



UNIVERSIDAD DE BUENOS AIRES  
FACULTAD DE INGENIERÍA  
Año 2014 – 2<sup>do</sup> Cuatrimestre

## **ALGORITMOS Y PROGRAMACIÓN II (75.04)**

TRABAJO PRÁCTICO 1

TEMA: Programación C++

FECHA: Miércoles 10 de diciembre de 2014

ENTREGADOS HASTA LA FECHA:

- ◆ TP0: 23/10/2014
- ◆ TP1: 10/12/2014

INTEGRANTES:

Zaragoza, Juan Manuel - # 92.308  
<juanmanuelzar@gmail.com>

Ferrari Bihurriet, Francisco - # 92.275  
<fferrari@fi.uba.ar>

# 1 Objetivo

El principal objetivo del presente trabajo práctico consiste en implementar una herramienta para procesar imágenes, la cual recibe una imagen y una función de procesamiento no estandarizada para obtener una nueva imagen dependiendo de la tipo de función elegida.

El otro objetivo es ejercitar conceptos básicos de programación C++ e implementar TDAs.

## 2 Descripción y modo de uso de la herramienta

El modo de uso de la herramienta mencionada:

```
$ ./tp1 [-i /dirección/a/archivo/entrada/*.png] [-r región] [-o /dirección/a/archivo/salida/*.png] [-f funciones]
```

donde:

- /dirección/a/archivo/entrada/\*.png: nombre y dirección de la imagen PNG que quiere ser procesada. En caso de no especificarse una, la herramienta adoptará la entrada estándar.
- /dirección/a/archivo/salida/\*.png: nombre y dirección de la imagen de salida procesada por la herramienta. En caso de no especificarse una, la herramienta adoptará la salida estándar.
- funciones: las funciones soportadas son expresiones algebraicas que opere números complejos. Por ej.:  $2 \cdot \exp(z) + \sin(z^2) + 5$ .
- región: región del plano complejo donde se quiere trabajar, expresada como  $\langle \text{ancho}, \text{alto}, \text{x\_centro}, \text{y\_centro} \rangle$ . Por ej.:  $\langle 2, 2, 0, 0 \rangle$ , que es la opción por defecto.

## 3 Diseño e Implementación

Se identificaron diferentes maneras de resolver el problema planteado, donde la más conveniente fue creando las clases para las entidades más importantes que involucran al trabajo práctico número 1. Obviamente, algunas clases que se utilizan en este trabajo no serán descriptas porque ya fueron hechas en el trabajo práctico número 0.

### 3.1 Problemas identificados

El primer problema encontrado fue, cómo a partir de un string pasado como parámetro de entrada podíamos convertirlo en una función evaluable. Para resolver este problema utilizamos la notación polaca inversa o RPN e implementamos el *algoritmo shunting yard* el cual se encarga de convertir una expresión infija en otra postfija.

Luego, para implementar el algoritmo mencionado, necesitábamos alguna entidad que se encargara de apilar/acolar los operadores/operandos que recibíamos de la entrada. Para esto se implementaron las clases *stack* y *queue* que representan una pila y una cola, respectivamente. Estas clases, se ayudaron de la clase *node*, la cual representa un nodo que posee un valor y un puntero a otro nodo.

Debido a la dificultad de manipular los operadores/operandos/funciones como si fuesen cadenas de texto, se implementó la clase *token* que representa cualquier elemento que puede intervenir en la expresión algebraica.

Además, en complemento a estas entidades, se creó la clase `optree` que representa un árbol en memoria de la expresión RPN que ayuda a calcular la expresión anteriormente mencionada con los valores del píxel a operar entre sus atributos.

## 3.2 Resolución

El programa principal (main) se encarga de delegar la validación de argumentos en línea de comandos, la lectura del archivo de entrada (o en su defecto, la entrada estándar), la creación de la imagen de entrada en memoria y carga con los valores del archivo, la creación de la imagen de salida en memoria, y en cada posición de ésta, la asignación del píxel proveniente de aplicar una función compleja a la posición del píxel en la imagen de entrada, y por último, la grabación de la imagen de salida (o en su defecto, la salida estándar) con el resultado de la transformación mencionada.

Para aplicar la función proveniente como argumento en línea de comando, se utilizó un método que la convierte en una cola de tokens que es convertida a la notación RPN para que luego se pueda evaluar cada píxel de la imagen.

### 3.2.1 Soluciones del trabajo práctico número 0

Las soluciones implementadas en el trabajo práctico 0 sirvieron para el presente trabajo, pero no serán descriptas detalladamente porque el objeto del presente radica en las clases que mencionaremos en los siguientes puntos.

