencias.
Quantifiers	Reúne símbolos que representan cuantificadores.
Computational complexity	Reúne símbolos usados en la teoría de complejidad computacional.
Subscripts	Reúne símbolos que representan subíndices numéricos.

Además, para ofrecer compatibilidad con otros sistemas de escritura y no descartar el uso de símbolos adicionales dentro de los comentarios de los programas, se incluyeron algunas *categorías secundarias* adicionales que corresponden estrictamente con determinados rangos de caracteres provistos por el estándar *Unicode*.

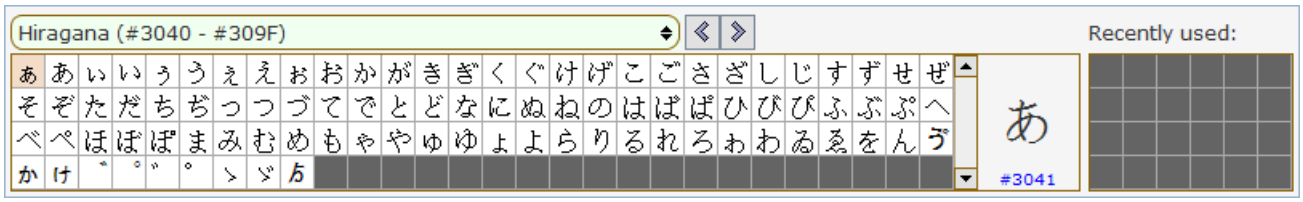
Tabla 8.4. *Categorías adicionales brindadas por el mapa de caracteres de GOLD 3.*

Rango *Unicode*	Nombre oficial en inglés
`0x0020-0x007F`	*Basic Latin*
`0x00A0-0x00FF`	*Latin-1 Supplement*
`0x0370-0x03FF`	*Greek and Coptic*
`0x2000-0x206F`	*General Punctuation*
`0x2070-0x209F`	*Superscripts and Subscripts*
`0x20A0-0x20CF`	*Currency Symbols*

[4] La sección §A.5.1 presenta cada uno de los ciento doce símbolos con su correspondiente código *Unicode*, atajo de teclado y descripción.

0x2100-0x214F	*Letterlike Symbols*
0x2150-0x218F	*Number Forms*
0x2190-0x21FF	*Arrows*
0x2200-0x22FF	*Mathematical Operators*
0x2300-0x23FF	*Miscellaneous Technical*
0x2460-0x24FF	*Enclosed Alphanumerics*
0x25A0-0x25FF	*Geometric Shapes*
0x2600-0x26FF	*Miscellaneous Symbols*
0x2700-0x27BF	*Dingbats*
0x27C0-0x27EF	*Miscellaneous Mathematical Symbols-A*
0x27F0-0x27FF	*Supplemental Arrows-A*
0x2900-0x297F	*Supplemental Arrows-B*
0x2980-0x29FF	*Miscellaneous Mathematical Symbols-B*
0x2A00-0x2AFF	*Supplemental Mathematical Operators*
0x3040-0x309F	*Hiragana*
0x30A0-0x30FF	*Katakana*
0x0020-0xFFFF	*All characters*

Figura 8.7. *Silabario Hiragana para ofrecer compatibilidad con la escritura japonesa.*



El paquete `gold.dsl.ui.charmap` contiene las clases que implementan el mapa de caracteres como una vista del entorno de programación *Eclipse*:

- `GoldDSLCharacterGroup`. Representa una categoría de caracteres cuyos atributos son su nombre y la lista de caracteres *Unicode* que contiene.

- `GoldDSLRecentCharacters`. Representa la estructura de datos que administra la lista con los símbolos más recientemente usados, mediante un caché *LRU* (*Least Recently Used*).

- `GoldDSLCharactersTable`. Representa una tabla de símbolos capaz de desplegar los caracteres pertenecientes a la categoría seleccionada o a la lista de símbolos más recientemente usados.

- `GoldDSLCharacterManager`. Representa el administrador de todas las categorías de caracteres, registrando para cada carácter su descripción, su atajo de teclado y el mejor tipo de letra que es capaz de desplegarlo, escogido semiautomáticamente [†5].

- `GoldDSLCharacterMap`. Representa el mapa de caracteres de *GOLD*, implementado a través de un componente gráfico *Swing* [80] que contiene el campo de selección de categoría, los botones de control, la tabla de símbolos, la etiqueta informativa y la tabla de símbolos recientes.

[5] La tipografía de los símbolos *GOLD* se seleccionó manualmente buscando la que ofreciera la mejor visualización, y la tipografía de cada uno de los demás símbolos *Unicode* se determinó con un proceso automático que encuentra el primero de los siguientes tipos de letra que es capaz de desplegarlo: *Lucida Sans Unicode*, *Dialog*, *GoldRegular*, *Courier New*.

- `GoldDSLCharacterMapView`. Representa la vista *Eclipse* que alberga el mapa de caracteres, implementando el patrón *Adapter* para poder tratar el componente gráfico *Swing* [80] como un componente gráfico *SWT* [81] (*Standard Widget Toolkit*) que actúe como una vista instalable en el entorno de programación *Eclipse*.

Figura 8.8. *Diagrama de clases del paquete* `gold.dsl.ui.charmap`.

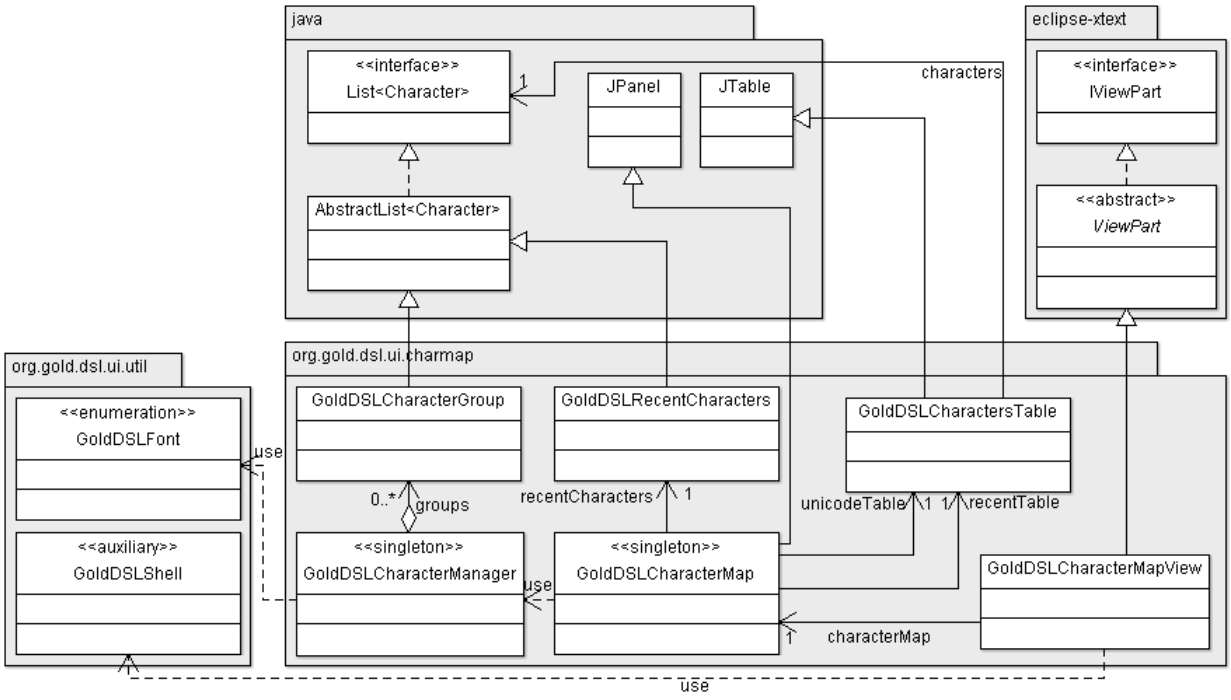
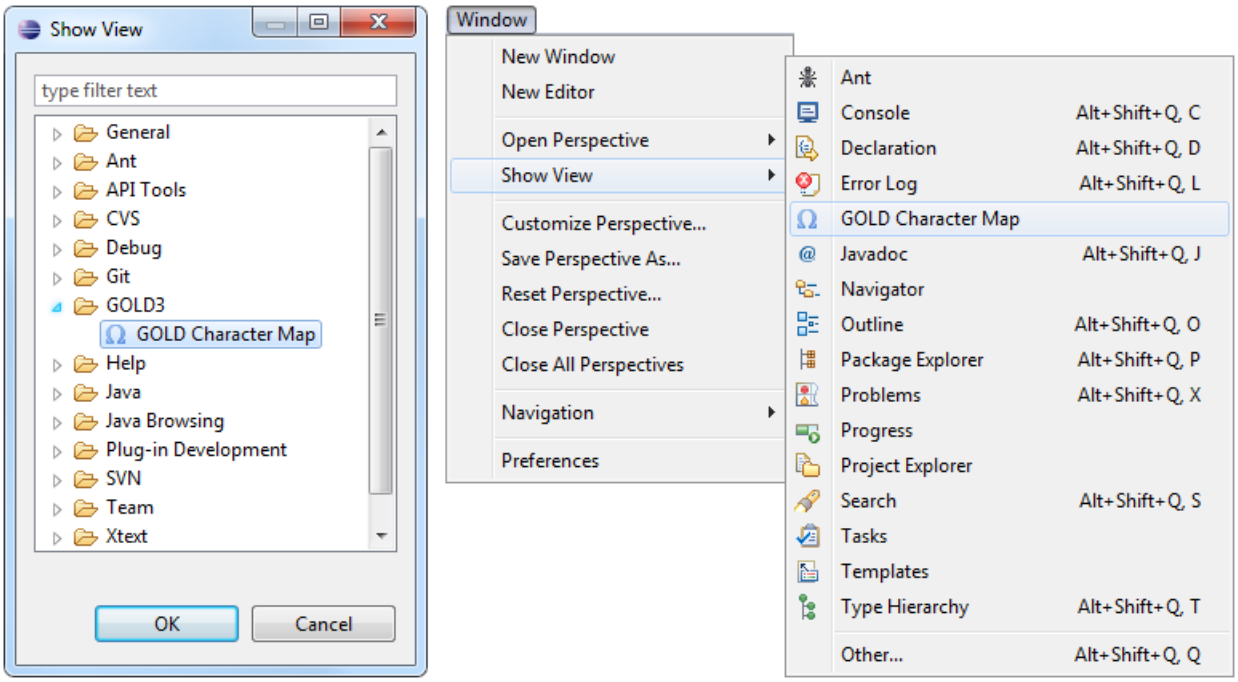


Figura 8.9. *Mapa de caracteres de GOLD 3, distribuido como una vista que se puede instalar en Eclipse.*



Las clases `GoldDSLCharacterManager` y `GoldDSLCharacterMap` implementan el patrón *Singleton* para asegurar que

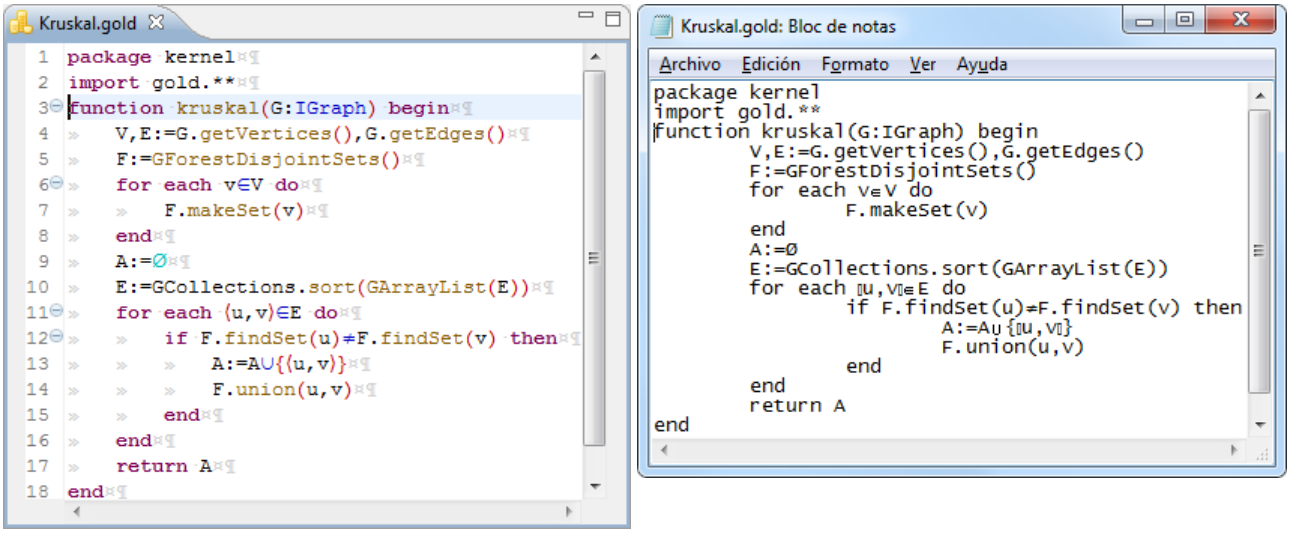
siempre exista una sola instancia de éstas en ejecución, garantizando así que el mapa de caracteres sea compartido por todas las instancias del editor de texto de *GOLD*, sin importar el proyecto que se esté modificando. Por otro lado, la clase auxiliar `org.gold.dsl.ui.util.GoldDSLShell` brinda los mecanismos necesarios para adaptar ventanas *Swing* (`javax.swing.JDialog`) como ventanas *SWT* (`org.eclipse.swt.widgets.Shell`), y componentes *Swing* (`javax.swing.JComponent`) como componentes *SWT* (`org.eclipse.swt.widgets.Composite`).

8.1.3. Editor de código fuente (*source code editor*)

El editor de código fuente es el componente que permite la edición de programas codificados en *GOLD* integrando todas las funcionalidades especiales implementadas en el ambiente de desarrollo, como el resaltado de la sintaxis y la validación de errores de compilación. Está configurado para desplegarse automáticamente cada vez que el usuario inicie la edición de un archivo con extensión `.gold` (sensible a las minúsculas/mayúsculas) y acepta la inserción de símbolos especiales a través del mapa de caracteres, ya sea haciendo doble clic en la tabla de símbolos o digitando en el teclado su atajo correspondiente. Aunque es posible modificar los archivos con extensión `.gold` mediante un editor de texto tradicional como *Bloc de notas*, *WordPad* o *gedit*, no se recomienda su uso puesto que los caracteres especiales no se despliegan correctamente y no se contaría con las ayudas suministradas por el editor como la numeración de los renglones, el resaltado de la línea actual y el revelamiento de los espacios en blanco.

Figura 8.10. *Algunos editores de código fuente para los programas escritos en GOLD 3.*

(a) *GOLD Editor.* **(b)** *Bloc de notas.*



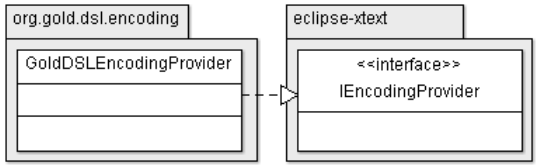
8.1.4. Proveedor de codificación de caracteres (*encoding provider*)

El proveedor de codificación de caracteres (*encoding provider*) configura el mecanismo usado para codificar caracteres *Unicode* como bytes y viceversa [6], que se utiliza para almacenar los programas *GOLD* (vistos como flujos de caracteres *Unicode*) en archivos del sistema operativo (vistos como flujos de bytes) y para cargarlos posteriormente con el proceso inverso. La codificación de caracteres usada por *GOLD* es el formato *UTF-8* (*8-bit UCS Transformation Format*), que es capaz de representar todos los símbolos pertenecientes al conjunto de caracteres *Unicode*, a diferencia del formato *ISO-8859-1* que únicamente incluye letras del alfabeto latino y algunos símbolos especiales. Si se intenta editar un archivo *GOLD* con un editor que no esté configurado con la codificación *UTF-8* o si se crea un nuevo archivo *GOLD* con otra codificación, éste no es procesado correctamente.

La codificación *UTF-8* se usa en *GOLD* en todos los procesos que requieren transformar cadenas de caracteres

en flujos de bytes. Incluso, la totalidad del código fuente de *GOLD* está codificado en ese formato. La clase `gold.dsl.encoding.GoldDSLEncodingProvider` es la responsable de configurar la codificación *UTF-8* como la predefinida para todos los recursos alojados en archivos con extensión `.gold`.

Figura 8.11. *Diagrama de clases del paquete* `gold.dsl.encoding`.



8.1.5. Resaltado de la sintaxis (*syntax highlighting*)

El resaltado de la sintaxis (*syntax highlighting*) es una característica que asigna atributos visuales distintos a cada uno de los tokens que componen un programa *GOLD* después de realizar el análisis léxico y sintáctico, alterando con ahínco determinadas propiedades del texto como su estilo (normal, *cursiva* o **negrilla**), su tipo de letra (una de las cuatro tipografías enumeradas por la clase `org.gold.dsl.ui.util.GoldDSLFont`: *Lucida Sans Unicode*, *Dialog*, *Gold Regular* y *Courier New*), su color, y su tamaño.

Figura 8.12. *Resaltado de la sintaxis del algoritmo Insertion-Sort, escrito en GOLD 3.*

```
procedure insertionSort(A) begin
    for j:=1 to |A|-1 do
        key,i:=A[j],j-1
        // Insert A[j] into the sorted sequence A[0..j-1]
        while i≥0 and A[i]>key do
            A[i+1],i:=A[i],i-1
        end
        A[i+1]:=key
    end
end
```

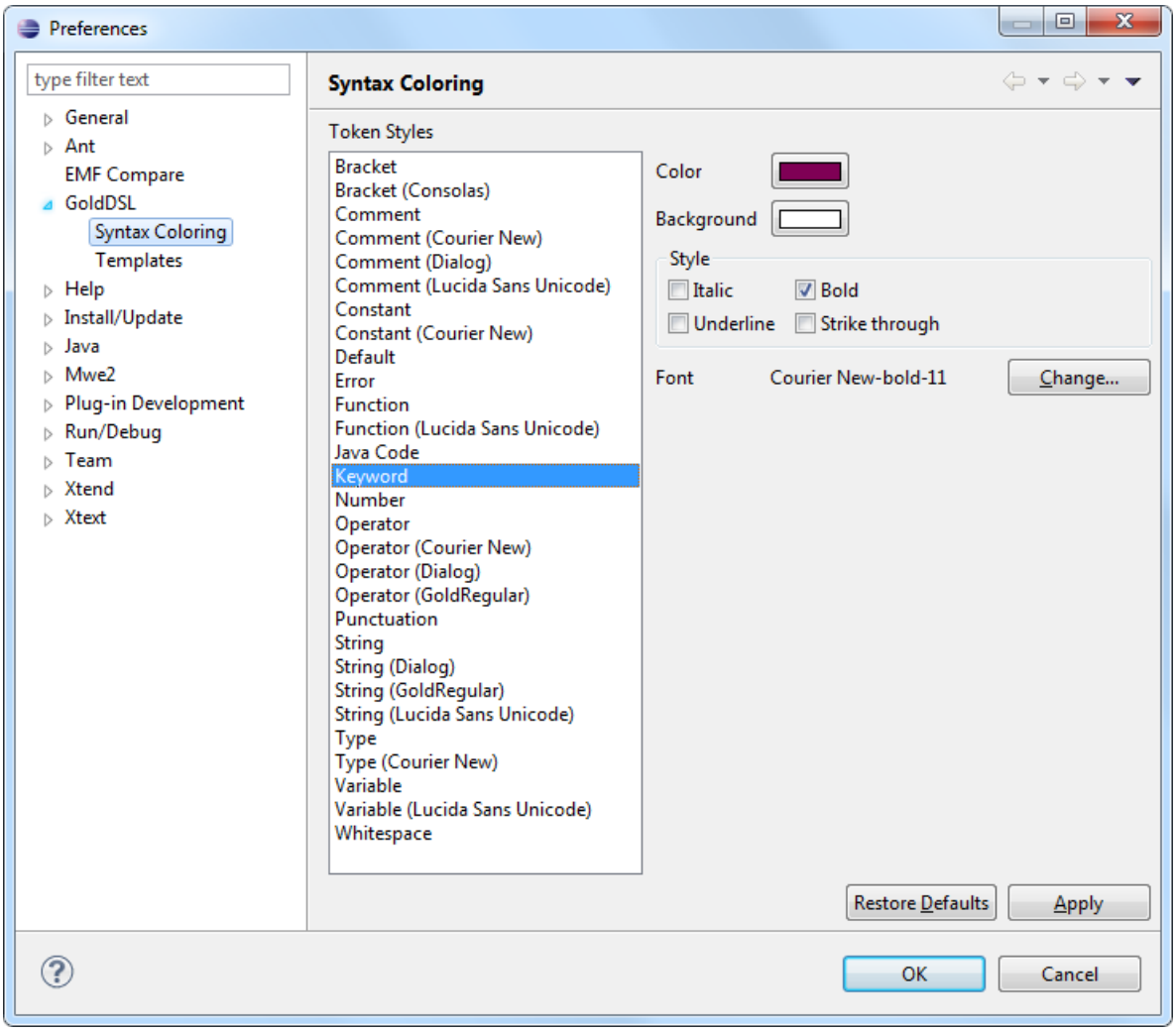
No sólo es importante considerar la coloración de la sintaxis (*syntax coloring*); también es necesario modificar el tipo de letra, el estilo y el tamaño del texto por diversas razones. Por ejemplo, los comentarios suelen verse mejor en cursiva, las palabras reservadas (*reserved words*) suelen verse mejor en negrilla y el código fuente es más legible en un tipo de letra monoespaciado. Como regla, los atributos visuales asignados a cada token dependen de su tipo, que en la mayoría de los casos es determinado luego del análisis sintáctico, salvo algunos casos particulares que son resueltos con la ayuda de un análisis semántico básico. No obstante, los atributos visuales terminan dependiendo también del sistema operativo para asegurar un despliegue uniforme que no se vea afectado por detalles técnicos relacionados con el comportamiento distinto de las tipografías *TrueType* en máquinas *Windows* y *Unix*.

Tabla 8.5. *Atributos visuales asignados por defecto a cada tipo de token de GOLD 3, en sistemas Windows.*

Tipo de token	Estilo	Tipo de letra	Color (r,g,b)	Tamaño
Espacios en blanco (*Whitespace*)	Normal	*Courier New*	Gris (80,80,80)	11 pt
Palabra clave (*Keyword*)	**Negrilla**	*Courier New*	Escarlata (127,0,85)	11 pt
Variable (*Variable*)	Normal	*Courier New*	Negro (0,0,0)	11 pt
Función (*Function*)	Normal	*Courier New*	Ocre (130,90,0)	11 pt
Número (*Number*)	Normal	*Courier New*	Azul claro (128,128,255)	11 pt
Constante (*Constant*)	Normal	*Gold Regular*	Cyan oscuro (0,192,192)	11 pt
Tipo (*Type*)	Normal	*Gold Regular*	Azul oscuro (0,0,192)	11 pt
Operador (*Operator*)	Normal	*Lucida Sans Unicode*	Azul oscuro (0,0,192)	11 pt

Cadena de texto (*String*)	Normal	*Courier New*	Magenta oscuro (192,0,192)	11 pt
Signo de puntuación (*Punctuation*)	Normal	*Courier New*	Negro (0,0,0)	11 pt
Paréntesis (*Bracket*)	Normal	*Gold Regular*	Rojo oscuro (192,0,0)	11 pt
Comentario (*Comment*)	Normal	*Gold Regular*	Oliva (63,127,95)	11 pt
Error (*Error*)	Normal	*Gold Regular*	Rosado (255,120,120)	11 pt
Código *Java* (*Java Code*)	Normal	*Courier New*	Gris (80,80,80)	11 pt
Predeterminado (*Default*)	Normal	*Gold Regular*	Negro (0,0,0)	11 pt

Figura 8.13. *Página de preferencias para configurar el resaltado de la sintaxis en GOLD 3.*



El resaltado de la sintaxis mejora considerablemente la legibilidad de un programa computacional [6] y ayuda a la detección y corrección de errores de compilación, puesto que se vuelve más sencillo distinguir los diferentes elementos sintácticos que componen el código fuente sin alterar su semántica asociada. El proceso llevado a cabo para asignar atributos visuales a cada tipo de token se implementó con *Xtext* [6] y está dividido en dos etapas:

1. *Resaltado léxico-sintáctico*. Realiza un análisis léxico-sintáctico liviano y rudimentario que se ejecuta instantáneamente [6] después de cada golpe de teclado (*keystroke*) para resaltar con rapidez los tokens cuyo tipo pueda ser inferido a partir de su cadena de texto, independientemente de los tokens que estén alrededor. Por ejemplo, los comentarios y las palabras reservadas del lenguaje son resaltados por este mecanismo.

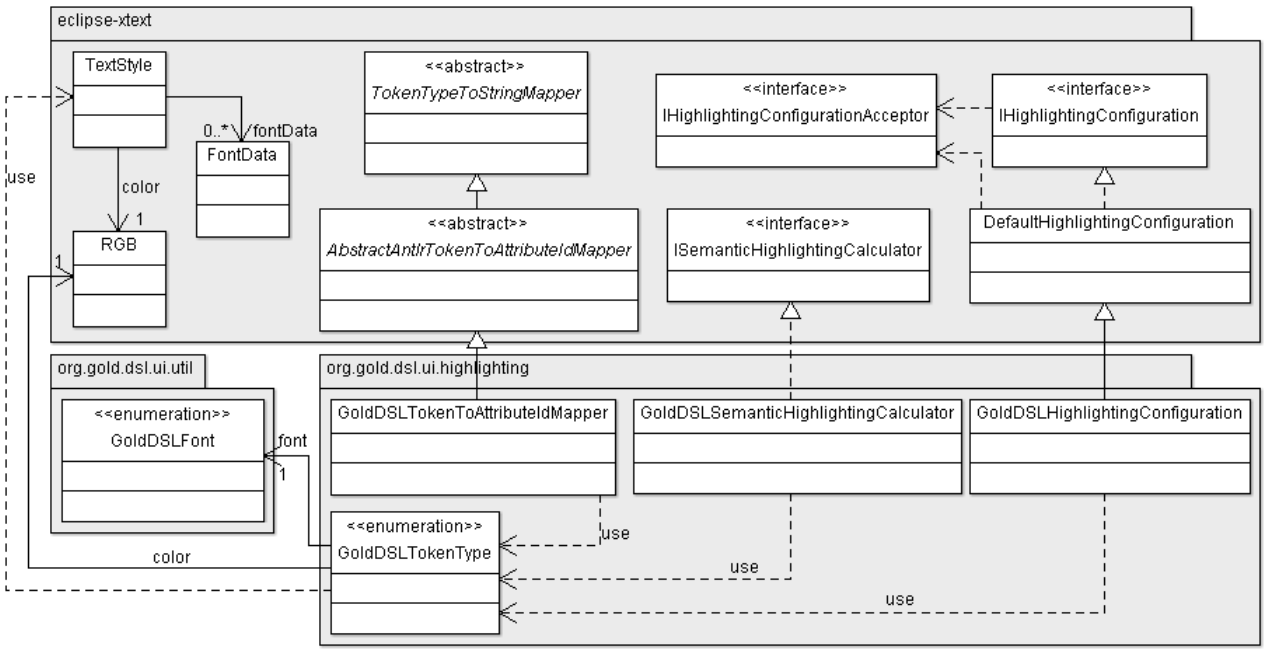
2. *Resaltado semántico*. Realiza un análisis semántico básico que se ejecuta asincrónicamente [6] como un proceso de fondo (*background process*) para identificar algunos tokens cuyo tipo no fue detectado por el análisis léxico-sintáctico, y luego les aplica los cambios avanzados de formato que haya a lugar, con base en el significado de los diferentes elementos que los circundan. Por ejemplo, los nombres de función son resaltados por este mecanismo ya que son identificadores que están inmediatamente seguidos por un paréntesis izquierdo.

Todos los estilos de resaltado descritos en la tabla 8.5 pueden ser personalizados [6] en la página de preferencias del lenguaje bajo el menú *Window* de *Eclipse*. Es importante anotar que algunos estilos están definidos sobre varios tipos de letra para permitir la escritura de textos mezclando caracteres pertenecientes a diferentes tipografías, principalmente dentro de las cadenas de texto y los comentarios.

El paquete `gold.dsl.ui.highlighting` contiene las clases que implementan el resaltador de sintaxis usando los artefactos provistos por *Xtext* [6]:

- `GoldDSLTokenType`. Representa la enumeración de los tipos de token consignados en la tabla 8.5.
- `GoldDSLTokenToAttributeIdMapper`. Representa el resaltador de sintaxis que desarrolla el proceso léxico-sintáctico que da el formato definitivo a la mayoría de los tokens.
- `GoldDSLSemanticHighlightingCalculator`. Representa el resaltador de sintaxis que desarrolla el proceso semántico para refinar la asignación de atributos visuales a los tokens, luego de aplicar el procedimiento léxico-sintáctico.
- `GoldDSLHighlightingConfiguration`. Representa el componente responsable de configurar e instalar todos los estilos de resaltado predefinidos, poblando la página de preferencias.

Figura 8.14. *Diagrama de clases del paquete* `gold.dsl.ui.highlighting`.



8.1.6. Formateado de código (*code formatting*)

El formateado de código (*code formatting*) es un proceso automático que modifica, elimina e inserta caracteres ocultos (espacios (‘ ’), tabulaciones (‘`\t`’), retornos de carro (‘`\r`’) y cambios de línea (‘`\n`’)) sobre el código fuente para someterlo a un determinado estándar de codificación [6]. Puede ser ejecutado oprimiendo simultáneamente las teclas `Control+Shift+F` y en general sirve para ordenar código fuente mal distribuido o mal indentado.

Figura 8.15. *Código del algoritmo Bubble-Sort, antes y después de ser formateado automáticamente en GOLD 3.*

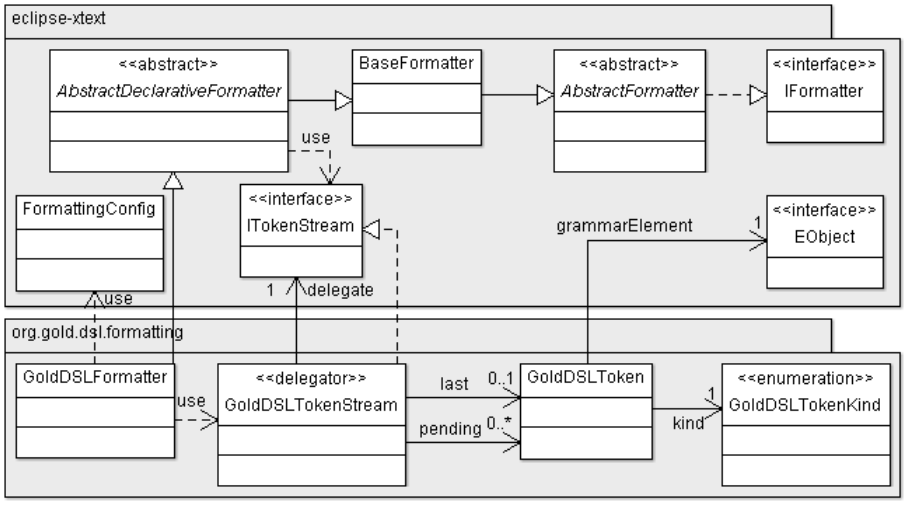
(a) *Antes de formatear.*

```
procedure¤¶
bubbleSort(¤¶
A) begin   for¤¶
  i:=0 to |   A¤¶
  |-1¤¶
        do¤¶
for j¤¶
:= |  A     | -1 downto i¤¶
+1 do if A[¤¶
j]<¤¶
A [j-1] then swap A[j]¤¶
←A[j-1] end end end end¤¶
```

(b) *Después de formatear.*

```
procedure bubbleSort(A) begin¤¶
»   for i:=0 to |A|-1 do¤¶
»   »   for j:=|A|-1 downto i+1 do¤¶
»   »   »   if A[j]<A[j-1] then¤¶
»   »   »   »   swap A[j]←A[j-1]¤¶
»   »   »   end¤¶
»   »   end¤¶
»   end¤¶
end¤¶
```

Figura 8.16. *Diagrama de clases del paquete* `org.gold.dsl.formatting`.



El paquete `org.gold.dsl.formatting` contiene las clases que implementan el formateador de código fuente basado en la arquitectura provista por *Xtext* [6]:

- `GoldDSLTokenKind`. Representa la enumeración que describe las especies de token que existen según el formateador de *Xtext* [6]: semánticos (*semantic*) y ocultos (*hidden*).

- `GoldDSLToken`. Representa un token que resulta después de realizar el análisis léxico particular al formateador, cuyos atributos son la especie del token (semántico u oculto), la producción de la gramática que generó el token (*grammar element*, de acuerdo con la terminología de *Xtext* [6]) y el texto correspondiente a la imagen del token.

- `GoldDSLTokenStream`. Representa un flujo de tokens que administra adecuadamente los espacios distintos a las tabulaciones y cambios de línea, eliminando espacios innecesarios e insertando espacios obligatorios, sin alterar la semántica del programa. Implementa el patrón *Delegation* que lidia con los espacios y encarga el resto de responsabilidades a un delegado que termina de realizar las operaciones de formateado.

- `GoldDSLFormatter`. Representa el formateador de código fuente que configura exhaustivamente las políticas correspondientes al estándar de codificación definido para los programas *GOLD*, estableciendo las reglas de indentación y los lugares donde deben ser insertadas las tabulaciones y los cambios de línea para lograr un resultado estéticamente agradable.

8.1.7. Autoedición de código (*autoedit strategies*)

La autoedición de código (*autoedit strategies*) es una característica que altera el código fuente para ayudar al programador a reducir el tiempo de desarrollo, asistiéndolo en las siguientes labores:

- *Indentación automática*. Inserta espacios en blanco antes de cada línea para someter el código fuente a una política de indentación previamente configurada. De esta manera, el desarrollador no tendría que preocuparse por añadir tabulaciones cada vez que oprima el retorno de carro.

- *Reemplazo de atajos*. Reemplaza cada atajo de teclado descrito en la sección §A.5.1 por su carácter correspondiente. De esta manera, el desarrollador no tendría que realizar movimientos de ratón para usar símbolos matemáticos especiales del mapa de caracteres.

- *Balanceo de paréntesis*. Añade automáticamente el símbolo que cierra un determinado tipo de paréntesis izquierdo, de acuerdo con la tabla A.12. De esta manera, el desarrollador no tendría que cerrar ningún paréntesis que abra.

- *Autocompletado de instrucciones*. Inserta automáticamente una plantilla (*template*) que ilustra la sintaxis de una determinada instrucción del lenguaje, de acuerdo con la tabla A.14. De esta manera, el desarrollador no tendría que memorizar la sintaxis particular de cada tipo de instrucción.

Todos los comportamientos descritos son ejecutados de forma automática a medida que el usuario digita el código fuente en el editor, ahorrándole tiempo valioso. Además de los atajos de teclado definidos para cada símbolo especial del lenguaje *GOLD*, se configuró un atajo adicional que reemplaza todo par de puntos seguidos '..' por el carácter *Unicode* `0x2025`, que es utilizado para describir intervalos cerrados de números enteros o de caracteres.

Figura 8.17. *Inserción paso a paso de símbolos matemáticos usando atajos de teclado de GOLD 3.*

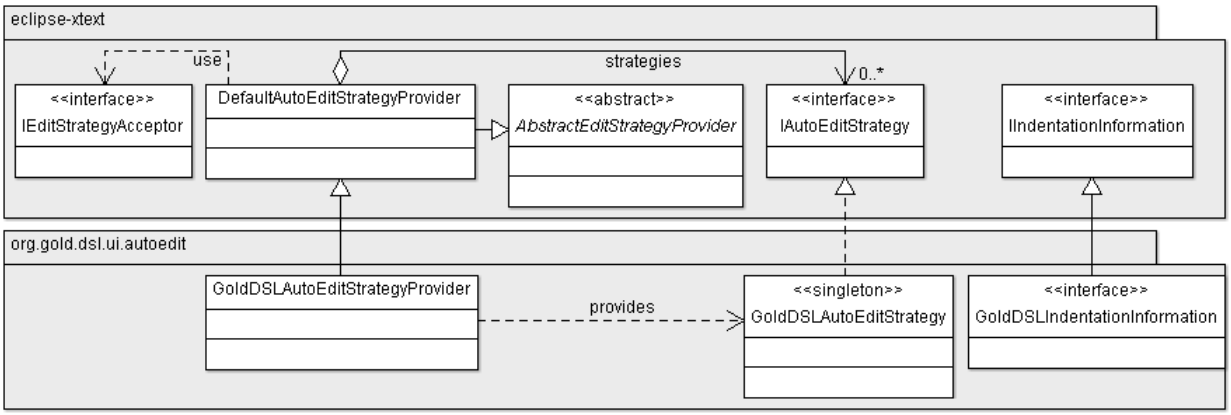
El paquete `org.gold.dsl.ui.autoedit` contiene las clases que implementan la autoedición de código fuente a través de las rutinas provistas por *Xtext* [6]:

- `GoldDSLIndentationInformation`. Representa la clase responsable de indicar el texto usado para elevar el nivel de indentación en el código fuente. Por defecto se está usando una tabulación ('`\t`').

- `GoldDSLAutoEditStrategy`. Representa la estrategia de edición de código que implementa el proceso de indentación automática, reemplazo de atajos, balanceo de paréntesis y autocompletado de instrucciones.

- `GoldDSLAutoEditStrategyProvider`. Representa el proveedor de estrategias de edición de código que combina los procesos antes mencionados con otros implementados por *Xtext*: balanceo de comillas simples ('···'), balanceo de comillas dobles ("···") y balanceo de símbolos de comentario multilínea (`/*···*/`).

Figura 8.18. *Diagrama de clases del paquete* `org.gold.dsl.ui.autoedit`.



8.1.8. Plegamiento de código (*code folding*)

El plegamiento de código (*code folding*) permite ocultar *bloques* (véase la sección §7.2.8) del código fuente para facilitar su lectura o edición. Esta funcionalidad es provista completamente por *Xtext* [6] y su comportamiento no fue configurado de manera especial en *GOLD*.

Figura 8.19. *Algoritmo de Kruskal, antes y después de plegar sus instrucciones repetitivas en GOLD 3.*

(a) *Antes de plegar.*

```
1  package kernel
2  import gold.**
3  function kruskal(G:IGraph) begin
4      V,E:=G.getVertices(),G.getEdges()
5      F:=GForestDisjointSets()
6      for each v∈V do
7          F.makeSet(v)
8      end
9      A:=∅
10     E:=GCollections.sort(GArrayList(E))
11     for each ⟨u,v⟩∈E do
12         if F.findSet(u)≠F.findSet(v) then
13             A:=A∪{⟨u,v⟩}
14             F.union(u,v)
15         end
16     end
17     return A
18 end
```

(b) *Después de plegar.*

```
1  package kernel
2  import gold.**
3  function kruskal(G:IGraph) begin
4      V,E:=G.getVertices(),G.getEdges()
5      F:=GForestDisjointSets()
6      for each v∈V do
9      A:=∅
10     E:=GCollections.sort(GArrayList(E))
11     for each ⟨u,v⟩∈E do
17     return A
18 end
```

8.1.9. Emparejamiento de paréntesis (*bracket matching*)

El emparejamiento de paréntesis (*bracket matching*) es una funcionalidad que resalta la pareja de un determinado paréntesis de apertura o de cierre de acuerdo con la tabla A.12.

El paquete `org.gold.dsl.ui.bracketmatching` contiene la clase `GoldDSLBracketMatcher`, que implementa el emparejador de paréntesis dependiendo de la posición del cursor y del tipo de paréntesis. Cada pareja de paréntesis se configura de tal manera que cuando el cursor esté delante de cualquier paréntesis, su contraparte se resalta. Este servicio facilita la escritura de expresiones anidadas, donde la identificación manual de paréntesis puede llegar a ser incómoda para el desarrollador.

Figura 8.20. *Diagrama de clases del paquete* `org.gold.dsl.ui.bracketmatching`.

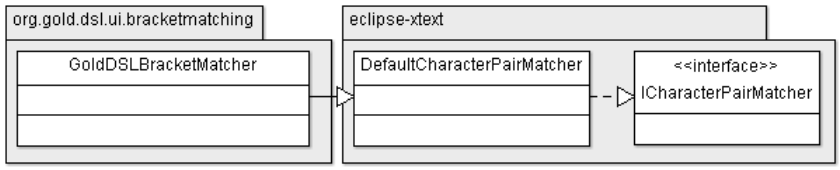
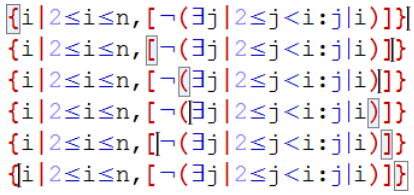


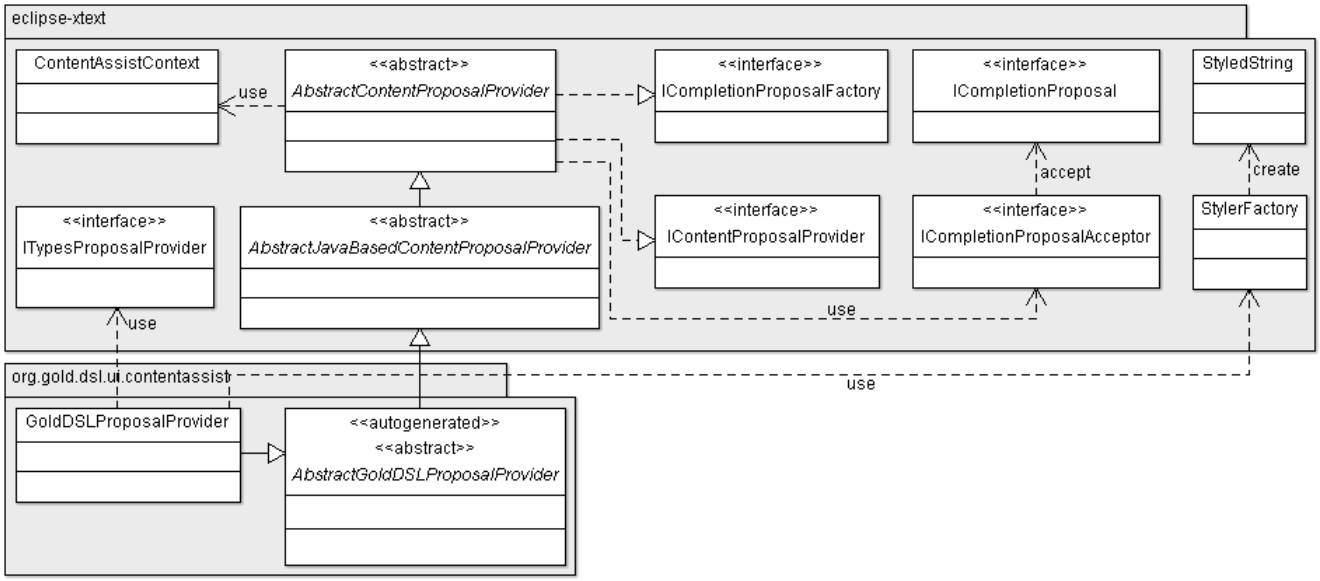
Figura 8.21. *Emparejamiento de paréntesis en GOLD 3, dependiendo de la posición del cursor.*



8.1.10. Ayudas de contenido (*content assist*)

Las ayudas de contenido (*content assist*) es una funcionalidad que acelera la escritura de código fuente autocompletando automáticamente algunos elementos del lenguaje como palabras reservadas, nombres de variable, nombres de función, tipos de datos, operadores, atributos y métodos, entre otros. Se activa oprimiendo simultáneamente las teclas `Control+Space` y, dependiendo del lugar donde se encuentre el cursor dentro del código fuente, le presenta al usuario una lista con las posibles opciones que tiene para completar lo que esté escribiendo y continuar con la edición del programa.

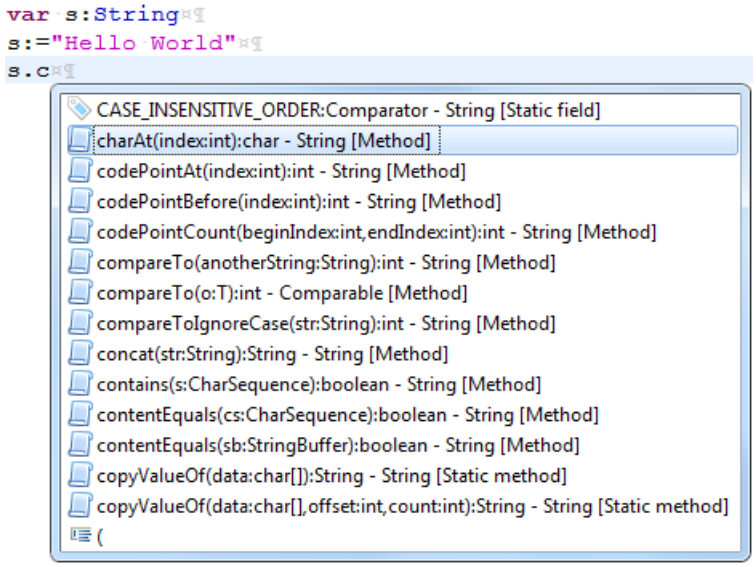
Figura 8.22. *Diagrama de clases del paquete* `org.gold.dsl.ui.contentassist`.



El paquete `org.gold.dsl.ui.contentassist` contiene la clase `GoldDSLProposalProvider`, que implementa las ayudas de contenido extendiendo la clase abstracta `AbstractGoldDSLProposalProvider` que es generada automáticamente por *Xtext*. Específicamente, la clase abstracta `AbstractGoldDSLProposalProvider` colabora con el autocompletado de palabras reservadas y operadores del lenguaje, mientras que la clase concreta `GoldDSLProposalProvider` colabora con el autocompletado del resto de características como nombres de variable, nombres de función, tipos de datos, atributos y métodos. Ambas clases actúan como un proveedor de propuestas (*proposal provider*) que alimenta la

lista de opciones que será presentada al usuario dependiendo del lugar donde se encuentre el cursor, del texto que se esté escribiendo y del contexto semántico asociado al fragmento de código que se esté editando, especificando exhaustivamente el comportamiento para el autocompletado de cada símbolo terminal o no terminal referenciado en cada producción de la gramática del lenguaje.

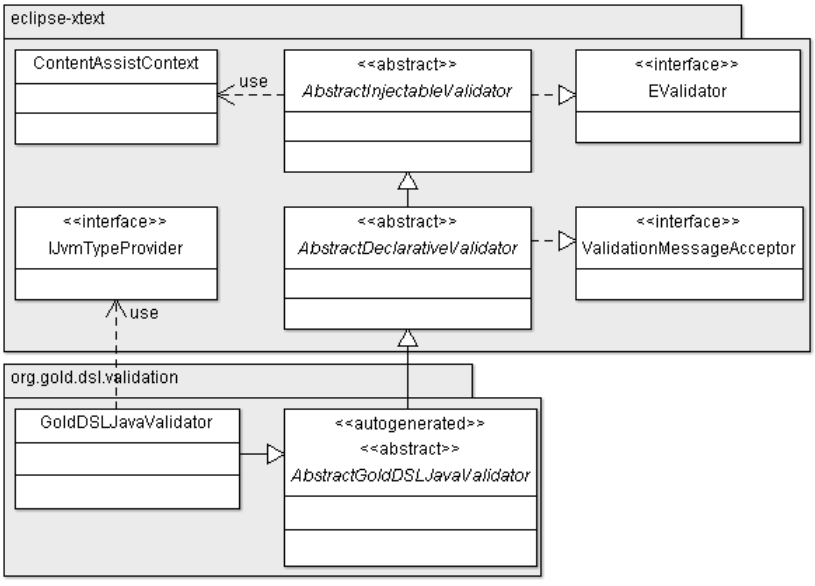
Figura 8.23. *Ayudas de contenido en GOLD 3, dependiendo de la posición del cursor.*



8.1.11. Validación de código (*code validation*)

La validación de código (*code validation*) es responsable de revisar que se estén cumpliendo las restricciones semánticas del lenguaje que el análisis léxico-sintáctico no fue capaz de detectar. Mientras el usuario escribe el código fuente en tiempo de desarrollo, la validación de código resalta automáticamente los errores (*errors*) de compilación dando descripciones detalladas de cada uno de éstos. Además de los errores, la validación emite advertencias (*warnings*) que pueden conllevar a potenciales errores en tiempo de ejecución.

Figura 8.24. *Diagrama de clases del paquete* `org.gold.dsl.validation`.



El paquete `org.gold.dsl.validation` contiene la clase `GoldDSLJavaValidator`, que implementa la validación semántica de código extendiendo la clase abstracta `AbstractGoldDSLJavaValidator` que es generada automáticamente por *Xtext*. A grandes rasgos, la clase concreta `GoldDSLJavaValidator` utiliza el metamodelo de clases creado por *Xtext*, que representa los elementos sintácticos del lenguaje mediante una estructura arbórea denominada *Abstract Syntax Tree* (*AST*), para detectar todos los errores de compilación relacionados con la semántica del lenguaje y para generar determinadas advertencias. Sin el validador de código, el usuario podría escribir programas bien formados sintácticamente que no tengan sentido semántico, en especial si está accediendo atributos y métodos de algún objeto del *API* estándar de *Java*.

Figura 8.25. *Resaltado de errores de compilación y de advertencias en GOLD 3.*

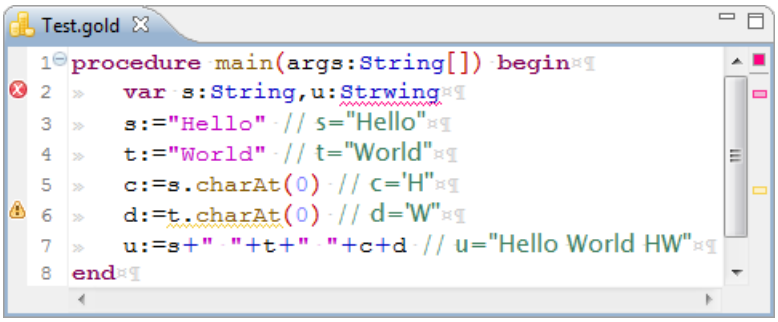
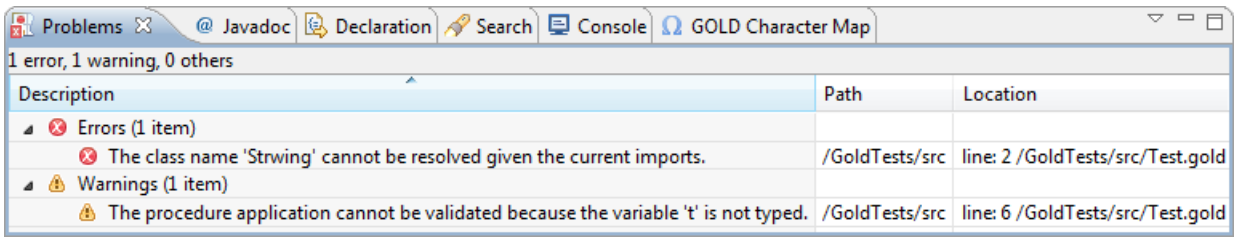


Figura 8.26. *Vista Eclipse que despliega los errores de compilación y las advertencias generadas por GOLD 3.*



8.1.12. Convertidores de valores (*value converters*)

Los convertidores de valores (*value converters*) transforman en su forma canónica los tokens que componen un programa *GOLD*, luego del análisis léxico-sintáctico, mediante los siguientes procesos:

- *Limpieza de caracteres no imprimibles*. Elimina los espacios (‘ ’), tabulaciones (‘`\t`’), retornos de carro (‘`\r`’) y cambios de línea (‘`\n`’) presentes en los identificadores y nombres calificados [†6].
- *Eliminación del carácter de escape*. Elimina el carácter de escape (‘`$`’) del principio de todos los identificadores, recordando que éste es usado para desambiguar palabras reservadas [†7].
- *Tratamiento de subíndices*. Reemplaza subíndices consecutivos por un guión bajo (‘_’) seguido de los dígitos numéricos correspondientes (véase la tabla A.11). Además, cada símbolo prima (′) que encuentre es reemplazado por un guión bajo (‘_’).

El proceso de conversión de valores es necesario para:

[6] En *GOLD* los nombres calificados (*qualified names*) son secuencias de identificadores (*identifiers*) separados por puntos.

[7] En *GOLD*, el signo pesos (‘`$`’) permite nombrar identificadores que coinciden con alguna palabra reservada del lenguaje, haciendo posible la declaración e invocación de métodos y variables cuyo nombre sea una palabra reservada. Por ejemplo, como el identificador `print` es una palabra reservada en *GOLD*, para declarar una variable o método con nombre `print` debe anteponerse el signo pesos, obteniendo `$print`. Más aún, instrucciones como `System.out.print("texto")` deben escribirse en la forma `System.out.$print("texto")`.

1. permitir la declaración de variables, procedimientos y funciones cuyo nombre sea alguna palabra reservada;
2. permitir el uso de clases cuyo nombre sea alguna palabra reservada;
3. permitir la invocación de métodos y atributos cuyo nombre sea alguna palabra reservada; y
4. facilitar la traducción de programas *GOLD* a código *Java*, dado que los identificadores en *Java* no pueden estar formados por subíndices.

Tabla 8.6. *Ejemplos de conversión de nombres calificados en GOLD 3.*

Nombre calificado original	Nombre calificado convertido
`System.out.println`	`System.out.println`
`System. out. println`	`System.out.println`
`$System.$out.$println`	`System.out.println`
`System.out.$print`	`System.out.print`
`System. out. $print`	`System.out.print`
$\texttt{Clase}_1\texttt{.funcion}_2$	`Clase_1.funcion_2`
x_{12}	`x_12`
$p_{12}q_{345}r_{s06789}$	`p_12q_345rs_06789`
x'	`x_`
x''	`x__`
x_1''	`x_1__`

Es importante recordar que los identificadores en *GOLD* pueden comenzar con el signo pesos (‘`$`’), usado para escapar (*to escape*) un identificador cuando hay conflicto con alguna palabra reservada definida con el mismo nombre. En *Xtext* se usa el acento circunflejo o *caret* (‘`^`’) para este mismo fin, en lugar del signo pesos (‘`$`’).

Además hay que tener en cuenta que, por defecto, *Xtext* somete los literales que representan caracteres y cadenas de texto a un tratamiento que elimina las comillas del principio y del final, y transforma automáticamente las secuencias de escape de la tabla A.13 por sus caracteres correspondientes. Para facilitar el proceso de traducción de programas *GOLD* a código *Java* se debió anular este comportamiento, evitando a toda costa la alteración de dichos literales.

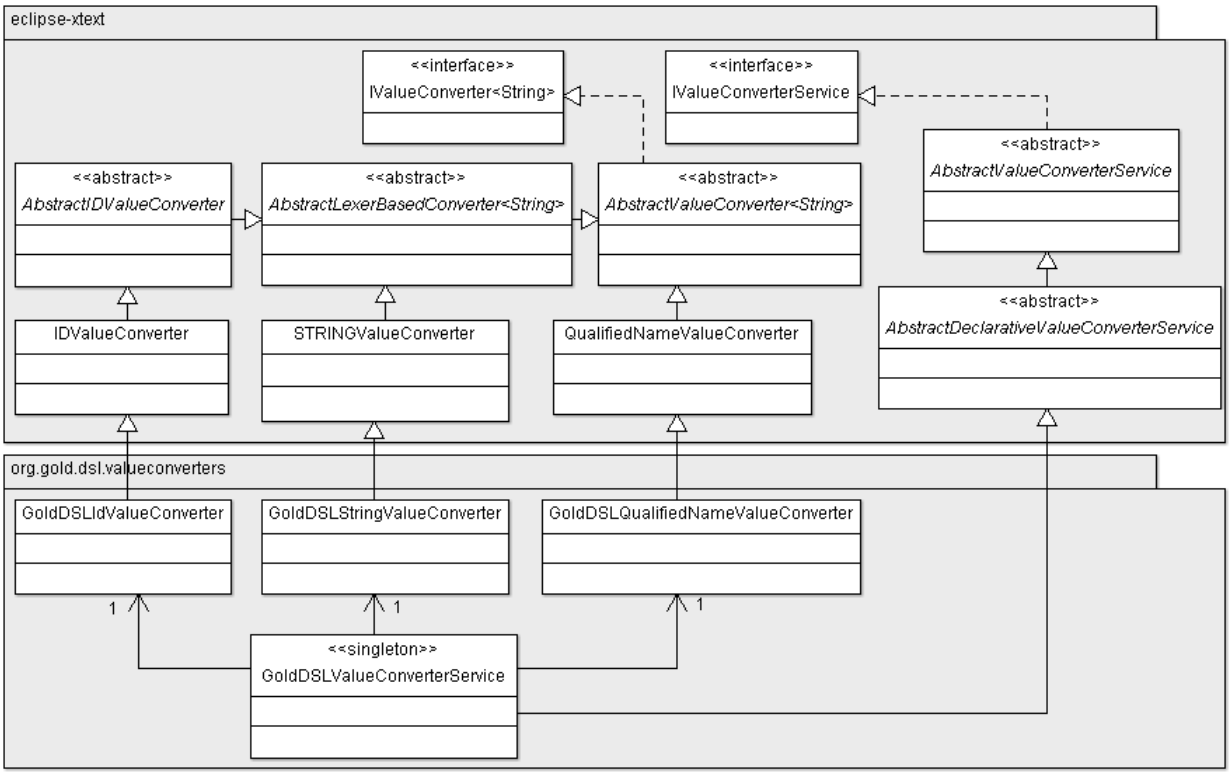
Figura 8.27. *Interoperabilidad entre las distintas formas de mencionar un nombre calificado en GOLD 3.*

```
procedure $main(args:String[]) begin
    var x1:int,x2:int
    x1,x2:=17,-15
    System.out.$print(x_1+";"+x1+";"+x1+";"+x_1+"\n")
    System.out.$print(x_2+";"+x2+";"+x2+";"+x_2+"\n")
end
```

El paquete `org.gold.dsl.valueconverters` contiene las clases que implementan los convertidores de valores a través de la estructura provista por *Xtext* [6]:

- `GoldDSLIdValueConverter`. Convierte identificadores de su forma original a su forma canónica limpiando espacios, removiendo el signo pesos y tratando los subíndices.
- `GoldDSLQualifiedNameValueConverter`. Convierte nombres calificados de su forma original a su forma canónica usando el convertidor de identificadores `GoldDSLIdValueConverter`.
- `GoldDSLStringValueConverter`. Deja intactos los literales que representan caracteres y cadenas de texto.
- `GoldDSLValueConverterService`. Crea y registra los convertidores de valores anteriormente mencionados.

