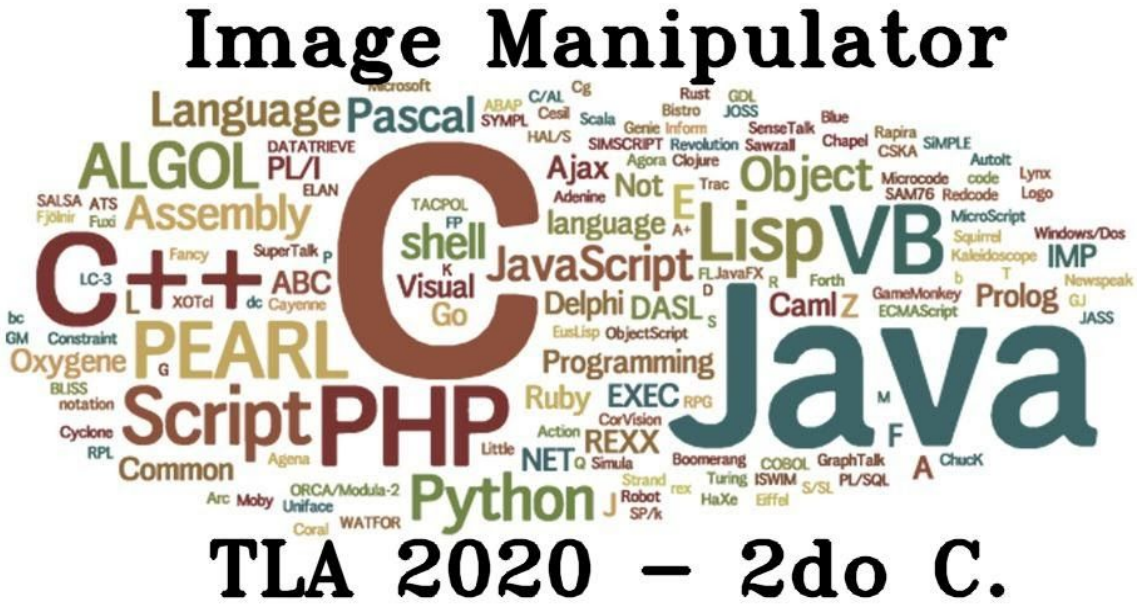


Trabajo Práctico Autómatas, Teoría de Lenguajes y Compiladores

Baiges, Matías Sebastián - 59076

Comerci Wolcanyik, Nicolás - 59520

Torrusio, Lucía - 59489

2020 - 2do. Cuatrimestre

Índice

Índice **2**

Idea subyacente y objetivo del lenguaje 3

Consideraciones realizadas 3

Descripción del desarrollo del TP 3

LEX 3

YACC 4

Análisis de funciones 4

Análisis de variables 4

Traducción a Python 6

Ejecutable 6

Códigos de ejemplo 6

Descripción de la gramática 10

Dificultades encontradas 14

Futuras extensiones 15

Benchmark 15

Conclusión 17

Referencias **18**

Idea subyacente y objetivo del lenguaje

Para este trabajo práctico se desarrolló un lenguaje tipado llamado *'Image Manipulator'* y el objetivo del mismo es, como su nombre lo indica, poder manipular imágenes de una manera sencilla con unas pocas líneas de código.

Image Manipulator es una gran base para poder crecer como lenguaje. Ofrece un uso simple para procesamiento de imágenes en el que no se necesita experiencia avanzada para poder utilizarlo. Permite realizar funcionalidades básicas en una línea, lo que en otros lenguajes llevaría muchas más, siendo poco agradable para un usuario inexperto.

Para lograr dicho objetivo, el código generado se traduce a Python y, además, se creó una librería de manejo de imágenes que está disponible de manera nativa en nuestro lenguaje.

Consideraciones realizadas

Se diseñó un lenguaje tipado, que aproveche el manejo vectorial y algunas librerías de Python, manteniendo sintaxis clásicas de programación del estilo del lenguaje C.

Se decidió utilizar un árbol como estructura de almacenamiento de la información procesada por LEX y YACC, para su posterior análisis con posibilidad de recorrer el código más de una vez.

Se decidió utilizar tablas de hashing para mejorar el orden temporal de algunos algoritmos de validaciones.

Se decidió validar tipos de variables y funciones, tanto en declaraciones como asignaciones.

Se mantuvo preferencia por definición y uso de funciones entendibles y claras, en vez de funciones eficientes, ya que al trabajar en grupo en un trabajo de tal magnitud, facilitaba la comprensión de los códigos realizados.