El principal problema del trabajo anterior, fue cómo transformar una imagen en un conjunto de puntos ubicados en el plano complejo que fue solucionado con algunas transformaciones matemáticas ya descriptas.

La clase `complex` es nuevamente utilizada para representar un número complejo y para operar con determinadas funciones sobre ellos. Se agregaron los métodos:

#### Operaciones binarias:

- `const complex operator_add(const complex &, const complex &);` suma dos complejos.
- `const complex operator_subt(const complex &, const complex &);` resta dos complejos.
- `const complex operator_mult(const complex &, const complex &);` multiplica dos complejos.
- `const complex operator_div(const complex &, const complex &);` divide dos complejos.
- `const complex operator_pow(const complex &, const complex &);` calcula la potencia entre dos complejos ( $w^z$ ).

#### Operaciones unarias:

- `const complex exp(const complex &);` función exponencial de un número complejo.
- `const complex log(const complex &);` logaritmo natural de un número complejo.
- `const complex sin(const complex &);` función seno de un número complejo.
- `const complex cos(const complex &);` función coseno de un número complejo.
- `const complex real_as_complex(const complex &);` obtiene la parte real de un número complejo, en la parte real de otro número complejo de salida.

- `const complex imag_as_complex(const complex &);` obtiene la parte imaginaria de un número complejo, en la parte real de otro número complejo de salida.
- `const complex abs_as_complex(const complex &);` obtiene el módulo de un número complejo, en la parte real de otro número complejo de salida.
- `const complex phase_as_complex(const complex &);` obtiene el argumento (fase) de un número complejo, en la parte real de otro número complejo de salida.

Por último, la clase **PGMimage** representa una imagen extraída de un archivo PGM en memoria. Posee los siguientes atributos: **\_magic\_number**, **\_width** (ancho de la imagen), **\_height** (alto de la imagen), **\_color\_depth** (profundidad de color) y **\_canvas** (lienzo en memoria).

### 3.2.2 Clase Node

La clase **node** representa un nodo. Un nodo posee un valor y un puntero a un nodo. Por lo tanto, es la estructura básica para implementar una pila o una cola.

#### 3.2.2.1 Constructores

La clase posee solo un constructor, el cual asigna un valor a su atributo **\_value** y un puntero al siguiente nodo en caso de que se especifique un nodo, sino un puntero a **NULL**.

### 3.2.3 Clase Stack

Esta clase representa una pila. Una pila es una estructura de datos en la que el modo de acceso a sus elementos es de tipo LIFO (del inglés Last In First Out, último en entrar, primero en salir) que permite almacenar y recuperar datos.

El único atributo que posee esta clase es un puntero a un nodo denominado **\_last**, que representa el último elemento ingresado en el contenedor.

#### 3.2.3.1 Constructores

La clase posee un solo constructor que inicializa a **NULL** su puntero **\_last**.

#### 3.2.3.2 destructores

La clase implementa un destructor que desapila todos los elementos almacenados en el contenedor.

#### 3.2.3.3 Métodos característicos

Existen algunos métodos característicos de la clase Stack:

- `bool isEmpty() const;` informa si la pila se encuentra vacía.
- `void push(const T& );` agrega (apila) un dato en la pila.
- `T pop();` devuelve el último dato ingresado y destruye el nodo.
- `T& topElement();` devuelve el elemento superior de la pila para poder operarlo o consultarlo.

### 3.2.4 Clase Queue

Esta clase representa una cola. Una pila es una estructura de datos en la que el modo de acceso a sus elementos es de tipo FIFO (del inglés First In First Out, primero en entrar, primero en salir) que permite almacenar y recuperar datos.

Posee dos atributos: un puntero a un nodo denominado **\_last** y otro denominado **\_first**.

#### 3.2.4.1 Constructores

La clase posee un solo constructor que inicializa a **NULL** sus punteros **\_last** y **\_first**.

#### 3.2.4.2 Destrucciones

La clase implementa un destructor que desencola todos los elementos almacenados en el contenedor hasta que no **\_first** apunte a **NULL**.

#### 3.2.4.3 Métodos característicos

Existen algunos métodos característicos de la clase Cola:

- `bool isEmpty() const`; informa si la cola se encuentra vacía.
- `void enqueue(const T& );` agrega (acola) un dato en la cola.
- `T dequeue()`; devuelve el primer dato ingresado y destruye el nodo.
- `T& frontElement()`; devuelve el primero elemento de la cola para ser modificado o consultado.
- `T& lastAdded()`; devuelve el último elemento de la cola para ser modificado o consultado.