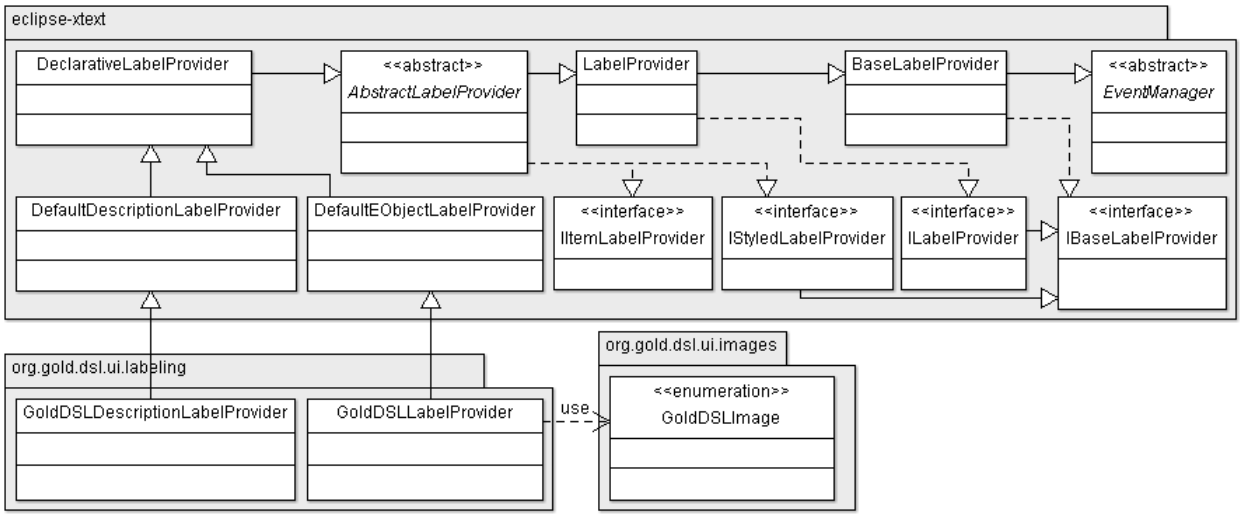
Figura 8.28. *Diagrama de clases del paquete* `org.gold.dsl.valueconverters`.



8.1.13. Proveedor de etiquetas (*label provider*)

El proveedor de etiquetas (*label provider*) es el componente responsable de especificar el texto que le será presentado al usuario cada vez que se necesite desplegar un determinado elemento semántico del lenguaje, ya sea en el esquema semántico (*outline view*), en los hipervínculos (*hyperlinks*) o en las propuestas (*content proposals*) dadas como ayudas de contenido (*content assist*), entre otros [6]. Además del texto, el componente también configura el ícono gráfico asociado a cada tipo de elemento usando primordialmente la galería de imágenes *Silk Icons* de *Famfamfam* [82].

Figura 8.29. *Diagrama de clases del paquete* `org.gold.dsl.ui.labeling`.



La clase `org.gold.dsl.ui.images.GoldDSLImage` representa una enumeración de todos los íconos gráficos que se utilizan como imagen de algún elemento en la interfaz gráfica de usuario del *IDE* de *GOLD*. Por otro lado, el paquete `org.gold.dsl.ui.labeling` contiene las clases que implementan el proveedor de etiquetas basado en *Xtext* [6]:

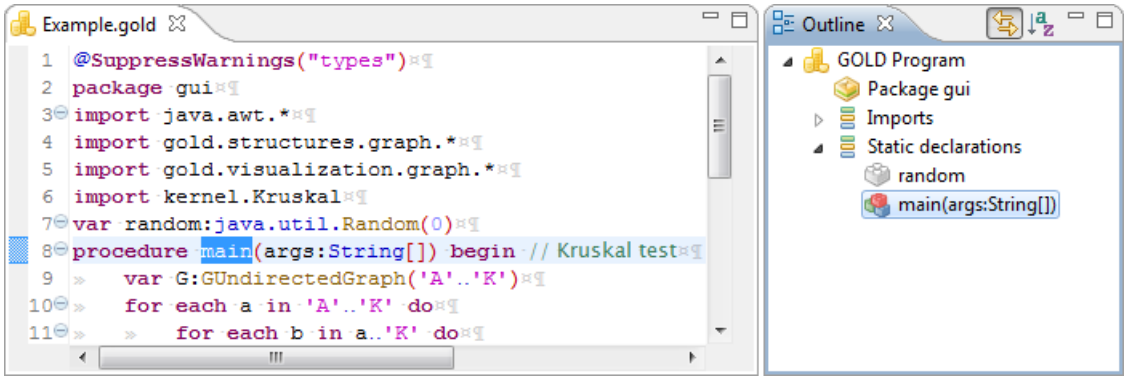
- `GoldDSLDescriptionLabelProvider`. Configura las etiquetas de texto y las imágenes correspondientes a cada registro del índice de referencias (*find references view*) [6].

- `GoldDSLLabelProvider`. Configura las etiquetas de texto y las imágenes correspondientes a cada elemento del modelo semántico.

8.1.14. Esquema semántico (*outline view*)

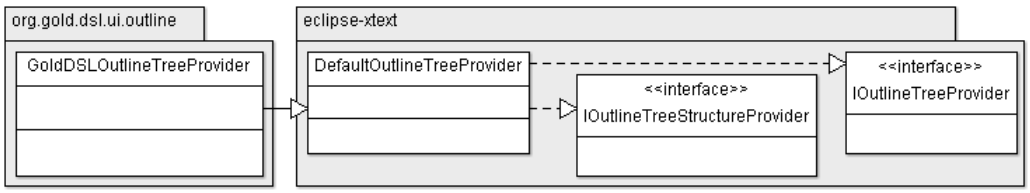
El esquema semántico (*outline view*) es una vista de *Eclipse* que despliega la estructura semántica de un programa para ayudar al desarrollador a navegar los distintos elementos que lo componen [6]. Suministra una visualización jerárquica del modelo semántico que permite ordenar alfabéticamente los elementos desplegados y resalta en el editor de código fuente el elemento actualmente seleccionado [6].

Figura 8.30. *Esquema semántico de GOLD 3, desplegando la estructura del código fuente de un programa.*



La clase `GoldDSLOutlineTreeProvider` del paquete `org.gold.dsl.ui.outline` configura los elementos que deben ser desplegados u ocultados dentro del esquema semántico para alterar su estructura, de acuerdo con la documentación de *Xtext* [6].

Figura 8.31. *Diagrama de clases del paquete* `org.gold.dsl.ui.outline`.



8.1.15. Contribuciones de menú (*menu contributions*)

Las contribuciones de menú (*menu contributions*) son puntos de extensión que enriquecen la barra de menú (*menu bar*) y la barra de herramientas (*tool bar*) del entorno de desarrollo de *Eclipse* a través de nuevas funcionalidades relacionadas con el lenguaje *GOLD*.

Figura 8.32. *Barra de menú y barra de herramientas de Eclipse con las contribuciones de GOLD 3.*

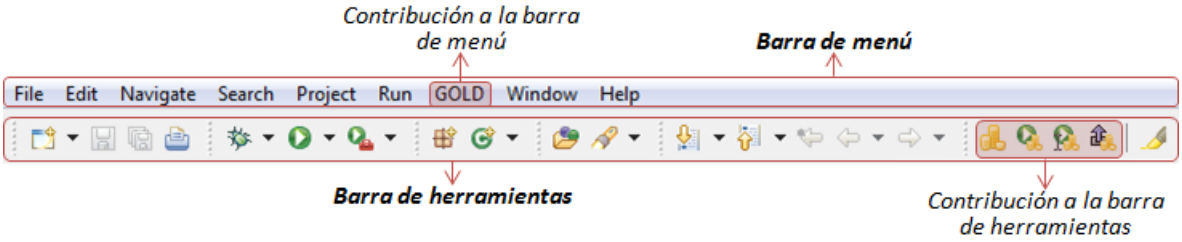


Figura 8.33. *Contribuciones de menú en GOLD 3.*

(a) *En la barra de menú.* **(b)** *En la barra de herramientas.*

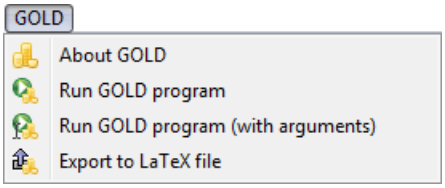


Figura 8.34. *Acerca de GOLD 3, cuyo logotipo principal fue tomado de Wikimedia Commons [83].*

Cada vez que el usuario active el editor de código fuente al iniciar la edición de algún programa *GOLD*, se añaden cuatro acciones a la barra de menú y a la barra de herramientas de *Eclipse*:

- *Acerca de GOLD* (*About GOLD*). Despliega la ventana *Acerca de GOLD* que muestra diversos datos del aplicativo como su nombre, su versión, sus autores y su página *WEB*.
- *Ejecutar programa GOLD* (*Run GOLD program*). Ejecuta en consola el programa *GOLD* que está siendo editado por el usuario.
- *Ejecutar programa GOLD con argumentos* (*Run GOLD program (with arguments)*). Ejecuta en consola el programa *GOLD* que está siendo editado por el usuario, solicitándole previamente los argumentos que le serán pasados al procedimiento propio *main*.

- *Exportar a LaTeX* (*Export to LaTeX file*). Exporta el programa *GOLD* que está siendo editado por el usuario como un archivo *LaTeX* que puede ser compilado para producir un documento en formato *PDF* susceptible de ser impreso.

Aunque dos de las acciones facilitan la ejecución de programas *GOLD*, es posible configurar los pormenores del ambiente de ejecución a través de la ventana *Run* → *Run Configurations* ... de *Eclipse* ajustando los distintos parámetros que se usarán para correr el método *main* del archivo *Java* que contiene la traducción del programa *GOLD*. Esto es posible pues cada archivo con extensión `.gold` presente en el directorio `src` es compilado automáticamente para generar su correspondiente traducción en un archivo con extensión `.java` localizado en el directorio `src-gen`.

Antes de usar alguno de los botones etiquetados con el texto *Run GOLD program*, debe guardarse el código fuente del archivo correspondiente y esperar a que el *Workspace* de *Eclipse* termine de actualizarse. De lo contrario, es posible que termine ejecutándose una versión previa del programa.

Figura 8.35. *Barra de progreso desplegada mientras se reconstruye el Workspace de Eclipse.*

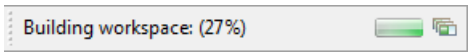
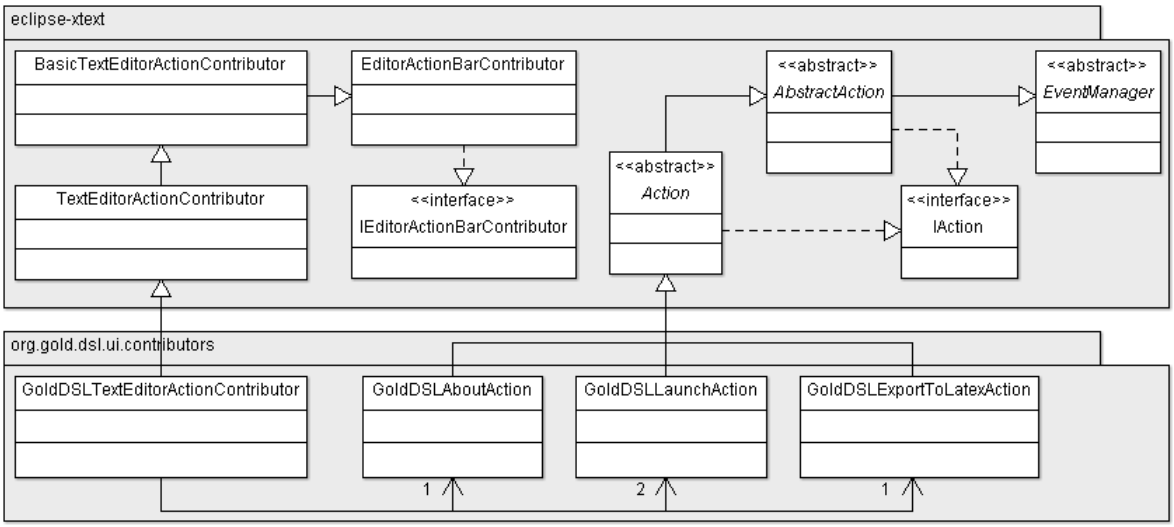


Figura 8.36. *Diagrama de clases del paquete* `org.gold.dsl.ui.contributors`.



El paquete `org.gold.dsl.ui.contributors` contiene las clases que implementan las contribuciones de menú:

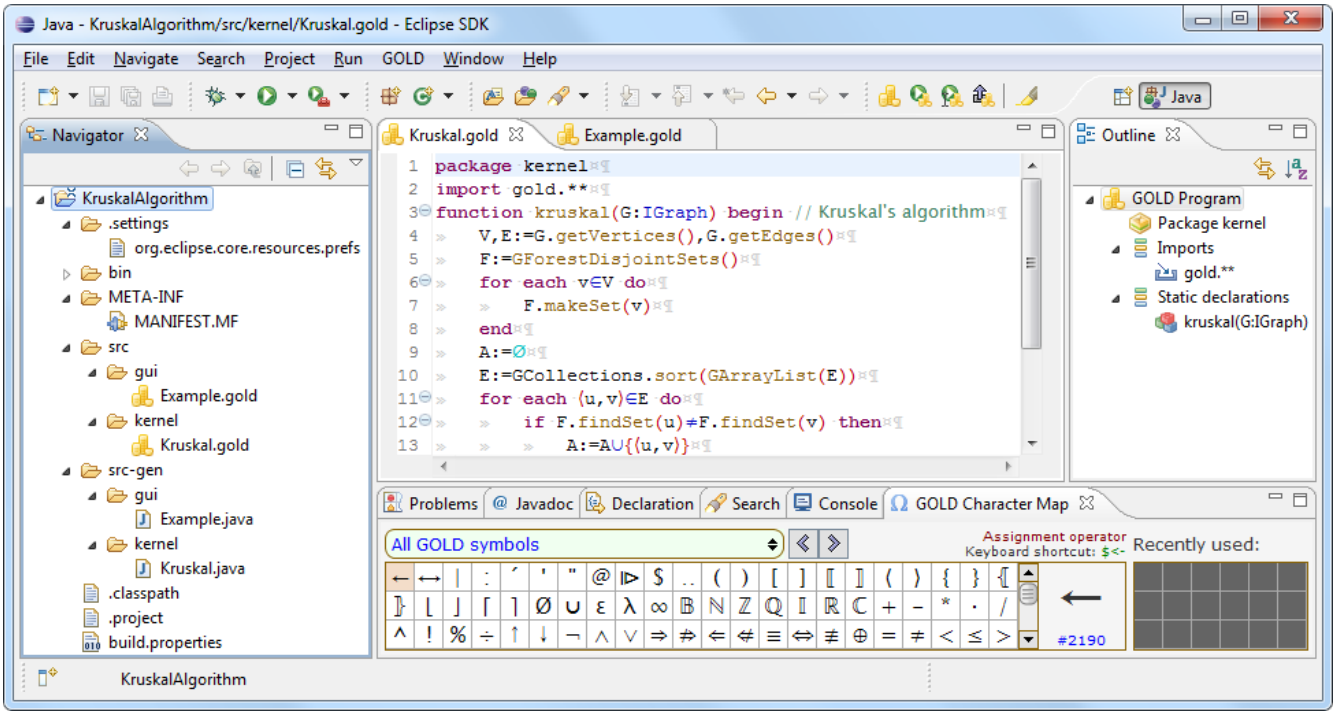
- `GoldDSLAboutAction`. Representa la acción que despliega el *Acerca de*.
- `GoldDSLLaunchAction`. Representa las dos acciones que ejecutan programas *GOLD*, con y sin argumentos.
- `GoldDSLExportToLatexAction`. Representa la acción que exporta un programa *GOLD* a *LaTeX*.
- `GoldDSLTextEditorActionContributor`. Añade todas las acciones a la barra de menú y a la barra de herramientas, contribuyendo con el entorno de programación *Eclipse*.

8.1.16. Generador de código (*code generator*)

El generador de código (*code generator*) es una rutina automática que traduce programas *GOLD* en programas *Java* que posteriormente pueden ser interpretados por la *Máquina Virtual de Java*. Cada archivo con extensión `.gold`

presente en el directorio `src` del proyecto del usuario es sometido a un proceso de compilación que lo transforma en un archivo con extensión `.java` ubicado en el directorio `src-gen`, que sirve como contenedor de todos los archivos generados automáticamente. De esta manera, la estructura de clases *GOLD* que haya definido el usuario en la carpeta `src` se replica para construir su correspondiente estructura de clases *Java* bajo la carpeta `src-gen`.

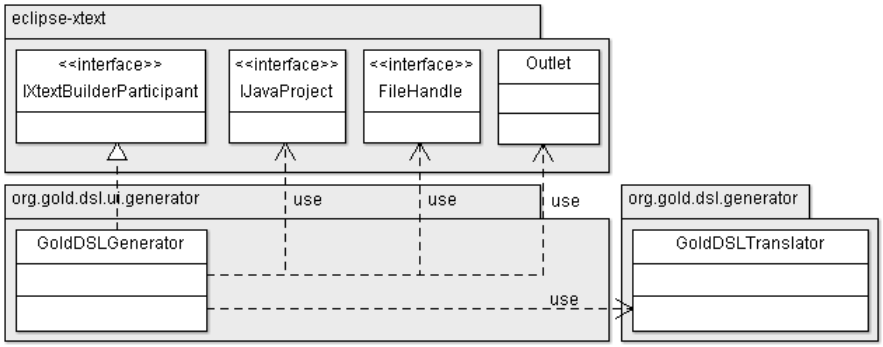
Figura 8.37. *Ejemplo de la estructura interna de los directorios* `src` *y* `src-gen` *de un proyecto GOLD 3.*



El mecanismo descrito permite que los programas *GOLD* puedan ser tratados como programas *Java*, susceptibles de:

- ser utilizados desde clases implementadas en *Java* o en *GOLD*;
- ser ejecutados desde fuera de *Eclipse* o desde otro *IDE*;
- ser distribuidos de forma independiente como ficheros con extensión `.java`;
- ser empaquetados en archivos *JAR*, para su posterior publicación; y
- ser probados intensivamente con herramientas como *JUnit* [56].

Figura 8.38. *Diagrama de clases del paquete* `org.gold.dsl.ui.generator`.



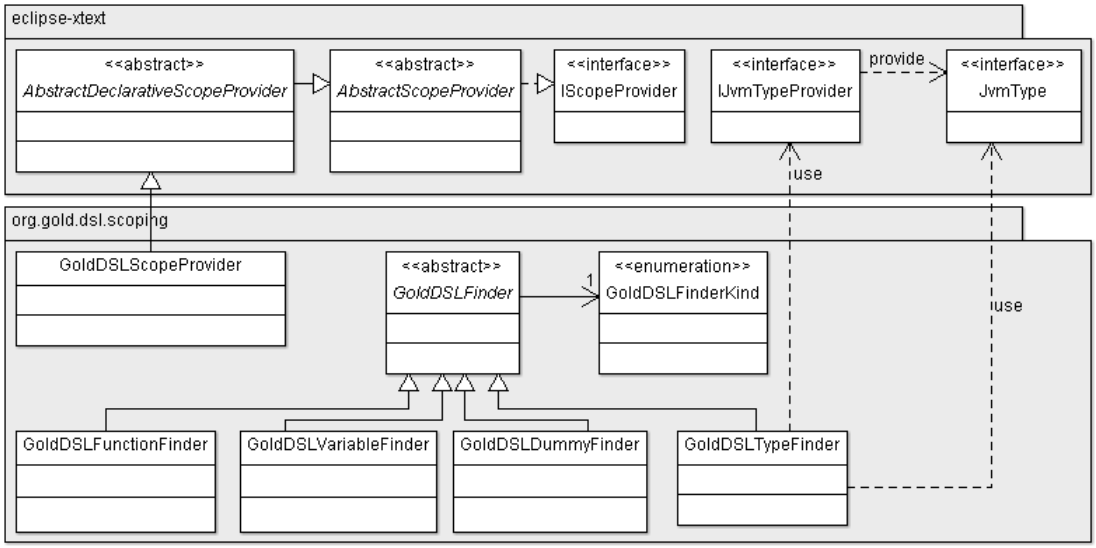
Cada vez que el usuario edite y guarde un programa *GOLD*, el generador de código automáticamente compilará la fuente para producir su correspondiente implementación *Java*. Dado que los programas *GOLD* pueden usar rutinas implementadas en otros programas *GOLD*, podría llegar a ser necesario ejecutar el comando *Proyect* → *Clean* ... de *Eclipse* para corregir errores de compilación relacionados con el uso de procedimientos declarados en archivos separados. Internamente la aplicación del comando *Clean* sobre un proyecto *GOLD* recompila en lote todos los ficheros con extensión `.java` para reconstruir la estructura de clases *Java* que les corresponde.

La clase `GoldDSLGenerator` del paquete `org.gold.dsl.ui.generator` se responsabiliza de invocar un compilador especializado cada vez que se detecte la modificación de algún archivo *GOLD*, y de velar por mantener una perfecta sincronización entre el directorio `src-gen` (donde se encuentran los archivos generados) y el directorio `src` (donde se encuentran los archivos fuente) cada vez que se detecte la inserción o eliminación de archivos *GOLD*. Más adelante, en la sección §8.2 se describirá el compilador que traduce individualmente archivos *GOLD* en archivos *Java*.

8.1.17. Proveedor de alcance (*scope provider*)

El proveedor de alcance (*scope provider*) establece la porción del código fuente de un programa *GOLD* sobre el que cada declaración es efectiva [40], enlazando cada identificador con el elemento sintáctico atado a su declaración, ya sea una variable, una función o un procedimiento. De esta manera se puede establecer la entidad específica que le corresponde a cada referencia dependiendo de su contexto, conociendo de antemano las entidades (variables, funciones o procedimientos) atadas a cada identificador en su respectiva declaración.

Figura 8.39. *Diagrama de clases del paquete* `org.gold.dsl.scoping`.



El paquete `org.gold.dsl.scoping` contiene las clases que implementan el proveedor de alcance a través de buscadores de entidades que respetan las reglas sobre alcance (*scope*) consignadas en la sección §7.2.8, dependiendo del contexto de la referencia que desea resolverse:

- `GoldDSLFinderKind`. Representa la enumeración de los tipos de búsqueda que se pueden realizar: `PREFIX` para buscar todas las entidades declaradas con un identificador que tenga algún prefijo que coincida con el texto de la referencia a resolver y `EQUALITY` para buscar todas las entidades declaradas con un identificador que corresponda exactamente con el texto de la referencia a resolver. En ambos casos la búsqueda es sensible a las mayúsculas (*case sensitive*).
- `GoldDSLFinder`. Representa un buscador de entidades abstracto en *GOLD*, respetando el alcance de cada declaración.

- `GoldDSLFunctionFinder`. Representa un buscador de entidades en *GOLD* considerando únicamente funciones y procedimientos.
- `GoldDSLVariableFinder`. Representa un buscador de entidades en *GOLD* considerando únicamente variables.
- `GoldDSLDummyFinder`. Representa un buscador de entidades en *GOLD* considerando únicamente variables ligadas a las cuantificaciones (*dummies*).
- `GoldDSLTypeFinder`. Representa un buscador de tipos primitivos de datos y de clases *Java* en *GOLD* dependiendo del *classpath* configurado y de los paquetes importados en el programa *GOLD* a través de sentencias de tipo *import*. Adicionalmente, resuelve llamados a constructores, accesos a atributos e invocaciones a métodos.
- `GoldDSLScopeProvider`. Representa el proveedor de alcance de *GOLD*, con implementación vacía. El comportamiento relacionado con el alcance está implementado en todas las clases que extienden de la clase abstracta `GoldDSLFinder` mas no en la clase `GoldDSLScopeProvider`.

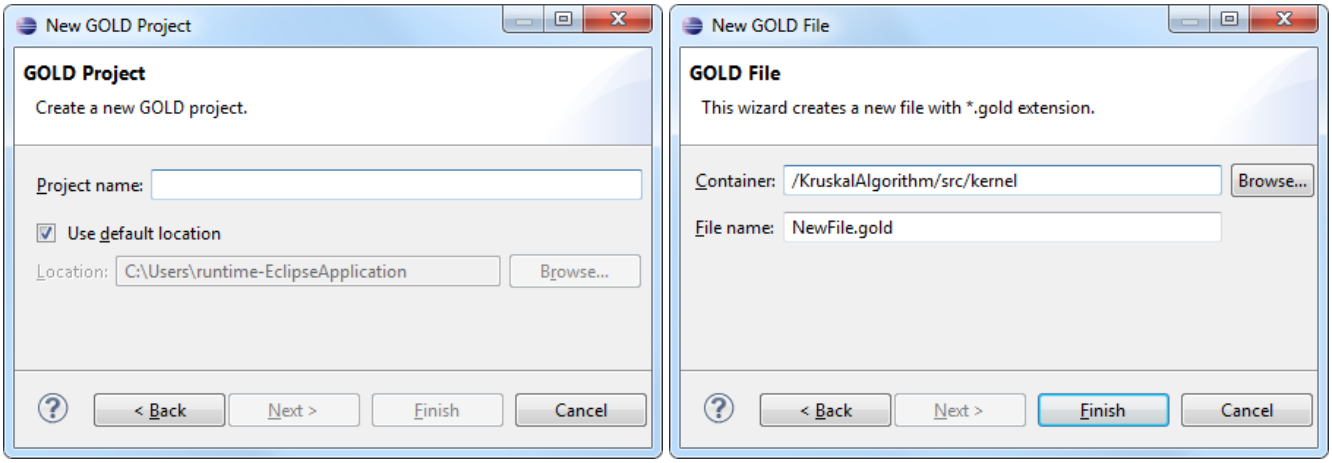
8.1.18. Asistentes (*wizards*)

Los asistentes (*wizards*) son ventanas gráficas que le permiten al usuario desarrollar una determinada actividad paso a paso de manera guiada a través del diligenciamiento de formularios. En *GOLD* se configuraron dos asistentes:

1. *Asistente de creación de proyectos* (*New project wizard*). Ayuda al usuario en el proceso de creación de un nuevo proyecto *GOLD* con la codificación de caracteres *UTF-8*.
2. *Asistente de creación de archivos* (*New file wizard*). Ayuda al usuario en el proceso de creación de un nuevo archivo *GOLD* con la extensión `.gold` y la codificación de caracteres *UTF-8*.

Figura 8.40. *Asistentes para la creación de proyectos y archivos en GOLD 3.*

(a) *Asistente para la creación de un nuevo proyecto GOLD.* **(b)** *Asistente para la creación de un nuevo archivo GOLD.*

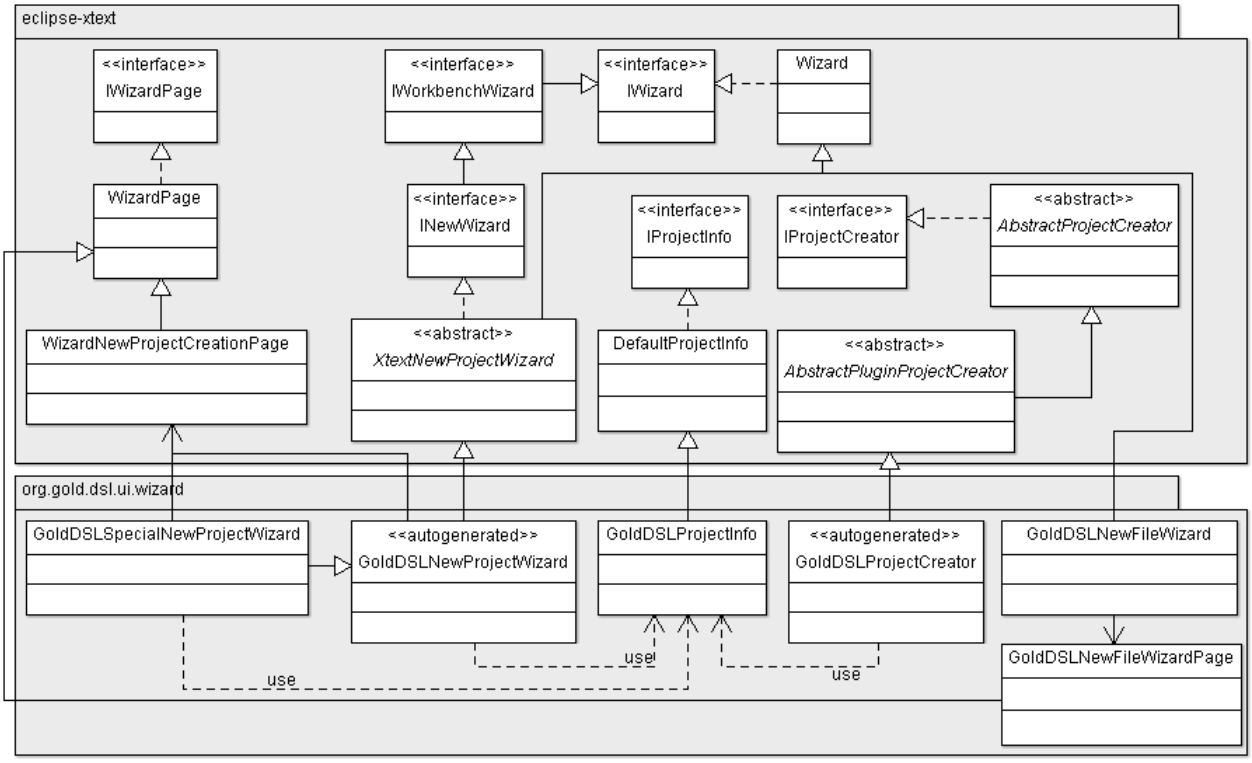


Si el usuario decide no utilizar el asistente de creación de proyectos para construir un nuevo proyecto *GOLD*, tendría que configurar manualmente su estructura para que referencie el *plug-in* de *GOLD*, importar las librerías *JUNG* [21] y *Apfloat* [53], y cambiar la codificación de caracteres a *UTF-8*. De manera similar, si el usuario decide no utilizar el asistente de creación de archivos para generar un nuevo archivo *GOLD*, tendría que asignarle manualmente la extensión `.gold` y cambiar la codificación de caracteres a *UTF-8*. Es importante que los proyectos y que los archivos *GOLD* estén codificados con el formato *UTF-8* por las razones expuestas en la sección §8.1.4.

El paquete `org.gold.dsl.ui.wizard` contiene las clases que implementan los asistentes:

- `GoldDSLNewProjectWizard`. Representa el asistente para la creación de un nuevo proyecto *GOLD*, configurado por defecto. La clase es generada automáticamente por *Xtext*.
- `GoldDSLSpecialNewProjectWizard`. Representa el asistente para la creación de un nuevo proyecto *GOLD*, especialmente configurado para asignarle la codificación de caracteres *UTF-8* a cada proyecto construido.
- `GoldDSLProjectCreator`. Representa la clase responsable de diligenciar el contenido por defecto de un proyecto recién creado.
- `GoldDSLProjectInfo`. Representa la información básica de un proyecto *GOLD* que va a ser creado. Únicamente almacena el nombre del proyecto en cuestión.
- `GoldDSLNewFileWizardPage`. Representa la única página del asistente para la creación de un nuevo archivo *GOLD*, verificando que el formulario esté correctamente diligenciado (en particular, que el archivo tenga un nombre válido).
- `GoldDSLNewFileWizard`. Representa el asistente para la creación de un nuevo archivo *GOLD*, especialmente configurado para asignarle la codificación de caracteres *UTF-8* a cada archivo construido.

Figura 8.41. *Diagrama de clases del paquete* `org.gold.dsl.ui.wizard`.



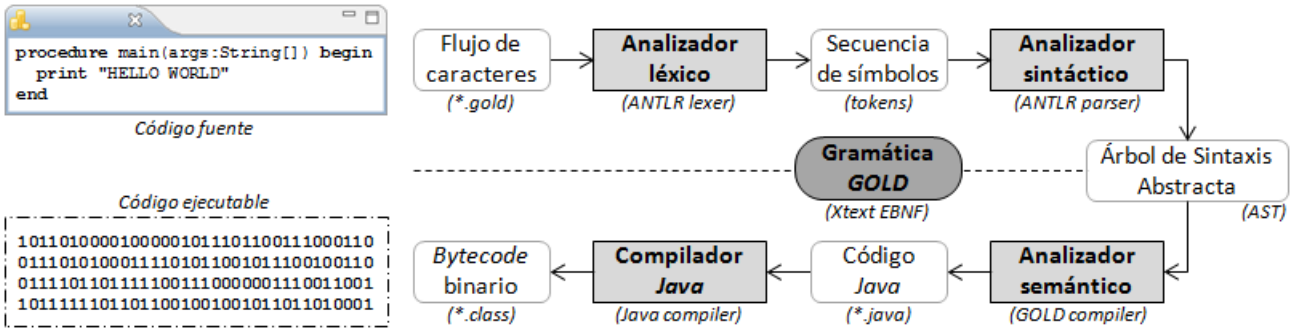
En caso de que en un proyecto *GOLD* aparezca un error de restricción de acceso (*access restriction*) del estilo ''*. . . is not accessible due to restriction on required library . . .* '' se debe poner la opción *Warning* bajo *Project → Properties → Java Compiler → Errors/Warnings → Deprecated and restricted API → Forbidden reference (access rules)* activando previamente el campo *Enable project specific settings* en la ventana *Errors/Warnings*. La misma opción se puede configurar por defecto para todos los proyectos bajo *Window → Preferences → Java → Compiler → Errors/Warnings → Deprecated and restricted API → Forbidden reference (access rules)*.

8.2. Núcleo del lenguaje

Otro módulo importante de un lenguaje de programación es su núcleo (*kernel*), que reúne los componentes que conforman su compilador: el analizador léxico, el analizador sintáctico y el analizador semántico. La implementación del analizador léxico-sintáctico y del modelo semántico fue generada automáticamente por el *framework Xtext 2.2.1* [6] a partir de la definición de la gramática del lenguaje en notación *BNF* extendida [5], sin involucrar trabajo humano adicional al empleado para construir la gramática. Por otro lado, el analizador semántico fue implementado manualmente recorriendo el modelo semántico para traducir programas *GOLD* a código *Java*.

La validación de código (*code validation*) que realiza las comprobaciones semánticas (*checks*) que no están explícitamente consignadas en la gramática del lenguaje, el resaltado de la sintaxis (*syntax highlighting*) y el resto de características relacionadas con el ambiente de programación se mencionan en la sección §8.1. No se debe olvidar que, como la infraestructura del lenguaje se distribuye como un *plug-in* de *Eclipse*, se están heredando gratuitamente todas las funcionalidades inherentes al entorno de desarrollo *Eclipse*, incluyendo su interfaz gráfica, su compilador *Java* y su librería *JDT* [7] (*Java Development Tools*).

Figura 8.42. *Proceso de compilación de programas escritos en GOLD 3.*



8.2.1. Gramática (*grammar*)

El aspecto más relevante de la implementación del núcleo del lenguaje es la definición de su gramática a través de la notación *EBNF* [5] [†8] en el formato establecido por *Xtext* [6]. La gramática *EBNF* definida en la sección §A.1.1 tuvo que adaptarse al formato de *Xtext* factorizando por la izquierda (*left factoring*) cada una de sus reglas de producción para evitar el uso de *backtracking* y *look-ahead*, y enriqueciendo la sintaxis con los distintos mecanismos proporcionados por *Xtext* para guiar la generación automática del modelo semántico del lenguaje.

El archivo `src/org/gold/dsl/GoldDSL.xtext` del proyecto `org.gold.dsl` contiene la implementación de la gramática *EBNF* en la notación provista por *Xtext* (véase la sección §A.1.2). Para construir automáticamente todos los artefactos del lenguaje incluyendo el analizador léxico-sintáctico y el modelo semántico se debe ejecutar el generador automático de código de *Xtext* [6] seleccionando el archivo `src/org/gold/dsl/GenerateGoldDSL.mwe2` y activando la opción *Run As* → *MWE2 Workflow* del menú contextual de *Eclipse*.

El *script* `GenerateGoldDSL.mwe2` está escrito en un lenguaje de propósito específico especial llamado *Modeling Workflow Engine 2* [6] (*MWE2*), y su labor principal es la de configurar el generador de código de *Xtext* para que sea capaz de producir los diferentes componentes del lenguaje en el directorio `src-gen` de cada uno de los proyectos que componen la implementación de *GOLD*. De esta manera, cuando se modifique la gramática del lenguaje se puede reconstruir toda su infraestructura automáticamente, incluyendo el analizador léxico-sintáctico, el modelo semántico y el *IDE* genérico que viene implementado por defecto. Luego, el comportamiento adicional puede implementarse añadiendo código en el directorio `src` de cada proyecto sin modificar ningún archivo del directorio `src-gen`.

[8] La notación *BNF* extendida (*EBNF*: *Extended Backus-Naur Form*) es un formalismo para definir gramáticas independientes del contexto que suele usarse para describir la sintaxis de los lenguajes de programación.

Figura 8.43. *Ejecución del generador de código de Xtext en GOLD 3.*

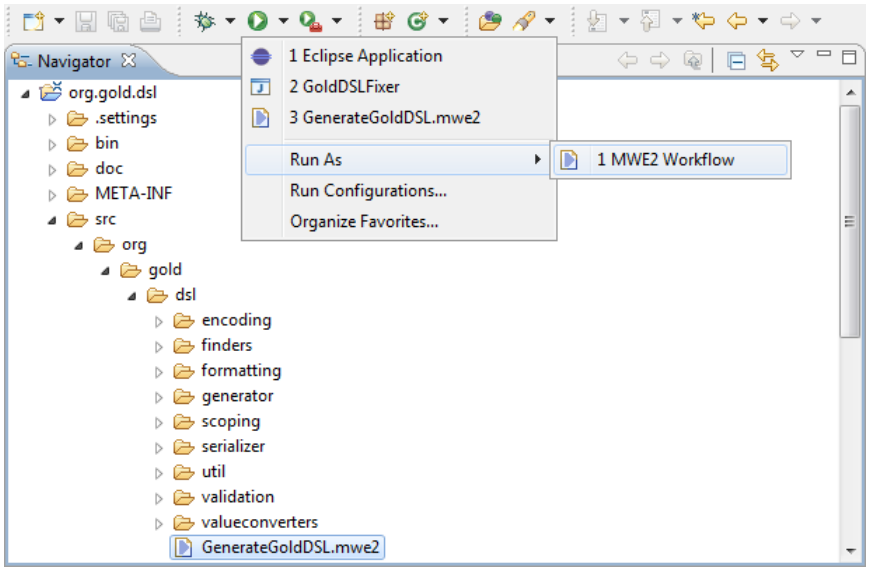
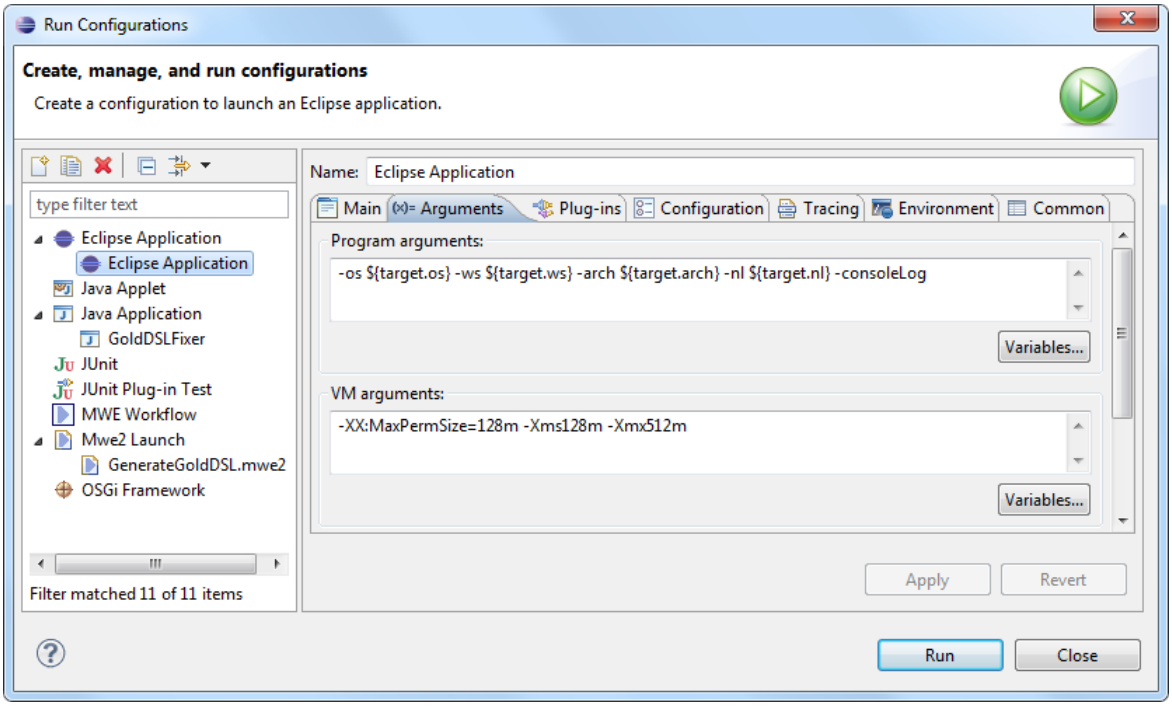


Figura 8.44. *Configuración de la nueva instancia de Eclipse para probar el plug-in de GOLD 3.*



Después de ejecutar la generación automática de código, una falla interna (*bug*) de *Xtext 2.2.1* provoca que el método `createSequence` de la clase `AbstractGoldDSLSemanticSequencer` del paquete `org.gold.dsl.serializer` desborde la longitud máxima definida en *Java* para el tamaño del código fuente que lo implementa, lanzando el error de compilación ‘‘*The code of method* `createSequence(EObject,EObject)` *is exceeding the 65535 bytes limit*’’. Para corregir este problema después de correr el *script* `GenerateGoldDSL.mwe2`, es necesario ejecutar el programa `org.gold.dsl.GoldDSLFixer`, que altera el contenido de la clase `AbstractGoldDSLSemanticSequencer` creando métodos auxiliares y reorganizando la implementación del método `createSequence` para que su contenido no supere el umbral de 65536 bytes.

Finalmente, para probar el *IDE* como un *plug-in* de *Eclipse*, se debe seleccionar la opción *Run* → *Run Configurations...* en la barra de menú de *Eclipse*, crear una configuración para ejecutar una nueva aplicación *Eclipse* (*Eclipse Application*) y poner como argumentos de la *Máquina Virtual* (*Arguments* → *VM arguments*) el texto `-XX:MaxPermSize=128m -Xms128m -Xmx512m` para darle memoria suficiente a la nueva instancia de *Eclipse* que albergará el *plug-in* [6]. A continuación se debe ejecutar una nueva instancia de *Eclipse* según la configuración recién creada, que tendrá instalado el *plug-in* de *GOLD* listo para agregar proyectos con la opción *New* → *Other* → *GOLD3* → *GOLD Project* y para crear archivos *GOLD* con la opción *New* → *Other* → *GOLD3* → *GOLD File*.

8.2.2. Analizador léxico (*lexer*)

El analizador léxico (*lexer*) es el componente que se responsabiliza de transformar la secuencia de caracteres *Unicode* que representa el código fuente de un programa *GOLD* en una secuencia de símbolos terminales (*tokens*). Luego del análisis léxico (*lexing*) cada uno de los *tokens* generados debe corresponder unívocamente con alguna palabra reservada (*reserved word*) o con algún símbolo terminal definido en la gramática bajo alguna regla léxica (*terminal rule*) descrita en la notación de *Xtext*.

Código 8.2. *Definición de los símbolos terminales de GOLD en Xtext, exceptuando las palabras reservadas.*

```
// -------------------------------------------------------------------------------------------
// TERMINAL FRAGMENTS
// -------------------------------------------------------------------------------------------
terminal fragment LETTER: // Latin alphabet and greek alphabet
  'A'..'Z'|'a'..'z'|'\u0391'..'\u03A9'|'\u03B1'..'\u03C9';
terminal fragment DIGIT: // Decimal digits
  '0'..'9';
terminal fragment SUBINDEX: // Numerical subscript digits
  '\u2080'..'\u2089';
terminal fragment HEX_DIGIT: // Hexadecimal digits
  '0'..'9'|'A'..'F'|'a'..'f';
terminal fragment HEX_CODE: // Unicode character scape code
  'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT;
// -------------------------------------------------------------------------------------------

// -------------------------------------------------------------------------------------------
// TERMINAL RULES
// -------------------------------------------------------------------------------------------
terminal CONSTANT: // Basic constants
    'TRUE'|'true'|'FALSE'|'false' // Boolean values
  | 'NIL'|'nil'|'NULL'|'null'     // Null pointer
  | '\u00D8'                      // Empty set/bag
  | '\u22C3'                      // Universe set
  | '\u025B'                      // Empty sequence
  | '\u03BB'                      // Empty string
  | '\u221E'                      // Positive infinity
;
terminal PRIMITIVE_TYPE: // Primitive types (basic mathematical sets)
    '\u212C' // Boolean values
  | '\u2115' // Natural numbers
  | '\u2124' // Integer numbers
  | '\u211A' // Rational numbers
  | '\u2148' // Irrational numbers
  | '\u211D' // Real numbers
  | '\u2102' // Complex numbers
;
terminal NUMBER: // Integer and floating point numbers
  DIGIT+ ('.' DIGIT+)? (('E'|'e') '-'? DIGIT+)?
```

```
39    ('L'|'l' // long    (java.lang.Long)
40    |'I'|'i' // int     (java.lang.Integer)
41    |'S'|'s' // short   (java.lang.Short)
42    |'B'|'b' // byte    (java.lang.Byte)
43    |'D'|'d' // double  (java.lang.Double)
44    |'F'|'f' // float   (java.lang.Float)
45    |'C'|'c' // char    (java.lang.Character)
46    )?
47  ;
48  terminal ID: // Identifiers
49    '$'? // Optional escape character
50    (LETTER|'_') (LETTER|'_'|DIGIT|SUBINDEX|'\u00B4')*;
51  terminal STRING: // Strings
52    '"'(('\\'('b'|'t'|'n'|'f'|'r'|HEX_CODE|'"'|"'"|'\\'))|!('\\'|'"'))*'"';
53  terminal CHARACTER: // Characters
54    "'"(('\\'('b'|'t'|'n'|'f'|'r'|HEX_CODE|'"'|"'"|'\\'))|!('\\'|"'"))"'";
55  terminal JAVA_CODE: // Java native code
56    '/?' -> '?/';
57  terminal ML_COMMENT: // Multi-line comments
58    '/*' -> '*/';
59  terminal SL_COMMENT: // Single-line comments
60    ('\u29D0'|'//') // Comment mark
61    !('\n'|'\r')*;
62  terminal WS: // Whitespaces
63    (' '|'\t'|'\r'|'\n')+;
64  // -------------------------------------------------------------------------------------
```

A partir de la gramática en notación *EBNF* (véase la sección §A.1.2), el analizador léxico de *GOLD* es generado automáticamente por *Xtext* usando *ANTLR* [57] (*ANother Tool for Language Recognition*), que es una herramienta para la construcción de compiladores que se encuentra integrada dentro de *Xtext*.

8.2.3. Analizador sintáctico (*parser*)

El analizador sintáctico (*parser*) es el componente que se responsabiliza de transformar la secuencia de símbolos terminales (*tokens*) generada por el analizador léxico (*lexer*) en un árbol de derivación cuya estructura está determinada por las reglas de producción de la gramática. Luego del análisis sintáctico (*parsing*), *Xtext* alimenta automáticamente un metamodelo de clases que representa los elementos sintácticos del lenguaje mediante una estructura de datos arbórea denominada *Árbol de Sintaxis Abstracta* (*AST*: *Abstract Syntax Tree*). Claramente, una representación jerárquica es más versátil que una representación textual cruda porque la primera facilita la navegación de los distintos elementos sintácticos que componen un programa *GOLD*.

A partir de la gramática en notación *EBNF* (véase la sección §A.1.2), el analizador sintáctico de *GOLD* también es generado automáticamente por *Xtext* usando *ANTLR* [57].

8.2.4. Modelo semántico (*semantic model*)

El resultado del análisis sintáctico llevado a cabo automáticamente por *Xtext* [6] a través de *ANTLR* [57] es un metamodelo de clases que representa los elementos sintácticos del lenguaje mediante una estructura de datos arbórea denominada *Árbol de Sintaxis Abstracta* (*AST*: *Abstract Syntax Tree*). Este árbol de sintaxis oficia como el modelo semántico (*semantic model*) [49] de un programa *GOLD*, proveyendo una estructura jerárquica que describe su significado conceptual en memoria principal a través de clases que representan cada uno de sus componentes semánticos.

Prácticamente todas las características del *IDE* deben recorrer el *Árbol de Sintaxis Abstracta* generado automáticamente por *Xtext* para escudriñar la estructura de cualquier programa *GOLD*, capturando su significado. El

modelo de clases producido automáticamente por *Xtext* que implementa el modelo semántico es un fiel reflejo de la gramática definida en la sección §A.1.2, donde cada símbolo no terminal es transformado en una clase cuyo nombre coincide con su identificador y cuyos miembros (métodos y atributos) corresponden con los distintos elementos que conforman su regla de producción.

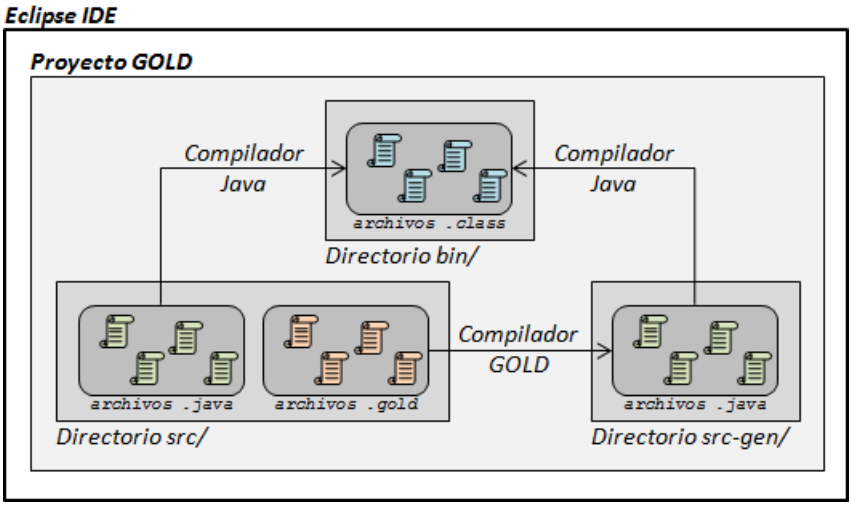
8.2.5. Analizador semántico (*semantic analyzer*)

El analizador semántico (*semantic analyzer*) de *GOLD* es el componente que traduce código *GOLD* a código *Java*. Recibe como entrada la referencia a un objeto de tipo `GoldProgram` representando la raíz del *Árbol de Sintaxis Abstracta* que modela un programa *GOLD* y entrega como respuesta una cadena de texto representando la clase implementada en *Java* producto de la traducción (véase la figura 8.42).

Cada vez que se detecta la modificación (en disco duro mas no en el editor de código) de un archivo con extensión `.gold` presente en el directorio `src` de algún proyecto del usuario, el generador de código fuente explicado en la sección §8.1.16 lee el contenido del archivo como un flujo de caracteres, ejecuta un análisis léxico-sintáctico sobre este flujo para obtener el modelo semántico que describe el programa, invoca el analizador semántico de *GOLD* pasándole como parámetro el modelo semántico obtenido, recibe la cadena de texto que el analizador semántico entrega como respuesta, y finalmente escribe la cadena de texto en un archivo con extensión `.java` ubicado dentro del directorio `src-gen` del mismo proyecto. De esta manera se traducen individualmente programas *GOLD* en programas *Java*, que posteriormente pueden ser interpretados por la *Máquina Virtual de Java* como si se tratase de código ejecutable.

En caso de que un programa *GOLD* tenga definido un procedimiento propio con nombre `main` que reciba un único parámetro de tipo `String[]`, la traducción generará una clase *Java* con un método estático *main(String[])* sin retorno, representando el método *main* de la aplicación. Cada uno de los demás procedimientos y funciones también se traducen en métodos estáticos de la clase *Java* que fue generada.

Figura 8.45. *Esquema que ilustra el proceso de compilación de archivos GOLD 3 en un proyecto creado en Eclipse.*



El proceso de compilación recién descrito permite que en un mismo proyecto *Eclipse* haya programas *GOLD* que usan rutinas implementadas en *Java* y clases *Java* que usan funciones y procedimientos implementados en *GOLD*. De esta manera los lenguajes *GOLD* y *Java* terminan siendo integrados de forma transparente para el desarrollador, quien los podría mezclar indistintamente en sus proyectos.

Todos los archivos implementados en *GOLD* son transformados en clases *Java* para que al final del proceso de compilación únicamente se tenga código *Java* que puede ser ejecutado y depurado en *Eclipse* como en cualquier

proyecto *Java*. El acercamiento planteado para procesar los programas escritos en *GOLD* es muy distinto al decidido en la implementación de *GOLD+*, su predecesor. El lenguaje *GOLD+* está sujeto a un proceso de *interpretación* [49] puesto que cada una de las instrucciones es analizada para ser inmediatamente ejecutada por una máquina especializada en prestar operaciones básicas sobre el dominio de los grafos. En contraste, el lenguaje *GOLD 3* está sujeto a un proceso de *compilación* [49] puesto que los programas escritos en archivos *GOLD* (con extensión `.gold`) son analizados para producir una salida intermedia que consiste de archivos *Java* (con extensión `.java`), que a su vez son procesados por el compilador estándar de *Java* (`javac`) para generar archivos (con extensión `.class`) conteniendo *bytecode Java* independiente de la plataforma, que termina siendo ejecutado por la *Máquina Virtual de Java* para obtener el comportamiento deseado con el pseudocódigo inicialmente desarrollado en *GOLD*. Este proceso de compilación es catalogado como *generación de código* [49] porque el pseudocódigo escrito en *GOLD* termina transformándose en código escrito en *Java* que está en capacidad de interactuar sin dificultad con el *API* estándar de *Java*, con librerías externas codificadas en *Java* y con clases *Java* implementadas por el usuario.

La clase `GoldDSLTranslator` del paquete `org.gold.dsl.generator` implementa el analizador semántico que traduce código *GOLD* a código *Java*, convirtiendo el modelo semántico de un programa *GOLD* en una cadena de caracteres que representa su correspondiente traducción como una clase codificada en *Java*. Internamente, la clase `GoldDSLTranslator` está compuesta por múltiples rutinas que recorren completamente el *Árbol de Sintaxis Abstracta* de un programa *GOLD*, escribiendo el código *Java* producto de la traducción en un *buffer* de caracteres de tipo `java.lang.StringBuilder`. Cada elemento sintáctico del lenguaje es traducido fielmente a *Java* usando la semántica denotacional definida en la sección §7.3.1 como se puede evidenciar en los siguientes fragmentos de código de la clase `GoldDSLTranslator`, que se exhiben como ejemplos representativos para ilustrar el proceso de traducción.

Código 8.3. *Traducción de una instrucción condicional if-then-else de GOLD a Java.*

```
1  private void translateIfThenElse(IfThenElse pIfThenElse) {
2    writeLine("if ("+cast("boolean",pIfThenElse.getGuard())+") {");
3    translateInstructions(pIfThenElse.getIf().getBody());
4    writeLine("}");
5    for (ElseIf elseIf:pIfThenElse.getElseIfs()) {
6      writeLine("else {");
7      writeLine("if ("+cast("boolean",elseIf.getGuard())+") {");
8      translateInstructions(elseIf.getBody());
9      writeLine("}");
10   }
11   if (pIfThenElse.getElse()!=null) {
12     writeLine("else {");
13     translateInstructions(pIfThenElse.getElse().getBody());
14     writeLine("}");
15   }
16   for (int counter=pIfThenElse.getElseIfs().size(); counter>0; counter--) {
17     writeLine("}");
18   }
19 }
```

Código 8.4. *Traducción de una sentencia while de GOLD a Java.*

```
1  private void translateWhile(While pWhile) {
2    writeLine("while (true) {");
3    writeLine("if (!("+cast("boolean",pWhile.getGuard())+")) break;");
4    translateInstructions(pWhile.getBody());
5    writeLine("}");
6  }
```

8.3. Librería de clases

El último gran componente del lenguaje *GOLD* es su biblioteca de clases, que provee un *API* (*Application Programming Interface*) que implementa algunas estructuras de datos fundamentales como los árboles, los grafos y los autómatas, y proporciona algunas rutinas útiles que facilitan su manipulación y visualización usando las librerías externas descritas en la sección §6.2:

- *JUNG* [21] (*Java Universal Network/Graph Framework*). Librería *Java* que ofrece un *API* extensible para el modelado, análisis y visualización de grafos (véase la sección §6.2.1).
- *JGraphT* [22]. Librería *Java* que suministra una colección de clases y de algoritmos diseñados para trabajar sobre teoría de grafos (véase la sección §6.2.2).
- *Implementaciones de referencia de Cormen et al.* [1]. Librería *Java* que suministra implementaciones de referencia para la mayoría de las estructuras de datos y algoritmos presentados en el libro *Introduction to Algorithms* [1] (véase la sección §6.2.3).
- *Apfloat* [53]. Librería *Java* que provee tipos de datos para la representación de números, y rutinas diseñadas para el desarrollo de operaciones aritméticas de precisión arbitraria sobre éstos (véase la sección §6.2.4).

El código fuente que implementa la librería de clases provista por *GOLD* está escrito en *Java* y se encuentra presente en el directorio `src-dist` del proyecto `org.gold.dsl` bajo el paquete `gold`. Aunque la biblioteca suministrada por *GOLD* es bastante completa, el usuario podría utilizar en sus programas *GOLD* cualquier otra librería especializada en teoría de grafos u otras estructuras de datos. Esto es posible porque cualquier clase *Java* puede ser mencionada desde cualquier programa *GOLD*, incluyendo:

- tipos primitivos del lenguaje de programación *Java*;
- clases del *API* estándar de *Java*;
- clases de librerías externas empaquetadas en archivos *JAR* o distribuidas en archivos *.class*;
- clases del usuario u otras personas, cuya implementación esté disponible en archivos *.java*; y
- clases de la librería suministrada por *GOLD*.