Se decidió implementar las funciones dadas ya que brindan las funcionalidades base del procesamiento de imagen, las necesarias para que un programador inexperto se pueda manejar con facilidad.

Descripción del desarrollo del TP

El desarrollo del lenguaje se realizó en pasos distinguidos.

LEX

En un principio se creó el archivo "*project.l*", en el cual se capturó, a través de expresiones regulares, las diferentes palabras distinguidas como lo son *for* y *while*; también expresiones como dígitos, signos, entre otros. Estas al detectarlas, se agrega información en caso de ser necesario devolviendo un tipo particular de dato (ejemplo en el caso de una constante entera se almacena en el union

declarado de *yyval,* como entero), y luego retornan un token que las caracteriza para ser utilizadas luego por yacc.

YACC

A partir del archivo "*project.l*" se define para ser procesado por yacc el archivo *"project.y"* que se encarga de crear un árbol de forma ascendente (*LALR(1)*). Cada una de las expresiones capturadas en el *"proyecto.l"* son hojas del árbol, a partir de ellos se van formando nodos, como *"primary_exp"* que puede estar compuesta por terminales como `id([a-zA-Z_][a-zA-Z_0-9]*)`, `int_const([0-9]+)`, `float_const ([0-9]+"."[0-9]+([eE][-+]?[0-9]+)?)`, o no terminales como lo es `'('conditional_exp')'`. Para la creación de este árbol sintáctico, se definieron distintos tipos de nodos (operadores - aquellos que no son hojas, sino que su significado se forma gracias a las hojas debajo del mismo-, constantes de tipo entero y float - estos últimos se almacenan como strings permitiendo mayor versatilidad a la hora de definir un float y procesarlo, lo cual evita que se puedan hacer optimizaciones al respecto; sin embargo, el hecho de que sea almacenado en un nodo distintivo permite que estas cuestiones puedan ser rápidamente subsanadas en futuras mejoras - así como también identificadores - símbolos de variables y de funciones -).

Análisis de funciones

Luego, se pasó a reconocer todas las funciones definidas junto con el tipo de retorno los tipos recibidos en sus parámetros. Esta primera pasada reconoce y almacena las funciones en una tabla de hashing, describiendo en caso de error si ya existe otra función declarada bajo el mismo nombre (sea nativa o no). Además, nos aseguramos de que se defina una función `main` (punto de entrada) que retorne int. El hecho de no estar validando que no tenga parámetros es simplemente por el hecho de permitir a futuro que se ejecute la salida del compilador con parámetros de entrada por consola.

El hecho de realizar una pasada sobre el árbol únicamente recolectando información de funciones, nos permite luego llamar a funciones que son declaradas '*más abajo*' en el código, y poder revisar si las llamadas a las distintas funciones son correctas (sus parámetros coinciden en tipos), además de asegurarse que las funciones llamadas durante la ejecución del programa realmente existan. Esto conlleva por supuesto, a perder la posibilidad de importar funciones externas que fueran declaradas por el usuario en otro archivo compilado bajo este compilador, ya que de permitir importar, no podríamos hacer validaciones de tipo sobre funciones que no se encuentran cargadas en el árbol correspondiente al código siendo compilado.

En caso de existir coalición de nombres de funciones, o de estar utilizando como nombre, el nombre de una función nativa, se informa al usuario por salida de error que hubo problemas validando las funciones y el programa termina, sin el archivo de salida por supuesto.

Análisis de variables

Tras haber cargado las funciones y validar que no haya errores, se procede a realizar un control de tipos (tanto en asignaciones, como en operaciones intermedias (suma, resta, módulo) y en llamadas a funciones).

Para realizar las validaciones correspondientes, se siguió el siguiente algoritmo:

Se cuenta de 2 pilas de enteros y 2 pilas de información de variables, además de una tabla de hash para los símbolos.

```
// Global list of all symbols info allocated
static struct var_info * allocated_symbols = NULL;

// Number of current symbols used
static struct sym_am * used_symbols_amounts = NULL;

// Symbols to be restored after current context finish
static struct restorable_var_info * to_restore_symbols = NULL;

// Number of symbols to be restored after current context finish
static struct sym_am * to_restore_symbols_amounts = NULL;
```

En la lista `allocated_symbols` se guarda la información de todos los símbolos que se van encontrando mientras se recorre el árbol recursivamente en in-order, junto con su tipo.
Al crearse un *contexto (scope) (declaración de función, '{' '}', for_in loop)* se crea un nuevo elemento en la pila `allocated_symbols` al igual que en la pila `used_symbols_amounts` que se corresponde con la cantidad de símbolos alocados bajo el contexto que se está creando.

