Autómatas, Teoría de Lenguajes y Compiladores:

Imagínate

Marco Scilipoti
mscilipoti@itba.edu.ar

Lautaro Hernando
lhernando@itba.edu.ar

Martin Ippolito
mippolito@itba.edu.ar

22 de junio de 2023

Índice

I. Introducción

Imaginate es un lenguaje diseñado para poder facilitar la creación de *collage* a través de imágenes y código. La filosofía se basa en proveer entropía al proceso de la generación de imágenes y asi ayudar al programador/diseñador elegir el mejor diseño a partir de diferentes imágenes. Este objetivo se representa a través de los métodos opcionales (prefijados con el símbolo *?*), los cuales se aplican o no de manera aleatoria. El principal objetivo de este lenguaje es facilitar el proceso creativo de creación de imágenes mediante la sencilla alteración (mediantes filtros y superposición de las mismas) utilizando código.

II. Descripción del desarrollo del proyecto y Dificultades Encontradas

I. Flex-Bison

Al comienzo fue difícil entender el ejemplo proporcionado por la cátedra por la modularización del mismo para ser escalable. Por lo tanto se hizo referencia al libro *Flex & Bison de John Levine* para poder comprender el funcionamiento de lo mismo. Luego de esto fue posible no solamente escribir una gramática, sino hacer una gramática que comunique claramente la sintaxis del lenguaje.

Luego del tomarse el trabajo de desarrollar la gramatica, no hubo inconvenientes en trasladar la misma a Bison, sin embargo se han tenido que hacer un par de modificaciones leves para el correcto funcionamiento del compilador. En primer lugar se realizaron modificaciones sobre ambiguedades en la misma y leves erro-

res cometidos en el diseño de la misma. Por otro lado, a medida que se fue desarrollado el trabajo, surgio la necesidad de hacer correcciones para la mejor interepretacion del lenguaje (Por ejemplo no se contemplaban los numeros reales con punto como separador decimal).

II. Argumentos-Parámetros

Luego de la primera entrega, implementando el backend, encontramos un error en el diseño de la gramática. Para la primer entrega se consideraba como equivalente los argumentos (formalmente los valores que se le pasan al *llamar* una función) y los parámetros (los valores que se definen en la *signature* de una función y son accedidos en su cuerpo). Por lo tanto se discutió con el profesor Agustin Golmar y se decidió de llevar a cabo un rediseño de esa sección de la gramática. Debido a este legacy cabe destacar que en términos de la gramática y el *Abstract Syntax Tree (AST)* se referencia como argumento a lo que formalmente seria un parámetro y viceversa. Es análogo a LaTeXy el hecho que la *tabla de contenidos* la denomina *índice* y no nos deja cambiarlo.

III. ADTs

Se hace uso de diferentes ADTs i *Abstract Data Types* a través del compilador. Por un lado para manejar los memory leaks se creo un *garbage collector* rudimentario que almacena en una lista encadenada todas las porciones de memoria que asigna para luego liberarlas todas al terminar el programa. Luego se implemento un ADT de un *hashmap* para manejar la tabla de símbolos de funciones y valores (por ser C se efectuó literalmente una copia de este para cada tabla). Por ultimo se implemento otra lista encadenada para poder almacenar los errores de compilación y poder mostrar por salida a estándar todos los errores al usuario y no solamente de a uno.

IV. Code generation y Python

El ultimo paso del trabajo fue la generación de código recorriendo el árbol AST. En términos de eficiencia, se a simplificado el recorrido del mismo gracias al desarrollo de la sintaxis y gramática del lenguaje. Al tener las definiciones (variables y métodos) exclusivamente en el comienzo del código, se debe realizar simplemente un solo recorrido al árbol. Para la generación de imágenes, se ha utilizado la librería *Python Pillow* , el uso de la misma fue bastante sencillo. La principal dificultad estuvo en modelar el código python que se generaría al recorrer el árbol AST, se han realizado múltiples planteamientos hasta encontrar una estructura que se adapte al árbol AST.

III. Fases de compilación

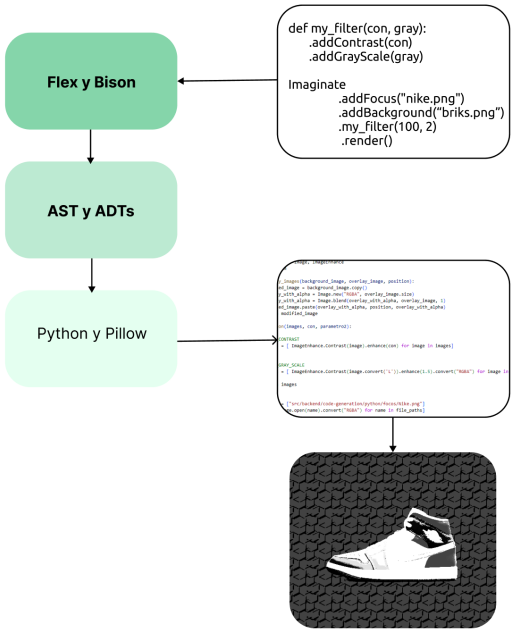

Figura 1: *fases de compilación*

Como se ve en la figura se encuentran las etapas comunes a todos los proyectos (flex, bison, AST). Asimismo se implementaron diferentes ADTS para poder plantear la tabla de símbolos, y partiendo del AST crear una tabla de símbo-