En la biblioteca de clases se aplican las siguientes convenciones de nombres para facilitar su utilización:

- Toda interfaz tiene un nombre que comienza por `I` (e.g., `ISet`, `IGraph`).
- Toda anotación `@interface` tiene un nombre que comienza por `@IIs` (e.g., `@IIsUndirected`).
- Toda clase tiene un nombre que comienza por `G` (e.g., `GRedBlackTreeSet`, `GDirectedGraph`).
- Toda clase abstracta tiene un nombre que comienza por `GAbstract` (e.g., `GAbstractSet`, `GAbstractGraph`).
- Toda clase concreta tiene un nombre que comienza por `G` pero que no comienza por `GAbstract` (e.g., `GRedBlackTreeSet`, `GDirectedGraph`).
- Toda clase concreta que representa un adaptador (*adapter*) de una determinada estructura de datos tiene un nombre que comienza por `GAdaptor` (e.g., `GAdaptorSet`, `GAdaptorJungGraph`).
- Toda clase concreta que representa una vista (*view*) de una determinada estructura de datos tiene un nombre que comienza por `GView` (e.g., `GViewJungGraph`).

El diagrama de paquetes, los diagramas de clases *UML* y la descripción de cada clase perteneciente a la librería *GOLD* se editaron en *ArgoUML* [84] y pueden consultarse en la documentación técnica bajo la sección §A.6.2.

8.3.1. Estructuras de datos

GOLD permite la declaración y manipulación de las siguientes estructuras de datos:

Tabla 8.7. *Estructuras de datos e implementaciones provistas por GOLD 3.*

Estructura de datos	Implementaciones suministradas
Tupla (*tuple*)	Tuplas vacías, *singletons*, pares ordenados, *n*-tuplas.
Lista/secuencia (*list*/*sequence*)	Vectores dinámicos, Listas doblemente encadenadas.
Conjunto (*set*)	Árboles Rojinegros, Árboles *AVL*, Tablas de *Hashing*, *Skip Lists*.
Bolsa/multiconjunto (*bag*/*multiset*)	Árboles Rojinegros, Árboles *AVL*, Tablas de *Hashing*, *Skip Lists*.
Pila (*stack*)	Vectores dinámicos, Listas doblemente encadenadas.
Cola (*queue*)	Vectores dinámicos circulares, Listas doblemente encadenadas.
Bicola (*deque*)	Vectores dinámicos circulares, Listas doblemente encadenadas.
Montón (*heap*)	*Binary heaps*, *Binomial heaps*, *Fibonacci heaps*, Árboles Rojinegros.
Conjuntos disyuntos (*disjoint-sets*)	*Disjoint-set forests*, Listas encadenadas.
Árbol binario (*binary tree*)	Árboles sencillamente encadenados.
Árbol eneario (*n-ary tree*)	Vectores dinámicos, *Quadtrees*, *Tries*.
Asociación llave-valor (*map*)	Árboles Rojinegros, Árboles *AVL*, Tablas de *Hashing*, *Skip Lists*.
Asociación llave-valores (*multimap*)	Árboles Rojinegros, Árboles *AVL*, Tablas de *Hashing*, *Skip Lists*.
Grafo (*graph*)	Listas de adyacencia, Matrices de adyacencia, Representaciones implícitas.
Autómata (*automaton*)	Representaciones explícitas (Tablas de *Hashing*), Representaciones implícitas.
Autómata con respuesta (*transducer*)	Representaciones explícitas (Tablas de *Hashing*), Representaciones implícitas.
Autómata de pila (*pushdown automaton*)	Representaciones explícitas (Tablas de *Hashing*).

El paquete `gold.structures` contiene una multitud de clases que implementan las estructuras de datos enumeradas en la tabla 8.7 a través de interfaces sofisticadas que facilitan su manipulación desde un punto de vista matemático †[9], organizadas en varios subpaquetes:

Tabla 8.8. *Subpaquetes que conforman el paquete* `gold.structures` *de GOLD 3.*

Subpaquete	Estructuras de datos
`automaton`	Autómatas determinísticos (*deterministic automata*), autómatas no determinísticos (*nondeterministic automata*), autómatas con respuesta (*transducers*), autómatas de pila (*pushdown automata*).
`bag`	Bolsas/multiconjuntos (*bags*/*multisets*).
`collection`	Colecciones (*collections*).
`deque`	Bicolas (*deques*).
`disjointset`	Conjuntos disyuntos (*disjoint-set data structure*).
`graph`	Grafos dirigidos (*directed graphs*), grafos no dirigidos (*undirected graphs*).
`heap`	Montones (*heaps*).
`list`	Listas/secuencias (*lists*/*sequences*).
`map`	Asociaciones llave-valor (*maps*).
`multimap`	Asociaciones llave-valores (*multimaps*).
`point`	Puntos del plano cartesiano (*points*), puntos con coordenadas enteras (*lattice points*).
`queue`	Colas (*queues*).
`set`	Conjuntos (*sets*), conjuntos con complemento finito (*cofinite sets*).
`stack`	Pilas (*stacks*).
`tree`	Árboles (*trees*).
`tree.binary`	Árboles binarios (*binary trees*).
`tree.nary`	Árboles enearios (*n-ary trees*).
`tuple`	Tuplas (*tuples*).

[9] Por ejemplo, el método `retainAll` de la interfaz `java.util.Set` tiene como contraparte en *GOLD* el método `intersection` de la interfaz `gold.structures.set.ISet`. Desde un punto de vista matemático, el nombre `intersection` es más apropiado que el nombre `retainAll` para la denotar la operación de intersección de conjuntos.

Con el fin de asegurar una completa compatibilidad entre las interfaces provistas por *GOLD* y las provistas por *Java*, se ofrecen dos mecanismos diseñados exclusivamente para que puedan trabajar conjunta e indistintamente:

1. *Adaptadores* (*Adapters*). Muchas clases del *API* estándar de *Java 6* que representan estructuras de datos se pueden tratar como objetos *GOLD* mediante el uso de clases especializadas denominadas *adaptadores* (*adapters*), que implementan el patrón *Adapter* para traducir las interfaces del *API* de *Java* en interfaces compatibles con *GOLD*. Internamente, cada adaptador convierte los llamados a métodos de una interfaz *GOLD* en invocaciones sobre una interfaz *Java* específica.

2. *Vistas* (*Views*). Todas las clases que representan estructuras de datos en *GOLD* tienen un método que retorna una vista que adapta la estructura como un objeto que implementa su respectiva interfaz *Java*. La estructura retornada refleja la original, de tal forma que cualquier cambio sobre el objeto entregado es también realizado sobre el objeto original, y viceversa.

La mayoría de estructuras de datos en *GOLD* fueron implementadas usando el patrón *Delegation*, donde cada clase del *API* de *GOLD* (que actúa como *delegador*) delega la ejecución de sus operaciones a alguna clase *Java* (que actúa como *delegado*) que define detalladamente las propiedades y el comportamiento interno particular a la estructura de datos, pudiendo reutilizar así el código fuente de librerías externas como el *API* estándar de *Java*, *JUNG* [21] y *JGraphT* [22], entre otras. Por ejemplo, la clase `gold.structures.set.GRedBlackTreeSet` representa un conjunto implementado con Árboles Rojinegros que delega todas sus operaciones a una instancia de la clase `java.util.TreeSet`, ahorrándose la difícil labor de reimplementar esta estructura de datos *desde cero* sin aprovechar la implementación suministrada por el *API* estándar de *Java*, que ha sido probada intensivamente por millones de personas. Además, la biblioteca de clases de *GOLD* fue implementada siguiendo los patrones de asignación de responsabilidades *GRASP* (*General Responsibility Assignment Software Patterns*) como el patrón *Information Expert* y los principios de *Alta Cohesión* (*High Cohesion*) y *Bajo Acoplamiento* (*Low Coupling*).

De esta manera, el desarrollador puede usar en sus programas *GOLD* estructuras de datos de la librería *GOLD* o de la librería estándar de *Java*, intercambiando las interfaces de ambas a través de los adaptadores y de las vistas provistas, dependiendo de lo que más le convenga. Más aún, el usuario podría usar estructuras de datos importadas de otras librerías como *JUNG* [21] y *JGraphT* [22] (por ejemplo), o codificar las propias extendiendo alguna clase abstracta o implementando cualquiera de las interfaces ofrecidas por *GOLD* o por *Java*. De hecho, muchas de las estructuras de datos suministradas por *GOLD* son adaptaciones de implementaciones provistas por alguna librería externa, a través de los patrones *Adaptor* y *Delegation*:

- *JGraphT* [22]. Provee a *GOLD* la implementación de montones (*heaps*) a través de *Fibonacci heaps* [1] (`com.jgrapht.util.FibonacciHeap`).

- *Implementaciones de referencia de Cormen et al.* [1]. Provee a *GOLD* la implementación de montones (*heaps*) a través de *Binomial heaps* [1] (`com.mhhe.clrs2e.BinomialHeap`).

- *API estándar de Java*. Provee a *GOLD* implementaciones de un sinnúmero de estructuras de datos, incluyendo vectores dinámicos (`java.util.ArrayList`), vectores dinámicos circulares (`java.util.ArrayDeque`), listas doblemente encadenadas en anillo (`java.util.LinkedList`), *Binary heaps* (`java.util.PriorityQueue`), Árboles Rojinegros (`java.util.TreeSet`, `java.util.TreeMap`), Tablas de *Hashing* (`java.util.Hashtable`, `java.util.HashSet`, `java.util.HashMap`, `java.util.LinkedHashSet`, `java.util.LinkedHashMap`), y *Skip-Lists* (`java.util.concurrent.ConcurrentSkipListSet`, `java.util.concurrent.ConcurrentSkipListMap`).

Las implementaciones mencionadas en la tabla 8.7 que no fueron enumeradas en la lista anterior se desarrollaron exclusivamente para *GOLD* sin adaptar estructuras de datos de librerías externas [†10].

[10] Para mayor información, véase la descripción detallada de cada clase de la librería *GOLD* en la sección §A.6.2.

8.3.2. Tipos de datos numéricos

GOLD permite la manipulación de valores pertenecientes a conjuntos matemáticos como los caracteres, los booleanos ($\mathbb{B}$), los naturales ($\mathbb{N}$), los enteros ($\mathbb{Z}$), los racionales ($\mathbb{Q}$), los reales ($\mathbb{R}$) y los complejos ($\mathbb{C}$) a través de tipos primitivos de datos y de librerías de alto rendimiento para el desarrollo de operaciones aritméticas de precisión arbitraria. Para representar tipos numéricos en *GOLD* se usa explícitamente la librería *Apfloat* [53], aunque el usuario estaría en libertad de utilizar otras implementaciones como las clases provistas por *Java* para representar números de precisión arbitraria (`java.math.BigInteger` y `java.math.BigDecimal`), los ocho tipos primitivos de *Java* (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`) o cualquier otra librería especializada en manejar números grandes. Cada uno de los símbolos de los tipos primitivos de *GOLD* se puede tratar como un alias de su correspondiente implementación, según lo descrito en la tabla 8.10 (e.g., $\mathbb{Z}$ es una abreviación del nombre calificado `org.apfloat.Apint`).

Tabla 8.9. *Tipos primitivos del lenguaje de programación Java, con el rango de valores que pueden representar.*

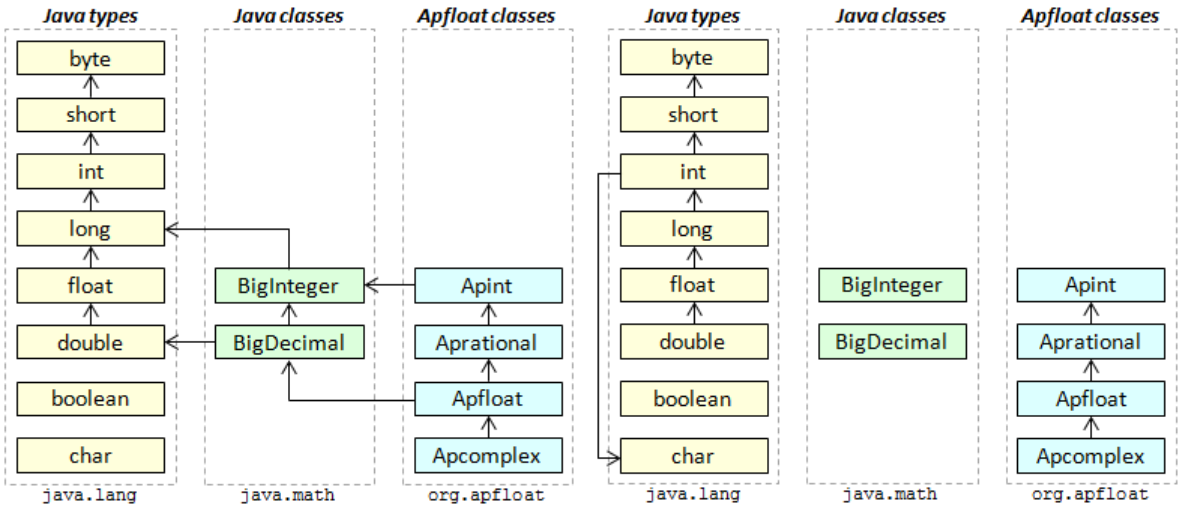
Tipo primitivo	Bits	Rango de valores	Descripción	*Wrapper*
`boolean`	8	*true*, *false*	Valores booleanos	`java.lang.Boolean`
`char`	16	Estándar *Unicode*	Caracteres	`java.lang.Character`
`byte`	8	$-2^7 \leq x \leq 2^7 - 1$	Valores enteros de 8 bits	`java.lang.Byte`
`short`	16	$-2^{15} \leq x \leq 2^{15} - 1$	Valores enteros de 16 bits	`java.lang.Short`
`int`	32	$-2^{31} \leq x \leq 2^{31} - 1$	Valores enteros de 32 bits	`java.lang.Integer`
`long`	64	$-2^{63} \leq x \leq 2^{63} - 1$	Valores enteros de 64 bits	`java.lang.Long`
`float`	32	$1.40 \cdot 10^{-45} \leq \lvert x \rvert \leq 3.40 \cdot 10^{38}$	Valores flotantes de 32 bits	`java.lang.Float`
`double`	64	$4.94 \cdot 10^{-324} \leq \lvert x \rvert \leq 1.80 \cdot 10^{308}$	Valores flotantes de 64 bits	`java.lang.Double`

Tabla 8.10. *Tipos primitivos de datos particulares a GOLD 3.*

Tipo primitivo	Identificador	Implementación
Booleanos	$\mathbb{B}$	`java.lang.Boolean`
Naturales	$\mathbb{N}$	`org.apfloat.Apint`
Enteros	$\mathbb{Z}$	`org.apfloat.Apint`
Racionales	$\mathbb{Q}$	`org.apfloat.Aprational`
Reales	$\mathbb{R}$	`org.apfloat.Apfloat`
Complejos	$\mathbb{C}$	`org.apfloat.Apcomplex`

Figura 8.46. *Tipos numéricos del API de Java y de Apfloat.*

(a) *Jerarquía de clases numéricas en GOLD.* **(b)** *Jerarquía de clases numéricas en Java/Apfloat.*



Toda operación aritmética que involucre dos valores numéricos a y b que correspondan a un tipo primitivo de *Java* o una clase que represente un número de precisión arbitraria de *Java* o de *Apfloat* se realiza respetando la jerarquía de clases numéricas presentada en el esquema 8.46(a):

1. Sea T el tipo asociado al valor a y U el tipo asociado al valor b.

2. Sea V el *ancestro común más cercano* (*nearest common ancestor*) [†11] a los tipos T y U en el grafo dirigido acíclico descrito en la ilustración 8.46(a). Si tal ancestro no existe, entonces se lanza un error de ejecución indicando que la operación no se puede efectuar debido a una incongruencia de tipos.

3. Se convierten los valores a y b al tipo V mediante *casts* especializados, y luego se efectúa la operación entre éstos dando un resultado de tipo V.

El mecanismo descrito permite realizar operaciones aritméticas entre valores numéricos de distinto tipo sin tener que convertir tipos (*cast*) explícitamente, lo que facilita la programación de rutinas que usan intensivamente números de precisión arbitraria. De esta manera sería posible operar números entre sí sin preocuparse por su tipo, ya sean valores que correspondan con un tipo primitivo de *Java* (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float`, `double`) o instancias que correspondan con una clase que represente números de precisión arbitraria (`BigInteger`, `BigDecimal`, `Apint`, `Aprational`, `Apfloat`, `Apcomplex`).

Por ejemplo, para sumar un número `a` de tipo `BigDecimal` con un número `b` de tipo `BigInteger` en *Java* se debería escribir la expresión `a.add(new BigDecimal(b))`, mientras que en *GOLD* se podría escribir $a+b$. Asimismo, expresiones *Java* del estilo `z.divide(y).multiply(x.multiply(y.subtract(x.multiply(BigInteger.valueOf(2)))))` (donde `x`, `y` y `z` son de tipo `BigInteger`) pueden ser escritas en *GOLD* en la forma $(z \div y) \cdot x \cdot (y - x \cdot 2)$, que evidentemente es una expresión más cómoda de escribir.

Al momento de realizar operaciones aritméticas, *Java* emplea un mecanismo similar de conversión de tipos (*casting*), pero aplicado únicamente sobre sus ocho tipos primitivos de datos como se describe en el esquema 8.46(b). Extendiendo el mecanismo como se ilustra en 8.46(a) se fomenta la utilización de números de precisión arbitraria en programas escritos en el lenguaje *GOLD*.

8.3.3. Apariencia del entorno gráfico (*Look and Feel*)

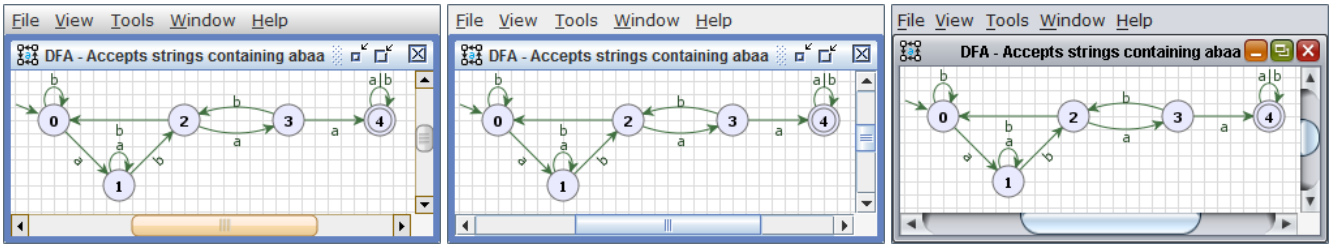
El paquete `gold.swing.look` provee un conjunto de clases diseñadas bajo la arquitectura *Swing* [80] que implementan el aspecto y comportamiento (*Look and Feel*) de los componentes gráficos de la interfaz de usuario (*GUI*: *Graphical User Interface*) que usan los visualizadores de estructuras de datos implementados en el paquete `gold.visualization`. El *Look and Feel* de *GOLD* está basado en el de *Java* (denominado *Metal*) y se instala por defecto cuando el usuario ejecuta el procedimiento propio *main* de alguna aplicación *GOLD*. Además, puede instalarse fácilmente en cualquier aplicación *Java* invocando el método `installGoldLookAndFeel` de la clase `gold.swing.util.GUtilities`.

La apariencia provista por *GOLD* emplea una gama de colores consistente con la temática de usar objetos y tonalidades relacionadas con el oro (*gold* en inglés). Sin embargo, si el usuario no desea usar esta apariencia entonces podría instalar cualquier otra, por ejemplo algún *Look and Feel* suministrado por la librería *Swing* [80] de *Java*.

[11] Dado un grafo dirigido acíclico $G = \langle V, E \rangle$, el *ancestro común más cercano* (*nearest common ancestor*) de dos vértices $a \in V$ y $b \in V$ es el vértice $c \in V$ que es ancestro tanto de a como de b y que minimiza la expresión $max(dist(c,a), dist(c,b))$ donde $dist(v_1, v_2)$ es el costo de la ruta más corta de v_1 a v_2 en G suponiendo que todos sus arcos tienen costo 1. Si el vértice c no existe o no es único, se dice que a y b no tienen ancestro común más cercano. Esta definición extiende el concepto de *ancestro común más cercano* (*least common ancestor*) en árboles con raíz (*rooted trees*) estudiado en el texto *Introduction to Algorithms* de Thomas Cormen et al. [1].

Figura 8.47. *Distintos aspectos (Look and Feel) para las interfaces gráficas en GOLD 3.*

(a) *GOLD Look and Feel.* **(b)** *Metal Look and Feel.* **(c)** *Nimbus Look and Feel.*



(d) *CDE/Motif Look and Feel.* **(e)** *Windows Look and Feel.* **(f)** *Mac OS Look and Feel.*

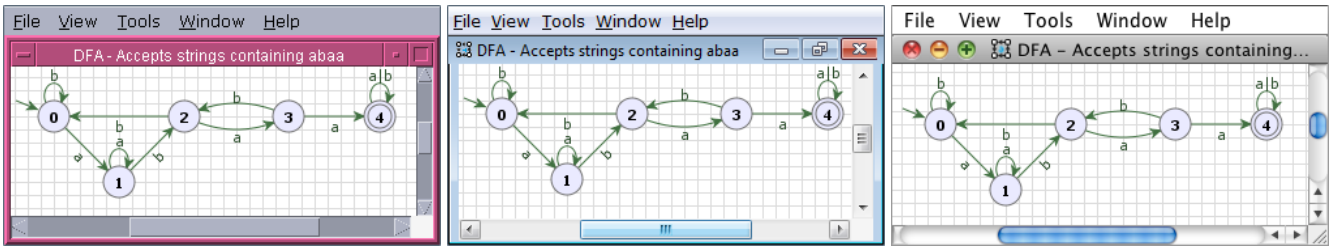
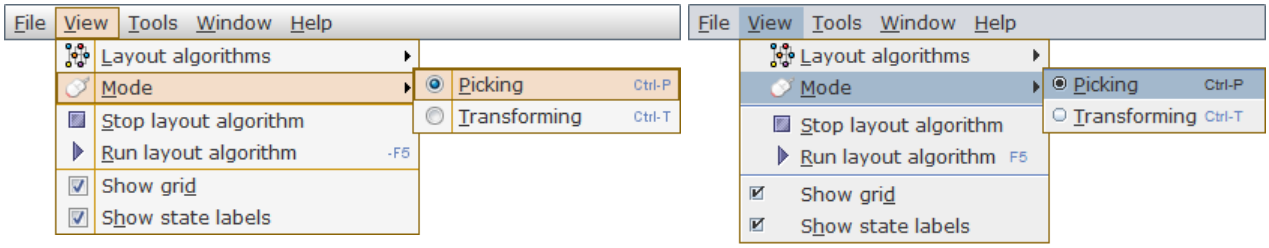
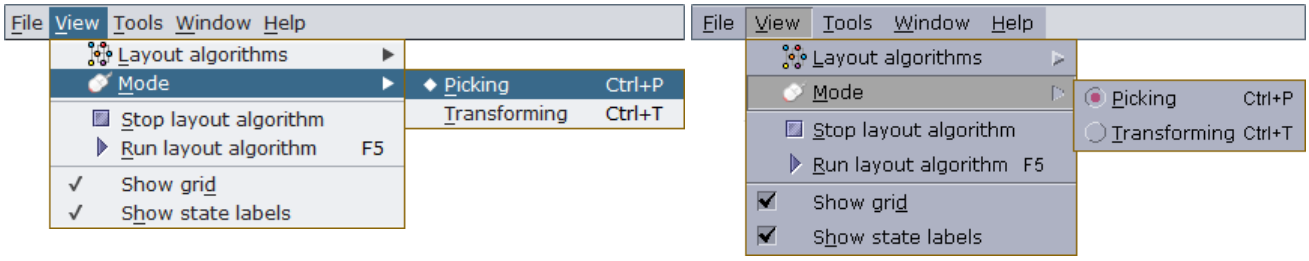


Figura 8.48. *Diferencias de apariencia en GOLD 3 entre los distintos Look and Feel.*

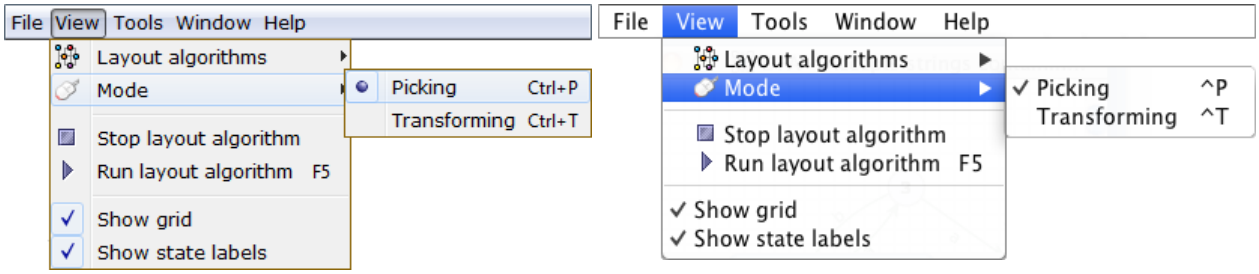
(a) *GOLD Look and Feel.* **(b)** *Metal Look and Feel.*



(c) *Nimbus Look and Feel.* **(d)** *CDE/Motif Look and Feel.*



(e) *Windows Look and Feel.* **(f)** *Mac OS Look and Feel.*



8.3.4. Visualizadores especializados

El paquete `gold.visualization` contiene clases especializadas en visualizar estructuras de datos como árboles, grafos y autómatas, a través de componentes gráficos que usan intensivamente la librería *JUNG* [21] (*Java Universal Network/Graph Framework*), que ofrece un potente visualizador altamente configurable que permite cambiar el aspecto de los nodos y de los arcos, así como el algoritmo utilizado para ubicar los nodos (*layout algorithm*) dentro de la superficie de dibujo. *GOLD* provee un conjunto simplificado de funciones para alterar la apariencia de los grafos en tiempo de ejecución, facilitando la depuración y animación paso a paso de la operación de cualquier algoritmo sobre grafos de forma interactiva. Además, *GOLD* suministra adaptadores para convertir grafos representados en *GOLD* en grafos representados con las estructuras de datos provistas por *JUNG*, y viceversa.

Figura 8.49. *Visualizador de grafos de GOLD 3, desplegando un Quadtree con 16 píxeles.*

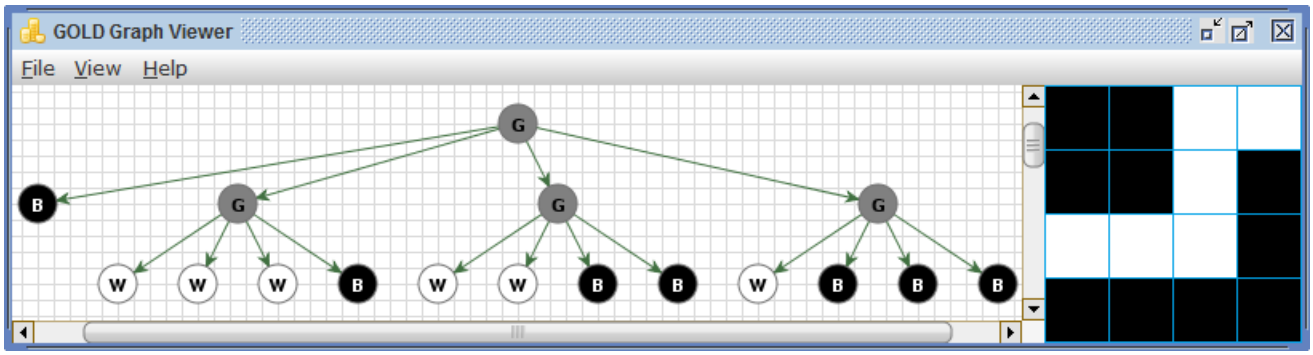
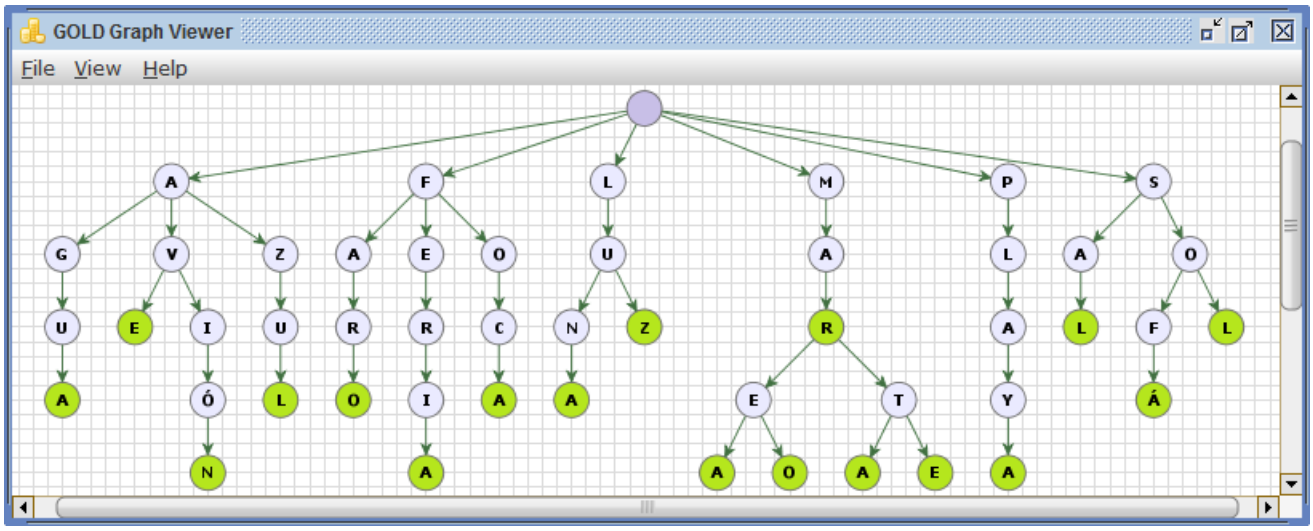


Figura 8.50. *Visualizador de grafos de GOLD 3, desplegando un Trie con 18 palabras.*



Por medio de la librería *JUNG* se puede cambiar por completo la apariencia visual de los grafos, configurando propiedades como las siguientes:

- *Vértices*. Etiqueta de texto (*label*), forma geométrica (*shape*), color de borde (*border color*), estilo de borde (*border stroke*), color de relleno (*background color*), descripción emergente (*tooltip text*), tipo de letra (*font*).
- *Arcos*. Etiqueta de texto (*label*), forma geométrica (*shape*), estilo de trazo (*stroke*), color de trazo (*color*), tipo de letra (*font*), posición de la etiqueta con respecto a la forma geométrica (*label position*).

- *Flechas*. Forma geométrica (*shape*), color de borde (*border color*), estilo de borde (*border stroke*), color de relleno (*background color*).

Figura 8.51. *Visualizador de autómatas de GOLD 3, desplegando un grafo aleatorio con 40 nodos.*

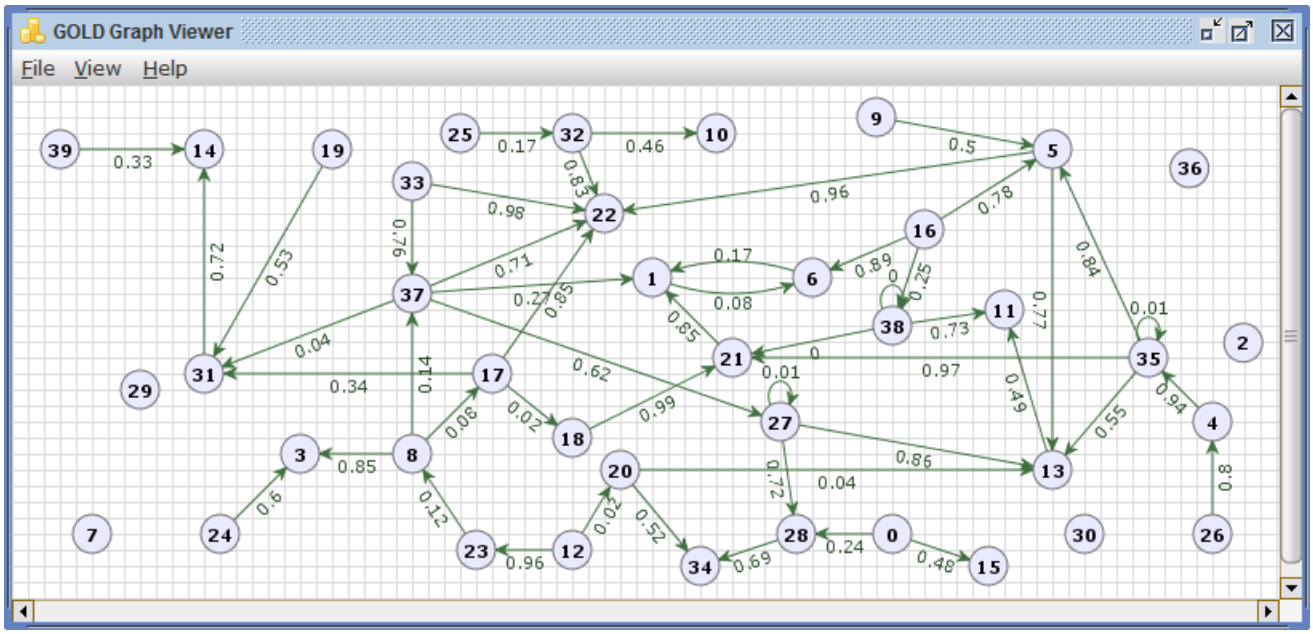
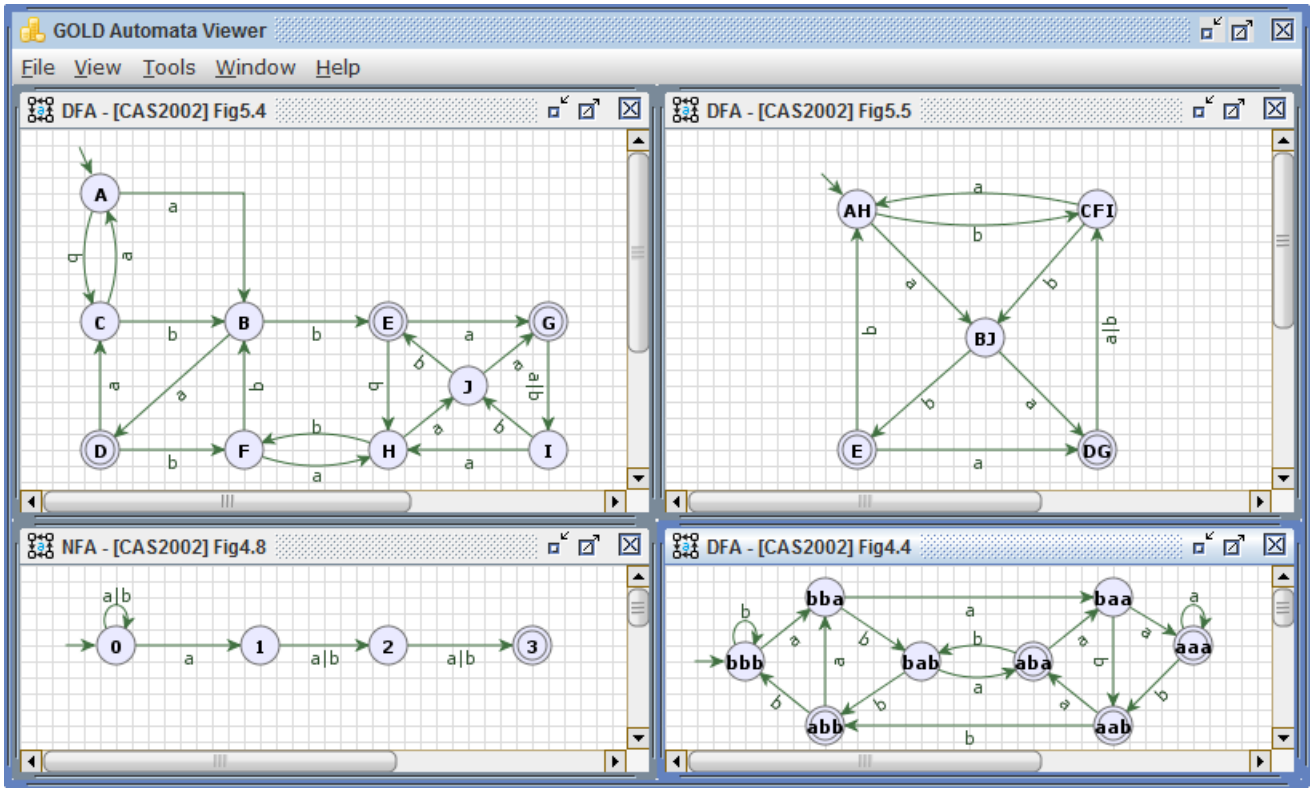


Figura 8.52. *Visualizador de autómatas de GOLD 3, desplegando cuatro ejemplos del libro de Rafel Cases [39].*



Parte III

Resultados

Capítulo 9

Ejemplos

En este capítulo se recopilan algunos programas de ejemplo que se codificaron en *GOLD 3*, para ilustrar el uso de las distintas instrucciones del lenguaje, así como su librería de clases. Aunque la lista de ejemplos no pretende ser exhaustiva, ésta cubre una gran cantidad de técnicas de programación estudiadas en la literatura. Todos los ejemplos se encuentran disponibles en proyectos *Eclipse* bajo el directorio `/Data/Tests/`.

9.1. Matemáticas discretas

9.1.1. Funciones básicas

9.1.1.1. Función de Fibonacci

La *función de Fibonacci* se define recursivamente de la siguiente manera, donde $n \in \mathbb{N}$:

$$fib(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fib(n-2) + fib(n-1) & \text{si } n \geq 2 \end{cases}$$

Para implementar la función de Fibonacci basta definir un procedimiento recursivo (véase el código 9.1). Sin embargo, una implementación ingenua conlleva problemas de eficiencia y de desbordamiento del tipo de datos.

Código 9.1. *Función de Fibonacci implementada recursivamente, usando el tipo de datos* `int`.

```
1 function fib(n) begin
2   if n=0 then
3     return 0
4   elseif n=1 then
5     return 1
6   else
7     return fib(n-1)+fib(n-2)
8   end
9 end
```

La función implementada en el código 9.1 tiene complejidad temporal $O(\varphi^n)$ (donde $\varphi = (1+\sqrt{5})/2$ es el número de oro) y no da resultados correctos para $n \geq 47$, pues el tipo de datos `int` se desborda. Simplificar la función a través de una macro (véase el código 9.2) no soluciona ninguno de los dos problemas.

Código 9.2. *Función de Fibonacci implementada recursivamente como una macro, usando el tipo de datos* `int`.

```
1 fib(n):= n=0?0:(n=1?1:fib(n-1)+fib(n-2))
```

Para solucionar el problema de eficiencia se puede implementar la función de Fibonacci a través de un procedimiento iterativo que evita el uso de variables temporales usando asignaciones simultáneas (véase el código 9.3).

Código 9.3. *Función de Fibonacci implementada iterativamente, usando el tipo de datos* `int`.

```
function fib(n) begin
  a,b := 0,1
  for i := 1 to n do
    a,b := b,a+b
  end
  return a
end
```

La función implementada en el código 9.3 tiene complejidad temporal $O(n)$, pero sigue teniendo los mismos problemas de desbordamiento. Para corregir esto, es suficiente usar números de precisión arbitraria a través del tipo de datos $\mathbb{Z}$ (véase el código 9.4), que es implementado con la clase `org.apfloat.Apint` de la librería *Apfloat*. La sentencia $\mathbb{Z}(0)$ abrevia la expresión `org.apfloat.Apint`(0), que a su vez abrevia la invocación de constructor de clase `new org.apfloat.Apint`(0).

Código 9.4. *Función de Fibonacci implementada iterativamente, usando números de precisión arbitraria.*

```
function fib(n) begin
  a,b := ℤ(0),ℤ(1)
  for i := 1 to n do
    a,b := b,a+b
  end
  return a
end
```

Para mejorar la legibilidad, las variables a y b se pueden declarar explícitamente como de tipo $\mathbb{Z}$ (véase el código 9.5), que es un sinónimo de la clase `org.apfloat.Apint` de la librería *Apfloat*. Una vez declaradas las dos variables, cualquier asignación que se realice sobre éstas será sometida a una conversión de tipos (por ejemplo, en la asignación $a,b := 0,1$, los valores 0 y 1 de tipo `int` son convertidos automáticamente al tipo `Apint` antes de ser asignados).

Código 9.5. *Función de Fibonacci implementada iterativamente, usando números de precisión arbitraria y declarando variables explícitamente.*

```
function fib(n) begin
  var a:ℤ,b:ℤ
  a,b := 0,1
  for i := 1 to n do
    a,b := b,a+b
  end
  return a
end
```

Es posible mejorar la complejidad temporal de la función, calculándola a través de la fórmula

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n+1} = \begin{pmatrix} fib(n+2) & fib(n+1) \\ fib(n+1) & fib(n) \end{pmatrix}$$

Dada una matriz cuadrada A de tamaño $m \times m$ (con $m \geq 2$) y un número natural n, el siguiente hecho permite calcular $A^n = \underbrace{A \cdot A \cdot \ldots \cdot A}_{\text{n veces}}$ en tiempo logarítmico, usando el principio de *Dividir y Conquistar*:

$$A^n = \begin{cases} I_m & \text{si } n = 0 \text{ (donde } I_m \text{ es la matriz identidad de tamaño } m \times m) \\ A & \text{si } n = 1 \\ \left(A^k\right)^2 & \text{si } n = 2 \cdot k \text{ para algún } k \in \mathbb{N} \text{ (es decir, si } n \text{ es par)} \\ A \cdot \left(A^k\right)^2 & \text{si } n = 2 \cdot k + 1 \text{ para algún } k \in \mathbb{N} \text{ (es decir, si } n \text{ es impar)} \end{cases}$$

Aplicando lo anterior se puede calcular el Fibonacci de n mediante una función con complejidad temporal $O(\log_2 n)$ (véase el código 9.6), extrayendo el valor ubicado en la segunda fila y segunda columna de la matriz

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n+1}$$

Código 9.6. *Función de Fibonacci implementada en tiempo logarítmico, usando números de precisión arbitraria.*

```
function matrixMul(A,B) begin // 2×2 matrix multiplication A*B
  r00 := A[0][0]·B[0][0]+A[0][1]·B[1][0]
  r01 := A[0][0]·B[0][1]+A[0][1]·B[1][1]
  r10 := A[1][0]·B[0][0]+A[1][1]·B[1][0]
  r11 := A[1][0]·B[0][1]+A[1][1]·B[1][1]
  return [[r00,r01],[r10,r11]]
end
function matrixPow(A,n) begin // 2×2 matrix exponentiation C^n
  if n=0 then
    return [[ℤ(1),ℤ(0)],[ℤ(0),ℤ(1)]] // 2×2 identity
  elseif n=1 then
    return A
  elseif n%2=0 then
    D := matrixPow(A,n÷2)
    return matrixMul(D,D)
  else
    D := matrixPow(A,n÷2)
    return matrixMul(matrixMul(D,D),A)
  end
end
function fib(n) begin
  R := matrixPow([[ℤ(1),ℤ(1)],[ℤ(1),ℤ(0)]],n+1)
  return R[1][1]
end
```

Para probar cualquiera de las versiones presentadas basta invocar la función (véase el código 9.7).

Código 9.7. *Procedimiento que ilustra el uso de la función de Fibonacci.*

```
procedure main(args:String[]) begin
  for i := 0 to 200 do
    print "fib(",i,")=",fib(i)
  end
end
```

En adelante se usarán números de precisión arbitraria en los ejemplos, de ser necesario. Por ejemplo, la expresión $(1+\texttt{Math.\$sqrt}(5))/2$ calcula el número de oro como un valor de tipo `double`, la expresión $(1+\texttt{sqrt}(5))/2$ es una abreviación de la anterior, la expresión $(1+\texttt{sqrt}(\mathbb{R}(5)))/2$ calcula el número de oro como un valor de tipo `Apfloat` de 128 decimales (por defecto), y la expresión $(1+\texttt{sqrt}(\mathbb{R}(5,500)))/2$ calcula el número de oro como un valor de tipo `Apfloat` de 500 decimales (configurados como un argumento pasado al método constructor $\mathbb{R}(\ldots)$).

9.1.1.2. Factorial

La función *factorial* se define recursivamente de la siguiente manera, donde $n \in \mathbb{N}$:

$$fact(n) = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot fact(n-1) & \text{si } n \geq 1 \end{cases}$$

Se puede implementar una versión recursiva con complejidad temporal $O(n)$ aplicando la definición textualmente (véase el código 9.8).

Código 9.8. *Factorial implementado recursivamente.*

```
function fact(n) begin
  if n=0 then
    return ℤ(1)
  else
    return n*fact(n-1)
  end
end
```

La implementación se puede acortar (véase el código 9.9) usando expresiones condicionales (operador `?`).

Código 9.9. *Factorial implementado recursivamente con expresiones condicionales.*

```
function fact(n) begin
  return n=0?ℤ(1):n*fact(n-1)
end
```

Adicionalmente, para acortar más el código se puede definir la función como una macro (véase el código 9.10).

Código 9.10. *Factorial implementado recursivamente como una macro.*

```
fact(n):= n=0?ℤ(1):n*fact(n-1)
```

Aunque es fácil implementar la función iterativamente, resulta más interesante definirla en términos de una multiplicatoria de números de precisión arbitraria implementados con la clase `Apint` (véase el código 9.11).

Código 9.11. *Factorial implementado mediante una cuantificación.*

```
fact(n):= (Πi|1≤i≤n:ℤ(i))
```

Para probar cualquiera de las versiones presentadas basta invocar la función (véase el código 9.12). Observe la presencia de la instrucción `assert` para validar en tiempo de ejecución que los resultados sean los correctos.

Código 9.12. *Procedimiento que ilustra el uso de la función factorial.*

```
procedure main(args:String[]) begin
  for i:=0 to 200 do
    print i+"!="+fact(i)
    assert fact(i)=i!
  end
end
```

9.1.1.3. Binomial

El *coeficiente binomial* se define recursivamente de la siguiente manera, donde $n \in \mathbb{N}$, $k \in \mathbb{N}$, y $0 \leq k \leq n$:

$$binomial(n,k) = \begin{cases} 1 & \text{si } k=0 \text{ o } k=n \\ binomial(n-1,k) + binomial(n-1,k-1) & \text{si } 0<k<n \end{cases}$$

Se puede implementar una versión recursiva aplicando la definición textualmente (véase el código 9.13).

Código 9.13. *Binomial implementado recursivamente.*

```
function binomial(n,k) begin
```

```
2   if k=0 ∨ k=n then
3     return ℤ(1)
4   else
5     return binomial(n-1,k)+binomial(n-1,k-1)
6   end
7 end
```

Sabiendo que $binomial(n,k) = n!/(k!\cdot(n-k)!)$, se implementa la función como una macro (véase el código 9.14).

Código 9.14. *Binomial implementado como una macro, usando factoriales.*

```
1 binomial(n,k):=n!/(k!·(n-k)!)
```

Para mejorar la eficiencia del procedimiento, logrando una complejidad temporal de $O(n\downarrow n-k)$ (véase el código 9.15), se puede aplicar la siguiente fórmula:

$$binomial(n,k) = \begin{cases} 1 & \text{si } k=0 \text{ o } k=n \\ binomial(n,n-k) & \text{si } n-k<k \\ binomial(n,k-1)\cdot\frac{n-k+1}{k} & \text{de lo contrario} \end{cases}$$

Código 9.15. *Binomial implementado recursivamente, con complejidad lineal en sus parámetros.*

```
1 function binomial(n,k) begin
2   if k=0 or k=n then
3     return ℕ(1)
4   elseif n-k<k then
5     return binomial(n,n-k)
6   else
7     return binomial(n,k-1)·(n-k+1)÷k
8   end
9 end
```

La anterior versión se puede traducir rápidamente en un procedimiento iterativo (véase el código 9.16).

Código 9.16. *Binomial implementado iterativamente, con complejidad lineal en sus parámetros.*

```
1 function binomial(n,k) begin
2   var r:ℕ
3   r:=1
4   for i:=1 to n-k↓k do
5     r:=r*(n-i+1)/i
6   end
7   return r
8 end
```

Para probar cualquiera de las versiones presentadas basta invocar la función (véase el código 9.17). Observe la presencia de la instrucción `assert` para validar en tiempo de ejecución que los resultados sean los correctos.

Código 9.17. *Procedimiento que ilustra el uso de la función binomial.*

```
1 procedure main(args:String[]) begin
2   for n:=0 to 10 do
3     for k:=0 to n do
4       print "binomial("+n+","+k+")="+binomial(n,k)
5       assert binomial(n,k)=n!/(k!·(n-k)!)
6     end
7   end
8 end
```

9.1.2. Teoría de números

9.1.2.1. Algoritmo de Euclides

El algoritmo de Euclides permite calcular el máximo común divisor (*gcd*: *greatest common divisor*) recursivamente a través de la fórmula

$$gcd(a,b) = \begin{cases} a & \text{si } b=0 \\ gcd(b, a \bmod b) & \text{si } b \neq 0 \end{cases}$$

Dado que `gcd` es una palabra reservada de *GOLD* que representa la función que calcula el máximo común divisor, para implementar cualquier procedimiento externo que la implemente se debe usar otro nombre (e.g., `GCD`) o escaparlo con el signo pesos (e.g., `$gcd`).

Código 9.18. *Algoritmo de Euclides implementado recursivamente.*

```
function GCD(a,b) begin
  if b=0 then
    return a
  else
    return GCD(b,a%b)
  end
end
```

Código 9.19. *Algoritmo de Euclides implementado recursivamente como una macro.*

```
GCD(a,b):= b=0?a:GCD(b,a%b)
```

Código 9.20. *Algoritmo de Euclides implementado iterativamente.*

```
function GCD(a,b) begin
  while b≠0 do
    a,b:= b,a%b
  end
  return a
end
```

Código 9.21. *Procedimiento para probar el algoritmo de Euclides.*

```
procedure test_gcd(a,b) begin
  print "gcd("+a+","+b+")="+GCD(a,b)
  assert GCD(a,b)=gcd(a,b)
end
procedure main(args:String[]) begin
  for i:= 0 to 100 do
    for j:= 0 to 100 do
      test_gcd(i,j)
    end
  end
  test_gcd(21,49)
  test_gcd(51552512224353291L,18423383840456761L)
  test_gcd(ℤ("74033440182726064189114093372 2"),ℤ("4700937512298234422957095936 86"))
end
```

9.1.2.2. Algoritmo extendido de Euclides

Dados dos números enteros a y b, el algoritmo extendido de Euclides permite encontrar dos números enteros x y y tales que $gcd(a,b) = x \cdot a + y \cdot b$.

Código 9.22. *Algoritmo extendido de Euclides implementado iterativamente.*

```
// Extended Euclidean algorithm (taken from Wikipedia)
function extended_gcd(a,b) begin
  x,lastx := 0,1
  y,lasty := 1,0
  while b≠0 do
    q := a÷b
    a,b := b,a%b
    x,lastx := lastx-q·x,x
    y,lasty := lasty-q·y,y
  end
  return ⟨lastx,lasty⟩
end
```

Código 9.23. *Procedimiento para probar el algoritmo extendido de Euclides.*

```
procedure test_gcd(a,b) begin
  r := extended_gcd(a,b)
  x,y := r[0],r[1]
  print "gcd(",a,",",b,")=",x·a+y·b,"=",x,"*",a,"+",y,"*",b
  assert x·a+y·b=gcd(a,b)
end
procedure main(args:String[]) begin
  for i := 0 to 100 do
    for j := 0 to 100 do
      test_gcd(i,j)
    end
  end
  test_gcd(21,49)
  test_gcd(51552512224353291L,18423383840456761L)
  test_gcd(ℤ("74033440182726064189114093372 2"),ℤ("4700937512298234422957095936 86"))
end
```

Código 9.24. *Fragmento de la salida por consola del programa 9.23.*

```
...
gcd(64,35)=1=-6*64+11*35
gcd(64,36)=4=4*64+-7*36
gcd(64,37)=1=11*64+-19*37
gcd(64,38)=2=3*64+-5*38
gcd(64,39)=1=-14*64+23*39
gcd(64,40)=8=2*64+-3*40
gcd(64,41)=1=-16*64+25*41
gcd(64,42)=2=2*64+-3*42
gcd(64,43)=1=-2*64+3*43
gcd(64,44)=4=-2*64+3*44
gcd(64,45)=1=19*64+-27*45
...
gcd(100,98)=2=1*100+-1*98
gcd(100,99)=1=1*100+-1*99
gcd(100,100)=100=0*100+1*100
...
```

9.1.2.3. Teorema chino del residuo

El *teorema chino del residuo* permite resolver el sistema de congruencias simultáneas $x \equiv a_i \ (mod\ n_i)$ para cada $0 \leq i < k$ (donde los n_i son primos relativos dos a dos), encontrando una solución x módulo $n_0 \cdot n_1 \cdot \ldots \cdot n_{k-1}$. Para implementar el procedimiento se necesita el algoritmo extendido de Euclides estudiado en la sección §9.1.2.2.

Código 9.25. *Teorema chino del residuo implementado iterativamente.*

```
1  // Solves the system x≡a[i] (mod n[i]) for 0≤i≤k-1
2  function chinese(a,n) begin
3    assert |a|=|n| ∧ (∀i,j|0≤i<|a|,i<j<|a|:gcd(n[i],n[j])=1)
4    x,k,N := 0,|a|,(Πy|y∈n:y)
5    matrix := ⟨extended_gcd(n[i],N/n[i])|0≤i<k⟩
6    x := (Σi|0≤i<k:a[i]·matrix[i][1]·N/n[i])
7    assert (∀i|0≤i<k : x mod n[i] = a[i] mod n[i])
8    return x mod N
9  end
```

Observe cómo en el código 9.25 se está validando en tiempo de ejecución la precondición y la postcondición del algoritmo a través de la instrucción `assert`. Como precondición se exige que los valores $n_0, n_1, \ldots, n_{k-1}$ sean primos relativos dos a dos, y como postcondición se exige que el valor x sea solución del sistema de congruencias.

Código 9.26. *Procedimiento para probar el teorema chino del residuo.*

```
1  procedure test_chinese(a,n) begin
2    k,x := |a|,chinese(a,n)
3    for i := 0 to k-1 do
4      print x+" == "+a[i]+" (mod "+n[i]+")"
5    end
6    print "Solution: "+x+" (mod ",(Πy|y∈n:y)+")"
7    print String(char[32]).replace(0c,'-')
8  end
9  procedure main(args:String[]) begin
10   test_chinese([[0,0,0]],[[3,4,5]])
11   test_chinese([[2,3,1]],[[3,4,5]])
12   test_chinese([[71,27,91,20]],[[14,33,17,65]])
13 end
```

Código 9.27. *Fragmento de la salida por consola del programa 9.26.*

```
0 == 0 (mod 3)
0 == 0 (mod 4)
0 == 0 (mod 5)
Solution: 0 (mod 60)
--------------------------------
11 == 2 (mod 3)
11 == 3 (mod 4)
11 == 1 (mod 5)
Solution: 11 (mod 60)
--------------------------------
351345 == 71 (mod 14)
351345 == 27 (mod 33)
351345 == 91 (mod 17)
351345 == 20 (mod 65)
Solution: 351345 (mod 510510)
--------------------------------
```

9.1.2.4. Función indicatriz de Euler

Dado un número natural $n \geq 2$, la función indicatriz de Euler (*Euler's totient function*) puede ser definida así:

$$\varphi(n) = |\{i \in \mathbb{N} |\ 1 \leq i \leq n \wedge gcd(n,i) = 1\}|$$

Código 9.28. *Función indicatriz de Euler implementada usando cardinalidad de conjuntos.*

```
1 φ(n):=|{i|1≤i≤n,[gcd(n,i)=1]}|
```

Código 9.29. *Función indicatriz de Euler implementada usando sumatorias.*

```
1 φ(n):=(Σi|1≤i≤n,[gcd(n,i)=1]:1)
```

Código 9.30. *Función indicatriz de Euler implementada iterativamente (primera versión).*

```
1  function φ(n) begin // Euler's totient function
2    var r:ℕ,p:ℕ
3    r,p:=1,2
4    while p^2≤n do
5      k:=0
6      while n%p=0 do
7        n,k:=n÷p,k+1
8      end
9      if k>0 then
10       r:=r·p^(k-1)·(p-1)
11     end
12     p:=p+1
13   end
14   if n>1 then
15     r:=r·(n-1)
16   end
17   return r
18 end
```

Código 9.31. *Función indicatriz de Euler implementada iterativamente (segunda versión).*

```
1  function φ(n) begin // Euler's totient function
2    var r:ℕ,p:ℕ
3    r,p:=n,2
4    while p^2≤n do
5      if n%p=0 then
6        repeat
7          n:=n÷p
8        until n%p≠0
9        r:=r·(p-1)/p
10     end
11     p:=p+1
12   end
13   if n>1 then
14     r:=r·(n-1)/n
15   end
16   return r
17 end
```

Código 9.32. *Procedimiento para probar la función indicatriz de Euler.*

```
procedure main(args:String[]) begin
  for i:=2 to 200 do
    print "phi(",i,")=",φ(i)
    assert φ(i)=(Σk|1≤k<i,[gcd(i,k)=1]:1)
  end
end
```

9.1.2.5. Números primos

Un número natural n es primo si $n \geq 2$ y sus únicos divisores positivos son 1 y n. Se puede demostrar que todo número $n \geq 2$ es primo si y sólo si no tiene divisores entre 2 y $\sqrt{n}$. Para encontrar los números primos entre 2 y n se puede usar la definición, aplicándola para todo número i tal que $2 \leq i \leq n$ (en los ejemplos mostrados, $j|i$ abrevia la expresión $i \ mod \ j = 0$ usando el operador de divisibilidad ($|$)).