Al declararse un símbolo bajo el contexto actual, se procede a almacenar su información en la pila `allocated_symbols` y luego incrementar la cantidad de elementos (del tope de la pila `used_symbols_amounts`).

Si un símbolo ya había sido declarado en un *contexto padre (scope superior)* , es decir, aparece en la tabla de símbolos, entonces el símbolo se persiste en la pila de símbolos para restaurar al finalizar el contexto (`to_restore_symbols`) y se incrementa la cantidad de elementos a restaurar (del tope de la pila `to_restore_symbols_amounts`) .

Al finalizar el contexto, primero se *'popean'* tantos elementos de la pila `allocated_symbols` como haya declarados en el tope de la pila `used_symbols_amounts` , liberándose estos símbolos, y finalmente se libera el tope de la pila `used_symbols_amounts`. Luego, se restauran a la tabla de símbolos los elementos que habían sido removidos de la misma durante el contexto actual. Para esto se *'popean'* tantos elementos de la pila `to_restore_symbols` como haya declarados en el tope de la pila `to_restore_symbols_amounts`, restaurandose en la tabla de símbolos, y posteriormente liberando el tope de la pila `to_restore_symbols_amounts`.

De esta forma, se consigue un uso eficiente de memoria pues se dispone de una única tabla de símbolos en cada momento de validación de tipos, cuyo algoritmo de búsqueda es de O(1)..

A la par que se realiza este recorrido (también in-fijo) del árbol, se establecen validaciones de tipos en las distintas declaraciones y asignaciones, así como en las llamadas a funciones, obteniendo los tipos

de la tabla de símbolos. Durante el recorrido, se valida que cada variable utilizada haya sido declarada dentro del contexto actual, además de hacer las debidas comprobaciones.

Una vez que se terminan de hacer todas las validaciones, se informa por salida de error en caso de que haya errores. En caso contrario, se procede a liberar los recursos utilizados para realizar las validaciones de tipos, al igual que los recursos utilizados para los chequeos de funciones.

Traducción a Python

Una vez analizados los tipos utilizados, y solo si no ha habido errores, se sucede con la traducción, es decir pasar el código escrito en un archivo que arbitrariamente definimos de extensión *.im* a sintaxis de python *(.py)*. Para esto se creó una función llamada *pybody()* (y *pybody_ind()* para manejar la indentación en la sintaxis de python, que permite diferenciar los scopes manejados *)* que escribe dentro del archivo de salida en sintaxis de python los elementos que se van procesando. Para realizar la traducción se comenzó por la raíz del árbol armado anteriormente. Cada nodo fue pensado como una función, de manera tal que (cuando se procesa un nodo particular) si su nodo hijo es un terminal entonces se imprime con *pybody()*, sino se llama a la función del próximo nodo.

Ejecutable

Finalmente, para ejecutar el archivo de salida (*out*), se puede ejecutar en la carpeta base del repositorio el comando *./imma out* (o correr el archivo *out* con python 3.8: *python3 out*).

Se incluyeron los siguientes programas para ayudar a entender la sintaxis:

- *hello_world.im*
- *before_after.im*
- *mirror.im*
- *invert_colors.im*
- *Disorganize_image.im*

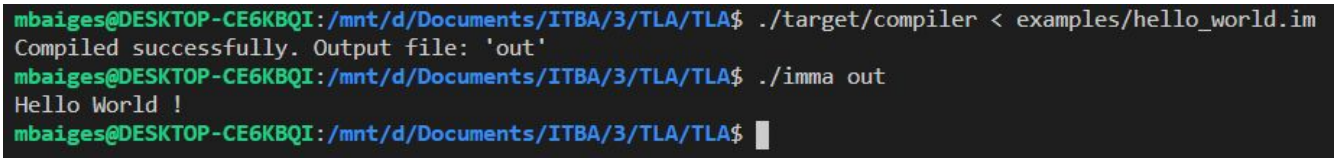
Códigos de ejemplo

./target/compiler < examples/hello_world.im (produce out)

o

make compile INPUT=examples/hello_world.im (produce out)