### 3.2.5 Biblioteca Parser

Estos archivos (`parser.cpp` y `parser.h`) representan un parseador que contiene una clase **token** que representa un token el cual puede ser un **string** o un **double** para almacenar cada uno de los elementos de la expresión *infija* de entrada algunos de los cuales formarán la expresión *RPN*. Además, estos archivos contienen utilidades auxiliares que parsean una entrada de string en tokens para luego convertirla a una expresión RPN, y los arreglos de strings asociadas a ciertos tokens, con sus correspondientes arreglos de punteros a función o valores asociados.

#### 3.2.5.1 Clase Token: Constructores

La clase posee tres constructores: un constructor que recibe un **double** y setea un **TOKEN\_IS\_VALUE** en su atributo **string**; otro constructor que recibe un **string** y setea un **NaN** en su atributo **double**; y por último, un constructor por copia.

#### 3.2.5.2 Clase Token: Métodos getters

Existen algunos métodos para “traducir” el token:

- `string getAsString() const`; obtiene el token como string (devuelve número en string si es del tipo 'valor')
- `double getAsDouble() const`; obtiene el token como un double (devuelve NaN, si no es del tipo 'valor')

### 3.2.5.3 Clase Token: Métodos auxiliares para identificar los tokens

Métodos booleanos que permiten definir el token:

- `bool isValue() const`; ¿Es del tipo 'valor'?
- `bool isParenthesis() const`; ¿Es un paréntesis?
- `bool isOpenParenthesis() const`; ¿Es un paréntesis abierto?
- `bool isClosedParenthesis() const`; ¿Es un paréntesis cerrado?
- `bool isOperator() const`; ¿Es un operador?
- `bool isSpecial() const`; ¿Es un token especial? Los token especiales son "j", "e", "pi" y "z".
- `bool isFunction() const`; ¿Es una función? Las funciones pueden ser "exp", "ln", "sin", "cos", "re", "im", "abs" y "phase".

### 3.2.5.4 Clase Token: Método para utilitario del conversor a RPN

El método `int precedence() const`; es utilizado para informar la precedencia de un token operador o función.

### 3.2.5.5 Funciones de la biblioteca para la conversión de la expresión de entrada en notación RPN

Como se ya se ha mencionado, estas funciones se encuentran dentro del archivo `parser.cpp` y se encargan de la conversión a RPN:

- `int find_in_list(const string [], const string &)`; función para buscar en una lista con centinela de cadena vacía, si no encuentra la string buscada, devuelve *NOT\_FOUND*, que vale -1, si en cambio es encontrada, devuelve su posición en el arreglo de strings. Esta función es utilizada en conjunto con las macros de función `is_operator()`, `is_special()`, `is_function()`; y dentro de la clase `optree_node`, para obtener los punteros asociados a las funciones (operadores) o los complejos (operandos especiales).
- `void parse_expression_in_tokens(const string &, queue<token> &)`; función que parsea el string de entrada en una **cola** de tokens.
- `void convert_to_RPN(stack<token> &, queue<token> &)`; función que recibe una cola de tokens (proveniente del método anterior) y devuelve una **pila** que representa la notación polaca inversa (al desapilarla se obtiene una lectura de derecha a izquierda de la misma).

## 3.2.6 Clase Optree Node

Esta clase representa un nodo en el árbol de operaciones.

### 3.2.6.1 Constructores

La clase posee dos constructores: uno por defecto que inicializa todos sus parámetros a **NULL**: el dato del nodo, las operaciones sobre los mismos, el hijo izquierdo, el hijo derecho y el padre; y otro que inicializa una hoja con un token, y acepta un puntero al padre como argumento opcional.

### 3.2.6.2 Destructor

El destructor se encarga de eliminar el nodo (su memoria dinámica, si es que utiliza) y sus hijos, tanto izquierdo como derecho (si es que tiene).

### 3.2.6.3 Métodos disponibles

- `const complex operate(const complex *z);` opera recursivamente el subárbol que cuelga del nodo en cuestión, recibiendo como parámetro un puntero al complejo **z** de la operación. Devuelve el complejo resultante.
- `bool simplify();` **privado**, simplifica el subárbol de operaciones que cuelga del nodo, realizando las operaciones del mismo que no dependen del complejo **z** de la operación, cuyo resultado no cambiará durante el recorrido de la imagen, esto busca evitar la repetición de cálculos innecesarios. Con una estrategia recursiva se buscan los nodos que no dependen de **z**, el booleano que devuelve el método es verdadero si el subárbol depende de **z** (este valor tiene sentido utilizado en las invocaciones recursivas, en la invocación externa puede ser descartado). Una vez encontrado un nodo que no depende de **z** y a su vez tiene hijos, el subárbol es reemplazado por un único nodo con el resultado de la operación (que no es **z** dependiente). Ambos procesos, búsqueda y simplificación se realizan en la misma recursión.