Código 9.33. *Función para encontrar los primos desde 2 hasta n, usando el operador de divisibilidad ($|$).*

```
function primes(n) begin
  return {i|2≤i≤n,[¬(∃j|2≤j≤sqrt(i):j|i)]}
end
```

Código 9.34. *Macro para encontrar los primos desde 2 hasta n, usando el operador de divisibilidad ($|$).*

```
primes(n):={i|2≤i≤n,[¬(∃j|2≤j≤sqrt(i):j|i)]}
```

Código 9.35. *Macro para encontrar los primos desde 2 hasta n, usando el operador de anti-divisibilidad ($\nmid$).*

```
primes(n):={i|2≤i≤n,[(∀j|2≤j≤sqrt(i):j∤i)]}
```

Código 9.36. *Macro para encontrar los primos desde 2 hasta n, usando el operador módulo.*

```
primes(n):={i|2≤i≤n,[(∀j|2≤j≤sqrt(i):i%j≠0)]}
```

Código 9.37. *Criba de Eratóstenes para encontrar los primos desde 2 hasta n.*

```
function primes(n) begin   // Sieve of Eratosthenes
  var b:boolean[n+1]
  for i:=2 to n do
    b[i]:=TRUE
  end
  for i:=2 to sqrt(n) do
    if b[i] then
      for j:=i*i to n by i do
        b[j]:=FALSE
      end
    end
  end
  return {i|2≤i≤n,[b[i]]}
end
```

Un número primo de Mersenne es un primo de la forma $2^p - 1$, donde p es un número primo (necesariamente p debe ser primo porque de lo contrario, $2^p - 1$ no sería primo). Para encontrar números primos de Mersenne existe un algoritmo eficiente denominado *test de Lucas-Lehmer* (véase el código 9.38).

Código 9.38. *Programa que usa el test de Lucas-Lehmer para encontrar números primos de Mersenne.*

```
1  primes(n):={i|2≤i≤n,[¬(∃j|2≤j≤sqrt(i):i%j=0)]}
2  function mersenne(p) begin   //Lucas-Lehmer test
3    s,Mp:=ℤ(4),ℤ(2)^p-1
4    for i:=1 to p-2 do
5      s:=(s^2-2) mod Mp
6    end
7    return s=0
8  end
9  procedure main(args:String[]) begin
10   time:=System.currentTimeMillis()
11   n:=2000
12   P:=primes(n)
13   for i:=2 to n do
14     if i∈P and mersenne(i) then
15       print "2^"+i+" is prime"
16       print "  Ellapsed time: ",(System.currentTimeMillis()-time),"ms"
17     end
18   end
19 end
```

9.2. Análisis numérico

9.2.1. Cálculo

9.2.1.1. Cálculo integral

Para calcular integrales numéricamente existen métodos como la regla de Simpson y las sumas de Riemann. Por un lado, la regla de Simpson establece que

$$\int_a^b f(x)\mathrm{d}x \approx \frac{b-a}{6}\cdot\left(f(a)+4\cdot f\left(\frac{a+b}{2}\right)+f(b)\right)$$

Por otro lado, las sumas de Riemann sugieren que (dado un natural n arbitrariamente grande)

$$\int_a^b f(x)\mathrm{d}x \approx \sum_{i=0}^{n} f\left(a+\frac{(b-a)\cdot i}{n}\right)\cdot\frac{b-a}{n}$$

Código 9.39. *Cálculo numérico de integrales con la regla de Simpson y sumas de Riemann.*

```
f(x) := ln(x)
g(x) := exp(x)+sin(x)
h(x) := g(f(x))
f₁(x) := x*(ln(x)-1)
g₁(x) := exp(x)-cos(x)
h₁(x) := (1/2)*x*(x+sin(ln(x))-cos(ln(x)))
simpson(ξ:IMethod,a,b) := (b-a)/6*(ξ(a)+4*ξ((a+b)/2)+ξ(b))
riemann(ξ:IMethod,a,b,n) := (Σi|0≤i≤n:ξ(a+(b-a)*i/n)*(b-a)/n)
riemann(ξ:IMethod,a,b) := riemann(ξ,a,b,1000)
procedure main(args:String[]) begin
  print "f"
  print "Simpson : "+simpson(f,2,5)
  print "Riemann : "+riemann(f,2,5)
  print "Integral: "+(f₁(5)-f₁(2))
  print "g"
  print "Simpson : "+simpson(g,2,5)
  print "Riemann : "+riemann(g,2,5)
  print "Integral: "+(g₁(5)-g₁(2))
  print "h"
  print "Simpson : "+simpson(h,2,5)
  print "Riemann : "+riemann(h,2,5)
  print "Integral: "+(h₁(5)-h₁(2))
end
```

Código 9.40. *Método de integración numérica por sumas de Riemman, implementado iterativamente.*

```
function riemann(ξ:IMethod,a,b) begin
  n := 1000
  s := 0
  for i := 0 to n do
    x := a+(b-a)*i/n
    s := s+ξ(x)*(b-a)/n
  end
  return s
end
```

Código 9.41. *Variante para el método main del programa 9.39, manipulando funciones como valores.*

```
procedure main(args:String[]) begin
  for each z:IMethod∈⟨f,g,h⟩ do
    z₁:IMethod := z=f?f₁:(z=g?g₁:h₁)
    print z.getMethodName()
    print "Simpson : "+simpson(z,2,5)
    print "Riemann : "+riemann(z,2,5)
    print "Integral: "+(z₁(5)-z₁(2))
  end
end
```

Código 9.42. *Salida por consola del programa 9.39.*

```
f
Simpson : 3.656818483487759
Riemann : 3.664348853690138
Integral: 3.660895201050611
g
Simpson : 143.4056316388403
Riemann : 140.55802915708435
Integral: 140.3242939816356
h
Simpson : 13.218811835029978
Riemann : 13.237948344859236
Integral: 13.224991316943878
```

9.2.2. Métodos numéricos

9.2.2.1. Método de la bisección

Para aproximar numéricamente la raíz de una función $f(x)$ con una precisión ε, existe el método de la bisección.

Código 9.43. *Método de la bisección, implementado recursivamente.*

```
sgn(v) := v=0?0:(v<0?-1:+1)
function bisection(ξ:IMethod,a,b,ε) begin
  assert a≤b and ε>0 and sgn(f(a))*sgn(f(b))≤0
  c := (a+b)/2
  if b-a<ε then
    return c
  elseif sgn(f(c))=sgn(f(a)) then
    return bisection(ξ,c,b,ε)
  else
    return bisection(ξ,a,c,ε)
  end
end
```

Código 9.44. *Método de la bisección, implementado iterativamente.*

```
sgn(v) := v=0?0:(v<0?-1:+1)
function bisection(ξ:IMethod,a,b,ε) begin
  assert a≤b and ε>0 and sgn(f(a))*sgn(f(b))≤0
  while b-a≥ε do
    c := (a+b)/2
    if sgn(f(c))=sgn(f(a)) then
```

```
      a := c
    else
      b := c
    end
  end
  return (a+b)/2
end
```

A continuación se exhiben dos ejemplos que utilizan el método de la bisección para encontrar una solución a la ecuación $x = cos(x)$ mediante la búsqueda de una raíz de la función $f(x) = cos(x) - x$ en el intervalo cerrado $[0,2]$: el programa 9.45 que usa números de tipo `double` y una precisión de $\varepsilon = 10^{-10}$, y el programa 9.46 que usa números de precisión arbitraria y una precisión de $\varepsilon = 10^{-40}$.

Código 9.45. *Aplicación del método de la bisección, trabajando sobre números de tipo* `double`.

```
f(x)=cos(x)-x
procedure main(args:String[]) begin
  x := bisection(f,0,2,1E-10)
  print "x=",x,";f(x)=",f(x)
end
```

Código 9.46. *Aplicación del método de la bisección, trabajando sobre números de precisión arbitraria.*

```
f(x)=cos(x)-x
procedure main(args:String[]) begin
  x := bisection(f,ℝ(0,50),ℝ(2,50),1E-40)
  print "x=",x,";f(x)=",f(x)
end
```

9.3. Arreglos

9.3.1. Algoritmos de ordenamiento

9.3.1.1. Insertion-sort

Código 9.47. *Algoritmo de ordenamiento Insertion-sort.*

```
// ------------------------------------------------------------------------------------------
// Introduction to Algorithms, Thomas Cormen et al., 2nd Edition, pg 17.
// Insertion Sort.
// Temporal complexity: O(n^2).
// ------------------------------------------------------------------------------------------
procedure insertionSort(A) begin
  for j:=1 to |A|-1 do
    key,i:=A[j],j-1
    // Insert A[j] into the sorted sequence A[0..j-1]
    while i≥0 and A[i]>key do
      A[i+1],i:=A[i],i-1
    end
    A[i+1]:=key
  end
end
```

9.3.1.2. Selection-sort

Código 9.48. *Algoritmo de ordenamiento Selection-sort.*

```
// ------------------------------------------------------------------------------------------
// Selection Sort.
// Temporal complexity: O(n^2).
// ------------------------------------------------------------------------------------------
procedure selectionSort(A) begin
  for i:=0 to |A|-2 do
    k:=i
    for j:=i+1 to |A|-1 do
      if A[j]<A[k] then
        k:=j
      end
    end
    swap A[k]↔A[i]
  end
end
```

9.3.1.3. Merge-sort

Código 9.49. *Algoritmo de ordenamiento Merge-sort.*

```
// ------------------------------------------------------------------------------------------
// Introduction to Algorithms, Thomas Cormen et al., 2nd Edition, pg 29.
// Merge Sort.
// Temporal complexity: O(n·log2(n)).
// ------------------------------------------------------------------------------------------
procedure merge(A,B,p,q,r) begin
```

```
7   // Merge A[p..q] and A[q+1..r] into B[p..r]
8   i,j,k:=p,q+1,p
9   while i≤q or j≤r do
10    if j>r or (i≤q and A[i]≤A[j]) then
11      B[k],i,k:=A[i],i+1,k+1
12    else
13      B[k],j,k:=A[j],j+1,k+1
14    end
15  end
16  // Copy B[p..r] into A[p..r]
17  for k:=p to r do
18    A[k]:=B[k]
19  end
20end
21procedure mergeSort(A,B,p,r) begin
22  if p<r then
23    q:=⌊(p+r)/2⌋   // i.e., q:=(p+r)÷2
24    mergeSort(A,B,p,q)   // Sort the first half
25    mergeSort(A,B,q+1,r)   // Sort the second half
26    merge(A,B,p,q,r)     // Merge the two sorted halves
27  end
28end
29procedure mergeSort(A) begin
30  mergeSort(A,GToolkit.clone(A),0,|A|-1)
31end
```

9.3.1.4. Bubble-sort

Código 9.50. *Algoritmo de ordenamiento Bubble-sort.*

```
1  // ---------------------------------------------------------------------------------------
2  // Introduction to Algorithms, Thomas Cormen et al., 2nd Edition, pg 38.
3  // Bubble Sort.
4  // Temporal complexity: O(n^2).
5  // ---------------------------------------------------------------------------------------
6  procedure bubbleSort(A) begin
7    for i:=0 to |A|-1 do
8      for j:=|A|-1 downto i+1 do
9        if A[j]<A[j-1] then
10         swap A[j]↔A[j-1]
11       end
12     end
13   end
14 end
```

9.3.1.5. Heap-sort

Código 9.51. *Algoritmo de ordenamiento Heap-sort.*

```
1  // ---------------------------------------------------------------------------------------
2  // Introduction to Algorithms, Thomas Cormen et al., 2nd Edition, pg 136.
3  // Based on http://www.cs.ubc.ca/~harrison/Java/HeapSortAlgorithm.java.html
4  // Heap Sort.
5  // Temporal complexity: O(n·log2(n)).
6  // ---------------------------------------------------------------------------------------
```

```
7  procedure heapify(A,p,n) begin
8    m:=A[p]
9    while p<n÷2 do
10     i:=2·p+1
11     if i+1<n and A[i]<A[i+1] then
12       i:=i+1
13     end
14     if m≥A[i] then
15       finalize
16     end
17     swap A[p]↔A[i]
18     p:=i
19   end
20 end
21 procedure buildHeap(A) begin
22   for i:=(|A|÷2)-1 downto 0 do
23     heapify(A,i,|A|)
24   end
25 end
26 procedure heapSort(A) begin
27   buildHeap(A)
28   for i:=|A|-1 downto 1 do
29     swap A[0]↔A[i]
30     heapify(A,0,i)
31   end
32 end
```

9.3.1.6. Quick-sort

Código 9.52. *Algoritmo de ordenamiento Quick-sort.*

```
1  // ------------------------------------------------------------------------------------
2  // Introduction to Algorithms, Thomas Cormen et al., 2nd Edition, pg 146.
3  // Quick Sort.
4  // Amortized temporal complexity: O(n·log2(n)).
5  // ------------------------------------------------------------------------------------
6  function partition(A,p,r) begin
7    swap A[r]↔A[GToolkit.random(p,r)] // Randomized version (pg 154)
8    x,i:=A[r],p-1
9    for j:=p to r-1 do
10     if A[j]≤x then
11       i:=i+1
12       swap A[i]↔A[j]
13     end
14   end
15   swap A[i+1]↔A[r]
16   return i+1
17 end
18 procedure quickSort(A,p,r) begin
19   if p<r then
20     q:=partition(A,p,r)
21     quickSort(A,p,q-1)
22     quickSort(A,q+1,r)
23   end
24 end
25 procedure quickSort(A) begin
26   quickSort(A,0,|A|-1)
```

```
27 end
```

9.3.1.7. Stooge-sort

Código 9.53. *Algoritmo de ordenamiento Stooge-sort.*

```
1  // ------------------------------------------------------------------------------------
2  // Introduction to Algorithms, Thomas Cormen et al., 2nd Edition, pg 161.
3  // Stooge Sort.
4  // Temporal complexity: O(n^(ln(3)/ln(1.5))).
5  // ------------------------------------------------------------------------------------
6  procedure stoogeSort(A,i,j) begin
7    if A[i]>A[j] then
8      swap A[i]↔A[j]
9    end
10   if i+1≥j then
11     finalize
12   end
13   k:=⌊(j-i+1)/3⌋   // Round down (i.e., k:=(j-i+1)÷3).
14   stoogeSort(A,i,j-k)   // First two-thirds.
15   stoogeSort(A,i+k,j)   // Last two-thirds.
16   stoogeSort(A,i,j-k)   // First two-thirds again.
17 end
18 procedure stoogeSort(A) begin
19   stoogeSort(A,0,|A|-1)
20 end
```

9.3.1.8. Prueba de los distintos algoritmos de ordenamiento

Los algoritmos de ordenamiento presentados reciben cualquier estructura de datos que pueda ser accedida a través de subíndices: arreglos, tuplas y listas. Los arreglos recibidos como parámetro pueden ser de cualquier tipo, incluyendo los ocho tipos básicos de *Java* y todas las clases.

Código 9.54. *Programa que prueba los algoritmos de ordenamiento estudiados.*

```
1  using java.util.*
2  using gold.**
3  count(A,v):=(Σi|0≤i<|A|,[v=A[i]]:1) // i.e., count(A,v):=v#A
4  isSorted(A):=(∀i|0≤i≤|A|-2:A[i]≤A[i+1])
5  isPermutation(A,B):=|A|=|B|∧(∀i|0≤i<|A|:A[i]#A=A[i]#B)
6  randomList(n):=⟨GToolkit.random(64)|1≤x≤n⟩
7  function randomArray(n) begin
8    var A:int[n]
9    for i:=0 to n-1 do
10     A[i]:=GToolkit.random(64)
11   end
12   return A
13 end
14 function test_sort(A) begin
15   for each method:IMethod∈⟨insertionSort,selectionSort,mergeSort,bubbleSort,heapSort,
16                            quickSort,stoogeSort⟩ do
17     B:=GToolkit.clone(A)
18     method(B)
19     print A,"\r\n",B,"\r\n"
20     assert isSorted(B) and isPermutation(A,B)
```

```
21   end
22   print ""
23 end
24 procedure main(args:String[]) begin
25   test_sort(⟦ℤ(17)!,ℤ(5)!,ℤ(32)!,ℤ(8)!⟧)
26   test_sort(⟦51,42,11,71,92,31⟧)
27   test_sort(("HELLO WORLD!").toCharArray())
28   test_sort(GArrayList(⟨51,42,11,71,92,31⟩))
29   test_sort(GLinkedList(⟨51,42,11,71,92,31⟩))
30   test_sort(GTuple(⟨51,42,11,71,92,31⟩))
31   test_sort(LinkedList(Arrays.asList(51,42,11,71,92,31)))
32   test_sort(ArrayList(Arrays.asList(51,42,11,71,92,31)))
33   test_sort(randomArray(50))
34   test_sort(randomList(50))
35 end
```

9.3.2. Permutaciones

9.3.2.1. Permutaciones de una lista

Código 9.55. *Función recursiva que halla las permutaciones de una lista.*

```
1  using gold.**
2  procedure permutations(data,set:ISet,stack:IStack,result:IList) begin
3    if |stack|=|data| then
4      result.addLast(GArrayList(stack))
5    else
6      for each x∈data do
7        if x∉set then
8          stack.push(x)
9          set.add(x)
10         permutations(data,set,stack,result)
11         set.remove(x)
12         stack.pop()
13       end
14     end
15   end
16 end
17 function permutations(data):IList begin
18   result := GArrayList()
19   permutations(data,GHashTableSet(),GArrayStack(),result)
20   return result
21 end
```

Código 9.56. *Procedimiento para probar la función que halla las permutaciones de una lista.*

```
1  procedure main(args:String[]) begin
2    for each p∈permutations(⟦1,2,3,4⟧) do
3      print p
4    end
5  end
```

9.3.2.2. Permutaciones de una bolsa

Código 9.57. *Función recursiva que halla las permutaciones de una bolsa (primera versión).*

```
using gold.**
procedure permutations(frequencies:IMap,stack:IStack,result:IList) begin
  if (∀x|x∈frequencies.keys():frequencies.get(x)=0) then
    result.addLast(GArrayList(stack))
  else
    for each x∈frequencies.keys() do
      if frequencies.get(x)>0 then
        stack.push(x)
        frequencies.put(x,frequencies.get(x)-1)
        permutations(frequencies,stack,result)
        frequencies.put(x,frequencies.get(x)+1)
        stack.pop()
      end
    end
  end
end
function permutations(data):IList begin
  result := GArrayList()
  frequencies := GLinkedHashTableMap()
  for each x∈data do
    if ¬frequencies.containsKey(x) then
      frequencies.put(x,1)
    else
      frequencies.put(x,frequencies.get(x)+1)
    end
  end
  permutations(frequencies,GArrayStack(),result)
  return result
end
```

Código 9.58. *Función recursiva que halla las permutaciones de una bolsa (segunda versión).*

```
using gold.**
procedure permutations(frequencies:IMap,stack:IStack,result:IList) begin
  if (∀x|x∈frequencies.keys():frequencies[x]=0) then
    result.addLast(GArrayList(stack))
  else
    for each x∈frequencies.keys() do
      if frequencies[x]>0 then
        stack.push(x)
        frequencies[x] := frequencies[x]-1
        permutations(frequencies,stack,result)
        frequencies[x] := frequencies[x]+1
        stack.pop()
      end
    end
  end
end
function permutations(data):IList begin
  result := GArrayList()
  frequencies := GLinkedHashTableMap()
  for each x∈data do
    if x∉frequencies.keys() then
      frequencies[x] := 1
    else
      frequencies[x] := frequencies[x]+1
    end
```

```
26   end
27   permutations(frequencies,GArrayStack(),result)
28   return result
29 end
```

Código 9.59. *Función recursiva que halla las permutaciones de una bolsa (tercera versión).*

```
1  using gold.**
2  procedure permutations(frequencies:IMap,stack:IStack,result:IList) begin
3    if (∀x|x∈frequencies.keys():frequencies[x]=0) then
4      result.addLast(GArrayList(stack))
5    else
6      for each x∈frequencies.keys() do
7        if frequencies[x]>0 then
8          stack.push(x)
9          frequencies[x]:=frequencies[x]-1
10         permutations(frequencies,stack,result)
11         frequencies[x]:=frequencies[x]+1
12         stack.pop()
13       end
14     end
15   end
16 end
17 function permutations(data):IList begin
18   result,frequencies:=GArrayList(),GLinkedHashTableMap()
19   for each x∈data do
20     v:=frequencies[x]
21     frequencies[x]:=v≠NIL?v+1:1
22   end
23   permutations(frequencies,GArrayStack(),result)
24   return result
25 end
```

Código 9.60. *Procedimiento para probar la función que halla las permutaciones de una bolsa.*

```
1  procedure main(args:String[]) begin
2    for each p∈permutations(⟦'h','e','l','l','o'⟧) do
3      print p
4    end
5  end
```

9.3.2.3. Problema de las ocho reinas

Código 9.61. *Solución al problema de las ocho reinas (primera versión).*

```
1  var solutions:int
2  procedure solve(board:char[][]) begin
3    solutions:=0
4    solve(board,0)
5    print solutions," solutions found"
6  end
7  procedure solve(board:char[][],col:int) begin
8    if col=8 then
9      for each row∈board do
10       System.out.println(new String(row))
```

```
    end
    solutions := solutions+1
    System.out.println()
  else
    for row := 0 to 7 do
      board[row][col] := 'R'
      if (¬check(board,row,col)) then solve(board,col+1) end
      board[row][col] := '.'
    end
  end
end
function check(board:char[][],i:int,j:int):𝔹 begin
  for u := 0 to 7 do
    for v := 0 to 7 do
      if ¬(i=u∧j=v)∧board[u][v]='R'∧(i=u∨j=v∨abs(i-u)=abs(j-v)) then
        return true
      end
    end
  end
  return false
end
procedure main(args:String[]) begin
  var board:char[8][8]
  for i := 0 to 7 do
    for j := 0 to 7 do
      board[i][j] := '.'
    end
  end
  solve(board)
end
```

Código 9.62. *Solución al problema de las ocho reinas (segunda versión).*

```
procedure solve(board:char[][]) begin
  print solve(board,0)," solutions found"
end
function solve(board:char[][],col:int):int begin
  if col=8 then
    for each row∈board do print String(row) end
    print ""
    return 1
  else
    r := 0
    for row := 0 to 7 do
      board[row][col] := 'R'
      if (¬check(board,row,col)) then r := r+solve(board,col+1) end
      board[row][col] := '.'
    end
    return r
  end
end
function check(board:char[][],i:int,j:int):𝔹 begin
  return (∃u,v|⟨u,v⟩∈(0..7)^2:¬(i=u∧j=v)∧board[u][v]='R'∧(i=u∨j=v∨|i-u|=|j-v|))
end
procedure main(args:String[]) begin
  var board:char[8][8]
  for i := 0 to 7 do
    GArrays.fill(board[i],'.')
```

```
26   end
27   solve(board)
28 end
```

9.3.3. Matrices

9.3.3.1. Suma de matrices

Código 9.63. *Algoritmo que suma matrices.*

```
1  rows(A):=|A|
2  cols(A):=|A|>0?|A[0]|:0
3  function sum(A,B) begin
4    assert rows(A)=rows(B) and cols(A)=cols(B)
5    n,m:=rows(A),cols(A)
6    C:=Object[n][m]
7    for i:=0 to n-1 do
8      for j:=0 to m-1 do
9        C[i][j]:=A[i][j]+B[i][j]
10     end
11   end
12   return C
13 end
```

Código 9.64. *Procedimiento para probar la suma de matrices.*

```
1 procedure main(args:String[]) begin
2   print sum([[1,7],[3,4]],[[5,2],[9,1]])
3   print sum([[1,7],[3,4],[0,7]],[[5,2],[9,1],[7,2]])
4 end
```

9.3.3.2. Multiplicación de matrices

Código 9.65. *Algoritmo que multiplica matrices (primera versión).*

```
1  rows(A):=|A|
2  cols(A):=|A|>0?|A[0]|:0
3  function mult(A,B) begin
4    assert cols(A)=rows(B)
5    p,q,r:=rows(A),cols(A),cols(B)
6    C:=Object[p][r]
7    for i:=0 to p-1 do
8      for j:=0 to r-1 do
9        sum:=0
10       for k:=0 to q-1 do
11         sum:=sum+A[i][k]·B[k][j]
12       end
13       C[i][j]:=sum
14     end
15   end
16   return C
17 end
```

Código 9.66. *Algoritmo que multiplica matrices (segunda versión).*

```
rows(A):=|A|
cols(A):=|A|>0?|A[0]|:0
function mult(A,B) begin
  assert cols(A)=rows(B)
  p,q,r:=rows(A),cols(A),cols(B)
  C:=Object[p][r]
  for i:=0 to p-1 do
    for j:=0 to r-1 do
      C[i][j]:=(Σk|0≤k<q:A[i][k]·B[k][j])
    end
  end
  return C
end
```

Código 9.67. *Procedimiento para probar la multiplicación de matrices.*

```
procedure main(args:String[]) begin
  print mult([[1,7],[3,4]],[[5,2],[9,1]])
  print mult([[1,7],[3,4],[0,7]],[[5,2],[9,1]])
end
```

9.3.3.3. Método de Gauss-Jordan

Código 9.68. *Método de Gauss-Jordan.*

```
rows(A):=|A|
cols(A):=|A|>0?|A[0]|:0
function extendedGaussJordan(A) begin
  assert rows(A)>0 and cols(A)>0
  n,m,det,ε:=rows(A),cols(A),1,1E-12
  for i:=0 to n-1 do
    w:=i
    for u:=i+1 to n-1 do
      if |A[u][i]|>|A[w][i]| then w:=u end
    end
    if |A[w][i]|<ε then return 0 end // A is not invertible
    if w≠i then A[w],A[i],det:=A[i],A[w],det*-1 end
    z,det:=A[i][i],det*A[i][i]
    for v:=i to m-1 do A[i][v]:=A[i][v]/z end
    for u:=0 to n-1 do
      z:=A[u][i]
      if u≠i ∧ |z|≥ε then
        for v:=i to m-1 do A[u][v]:=A[u][v]-z*A[i][v] end
      end
    end
  end
  return det
end
function determinant(A) begin
  assert rows(A)=cols(A) and rows(A)>0
  n:=rows(A)
  B:=Object[n][n]
  for i:=0 to n-1 do // Deep clonation to preserve A
    B[i]:=GToolkit.clone(A[i])
```

```
30    end
31    return extendedGaussJordan(B)
32  end
33  function inverse(A) begin
34    assert rows(A)=cols(A) and rows(A)>0
35    n := rows(A)
36    B := Object[n][n*2]
37    for i := 0 to n-1 do // Copy A into B and extend it with the identity matrix
38      for j := 0 to n-1 do B[i][j] := A[i][j] end
39      for j := 0 to n-1 do B[i][n+j] := i=j?1:0 end
40    end
41    extendedGaussJordan(B)
42    for i := 0 to n-1 do // Remove the left half of B
43      B[i] := GArrays.copyOfRange(B[i],n,n*2)
44    end
45    return B
46  end
```

Código 9.69. *Procedimiento para probar el método de Gauss-Jordan.*

```
1  procedure main(args:String[]) begin
2    matrix := [[1,9,7],[5,2,6],[7,4,8]]
3    print determinant(matrix)
4    print inverse(matrix)
5  end
```

9.3.4. Funciones estadísticas

9.3.4.1. Promedio aritmético

Código 9.70. *Macro que calcula el promedio de un conjunto de datos.*

```
1  μ(data)=(Σx|x∈data:x)/|data|
```

9.3.4.2. Desviación estándar

Código 9.71. *Función que calcula la desviación estándar de un conjunto de datos.*

```
1  function σ(data) begin
2    n := |data|
3    μ := (Σx|x∈data:x)/n
4    return sqrt((Σx|x∈data:(x-μ)^2)/n)
5  end
```

9.4. Técnicas avanzadas de programación

9.4.1. Dividir y Conquistar

En esta sección se presentan algunos algoritmos que siguen la técnica de *Dividir y Conquistar*, adicionales a los que ya se estudiaron como algoritmos de ordenamiento en la sección §9.3.1 (*Merge-Sort*, *Quick-Sort*, *Stooge-Sort*).

9.4.1.1. Búsqueda Binaria

Código 9.72. *Algoritmo recursivo de búsqueda binaria.*

```
1  function binarySearch(data,p,r,key) begin
2    if p>r then return -(p+1) end // key not found
3    m := (p+r)÷2
4    v := data[m]
5    if key<v then
6      return binarySearch(data,p,m-1,key)
7    elseif key>v then
8      return binarySearch(data,m+1,r,key)
9    else
10     return m // key found
11   end
12 end
13 function binarySearch(data,key) begin
14   return binarySearch(data,0,|data|-1,key)
15 end
```

Código 9.73. *Algoritmo iterativo de búsqueda binaria.*

```
1  function binarySearch(data,p,r,key) begin
2    assert (∀i|p≤i<r:data[i]≤data[i+1])
3    while p≤r do
4      m := (p+r)÷2
5      v := data[m]
6      if key<v then
7        r := m-1
8      elseif key>v then
9        p := m+1
10     else
11       return m // key found
12     end
13   end
14   return -(p+1) // key not found
15 end
16 function binarySearch(data,key) begin
17   return binarySearch(data,0,|data|-1,key)
18 end
```

9.4.1.2. Algoritmo de selección *Quick-Select*

Código 9.74. *Algoritmo de selección Quick-Select (primera versión).*

```
1  function quickSelect(data,index) begin
2    return quickSelect(GToolkit.clone(data),0,|data|-1,index)
3  end
```

```
4  // C. A. R. Hoare's Quick Select Algorithm (taken from Wikipedia)
5  function quickSelect(data,p,r,index) begin
6    while TRUE do
7      s,m:=p,(p+r)÷2
8      data[m],data[r]:=data[r],data[m]
9      piv:=data[r]
10     for j:=p to r-1 do
11       if data[j]<piv then
12         data[s],data[j],s:=data[j],data[s],s+1
13       end
14     end
15     data[s],data[r]:=data[r],data[s]
16     if index=s then
17       return data[index]
18     elseif index<s then
19       r:=s-1
20     else
21       p:=s+1
22     end
23   end
24 end
```

Código 9.75. *Algoritmo de selección Quick-Select (segunda versión).*

```
1  function quickSelect(data,index) begin
2    return quickSelect(GToolkit.clone(data),0,|data|-1,index)
3  end
4  // C. A. R. Hoare's Quick Select Algorithm (taken from Wikipedia)
5  function quickSelect(data,p,r,index) begin
6    while TRUE do
7      s,m:=p,(p+r)÷2
8      swap data[m]↔data[r]
9      piv:=data[r]
10     for j:=p to r-1 do
11       if data[j]<piv then
12         swap data[s]↔data[j]
13         s:=s+1
14       end
15     end
16     swap data[s]↔data[r]
17     if index=s then
18       return data[index]
19     elseif index<s then
20       r:=s-1
21     else
22       p:=s+1
23     end
24   end
25 end
```

9.4.1.3. Potenciación en tiempo logarítmico

Código 9.76. *Algoritmo de potenciación en tiempo logarítmico.*

```
1  function modPow(a,b,m) begin // Calculates a^b mod m
2    if b=0 then
```

```
3     return 1
4   elseif b=1 then
5     return a mod m
6   elseif b mod 2 = 0 then
7     r := modPow(a,b÷2,m)
8     return r·r mod m
9   else
10    r := modPow(a,b÷2,m)
11    return ((r·r mod m)·a) mod m
12  end
13 end
```

9.4.1.4. Algoritmo de Karatsuba

Código 9.77. *Algoritmo de Karatsuba (primera versión).*

```
1  var RADIX:int(2)
2  digits(v) := v=0?1:⌊log(v,RADIX)⌋+1
3  function karatsuba(a,b) begin
4    if a<1000000 and b<1000000 then
5      return a·b
6    else
7      n,m := digits(a),digits(b)
8      d := RADIX^((n↑m)÷2)
9      a1,a2 := a÷d,a%d
10     b1,b2 := b÷d,b%d
11     x,z := karatsuba(a2,b2),karatsuba(a1,b1)
12     y := karatsuba(a1+a2,b1+b2)-x-z
13     return x+d·(y+d·z)
14   end
15 end
```

Código 9.78. *Algoritmo de Karatsuba (segunda versión).*

```
1  using java.math.*
2  // Based on the Java code written by Robert Sedgewick y Kevin Wayne
3  // Reference: http://introcs.cs.princeton.edu/java/78crypto/Karatsuba.java.html
4  function karatsuba(x:BigInteger,y:BigInteger):BigInteger begin
5    n := max(x.bitLength(),y.bitLength())
6    if n≤2000 then
7      return x*y
8    else
9      n := (n÷2)+(n%2)
10     var b:BigInteger,a:BigInteger,d:BigInteger,c:BigInteger
11     b := x.shiftRight(n)
12     a := x.subtract(b.shiftLeft(n))
13     d := y.shiftRight(n)
14     c := y.subtract(d.shiftLeft(n))
15     var ac:BigInteger,bd:BigInteger,abcd:BigInteger
16     ac := karatsuba(a,c)
17     bd := karatsuba(b,d)
18     abcd := karatsuba(a.add(b),c.add(d))
19     return ac.add(abcd.subtract(ac).subtract(bd).shiftLeft(n)).add(bd.shiftLeft(n*2))
20   end
21 end
```

Código 9.79. *Algoritmo de Karatsuba (tercera versión).*

```
using java.math.*
// Based on the Java code written by Robert Sedgewick y Kevin Wayne
// Reference: http://introcs.cs.princeton.edu/java/78crypto/Karatsuba.java.html
function karatsuba(x:BigInteger,y:BigInteger):BigInteger begin
  n := x.bitLength()↑y.bitLength()
  if n≤2000 then
    return x·y
  else
    n := (n÷2)+(n%2)
    b,d := x.shiftRight(n),y.shiftRight(n)
    a,c := x-b.shiftLeft(n),y-d.shiftLeft(n)
    ac,bd,abcd := karatsuba(a,c),karatsuba(b,d),karatsuba(a+b,c+d)
    return ac+((abcd-ac-bd).shiftLeft(n))+(bd.shiftLeft(n·2))
  end
end
```

9.4.1.5. Torres de Hanoi

Código 9.80. *Algoritmo que soluciona el juego de las Torres de Hanoi.*

```
using gold.structures.stack.*
var stacks:IStack[3],counter:int
procedure hanoi(n) begin
  stacks[0] := GArrayStack(⟨n-i|0≤i<n⟩)
  stacks[1] := GArrayStack()
  stacks[2] := GArrayStack()
  counter := 0
  print stacks
  hanoi(0,1,2,n)
  print "Game solved in ",counter," movement(s)"
end
procedure hanoi(a,b,c,n) begin
  if n>0 then
    hanoi(a,c,b,n-1)
    call stacks[c].push(stacks[a].pop())
    counter := counter+1
    print stacks
    hanoi(b,a,c,n-1)
  end
end
procedure main(args:String[]) begin
  hanoi(3)
end
```

Código 9.81. *Salida por consola del programa 9.80.*

```
[[3,2,1], [], []]
[[3,2], [], [1]]
[[3], [2], [1]]
[[3], [2,1], []]
[[], [2,1], [3]]
[[1], [2], [3]]
[[1], [], [3,2]]
[[], [], [3,2,1]]
Game solved in 7 movement(s)
```

9.4.2. Programación dinámica

9.4.2.1. Subsecuencia común más larga (*longest common subsequence*)

Código 9.82. *Algoritmo que calcula la subsecuencia común más larga de dos cadenas de texto.*

```
function lcs(A:String,B:String) begin
  n,m:=|A|,|B|
  var τ:int[n+1][m+1]
  for i:=0 to n do
    for j:=0 to m do
      if i=0 or j=0 then
        τ[i][j]:=0
      elseif A[i-1]=B[j-1] then
        τ[i][j]:=τ[i-1][j-1]+1
      else
        τ[i][j]:=max(τ[i-1][j],τ[i][j-1])
      end
    end
  end
  result,i,j:="",n,m
  while i>0 and j>0 do
    if τ[i][j]=τ[i-1][j-1]+1 then
      result,i,j:=result+A[i-1],i-1,j-1
    elseif τ[i][j]=τ[i-1][j] then
      i:=i-1
    else
      j:=j-1
    end
  end
  return ⟨τ[n][m],result⟩
end
```

Código 9.83. *Algoritmo que calcula la longitud de la subsecuencia común más larga de dos cadenas de texto.*

```
function lcs(A:String,B:String) begin
  n,m:=|A|,|B|
  var τ1:int[m+1],τ2:int[m+1]
  for i:=1 to n do
    for j:=1 to m do
      if A[i-1]=B[j-1] then
        τ2[j]:=τ1[j-1]+1
      else
        τ2[j]:=max(τ1[j],τ2[j-1])
      end
    end
    τ1,τ2:=τ2,τ1
  end
  return τ1[m]
end
```

9.4.2.2. Subsecuencia creciente más larga (*longest increasing subsequence*)

Código 9.84. *Algoritmo que calcula la longitud de la subsecuencia creciente más larga de una lista de números.*

```
function lis(A) begin
```

```
2   n := |A|
3   if n=0 then return 0 end
4   var τ:int[n]
5   for i := 0 to n-1 do
6     m := (↑j|0≤j<i,[A[j]<A[i]]:τ[j])
7     τ[i] := m≠-∞?m+1:1
8   end
9   return (↑i|0≤i<n:τ[i])
10 end
```

9.4.3. Geometría computacional

9.4.3.1. Perímetro y área de polígonos

Código 9.85. *Función que calcula la distancia entre dos puntos.*

```
1 distance(a,b) := sqrt(pow(b[0]-a[0],2)+pow(b[1]-a[1],2))
```

Código 9.86. *Función que calcula el perímetro de un polígono.*

```
1 perimeter(P) := (Σi|0≤i<|P|:distance(P[i],P[(i+1)%|P|]))
```

Código 9.87. *Función que calcula el área de un polígono.*

```
1 area(P) := abs((Σi,j|0≤i<|P|,j=(i+1)%|P|:P[i][0]·P[j][1]-P[i][1]·P[j][0]))/2
```

9.4.3.2. Método de Graham para hallar envolventes convexas (*convex hulls*)

Código 9.88. *Método de Graham para encontrar la envolvente convexa (convex hull) de una colección de puntos.*

```
1  using gold.**
2  sgn(x) := |x|<1e-13?0:(x<0?-1:+1)
3  norm2(⟨x,y⟩) := x^2+y^2
4  cruz(⟨x1,y1⟩,⟨x2,y2⟩) := x1·y2-x2·y1
5  cruz(a,b,c) := cruz(a,b)+cruz(b,c)+cruz(c,a)
6  δ(a,b) := sgn(sgn(cruz(b,a))≠0?cruz(b,a):norm2(a)-norm2(b))
7  function grahamScan(points) begin // Graham's Scan
8    u,v := NIL,NIL
9    for each ⟨x,y⟩∈points do
10     if u=NIL∨y<v∨(y=v∧x>u) then u,v := x,y end
11   end
12   points,r := GCollections.sort(⟨⟨x-u,y-v⟩|⟨x,y⟩∈points⟩,δ),GArrayList()
13   for each p∈points do
14     while |r|≥2∧sgn(cruz(r[|r|-1],p,r[|r|-2]))≤0 do
15       r.removeLast()
16     end
17     r.addLast(p)
18   end
19   return ⟨⟨x+u,y+v⟩|⟨x,y⟩∈r⟩
20 end
```

Código 9.89. *Procedimiento para probar el método de Graham.*

```
procedure main(args:String[]) begin
  print grahamScan([[0,0],[1,0],[2,0],[1,1],[0,2],[-2,0],[-1,1],[-1,0],[0,2]])
end
```

9.4.4. *Stringology*

9.4.4.1. Algoritmo de Knuth-Morris-Pratt para búsqueda de subcadenas

Código 9.90. *Algoritmo de Knuth-Morris-Pratt para búsqueda de subcadenas.*

```
// Knuth-Morris-Pratt table-building algorithm, based on the Wikipedia
function kmp_table(S:char[]):int[] begin
  var T:int[S.length+1]
  i,j:=2,0
  T[0],T[1]:=-1,0
  while i≤S.length do
    if S[i-1]=S[j] then
      T[i],i,j:=j+1,i+1,j+1
    elseif j>0 then
      j:=T[j]
    else
      T[i],i:=0,i+1
    end
  end
  return T
end
// Knuth-Morris-Pratt string searching algorithm, based on the Wikipedia
function indexOf(W:char[],S:char[],T:int[]):int begin
  if W.length=0 then return 0 end
  m,i:=0,0
  while m+i<S.length do
    if W[i]=S[m+i] then
      if i=W.length-1 then return m end
      i:=i+1
    else
      m:=m+i-T[i]
      if i>0 then i:=T[i] end
    end
  end
  return -1
end
function indexOf(W:char[],S:char[]):int begin
  return indexOf(W,S,kmp_table(S))
end
function indexOf(W:String,S:String):int begin
  return indexOf(W.toCharArray(),S.toCharArray())
end
function stringPeriod(S:char[]):int begin
  t:=S.length
  T:=kmp_table(S)
  k:=t-T[t]
  return t%k=0?t÷k:1
end
function period(S:String):int begin
  return period(S.toCharArray())
```

```
46 end
```

Código 9.91. *Procedimiento para probar el algoritmo de Knuth-Morris-Pratt.*

```
1 procedure main(args:String[]) begin
2   print indexOf("Morris","Knuth-Morris-Pratt")
3   print indexOf("Dijkstra","Knuth-Morris-Pratt")
4   print period("abc")
5   print period("abcabc")
6   print period("abcabcabc")
7   print period("abcabcabcabc")
8   print period("abcabcabcabcabc")
9 end
```

9.5. Algoritmos sobre estructuras de datos

9.5.1. Árboles

9.5.1.1. Peso de un árbol

Código 9.92. *Función recursiva que calcula el peso de un árbol binario.*

```
function weight(tree:IBinaryTree):int begin
  if tree.isEmpty() then
    return 0
  else
    return 1+weight(tree.getLeftSubtree())+weight(tree.getRightSubtree())
  end
end
```

Código 9.93. *Función recursiva que calcula el peso de un árbol eneario.*

```
function weight(tree:ITree):int begin
  if tree.isEmpty() then
    return 0
  else
    return 1+(Σsubtree|subtree∈tree.getSubtrees():weight(subtree))
  end
end
```

Código 9.94. *Procedimiento para probar la función que calcula el peso de un árbol.*

```
procedure main(args:String[]) begin
  preorder:=⟨72,25,61,47,98,50,52,80,67,34⟩
  inorder:=⟨61,25,47,72,98,52,50,67,80,34⟩
  tree:=GRecursiveBinaryTree.reconstruct(preorder,inorder)
  print weight(tree)
end
```

9.5.1.2. Altura de un árbol

Código 9.95. *Función recursiva que calcula la altura de un árbol binario.*

```
function height(tree:IBinaryTree):int begin
  if tree.isEmpty() then
    return 0
  else
    return 1+(height(tree.getLeftSubtree())↑height(tree.getRightSubtree()))
  end
end
```

Código 9.96. *Función recursiva que calcula la altura de un árbol eneario.*

```
function height(tree:ITree):int begin
  if tree.isEmpty() then
    return 0
  else
    return 1+(↑subtree|subtree∈tree.getSubtrees():height(subtree))
```

```
6   end
7 end
```

Código 9.97. *Procedimiento para probar la función que calcula la altura de un árbol.*

```
1 procedure main(args:String[]) begin
2   preorder := ⟨72,25,61,47,98,50,52,80,67,34⟩
3   inorder := ⟨61,25,47,72,98,52,50,67,80,34⟩
4   tree := GRecursiveBinaryTree.reconstruct(preorder,inorder)
5   print height(tree)
6 end
```

9.5.1.3. Conteo de ocurrencias de un valor en un árbol

Código 9.98. *Función que cuenta cuántas veces ocurre un determinado valor en un árbol binario.*

```
1  function count(tree:IBinaryTree,value):int begin
2    if tree.isEmpty() then
3      return 0
4    else
5      a := tree.getRoot()=value?1:0
6      b := count(tree.getLeftSubtree(),value)
7      c := count(tree.getRightSubtree(),value)
8      return a+b+c
9    end
10 end
```

Código 9.99. *Función que cuenta cuántas veces ocurre un determinado valor en un árbol eneario.*

```
1 function count(tree:ITree,value):int begin
2   if tree.isEmpty() then
3     return 0
4   else
5     a := tree.getRoot()=value?1:0
6     b := (Σsubtree|subtree∈tree.getSubtrees():count(subtree,value))
7     return a+b
8   end
9 end
```

Código 9.100. *Procedimiento para probar la función que cuenta el número de ocurrencias de un valor en un árbol.*

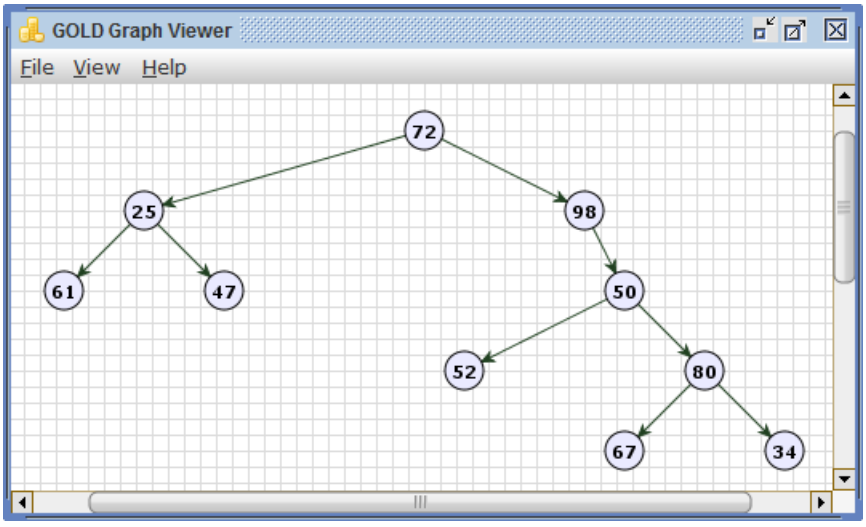
```
1  ξ() := GRecursiveBinaryTree() // Empty binary tree
2  ξ(key,left,right) := GRecursiveBinaryTree(key,left,right) // Binary tree
3  procedure main(args:String[]) begin
4    var tree:ITree
5    θ := ξ()
6    tree := ξ(9,ξ(1,ξ(7,θ,θ),ξ(7,θ,θ)),ξ(2,θ,ξ(6,ξ(9,ξ(4,ξ(7,θ,θ),θ),θ),ξ(5,θ,θ))))
7    print tree
8    for i := 0 to 9 do
9      print "i=",i,",count=",count(tree,i)
10     assert count(tree,i)=tree.count(i)
11   end
12 end
```

9.5.1.4. Reconstrucción de árboles binarios

Código 9.101. *Función recursiva para reconstruir un árbol binario dado su preorden y su inorden.*

```
using gold.**
isPermutation(A,B):=|A|=|B|∧(∀i|0≤i<|A|:A[i]#A=A[i]#B)
areDistinct(A):=(∀i,j|0≤i<|A|,i<j<|A|:A[i]≠A[j])
function search(A,key) begin
  for i:=0 to |A|-1 do
    if A[i]=key then return i else end
  end
  return -1
end
function reconstruct(preord,inord):IBinaryTree begin
  assert |preord|=|inord| and isPermutation(preord,inord) and areDistinct(preord)
  n:=|inord|
  if n=0 then
    return GRecursiveBinaryTree()
  else
    key:=preord[0]
    index:=search(inord,key)
    left:=reconstruct(preord[1..index],inord[0..index-1])
    right:=reconstruct(preord[index+1..n-1],inord[index+1..n-1])
    return GRecursiveBinaryTree(key,left,right)
  end
end
```

Figura 9.1. *Ventana gráfica desplegada después de ejecutar el programa 9.102.*



Código 9.102. *Procedimiento para probar la función que reconstruye árboles binarios.*

```
procedure main(args:String[]) begin
  preorder:=⟨72,25,61,47,98,50,52,80,67,34⟩
  inorder:=⟨61,25,47,72,98,52,50,67,80,34⟩
  tree:=reconstruct(preorder,inorder)
  print tree
  print "Preorder :",tree.preOrderTraversal()
  print "Inorder  :",tree.inOrderTraversal()
```

```
8   print "Postorder:",tree.postOrderTraversal()
9   print "Levels   :",tree.levelOrderTraversal()
10  GGraphFrame.show(tree)
11 end
```

Código 9.103. *Salida por consola del programa 9.102.*

```
1 [72:[25:[61:Ø,Ø],[47:Ø,Ø]],[98:Ø,[50:[52:Ø,Ø],[80:[67:Ø,Ø],[34:Ø,Ø]]]]]
2 Preorder :[72,25,61,47,98,50,52,80,67,34]
3 Inorder  :[61,25,47,72,98,52,50,67,80,34]
4 Postorder:[61,47,25,52,67,34,80,50,98,72]
5 Levels   :[72,25,98,61,47,50,52,80,67,34]
```

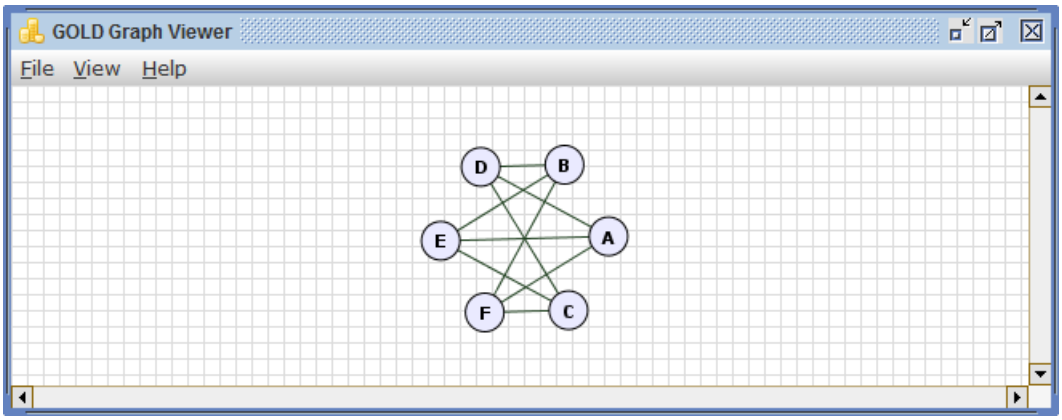
9.5.2. Grafos

9.5.2.1. Definición de grafos

Código 9.104. *Definición y configuración de la visualización del grafo K3,3.*

```
1 using gold.**
2 procedure main(args:String[]) begin
3   graph := GUndirectedGraph('A'..'F',('A'..'C')×('D'..'F'))
4   frame := GGraphFrame(graph)
5   frame.getPainter().setShowEdgeCosts(false)
6   frame.setVisible(true)
7 end
```

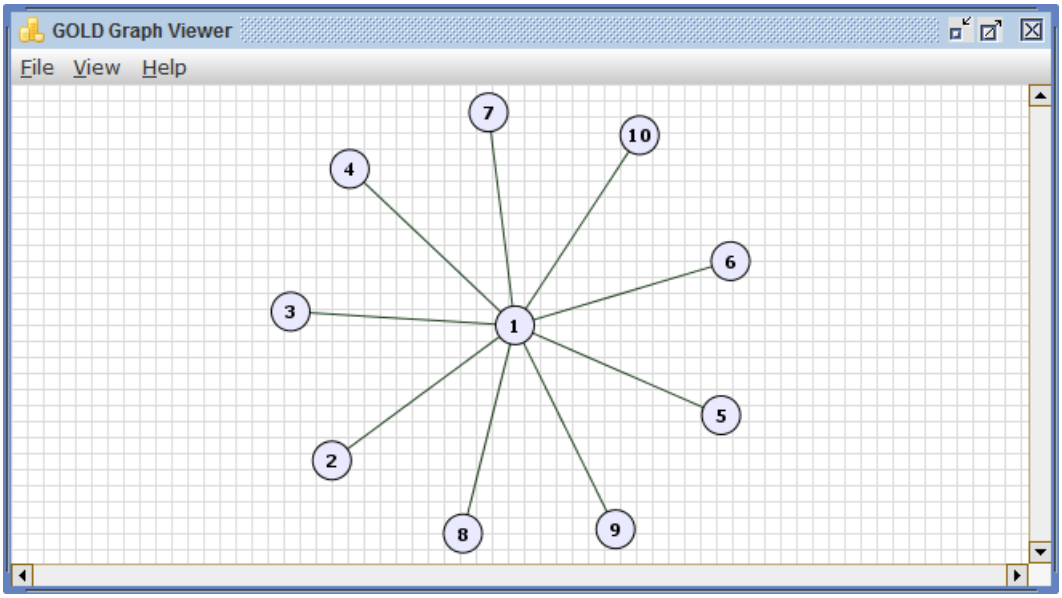
Figura 9.2. *Ventana gráfica desplegada después de ejecutar el programa 9.104.*



Código 9.105. *Definición y configuración de la visualización de un grafo estrellado de nueve puntas.*

```
1 using gold.**
2 procedure main(args:String[]) begin
3   graph := GUndirectedGraph(1..10,{⟨1,k⟩|k∈2..10})
4   frame := GGraphFrame(graph)
5   frame.getPainter().setShowEdgeCosts(false)
6   frame.getPainter().setLayoutType(GLayoutType.FR) // Fruchterman-Reingold layout
7   frame.setVisible(true)
8 end
```

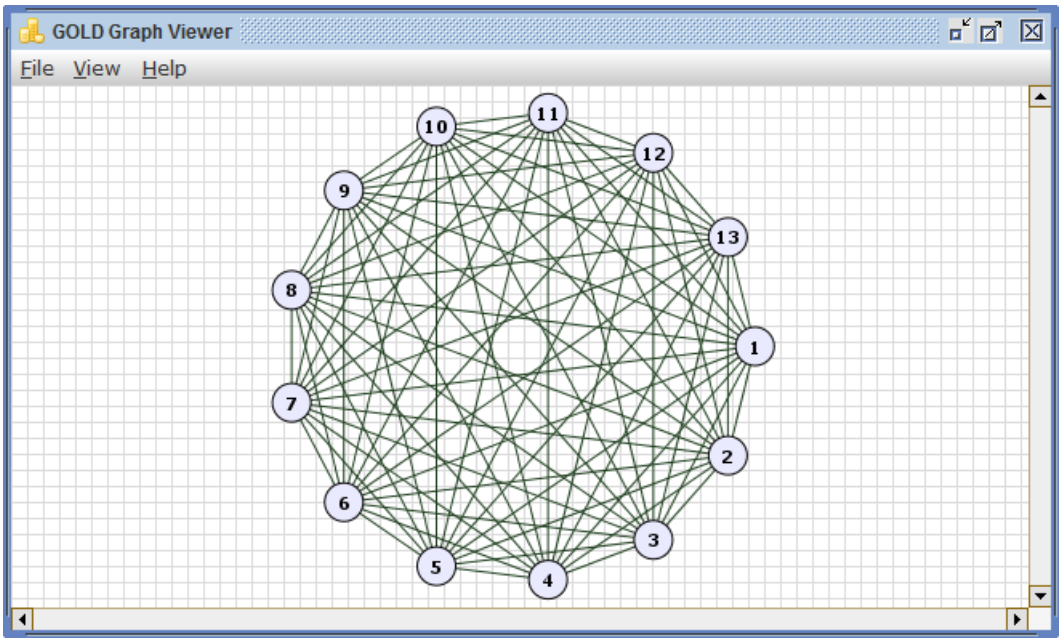
Figura 9.3. *Ventana gráfica desplegada después de ejecutar el programa 9.105.*



Código 9.106. *Definición y configuración de la visualización de un grafo completo con 13 nodos.*

```
1 using gold.**
2 procedure main(args:String[]) begin
3   graph := GUndirectedGraph(1..13,{⟨u,v⟩|⟨u,v⟩∈(1..13)^2,[u≠v]})
4   frame := GGraphFrame(graph)
5   frame.getPainter().setShowEdgeCosts(false)
6   frame.getPainter().setLayoutType(GLayoutType.CIRCLE)
7   frame.setVisible(true)
8 end
```

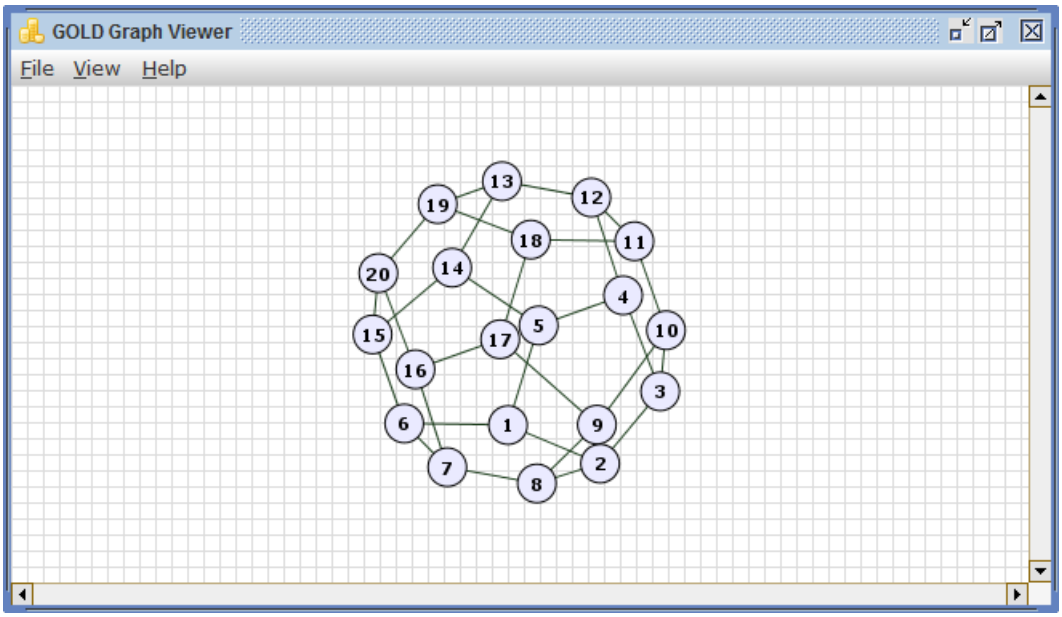
Figura 9.4. *Ventana gráfica desplegada después de ejecutar el programa 9.106.*



Código 9.107. *Definición y configuración de la visualización de un grafo en forma de dodecaedro.*

```
using gold.**
procedure main(args:String[]) begin
  V := 1..20
  E := {⟨x,x+1⟩|x∈V,[5∤x]}∪{⟨1,5⟩,⟨6,15⟩,⟨10,11⟩,⟨16,20⟩}
  E := E∪{⟨x,2·(x+2)⟩|1≤x≤5}∪{⟨2·(y-13)+1,y⟩|16≤y≤20}
  graph := GUndirectedGraph(V,E)
  frame := GGraphFrame(graph)
  frame.getPainter().setShowEdgeCosts(false)
  frame.setVisible(true)
end
```