Salida: ./imma out

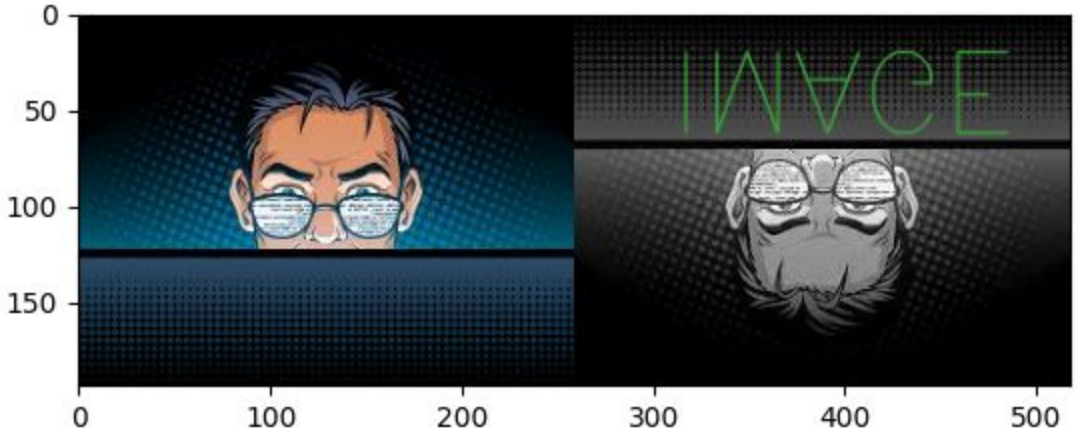


./target/compiler < examples/before_after.im (produce out)

o

make compile INPUT=examples/before_after.im (produce out)

Salida: ./imma out

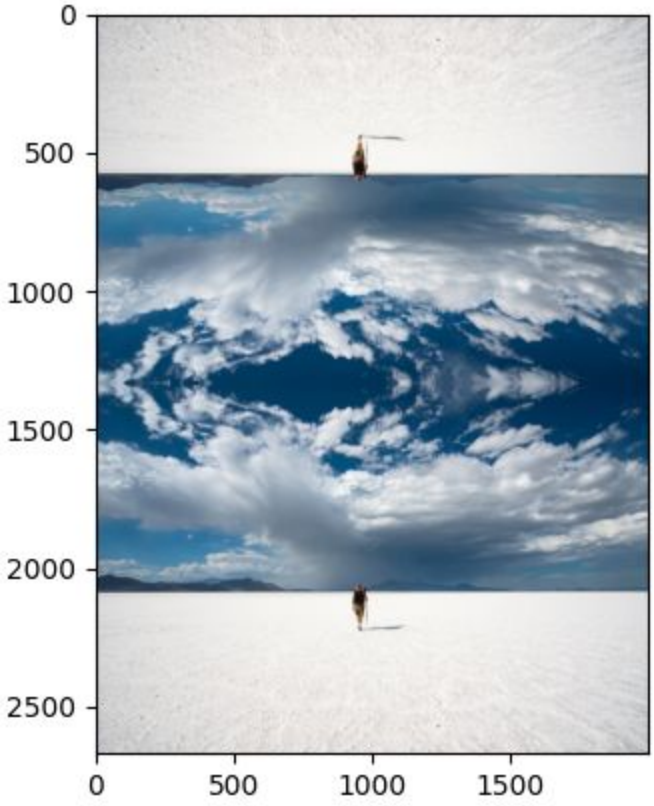


./target/compiler < examples/mirror.im (produce out)

o

make compile INPUT=examples/mirror.im (produce out)