## 3.2.7 Clase Optree

Esta clase representa el árbol de operaciones, formado por nodos del tipo `optree_node`, que contiene información (punteros) de cuál es la raíz del árbol (`_root`) y cuál es el complejo que cambiará al iterar los píxeles, durante el recorrido de la imagen (`_asoc_pixel`).

### 3.2.7.1 Constructores

La clase posee un constructor por defecto que genera un árbol vacío (ambos atributos en **NULL**). Por otro lado está el constructor `optree(stack<token> &, complex &);` que carga el árbol a partir de una pila de tokens en formato *RPN* y una referencia al complejo que se va a utilizar como **z**, es decir el que va a cambiar en cada evaluación de la expresión para que sea apuntado por `_asoc_pixel`. Este constructor realiza una simplificación del árbol mediante el método interno `simplify()` de la clase `optree_node` (privado, la clase nodo usa el operador **friend** para la clase `optree`), descartando su valor booleano de devolución.

### 3.2.7.2 Destructor

Simplemente invoca al destructor de la raíz, que se encargará recursivamente del árbol completo.

### 3.2.7.3 Operar

Invoca al método `operar` de la raíz, previas validaciones, pasándole como argumento el puntero a **z** seteado en el constructor.

## 3.2.8 Programa principal

El programa principal realiza los siguientes pasos:

1. Por línea de comando se validan los argumentos, se setean los flujos/archivos de donde se leerán y/o escribirán las imágenes PGM y se arma la expresión RPN en una pila de tokens.

2. Se abren los archivos en caso de no haberse elegido las entradas/salidas estándar.
3. Se crea una nueva imagen en memoria, se lee desde el flujo/archivo la imagen y se sube a memoria.
4. Se crea una nueva imagen en memoria pero de salida con los tamaños y profundidad de la imagen de entrada o los seleccionados por el usuario por línea de comandos.
5. Se crean dos complejos con constructores por defecto para los planos de entrada y salida.
6. Se genera un árbol de operaciones a través de la pila RPN mencionada en el punto 1), configurando el complejo asociado al píxel en el plano de salida (el z de la expresión de entrada).
7. Se comienza a recorrer la imagen de salida, a los índices se les aplica la función que los convierte en un complejo (plano de salida), para luego aplicar la expresión ingresada por CLA a través del árbol de operaciones.
8. Con el complejo resultante (plano de entrada), se obtiene la posición en el lienzo de entrada y se copia su valor en la nueva imagen (lienzo de salida). Se reinicia el ciclo, continuando el recorrido descrito a partir del punto 7).
9. Por último, el lienzo de salida que se encuentra en memoria se escribe en el flujo/archivo de salida.

## 4 Proceso de compilación

Para compilar los archivos en entornos *Unix-like*, fue diseñado el *Makefile*, con las siguientes opciones:

➔ Para generar todos los archivos objeto, más el ejecutable en el directorio `./bin/`

```
$ make
```

Este comando crea el directorio `./bin/` compilando el código fuente para obtener los siguientes archivos en código objeto:

- `PGMImage.o`: Clase para manejo de imágenes PGM.
- `complex.o`: Clase para manejo de número complejos.
- `utils.o`: Utilidades del programa, como las funciones internas y los parsers de CLA.
- `cmdline.o`: Manejo de CLA, provisto por los docentes, levemente modificado.
- `parser.o`: Biblioteca de parseo de la expresión de entrada.
- `optree.o`: Clase para la creación, simplificación y evaluación del árbol de operaciones.
- `main.o`: Programa principal, propiamente dicho.