Figura 9.5. *Ventana gráfica desplegada después de ejecutar el programa 9.107.*

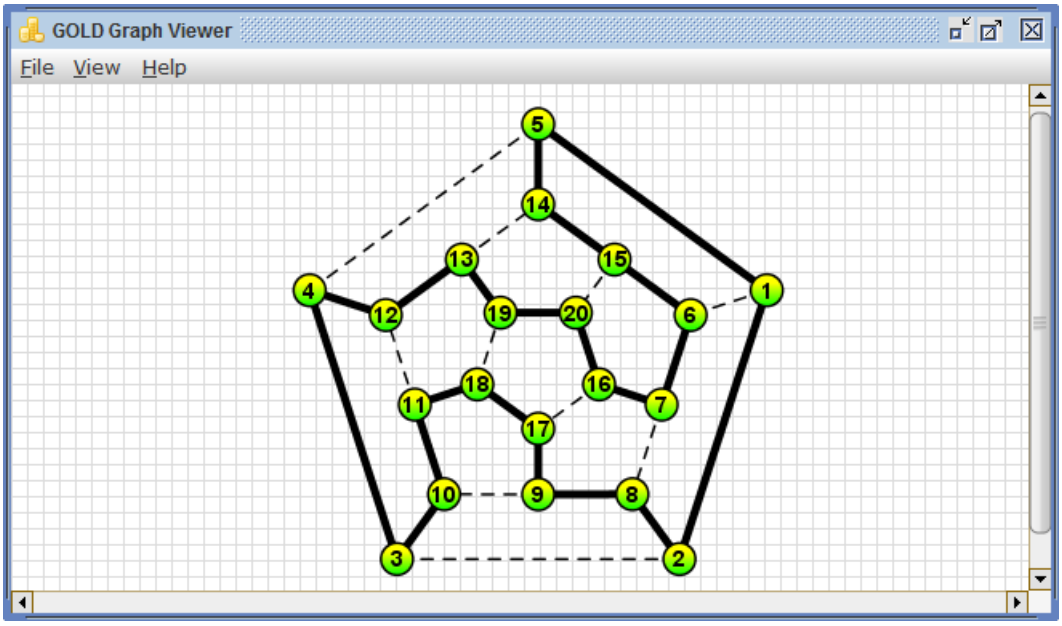


Código 9.108. *Definición y configuración de la visualización de un grafo con un ciclo hamiltoniano.*

```
using java.awt.*
using gold.**
procedure main(args:String[]) begin
  var layout:edu.uci.ics.jung.algorithms.layout.Layout,p:GGraphPainter
  V := 1..20
  E := {⟨x,x+1⟩|x∈V,[5∤x]}∪{⟨1,5⟩,⟨6,15⟩,⟨10,11⟩,⟨16,20⟩}
  E := E∪{⟨x,2·(x+2)⟩|1≤x≤5}∪{⟨2·(y-13)+1,y⟩|16≤y≤20}
  graph := GUndirectedGraph(V,E)
  frame := GGraphFrame(graph)
  p := frame.getPainter()
  p.setShowEdgeCosts(false)
  p.setLayoutType(GLayoutType.CIRCLE)
  p.getVertexShapeTransformer().setAll(GShapes.circle(0,0,10))
  p.getVertexFillPaintTransformer().setAll(GradientPaint(0,-5,Color.YELLOW,0,10,Color.GREEN))
  p.getVertexFontTransformer().setAll(Font("Arial",Font.BOLD,14))
  p.getEdgeStrokeTransformer().setAll(BasicStroke(1.5f))
  p.getVertexStrokeTransformer().setAll(BasicStroke(1.5f))
  p.getEdgeDrawPaintTransformer().setAll(Color.BLACK)
  p.getVertexDrawPaintTransformer().setAll(Color.BLACK)
```

```
20    layout := frame.getPainter().getLayout()
21    for each v∈V do
22      r,α,i := 0,0,0
23      if 6≤v≤15 then
24        r,α,i := v%2=0?100:81,v%2=0?0:36,(v-6)÷2
25      else
26        r,α,i := v≤5?150:40,v≤5?0:36,v-(v≤5?1:16)
27      end
28      x0,y0,θ := 260,180,Math.toRadians(α)+(Math.PI·2/5)·(i-0.25)
29      layout.setLocation(v,GPoint(⟨x0+r*cos(θ),y0+r*sin(θ)⟩))
30    end
31    stack := GArrayStack()
32    result := hamiltonianCycle(graph,stack,1)
33    if result then
34      var arr:Object[]
35      arr := stack.toArray()
36      edges := {⟨arr[i-1],arr[i]⟩|1≤i<|arr|}∪{⟨arr[i],arr[i-1]⟩|1≤i<|arr|}
37      dash := BasicStroke(1.5f,BasicStroke.CAP_ROUND,BasicStroke.JOIN_MITER,1,float[][7,7],0)
38      frame.getPainter().getEdgeStrokeTransformer().setDefaultValue(dash)
39      frame.getPainter().getEdgeStrokeTransformer().setAll(edges,BasicStroke(5.0f))
40    end
41    frame.setVisible(true)
42  end
43  function hamiltonianCycle(G:IGraph,P:IStack,v) begin
44    P.push(v)
45    if |G.getVertices()|=|P| ∧ P[0]∈G.getSuccessors(v) then
46      P.push(P[0]) // Close the cycle
47      return true // A hamiltonian cycle was found
48    end
49    for each w∈G.getSuccessors(v) do
50      if w∉P ∧ hamiltonianCycle(G,P,w) then return true end
51    end
52    P.pop()
53    return false
54  end
```

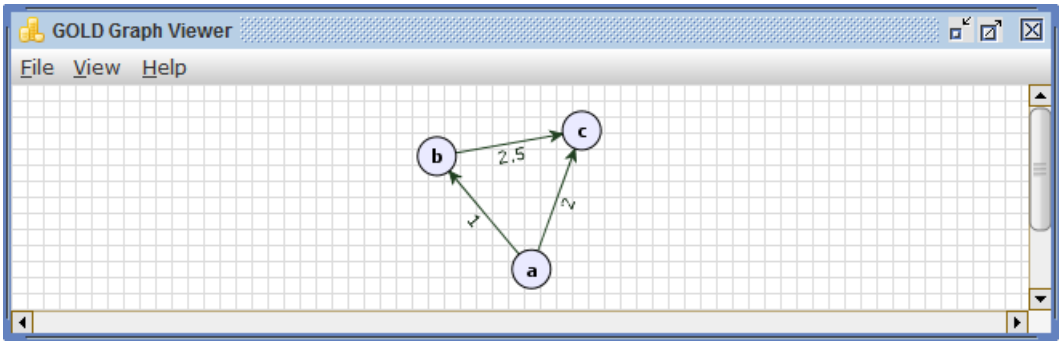
Figura 9.6. *Ventana gráfica desplegada después de ejecutar el programa 9.108.*



Código 9.109. *Definición y configuración de un grafo con costos (primer ejemplo).*

```
using gold.structures.graph.*
using gold.visualization.graph.*
procedure main(args:String[]) begin
  graph := GDirectedGraph({'a','b','c'})
  graph.addEdge('a','b',1.0)
  graph.addEdge('b','c',2.5)
  graph.addEdge('a','c',2.0)
  GGraphFrame.show(graph)
end
```

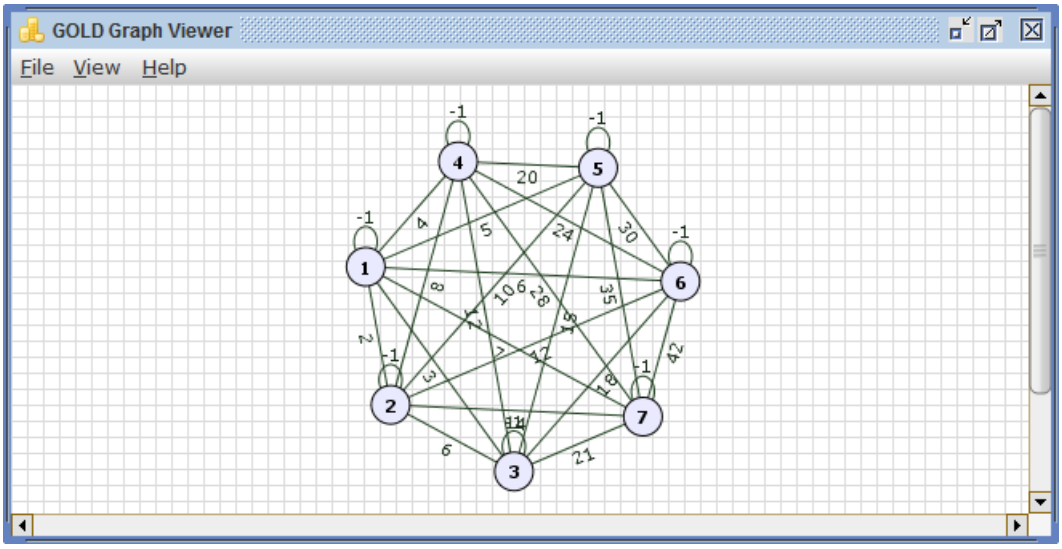
Figura 9.7. *Ventana gráfica desplegada después de ejecutar el programa 9.109.*



Código 9.110. *Definición y configuración de un grafo con costos (segundo ejemplo).*

```
using gold.**
procedure main(args:String[]) begin
  graph := GUndirectedGraph(1..7,{⟨i,j,i=j?-1:i*j⟩|1≤i≤7,i≤j≤7})
  GGraphFrame.show(graph)
end
```

Figura 9.8. *Ventana gráfica desplegada después de ejecutar el programa 9.110.*



9.5.2.2. *Breadth-First Search* (*BFS*)

Código 9.111. *Breadth-First Search (primera versión).*

```
using gold.**
// Introduction to Algorithms, Thomas Cormen et al., 2nd Edition, pg 532.
// Breadth First Search (BFS), iterative version
function bfs(G:IGraph,s) begin
  V:=G.getVertices()
  color,d,π:=GHashTableMap(|V|),GHashTableMap(|V|),GHashTableMap(|V|)
  for each u∈V do
    color[u],d[u],π[u]:="WHITE",∞,NIL
  end
  color[s],d[s],π[s]:="GRAY",0,NIL
  Q:=GArrayQueue()
  Q.enqueue(s)
  while Q≠∅ do
    u:=Q.dequeue()
    for each v∈G.getSuccessors(u) do
      if color[v]="WHITE" then
        color[v],d[v],π[v]:="GRAY",d[u]+1,u
        Q.enqueue(v)
      end
    end
    color[u]:="BLACK"
  end
  return ⟨color,d,π⟩
end
```

Código 9.112. *Breadth-First Search (segunda versión).*

```
using gold.**
// Breadth First Search (BFS), simple iterative version
function bfs(G:IGraph,s) begin
  visited,queue,list:=GHashTableSet(|G.getVertices()|),GArrayQueue(),GArrayList()
  visited.add(s)
  queue.enqueue(s)
  while queue≠∅ do
    u:=queue.dequeue()
    list.addLast(u)
    for each v∈G.getSuccessors(u) do
      if v∉visited then
        visited.add(v)
        queue.enqueue(v)
      end
    end
  end
  return list
end
```

9.5.2.3. *Depth-First-Search* (*DFS*)

Código 9.113. *Depth-First Search (primera versión).*

```
using gold.**
// Introduction to Algorithms, Thomas Cormen et al., 2nd Edition, pg 541.
```

```
3  // Depth First Search (DFS), recursive version
4  var time:int
5  function dfs(G:IGraph) begin
6    V := G.getVertices()
7    d,f := GHashTableMap(|V|),GHashTableMap(|V|)
8    color,π := GHashTableMap(|V|),GHashTableMap(|V|)
9    for each u∈V do
10     color[u],π[u] := "WHITE",NIL
11   end
12   time := 0
13   for each u∈V do
14     if color[u]="WHITE" then
15       dfs_visit(G,u,d,f,color,π)
16     end
17   end
18   return ⟨d,f,color,d,π⟩
19 end
20 procedure dfs_visit(G:IGraph,u,d:IMap,f:IMap,color:IMap,π:IMap) begin
21   time := time+1
22   color[u],d[u] := "GRAY",time      // White vertex u has just been discovered
23   for each v∈G.getSuccessors(u) do // Explore edge (u,v)
24     if color[v]="WHITE" then
25       π[v] := u
26       dfs_visit(G,v,d,f,color,π)
27     end
28   end
29   time := time+1
30   color[u],f[u] := "BLACK",time    // Blacken u; it is finished
31 end
```

Código 9.114. *Depth-First Search (segunda versión).*

```
1  using gold.**
2  // Depth First Search (DFS), simple recursive version
3  function dfs(G:IGraph,s) begin
4    visited,list := GHashTableSet(|G.getVertices()|),GArrayList()
5    dfs(G,s,visited,list)
6    return list
7  end
8  procedure dfs(G:IGraph,s,visited:ISet,list:IList) begin
9    if s∉visited then
10     visited.add(s)
11     list.add(s)
12     for each t∈G.getSuccessors(s) do
13       dfs(G,t,visited,list)
14     end
15   end
16 end
```

9.5.2.4. Algoritmo de Dijkstra

Código 9.115. *Algoritmo de Dijkstra (primera versión).*

```
1  using gold.**
2  // Introduction to Algorithms, Thomas Cormen et al., 2nd Edition, pg 595.
3  // Dijkstra's algorithm
```

```
4  function dijkstra(G:IGraph,s) begin
5    V:=G.getVertices()
6    d,π:=GHashTableMap(|V|),GHashTableMap(|V|)
7    for each v∈V do
8      d[v],π[v]:=∞,NIL
9    end
10   d[s]:=0
11   Q:=GFibonacciHeap(d)
12   while Q≠∅ ∧ Q.minimumKey()≠∞ do
13     u:=Q.extractMinimum()
14     for each v∈G.getSuccessors(u) do
15       if v∈Q ∧ d[v]>d[u]+G.getCost(u,v) then
16         d[v],π[v]:=d[u]+G.getCost(u,v),u
17         Q.decreaseKey(v,d[v])
18       end
19     end
20   end
21   return ⟨d,π⟩
22 end
```

Código 9.116. *Algoritmo de Dijkstra (segunda versión).*

```
1  using gold.**
2  // Dijkstra's algorithm, second version
3  function dijkstra2(G:IGraph,s) begin
4    d,π,Q:=GHashTableMap(),GHashTableMap(),GFibonacciHeap()
5    d[s]:=0
6    Q.insert(s,0)
7    while Q≠∅ do
8      u:=Q.extractMinimum()
9      for each v∈G.getSuccessors(u) do
10       if d[v]=NIL ∨ d[v]>d[u]+G.getCost(u,v) then
11         d[v],π[v]:=d[u]+G.getCost(u,v),u
12         Q.insert(v,d[v]) // Decrease key if it is present
13       end
14     end
15   end
16   return ⟨d,π⟩
17 end
```

Código 9.117. *Algoritmo de Dijkstra (tercera versión).*

```
1  using gold.**
2  // Dijkstra's algorithm, third version
3  function dijkstra3(G:IGraph,s,t) begin
4    d,Q:=GHashTableMap(),GFibonacciHeap()
5    d[s]:=0
6    Q.insert(s,0)
7    while Q≠∅ do
8      u:=Q.extractMinimum()
9      if u=t then
10       return d[u]
11     end
12     for each v∈G.getSuccessors(u) do
13       if d[v]=NIL ∨ d[v]>d[u]+G.getCost(u,v) then
14         d[v]:=d[u]+G.getCost(u,v)
15         Q.insert(v,d[v]) // Decrease key if it is present
16       end
```

```
    end
  end
  return ∞
end
```

9.5.2.5. Algoritmo *bucket shortest path*

Código 9.118. *Algoritmo bucket shortest path.*

```
using gold.**
// Bucket priority queue shortest path algorithm
function bucketShortestPath(G:IGraph,s,t,L:int) begin
  buckets := ⟨GArrayQueue()|0≤i≤L⟩
  visited := GHashTableSet(G.getVertexCount())
  buckets.get(0).enqueue(s)
  cost := 0
  while (∃i|0≤i≤L:buckets[i]≠∅) do
    first := buckets.getFirst()
    while first≠∅ do
      u := first.dequeue()
      if u∉visited then
        if u=t then return cost end
        visited.add(u)
        for each v∈G.getSuccessors(u) do
          c := (⌊G.getCost(u,v)+0.5⌋ as int)
          buckets.get(c).enqueue(v)
        end
      end
    end
    cost := cost+1
    buckets.addLast(buckets.removeFirst())
  end
  return ∞
end
```

9.5.2.6. Algoritmo de Bellman-Ford

Código 9.119. *Algoritmo de Bellman-Ford.*

```
using gold.**
// Introduction to Algorithms, Thomas Cormen et al., 2nd Edition, pg 588.
// Bellman-Ford algorithm
function bellmanFord(G:IGraph,s) begin
  V,E := G.getVertices(),G.getEdges()
  d,π := GHashTableMap(|V|),GHashTableMap(|V|)
  for each v∈V do
    d[v],π[v] := ∞,NIL
  end
  d[s] := 0
  for i := 1 to |V|-1 do
    for each ⟨u,v⟩∈E do
      if d[v]>d[u]+G.getCost(u,v) then
        d[v],π[v] := d[u]+G.getCost(u,v),u
      end
    end
```

```
17    end
18    for each ⟨u,v⟩∈E do
19      if d[v]>d[u]+G.getCost(u,v) then
20        error "Negative-weight cycle found!"
21      end
22    end
23    return ⟨d,π⟩
24  end
```

9.5.2.7. Algoritmo de Floyd-Warshall

Código 9.120. *Algoritmo de Floyd-Warshall (primera versión).*

```
1  using gold.**
2  // Introduction to Algorithms, Thomas Cormen et al., 2nd Edition, pg 630.
3  // Floyd-Warshall algorithm
4  function floydWarshall(G:IGraph) begin
5    var d:double[][]
6    n,d:=G.getVertexCount(),G.getCostMatrix()
7    for k:=0 to n-1 do
8      for i:=0 to n-1 do
9        for j:=0 to n-1 do
10         d[i][j]:=d[i][j]↓d[i][k]+d[k][j]
11       end
12     end
13   end
14   return d
15 end
```

Código 9.121. *Algoritmo de Floyd-Warshall (segunda versión).*

```
1  using gold.**
2  // Floyd-Warshall algorithm, generic version
3  function floydWarshall(G:IGraph,d,f:IMethod,g:IMethod) begin
4    n:=G.getVertexCount()
5    for k:=0 to n-1 do
6      for i:=0 to n-1 do
7        for j:=0 to n-1 do
8          d[i][j]:=f(d[i][j],g(d[i][k],d[k][j]))
9        end
10     end
11   end
12   return d
13 end
14 $min(a,b):=a↓b
15 $sum(a,b):=a+b
16 floydWarshallMinSum(G:IGraph):=floydWarshall(G,G.getCostMatrix(),$min,$sum)
17 $max(a,b):=a↑b
18 $mul(a,b):=a·b
19 floydWarshallMaxMul(G:IGraph):=floydWarshall(G,G.getCostMatrix(),$max,$mul)
20 $or(a,b):=a∨b
21 $and(a,b):=a∧b
22 transitiveClosure(G:IGraph):=floydWarshall(G,G.getAdjacencyMatrix(),$or,$and)
```

9.5.2.8. Algoritmo de Kruskal

Código 9.122. *Algoritmo de Kruskal (primera versión).*

```
1  using gold.**
2  // Introduction to Algorithms, Thomas Cormen et al., 2nd Edition, pg 569.
3  // Kruskal's algorithm
4  function kruskal(G:IGraph) begin
5    V,E,F := G.getVertices(),G.getEdges(),GForestDisjointSets()
6    A := ∅
7    for each v∈V do
8      F.makeSet(v)
9    end
10   E := GCollections.sort(GArrayList(E))
11   for each ⟨u,v⟩∈E do
12     if F.findSet(u)≠F.findSet(v) then
13       A := A∪{⟨u,v⟩}
14       F.union(u,v)
15     end
16   end
17   return A
18 end
```

Código 9.123. *Algoritmo de Kruskal (segunda versión).*

```
1  using gold.**
2  // Introduction to Algorithms, Thomas Cormen et al., 2nd Edition, pg 569.
3  // Kruskal's algorithm, second version
4  function kruskal2(G:IGraph) begin
5    V,E := G.getVertices(),GArrayList(G.getEdges())
6    F := GForestDisjointSets(V)
7    GCollections.sort(E)
8    A := ∅
9    for each ⟨u,v⟩∈E do
10     if F.findSet(u)≠F.findSet(v) then
11       A := A∪{⟨u,v⟩}
12       F.union(u,v)
13     end
14   end
15   return A
16 end
```

Código 9.124. *Algoritmo de Kruskal (tercera versión).*

```
1  using gold.**
2  // Introduction to Algorithms, Thomas Cormen et al., 2nd Edition, pg 569.
3  // Kruskal's algorithm, third version
4  function kruskal3(G:IGraph) begin // Kruskal's algorithm
5    var F:GForestDisjointSets(|G.getVertices()|)
6    for each v∈G.getVertices() do
7      F.makeSet(v)
8    end
9    var E:GArrayList(G.getEdges())
10   GCollections.sort(E)
11   A := ∅
12   for each e:IEdge∈E do
13     u,v := e.getSource(),e.getTarget()
```

```
14      if F.findSet(u)≠F.findSet(v) then
15        A := A∪{⟨u,v⟩}
16        F.union(u,v)
17      end
18    end
19    return A
20  end
```

Código 9.125. *Algoritmo de Kruskal (cuarta versión).*

```
1  using gold.**
2  // Introduction to Algorithms, Thomas Cormen et al., 2nd Edition, pg 569.
3  // Kruskal's algorithm, fourth version
4  function kruskal4(G:IGraph) begin
5    V,E,F := G.getVertices(),GArrayList(G.getEdges()),GForestDisjointSets()
6    for each v∈V do
7      F.makeSet(v)
8    end
9    GCollections.sort(E,ρ)
10   A := ∅
11   for each ⟨u,v⟩∈E do
12     if F.findSet(u)≠F.findSet(v) then
13       A := A∪{⟨u,v⟩}
14       F.union(u,v)
15     end
16   end
17   return A
18 end
19 //ρ(x:IEdge,y:IEdge) := x.getCost()<y.getCost()?-1:(x.getCost()=y.getCost()?0:+1)
20 ρ(x:IEdge,y:IEdge) := x.getCost().compareTo(y.getCost())
```

9.5.2.9. Algoritmo de Prim-Jarník

Código 9.126. *Algoritmo de Prim-Jarník.*

```
1  using gold.**
2  // Introduction to Algorithms, Thomas Cormen et al., 2nd Edition, pg 572.
3  // Prim's algorithm
4  prim(G:IGraph) := prim(G,G.getVertices().pick())
5  function prim(G:IGraph,r) begin
6    V := G.getVertices()
7    key,π := GHashTableMap(|V|),GHashTableMap(|V|)
8    for each u∈V do
9      key[u],π[u] := ∞,NIL
10   end
11   key[r] := 0
12   Q := GFibonacciHeap(key)
13   while Q≠∅ ∧ Q.minimumKey()≠∞ do
14     u := Q.extractMinimum()
15     for each v∈G.getSuccessors(u) do
16       if v∈Q ∧ G.getCost(u,v)<key[v] then
17         π[v],key[v] := u,G.getCost(u,v)
18         Q.decreaseKey(v,key[v])
19       end
20     end
21   end
```

```
22    return {⟨π[u],u⟩|u∈V,[π[u]≠NIL]}
23  end
```

9.5.2.10. Algoritmo de Borůvka

Código 9.127. *Algoritmo de Borůvka.*

```
1  using gold.**
2  // Borůvka's algorithm
3  // {Pre Q: G is connected and contains edges of distinct weights}
4  function boruvka(G:IGraph) begin
5    F := GForestDisjointSets(G.getVertices())
6    M,T := {{v}|v∈G.getVertices()},∅
7    while |M|>1 do
8      E := ∅
9      for each component∈M do
10       c,e := ∞,NIL
11       for each v∈component do
12         for each w∈G.getSuccessors(v) do
13           if F.findSet(v)≠F.findSet(w) ∧ G.getCost(v,w)<c then
14             c,e := G.getCost(v,w),⟨v,w⟩
15           end
16         end
17       end
18       if e≠NIL then
19          E := E∪{e}
20       end
21     end
22     for each ⟨v,w⟩∈E do
23       if F.findSet(v)≠F.findSet(w) then
24         F.union(v,w)
25         T := T∪{⟨v,w⟩}
26         Z := {s|s∈M,[v∈s∨w∈s]}
27         M := (M\Z)∪{(∪x|x∈Z:x)}
28       end
29     end
30   end
31   return T
32 end
```

9.5.3. Redes de flujo

9.5.3.1. Algoritmo de Edmonds-Karp

Código 9.128. *Algoritmo de Edmonds-Karp.*

```
1  using gold.**
2  // Ford-Fulkerson / Edmonds-Karp algorithm
3  function edmondsKarp(G:IGraph,s,t) begin
4    τ,V,E := GHashTableMap(),G.getVertices(),G.getEdges()
5    for each v∈V do
6      τ[v] := τ.size()
7    end
8    δ,n := 0,|V|
9    ζ,flow,capacity := GHashTableMap(),double[n][n],G.getCostMatrix()
```

```
  for u:=0 to n-1 do
    for v:=0 to n-1 do
      if capacity[u][v]=∞ then
        capacity[u][v]:=0
      end
    end
  end
  neighbors:=GHashTableMultiMap()
  for each ⟨u,v⟩∈G.getEdges() do
    neighbors.add(u,v)
    neighbors.add(v,u)
  end
  flag:=true
  while flag do
    π,Q:=GHashTableMap(),GArrayQueue()
    ζ[s]:=∞
    Q.enqueue(s)
    flag:=false
    while Q≠∅ ∧ ¬flag do
      u:=Q.dequeue()
      for each v∈neighbors[u] do
        z:=capacity[τ[u]][τ[v]]-flow[τ[u]][τ[v]]
        if z>0∧π[v]=NIL then
          x:=ζ[u]↓z
          π[v],ζ[v]:=u,x
          if v=t then
            δ,w:=δ+x,t
            while w≠s do
              u:=π[w]
              flow[τ[u]][τ[w]]:=flow[τ[u]][τ[w]]+x
              flow[τ[w]][τ[u]]:=flow[τ[w]][τ[u]]-x
              w:=u
            end
            flag:=true
            break
          end
          Q.enqueue(v)
        end
      end
    end
  end
  return δ
end
```

9.5.4. Autómatas

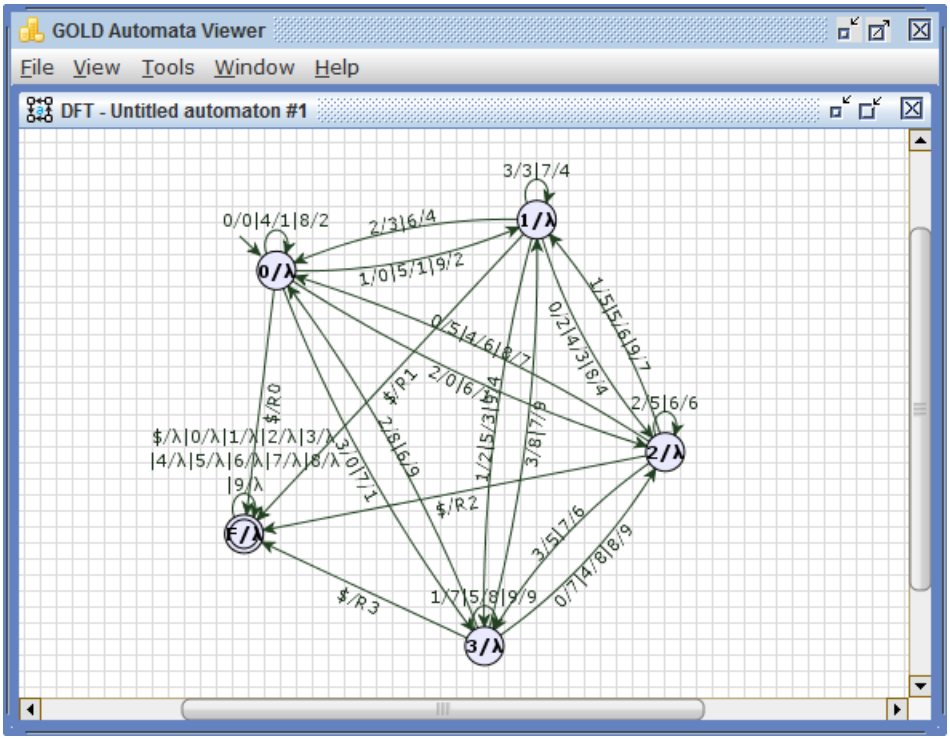
9.5.4.1. Definición de autómatas

Código 9.129. *Definición de un autómata con respuesta, que divide por 4 en base 10.*

```
using gold.structures.automaton.*
using gold.visualization.automaton.*
var n:int(4)
procedure main(args:String[]) begin // Dividir por n en base 10
  Q,Σ,Σ′,q0,F,g:={x|0≤x<n}∪{"F"},"0123456789$","0123456789R",0,{"F"},NIL
  GAutomataFrame.show(GDeterministicTransducer(Q,Σ,Σ′,q0,F,δ,g,h))
end
```

```
function δ(q,σ) begin
  if σ='$' or q="F" then
    return "F"
  else
    return (q*10+(σ-'0'))%n
  end
end
function h(q,σ) begin
  if q="F" then
    return λ
  elseif σ='$' then
    return "R"+q
  else
    return (q*10+(σ-'0'))÷n
  end
end
```

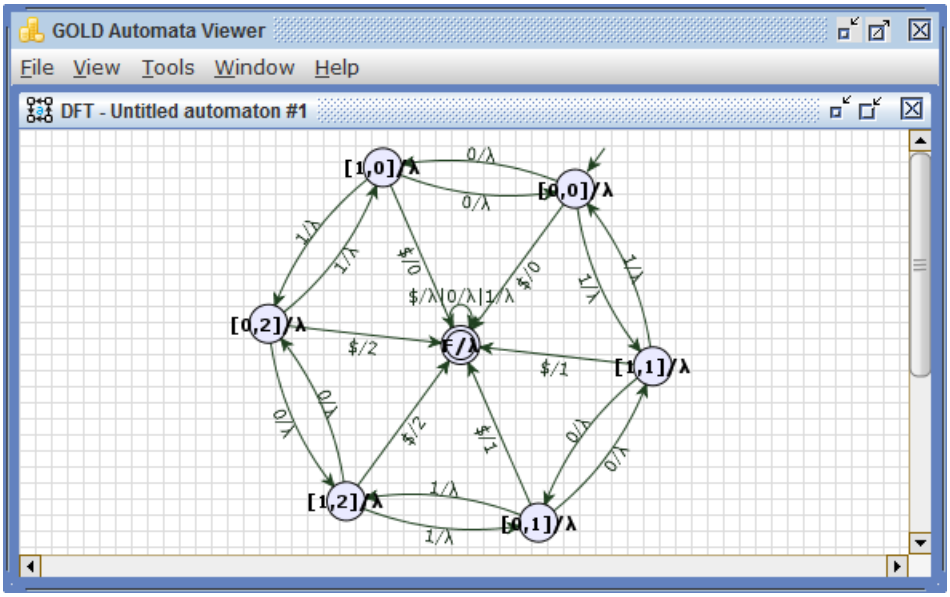
Figura 9.9. *Ventana gráfica desplegada después de ejecutar el programa 9.129.*



Código 9.130. *Definición de un autómata con respuesta, que calcula el residuo al dividir por 3 en base 2.*

```
using gold.**
procedure main(args:String[]) begin // Residuo al dividir por 3 en base 2
  Q,Σ,Σ',q0,F,g := {⟨p,r⟩|0≤p≤1,0≤r≤2}∪{"F"},"01$","012",⟨0,0⟩,{"F"},NIL
  GAutomataFrame.show(GDeterministicTransducer(Q,Σ,Σ',q0,F,δ,g,h))
end
δ(s,σ) := "F"
δ(⟨p,r⟩,σ) := σ='$'?"F":⟨(p+1)%2,(2|p?r+(σ-'0'):r+3-(σ-'0'))%3⟩
h(s,σ) := λ
h(⟨p,r⟩,σ) := σ='$'?r:λ
```

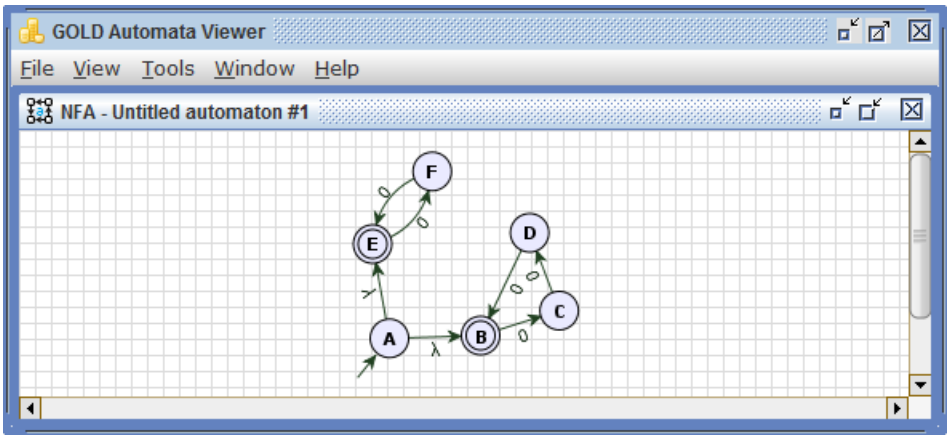
Figura 9.10. *Ventana gráfica desplegada después de ejecutar el programa 9.130.*



Código 9.131. *Definición de un autómata no determinístico que reconoce cadenas con una cantidad de ceros que es múltiplo de 2 o múltiplo de 3.*

```
1  using gold.structures.automaton.*
2  using gold.visualization.automaton.*
3  procedure main(args:String[]) begin
4    Q,Σ,q0,F := 'A'..'F',{'0'},'A',{'B','E'}
5    M := GNondeterministicAutomaton(Q,Σ,q0,F)
6    M.addDelta('A','λ','B')
7    M.addDelta('B','0','C')
8    M.addDelta('C','0','D')
9    M.addDelta('D','0','B')
10   M.addDelta('A','λ','E')
11   M.addDelta('E','0','F')
12   M.addDelta('F','0','E')
13   GAutomataFrame.show(M)
14 end
```

Figura 9.11. *Ventana gráfica desplegada después de ejecutar el programa 9.131.*



9.5.4.2. Unión de autómatas determinísticos finitos

Código 9.132. *Algoritmo de unión de autómatas determinísticos finitos.*

```
using gold.**
// Reference: Introduction to the theory of computation, Michael Sipser, 2nd edition
function union(A:IAutomaton,B:IAutomaton):IAutomaton begin
  Qa,Σa,q0a,Fa := A.getStates(),A.getAlphabet(),A.getInitialState(),A.getAcceptStates()
  Qb,Σb,q0b,Fb := B.getStates(),B.getAlphabet(),B.getInitialState(),B.getAcceptStates()
  assert Σa=Σb ∧ A.isDeterministic() ∧ B.isDeterministic()
  Q,Σ,q0,F := Qa×Qb,Σa,⟨q0a,q0b⟩,(Fa×Qb)∪(Qa×Fb)
  M := GDeterministicAutomaton(Q,Σ,q0,F)
  for each ⟨x,y⟩∈Q do
    for each σ∈Σ do
      δx,δy := A.getDelta(x,σ),B.getDelta(y,σ)
      M.setDelta(⟨x,y⟩,σ,⟨δx,δy⟩) // δ(⟨x,y⟩,σ)=⟨δx,δy⟩
    end
  end
  return M
end
```

9.5.4.3. Intersección de autómatas determinísticos finitos

Código 9.133. *Algoritmo de intersección de autómatas determinísticos finitos.*

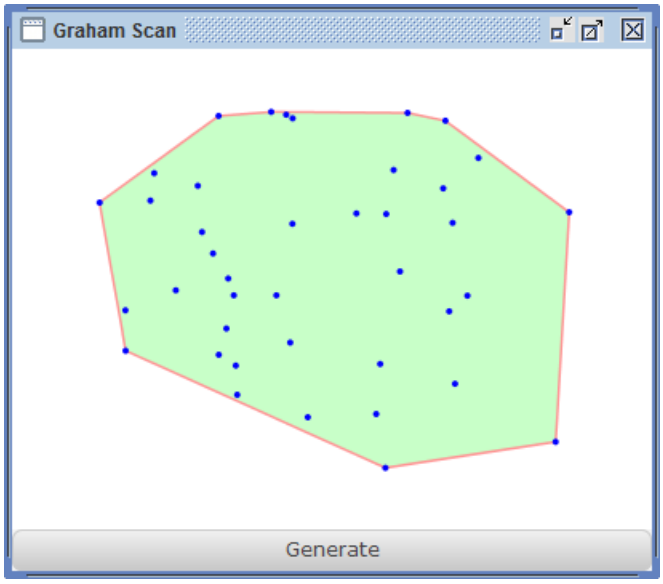
```
using gold.**
// Reference: Introduction to the theory of computation, Michael Sipser, 2nd edition
function intersection(A:IAutomaton,B:IAutomaton):IAutomaton begin
  Qa,Σa,q0a,Fa := A.getStates(),A.getAlphabet(),A.getInitialState(),A.getAcceptStates()
  Qb,Σb,q0b,Fb := B.getStates(),B.getAlphabet(),B.getInitialState(),B.getAcceptStates()
  assert Σa=Σb ∧ A.isDeterministic() ∧ B.isDeterministic()
  Q,Σ,q0,F := Qa×Qb,Σa,⟨q0a,q0b⟩,Fa×Fb
  M := GDeterministicAutomaton(Q,Σ,q0,F)
  for each ⟨x,y⟩∈Q do
    for each σ∈Σ do
      δx,δy := A.getDelta(x,σ),B.getDelta(y,σ)
      M.setDelta(⟨x,y⟩,σ,⟨δx,δy⟩) // δ(⟨x,y⟩,σ)=⟨δx,δy⟩
    end
  end
  return M
end
```

9.6. Otras aplicaciones

9.6.1. Interfaces gráficas

Para ilustrar el desarrollo de interfaces gráficas en *GOLD* a través de la librería *Swing* se implementó una aplicación de escritorio básica que encuentra la envolvente convexa (*convex hull*) de una nube de puntos generada al azar, usando el método de Graham (véase la sección §9.4.3.2). Dentro del código fuente se embebe código nativo *Java* (`/?...?/`) para implementar un `ActionListener` que especifica la acción que se debe realizar cuando se haga clic sobre el botón *Generate* de la ventana. Hay que saber que el nombre del programa es `ConvexHullApplication.gold`, pues éste se necesita para referenciar la clase interna declarada en el código nativo (`ConvexHullApplication@GenerateActionListener`).

Figura 9.12. *Ventana gráfica desplegada después de ejecutar el programa 9.102.*



Código 9.134. *Aplicación de escritorio que muestra gráficamente los resultados del método de Graham.*

```
1  using java.awt.*
2  using java.awt.event.*
3  using java.awt.image.*
4  using javax.swing.*
5  using gold.structures.list.*
6  using gold.swing.util.*
7  using gold.visualization.util.*
8  sgn(x) := |x|<1e-13?0:(x<0?-1:+1)
9  norm2(⟨x,y⟩) := x^2+y^2
10 cruz(⟨x1,y1⟩,⟨x2,y2⟩) := x1·y2-x2·y1
11 cruz(a,b,c) := cruz(a,b)+cruz(b,c)+cruz(c,a)
12 δ(a,b) := sgn(sgn(cruz(b,a))≠0?cruz(b,a):norm2(a)-norm2(b))
13 function grahamScan(points) begin // Graham's Scan
14   u,v := NIL,NIL
15   for each ⟨x,y⟩∈points do
16     if u=NIL∨y<v∨(y=v∧x>u) then u,v := x,y end
17   end
18   points,r := GCollections.sort(⟨⟨x-u,y-v⟩|⟨x,y⟩∈points⟩,δ),GArrayList()
19   for each p∈points do
20     while |r|≥2∧sgn(cruz(r[|r|-1],p,r[|r|-2]))≤0 do
21       r.removeLast()
```

```
    end
    r.addLast(p)
  end
  return ⟨⟨x+u,y+v⟩|⟨x,y⟩∈r⟩
end
var W:int(400),H:int(300),IMAGE:GImage(W,H),LABEL:JLabel()
generatePoints(n):=⟨⟨W/8+Math.random()·W·6/8,H/8+Math.random()·H·6/8⟩|1≤i≤n⟩
procedure refresh() begin
  var graphics:Graphics2D
  points:=generatePoints(40)
  convexHull:=grahamScan(points)
  IMAGE.clean(Color.WHITE)
  graphics:=IMAGE.createQualityGraphics()
  graphics.setStroke(BasicStroke(1.5f))
  graphics.setColor(Color(200,255,200))
  graphics.fill(GShapes.polygon(GShapes.points(convexHull)))
  graphics.setColor(Color(255,150,150))
  graphics.draw(GShapes.polygon(GShapes.points(convexHull)))
  graphics.setColor(Color.BLUE)
  for each ⟨x,y⟩∈points do
    graphics.fill(GShapes.circle(x,y,2.0))
  end
  LABEL.setIcon(ImageIcon(IMAGE))
end
procedure main(args:String[]) begin
  frame,button:=JFrame("Graham Scan"),JButton("Generate")
  button.addActionListener(ConvexHullApplication@GenerateActionListener())
  refresh()
  frame.getContentPane().add(LABEL,BorderLayout.CENTER)
  frame.getContentPane().add(button,BorderLayout.SOUTH)
  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
  frame.pack()
  GUtilities.show(frame)
end
/?
public static class GenerateActionListener implements ActionListener {
  public void actionPerformed(ActionEvent e) {
    refresh();
  }
}
?/
```

9.6.2. Rutinas de entrada/salida

Como dentro de *GOLD* se puede usar cualquier clase *Java*, entonces se tiene disponible la librería de entrada/salida suministrada en el paquete `java.io`. En particular, se pueden utilizar las clases `BufferedReader`, `BufferedWriter` y `Scanner`. Para ilustrar el manejo de la entrada/salida y la creación de grafos, se presentan a continuación programas que solucionan tres ejercicios de competencias de programación: *Edgetown's Traffic Jams* de la competencia colombiana de programación del año 2011, *Angry Programmer* de la competencia colombiana de programación del año 2008, y *Lazy Jumping Frog* de la competencia suramericana de programación del año 2006. Los enunciados de los ejercicios se pueden encontrar en el directorio `/Data/Tests/Contests/data/` de la distribución.

Código 9.135. *Programa que resuelve el ejercicio Edgetown's Traffic Jams.*

```
@SuppressWarnings("types")
using java.io.*
```

```
var tm:long(System.currentTimeMillis())
procedure main(args:String[]) begin
  var m1:double[][],m2:double[][]
  br := BufferedReader(FileReader("data/edgetown.in"))
  while true do
    n := (br.readLine() as int)
    if n=0 then break end
    m1,m2 := read(br,n),read(br,n)
    line := br.readLine().split(" ")
    A,B := (line[0] as int),(line[1] as int)
    print (∀i,j|0≤i<n,0≤j<n:m2[i][j]≤A·m1[i][j]+B)?"Yes":"No"
  end
  print "Execution time : ",(System.currentTimeMillis()-tm),"ms"
end
function read(br,n) begin
  var m:double[n][n]
  for i := 0 to n-1 do
    for j := 0 to n-1 do
      m[i][j] := i=j?0:∞
    end
  end
  for i := 0 to n-1 do
    line := br.readLine().split(" ")
    for k := 1 to |line|-1 do
      m[i][(line[k] as int)-1]=1
    end
  end
  GGraphs.floydWarshall(m)
  return m
end
```

Código 9.136. *Programa que resuelve el ejercicio Angry Programmer.*

```
using gold.**
var tm:long(System.currentTimeMillis())
procedure main(args:String[]) begin
  sc := java.util.Scanner(java.io.File("data/angry.in"))
  while TRUE do
    M,W := sc.nextInt(),sc.nextInt()
    if M=0 ∧ W=0 then
      break
    end
    G := GDirectedGraph(0..M*2-1)
    for k := 1 to M-2 do
      i,c := sc.nextInt()-1,sc.nextLong()
      G.addEdge(i*2,i*2+1,c)
      G.addEdge(i*2+1,i*2,c)
    end
    for k := 0 to W-1 do
      i,j,c := sc.nextInt()-1,sc.nextInt()-1,sc.nextLong()
      G.addEdge(i*2,j*2+1,c)
      G.addEdge(j*2,i*2+1,c)
    end
    print Math.round(GGraphs.edmondsKarp(G,0*2,(M-1)*2+1))
  end
  print "Execution time : ",(System.currentTimeMillis()-tm),"ms"
end
```

Código 9.137. *Programa que resuelve el ejercicio Lazy Jumping Frog.*

```
@SuppressWarnings("types")
using gold.**
var tm:long(System.currentTimeMillis())
var C,R,matrix:double[][],water
procedure main(args:String[]) begin
  matrix:= double[][][[7,6,5,6,7],[6,3,2,3,6],[5,2,0,2,5],[6,3,2,3,6],[7,6,5,6,7]]
  sc:= java.util.Scanner(java.io.File("data/frog.in"))
  while TRUE do
    C,R:= sc.nextInt(),sc.nextInt()
    if C=0 and R=0 then
      break
    end
    frog,toad,W:=⟨sc.nextInt(),sc.nextInt()⟩,⟨sc.nextInt(),sc.nextInt()⟩,sc.nextInt()
    water:=∅
    for w:=1 to W do
      C1,R1,C2,R2:= sc.nextInt(),sc.nextInt(),sc.nextInt(),sc.nextInt()
      water.union((C1..C2)×(R1..R2)) // water:=water∪((C1..C2)×(R1..R2)) is very slow
    end
    if (frog in water) or (toad in water) then // i.e., if frog∈water ∨ toad∈water
      print "impossible"
    else
      graph:= GImplicitUndirectedGraph((1..C)×(1..R),successors,cost)
      result:= GGraphs.dijkstra(graph,frog,toad)
      print result=∞?"impossible":⌊result+0.5⌋
    end
  end
  print "Execution time : ",(System.currentTimeMillis()-tm),"ms"
end
successors(⟨c,r⟩):=(((c-2↑1)..(c+2↓C))×((r-2↑1)..(r+2↓R)))\{⟨c,r⟩}\water
cost(⟨c1,r1⟩,⟨c2,r2⟩):= matrix[|c2-c1|+2][|r2-r1|+2]
```

Capítulo 10

Conclusiones

El resultado final de este proyecto es un lenguaje de programación imperativo denominado *GOLD* (*Graph Oriented Language Domain* por sus siglas en inglés), dotado de un entorno de programación completo y potente, que puede ser estudiado como un lenguaje de propósito general que facilita la escritura de rutinas que utilizan intensivamente objetos matemáticos expresados en la notación acostumbrada en los libros de texto. También puede verse como un lenguaje de propósito específico que facilita la escritura de algoritmos sobre estructuras de datos avanzadas como árboles, grafos y autómatas a través de una sintaxis muy cercana al pseudocódigo trabajado en la referencia *Introduction to Algorithms* de Thomas Cormen et al. [1].

Este capítulo resume las características del producto implementado, y propone una serie de requerimientos que pueden ser asumidos como trabajo futuro en las siguientes versiones del lenguaje. En la sección §10.1 (*Trabajo desarrollado*) se recapitula el proceso realizado durante el proyecto de tesis comentando las características del software, haciendo énfasis en sus ventajas (*pros*). Luego, en la sección §10.2 (*Trabajo futuro*), se analizan las deficiencias y desventajas (*contras*) como una oportunidad para mejorar la herramienta.

10.1. Trabajo desarrollado

Aunque en la literatura moderna se ha investigado mucho sobre la algorítmica especializada en manipular estructuras de datos como los grafos, en el análisis del estado del arte (véase el capítulo §3) se concluyó que en la actualidad no existe una herramienta adecuada para facilitar el desarrollo de grandes proyectos que requieran la codificación de algoritmos expresados en términos de objetos matemáticos estudiados en el cálculo proposicional, el cálculo de predicados, la teoría de números, la teoría de secuencias, la teoría de conjuntos, la teoría de grafos y la teoría de autómatas, entre otros. En la mayoría de los casos, los desarrolladores de estas aplicaciones deben usar una determinada librería bajo un lenguaje de propósito general, o herramientas limitadas para la definición de grafos mediante lenguajes de propósito específico o interfaces gráficas.

Los tres proyectos que antecedieron este trabajo de tesis fueron, en orden cronológico y de forma progresiva, *CSet: un lenguaje para composición de conjuntos* [2] de Víctor Hugo Cárdenas (2008), *GOLD: un lenguaje orientado a grafos y conjuntos* [3] de Luis Miguel Pérez (2009) y *GOLD+: lenguaje de programación para la manipulación de grafos: extensión de un lenguaje descriptivo a un lenguaje de programación* [4] de Diana Mabel Díaz (2010). Durante el estudio de los antecedentes (véase el capítulo §2) se analizaron detalladamente los productos desarrollados en cada uno de estos proyectos, especialmente el lenguaje *GOLD+* [4], concluyendo que los resultados eran insuficientes para cumplir con la meta de ofrecer un lenguaje expresivo que tuviera las bondades de un lenguaje de propósito general orientado a objetos como *Java* y *C++*, y a la vez facilitara el uso de objetos matemáticos sofisticados, en particular los grafos. *GOLD+* [4] permite la manipulación algorítmica de grafos con un lenguaje de programación incipiente que está basado en un conjunto limitado de instrucciones de control que no es lo

suficientemente expresivo pues restringe arbitrariamente la forma de escribir comandos y además no proporciona una librería que los desarrolladores puedan usar para manipular objetos distintos a los grafos. En su limitado campo de aplicación, *GOLD+* [4] únicamente permite describir grafos y realizar determinadas operaciones básicas entre éstos, no siendo lo suficientemente potente como para apoyar la implementación exitosa de algoritmos clásicos como el de Dijkstra. Todo lo anterior reveló el principal problema que tuvo cada precursor de este proyecto (exceptuando *CSet*): su diseño constituía un lenguaje de propósito específico limitado que únicamente estaba enfocado en los grafos, prohibiendo el uso de librerías externas y de otros objetos matemáticos importantes.

Pensando en el diseño de un lenguaje de programación de propósito general cuya sintaxis evoque el estilo de codificación de pseudocódigos como los trabajados en el libro *Introduction to Algorithms* de Thomas Cormen et al. [1], fomentando la utilización de entidades matemáticas formales como los grafos y otras estructuras de datos, se enunció una serie de requerimientos básicos (véase el capítulo §5) clasificados usando el estándar internacional *ISO/IEC* 9126 [52] y los criterios expuestos en los textos *Programming languages : design and implementation* de Pratt y Zelkowitz [41], y *Programming Language Design Concepts* de Watt [40]. Tales requerimientos guiaron por completo el diseño e implementación del lenguaje *GOLD*, influyendo sobre muchas decisiones que se debieron tomar como la escogencia de herramientas, recordando que la inspiración nunca dejaba de ser el texto de Cormen.

Para fomentar la utilización de librerías externas como el *API* estándar de *Java* y la manipulación de cualquier estructura de datos, fue necesario rediseñar *GOLD* como un lenguaje de propósito general completamente nuevo que sirviera para resolver problemas sobre una gran cantidad de dominios. Al mismo tiempo, *GOLD* debe actuar como un lenguaje de propósito específico que facilite la programación de algoritmos sobre el dominio particular de los grafos y otras estructuras de datos fundamentales. Estos hechos definieron los lineamientos generales que fueron tenidos en cuenta al momento de diseñar el lenguaje, a través de la definición de su sintaxis, su semántica y su pragmática (véase el capítulo §7). Por lo tanto, para que el lenguaje pudiera ser integrado en grandes proyectos de software que requirieran una manipulación exhaustiva de objetos matemáticos, complementándose con un lenguaje de propósito general orientado a objetos como *Java* o *C++*, fue necesario descartar el enfoque tomado en *GOLD+* que sometía los programas a un proceso de interpretación que ejecutaba cada instrucción en una máquina abstracta restringida. Por esta razón se decidió someter el lenguaje a un proceso de compilación capaz de transformar los programas *GOLD* en código fuente escrito en un lenguaje de programación orientado a objetos con bases fuertes, en este caso *Java*.

Durante la etapa de implementación (véase el capítulo §8) se resolvieron los detalles técnicos relacionados con la implantación del producto, como el desarrollo de su entorno de desarrollo integrado (*IDE*) y de su compilador, que incluye el analizador léxico-sintáctico, el modelo semántico y el traductor a código *Java*. Al someter los programas *GOLD* a un proceso de compilación capaz de transformarlos en archivos *Java*, se faculta a los programadores para que puedan mezclar en sus proyectos código *GOLD* con código *Java*, potenciando enormemente la aplicación de ambos lenguajes puesto que cada uno se beneficia de las capacidades del otro. A grandes rasgos, *GOLD* se beneficia de *Java* aprovechando su *API* estándar y todos los conceptos de la programación orientada a objetos; por otro lado, *Java* se beneficia de *GOLD* aprovechando su versatilidad para expresar y manipular objetos matemáticos de diversos dominios con una sintaxis cercana al pseudocódigo. Lo mejor de ambos mundos es brindado al usuario final, quien termina favoreciéndose. Más aún, cualquier librería *Java* (especialmente las relacionadas con estructuras de datos) puede eventualmente convertirse en un aliado de *GOLD* más que en un rival, porque ambas estarían en capacidad de establecer una relación simbiótica de mutualismo en la que *GOLD* facilite la manipulación de la librería con su sintaxis y la librería enriquezca a *GOLD* con sus servicios.

La escogencia de *Java* como lenguaje anfitrión redundó en beneficios para *GOLD* a través del uso de herramientas y librerías externas específicamente diseñadas para *Java* (véase el capítulo §6). La implementación del producto fue realizada completamente en el *framework Xtext* [6], que fue usado para generar automáticamente el analizador léxico-sintáctico y el modelo semántico del lenguaje. Además, *Xtext* [6] apoyó sustancialmente la creación del *IDE* embebido dentro del entorno de programación *Eclipse* [7], que es familiar a muchos ingenieros que trabajan

sobre la plataforma *Java*. Sin la utilización de *Xtext* y *Eclipse*, la implementación del lenguaje posiblemente hubiese necesitado la participación de varios desarrolladores durante varios años. Adicionalmente, como desde cualquier programa *GOLD* se puede utilizar cualquier clase *Java*, se ahorró una gran cantidad de trabajo mediante la importación de las librerías externas *JUNG* [21] para apoyar el proceso de visualización de las estructuras de datos, *JGraphT* [22] y la librería de referencia del libro de Cormen [23] para implementar algunas estructuras de datos, y *Apfloat* [53] para incluir tipos de datos que representan números de precisión arbitraria. Finalmente, para enriquecer la biblioteca de clases disponible y centralizar el conocimiento relacionado con las estructuras de datos, se programó un conjunto sofisticado de clases bajo un paquete denominado `gold` que ofrece a los desarrolladores un sinnúmero de implementaciones de las estructuras de datos más básicas (incluyendo listas, conjuntos, bolsas, pilas, colas, bicolas, montones, árboles, asociaciones llave-valor, grafos y autómatas), una colección de visualizadores sofisticados para presentar y manipular gráficamente algunas de éstas (específicamente árboles, *Quadtrees*, *Tries*, grafos y autómatas), y una serie de rutinas estáticas para facilitar la utilización de la librería.

La expresividad del lenguaje se puso a prueba con algunos ejemplos (véase el capítulo §9), ilustrando los comandos y las sentencias definidas en su gramática a través de la implementación de varios algoritmos clásicos sobre teoría de grafos y otras estructuras de datos. Aunque no se realizaron formalmente, es necesario diseñar e implementar una colección de pruebas unitarias con la ayuda de la librería *JUnit* [56] para revisar el funcionamiento de algunos módulos por separado, y pruebas de integración para garantizar el funcionamiento de todo el sistema. Las pruebas deberían poderse ejecutar automáticamente en lote para detectar fallos o inconsistencias luego de cualquier modificación que sufra el software, considerando tanto el núcleo del lenguaje (el analizador léxico-sintáctico, el modelo semántico y el traductor a código *Java*) como la librería de clases suministrada por *GOLD*. Finalmente, se comentaron algunos aspectos técnicos y se complementó el diseño con algunos diagramas *UML* (véase el anexo A) editados en *ArgoUML* [84] .

Sin duda alguna, *GOLD* es un lenguaje que permite a los investigadores e ingenieros manipular de una manera sencilla objetos matemáticos como estructuras de datos, acercando el mundo de la programación a un nicho de usuarios que no necesitan ser expertos programadores, tanto en entornos académicos como en entornos empresariales. Que el lenguaje sirva para codificar algoritmos de una manera cómoda sobre el dominio específico a los grafos es una mera consecuencia colateral de haberlo diseñado como un lenguaje de programación general con una sintaxis estilo pseudocódigo que permite la utilización de objetos matemáticos clásicos como las estructuras de datos (en particular, los grafos). En todo caso, no hay que olvidar las raíces del proyecto, pues los grafos fueron precisamente los objetos que dieron lugar a la motivación que desencadenó el nacimiento de *GOLD* como trabajo de investigación. Tampoco hay que olvidar que *GOLD* elevó su potencial gracias a *Java*, facilitando actividades no mencionadas anteriormente, como el desarrollo de interfaces gráficas y la manipulación de archivos.

10.2. Trabajo futuro

GOLD es un lenguaje con un amplio potencial que debe ser explotado al máximo. Para mejorar algunos componentes que quedaron imperfectos, completar algunos requerimientos que no se cumplieron y suplir algunas funcionalidades adicionales que son deseables, buscando la evolución futura del lenguaje (considerando tanto su núcleo como su *IDE*), se sugiere atacar tres flancos:

- Documentación:
  - Publicar oficialmente la herramienta en un sitio *WEB* centralizado que distribuya el *plug-in* listo para ser instalado, brinde soporte en línea a la comunidad, y provea actualizaciones periódicas, todo bajo un proceso de mantenimiento comandado por la Institución.
  - Redactar un manual de usuario y un tutorial de programación completo, claro y autocontenido para enseñar a los desarrolladores la sintaxis, semántica y pragmática de *GOLD*, así como el uso de su entorno de desarrollo integrado (*IDE*).