Salida: ./imma out

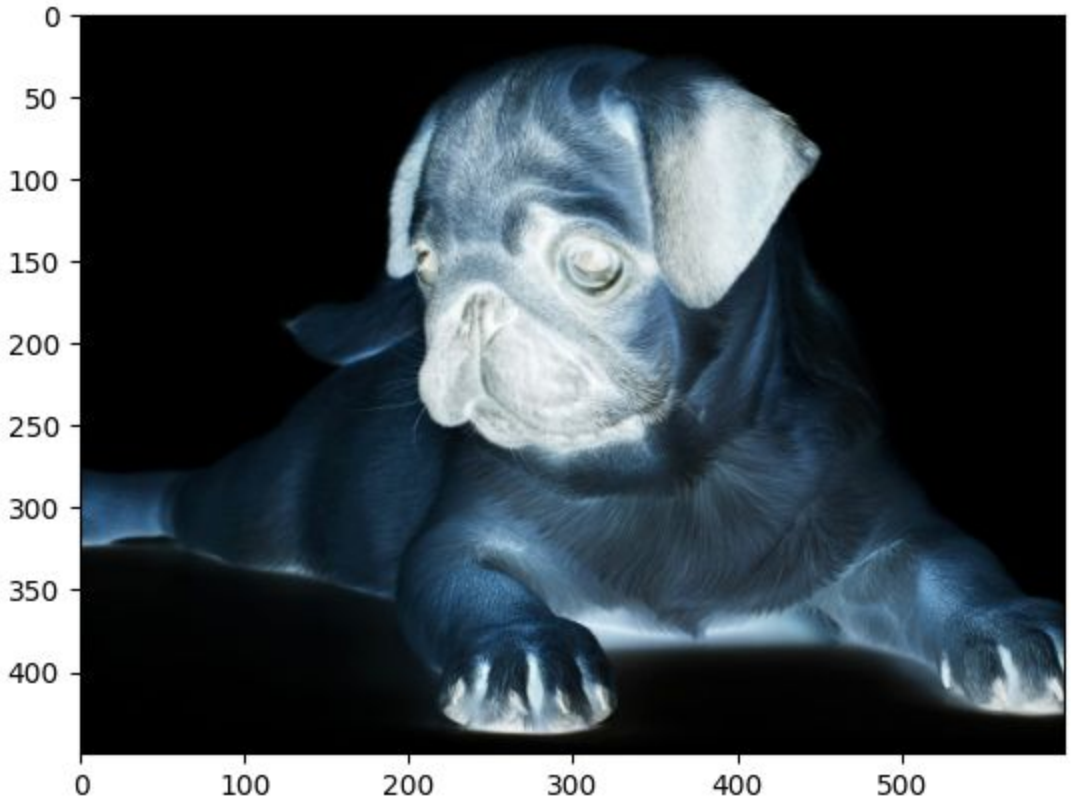


./target/compiler < examples/invert_colors.im (produce out)

o

make compile INPUT=examples/invert_colors.im (produce out)

Salida: ./imma out

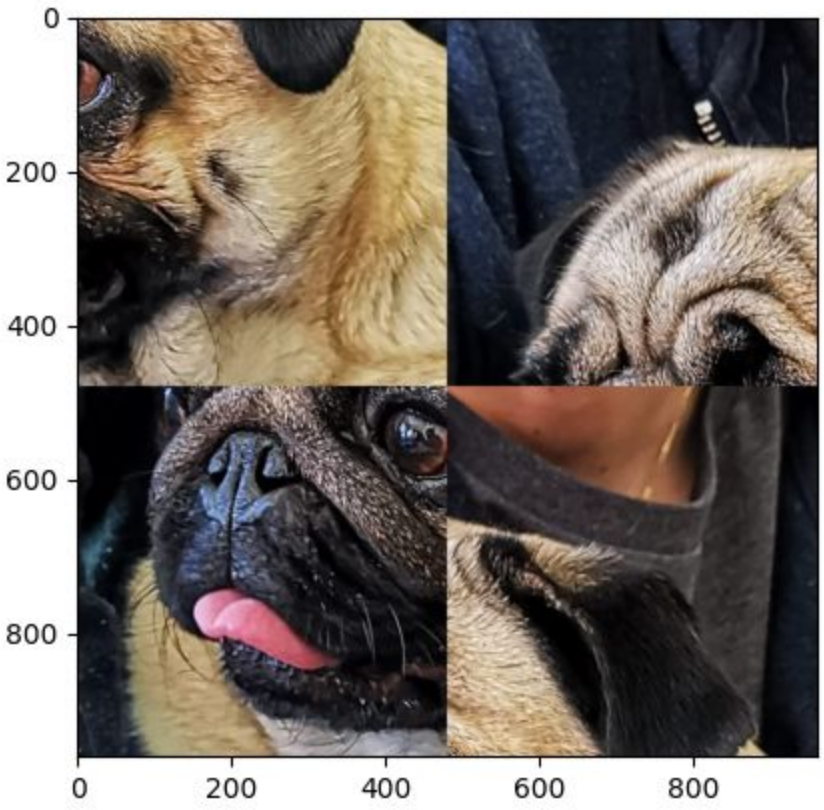


./target/compiler < examples/disorganize_image.im (produce out)

o

make compile INPUT=examples/disorganize_image.im (produce out)

Salida: ./imma out



Descripción de la gramática

Esta es la descripción de la gramática completa de *Image Manager*. Se indicarán los tipos de datos, delimitadores, operadores, bloque condicional, bloque do-while, bloque-for, funciones nativas, declaración de funciones y comentarios.