Luego estos archivos son “*linkeados*” en el ejecutable `./bin/tp1`

➔ Para crear el ejecutable con la información y las etiquetas de debugging

```
$ make debug
```

➔ Para “limpiar” (eliminar todos los binarios, y el directorio `./bin/`)

```
$ make clean
```



→ Para eliminar sólo los archivos objeto

```
$ make objclean
```

→ Para compilar eliminando los archivos temporarios

```
$ make deltemps
```

→ Para instalar (\*)

```
# make install
```

→ Desinstalar (\*)

```
# make uninstall
```

(\*): Se requieren permisos de superusuario

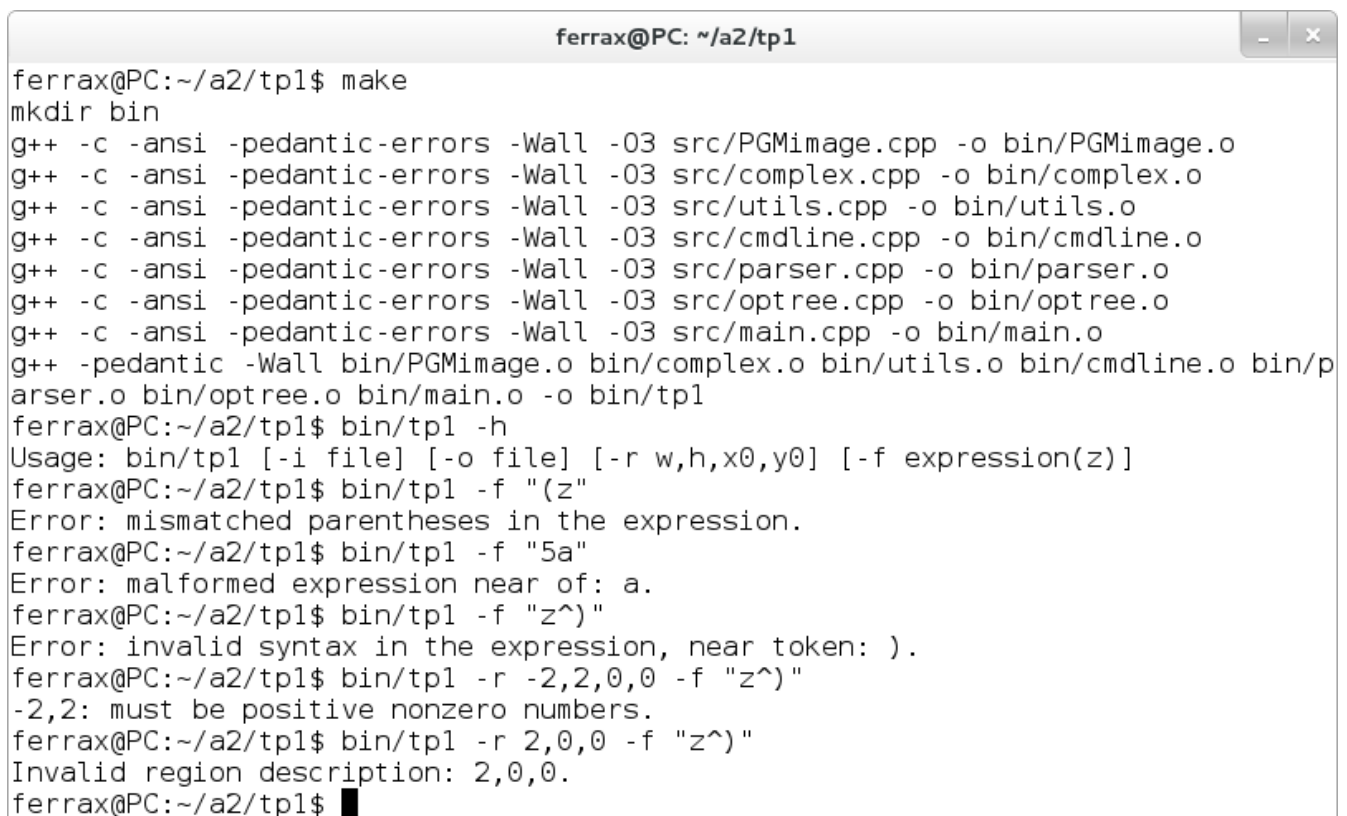
NOTA: también fue probado en *Windows* con *MinGW*, donde las utilidades extra no funcionan, pero sí el comando make a secas.

## 5 Ejecución y resultados

En esta sección se describirán los diferentes modos de uso y resultados de las sucesivas pruebas realizadas.