  - Documentar clara y completamente la librería de clases suministrada por *GOLD* usando el estándar *Javadoc* para generar el *API* en formato *HTML*, de tal forma que pueda ser publicado como material de apoyo para quienes van a utilizar el lenguaje.
  - Documentar clara y completamente la implementación interna de *GOLD* usando el estándar *Javadoc* para generar el *API* en formato *HTML*, de tal forma que pueda ser distribuido como material de apoyo para quienes van a mantener la infraestructura del lenguaje.
  - Complementar la batería de ejemplos suministrados en el capítulo §9 con aplicaciones a otras ramas de la ciencia, incluyendo la ingeniería industrial y la ingeniería civil, para ilustrar el gran potencial que tiene *GOLD* como lenguaje de propósito general para la solución de problemas sobre una diversidad de dominios.
  - Diseñar e implementar pruebas unitarias para revisar el funcionamiento de cada módulos por separado y pruebas de integración para garantizar el funcionamiento de todo el sistema, considerando tanto el núcleo del lenguaje (el analizador léxico-sintáctico, el modelo semántico y el traductor a código *Java*) como la librería de clases suministrada.

- Implementación:

  - Extender la sintaxis y semántica del lenguaje para permitir la escritura de comandos de la forma *try-catch*, la declaración de clases e interfaces, y la declaración de entidades genéricas (*genericidad*), acercando el lenguaje al paradigma de la programación orientada a objetos.
  - Extender el lenguaje para permitir la implementación de procesos concurrentes a través de hilos (*threads*) y de métodos de sincronización diseñados para coordinar el paralelismo en tiempo de ejecución, acercando el lenguaje al paradigma de la programación concurrente.
  - Enriquecer la librería de clases con aplicaciones gráficas más potentes que actúen como herramientas de escritorio sofisticadas para la edición y manipulación de estructuras de datos, compitiendo con los productos de software existentes para la simulación de procesos sobre grafos y autómatas.
  - Incluir máquinas como Redes de Petri y Máquinas de Turing, que complementen las implementaciones existentes sobre autómatas, autómatas con respuesta y autómatas de pila, buscando que *GOLD* pueda ser utilizado como la herramienta predilecta en cursos de lenguajes.
  - Implementar procesos de optimización durante la síntesis del código ejecutable, buscando que el compilador sea capaz de producir programas más eficientes que no sufran de los retrasos causados por el uso indiscriminado de la *reflexión* de *Java* [62].
  - Implementar un compilador que sea capaz de traducir programas *GOLD* en programas escritos en *C++* para que los programadores puedan mezclar código fuente de ambos lenguajes en sus proyectos, aprovechando la librería estándar de *C++* (*STL*: *Standard Template Library*) con las bondades de *GOLD*.
  - Implementar nuevas estructuras de datos que complementen las ya proporcionadas por *GOLD*.
  - Implementar rutinas especializadas en importar grafos de fuentes de datos externas para su posterior procesamiento en *GOLD*.
  - Implementar el estándar *JSR 233* [33] para permitir el uso de *GOLD* como un lenguaje de *scripting* embebido dentro de *Java*.

- Investigación:

  - Dotar al lenguaje (o a algún subconjunto razonable de éste) de una semántica axiomática que enuncie una colección de teoremas de corrección con los que se pueda verificar programas escritos en *GOLD*, demostrando formalmente la corrección de éstos (su eficacia).
  - Estudiar el impacto que tendría la incorporación de instrucciones no determinísticas sobre *GOLD*, evocando el *Lenguaje de Comandos Guardados GCL*.

- Analizar distintas formas en las que *GOLD* se podría acercar a otros paradigmas, como la programación orientada a objetos, la programación concurrente y la programación funcional.
- Explorar el uso de otros lenguajes imperativos o de lenguajes funcionales como huéspedes de *GOLD* durante el proceso de traducción.

Parte IV

Apéndices

Apéndice A

Documentación técnica

En este capítulo se encuentra la documentación técnica del producto, incluyendo la gramática del lenguaje en notación *EBNF* [5] y en notación *Xtext* [6], algunos diagramas de diseño en notación *UML* (*Unified Modeling Language*), y algunas instrucciones para generar e instalar el *plug-in* en *Eclipse* [7].

A.1. Gramática

A.1.1. Gramática *EBNF*

Para definir la gramática de *GOLD* en notación *EBNF* [5] se usó la variante establecida por la *W3C* [46] en vez de la variante establecida por el estándar *ISO/IEC* 14977 [45], porque la segunda resultaba menos legible. Simplificando la variante de la *W3C*, en cada regla de la gramática se define un símbolo en la forma `symbol::=expression`, donde `expression` es una expresión formada usando las siguientes sentencias [46] ordenadas de mayor a menor según precedencia, siendo posible la adición de comentarios delimitados entre las cadenas `/*` y `*/` (o entre la cadena `//` y el próximo fin de línea):

- `(`*E*`)` para reconocer la expresión *E*;
- `#x`*NNNN* para reconocer el carácter *Unicode* cuyo código hexadecimal es *NNNN*;
- `[#x`*NNNN*`-#x`*MMMM*`]` para reconocer el rango de caracteres *Unicode* cuyos códigos se encuentran entre el valor hexadecimal *NNNN* (inclusive) y el valor hexadecimal *MMMM* (inclusive);
- `"`*S*`"` para reconocer la cadena de texto *S*;
- `'`*S*`'` para reconocer la cadena de texto *S*;
- *E*`*` para reconocer cero o más ocurrencias de la expresión *E* (*cero o más veces E*);
- *E*`+` para reconocer una o más ocurrencias de la expresión *E* (*una o más veces E*);
- *E*`?` para reconocer *E* o nada (*cero o una vez E*), siendo *E* una expresión *opcional*;
- *E*`-`*F* para reconocer cualquier cadena de texto que sea reconocida por *E* pero no por *F* (*diferencia*), siendo *E* y *F* expresiones;
- *E* *F* para reconocer *E* seguido de *F* (*concatenación*), siendo *E* y *F* expresiones; y
- *E*`|`*F* para reconocer *E* o *F* pero no ambas (*alternación*), siendo *E* y *F* expresiones.

Código A.1. *Definición de la gramática de GOLD en la notación EBNF [46].*

```
// ************************************************************************************
// * TERMINAL SYMBOLS                                                                 *
// ************************************************************************************
ALL         ::= [#x0000-#xFFFF]
LF          ::= #x000A
CR          ::= #x000D
TAB         ::= #x0009
LETTER      ::= [#x0041-#x005A] | [#x0061-#x007A] | [#x0391-#x03A9] | [#x03B1-#x03C9]
DIGIT       ::= [#x0030-#x0039]
SUBINDEX    ::= [#x2080-#x2089]
HEX_DIGIT   ::= [#x0030-#x0039] | [#x0041-#x0046] | [#x0061-#x0066]
HEX_CODE    ::= 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT
ID          ::= '$'? (LETTER|'_') (LETTER|'_'|DIGIT|SUBINDEX|'')*
QN          ::= ID ('.' ID)* ('@' ID)*
CONSTANT    ::= 'TRUE'|'true'|'FALSE'|'false'|'NIL'|'nil'|'NULL'|'null'|'∅'|'𝐔'|'ε'|'λ'|'∞'
NUMBER      ::= DIGIT+ ('.' DIGIT+)? (('E'|'e') '-'? DIGIT+)?
                ('L'|'l'|'I'|'i'|'S'|'s'|'B'|'b'|'D'|'d'|'F'|'f'|'C'|'c')?
STRING      ::= '"' (('\'('b'|'t'|'n'|'f'|'r'|HEX_CODE|'"'|"'"|'\')) | (ALL-('\'|'"')))* '"'
CHARACTER   ::= "'" (('\'('b'|'t'|'n'|'f'|'r'|HEX_CODE|'"'|"'"|'\')) | (ALL-('\'|"'"))) "'"
JAVA_CODE   ::= '/?' ((ALL-'?') | ('?' (ALL-'/')))* '?/'
ML_COMMENT  ::= '/*' ((ALL-'*') | ('*' (ALL-'/')))* '*/'
SL_COMMENT  ::= ('⧐' | '//') (ALL-(CR|LF))*
WS          ::= (' '|TAB|CR|LF)+

// ************************************************************************************
// * NON TERMINAL SYMBOLS - PROGRAM STRUCTURE                                         *
// ************************************************************************************
GoldProgram ::= Annotation* Package? Import* (VariableDecl|ProcedureDecl|JAVA_CODE)*
Annotation  ::= '@' 'SuppressWarnings' '(' STRING ')'
              | '@' 'SuppressWarnings' '(' '{' (STRING (',' STRING)*)? '}' ')'
Package     ::= 'package' QN
Import      ::= ('import'|'include'|'using') QN ('.*'|'.**')?

// ************************************************************************************
// * NON TERMINAL SYMBOLS - TYPES                                                     *
// ************************************************************************************
Class ::= '𝔹'|'ℕ'|'ℤ'|'ℚ'|'ℝ'|'ℂ'
        | 'boolean'|'char'|'byte'|'short'|'int'|'long'|'float'|'double'
        | QN
Type  ::= Class ('[' ']')*

// ************************************************************************************
// * NON TERMINAL SYMBOLS - VARIABLE DECLARATIONS                                     *
// ************************************************************************************
Variable     ::= ID                                    // Non-typed variable
               | ID ':' Type                           // Typed variable
               | ID ':' Class '(' ExpressionList? ')'  // Idem, with class constructor call
               | ID ':' Class ('[' Expression ']')+    // Idem, with array constructor call
VariableDecl ::= 'var' Variable (',' Variable)*

// ************************************************************************************
// * NON TERMINAL SYMBOLS - PROCEDURE DECLARATIONS                                    *
// ************************************************************************************
Parameter    ::= ID (':' Type)? | '⟨' ID (',' ID)* '⟩'
Header       ::= ID '(' (Parameter (',' Parameter)*)? ')'
VoidReturn   ::= ':' 'void'
ReturnType   ::= ':' Type
```

```
ProcedureDecl ::= 'procedure' Header VoidReturn? 'begin' Command* 'end'  // Proper procedure
                | 'function'? Header ReturnType? 'begin' Command* 'end'  // Function
                | 'function'? Header ReturnType? (':='|'=') Expression   // Function macro

// ********************************************************************************************
// * NON TERMINAL SYMBOLS - COMMANDS (INSTRUCTIONS)                                           *
// ********************************************************************************************
Command        ::= VariableDecl               // Multiple variable declaration
                 | Skip                       // Empty instruction
                 | Abort                      // Abnormal termination instruction
                 | Call                       // Function/procedure/method invocation
                 | Assert                     // Assertion statement
                 | Assignment                 // Assignment instruction
                 | Swap                       // Swap instruction
                 | Conditional                // Conditional statement
                 | Repetition                 // Repetitive instruction
                 | Print                      // Print clause
                 | Error                      // Error clause
                 | Throw                      // Throw clause
                 | Return                     // Return clause
                 | Escape                     // Escape sequencer
                 | JAVA_CODE                  // Java native code
Skip           ::= 'skip'
Abort          ::= 'abort'
Call           ::= 'call'? Expression
Assert         ::= 'assert' Expression
Assignment     ::= AssignmentVars ('←'|':='|'=') ExpressionList
AssignmentVars ::= AssignmentVar (',' AssignmentVar)*
AssignmentVar  ::= ID                         // Variable access
                 | ID ('[' Expression ']')+   // Array/Structure access
                 | ID ':' Type                // Variable declaration + Variable access
Swap           ::= ('swap'|'exchange') SwapVar ('↔'|'with') SwapVar
SwapVar        ::= ID                         // Variable access
                 | ID ('[' Expression ']')+   // Array/Structure access
Conditional    ::= IfThenElse                 // If-then-else statement
                 | Switch                     // Switch statement
IfThenElse     ::= 'if' Expression 'then' Command*
                   ('elseif' Expression 'then' Command*)*
                   ('else' Command*)?
                   'end'
Switch         ::= 'switch' Expression 'begin'
                   ('case' Expression ':' Command*)+
                   ('default' ':' Command*)?
                   'end'
Repetition     ::= While                      // While statement
                 | DoWhile                    // Do-while statement
                 | Repeat                     // Repeat-until statement
                 | ForEach                    // For-each statement
                 | For                        // For statement
While          ::= 'while' Expression 'do' Command* 'end'
DoWhile        ::= 'do' Command* 'whilst' Expression
Repeat         ::= 'repeat' Command* 'until' Expression
ForEach        ::= 'for' 'each' ForEachVar ('∈'|'in') Expression 'do'
                   Command*
                   'end'
ForEachVar     ::= ID (':' Type)? | '⟨' ID (',' ID)* '⟩'
For            ::= 'for' Assignment ('to'|'downto') Expression ('by' Expression)? 'do'
                   Command*
                   'end'
```

```
Print          ::= 'print' ExpressionList
Error          ::= 'error' ExpressionList
Throw          ::= 'throw' Expression
Return         ::= 'return' Expression        // Return clause (expression)
                 | 'finalize'                 // Void return clause (nothing)
Escape         ::= 'break'                    // Break clause
                 | 'continue'                 // Continue clause

// ********************************************************************************************
// * NON TERMINAL SYMBOLS - EXPRESSIONS                                                       *
// ********************************************************************************************
/* EXPRESSION LIST ------------------------------------------------------------------------- */
ExpressionList: Expression (',' Expression)*
/* EXPRESSION ------------------------------------------------------------------------------ */
Expression ::= Condt
/* PRECEDENCE LEVEL  0 ~ NON ASSOCIATIVE --------------------------------------------------- */
Condt ::= Equiv
        | Equiv '?' Equiv ':' Equiv          // Conditional expression
/* PRECEDENCE LEVEL  1 ~ MUTUALLY ASSOCIATIVE ---------------------------------------------- */
Equiv ::= Implc
        | Equiv ('≡'|'⇔'|'eqv') Implc        // Equivalence / If and only if (booleans)
        | Equiv ('≢'|'⊕'|'xor') Implc        // Inequivalence / Exclusive or (booleans)
/* PRECEDENCE LEVEL  2 ~ MUTUALLY RIGHT ASSOCIATIVE ---------------------------------------- */
Implc ::= Consq
        | Consq '⇒' Implc                    // Implication (booleans)
        | Consq '⇏' Implc                    // Anti-implication (booleans)
/* PRECEDENCE LEVEL  3 ~ MUTUALLY LEFT ASSOCIATIVE ----------------------------------------- */
Consq ::= Disjc
        | Consq '⇐' Disjc                    // Consequence (booleans)
        | Consq '⇍' Disjc                    // Anti-consequence (booleans)
/* PRECEDENCE LEVEL  4 ~ ASSOCIATIVE ------------------------------------------------------- */
Disjc ::= Opcnj
        | Opcnj (('∨'|'or'|'||') Opcnj)+     // Disjunction (booleans)
        | Opcnj (('∧'|'and'|'&&') Opcnj)+    // Conjunction (booleans)
/* PRECEDENCE LEVEL  5 ~ MUTUALLY CONJUNCTIONAL -------------------------------------------- */
Opcnj ::= Opcns
        | Opcnj ('='|'==') Opcns             // Equality
        | Opcnj ('≠'|'!='|'<>') Opcns        // Inequality
        | Opcnj '<' Opcns                    // Less than (numbers)
        | Opcnj ('≤'|'<=') Opcns             // Less than or equal to (numbers)
        | Opcnj '>' Opcns                    // Greater than (numbers)
        | Opcnj ('≥'|'>=') Opcns             // Greater than or equal to (numbers)
        | Opcnj '|' Opcns                    // Divisibility (integer numbers)
        | Opcnj '∤' Opcns                    // Anti-divisibility (integer numbers)
        | Opcnj ('∈'|'in') Opcns             // Membership (sets, bags)
        | Opcnj '∉' Opcns                    // Anti-membership (sets, bags)
/* PRECEDENCE LEVEL  6 ~ MUTUALLY CONJUNCTIONAL -------------------------------------------- */
Opcns ::= Opseq
        | Opcns '⋈' Opseq                    // Disjoint operator (sets, bags)
        | Opcns '⊆' Opseq                    // Subset (sets, bags)
        | Opcns '⊈' Opseq                    // Not subset (sets, bags)
        | Opcns '⊇' Opseq                    // Superset (sets, bags)
        | Opcns '⊉' Opseq                    // Not superset (sets, bags)
        | Opcns ('⊂'|'⊊') Opseq              // Proper subset (sets, bags)
        | Opcns '⊄' Opseq                    // Not proper subset (sets, bags)
        | Opcns ('⊃'|'⊋') Opseq              // Proper superset (sets, bags)
        | Opcns '⊅' Opseq                    // Not proper superset (sets, bags)
/* PRECEDENCE LEVEL  7 ~ LEFT ASSOCIATIVE -------------------------------------------------- */
Opseq ::= Opapn
```

```
176         | Opapn ('◁' Opapn)+                // Prepend (sequences)
177         | Opapn ('^' Opapn)+                // Concatenation (sequences)
178 /* PRECEDENCE LEVEL  8 ~ RIGHT ASSOCIATIVE ------------------------------------------------ */
179 Opapn ::= Occur
180         | Occur '▷' Opapn                   // Append (sequences)
181 /* PRECEDENCE LEVEL  9 ~ NON ASSOCIATIVE --------------------------------------------------- */
182 Occur ::= Intvl
183         | Intvl '#' Intvl                   // Number of occurrences (bags, sequences)
184 /* PRECEDENCE LEVEL 10 ~ NON ASSOCIATIVE --------------------------------------------------- */
185 Intvl ::= Maxim
186         | Maxim '..' Maxim                  // Interval range (numbers)
187 /* PRECEDENCE LEVEL 11 ~ ASSOCIATIVE ------------------------------------------------------- */
188 Maxim ::= Addit
189         | Addit ('↑' Addit)+                // Maximum (numbers)
190         | Addit ('↓' Addit)+                // Minimum (numbers)
191 /* PRECEDENCE LEVEL 12 ~ LEFT MUTUALLY ASSOCIATIVE ----------------------------------------- */
192 Addit ::= Multp
193         | Addit '+' Multp                   // Addition (numbers)
194         | Addit '-' Multp                   // Subtraction (numbers)
195 /* PRECEDENCE LEVEL 13 ~ LEFT MUTUALLY ASSOCIATIVE ----------------------------------------- */
196 Multp ::= Opset
197         | Multp ('*'|'·') Opset             // Multiplication (numbers)
198         | Multp '/' Opset                   // Division (numbers)
199         | Multp ('%'|'mod') Opset           // Integer residue / Module (integer numbers)
200         | Multp ('÷'|'div') Opset           // Integer division / Quotient (integer numbers)
201         | Multp 'gcd' Opset                 // Greatest common divisor (integer numbers)
202         | Multp 'lcm' Opset                 // Least common multiple (integer numbers)
203 /* PRECEDENCE LEVEL 14 ~ LEFT ASSOCIATIVE --------------------------------------------------- */
204 Opset ::= Expon
205         | Expon ('∪' Expon)+                // Union (sets, bags)
206         | Expon ('∩' Expon)+                // Intersection (sets, bags)
207         | Expon ('\' Expon)+                // Difference (sets, bags)
208         | Expon ('△' Expon)+                // Symmetric difference (sets, bags)
209 /* PRECEDENCE LEVEL 15 ~ LEFT ASSOCIATIVE --------------------------------------------------- */
210 Expon ::= Carts
211         | Carts ('^' Carts)+                // Exponentiation (numbers)
212         | Carts ('^' Carts)+                // Cartesian power (sets)
213 /* PRECEDENCE LEVEL 16 ~ ASSOCIATIVE ------------------------------------------------------- */
214 Carts ::= Prefx
215         | Prefx ('×' Prefx)+                // Cartesian product (sets)
216 /* PRECEDENCE LEVEL 17 ~ UNARY PREFIX OPERATORS -------------------------------------------- */
217 Prefx ::= Suffx
218         | '+' Prefx                         // Unary plus sign (numbers)
219         | '-' Prefx                         // Unary minus sign (numbers)
220         | ('¬'|'not'|'!') Prefx             // Negation (booleans)
221         | '#' Prefx                         // Cardinality (sets, bags, sequences)
222         | '~' Prefx                         // Complement (sets)
223         | '℘' Prefx                         // Power set (sets)
224 /* PRECEDENCE LEVEL 18 ~ UNARY SUFFIX OPERATORS --------------------------------------------- */
225 Suffx ::= Brack
226         | Suffx '!'                         // Factorial (numbers)
227 /* PRECEDENCE LEVEL 19 ~ BRACKETS ---------------------------------------------------------- */
228 Brack ::= Basic
229         | '|' Expression '|'                // Cardinality (sets)
230         | '|' Expression '|'                // Absolute value (numbers)
231         | '⌊' Expression '⌋'                // Floor (numbers)
232         | '⌈' Expression '⌉'                // Ceiling (numbers)
233 /* PRECEDENCE LEVEL 20 ~ PRIMARY BASIC EXPRESSIONS ----------------------------------------- */
234 Basic ::= CONSTANT                                  // Constant literal
```

```
235         | NUMBER                                    // Number literal
236         | STRING                                    // String literal
237         | CHARACTER                                 // Character literal
238         | Type                                      // Type literal
239         | QN                                        // Variable/Procedure/Class access
240         | ID '(' ExpressionList? ')'                // Function application
241         | (Type '.')? 'class'                       // Type access
242         | Expression '.' ID                         // Field access
243         | Expression '.' ID '(' ExpressionList? ')'// Method call
244         | Expression ('[' Expression ']')+          // Array/Structure access
245         | 'new'? Class '(' ExpressionList? ')'      // Class constructor call
246         | 'new'? Class ('[' Expression ']')+        // Array constructor call
247         | 'new'? Type '⟦' ExpressionList? '⟧'       // Array constructor call
248         | '(' Expression ')'                        // Parenthesized expression
249         | '(' Expression (':'|'as') Type ')'        // Type conversion (cast)
250         | '⟦' ExpressionList? '⟧'                   // Array enumeration
251         | '{' ExpressionList? '}'                   // Set enumeration
252         | '⦃' ExpressionList? '⦄'                   // Bag enumeration
253         | '⟨' ExpressionList? '⟩'                   // List/Sequence enumeration
254         | '{' Body '|' Range '}'                    // Set comprehension
255         | '⦃' Body '|' Range '⦄'                    // Bag comprehension
256         | '⟨' Body '|' Range '⟩'                    // List/Sequence comprehension
257         | Gries                                     // Gries/Schneider-style quantification
258
259 // ***************************************************************************************
260 // * NON TERMINAL SYMBOLS - EXPRESSIONS - QUANTIFICATIONS                                *
261 // ***************************************************************************************
262 /* GRIES/SCHNEIDER-STYLE QUANTIFICATIONS ------------------------------------------- */
263 Gries ::= '(' 'Σ' Body '|' Range ')'                // Summation quantifier
264         | '(' 'Π' Body '|' Range ')'                // Product quantifier
265         | '(' '↓' Body '|' Range ')'                // Minimum quantifier
266         | '(' '↑' Body '|' Range ')'                // Maximum quantifier
267         | '(' '∩' Body '|' Range ')'                // Intersection quantifier
268         | '(' '∪' Body '|' Range ')'                // Union quantifier
269         | '(' '∀' Dummies '|' Range ':' Body ')'    // Universal quantifier
270         | '(' '∃' Dummies '|' Range ':' Body ')'    // Existential quantifier
271 /* QUANTIFICATION COMPONENTS ------------------------------------------------------- */
272 Dummies   ::= ID (',' ID)*
273 Body      ::= Expression
274 Range     ::= Fragment (',' Fragment)*
275 Fragment  ::= Condition                                      // Boolean condition
276             | DummyDecl                                     // Dummy declaration
277 Condition ::= '[' Expression ']'
278 DummyDecl ::= ID '=' Opseq                                  // Equality
279             | '⟨' ID (',' ID)* '⟩' '=' Opseq                // Equality
280             | ID ('∈'|'in') Opseq                           // Membership
281             | '⟨' ID (',' ID)* '⟩' ('∈'|'in') Opseq         // Membership
282             | ID '⊆' Opseq                                  // Subset
283             | ID ('⊂'|'⊊') Opseq                            // Proper subset
284             | Opseq ('<'|'≤'|'<=') ID ('<'|'≤'|'<=') Opseq  // Integer range
```

A.1.2. Gramática *Xtext*

La gramática *EBNF* definida en la sección §A.1.1 se implementó en *Xtext* [6] factorizando por la izquierda cada regla de producción (para evitar el *backtracking*) y enriqueciendo la sintaxis con los distintos mecanismos proporcionados por *Xtext* para guiar la generación automática del modelo semántico del lenguaje. El validador de código descrito en la sección §8.1.11 es responsable de revisar las reglas sintácticas consignadas en la gramática *EBNF* que no están explícitamente definidas en la gramática *Xtext*.

Código A.2. *Implementación de la gramática en Xtext [6].*

```
// -------------------------------------------------------------------------------------------
// GOLD DSL GRAMMAR DEFINITION
// -------------------------------------------------------------------------------------------
// Version: 3.0.10 (2012/01/25 20:05)
// Author : Alejandro Sotelo Arévalo
// -------------------------------------------------------------------------------------------

// -------------------------------------------------------------------------------------------
// GRAMMAR DECLARATION AND HIDDEN TOKEN SPECIFICATION
// -------------------------------------------------------------------------------------------
grammar org.gold.dsl.GoldDSL hidden(WS,ML_COMMENT,SL_COMMENT)
// -------------------------------------------------------------------------------------------

// -------------------------------------------------------------------------------------------
// IMPORT EXISTING EPackages
// -------------------------------------------------------------------------------------------
import "http://www.eclipse.org/emf/2002/Ecore" as ecore
import "http://www.eclipse.org/xtext/common/JavaVMTypes" as types
// -------------------------------------------------------------------------------------------

// -------------------------------------------------------------------------------------------
// GOLD EPackage GENERATION
// -------------------------------------------------------------------------------------------
generate goldDSL "http://wwwest.uniandes.edu.co/~a-sotelo/GOLD3/goldDSL"
// -------------------------------------------------------------------------------------------

// -------------------------------------------------------------------------------------------
// MAIN TOKEN (NON-TERMINAL START SYMBOL)
// -------------------------------------------------------------------------------------------
GoldProgram: // GOLD program
  (annotations+=Annotation)*           // Annotations
  package=Package                      // Package declaration
  imports=Imports                      // Import declarations
  staticDeclarations=StaticDeclarations // Static declarations
;
// -------------------------------------------------------------------------------------------

// -------------------------------------------------------------------------------------------
// TERMINAL FRAGMENTS
// -------------------------------------------------------------------------------------------
terminal fragment LETTER: // Latin alphabet and greek alphabet
  'A'..'Z'|'a'..'z'|'\u0391'..'\u03A9'|'\u03B1'..'\u03C9';
terminal fragment DIGIT: // Decimal digits
  '0'..'9';
terminal fragment SUBINDEX: // Numerical subscript digits
  '\u2080'..'\u2089';
terminal fragment HEX_DIGIT: // Hexadecimal digits
  '0'..'9'|'A'..'F'|'a'..'f';
```

```
terminal fragment HEX_CODE: // Unicode character scape code
  'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT;
// ------------------------------------------------------------------------------------------

// ------------------------------------------------------------------------------------------
// TERMINAL RULES
// ------------------------------------------------------------------------------------------
terminal CONSTANT: // Basic constants
    'TRUE'|'true'|'FALSE'|'false' // Boolean values
  | 'NIL'|'nil'|'NULL'|'null'     // Null pointer
  | '\u00D8'                      // Empty set/bag
  | '\u22C3'                      // Universe set
  | '\u025B'                      // Empty sequence
  | '\u03BB'                      // Empty string
  | '\u221E'                      // Positive infinity
;
terminal PRIMITIVE_TYPE: // Primitive types (basic mathematical sets)
    '\u212C' // Boolean values
  | '\u2115' // Natural numbers
  | '\u2124' // Integer numbers
  | '\u211A' // Rational numbers
  | '\u2148' // Irrational numbers
  | '\u211D' // Real numbers
  | '\u2102' // Complex numbers
;
terminal NUMBER: // Integer and floating point numbers
  DIGIT+ ('.' DIGIT+)? (('E'|'e') '-'? DIGIT+)?
  ('L'|'l' // long   (java.lang.Long)
  |'I'|'i' // int    (java.lang.Integer)
  |'S'|'s' // short  (java.lang.Short)
  |'B'|'b' // byte   (java.lang.Byte)
  |'D'|'d' // double (java.lang.Double)
  |'F'|'f' // float  (java.lang.Float)
  |'C'|'c' // char   (java.lang.Character)
  )?
;
terminal ID: // Identifiers
  '$'? // Optional escape character
  (LETTER|'_') (LETTER|'_'|DIGIT|SUBINDEX|'\u00B4')*;
terminal STRING: // Strings
  '"'(('\\'('b'|'t'|'n'|'f'|'r'|HEX_CODE|'"'|"'"|'\\'))|!('\\'|'"'))*'"';
terminal CHARACTER: // Characters
  "'"(('\\'('b'|'t'|'n'|'f'|'r'|HEX_CODE|'"'|"'"|'\\'))|!('\\'|"'"))"'";
terminal JAVA_CODE: // Java native code
  '/?' -> '?/';
terminal ML_COMMENT: // Multi-line comments
  '/*' -> '*/';
terminal SL_COMMENT: // Single-line comments
  ('\u29D0'|'//') // Comment mark
  !('\n'|'\r')*;
terminal WS: // Whitespaces
  (' '|'\t'|'\r'|'\n')+;
// ------------------------------------------------------------------------------------------

// ------------------------------------------------------------------------------------------
// PROGRAM MAIN ELEMENTS
// ------------------------------------------------------------------------------------------
Annotation: // Annotation statement
  {Annotation} '@' 'SuppressWarnings'
```

```
108   '(' (('{' (warnings+=STRING (',' warnings+=STRING)*)? '}') | (warnings+=STRING)) ')';
109 Package: // Package declaration
110   {Package} ('package' name=QualifiedName)?;
111 Imports: // Import declarations
112   {Imports} (items+=Import)*;
113 Import: // Import declaration
114   ('import'|'include'|'using') importedNamespace=QualifiedNameWithWildCard;
115 StaticDeclarations: // Static declarations
116   {StaticDeclarations} (items+=StaticDeclaration)*;
117 StaticDeclaration: // Static declaration
118   VariableDeclaration | FunctionDeclaration | JavaCode;
119 // ------------------------------------------------------------------------------------------
120
121 // ------------------------------------------------------------------------------------------
122 // IDENTIFIERS AND QUALIFIED NAMES
123 // ------------------------------------------------------------------------------------------
124 Identifier: // Identifier
125   id=ID;
126 QualifiedName: // Qualified name (the arroba sign is used to denote inner classes)
127   ID ('.' ID)* ('@' ID)*;
128 QualifiedNameWithWildCard: // Qualified name with wilcard (*) or superwilcard (**)
129   QualifiedName ('.*'|'.**')?;
130 // ------------------------------------------------------------------------------------------
131
132 // ------------------------------------------------------------------------------------------
133 // SUBSCRIPTS
134 // ------------------------------------------------------------------------------------------
135 Subscript: // Subscript
136   dimension+='[' ((type?=']' (dimension+='[' ']')*)
137                 |(indices+=Expression ']' (dimension+='[' indices+=Expression ']')*)
138                 );
139 // ------------------------------------------------------------------------------------------
140
141 // ------------------------------------------------------------------------------------------
142 // TYPE REFERENCES
143 // ------------------------------------------------------------------------------------------
144 Reference: // Type reference
145   id=QualifiedName | id=PRIMITIVE_TYPE;
146 Type: // Type reference with dimension subscripts
147   symbol=Reference (dimension+='[' ']')*;
148 SpecialType: // Type reference with arguments or subscripts
149   symbol=Reference (arguments=Arguments | subscript=Subscript)?;
150 // ------------------------------------------------------------------------------------------
151
152 // ------------------------------------------------------------------------------------------
153 // VARIABLE AND FUNCTION/PROCEDURE DECLARATIONS
154 // ------------------------------------------------------------------------------------------
155 Variable: // Single variable declaration
156   id=Identifier (':' type=SpecialType)?;
157 Parameter: // Single parameter declaration
158     ids+=Identifier (':' type=Type)?
159   | tuple?='\u27E8' ids+=Identifier (',' ids+=Identifier)* '\u27E9';
160 VariableDeclaration: // Multiple variable declaration
161   'var' variables+=Variable (',' variables+=Variable)*;
162 FunctionDeclaration: // Function/Procedure declaration
163   ('function'?|procedure?='procedure') id=Identifier
164   '(' (parameters+=Parameter (',' parameters+=Parameter)*)? ')' (':' returnType=Type)?
165   (('begin' (body+=Instruction)* 'end') | ((':='|'=') macro=Expression));
166 // ------------------------------------------------------------------------------------------
```

```
167
168 // ------------------------------------------------------------------------------------------
169 // VARIABLE AND FUNCTION APPLICATIONS
170 // ------------------------------------------------------------------------------------------
171 VariableApplication: // Variable application
172   id=Reference
173   ((arguments=Arguments | subscript=Subscript) ('.' chain=VariableApplication)?)?;
174 FunctionApplication: // Function application
175   id=Reference arguments=Arguments ('.' chain=VariableApplication)?;
176 // ------------------------------------------------------------------------------------------
177
178 // ------------------------------------------------------------------------------------------
179 // LANGUAGE COMMADS (INSTRUCTIONS)
180 // ------------------------------------------------------------------------------------------
181 Instruction: // Instruction command
182     VariableDeclaration | Skip | Abort | Call | Assert | Assignment | Swap
183   | Conditional | Repetition | Print | Error | Throw | Return | Escape | JavaCode;
184 Skip: // Empty instruction
185   {Skip} 'skip';
186 Abort: // Abnormal termination instruction
187   {Abort} 'abort';
188 Call: // Function/procedure/method invocation
189   function=FunctionApplication | 'call' expression=Expression;
190 Assert: // Assertion statement
191   'assert' condition=Expression;
192 Assignment: // Assignment instruction
193   variables=AssignmentVariables ('\u2190'|':='|'=') expressions=Expressions;
194 AssignmentVariables: // Assignment variable list
195   items+=AssignmentVariable (',' items+=AssignmentVariable)*;
196 AssignmentVariable: // Assignment variable
197   id=Identifier (('[' indices+=Expression ']')+ | ':' type=Type)?;
198 Swap: // Swap instruction
199   ('swap'|'exchange') first=SwapVariable ('\u2194'|'with') second=SwapVariable;
200 SwapVariable: // Swap variable
201   id=Identifier ('[' indices+=Expression ']')*;
202 Conditional: // Conditional statement
203     IfThenElse  // If-then-else statement
204   | Switch      // Switch statement
205 ;
206 IfThenElse: // If-then-else statement
207   'if' guard=Expression if=If (elseIfs+=ElseIf)* (else=Else)? 'end';
208 If: // If clause
209   {If} 'then' (body+=Instruction)*;
210 ElseIf: // Elseif clause
211   'elseif' guard=Expression 'then' (body+=Instruction)*;
212 Else: // Else clause
213   {Else} 'else' (body+=Instruction)*;
214 Switch: // Switch statement
215   'switch' control=Expression 'begin' (cases+=SwitchCase)+ (default=SwitchDefault)? 'end';
216 SwitchCase: // Switch case statement
217   'case' guard=Expression ':' (body+=Instruction)*;
218 SwitchDefault: // Switch default statement
219   {SwitchDefault} 'default' ':' (body+=Instruction)*;
220 Repetition: // Repetitive instruction
221     While    // While statement
222   | DoWhile  // Do-while statement
223   | Repeat   // Repeat-until statement
224   | ForEach  // For-each statement
225   | For      // For statement
```

```
;
While: // While statement
  'while' guard=Expression 'do' (body+=Instruction)* 'end';
DoWhile: // Do-while statement
  'do' (body+=Instruction)* 'whilst' guard=Expression;
Repeat: // Repeat-until statement
  'repeat' (body+=Instruction)* 'until' guard=Expression;
ForEach: // For-each statement
  'for' 'each' ( variables+=Identifier (':' type=Type)?
               | tuple?='\u27E8' variables+=Identifier (',' variables+=Identifier)* '\u27E9'
               ) ('\u2208'|'in') collection=Expression 'do'
  (body+=Instruction)*
  'end';
For: // For statement
  'for' start=Assignment (to?='to' | downto?='downto')
    limit=Expression ('by' step=Expression)? 'do'
  (body+=Instruction)*
  'end';
Print: // Print clause
  'print' messages=Expressions;
Error: // Error clause
  'error' messages=Expressions;
Throw: // Throw clause
  'throw' exception=Expression;
Return: // Return statement
  {Return} ('return' expression=Expression | 'finalize');
Escape:  // Escape sequencer break/continue
  break?='break' | continue?='continue';
JavaCode: // Java native code
  javaCode=JAVA_CODE;
// ------------------------------------------------------------------------------------------

// ------------------------------------------------------------------------------------------
// ARGUMENT AND EXPRESSION LISTS
// ------------------------------------------------------------------------------------------
Arguments: // Argument list
  {Arguments} '(' (list=Expressions)? ')';
Expressions: // Expression list
  terms+=Expression (',' terms+=Expression)*;
// ------------------------------------------------------------------------------------------

// ------------------------------------------------------------------------------------------
// EXPRESSIONS
// ------------------------------------------------------------------------------------------
Expression: // Any expression
  Condt;
Condt returns Expression: // Non associative
  Equiv
  ({ConditionalExpression.guard=current} '?'
    thenExpression=Equiv ':' elseExpression=Equiv)?;   // Conditional expression
Equiv returns Expression: // Mutually associative
  Implc (
  ({OpEquivY.left=current} ('\u2261'|'\u21D4'|'eqv')   // Equivalence / If and only if
  |{OpEquivN.left=current} ('\u2262'|'\u2295'|'xor')   // Inequivalence / Exclusive or
  ) right=Implc)*;
Implc returns Expression: // Mutually right associative
  Consq
  (({OpImplcY.left=current} '\u21D2' right=Implc)      // Implication
  |({OpImplcN.left=current} '\u21CF' right=Implc)      // Anti-implication
```

```
285   )?;
286 Consq returns Expression: // Mutually left associative
287   Disjc (
288   ({OpConsqY.left=current} '\u21D0'                    // Consequence
289   |{OpConsqN.left=current} '\u21CD'                    // Anti-consequence
290   ) right=Disjc)*;
291 Disjc returns Expression: // Associative
292   Opcnj
293   (({OpDisjcY.left=current} ('\u2228'|'or' |'||') right=Opcnj)+  // Disjunction
294   |({OpConjcY.left=current} ('\u2227'|'and'|'&&') right=Opcnj)+  // Conjunction
295   )?;
296 Opcnj returns Expression: // Mutually conjunctional
297   Opcns (
298   ({OpEqualY.left=current} ('='|'==')                  // Equality
299   |{OpEqualN.left=current} ('\u2260'|'!='|'<>')        // Inequality
300   |{OpLesstY.left=current} '<'                         // Less than
301   |{OpLessqY.left=current} ('\u2264'|'<=')             // Less than or equal to
302   |{OpGreatY.left=current} '>'                         // Greater than
303   |{OpGreaqY.left=current} ('\u2265'|'>=')             // Greater than or equal to
304   |{OpDivisY.left=current} '\u2223'                    // Divisibility
305   |{OpDivisN.left=current} '\u2224'                    // Anti-divisibility
306   |{OpMembrY.left=current} ('\u2208'|'in')             // Membership
307   |{OpMembrN.left=current} '\u2209'                    // Anti-membership
308   ) right=Opcns)*;
309 Opcns returns Expression: // Mutually conjunctional
310   Opseq (
311   ({OpDisjoY.left=current} '\u22C8'                    // Disjoint
312   |{OpSbsetY.left=current} '\u2286'                    // Subset
313   |{OpSbsetN.left=current} '\u2288'                    // Not subset
314   |{OpSpsetY.left=current} '\u2287'                    // Superset
315   |{OpSpsetN.left=current} '\u2289'                    // Not superset
316   |{OpPssetY.left=current} ('\u2282'|'\u228A')         // Proper subset
317   |{OpPssetN.left=current} '\u2284'                    // Not proper subset
318   |{OpPpsetY.left=current} ('\u2283'|'\u228B')         // Proper superset
319   |{OpPpsetN.left=current} '\u2285'                    // Not proper superset
320   ) right=Opseq)*;
321 Opseq returns Expression: // Left associative
322   Opapn
323   (({OpPrepdY.left=current} '\u22B3' right=Opapn)+     // Prepend
324   |({OpConctY.left=current} '\u2303' right=Opapn)+     // Concatenation
325   )?;
326 Opapn returns Expression: // Right associative
327   Occur
328   (({OpAppedY.left=current} '\u22B2' right=Opapn)      // Append
329   )?;
330 Occur returns Expression: // Non associative
331   Intvl
332   (({OpOccurY.left=current} '#' right=Intvl)          // Number of occurrences
333   )?;
334 Intvl returns Expression: // Non associative
335   Maxim
336   (({OpIntvlY.left=current} '\u2025' right=Maxim)     // Interval range
337   )?;
338 Maxim returns Expression: // Associative
339   Addit
340   (({OpMaximY.left=current} '\u2191' right=Addit)+     // Maximum
341   |({OpMinimY.left=current} '\u2193' right=Addit)+     // Minimum
342   )?;
343 Addit returns Expression: // Left mutually associative
```

```
344   Multp (
345   ({OpAdditY.left=current} '+'                          // Addition
346   |{OpSubtrY.left=current} '-'                          // Subtraction
347   ) right=Multp)*;
348 Multp returns Expression: // Left mutually associative
349   Opset (
350   ({OpMultpY.left=current} ('*'|'\u00B7')               // Multiplication
351   |{OpDividY.left=current} '/'                          // Division
352   |{OpModulY.left=current} ('\u0025'|'mod')             // Integer residue / Module
353   |{OpQuotnY.left=current} ('\u00F7'|'div')             // Integer division / Quotient
354   |{OpGrtcdY.left=current} 'gcd'                        // Greatest common divisor
355   |{OpLstcmY.left=current} 'lcm'                        // Least common multiple
356   ) right=Opset)*;
357 Opset returns Expression: // Left associative
358   Expon
359   (({OpUnionY.left=current} '\u222A' right=Expon)+     // Union
360   |({OpInterY.left=current} '\u2229' right=Expon)+     // Intersection
361   |({OpDiffeY.left=current} '\u005C' right=Expon)+     // Difference
362   |({OpSymmdY.left=current} '\u2206' right=Expon)+     // Symmetric difference
363   )?;
364 Expon returns Expression: // Left associative
365   Carts (
366   ({OpExponY.left=current} '^'                          // Exponentiation, Cartesian power
367   ) right=Carts)*;
368 Carts returns Expression: // Associative
369   Prefx
370   ({OpCartsY.operands+=current} ('\u00D7' operands+=Prefx)+)?  // Cartesian product
371 ;
372 Prefx returns Expression: // Unary prefix operators
373     {OpPlussY} '+' operand=Prefx                        // Unary plus sign
374   | {OpMinusY} '-' operand=Prefx                        // Unary minus sign
375   | {OpNegatY} ('\u00AC'|'not'|'!') operand=Prefx       // Negation
376   | {OpCardiY} '#' operand=Prefx                        // Cardinality
377   | {OpComplY} '~' operand=Prefx                        // Complement
378   | {OpPwsetY} '\u2118' operand=Prefx                   // Power set
379   | Suffx
380 ;
381 Suffx returns Expression: // Unary suffix operators
382   Brack
383   ({OpFactrY.operand=current} '!')*                 // Factorial
384 ;
385 Brack returns Expression: // Brackets
386     {OpAbsolY} '|' operand=Expression '|'               // Cardinality, Absolute value
387   | {OpFloorY} '\u230A' operand=Expression '\u230B'    // Floor
388   | {OpCeilnY} '\u2308' operand=Expression '\u2309'    // Ceiling
389   | Comph
390 ;
391 Comph returns Expression: // Enumerations and comprehensions (arrays, sets, bags, sequences)
392     {Arr} ('new'? type=Type)? '\u27E6' (elements=Expressions)? '\u27E7'  // Arrays
393   | {Set} '{' (content=Builder)? '}'                                     // Sets
394   | {Bag} '\u2983' (content=Builder)? '\u2984'                           // Bags
395   | {Seq} '\u27E8' (content=Builder)? '\u27E9'                           // Sequences
396   | Basic
397 ;
398 Basic returns Expression: // Primary basic expressions
399     {StringLiteral} value=STRING                        // String literal
400   | {CharacterLiteral} value=CHARACTER                  // Character literal
401   | {ConstantLiteral} value=CONSTANT                    // Constant literal
402   | {NumberLiteral} value=NUMBER                        // Number literal
```

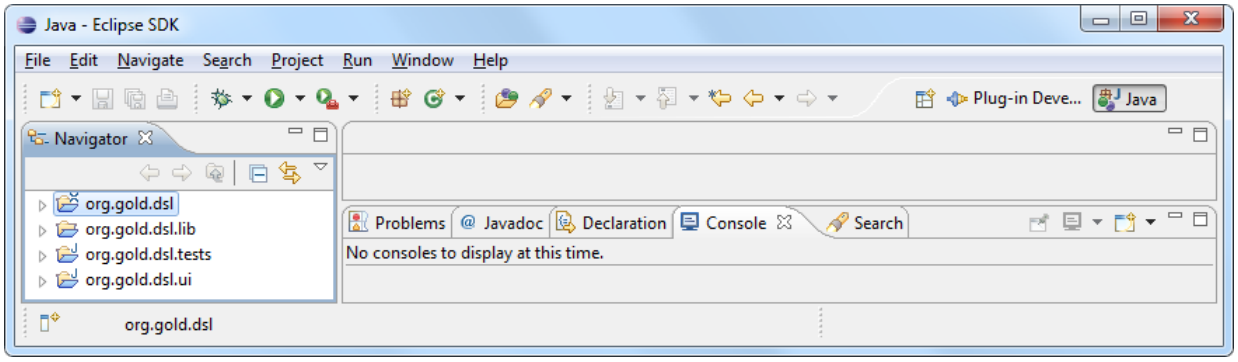
```
403   | {VariableLiteral} application=VariableApplication     // Variable application
404   | {ConstructorApplication} 'new' type=SpecialType       // Constructor application
405   | {BasicFunctionApplication}                            // Basic function application
406     id=('max'|'min'|'gcd'|'lcm'|'abs'|'pow'|'sqrt'|'cbrt'|'root'|'ln'|'log'|'exp'
407        |'sin'|'cos'|'tan'|'sinh'|'cosh'|'tanh'
408        |'asin'|'acos'|'atan'|'asinh'|'acosh'|'atanh'
409        ) '(' operands=Expressions ')'
410   | '('
411     ( {ParenthesizedExpression}                           // Parenthesized expression
412       expression=Expression (cast?=(':'|'as') type=Type)? ')'
413       ('.' application=VariableApplication)?
414     | {GriesQuantification}                               // Gries-style quantification
415       (frall?='\u2200' // Universal quantifier
416       |exist?='\u2203' // Existential quantifier
417       |sumat?='\u2211' // Summation quantifier
418       |prodc?='\u220F' // Product quantifier
419       |maxim?='\u2191' // Maximum quantifier
420       |minim?='\u2193' // Minimum quantifier
421       |union?='\u222A' // Union quantifier
422       |inter?='\u2229' // Intersection quantifier
423       ) dummies+=ID (',' dummies+=ID)* '|' range=Range ':' body=Expression ')'
424     )
425 ;
426 Builder: // Enumeration and comprehension builder (left factored)
427   members+=Expression (inComprehension?='|' range=Range | (',' members+=Expression)*);
428 Range: // Range conditions
429   conditions+=RangeCondition (',' conditions+=RangeCondition)*;
430 RangeCondition: // Range condition
431   dummy=Dummy | '[' expression=Expression ']';
432 Dummy:  // Dummy declaration
433   Opseq
434   ( {DummyInterval.left=current}       // Dummy declaration over a closed/open interval
435     (lesst1?='<'                 // Less than
436     |lessq1?=('\u2264'|'<=')     // Less than or equal to
437     )
438     id=Opseq
439     (lesst2?='<'                 // Less than
440     |lessq2?=('\u2264'|'<=')     // Less than or equal to
441     )
442     right=Opseq
443   | {DummyUniverse.id=current}  // Dummy declaration over a collection
444     (equal?='='                  // Equality
445     |membr?=('\u2208'|'in')      // Membership
446     |sbset?='\u2286'             // Subset
447     |psset?=('\u2282'|'\u228A') // Proper subset
448     )
449     expression=Opseq
450   );
451 // ------------------------------------------------------------------------------------------
```

A.2. Generación del *plug-in*

Para generar un instalador del producto, se debe exportar el proyecto *Xtext* que contiene la implementación de *GOLD 3* como un *plug-in* listo para instalar en *Eclipse*, a través del siguiente procedimiento:

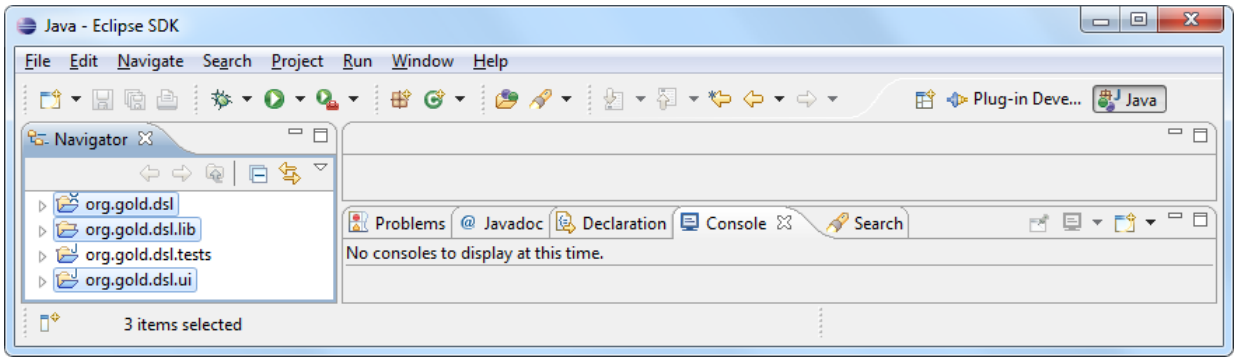
1. Si en la máquina no está instalado *Eclipse 3.7.1* (*Indigo*) con el *plug-in* de *Xtext 2.2.1*:
   - Dependiendo de su sistema operativo, descargue la última versión de *Eclipse-Xtext* en la sección *Downloads* → *Eclipse Xtext 2.2.1 Distribution (Indigo)* de la página http://xtext.itemis.com/, o en su defecto, ubique el instalador de *Eclipse-Xtext* presente en el directorio `/Eclipse-Xtext` de la distribución de *GOLD 3* (e.g., `/Eclipse-Xtext/Windows32/eclipse-SDK-3.7.1-Xtext-2.2.1-win32.zip` para sistemas Windows de 32 o 64 bits).
   - Instale la herramienta *Eclipse-Xtext*, que ya tiene integrado por defecto el *plug-in* de *Xtext*. No se aconseja instalar *Eclipse*, y luego por separado la distribución de *Xtext*, porque se podrían producir errores internos.
2. Importe dentro de *Eclipse-Xtext* los proyectos `org.gold.dsl`, `org.gold.dsl.lib`, `org.gold.dsl.tests` y `org.gold.dsl.ui`, presentes en el directorio `/Sources` de la distribución de *GOLD 3*.

Figura A.1. *Importación de los proyectos que implementan GOLD 3, en Eclipse.*



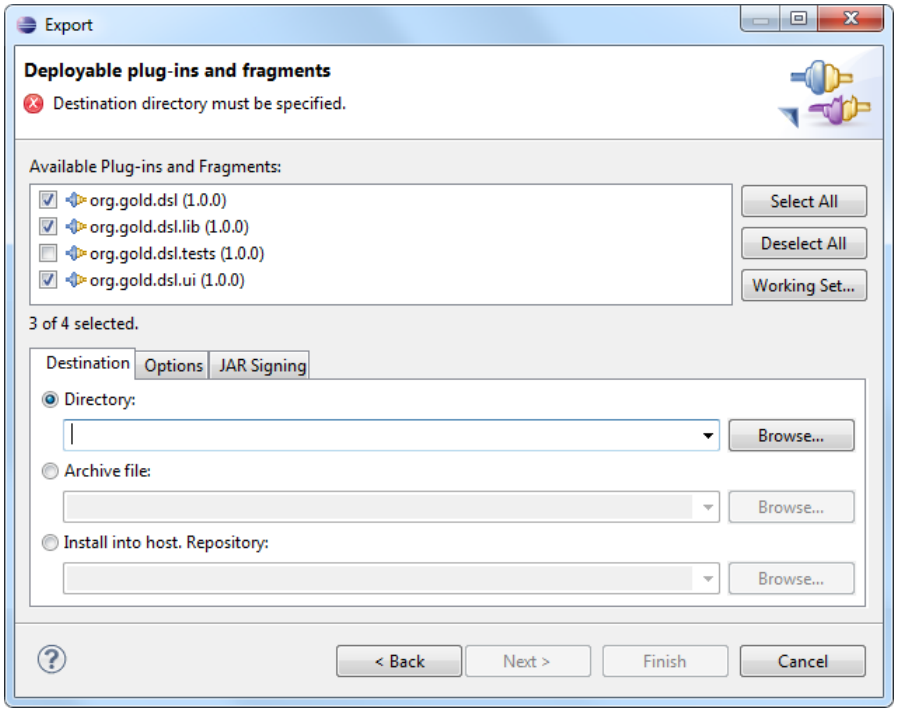
3. Seleccione los proyectos `org.gold.dsl`, `org.gold.dsl.lib` y `org.gold.dsl.ui` en la vista *Navigator* de *Eclipse-Xtext*, dejando sin seleccionar el proyecto `org.gold.dsl.tests`.

Figura A.2. *Selección de los proyectos que componen GOLD 3 en Eclipse, exceptuando* `org.gold.dsl.tests`.



4. Abra el asistente de publicación de *plug-ins* bajo *File* → *Export...* → *Plug-in Development* → *Deployable plug-ins and fragments*, asegurándose de que estén seleccionados los proyectos `org.gold.dsl`, `org.gold.dsl.lib` y `org.gold.dsl.ui`, pero no el proyecto `org.gold.dsl.tests`.

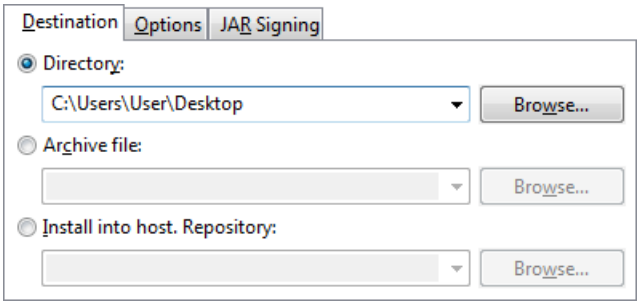
Figura A.3. *Asistente para la generación del plug-in de GOLD 3 en Eclipse.*



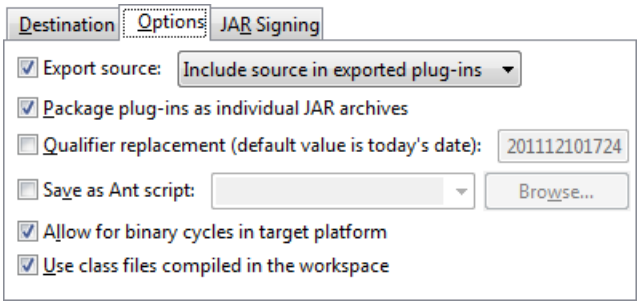
5. Configure las opciones para la generación del *plug-in* de *GOLD 3*, seleccionando la opción *Directory* en la pestaña *Destination* para escoger el directorio donde se van a exportar los archivos de instalación, y activando la opción *Use class files compiled in the workspace* en la pestaña *Options* para no tener problema con la codificación de los archivos compilados. Además, si se desea exportar el código fuente de *GOLD*, se debe activar la casilla *Export source* y seleccionar la opción *Include source in exported plug-ins*, bajo la pestaña *Options*. Esto último permite que las ayudas de contenido (*content assist*) tengan acceso a los nombres de los parámetros de los métodos y constructores de las clases que componen la librería *GOLD*.

Figura A.4. *Configuración de la generación del plug-in de GOLD 3 en Eclipse.*

(a) *Pestaña Destination.*



(b) *Pestaña Options.*



6. Haga clic en el botón *Finish* y espere a que *Eclipse-Xtext* realice las operaciones. Al terminar el proceso de generación del *plug-in* de *GOLD 3*, quedarán tres archivos en el subdirectorio `plugins/` bajo el directorio especificado en el paso anterior: `org.gold.dsl_3.0.0.jar`, `org.gold.dsl.lib_3.0.0.jar` y `org.gold.dsl.ui_3.0.0.jar`. Si desea alterar el identificador de la versión (e.g., `_3.0.0`), debe modificar el archivo `plugin.xml` del proyecto `org.gold.dsl`, o configurar la opción *Qualifier replacement (default value is today's date)* bajo la pestaña *Options*.

A.3. Instalación del *plug-in*

Para instalar el *plug-in* de *GOLD 3* se deben aplicar los siguientes pasos:

1. Dependiendo de su sistema operativo, instale una nueva copia de *Eclipse 3.7.1* (*Indigo*) con el *plug-in* de *Xtext 2.2.1*, como se describió en la sección §A.2.

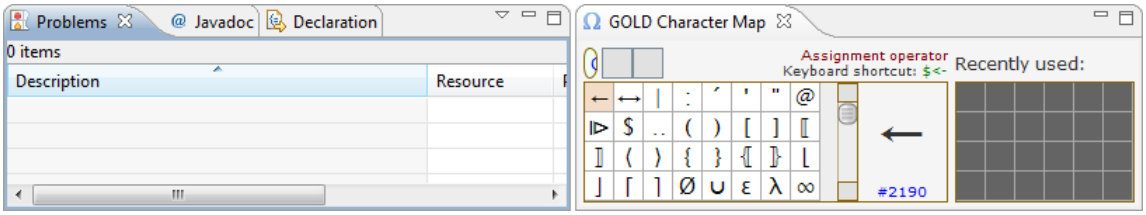
2. Diríjase al directorio `/Plug-in` de la distribución de *GOLD 3*, y copie los archivos `org.gold.dsl_3.0.0.jar`, `org.gold.dsl.lib_3.0.0.jar` y `org.gold.dsl.ui_3.0.0.jar` dentro del directorio `plugins` de la versión instalada de *Eclipse*.