Nombre: Image Manager
Finalidad: Manejo de imágenes de manera nativa, usando Python como código final.
Tipos de datos:

- Numéricos
  - Entero: `int`
  - Float: `float`
- Cadenas: `string`
- Imagenes: `img(*)`

(*) El tipo de dato subyacente en python se corresponde con una matriz de 3 dimensiones manipulada con el uso de la librería OpenCV.

Delimitadores

"{"
"}"
";"

Operadores

- Aritméticos
  - *+*
  - *-*
  - ***
  - */*
  - *%*
  - *++*
  - *--*

- Relacionales
  - *<*
  - *<=*
  - *>*
  - *>=*
  - *==*
  - *!=*

- Lógicos
  - *&&*
  - *||*

- De Asignación
  - *=*
  - *+=*
  - *-=*
  - */=*
  - **=*
  - *%=*
  - *>>=*
  - *<<=*

- Binarios
  - **^**
  - **<<**
  - **>>**
  - **&**
  - **|**

<u>Bloque Condicional</u>

```
if (cond1) {
      //codigo
}
else {
      //codigo
```

```
}
```

Bloque Do-While

- ```
  while(cond) {
      //codigo
  }
  ```
- ```
  do{
      //codigo
  }while(cond)
  ```

Bloque For

- ```
  for( initial ; cond ; after_each_round ) {
      //codigo
  }
  ```
- ```
  for(type var in array) {
      //codigo
  }
  ```

Funciones nativas

`img load_image(string image_path)`

Esta función permite subir una imagen para poder comenzar a trabajar con ella. Se le pasa como parámetro el *path absoluto* de la imagen con la que queremos trabajar en forma de *string.* Retorna la imagen con tipo *img*.

`show_image(img image)`

Esta función permite mostrar la imagen que se le pasa por parámetro.

`show_image_cmap(img image, string cmap)`

Esta función es un caso específico de la anterior, necesariamente hay que utilizarla cuando en la imagen en cuestión se utilizó la función `gray_scale_image()`.En `cmap` se le pasa el string `"gray"`.

`save_image(string image_path, img image)`

Esta función se utiliza para guardar una imagen (pasada en el segundo parámetro con tipo *img*) en el path indicado pasado en el primer parámetro con tipo string (hay que pasar el path absoluto).

`save_image_cmap(string image_path,img image,string cmap)`

Esta función es un caso específico de la anterior, necesariamente hay que utilizarla cuando en la imagen en cuestión se utilizó la función `gray_scale_image()`. En `cmap` se le debe pasar el string `"gray"`.

`img collage(img[] images1, img[] images2, img[] images3, ...)`

Esta función crea un collage a partir de distintos arrays de imágenes, como mínimo se le debe pasar un parámetro(es decir un array de imagenes). Las imágenes que estén en un mismo array se unen horizontalmente. Esas uniones luego se unen verticalmente entre sí. El conjunto de imágenes en el primer parámetro se sitúa en la parte superior de la imagen resultante, seguida por el segundo conjunto y así sucesivamente. `collage()` retorna la nueva imagen creada de tipo `img`.

`img gray_scale_image(img image)`
Esta función a partir de una imagen (de tipo `img`) crea una nueva imagen en escala de grises.
`NOTA:` Cuando esta función es utilizada la imagen retornada no puede ser utilizada normalmente, se aconseja guardar la imagen con la función `save_image_cmap(string image_path,img image,"gray")` y luego cargarla devuelta con la función `load_image()`. Esta nueva imagen puede ser utilizada normalmente.

`img image_mirror(img image, string axis)`
Esta función, como el nombre lo indica, crea una nueva imagen espejada. Como primer parámetro recibe la imagen que se desea espejar (de tipo `img`) y como segundo parámetro "x" o "y" indicando sobre que eje se la quiere espejar. `image_mirror()` retorna la imagen resultante de tipo `img`.

`img invert_colors(img image)`
Esta función retorna una imagen nueva invirtiendo los colores de una imagen pasada por parámetro.

`img resize_image(img image, int percentage)`
Esta función modifica el tamaño de una imagen manteniendo sus proporciones. Para esto recibe en el primer parámetro la imagen en cuestión y en el segundo la razón en formato de porcentaje) por la cual se la quiere reducir/expandir(de 0 a 100 se reduce en tamaño, de 101 en adelante se expande).

`img rotate_image(img image, int angle)`
Esta función se encarga de rotar una imagen. Recibe como primer parámetro la imagen en cuestión, como segundo parámetro un int representando el ángulo en centígrados.