### 5.1 Pruebas iniciales

A continuación se exhiben los resultados al realizar las siguientes acciones: compilación, ejecuciones con mensaje de ayuda, expresiones mal formadas y regiones inválidas.



```
ferrax@PC: ~/a2/tp1
ferrax@PC:~/a2/tp1$ make
mkdir bin
g++ -c -ansi -pedantic-errors -Wall -O3 src/PGMImage.cpp -o bin/PGMImage.o
g++ -c -ansi -pedantic-errors -Wall -O3 src/complex.cpp -o bin/complex.o
g++ -c -ansi -pedantic-errors -Wall -O3 src/utils.cpp -o bin/utils.o
g++ -c -ansi -pedantic-errors -Wall -O3 src/cmdline.cpp -o bin/cmdline.o
g++ -c -ansi -pedantic-errors -Wall -O3 src/parser.cpp -o bin/parser.o
g++ -c -ansi -pedantic-errors -Wall -O3 src/optree.cpp -o bin/optree.o
g++ -c -ansi -pedantic-errors -Wall -O3 src/main.cpp -o bin/main.o
g++ -pedantic -Wall bin/PGMImage.o bin/complex.o bin/utils.o bin/cmdline.o bin/p
arser.o bin/optree.o bin/main.o -o bin/tp1
ferrax@PC:~/a2/tp1$ bin/tp1 -h
Usage: bin/tp1 [-i file] [-o file] [-r w,h,x0,y0] [-f expression(z)]
ferrax@PC:~/a2/tp1$ bin/tp1 -f "(z"
Error: mismatched parentheses in the expression.
ferrax@PC:~/a2/tp1$ bin/tp1 -f "5a"
Error: malformed expression near of: a.
ferrax@PC:~/a2/tp1$ bin/tp1 -f "z^)"
Error: invalid syntax in the expression, near token: ).
ferrax@PC:~/a2/tp1$ bin/tp1 -r -2,2,0,0 -f "z^)"
-2,2: must be positive nonzero numbers.
ferrax@PC:~/a2/tp1$ bin/tp1 -r 2,0,0 -f "z^)"
Invalid region description: 2,0,0.
ferrax@PC:~/a2/tp1$
```

## 5.2 Pruebas con imágenes PGM

Se mostrarán los diferentes resultados obtenidos al correr diferentes pruebas. Notar que no es necesario hacer una captura de la consola luego de ejecutarse el comando, por lo tanto, solo mostraremos la imagen origen y destino junto al comando ejecutado.

Primero mostraremos la imagen con la función identidad para entender cual es la imagen original y luego mostraremos la imagen transformada.

NOTA: imágenes posteriormente convertidas y escaladas para el informe.

### 5.2.1 Imágenes del enunciado

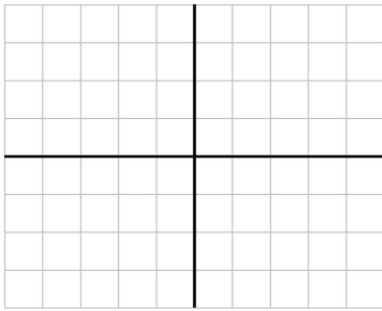


Imagen 1: grid-id.pgm

```
./tp1 -i grid.pgm -o  
grid-id.pgm -f z
```

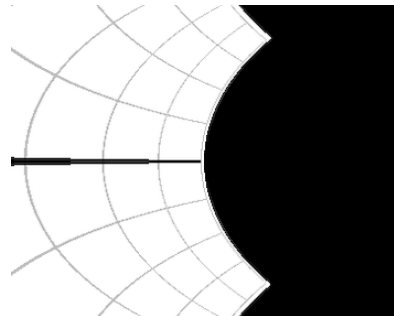


Imagen 2: grid-exp.pgm

```
./tp1 -i grid.pgm  
-o grid-exp.pgm -f exp(z)
```

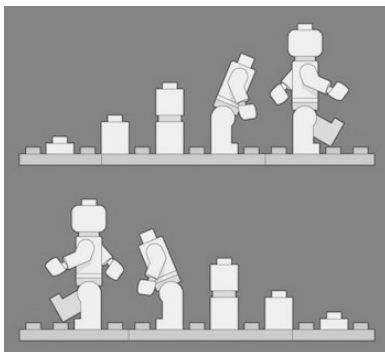


Imagen 3: evolution-id.pgm

```
./tp1 -i evolution.pgm  
-o evolution-id.pgm -f z
```

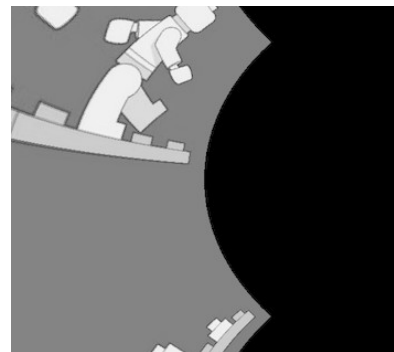


Imagen 4: evolution-exp.pgm

```
./tp1 -i evolution.pgm  
-o evolution-exp.pgm -f exp(z)
```