3. Ejecute la aplicación *Eclipse* recién instalada, configurando la ubicación del *workspace* y cerrando la pantalla *Welcome to Eclipse* (si aparece).

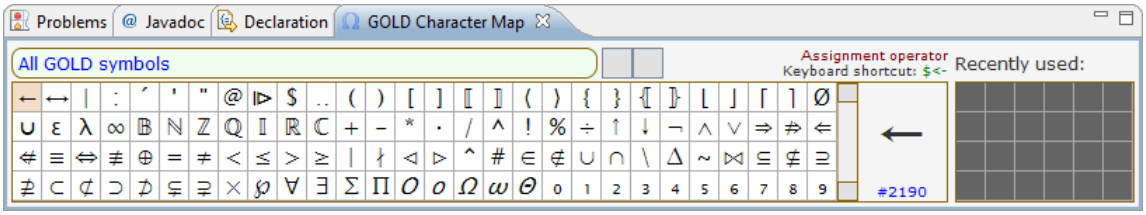
4. Ubique el mapa de caracteres de *GOLD* dentro de la barra de pestañas del extremo inferior de la ventana de *Eclipse* (e.g., a la derecha de la vista *Declaration* de *Eclipse*).

Figura A.5. *Reubicación del mapa de caracteres de GOLD 3 en Eclipse.*

(a) *Antes de reubicar.*



(b) *Después de reubicar.*



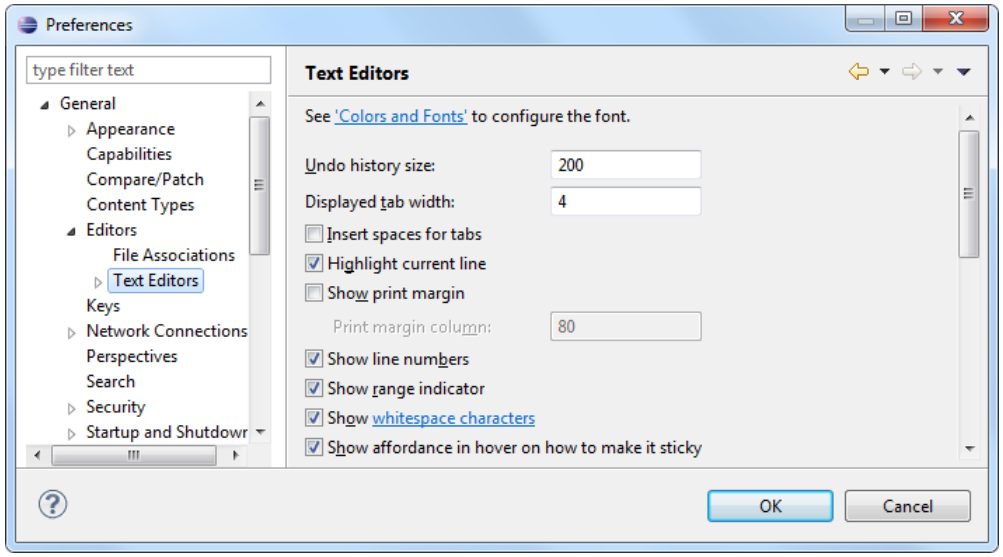
5. Si el mapa de caracteres no se despliega correctamente, instale el tipo de letra `/Plug-in/GoldRegular.ttf` en el sistema operativo (e.g., bajo *Windows* basta copiar el archivo `ttf` dentro del directorio `C:\Windows\Fonts`), y luego reinicie *Eclipse* [†1].

6. Active la vista *Navigator* de *Eclipse*, bajo la opción *Window* → *Show View* → *Navigator*.

7. En la ventana *Window* → *Preferences* → *General* → *Editors* → *Text Editors*, active las opciones *Show line numbers* y *Show whitespace characters*.

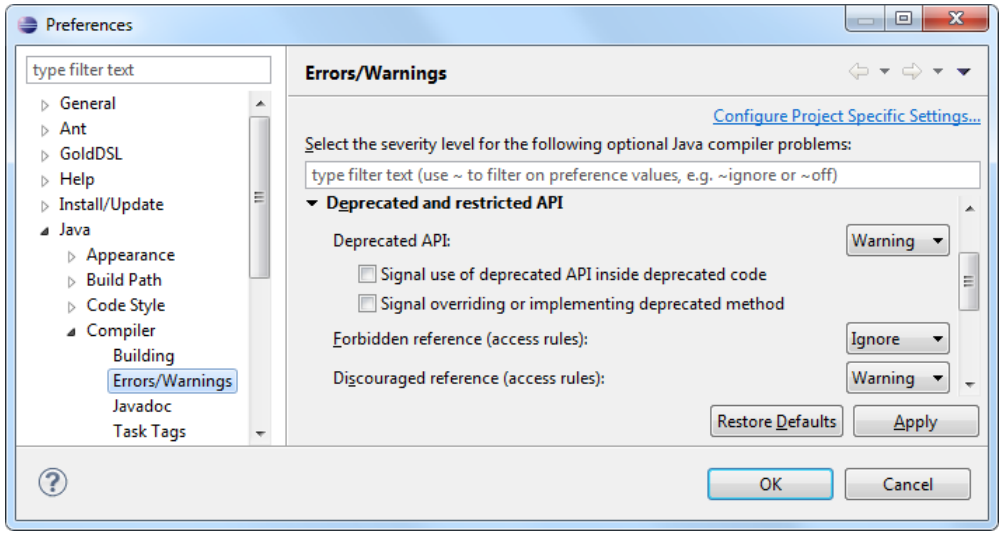
[1] Para desinstalar el tipo de letra `GoldRegular.ttf` en *Windows 7*, se debe eliminar la entrada con nombre *GoldRegular* de la ubicación `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Fonts` del *Registro de Windows*, disponible bajo el comando `regedit`. En sistemas *Windows 7* de 64 bits también debe eliminarse la entrada *GoldRegular* de la ubicación `HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Fonts`.

Figura A.6. *Configuración del editor de texto de Eclipse, para trabajar con GOLD 3.*



8. Para evitar errores de restricción de acceso (*access restriction*) del estilo ‘‘*... is not accessible due to restriction on required library ...*’’, se debe poner la opción *Warning* o *Ignore* bajo *Window* → *Preferences* → *Java* → *Compiler* → *Errors/Warnings* → *Deprecated and restricted API* → *Forbidden reference (access rules)*.

Figura A.7. *Configuración del compilador de Java en Eclipse, para trabajar con GOLD 3.*



9. Para que las fuentes del *JDK* queden encadenadas a *Eclipse*, se debe configurar en *Eclipse* la ruta donde se encuentra el archivo `src.zip` de *Java* (e.g., `C:\Program Files (x86)\Java\jdk1.6.0_24\src.zip`) bajo la opción *Window* → *Preferences* → *Java* → *Installed JREs* → `jre6` → *Edit...* → `rt.jar` → *Source Attachment...* → *External File*. La anterior configuración sirve para que en las ayudas de contenido (*content assist*) de *Java* y de *GOLD* se desplieguen los nombres de los parámetros de los métodos y constructores cada vez que se opriman las teclas `Control+Space` sobre alguno de estos elementos.

A.4. Contenido de la distribución

El disco óptico en formato *DVD* con la distribución de *GOLD 3* contiene el código fuente de la implementación y otros archivos importantes, descritos en la tabla A.1.

Tabla A.1. *Descripción de los directorios presentes en la distribución de GOLD 3.*

Directorio	Descripción
`/Data/`	Archivos internos de *GOLD 3*.
`/Data/Fonts/`	Tipografías usadas en la interfaz gráfica del *IDE* de *GOLD 3*, incluyendo el *script FontForge* [77] que genera el tipo de letra *GOLD Regular* (véase la tabla 8.2).
`/Data/LaTeX/`	Programa *Java* que genera la tabla de símbolos usada para convertir caracteres *Unicode* en códigos *LaTeX* durante la exportación de archivos *GOLD* a *LaTeX*.
`/Data/Preferences/`	Configuración de las preferencias de *Eclipse* en formato `epf` y formateador de código (*code formatter*) en formato `xml`, usados durante la implementación del código fuente.
`/Data/Screenshots/`	Capturas de pantalla (*screenshots*) de ejemplo que ilustran varios escenarios donde se usa el lenguaje *GOLD 3* dentro del entorno de desarrollo integrado.
`/Data/Tests/`	Proyectos *GOLD* creados en *Eclipse* que contienen una diversidad de ejemplos que ilustran la utilización del lenguaje *GOLD 3*.
`/Data/UML/`	Diagramas de diseño *UML* de *GOLD 3*, que incluyen diagramas de clases y de paquetes editados en el programa *ArgoUML* [84].
`/Eclipse-Xtext/`	Instaladores de *Eclipse* que tienen embebido el *plug-in* de *Xtext*.
`/Eclipse-Xtext/LinuxGTK32/`	Instalador de *Eclipse 3.7.1* (*Indigo*) integrado con *Xtext 2.2.1*, para sistemas *Linux* de 32 bits (no sirve para procesadores de 64 bits).
`/Eclipse-Xtext/LinuxGTK64/`	Instalador de *Eclipse 3.7.1* (*Indigo*) integrado con *Xtext 2.2.1*, para sistemas *Linux* de 64 bits (no sirve para procesadores de 32 bits).
`/Eclipse-Xtext/MacOSX32/`	Instalador de *Eclipse 3.7.1* (*Indigo*) integrado con *Xtext 2.2.1*, para sistemas *MacOS X Cocoa* de 32 bits (no sirve para procesadores de 64 bits).
`/Eclipse-Xtext/MacOSX64/`	Instalador de *Eclipse 3.7.1* (*Indigo*) integrado con *Xtext 2.2.1*, para sistemas *MacOS X Cocoa* de 64 bits (no sirve para procesadores de 32 bits).
`/Eclipse-Xtext/Windows32/`	Instalador de *Eclipse 3.7.1* (*Indigo*) integrado con *Xtext 2.2.1*, para sistemas *Windows* de 32 bits (también sirve para procesadores de 64 bits).
`/Eclipse-Xtext/Windows64/`	Instalador de *Eclipse 3.7.1* (*Indigo*) integrado con *Xtext 2.2.1*, para sistemas *Windows* de 64 bits (no sirve para procesadores de 32 bits).
`/Libraries/`	Librerías externas usadas en *GOLD 3*.
`/Libraries/Apfloat/`	Archivos de distribución, código fuente, documentación y empaquetado *JAR* de la librería *Apfloat* [53], versión 1.6.2.
`/Libraries/clrs2e/`	Archivos de distribución, código fuente y documentación de la librería que contiene las implementaciones de referencia de Cormen et al. [23].
`/Libraries/JGraphT/`	Archivos de distribución, código fuente, documentación y empaquetado *JAR* de la librería *JGraphT* [22], versión 0.8.2.
`/Libraries/JUNG2/`	Archivos de distribución, código fuente, documentación y empaquetados *JAR* de la librería *JUNG 2* [21], versión 2.0.1.
`/Plug-in/`	Instalador de *GOLD 3*, distribuido como un *plug-in* de *Eclipse*.
`/Sources/`	Código fuente que implementa *GOLD 3*.
`/Sources/org.gold.dsl/`	Proyecto *Eclipse-Xtext* que contiene la implementación del núcleo del lenguaje y de los aspectos no visuales del *IDE* de *GOLD 3*.
`/Sources/org.gold.dsl.lib/`	Proyecto *Eclipse-Xtext* que contiene los empaquetados *JAR* de las librerías *JUNG 2.0.1* [21] y *Apfloat 0.8.2* [53].
`/Sources/org.gold.dsl.tests/`	Proyecto *Eclipse-Xtext* donde se debe alojar la implementación de las pruebas que se vayan a realizar sobre el núcleo del lenguaje *GOLD 3*.
`/Sources/org.gold.dsl.ui/`	Proyecto *Eclipse-Xtext* que contiene la implementación de los aspectos visuales del *IDE* de *GOLD 3*.

A.5. Tablas

A.5.1. Símbolos

Tabla A.2. *Símbolos técnicos del lenguaje GOLD 3.*

Símbolo	Código	Atajo	Descripción
←	0x2190	$<−	Operador de asignación de valores a variables.
↔	0x2194	$:with	Operador de intercambio de valores de variables (*swap*).
|	0x007C		Tal que / Cardinalidad (colecciones) / Valor absoluto (números).
:	0x003A		Dos puntos (*colon*).
_	0x005F		Guión bajo (*underscore*).
′	0x00B4	$:prime	Símbolo prima (*prime symbol*).
'	0x0027		Comilla sencilla para expresar literales de tipo carácter (*Character*).
"	0x0022		Comilla doble para expresar literales de tipo cadena de texto (*String*).
@	0x0040		Marca de inicio de anotación.
⧐	0x29D0	$/	Marca de inicio de comentario.
$	0x0024		Marca de escape de literales y de atajos de teclado.
..	0x2025	$..	Intervalo cerrado de caracteres o de números enteros.
(	0x0028		Paréntesis circular izquierdo (*left parentheses*).
)	0x0029		Paréntesis circular derecho (*right parentheses*).
[	0x005B		Corchete izquierdo para acceder posiciones de un arreglo (*left square bracket*).
]	0x005D		Corchete derecho para acceder posiciones de un arreglo (*right square bracket*).
⟦	0x27E6	$[	Corchete blanco izquierdo para expresar arreglos (*left white square bracket*).
⟧	0x27E7	$]	Corchete blanco derecho para expresar arreglos (*right white square bracket*).
⟨	0x27E8	$(	Paréntesis angular izquierdo para expresar secuencias (*left angle bracket*).
⟩	0x27E9	$)	Paréntesis angular derecho para expresar secuencias (*right angle bracket*).
{	0x007B		Llave izquierda para expresar conjuntos (*left curly bracket*).
}	0x007D		Llave derecha para expresar conjuntos (*right curly bracket*).
⦃	0x2983	${	Llave blanca izquierda para expresar bolsas (*left white curly bracket*).
⦄	0x2984	$}	Llave blanca derecha para expresar bolsas (*right white curly bracket*).
⌊	0x230A	$:lfloor	Piso izquierdo (*left floor*).
⌋	0x230B	$:rfloor	Piso derecho (*right floor*).
⌈	0x2308	$:lceil	Techo izquierdo (*left ceiling*).
⌉	0x2309	$:rceil	Techo derecho (*right ceiling*).

Tabla A.3. *Constantes matemáticas del lenguaje GOLD 3.*

Símbolo	Código	Atajo	Descripción
∅	0x00D8	$:O	Conjunto vacío / Bolsa vacía.
U	0x22C3	$:U	Conjunto universal.
ε	0x025B	$:S	Secuencia vacía.
λ	0x03BB	$:L	Cadena de texto vacía.
∞	0x221E	$:oo	Infinito positivo.

Tabla A.4. *Conjuntos matemáticos básicos del lenguaje GOLD 3.*

Símbolo	Código	Atajo	Descripción
𝔹	0x212C	$:B	Valores booleanos.
ℕ	0x2115	$:N	Números naturales.
ℤ	0x2124	$:Z	Números enteros.
ℚ	0x211A	$:Q	Números racionales.
ℝ	0x211D	$:R	Números reales.
ℂ	0x2102	$:C	Números complejos.

Tabla A.5. *Operadores aritméticos del lenguaje GOLD 3.*

Símbolo	Código	Atajo	Descripción
+	0x002B		Adición (suma) / Más unario.
-	0x002D		Sustracción (resta) / Menos unario.
*	0x002A		Multiplicación.
·	0x00B7	$:mul	Multiplicación.
/	0x002F		División.
^	0x005E	$:pow	Potenciación numérica / Potenciación cartesiana.
!	0x0021		Factorial.
%	0x0025		Residuo de la división entera (*mod*).
÷	0x00F7	$%	Cociente de la división entera (*div*).
↑	0x2191	$:max	Máximo.
↓	0x2193	$:min	Mínimo.

Tabla A.6. *Operadores booleanos del lenguaje GOLD 3.*

Símbolo	Código	Atajo	Descripción
¬	0x00AC	$:not	Negación (no).
∧	0x2227	$:and	Conjunción (y).
∨	0x2228	$:or	Disyunción (o).
⇒	0x21D2	$:imp	Implicación (implica).
⇏	0x21CF	$!imp	Anti-implicación.
⇐	0x21D0	$:con	Consecuencia.
⇍	0x21CD	$!con	Anti-consecuencia.
≡	0x2261	$:eqv	Equivalencia (si y sólo si).
⇔	0x21D4	$:iff	Equivalencia (si y sólo si).
≢	0x2262	$!eqv	Inequivalencia (xor, o exclusivo).
⊕	0x2295	$:xor	Inequivalencia (xor, o exclusivo).

Tabla A.7. *Operadores de comparación del lenguaje GOLD 3.*

Símbolo	Código	Atajo	Descripción
=	0x003D		Igualdad (igual a).
≠	0x2260	$!=	Desigualdad (diferente de).
<	0x003C		Menor que.
≤	0x2264	$<=	Menor o igual que.
>	0x003E		Mayor que.
≥	0x2265	$>=	Mayor o igual que.
|	0x2223	$:|	Divisibilidad (divide a).
∤	0x2224	$!|	Anti-divisibilidad (no divide a).

Tabla A.8. *Operadores sobre colecciones del lenguaje GOLD 3.*

Símbolo	Código	Atajo	Descripción
◁	0x22B3	$<|	Insertar un elemento al principio de una secuencia (*prepend*).
▷	0x22B2	$|>	Insertar un elemento al final de una secuencia (*append*).
ˆ	0x2303	$:cat	Concatenación de secuencias.
#	0x0023		Cardinalidad / Número de ocurrencias de un elemento.
∈	0x2208	$:in	Pertenencia (pertenece a).
∉	0x2209	$!in	Anti-pertenencia (no pertenece a).
∪	0x222A	$:cup	Unión de conjuntos / Unión de bolsas.
∩	0x2229	$:cap	Intersección de conjuntos / Intersección de bolsas.
\	0x005C	$:dif	Diferencia de conjuntos / Diferencia de bolsas.
△	0x2206	$:sym	Diferencia simétrica de conjuntos / Diferencia simétrica de bolsas.
∼	0x007E	$:neg	Complemento de un conjunto.
⋈	0x22C8	$:bowtie	Conjuntos disyuntos / Bolsas disyuntas.

$\subseteq$	0x2286	$:subset	Subconjunto / Subbolsa.
$\nsubseteq$	0x2288	$!subset	No subconjunto / No subbolsa.
$\supseteq$	0x2287	$:supset	Superconjunto / Superbolsa.
$\nsupseteq$	0x2289	$!supset	No superconjunto / Nosuperbolsa.
$\subset$	0x2282	$:psubset	Subconjunto propio / Subbolsa propia.
$\not\subset$	0x2284	$!psubset	No subconjunto propio / No subbolsa propia.
$\supset$	0x2283	$:psupset	Superconjunto propio / Superbolsa propia.
$\not\supset$	0x2285	$!psupset	No superconjunto propio / No superbolsa propia.
$\subsetneq$	0x228A	$:nsubset	Subconjunto propio / Subbolsa propia.
$\supsetneq$	0x228B	$:nsupset	Superconjunto propio / Superbolsa propia.
$\times$	0x00D7	$:X	Producto cartesiano de conjuntos (producto cruz).
$\wp$	0x2118	$:P	Conjunto potencia de un conjunto.

Tabla A.9. *Cuantificadores del lenguaje GOLD 3.*

Símbolo	Código	Atajo	Descripción
$\forall$	0x2200	$:A	Cuantificador universal (para todo).
$\exists$	0x2203	$:E	Cuantificador existencial (existe).
Σ	0x2211	$+	Cuantificador de suma (sumatoria).
Π	0x220F	$*	Cuantificador de multiplicación (multiplicatoria, productoria).

Tabla A.10. *Funciones de complejidad computacional del lenguaje GOLD 3.*

Símbolo	Código	Atajo	Descripción
O	0x2375	$:bigoh	Notación Big-Oh (O).
o	0x2376	$:smalloh	Notación Small-Oh (o).
Ω	0x2377	$:bigomega	Notación Big-Omega (Ω).
ω	0x2378	$:smallomega	Notación Small-Omega (ω).
Θ	0x2379	$:bigtheta	Notación Big-Theta (Θ).

Tabla A.11. *Subíndices numéricos del lenguaje GOLD 3.*

Símbolo	Código	Atajo	Descripción
$_0$	0x2080	$0	Subíndice 0.
$_1$	0x2081	$1	Subíndice 1.
$_2$	0x2082	$2	Subíndice 2.
$_3$	0x2083	$3	Subíndice 3.
$_4$	0x2084	$4	Subíndice 4.
$_5$	0x2085	$5	Subíndice 5.
$_6$	0x2086	$6	Subíndice 6.
$_7$	0x2087	$7	Subíndice 7.
$_8$	0x2088	$8	Subíndice 8.
$_9$	0x2089	$9	Subíndice 9.

A.5.2. Paréntesis

Tabla A.12. *Paréntesis de apertura (izquierdos) y paréntesis de cierre (derechos) en GOLD 3.*

Apertura	Cierre	Descripción
(	)	Paréntesis circular (*parentheses*).
[	]	Corchetes para acceder posiciones de un arreglo (*square brackets*).
⟦	⟧	Corchetes blancos para expresar arreglos (*white square brackets*).
⟨	⟩	Paréntesis angular para expresar secuencias (*angle brackets*).
{	}	Llaves para expresar conjuntos (*curly brackets*).
⦃	⦄	Llaves blancas para expresar bolsas (*white curly brackets*).
⌊	⌋	Piso (*floor*).
⌈	⌉	Techo (*ceiling*).
|	|	Valor absoluto (*absolute value*), cardinalidad (*cardinality*).

A.5.3. Secuencias de escape

Tabla A.13. *Secuencias de escape comunes a GOLD 3 y Java.*

Secuencia de escape	Carácter representado
`\n`	Salto de línea (*line feed*), nueva línea (*new line*).
`\r`	Retorno de carro (*carriage return*).
`\t`	Tabulación (*tab*).
`\b`	Retroceso (*backspace*).
`\f`	Salto de página (*form feed*), nueva página (*new page*).
`\\`	Diagonal inversa (*backslash*).
`\'`	Comilla sencilla (*single quote character*).
`\"`	Comilla doble (*double quote character*).
`\uXXXX`	Carácter *Unicode* con código hexadecimal `XXXX`.

A.5.4. Autocompletado de instrucciones

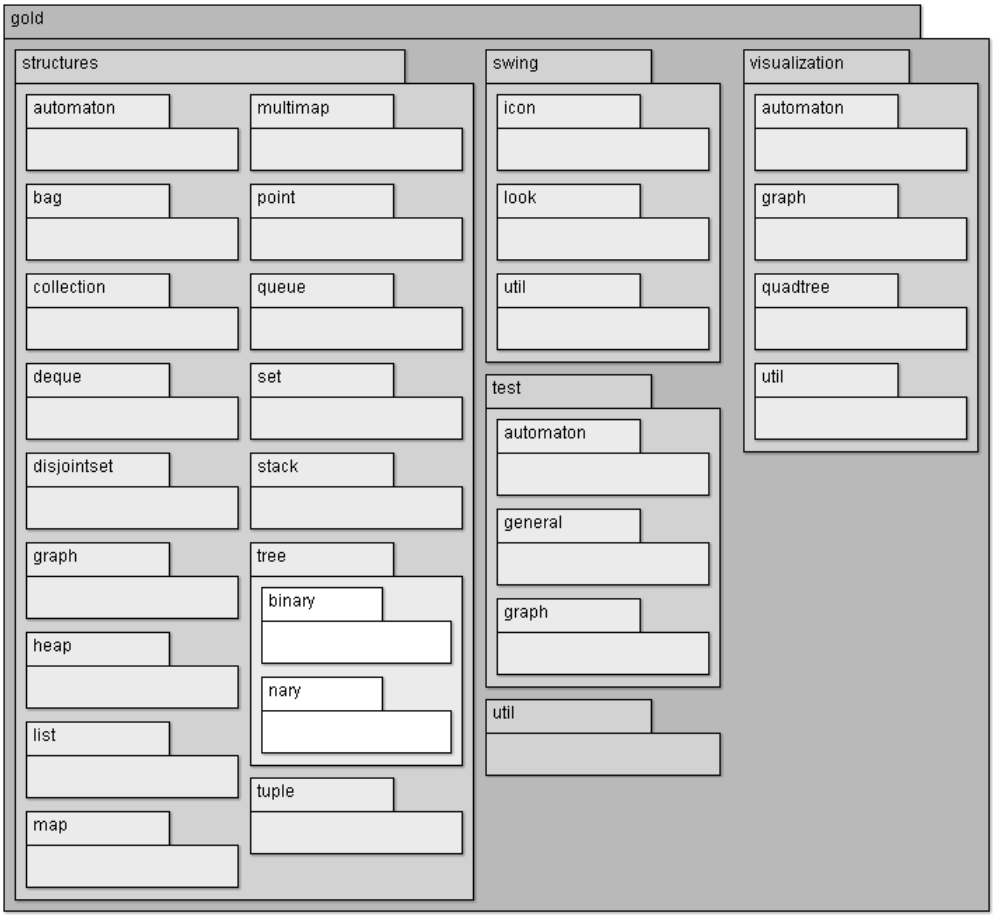
Tabla A.14. *Autocompletado de instrucciones en GOLD 3 (␣= espacio, ↩ = retorno de carro,* I = *cursor).*

Texto digitado	Plantilla insertada
function␣	function␣I()␣begin
procedure␣	procedure␣I()␣begin
if␣	if␣I␣then
elseif␣	elseif␣I␣then
switch␣	switch␣I␣begin
case␣	case␣I:
default␣	default␣I:
for␣	for␣I␣do
while␣	while␣I␣do
swap␣	swap␣I␣with␣
exchange␣	exchange␣I␣with␣
function␣...␣begin ↩	function␣...␣begin ↩ ␣␣I ↩ end ↩
procedure␣...␣begin ↩	procedure␣...␣begin ↩ ␣␣I ↩ end ↩
if␣...␣then ↩	if␣...␣then ↩ ␣␣I ↩ elseif␣TRUE␣then ↩ ␣␣↩ else ↩ ␣␣↩ end ↩
switch␣...␣begin ↩	switch␣...␣begin ↩ ␣␣case␣0: ↩ ␣␣␣␣I ↩ ␣␣case␣1: ↩ ␣␣␣␣↩ ␣␣default: ↩ ␣␣␣␣↩ end
for␣...␣do ↩	for␣...␣do ↩ ␣␣I ↩ end ↩
while␣...␣do ↩	while␣...␣do ↩ ␣␣I ↩ end ↩
repeat ↩	repeat ↩ ␣␣I ↩ until␣FALSE ↩
do ↩	do ↩ ␣␣I ↩ whilst␣TRUE ↩

A.6. Diagramas *UML*

A.6.1. Diagrama de paquetes

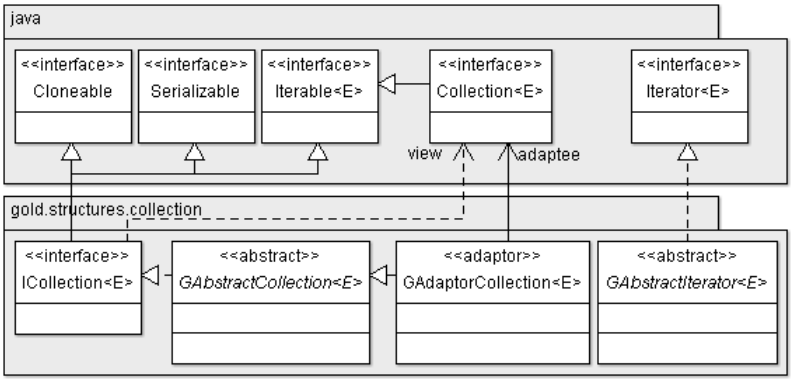
Figura A.8. *Diagrama de paquetes de la librería GOLD.*



A.6.2. Diagramas de clases

A.6.2.1. Paquete *gold.structures.collection*

Diagrama de clases del paquete `gold.structures.collection`.

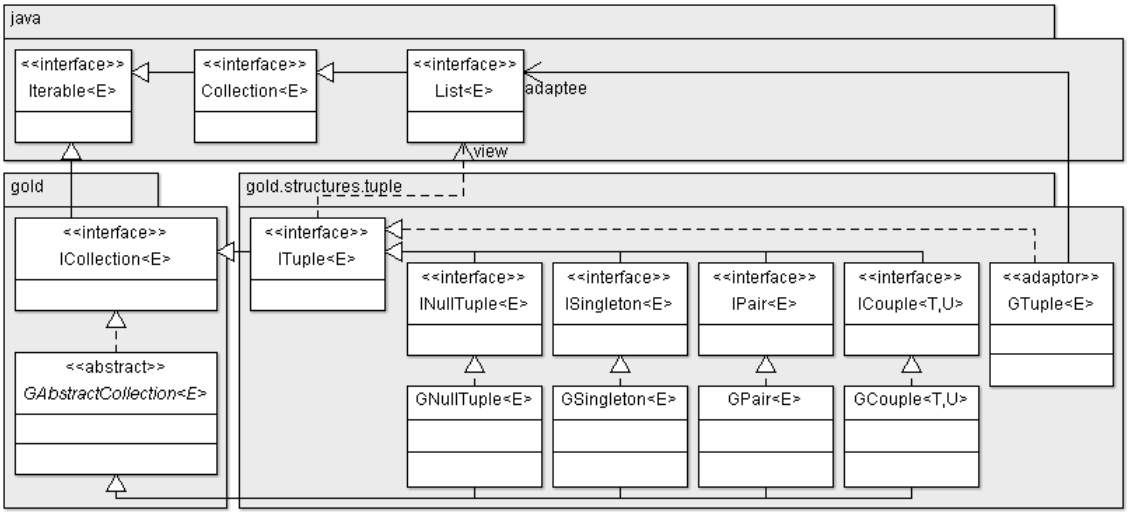


Clases que conforman el paquete `gold.structures.collection`.

Clase	Descripción
`ICollection<E>`	Interfaz que representa una colección de elementos de tipo `E`.
`GAbstractCollection<E>`	Clase abstracta que provee una implementación base de la interfaz `ICollection<E>` para reducir el esfuerzo que se debe realizar para implementar una colección *GOLD*.
`GAdaptorCollection<E>`	Clase que adapta una colección *Java* de tipo `java.util.Collection<E>` como una colección *GOLD* de tipo `ICollection<E>` mediante la aplicación del patrón *Adapter*.
`GAbstractIterator<E>`	Clase abstracta que implementa la interfaz `java.lang.Iterable<E>` para representar un iterador que prohíbe la eliminación de elementos de la colección iterada. Tiene dos métodos abstractos `hasNext` y `next` para controlar el proceso de iteración de una colección siguiendo el patrón *java.util.Iterator*, y tiene un método `remove` cuya implementación por defecto lanza una excepción de tipo `java.lang.UnsupportedOperationException` para evitar que se eliminen miembros de la colección subyacente.

A.6.2.2. Paquete *gold.structures.tuple*

Diagrama de clases del paquete `gold.structures.tuple`.

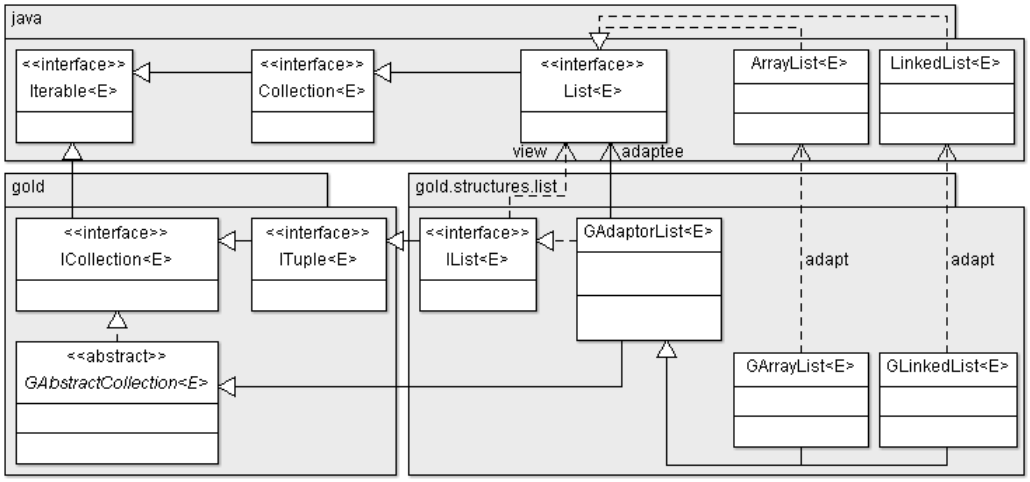


Clases que conforman el paquete `gold.structures.tuple`.

Clase	Descripción
`ITuple<E>`	Interfaz que representa una *n*-tupla finita de elementos de tipo `E` (una secuencia ordenada de *n* elementos de tipo `E` donde *n* es un número natural fijo).
`INullTuple<E>`	Interfaz que representa una 0-tupla de elementos de tipo `E` (una secuencia vacía de elementos de tipo `E`).
`ISingleton<E>`	Interfaz que representa una 1-tupla de elementos de tipo `E` (una secuencia con un elemento de tipo `E`).
`IPair<E>`	Interfaz que representa una 2-tupla de elementos de tipo `E` (un par ordenado con dos elementos de tipo `E`).
`ICouple<T,U>`	Interfaz que representa una 2-tupla de elementos de diferente tipo (una pareja de elementos donde el primer elemento es de tipo `T` y el segundo es de tipo `U`).
`GTuple<E>`	Clase que adapta una lista *Java* de tipo `java.util.List<E>` (o en su defecto, un arreglo de tipo `E[]`) como una tupla *GOLD* de tipo `ITuple<E>` mediante la aplicación del patrón *Adapter*.
`GNullTuple<E>`	Clase que implementa la interfaz `INullTuple<E>`.
`GSingleton<E>`	Clase que implementa la interfaz `ISingleton<E>` a través de un apuntador a un elemento de tipo `E`.
`GPair<E>`	Clase que implementa la interfaz `IPair<E>` a través de dos apuntadores a elementos de tipo `E`.
`GCouple<T,U>`	Clase que implementa la interfaz `ICouple<T,U>` a través de dos apuntadores a un elemento de tipo `T` y a un elemento de tipo `U`, respectivamente.

A.6.2.3. Paquete *gold.structures.list*

Diagrama de clases del paquete `gold.structures.list`.

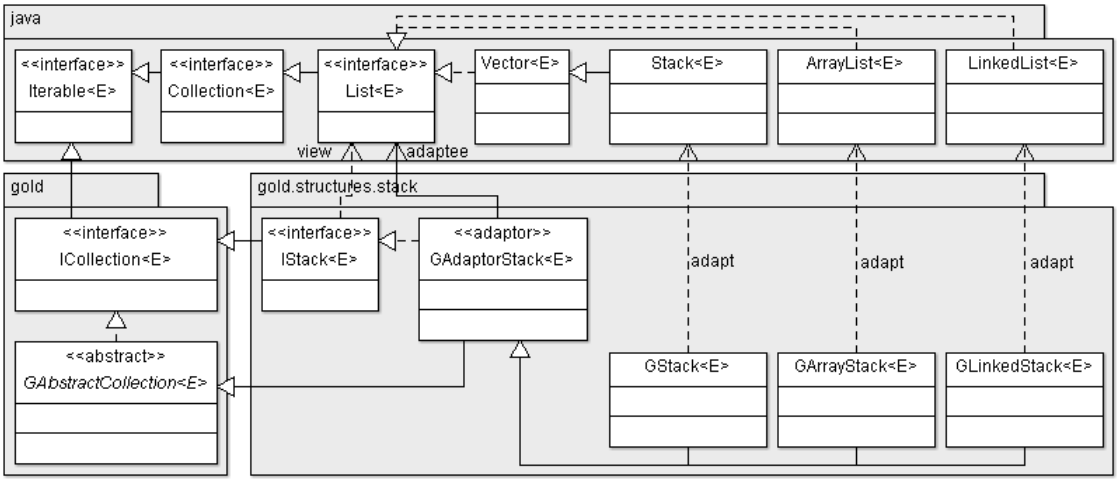


Clases que conforman el paquete `gold.structures.list`.

Clase	Descripción
`IList<E>`	Interfaz que representa una lista finita (i.e., una secuencia ordenada) de elementos de tipo `E`.
`GAdaptorList<E>`	Clase que adapta una lista *Java* de tipo `java.util.List<E>` como una lista *GOLD* de tipo `IList<E>` mediante la aplicación del patrón *Adapter*.
`GArrayList<E>`	Clase que implementa la interfaz `IList<E>` con vectores dinámicos (arreglos) de tamaño variable, adaptando la clase `java.util.ArrayList<E>` de *Java*.
`GLinkedList<E>`	Clase que implementa la interfaz `IList<E>` con una estructura lineal doblemente encadenada en anillo con encabezado, adaptando la clase `java.util.LinkedList<E>` de *Java*.

A.6.2.4. Paquete *gold.structures.stack*

Diagrama de clases del paquete `gold.structures.stack`.

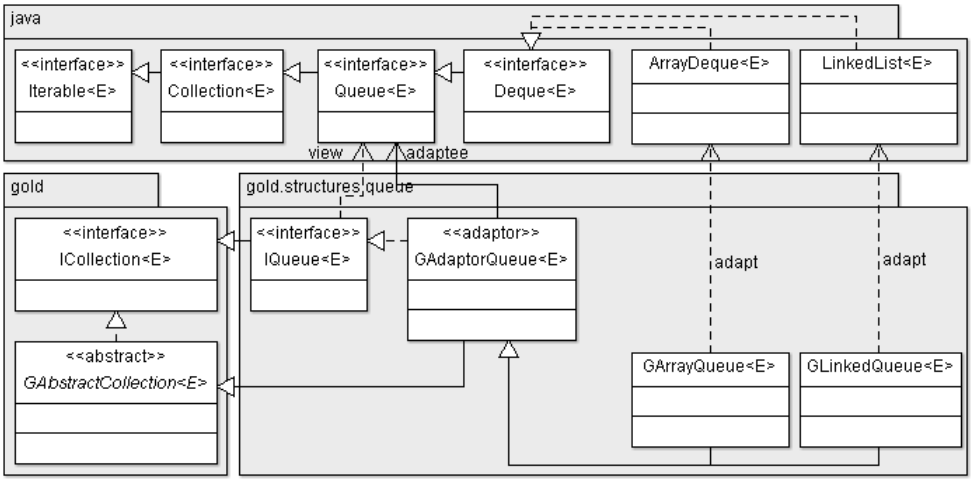


Clases que conforman el paquete `gold.structures.stack`.

Clase	Descripción
`IStack<E>`	Interfaz que representa una pila finita de elementos de tipo `E`.
`GAdaptorStack<E>`	Clase que adapta una lista *Java* de tipo `java.util.List<E>` como una pila *GOLD* de tipo `IStack<E>` mediante la aplicación del patrón *Adapter*.
`GStack<E>`	Clase que implementa la interfaz `IStack<E>` con vectores dinámicos (arreglos) de tamaño variable, adaptando la clase `java.util.Stack<E>` de *Java*.
`GArrayStack<E>`	Clase que implementa la interfaz `IStack<E>` con vectores dinámicos (arreglos) de tamaño variable, adaptando la clase `java.util.ArrayList<E>` de *Java*.
`GLinkedStack<E>`	Clase que implementa la interfaz `IStack<E>` con una estructura lineal doblemente encadenada en anillo con encabezado, adaptando la clase `java.util.LinkedList<E>` de *Java*.

A.6.2.5. Paquete *gold.structures.queue*

Diagrama de clases del paquete `gold.structures.queue`.

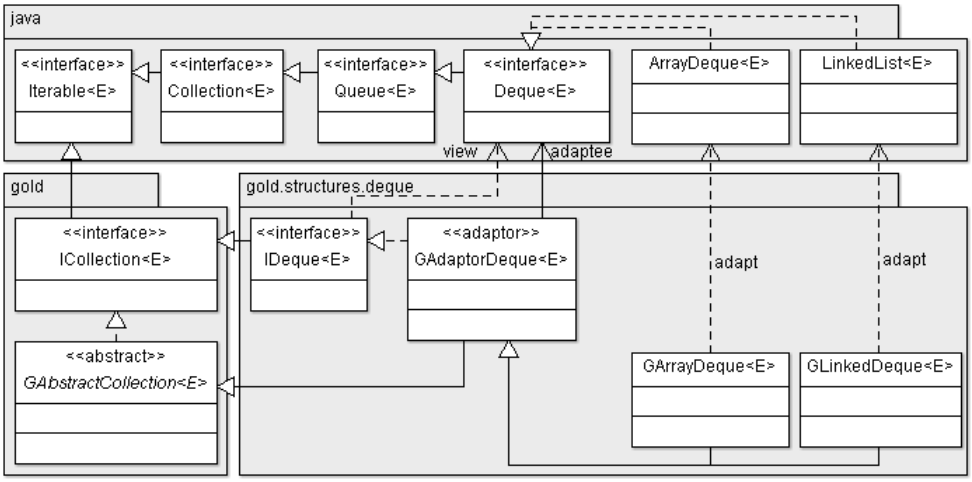


Clases que conforman el paquete `gold.structures.queue`.

Clase	Descripción
`IQueue<E>`	Interfaz que representa una cola finita de elementos de tipo `E`.
`GAdaptorQueue<E>`	Clase que adapta una cola *Java* de tipo `java.util.Queue<E>` como una cola *GOLD* de tipo `IQueue<E>` mediante la aplicación del patrón *Adapter*.
`GArrayQueue<E>`	Clase que implementa la interfaz `IQueue<E>` con vectores dinámicos (arreglos) circulares de tamaño variable, adaptando la clase `java.util.ArrayDeque<E>` de *Java*.
`GLinkedQueue<E>`	Clase que implementa la interfaz `IQueue<E>` con una estructura lineal doblemente encadenada en anillo con encabezado, adaptando la clase `java.util.LinkedList<E>` de *Java*.

A.6.2.6. Paquete *gold.structures.deque*

Diagrama de clases del paquete `gold.structures.deque`.

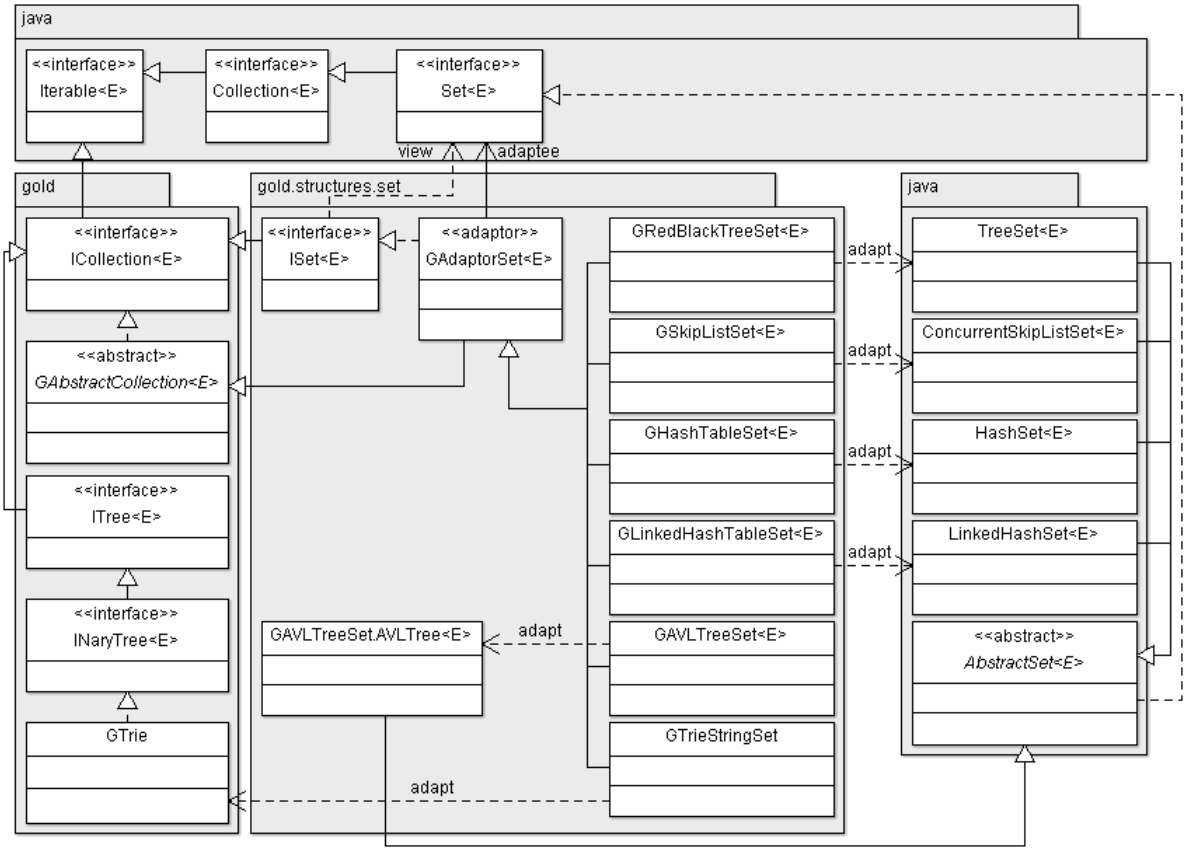


Clases que conforman el paquete `gold.structures.deque`.

Clase	Descripción
`IDeque<E>`	Interfaz que representa una bicola finita de elementos de tipo `E`.
`GAdaptorDeque<E>`	Clase que adapta una bicola *Java* de tipo `java.util.Deque<E>` como una bicola *GOLD* de tipo `IDeque<E>` mediante la aplicación del patrón *Adapter*.
`GArrayDeque<E>`	Clase que implementa la interfaz `IDeque<E>` con vectores dinámicos (arreglos) circulares de tamaño variable, adaptando la clase `java.util.ArrayDeque<E>` de *Java*.
`GLinkedDeque<E>`	Clase que implementa la interfaz `IDeque<E>` con una estructura lineal doblemente encadenada en anillo con encabezado, adaptando la clase `java.util.LinkedList<E>` de *Java*.

A.6.2.7. Paquete *gold.structures.set*

Diagrama de clases del paquete `gold.structures.set`.

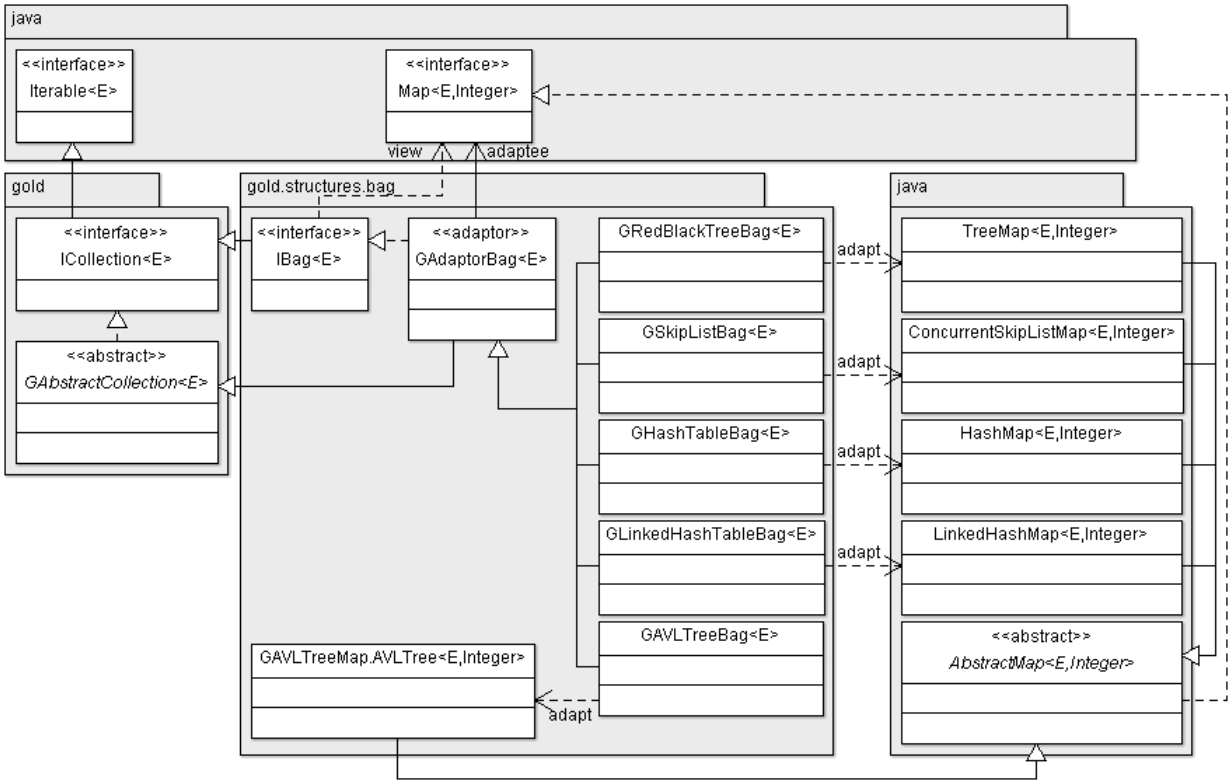


Clases que conforman el paquete `gold.structures.set`.

Clase	Descripción
`ISet<E>`	Interfaz que representa un conjunto (finito o con complemento finito) de elementos de tipo `E`.
`GAdaptorSet<E>`	Clase que adapta un conjunto *Java* de tipo `java.util.Set<E>` como un conjunto *GOLD* de tipo `ISet<E>` mediante la aplicación del patrón *Adapter*.
`GAVLTreeSet<E>`	Clase que implementa la interfaz `ISet<E>` con Árboles *AVL*.
`GHashTableSet<E>`	Clase que implementa la interfaz `ISet<E>` con Tablas de *Hashing*, adaptando la clase `java.util.HashSet<E>` de *Java*.
`GLinkedHashTableSet<E>`	Clase que implementa la interfaz `ISet<E>` con Tablas de *Hashing* con orden de iteración predecible, adaptando la clase `java.util.LinkedHashSet<E>` de *Java*.
`GRedBlackTreeSet<E>`	Clase que implementa la interfaz `ISet<E>` con Árboles Rojinegros, adaptando la clase `java.util.TreeSet<E>` de *Java*.
`GSkipListSet<E>`	Clase que implementa la interfaz `ISet<E>` con *Skip Lists*, adaptando la clase `java.util.concurrent.ConcurrentSkipListSet<E>` de *Java*.
`GTrieStringSet`	Clase que representa un conjunto de cadenas de texto de tipo `ISet<String>` implementado con *Tries*.

A.6.2.8. Paquete *gold.structures.bag*

Diagrama de clases del paquete `gold.structures.bag`.

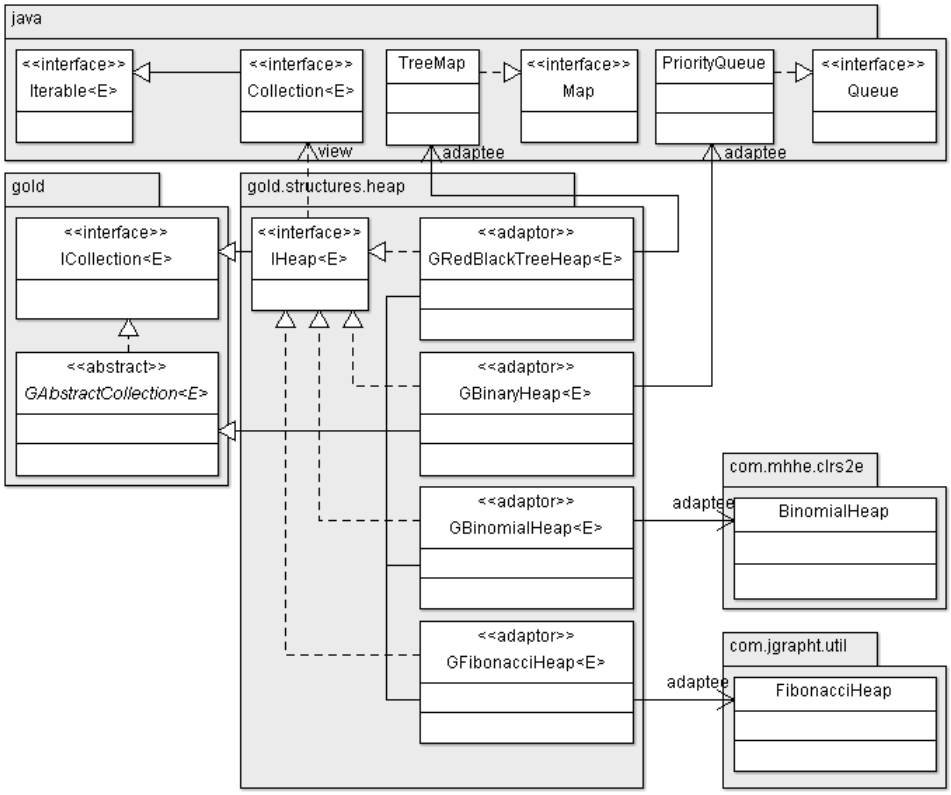


Clases que conforman el paquete `gold.structures.bag`.

Clase	Descripción
`IBag<E>`	Interfaz que representa una bolsa finita (i.e., un multiconjunto) de elementos de tipo `E`.
`GAdaptorBag<E>`	Clase que adapta una asociación llave-valor *Java* de tipo `java.util.Map<E,Integer>` como una bolsa *GOLD* de tipo `IBag<E>` mediante la aplicación del patrón *Adapter*.
`GAVLTreeBag<E>`	Clase que implementa la interfaz `IBag<E>` con Árboles *AVL*.
`GHashTableBag<E>`	Clase que implementa la interfaz `IBag<E>` con Tablas de *Hashing*, adaptando instancias de tipo `java.util.HashMap<E,Integer>` en *Java*.
`GLinkedHashTableBag<E>`	Clase que implementa la interfaz `IBag<E>` con Tablas de *Hashing* con orden de iteración predecible, adaptando instancias de tipo `java.util.LinkedHashMap<E,Integer>` de *Java*.
`GRedBlackTreeBag<E>`	Clase que implementa la interfaz `IBag<E>` con Árboles Rojinegros, adaptando instancias de tipo `java.util.TreeMap<E,Integer>` de *Java*.
`GSkipListBag<E>`	Clase que implementa la interfaz `IBag<E>` con *Skip Lists*, adaptando instancias de tipo `java.util.concurrent.ConcurrentSkipListMap<E,Integer>` de *Java*.

A.6.2.9. Paquete *gold.structures.heap*

Diagrama de clases del paquete `gold.structures.heap`.

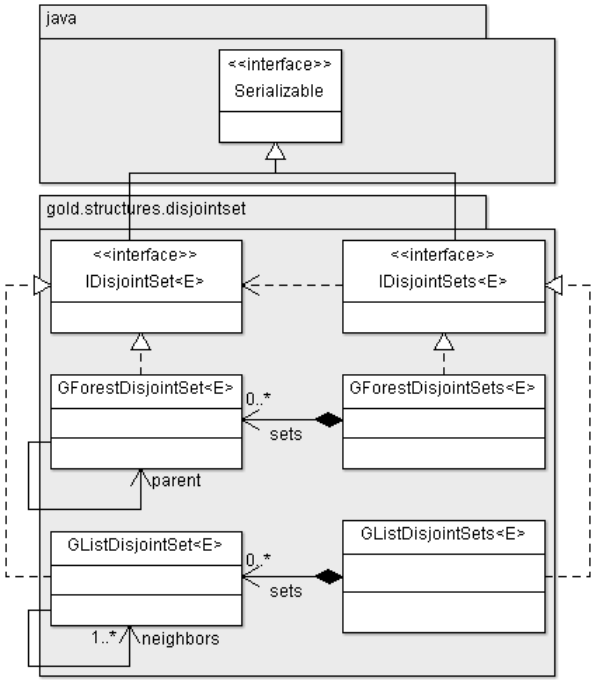


Clases que conforman el paquete `gold.structures.heap`.

Clase	Descripción
`IHeap<E>`	Interfaz que representa un montón finito de elementos de tipo `E`.
`GBinaryHeap<E>`	Clase que implementa la interfaz `IHeap<E>` con *Binary Heaps* [1], adaptando la clase `java.util.PriorityQueue` de *Java*.
`GBinomialHeap<E>`	Clase que implementa la interfaz `IHeap<E>` con *Binomial Heaps* [1], adaptando la clase `com.mhhe.clrs2e.BinomialHeap` de la librería de referencia de Cormen et al. [23].
`GFibonacciHeap<E>`	Clase que implementa la interfaz `IHeap<E>` con *Fibonacci Heaps* [1], adaptando la clase `com.jgrapht.util.FibonacciHeap` de la librería *JGraphT* [22].
`GRedBlackTreeHeap<E>`	Clase que implementa la interfaz `IHeap<E>` con Árboles Rojinegros, adaptando la clase `java.util.TreeMap` de *Java*.

A.6.2.10. Paquete *gold.structures.disjointset*

Diagrama de clases del paquete `gold.structures.disjointset`.

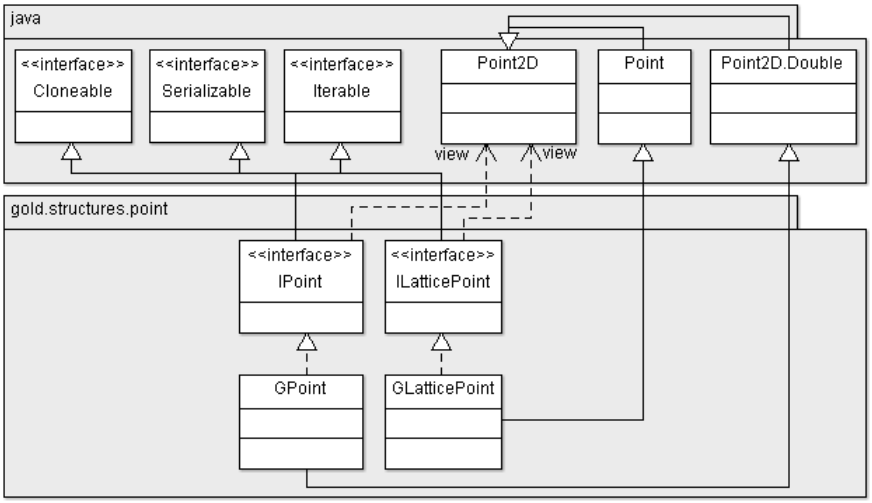


Clases que conforman el paquete `gold.structures.disjointset`.