`img write_on_image(img image, string text, int[2] cords, string font, int fontScale,int[3] color,int thickness, int orientation )`
Esta función permite escribir un string en una imagen. Para esto, recibe los siguientes parámetros:

1. `image:` Imagen sobre la que se imprimirá el texto.
2. `text:` Texto a imprimir en la imagen.
3. `cords:` Arreglo de 2 valores int que representa las coordenadas(x,y) donde se quiere posicionar el texto especificado
4. `font:` Fuente a utilizar en formato de string. Las diferentes opciones son:
    a. `"HERSHEY SIMPLEX"`
    b. `"HERSHEY PLAIN"`
    c. `"HERSHEY DUPLEX"`
    d. `"HERSHEY COMPLEX"`
    e. `"HERSHEY TRIPLEX"`
    f. `"HERSHEY COMPLEX SMALL"`
    g. `"HERSHEY SCRIPT SIMPLEX"`

    h. `"HERSHEY SCRIPT COMPLEX"`
5. `fontScale:` Entero que representa el tamaño del texto.
6. `color:` Arreglo de tipo int con 3 valores representando el color en formato rgb.
7. `thickness:` Representa qué tan grueso será el texto.
8. `orientation:` Si es un valor positivo el texto se imprime en la imagen normalmente, si es negativo este se invierte.

```
img crop_image(img image, int x1, int x2, int y1, int y2)
```
Esta función se encarga de recortar una imagen, `x1` representa la coordenada en x de el lado izquierdo de la imagen resultante, `x2` la coordenada del lado derecho, `y1` del lado superior y por último `y2` el lado inferior. Se retorna una nueva imagen resultado del recorte.

```
string substring(string string, int start, int end)
```
Esta función nos permite obtener una nueva subcadena cadena a partir de una dada. `substring()` recibe `start` refiriéndose al índice donde se quiere comenzar la subcadena, y `end` refiriéndose al índice donde se quiere que finalice la subcadena (incluyendolo a el).

```
int to_int(float num)
```
Esta función recibe un número en forma de `float` y devuelve ese mismo número en `int`.

Comentario

```
// Se utilizan "//" para comentar una línea

/*
  Se utiliza "/*" para iniciar un comentario y "*/" para finalizarlo
*/
```

Definición de funciones

```
type function_name(type param1,type param2){
      // code
      return returnvalue;
}
```

Dificultades encontradas

Durante todo el desarrollo del proyecto se cruzaron varios obstáculos. En un principio fue el planteamiento de la organización del proyecto, como dividir el código para que quede lo más claro posible. Se decidió priorizar la claridad ante todo. Se crearon varios archivos, uno dedicado para cada parte del desarrollo.

Un pequeño inconveniente fue tener que escribir código en python, ya que nuestro lenguaje se basa

en la traducción al mismo. Se tuvo que aprender rápidamente su sintaxis para poder crear las funciones requeridas.

Otro obstáculo importante fue la implementación de la detección de errores de tipos. Durante esta etapa se buscó mantener un buen diseño para poder hacer posible la detección de manera comprensible, pero sin verse afectada la performance del compilador.

A pesar de estos y otros obstáculos menores que hubieran sido encontrados a lo largo del desarrollo del compilador, se lograron superar y de esta manera se obtuvo un compilador que permite definir un lenguaje simple de procesamiento de imágenes.

Futuras extensiones

Al ser un lenguaje basado en python fácilmente se le podrían agregar más funciones de procesamiento de imágenes. Dentro de las posibilidades se encuentra trabajar con filtros, insertado de imágenes dentro de otra, clasificación de imágenes (en un principio se consideró la posibilidad de utilizar transfer learning, partiendo de un modelo de redes neuronales - VGG16 -, que no llegó a ser añadido ya que se prefirió invertir el tiempo del trabajo en mejorar la validación de tipos), reconocimiento de objetos, entre otros.
Lo que se pretende mantener, es la simplicidad , siendo un lenguaje agradable para usuarios inexpertos. Por esto no es necesario, en un principio, tener funcionalidades avanzadas.

Benchmark

Debido a que el compilador de nuestro lenguaje traduce el código a Python y este es un lenguaje interpretado, comparar el tiempo de ejecución de programas escritos en Image Manipulator frente a otros es directamente equivalente a comparar los tiempos de ejecución de Python frente a otros lenguajes.