Imagen 5: evolution-sqr.pgm

```
./tp1 -i evolution.pgm  
-o evolution-sqr.pgm -f z^2
```

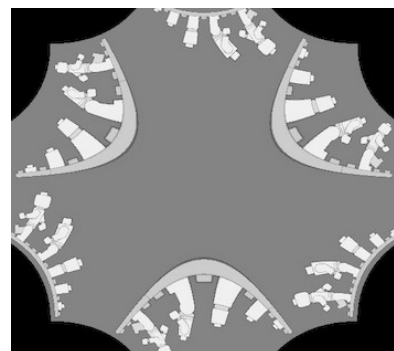


Imagen 6: evolution-cube.pgm

```
./tp1 -i evolution.pgm  
-o evolution-cube.pgm -f z^3
```

## 5.2.2 Otras imágenes y pruebas

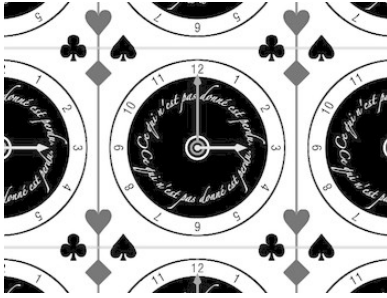


Imagen 7: clocks-1.pgm

```
./tp1 -i clocks.pgm -r 6.4,4.8,0,0  
-o clocks-1.pgm -f z
```

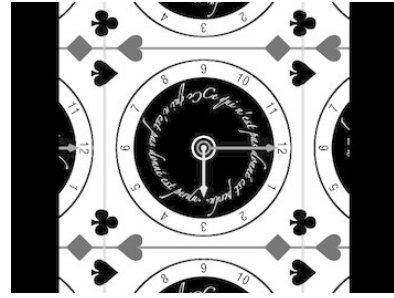


Imagen 8: clocks-2.pgm

```
./tp1 -i clocks.pgm -r 6.4,4.8,0,0  
-o clocks-2.pgm -f j*z
```

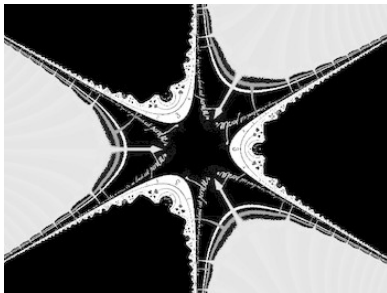


Imagen 9: clocks-3.pgm

```
./tp1 -i clocks.pgm -r 6.4,4.8,0,0  
-o clocks-3.pgm -f exp(z^3)
```

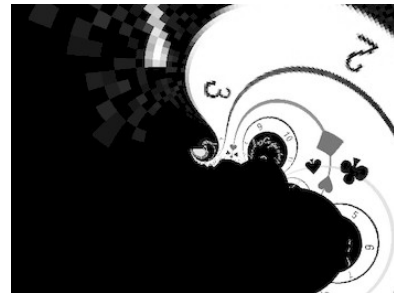


Imagen 10: clocks-4.pgm

```
./tp1 -i clocks.pgm -r 6.4,4.8,0,0  
-o clocks-4.pgm -f exp(z^j)
```

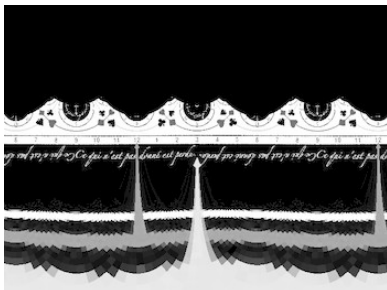


Imagen 11: clocks-5.pgm

```
./tp1 -i clocks.pgm -r 6.4,4.8,0,0  
-o clocks-5.pgm -f j^(-z)
```

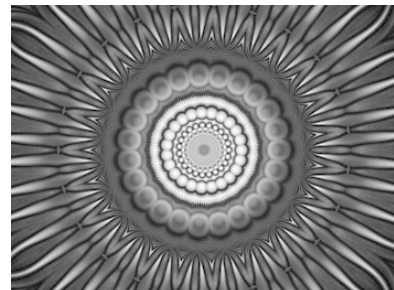


Imagen 12: pattern-1.pgm

```
./tp1 -i pattern.pgm -r 5,3.75,0,0  
-o pattern-1.pgm -f z
```