Clase	Descripción
`IDisjointSet<E>`	Interfaz que representa un conjunto de elementos de tipo `E` bajo una estructura de conjuntos disyuntos (*disjoint-set data structure*) [1].
`IDisjointSets<E>`	Interfaz que representa una estructura de conjuntos disyuntos (*disjoint-set data structure*) [1] de elementos de tipo `E`.
`GForestDisjointSet<E>`	Clase que implementa la interfaz `IDisjointSet<E>` bajo la implementación con *disjoint-set forests* [1].
`GForestDisjointSets<E>`	Clase que implementa la interfaz `IDisjointSets<E>` con *disjoint-set forests* [1].
`GListDisjointSet<E>`	Clase que implementa la interfaz `IDisjointSet<E>` bajo la implementación con listas encadenadas [1].
`GListDisjointSets<E>`	Clase que implementa la interfaz `IDisjointSets<E>` con una representación basada en listas encadenadas [1].

A.6.2.11. Paquete *gold.structures.point*

Diagrama de clases del paquete `gold.structures.point`.

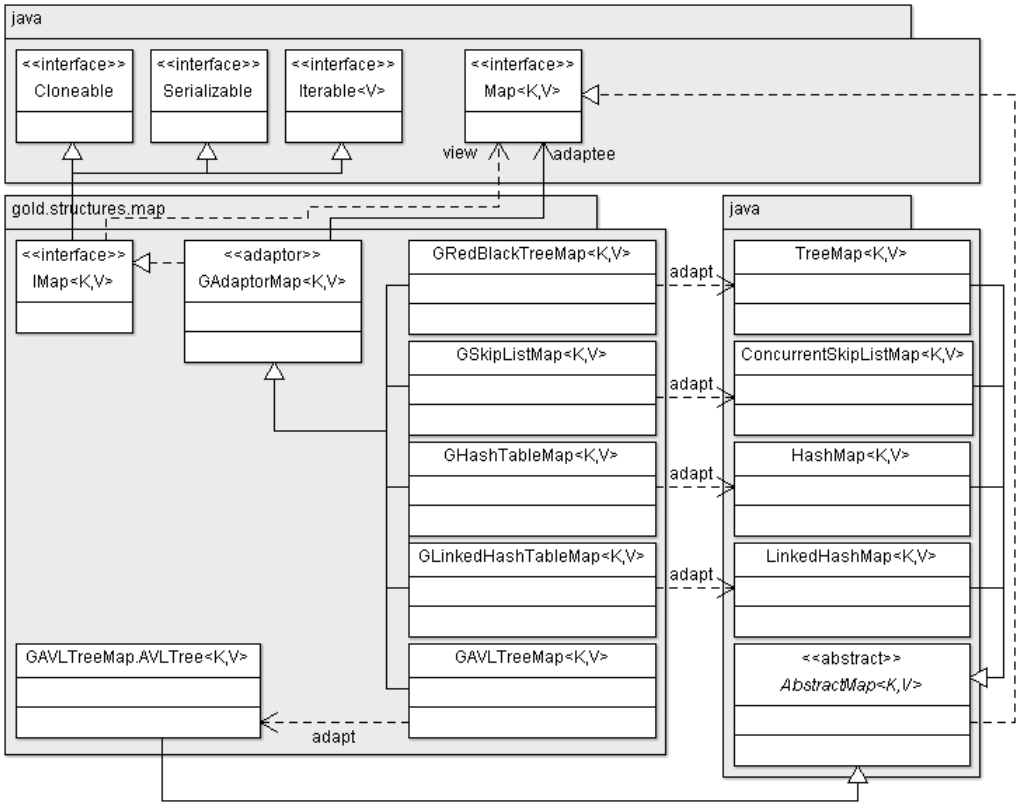


Clases que conforman el paquete `gold.structures.point`.

Clase	Descripción
`IPoint`	Interfaz que representa un punto perteneciente al plano cartesiano bidimensional.
`ILatticePoint`	Interfaz que representa un punto con coordenadas enteras perteneciente al plano cartesiano bidimensional.
`GPoint`	Clase que implementa la interfaz `IPoint<E>` usando el patrón *Delegation* para adaptar la clase `java.awt.geom.Point2D.Double` de *Java*.
`GLatticePoint`	Clase que implementa la interfaz `ILatticePoint<E>` usando el patrón *Delegation* para adaptar la clase `java.awt.Point` de *Java*.

A.6.2.12. Paquete *gold.structures.map*

Diagrama de clases del paquete `gold.structures.map`.

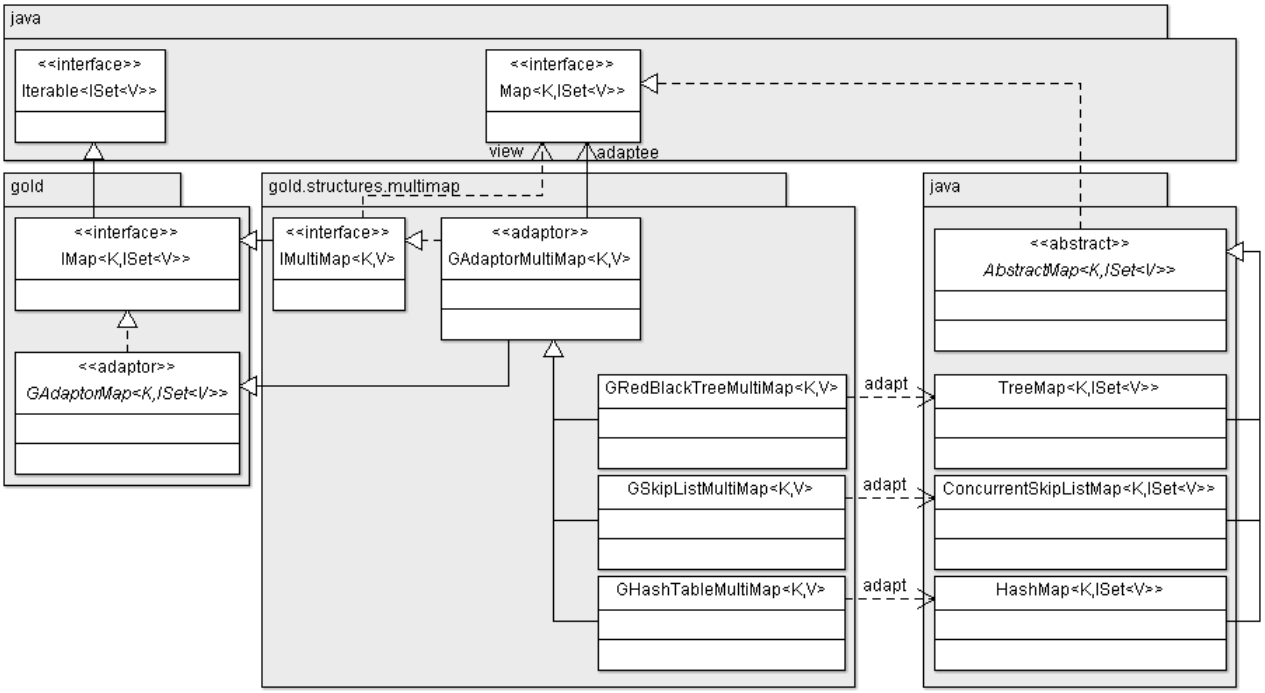


Clases que conforman el paquete `gold.structures.map`.

Clase	Descripción
`IMap<K,V>`	Interfaz que representa una asociación llave-valor cuyas llaves son elementos de tipo `K` y cuyos valores son elementos de tipo `V`.
`GAdaptorMap<K,V>`	Clase que adapta una asociación llave-valor *Java* de tipo `java.util.Map<K,V>` como una asociación llave-valor *GOLD* de tipo `IMap<K,V>` mediante la aplicación del patrón *Adapter*.
`GAVLTreeMap<K,V>`	Clase que implementa la interfaz `IMap<K,V>` con Árboles *AVL*.
`GHashTableMap<K,V>`	Clase que implementa la interfaz `IMap<K,V>` con Tablas de *Hashing*, adaptando la clase `java.util.HashMap<K,V>` de *Java*.
`GLinkedHashTableMap<K,V>`	Clase que implementa la interfaz `IMap<K,V>` con Tablas de *Hashing* con orden de iteración predecible, adaptando la clase `java.util.LinkedHashMap<K,V>` de *Java*.
`GRedBlackTreeMap<K,V>`	Clase que implementa la interfaz `IMap<K,V>` con Árboles Rojinegros, adaptando la clase `java.util.TreeMap<K,V>` de *Java*.
`GSkipListMap<K,V>`	Clase que implementa la interfaz `IMap<K,V>` con *Skip Lists*, adaptando la clase `java.util.concurrent.ConcurrentSkipListMap<K,V>` de *Java*.

A.6.2.13. Paquete *gold.structures.multimap*

Diagrama de clases del paquete `gold.structures.multimap`.

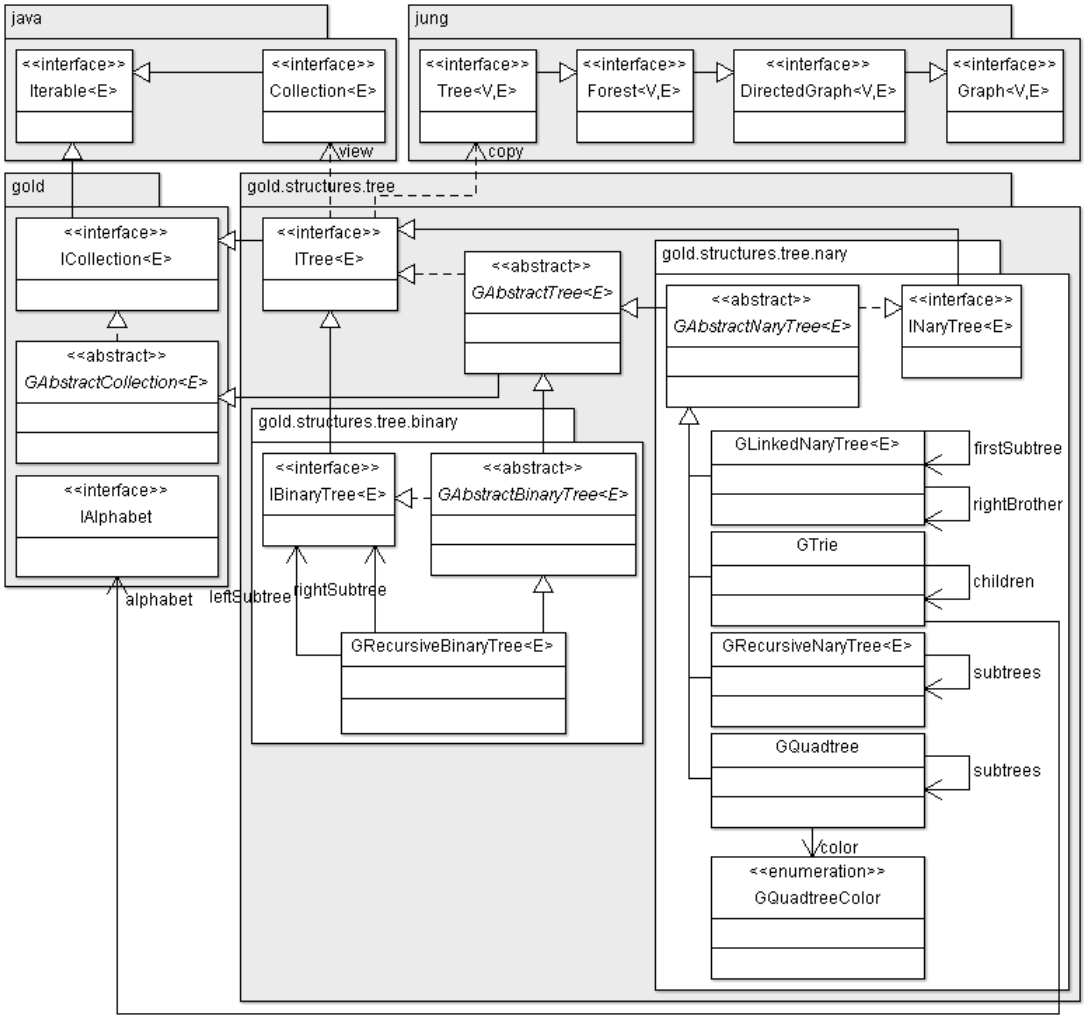


Clases que conforman el paquete `gold.structures.multimap`.

Clase	Descripción
`IMultiMap<K,V>`	Interfaz que representa una asociación llave-valor cuyas llaves son elementos de tipo `K` y cuyos valores son conjuntos de tipo `ISet<V>`.
`GAdaptorMultiMap<K,V>`	Clase que adapta una asociación llave-valor *Java* de tipo `java.util.Map<K,ISet<V>>` como una asociación llave-valor *GOLD* de tipo `IMultiMap<K,V>` mediante la aplicación del patrón *Adapter*.
`GHashTableMultiMap<K,V>`	Clase que implementa la interfaz `IMultiMap<K,V>` con Tablas de *Hashing*, adaptando instancias de la clase `java.util.HashMap<K,ISet<V>>` de *Java*.
`GRedBlackTreeMultiMap<K,V>`	Clase que implementa la interfaz `IMultiMap<K,V>` con Árboles Rojinegros, adaptando instancias de la clase `java.util.TreeMap<K,ISet<V>>` de *Java*.
`GSkipListMultiMap<K,V>`	Clase que implementa la interfaz `IMultiMap<K,V>` con *Skip Lists*, adaptando instancias de la clase `java.util.concurrent.ConcurrentSkipListMap<K,ISet<V>>` de *Java*.

A.6.2.14. Paquete *gold.structures.tree*

Diagrama de clases del paquete `gold.structures.tree`.



Clases que conforman el paquete `gold.structures.tree`.

Clase	Descripción
`ITree<E>`	Interfaz que representa un árbol finito de elementos de tipo `E`.
`GAbstractTree<E>`	Clase abstracta que provee una implementación base de la interfaz `ITree<E>` para reducir el esfuerzo que se debe realizar para implementar un árbol *GOLD*.

Clases que conforman el paquete `gold.structures.tree.binary`.

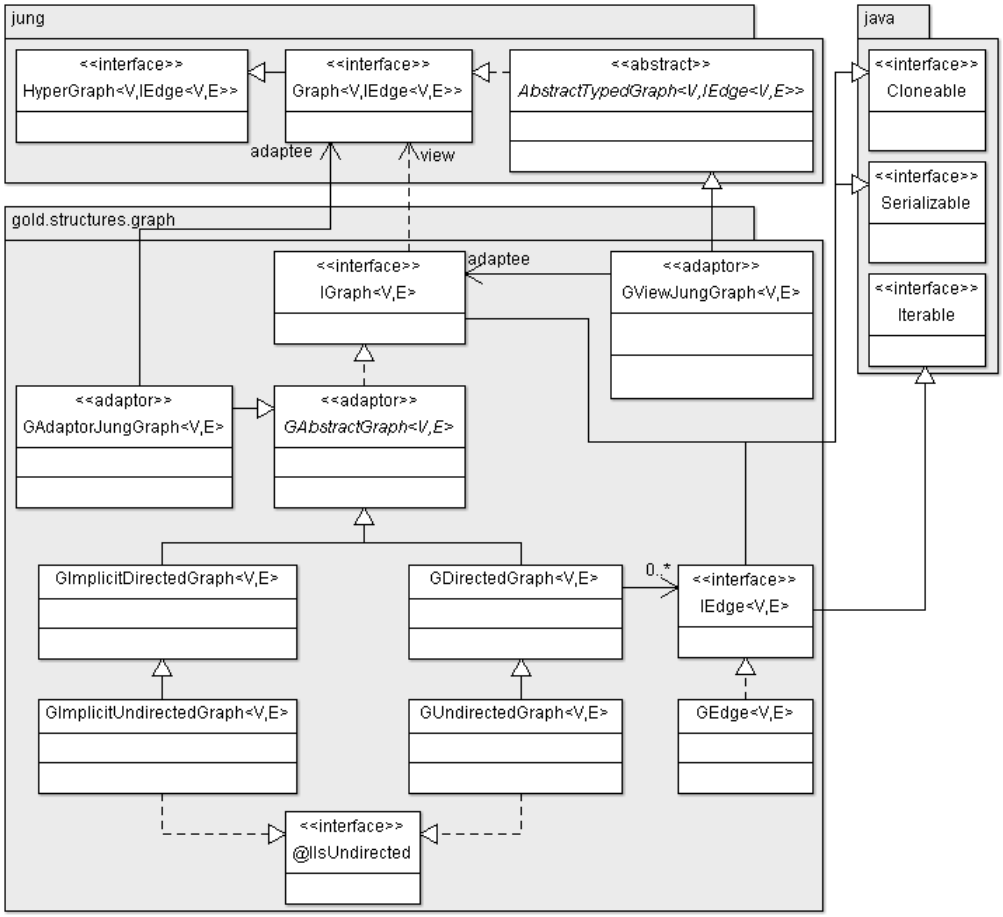
Clase	Descripción
`IBinaryTree<E>`	Interfaz que representa un árbol binario finito de elementos de tipo `E`.
`GAbstractBinaryTree<E>`	Clase abstracta que provee una implementación base de la interfaz `IBinaryTree<E>` para reducir el esfuerzo que se debe realizar para implementar un árbol binario *GOLD*.
`GRecursiveBinaryTree<E>`	Clase que implementa la interfaz `IBinaryTree<E>` con una estructura recursiva con apuntadores al subárbol izquierdo y al subárbol derecho de cada nodo.

Clases que conforman el paquete `gold.structures.tree.nary`.

Clase	**Descripción**
`INaryTree<E>`	Interfaz que representa un árbol eneario finito de elementos de tipo `E`.
`GAbstractNaryTree<E>`	Clase abstracta que provee una implementación base de la interfaz `INaryTree<E>` para reducir el esfuerzo que se debe realizar para implementar un árbol eneario *GOLD*.
`GRecursiveNaryTree<E>`	Clase que implementa la interfaz `IBinaryTree<E>` con una estructura recursiva con apuntadores a los subárboles de cada nodo.
`GLinkedNaryTree<E>`	Clase que implementa la interfaz `IBinaryTree<E>` con una estructura recursiva con apuntadores al primer subárbol y al hermano derecho de cada nodo.
`GQuadtree`	Clase que representa un *Quadtree* de tipo `INaryTree<GQuadtreeColor>` implementado con una estructura recursiva.
`GQuadtreeColor`	Enumeración que representa los posibles colores de un nodo de un *Quadtree*: blanco, negro y gris.
`GTrie`	Clase que implementa un *Trie* representando un conjunto de cadenas de texto.

A.6.2.15. Paquete *gold.structures.graph*

Diagrama de clases del paquete `gold.structures.graph`.



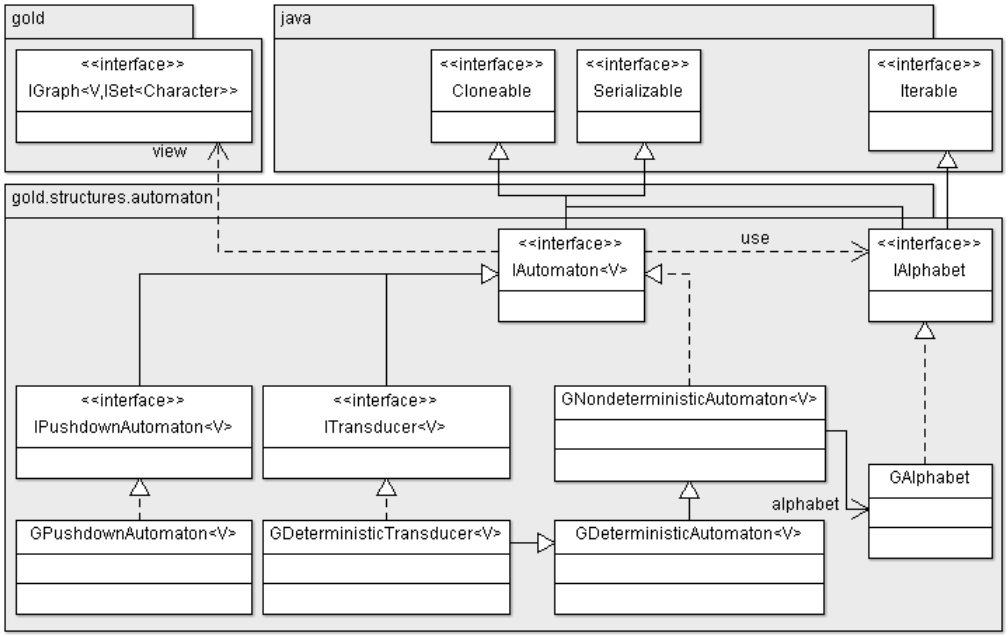
Clases que conforman el paquete `gold.structures.graph`.

Clase	Descripción
`IGraph<V,E>`	Interfaz que representa un grafo cuyos vértices son elementos de tipo `V` y cuyos arcos tienen valores (datos almacenados) de tipo `E`.
`IEdge<V,E>`	Interfaz que representa un arco que conecta elementos de tipo `V` y cuyo valor (dato almacenado) es de tipo `E`.
`@IIsUndirected`	Anotación para distinguir los grafos no dirigidos.
`GAbstractGraph<V,E>`	Clase abstracta que provee una implementación base de la interfaz `IGraph<V,E>` para reducir el esfuerzo que se debe realizar para implementar un grafo *GOLD*.
`GAdaptorJungGraph<V,E>`	Clase que adapta un grafo *JUNG* de tipo `edu.uci.ics.jung.graph.Graph<V,E>` como un grafo *GOLD* de tipo `IGraph<V,E>` mediante la aplicación del patrón *Adapter*.
`GViewJungGraph<V,E>`	Clase que adapta un grafo *GOLD* de tipo `IGraph<V,E>` como un grafo *JUNG* de tipo `edu.uci.ics.jung.graph.Graph<V,IEdge<V,E>>` mediante la aplicación del patrón *Adapter*.
`GEdge<V,E>`	Clase que implementa la interfaz `IEdge<V,E>`, almacenando el origen, el destino, el costo y el valor (dato almacenado) del nodo.
`GDirectedGraph<V,E>`	Clase que implementa la interfaz `IGraph<V,E>` como un grafo dirigido representado con listas de adyacencia.

`GUndirectedGraph<V,E>`	Clase que implementa la interfaz `IGraph<V,E>` como un grafo no dirigido representado con listas de adyacencia.
`GImplicitDirectedGraph<V,E>`	Clase que implementa la interfaz `IGraph<V,E>` como un grafo dirigido de representación implícita donde los sucesores de cada nodo y el costo de cada arco deben ser configurados a través de funciones provistas por el programador.
`GImplicitUndirectedGraph<V,E>`	Clase que implementa la interfaz `IGraph<V,E>` como un grafo no dirigido de representación implícita donde los sucesores de cada nodo y el costo de cada arco deben ser configurados a través de funciones provistas por el programador.

A.6.2.16. Paquete *gold.structures.automaton*

Diagrama de clases del paquete `gold.structures.automaton`.

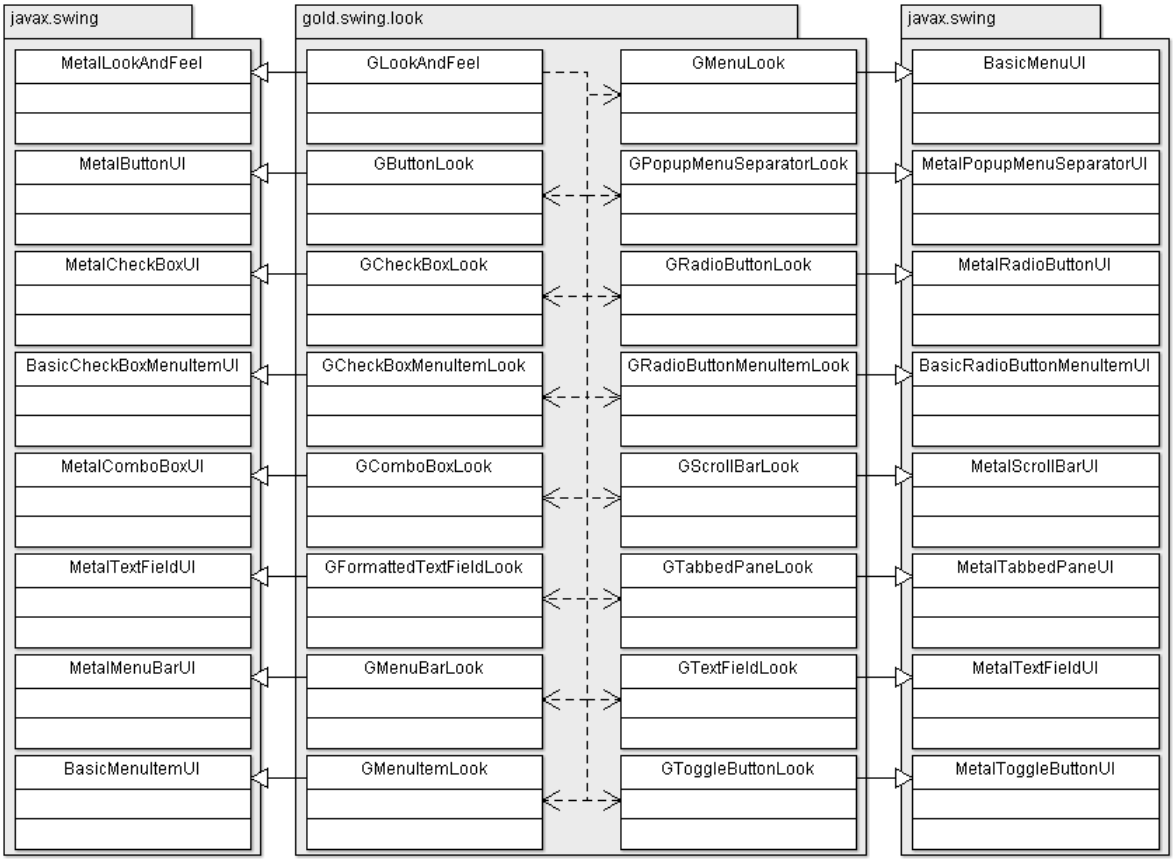


Clases que conforman el paquete `gold.structures.automaton`.

Clase	Descripción
`IAlphabet`	Interfaz que representa un alfabeto de caracteres (i.e., un conjunto finito de símbolos).
`IAutomaton<V>`	Interfaz que representa un autómata finito cuyos vértices son elementos de tipo `V`.
`ITransducer<V>`	Interfaz que representa un autómata finito determinístico con respuesta (en las transiciones y/o en los estados) cuyos vértices son elementos de tipo `V`.
`IPushdownAutomaton<V>`	Interfaz que representa un autómata finito de pila cuyos vértices son elementos de tipo `V`.
`GAlphabet`	Clase que implementa la interfaz `IAlphabet` con un arreglo de caracteres *Unicode*.
`GNondeterministicAutomaton<V>`	Clase que representa un autómata finito no determinístico mediante conjuntos y asociaciones llave-valor, implementando la interfaz *IAutomaton<V>*. La función de transición puede ser configurada a través de rutinas provistas por el programador.
`GDeterministicAutomaton<V>`	Clase que representa un autómata finito determinístico mediante conjuntos y asociaciones llave-valor, implementando la interfaz *IAutomaton<V>*. La función de transición puede ser configurada a través de rutinas provistas por el programador.
`GDeterministicTransducer<V>`	Clase que representa un autómata finito determinístico con respuesta (en las transiciones y/o en los estados) mediante conjuntos y asociaciones llave-valor, implementando la interfaz *ITransducer<V>*. La función de transición, la función de salida en las transiciones y la función de salida en los estados pueden ser configuradas a través de rutinas provistas por el programador.
`GPushdownAutomaton<V>`	Clase que representa un autómata de pila mediante conjuntos y asociaciones llave-valor, implementando la interfaz *IPushdownAutomaton<V>*.

A.6.2.17. Paquete *gold.swing.look*

Diagrama de clases del paquete `gold.swing.look`.

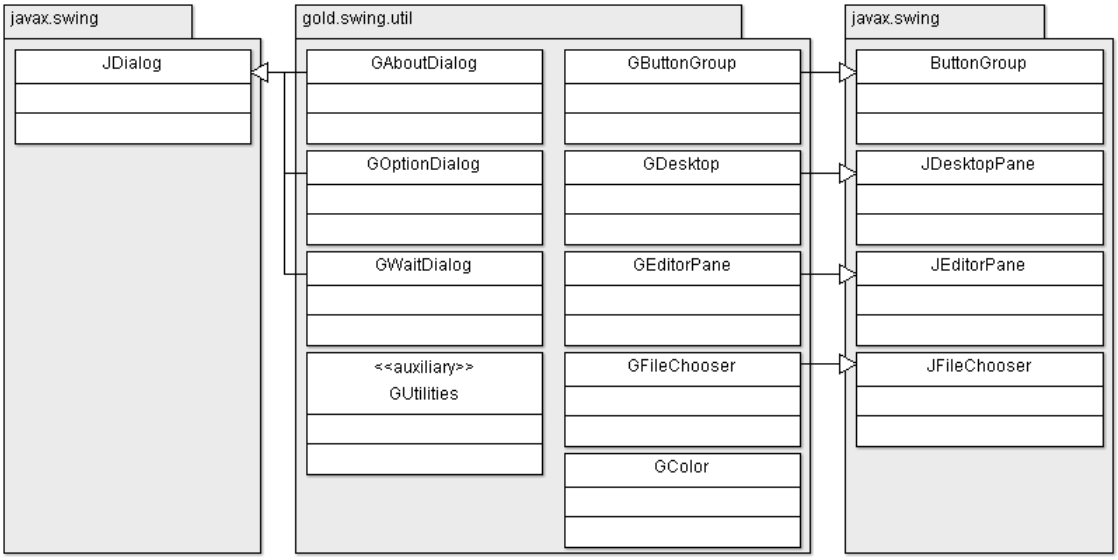


Clases que conforman el paquete `gold.swing.look`.

Clase	**Descripción**
`GLookAndFeel`	Clase que configura el *Look and Feel* de los componentes gráficos de *GOLD*.
`GButtonLook`	Clase que configura la apariencia de los botones (*button*).
`GCheckBoxLook`	Clase que configura la apariencia de los campos de verificación (*check box*).
`GCheckBoxMenuItemLook`	Clase que configura la apariencia de los campos de verificación de barras de menú (*check box menu item*).
`GComboBoxLook`	Clase que configura la apariencia de las listas desplegables (*combo box*).
`GFormattedTextFieldLook`	Clase que configura la apariencia de los campos con formato (*formatted text field*).
`GMenuBarLook`	Clase que configura la apariencia de las barras de menú (*menu bar*).
`GMenuItemLook`	Clase que configura la apariencia de los botones de las barras de menú (*menu item*).
`GMenuLook`	Clase que configura la apariencia de los menús desplegables (*menu popup*).
`GPopupMenuSeparatorLook`	Clase que configura la apariencia de los separadores de menú (*menu separator*).
`GRadioButtonLook`	Clase que configura la apariencia de los botones de elección (*radio button*).
`GRadioButtonMenuItemLook`	Clase que configura la apariencia de los botones de elección de barras de menú (*radio button menu item*).
`GScrollBarLook`	Clase que configura la apariencia de las barras de desplazamiento (*scroll bar*).
`GTabbedPaneLook`	Clase que configura la apariencia de los paneles con pestañas (*tabbed pane*).
`GTextFieldLook`	Clase que configura la apariencia de los campos de texto (*text field*).
`GToggleButtonLook`	Clase que configura la apariencia de los botones de pulsación (*toggle button*).

A.6.2.18. Paquete *gold.swing.util*

Diagrama de clases del paquete `gold.swing.util`.

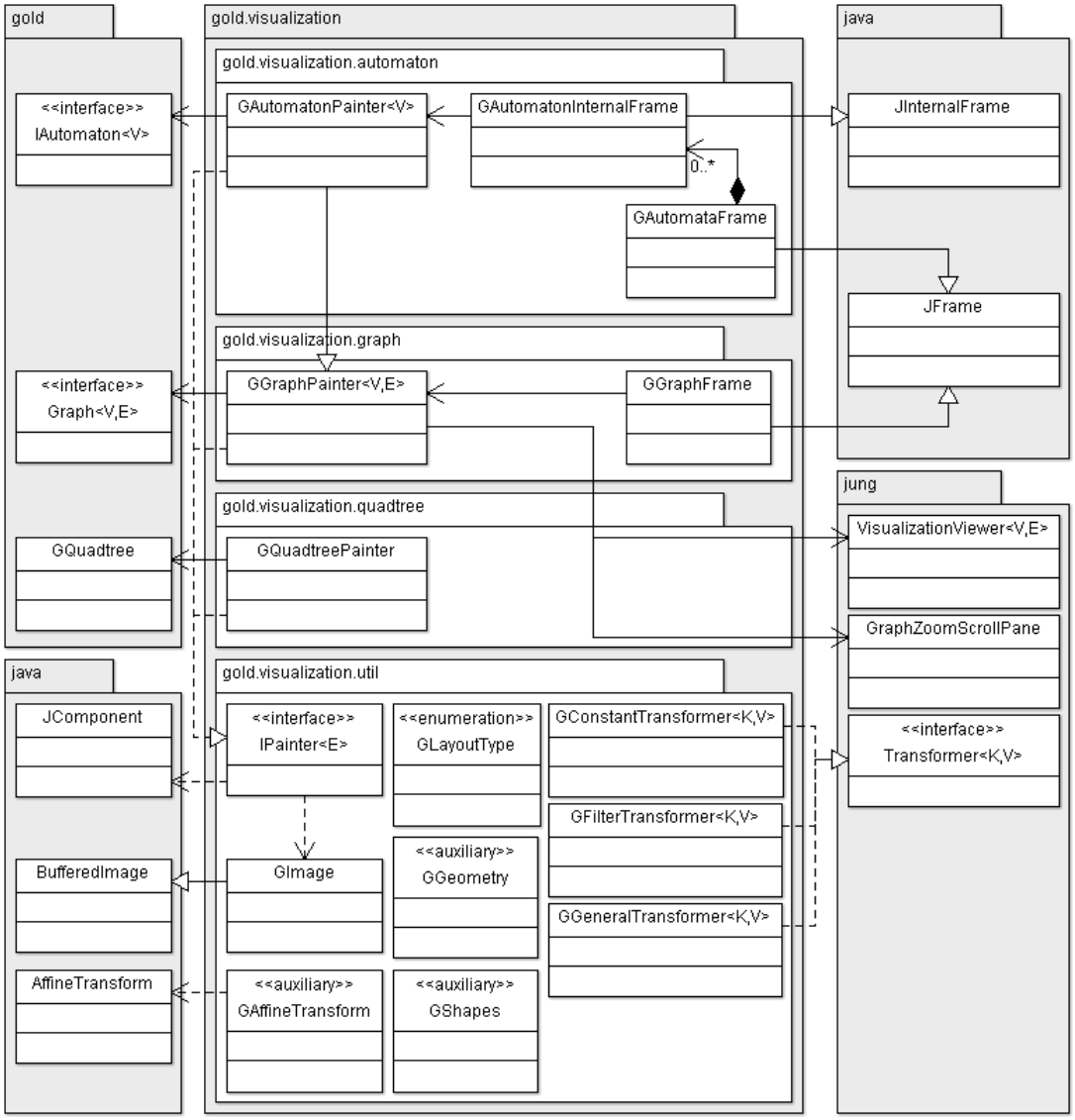


Clases que conforman el paquete `gold.swing.util`.

Clase	Descripción
`GAboutDialog`	Clase que representa la ventana *Acerca de* la aplicación (*About GOLD*).
`GButtonGroup`	Clase que representa un conjunto de botones de elección (*radio buttons*) mutuamente excluyentes.
`GColor`	Clase que enumera los colores de sistema usados para dar tonalidad a los componentes gráficos del *Look and Feel* de *GOLD*.
`GDesktop`	Clase que representa un componente gráfico usado para crear un escritorio virtual (`JDesktop`) compuesto por ventanas internas (`JInternalFrame`).
`GEditorPane`	Clase que representa un campo de texto capaz de desplegar código *HTML*.
`GFileChooser`	Clase que representa un diálogo diseñado para escoger archivos del sistema operativo.
`GOptionDialog`	Clase que representa un mensaje de diálogo diseñado para desplegar errores, advertencias, información útil o preguntas.
`GUtilities`	Clase que contiene una colección de rutinas útiles relacionadas con el entorno gráfico. Implementa el patrón *Utility*.
`GWaitDialog`	Clase que representa una ventana de espera que sirve de fondo para el desarrollo de alguna operación sincrónica que demande un tiempo considerable.

A.6.2.19. Paquete *gold.visualization*

Diagrama de clases del paquete `gold.visualization`.



Clases que conforman el paquete `gold.visualization.automaton`.

Clase	Descripción
`GAutomataFrame`	Clase que representa la ventana que despliega el visualizador de autómatas.
`GAutomatonInternalFrame`	Clase que representa una ventana interna (`JInternalFrame`) capaz de presentar gráficamente un autómata.
`GAutomatonPainter<V>`	Clase responsable de renderizar un autómata a través de la librería *JUNG*.

Clases que conforman el paquete `gold.visualization.graph`.

Clase	Descripción
`GGraphFrame`	Clase que representa una ventana capaz de desplegar gráficamente un grafo.
`GGraphPainter<V,E>`	Clase responsable de renderizar un grafo a través de la librería *JUNG*.

Clases que conforman el paquete `gold.visualization.quadtree`.

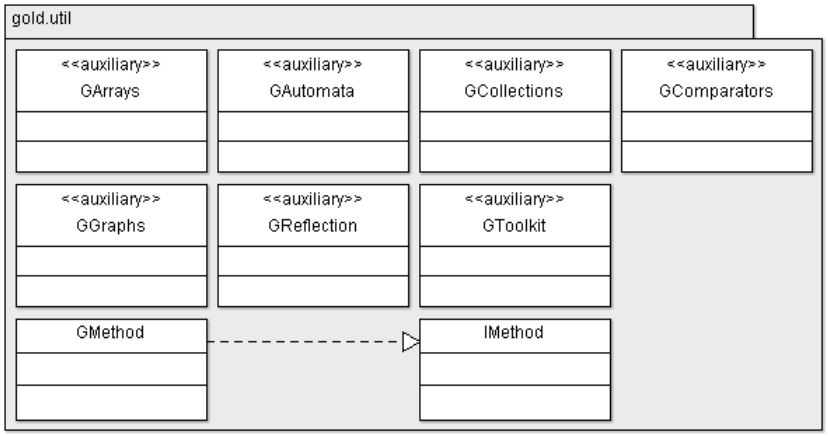
Clase	Descripción
`GQuadtreePainter`	Clase responsable de renderizar un *Quadtree* como una imagen blanco y negro.

Clases que conforman el paquete `gold.visualization.util`.

Clase	Descripción
`IPainter<E>`	Interfaz que representa un renderizador de estructuras de datos.
`GAffineTransforms`	Clase que contiene una colección de rutinas útiles para crear transformaciones afines. Implementa el patrón *Utility*.
`GConstantTransformer<K,V>`	Clase que representa un transformador especializado de valores que implementa `org.apache.commons.collections15.Transformer<K,V>`, donde para cada llave de tipo `K` se retorna un valor constante de tipo `V`.
`GFilterTransformer<K,V>`	Clase que representa un transformador especializado de valores que implementa `org.apache.commons.collections15.Transformer<K,V>`, donde para cada llave *k* de tipo `K` se retorna un valor constante *a* de tipo `V` o un valor constante *b* de tipo `V` dependiendo de si la llave *k* pertenece o no a cierto conjunto que actúa como filtro.
`GGeneralTransformer<K,V>`	Clase que representa un transformador especializado de valores que implementa `org.apache.commons.collections15.Transformer<K,V>`, donde para cada llave de tipo `K` se retorna un valor de tipo `V` a través de una asociación-llave valor representada con una Tabla de *Hashing*.
`GGeometry`	Clase que contiene una colección de rutinas útiles que desarrollan operaciones de geométricas y del álgebra lineal. Implementa el patrón *Utility*.
`GImage`	Clase que representa una imagen bidimensional cuyo contenido se encuentra en un *buffer* en memoria principal.
`GLayoutType`	Enumeración que representa los distintos tipos de algoritmo provistos por *JUNG* para ubicar los nodos de un grafo (*layout algorithm*) en una superficie de dibujo.
`GShapes`	Clase que contiene una colección de rutinas útiles para crear figuras geométricas. Implementa el patrón *Utility*.

A.6.2.20. Paquete *gold.util*

Diagrama de clases del paquete `gold.util`.



Clases que conforman el paquete `gold.util`.

Clase	Descripción
`IMethod`	Interfaz que representa un método de una clase.
`GArrays`	Clase que contiene una colección de rutinas útiles para manipular arreglos. Implementa el patrón *Utility*.
`GAutomata`	Clase que contiene una colección de rutinas útiles para desarrollar operaciones sobre autómatas. Implementa el patrón *Utility*.
`GCollections`	Clase que contiene una colección de rutinas útiles para manipular colecciones. Implementa el patrón *Utility*.
`GComparators`	Clase que contiene una colección de rutinas útiles para crear comparadores clásicos. Implementa el patrón *Utility*.
`GGraphs`	Clase que contiene una colección de implementaciones de algoritmos típicos sobre grafos. Implementa el patrón *Utility*.
`GMethod`	Clase que implementa la interfaz `IMethod` usando el mecanismo de *reflexión* (*reflection*) provisto por *Java* [62].
`GReflection`	Clase que contiene una colección de rutinas útiles para examinar en tiempo de ejecución los métodos y atributos de una clase usando el mecanismo de *reflexión* (*reflection*) provisto por *Java* [62]. Además, contiene una colección de rutinas útiles para evaluar expresiones lógico-aritméticas. Implementa el patrón *Utility*.
`GToolkit`	Clase que contiene una colección de rutinas útiles generales. Implementa el patrón *Utility*.

Bibliografía

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. Cambridge, MA, U.S.A. : The MIT Press, 2001. Second edition.

[2] Víctor Hugo Cárdenas Varón. *CSet: un lenguaje para composición de conjuntos*. Bogotá, Colombia : Uniandes, 2009. Tesis de maestría.

[3] Luis Miguel Pérez Díez. *GOLD: un lenguaje orientado a grafos y conjuntos*. Bogotá, Colombia : Uniandes, 2009. Tesis de pregrado.

[4] Diana Mabel Díaz Herrera. *GOLD+: lenguaje de programación para la manipulación de grafos: extensión de un lenguaje descriptivo a un lenguaje de programación*. Bogotá, Colombia : Uniandes, 2010. Tesis de maestría.

[5] Wikipedia, the free encyclopedia. *Extended Backus-Naur Form*. Sin fecha (recuperado el 6 de diciembre de 2011). (online) http://en.wikipedia.org/wiki/Extended_Backus-Naur_Form.

[6] The Eclipse Foundation. *Xtext Language Development Framework*, 2011. Versión 2.2.1: (online) http://www.eclipse.org/Xtext/, http://xtext.itemis.com/.

[7] The Eclipse Foundation. *Eclipse IDE*, 2011. Versión Indigo 3.7.1: (online) http://www.eclipse.org/.

[8] The Eclipse Foundation. *Eclipse Modeling Project*, 2011. (online) http://www.eclipse.org/modeling/.

[9] Sun Microsystems. *JavaCC: Java Compiler Compiler, The Java Parser Generator*, 2009. Versión 5.0: (online) http://javacc.java.net/.

[10] Rensselaer Polytechnic Institute. *XGMML (eXtensible Graph Markup and Modeling Language)*, 2001. (online) http://www.cs.rpi.edu/research/groups/pb/punin/public_html/XGMML/.

[11] Northwestern University. *GIDEN: a graphical environment for network optimization*, 1993-2004. Versión 4.0 alpha: (online) http://users.iems.northwestern.edu/~giden/.

[12] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. New York, NY, U.S.A. : Springer-Verlag, 1993.

[13] Groovy Community. *Groovy: A dynamic language for the Java platform*, 2011. Versión 1.8: (online) http://groovy.codehaus.org/.

[14] AT&T Labs Research. *Graphviz - Graph Visualization Software*, 2011. Versión 2.28: (online) http://graphviz.org/.

[15] University of Passau. *GML (Graph Modelling Language)*, 2010. (online) http://www.fim.uni-passau.de/en/fim/faculty/chairs/theoretische-informatik/projects.html.

[16] University of Passau. *GTL: Graph Template Library*, 2010. (online) http://www.fim.uni-passau.de/en/fim/faculty/chairs/theoretische-informatik/projects.html.

[17] The GraphML Team. *The GraphML File Format*, 2001-2007. (online) http://graphml.graphdrawing.org/.

[18] Ric Holt, Andy Schürr, Susan Elliott Sim, and Andreas Winter. *GXL: Graph eXchange Language*, July 2002. Versión 1.1: (online) http://www.gupro.de/GXL/.

[19] Microsoft Corporation. *Directed Graph Markup Language (DGML)*, 2009. (online) http://schemas.microsoft.com/vs/2009/dgml/.

[20] Alejandro Rodríguez Villalobos. *Grafos: software para la construcción, edición y análisis de grafos*. Universitat Politècnica de Catalunya, July 2011. Versión v. 1.3.3: (online) http://arodrigu.webs.upv.es/grafos/doku.php.

[21] The JUNG Framework Development Team. *JUNG: Java Universal Network/Graph Framework*, January 2010. Versión 2.0.1: (online) http://jung.sourceforge.net/.

[22] Barak Naveh and contributors. *JGraphT: a free Java graph library that provides mathematical graph-theory objects and algorithms*, 2003-2005. Versión 0.8.2: (online) http://jgrapht.sourceforge.net/.

[23] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *The com.mhhe.clrs2e package: Java classes to implement the algorithms in Parts I through VI of Introduction to Algorithms (second edition)*, 2003. (online) http://people.uncw.edu/adharg/courses/csc532/BookCD/.

[24] University of Passau. *Gravisto: Graph Visualization Toolkit*, 2010. (online) http://gravisto.fim.uni-passau.de/.

[25] Martin Erwig. *FGL - A Functional Graph Library*. Oregon State University, September 2002. (online) http://web.engr.oregonstate.edu/~erwig/fgl/.

[26] Christian Doczkal. *A Functional Graph Library, Based on Inductive Graphs and Functional Graph Algorithms*, March 2006. (online) http://www.st.cs.uni-saarland.de/edu/seminare/2005/advanced-fp/slides/doczkal.pdf.

[27] JGraph Ltd. *JGraph*, 2011. (online) http://www.jgraph.com/.

[28] Sandra Steinert, Greg Manning, and Detlef Plump. *GP (Graph Programs)*. University of York, 2009. (online) http://www.cs.york.ac.uk/plasma/wiki/index.php?title=GP_(Graph_Programs).

[29] Detlef Plump. *The Graph Programming Language GP*. University of York, 2009. (online) http://cai09.web.auth.gr/Lectures/talk.pdf.

[30] Marko A. Rodríguez. *Gremlin: A Graph-Based Programming Language*. Los Alamos National Laboratory, July 2011. (online) https://github.com/tinkerpop/gremlin/wiki, http://www.slideshare.net/slidarko/gremlin-a-graphbased-programming-language-3876581.

[31] W3C. *XML Path Language (XPath)*, November 1999. Versión 1.0: (online) http://www.w3.org/TR/xpath/.

[32] Derek Stainer. *Gremlin: A Graph-Based Programming Language*, August 2010. (online) http://www.nosqldatabases.com/main/2010/8/23/gremlin-a-graph-based-programming-language.html.

[33] Oracle Corporation. *Scripting for the Java Platform*, July 2006. (online) http://java.sun.com/developer/technicalArticles/J2SE/Desktop/scripting/.

[34] Werner C. Rheinboldt, Victor R. Basili, and Charles K. Mesztenyi. *GRAAL: On a programming language for graph algorithms*. *BIT*, 12:220--241, 1972. http://www.cs.umd.edu/~basili/publications/journals/J01.pdf.

[35] G. R. Garside and P. E. Pintelas. *GRAAP: An ALGOL 68 package for implementing graph algorithms*. *The Computer Journal*, 23(3):237--242, 1979. http://comjnl.oxfordjournals.org/content/23/3/237.full.pdf.

[36] Adam Drozdek. *Data Structures and Algorithms in Java*. New York, NY, U.S.A. : Cengage Learning, 2008. Third edition.

[37] Robert Lafore. *Data Structures & Algorithms in JAVA*. Indianapolis, IN, U.S.A. : Sams, 2003. Second edition.

[38] Jorge A. Villalobos S. *Diseño y manejo de estructuras de datos en C*. Bogotá, Colombia : McGraw-Hill, 1996.

[39] Rafel Cases Muñoz and Lluís Màrquez Villodre. *Lenguajes, gramáticas y autómatas: Curso básico*. México, México : Alfaomega, 2002.

[40] David A. Watt. *Programming Language Design Concepts*. Chichester, West Sussex, England : John Wiley & Sons, 2004.

[41] Terrence W. Pratt and Marvin V. Zelkowitz. *Programming languages : design and implementation*. Upper Saddle River, NJ, U.S.A. : Prentice-Hall, 1996. Third edition.

[42] Michael L. Scott. *Programming Language Pragmatics*. San Francisco, CA, U.S.A. : Morgan Kaufmann, 2000. First edition.

[43] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. Cambridge, MA, U.S.A. : The MIT Press, 1996. Second edition.

[44] Wikipedia, the free encyclopedia. *Backus-Naur Form*. Sin fecha (recuperado el 6 de diciembre de 2011). (online) http://en.wikipedia.org/wiki/Backus-Naur_Form.

[45] ISO/IEC. *ISO/IEC 14977: Syntactic metalanguage - Extended BNF*, 1996. (online) http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf.

[46] W3C. *Extensible Markup Language (XML) 1.0 - EBNF Notation*, November 2008. (online) http://www.w3.org/TR/REC-xml/#sec-notation.

[47] Wikipedia, the free encyclopedia. *Abstract machine*. Sin fecha (recuperado el 6 de diciembre de 2011). (online) http://en.wikipedia.org/wiki/Abstract_machine.

[48] Wikipedia, the free encyclopedia. *Disjoint-set data structure*. Sin fecha (recuperado el 6 de diciembre de 2011). (online) http://en.wikipedia.org/wiki/Disjoint-set_data_structure.

[49] Martin Fowler and Rebecca Parsons. *Domain-Specific Languages*. Upper Saddle River, NJ, U.S.A. : Addison-Wesley, 2011.

[50] Diomidis Spinellis. *Notable design patterns for domain-specific languages*. *New York, NY, U.S.A. : Journal of Systems and Software*, 56(1):91--99, February 2001. doi>10.1016/S0164-1212(00)00089-3, http://portal.acm.org/citation.cfm?id=371313.

[51] Marjan Mernik, Jan Heering, and Anthony M. Sloane. *When and how to develop domain-specific languages*. *New York, NY, U.S.A. : ACM Computing Surveys (CSUR)*, 37(4):316--344, December 2005. doi>10.1145/1118890.1118892, http://dl.acm.org/citation.cfm?doid=1118890.1118892.

[52] ISO/IEC. *ISO/IEC 9126: The Standard of Reference: Information technology - Software Product Evaluation - Quality characteristics and guidelines for their use*, 1991. (online) http://www.cse.dcu.ie/essiscope/sm2/9126ref.html.

[53] Mikko Tommila and colaborators. *Apfloat: a high performance arbitrary precision arithmetic library*, March 2011. Versión 1.6.2: (online) http://www.apfloat.org/.

[54] Wikipedia, the free encyclopedia. *Force-based algorithms (graph drawing)*. Sin fecha (recuperado el 6 de diciembre de 2011). (online) http://en.wikipedia.org/wiki/Force-based_algorithms_(graph_drawing).

[55] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. Cambridge, MA, U.S.A. : The MIT Press, 2009. Third edition.

[56] JUnit.org community. *JUnit.org Resources for Test Driven Development*, September 2011. Versión 4.10: (online) http://www.junit.org/.

[57] Terence Parr and colaborators. *ANTLR: Another Tool for Language Recognition*, July 2011. Versión v3.4: (online) http://www.antlr.org/.

[58] Tomihisa Kamada and Satoru Kawai. *An algorithm for drawing general undirected graphs*. *Information Processing Letters*, 31(1):7--15, April 1989. doi>10.1016/0020-0190(89)90102-6, http://www.sciencedirect.com/science/article/pii/0020019089901026.

[59] Thomas M. J. Fruchterman and Edward M. Reingold. *Graph drawing by force-directed placement*. *New York, NY, U.S.A. : Software-Practice & Experience (John Wiley & Sons, Inc)*, 21(11):1129--1164, November 1991. doi>10.1002/spe.4380211102, http://portal.acm.org/citation.cfm?id=137557.

[60] Oracle Corporation. *The Java Tutorials - Trail: Learning the Java Language - Lesson: Classes and Objects - Nested Classes*, 2011. (online) http://docs.oracle.com/javase/tutorial/java/javaOO/nested.html.

[61] Oracle Corporation. *Java Programming Language - Autoboxing*, 2010. (online) http://docs.oracle.com/javase/1.5.0/docs/guide/language/autoboxing.html.

[62] Oracle Corporation. *The Java Tutorials - Trail: The Reflection API*, 2011. (online) http://docs.oracle.com/javase/tutorial/reflect/.

[63] Wikipedia, the free encyclopedia. *List comprehension*. Sin fecha (recuperado el 6 de diciembre de 2011). (online) http://en.wikipedia.org/wiki/List_comprehension.

[64] Wikipedia, the free encyclopedia. *Set-builder notation*. Sin fecha (recuperado el 6 de diciembre de 2011). (online) http://en.wikipedia.org/wiki/Set-builder_notation.

[65] David Gries and Fred B. Schneider. *Calculational Logic: An introduction to teaching logic as a tool*. Sin fecha (recuperado el 6 de diciembre de 2011). (online) http://www.cs.cornell.edu/gries/Logic/intro.html.

[66] Wikipedia, the free encyclopedia. *Syntactic sugar*. Sin fecha (recuperado el 6 de diciembre de 2011). (online) http://en.wikipedia.org/wiki/Syntactic_sugar.

[67] Oracle Corporation. *The Java Tutorials - Trail: Learning the Java Language - Lesson: Language Basics - The switch Statement*, 2011. (online) http://docs.oracle.com/javase/tutorial/java/nutsandbolts/switch.html.

[68] Wikipedia, the free encyclopedia. *While*. Sin fecha (recuperado el 6 de diciembre de 2011). (online) http://en.wikipedia.org/wiki/While.

[69] Oracle Corporation. *Java Programming Language - Programming With Assertions*, 2002. (online) http://docs.oracle.com/javase/1.4.2/docs/guide/lang/assert.html.

[70] Wikipedia, the free encyclopedia. *Pseudorandomness*. Sin fecha (recuperado el 6 de diciembre de 2011). (online) http://en.wikipedia.org/wiki/Pseudorandomness.

[71] Wikipedia, the free encyclopedia. *Ad-hoc polymorphism*. Sin fecha (recuperado el 6 de diciembre de 2011). (online) http://en.wikipedia.org/wiki/Ad-hoc_polymorphism.

[72] Anne Kaldewaij. *Programming: The Derivation of Algorithms*. New York, NY, U.S.A. : Prentice-Hall, 1990.

[73] Rodrigo Cardoso Rodríguez. *Verificación y Desarrollo de Programas*. Bogotá, Colombia : Ediciones Uniandes, 1991.

[74] Štěpán Roh and contributors. *DejaVu Fonts*, February 2011. Versión 2.33: (online) http://dejavu-fonts.org/.

[75] Charles Bigelow and Kris Holmes. *Lucida Sans Font*, September 2010. Java Runtime Environment Fonts Directory, archivo java/jre6/lib/fonts/LucidaSansRegular.ttf.

[76] STI Pub Companies. *STIX Fonts*, December 2010. Versión 1.0.0: (online) http://www.stixfonts.org/.

[77] George Williams and contributors. *FontForge: An Outline Font Editor*, February 2011. Versión 20110222: (online) http://fontforge.sourceforge.net/.

[78] George Williams and contributors. *FontForge Scripting Tutorial*, February 2011. Versión 20110222: (online) http://fontforge.sourceforge.net/scripting-tutorial.html, http://fontforge.sourceforge.net/scripting-alpha.html.

[79] Alejandro Sotelo Arévalo. *LOGS2005: Editor de demostraciones en lógica ecuacional*. Bogotá, Colombia : Uniandes, 2006. Tesis de pregrado.

[80] Oracle Corporation. *The Java Tutorials - Trail: Creating a GUI With JFC/Swing*, 2011. (online) http://docs.oracle.com/javase/tutorial/uiswing/.

[81] The Eclipse Foundation. *SWT: The Standard Widget Toolkit*, 2011. (online) http://eclipse.org/swt/.

[82] Mark James. *Famfamfam Silk Icons*. Sin fecha (recuperado el 6 de diciembre de 2011). Versión 1.3: (online) http://www.famfamfam.com/.

[83] Wikimedia Foundation Inc. *Wikimedia Commons*. (online) http://commons.wikimedia.org/.

[84] CollabNet, Inc. *ArgoUML*, April 2011. Versión 0.32.2: (online) http://argouml.tigris.org/.

Agradecimientos

Agradezco a mis padres por el apoyo incondicional que me brindaron para lograr mis objetivos, a mi novia por la paciencia que tuvo cuando más tiempo necesitaba, y a Silvia Takahashi por influenciar positivamente mi gusto hacia la teoría de lenguajes, por su valiosa colaboración a lo largo del proyecto, y por sus oportunos consejos que permitieron salvar una gran cantidad de tiempo.

No hubiera sido posible alcanzar los resultados obtenidos sin la orientación dada por Silvia durante todos estos años a través de todas las fases que experimentó el proyecto, primero con Luis Miguel Pérez, luego con Diana Mabel Díaz y finalmente conmigo.