En el siguiente link se puede encontrar un benchmarking entre Python y C:

https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/python3-gcc.html

Se realizó un benchmark comparando el tiempo de compilación de nuestro lenguaje con C. Para esto se realizó un código equivalente en ambos lenguajes que calcula el factorial de un número y luego lo muestra en pantalla. Los códigos utilizados se encuentran en la carpeta *benchmarking.* Los resultados fueron los siguientes:

```
ltorrusio@LAPTOP-37UCDVFS:/mnt/c/Users/tluci/ITBA/TLA/TLA$ time gcc Examples/factorial_c.c -o factorial

real    0m0.388s
user    0m0.135s
sys     0m0.011s
ltorrusio@LAPTOP-37UCDVFS:/mnt/c/Users/tluci/ITBA/TLA/TLA$ time ./target/compiler < Examples/factorial.im
Compiled successfully. Output file: 'out'

real    0m0.080s
user    0m0.017s
sys     0m0.001s
```

Como se muestra en la imagen, el compilador de IM logró superar en tiempo al de C en este ejemplo en específico. Sin embargo hay que tener en cuenta que es un lenguaje muy reducido y por lo tanto hace que sea más rápido a la hora de compilar, a comparación a gcc que realiza extensivos chequeos e incluso optimizaciones.

Además, se hizo una prueba de rendimiento de un código que encuentra los primeros 1000 números primos. Los códigos utilizados están en la carpeta *benchmarking*. Primero se muestran los resultados de compilación:

```
mbaiges@DESKTOP-CE6KBQI:/mnt/d/Documents/ITBA/3/TLA/TLA$ time gcc -o primes_c benchmarking/primes.c

real    0m0.111s
user    0m0.024s
sys     0m0.016s
mbaiges@DESKTOP-CE6KBQI:/mnt/d/Documents/ITBA/3/TLA/TLA$ time ./target/compiler primes_im < benchmarking/primes.im
Compiled successfully. Output file: 'primes_im'

real    0m0.010s
user    0m0.003s
sys     0m0.000s
mbaiges@DESKTOP-CE6KBQI:/mnt/d/Documents/ITBA/3/TLA/TLA$
```

Luego vemos el resultado:

```
mbaiges@DESKTOP-CE6KBQI:/mnt/d/Documents/ITBA/3/TLA/TLA$ ./primes_c
(0) - 3
(1) - 5
(2) - 7
(3) - 11
(4) - 13
(5) - 17
(6) - 19
(7) - 23
(8) - 29
(9) - 31
```

```
mbaiges@DESKTOP-CE6KBQI:/mnt/d/Documents/ITBA/3/TLA/TLA$ ./imma primes_im
( 0 ) -  3
( 1 ) -  5
( 2 ) -  7
( 3 ) -  11
( 4 ) -  13
( 5 ) -  17
( 6 ) -  19
( 7 ) -  23
( 8 ) -  29
( 9 ) -  31
```

Y finalmente, se midieron los tiempos de ejecución:

```
mbaiges@DESKTOP-CE6KBQI:/mnt/d/Documents/ITBA/3/TLA/TLA$ time ./primes_c 1> /dev/null

real    0m0.007s
user    0m0.005s
sys     0m0.000s
mbaiges@DESKTOP-CE6KBQI:/mnt/d/Documents/ITBA/3/TLA/TLA$ time ./imma primes_im 1> /dev/null

real    0m0.648s
user    0m0.891s
sys     0m1.108s
```

Esta diferencia importante de rendimiento se debe a que, el código compilado con gcc, una vez compilado simplemente se ejecuta. Sin embargo, como el compilador de Image Manipulator traduce a Python, luego se ejecuta un lenguaje interpretado, lo cual demora considerablemente el tiempo de ejecución de cada programa.

Conclusión

Este proyecto creó un cierre perfecto a la materia de Autómatas, Teoría de Lenguajes y Compiladores. Permitió volcar los conocimientos aprendidos en papel y pasarlo a la práctica creando un lenguaje propio. Requirió además, de búsquedas por fuera de las brindadas durante el curso y de esta manera se fueron superando los obstáculos en el camino.

A partir de este trabajo práctico se logró entender cómo crear un lenguaje básico. Sin duda se utilizará como ayuda a proyectos futuros tanto dentro como por fuera de la facultad.

Referencias

https://stackoverflow.com/questions/56587746/create-list-with-variable-length-in-python
Create List with variable length in Python

https://www.epaperpress.com/lexandyacc/
Lex and Yacc Tutorial

https://github.com/attractivechaos/klib/blob/master/khash.h
Hash table