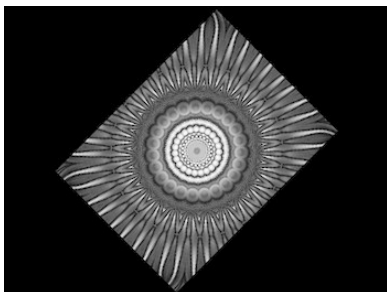


Imagen 13: pattern-2.pgm

```
./tp1 -i pattern.pgm -r 5,3.75,0,0  
-o pattern-2.pgm -f (z*1.7*e^(-j*pi/4))
```

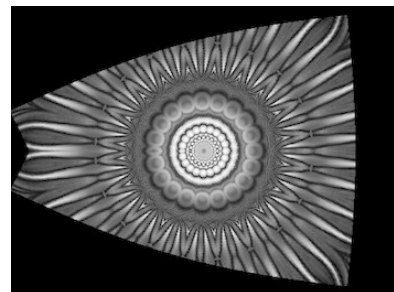


Imagen 14: pattern-3.pgm

```
./tp1 -i pattern.pgm -r 5,3.75,0,0  
-o pattern-3.pgm -f (z*1.5)^0.85
```

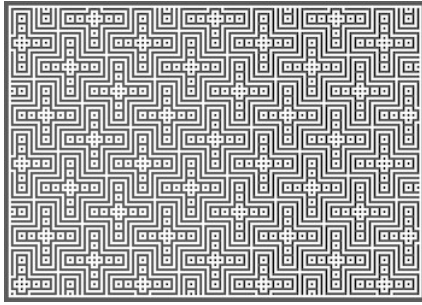


Imagen 15: 9097x6500-1.pgm

```
./tp1 -i 9097x6500.pgm -r 9.097,6.5,0,0  
-o 9097x6500-1.pgm -f z
```

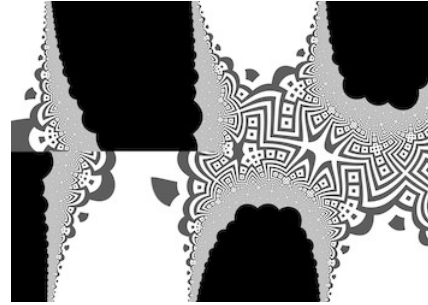


Imagen 16: 9097x6500-2.pgm

```
./tp1 -i 9097x6500.pgm -r 9.097,6.5,0,0  
-o 9097x6500-2.pgm -f z^(j*sin(z))
```

## 6 Conclusión

El trabajo práctico presentado, nos originó diferentes dificultades. Al principio se nos hizo difícil empezar con el parseo de la expresión, y lo hicimos sin tener bien en claro cómo haríamos luego para evaluarla. Después de replantear algunas estructuras y hacer el parseo más robusto, optamos por el camino del árbol de operaciones (la otra alternativa era evaluar utilizando una pila). Al elegir este camino surgieron nuevas problemáticas así como ideas, la primera fue realizar una clase “híbrida” que pudiera ser nodo de éste árbol, teniendo la opción de ser operador u operando, y que sea capaz de auto evaluarse. La segunda vino de la mano de una optimización, realizando las operaciones estáticas que permanecerán constantes al recorrer la imagen primero, simplificando el árbol y evitando cálculos repetitivos sin sentido. Esta optimización sólo es notoria en casos de imágenes muy grandes y/o casos en que la cantidad de operaciones independientes de  $z$  a simplificar es alta.

Es un trabajo interesante para terminar de aprender algunas técnicas y conceptos de C++, tales como el uso intensivo de clases. También resultó pedagógico en lo que respecta a estructuras de datos, como las pilas, colas y árboles, y sus operaciones más comunes.

A esto se suma lo entretenido de la transformación de imágenes mediante funciones holomorfas, que ahora pueden (a diferencia del tp0) ser ingresadas en tiempo de ejecución, haciendo más amena la tarea de resolver las dificultades, permitiendo “jugar” un poco mientras se trabaja.

## 7 Bibliografía

- Netpbm format (Wikipedia).  
[http://en.wikipedia.org/wiki/Netpbm\\_format](http://en.wikipedia.org/wiki/Netpbm_format)
- Holomorphic function (Wikipedia).  
[http://en.wikipedia.org/wiki/Holomorphic\\_function](http://en.wikipedia.org/wiki/Holomorphic_function)
- Conformal pictures (Wikipedia).  
[http://en.wikipedia.org/wiki/Conformal\\_pictures](http://en.wikipedia.org/wiki/Conformal_pictures)
- Standard C++ Library reference (cplusplus.com).  
<http://www.cplusplus.com/reference/>