los y presentar los datos necesarios a la etapa de generación de código. Este ultima centrada en generar código en *Python*, haciendo uso de la librería *Pillow* que puede ser ejecutado para crear la imagen o imagenes.

IV. Consideraciones adicionales

Como se ha mencionado anteriormente, se han hecho consideraciones posteriormente a las primeras entregas. En primer lugar las modificaciones a la gramática debido a características propias del lenguajes. Adicionalmente, posterior a las entregas previas, se ha incorporado una nueva funcionalidad para extender el alcance del trabajo que es la posibilidad de tener objetos como variables definidas.

V. Futuras extensiones y/o modificaciones

En termino de posibles extensiones la principal es agregar funcionalidad al manejo de imágenes. Se ha implementado un numero que se considera suficiente de funciones para modificar las imágenes pero hay funcionalidad que decidimos no implementar por un hecho de extensión del proyecto. Asimismo se podrían agregar elementos propios del lenguaje para modificar el flujo de ejecución, incluso a partir de las decisiones innatas del compilador. Por ejemplo, si se eligió cierto *focus* entonces se aplicarían ciertos filtros sobre otros.

Por otro lugar, una posible modificación a futuro es el manejo del tamaño y posición de las imágenes que se incorporan al render final, ya que la librearía utilizada permite esto fácilmente. Sin embargo, debido a la extensión del proyecto y la necesaria reestructuración de la sintaxis, no se ha logrado realizar.

VI. Conclusión

En conclusión, Imagínate se destaca no solo por su capacidad para facilitar la creación de imágenes de manera programática, sino también por su innovadora forma de incorporar la aleatoriedad en el proceso de diseño, permitiendo a los usuarios explorar una multitud de posibilidades y seleccionar las que mejor se adapten a sus necesidades.

VII. Anexo

I. Gramática

A continuación, se incluye una versión simplificada de la gramática. No incluye símbolos terminales, si no que incluye tokens que representa un conjunto de símbolos terminales. Aquellas palabras en minúsculas representan símbolos no terminales y en mayúscula símbolos terminales.

```
program -> assignments definitions
    imaginate

assignments -> assignment assignments
            | /* empty */

assignment -> VAL variableIdentifier COLON
    value
           | assignmentObject

assignmentObject -> VAL variableIdentifier
    COLON object

variableIdentifier -> IDENTIFIER

value -> STRING_IDENTIFIER
      | INTEGER

definitions -> definition definitions
            | /* empty */

definition -> DEF_KEYWORD IDENTIFIER
    argumentsBlock COLON methodChain

emptyParams -> OPEN_PARENTHESES
    CLOSE_PARENTHESES

imaginate -> IMAGINATE focus methodChain
    render
          | IMAGINATE foreachFocus
              methodChain render

focus -> DOT ADDFOCUS paramsBlock

foreachFocus -> DOT FOREACHFOCUS
    paramsBlock

methodChain -> method methodChain
            | /* empty */

method -> DOT optional methodIdentifier
    paramsBlock

paramsBlock -> OPEN_PARENTHESES params
    CLOSE_PARENTHESES

argumentsBlock -> OPEN_PARENTHESES
    arguments CLOSE_PARENTHESES

arguments -> argument
          | argument COMMA arguments
          | /* empty */

argument -> IDENTIFIER

optional -> QUESTION_SIGN
         | /* empty */

params -> param
       | param COMMA params
       | /* empty */

param -> STRING_IDENTIFIER
      | INTEGER
      | variableIdentifier
      | objectElement

objectElement -> variableIdentifier DOT
    variableIdentifier

render -> RENDER emptyParams
       | RENDER_ALL emptyParams

methodIdentifier -> ADDBACKGROUND
                 | ADDFLAVOUR
                 | PICKFLAVOUR
                 | ADDGRAYSCALE
                 | ADDBLACKANDWHITE
                 | ADDCONTRAST
                 | IDENTIFIER

object -> OPEN_CURLY_BRACE objectContent
    CLOSE_CURLY_BRACE

objectContent -> objectAssignment COMMA
    objectContent
              | objectAssignment
              | /* empty */

objectAssignment -> variableIdentifier
    COLON value
```

II. Prestaciones

- Se podrá exportar una imagen (o varias) por programa, donde la imagen exportada será el resultado de aplicar las operaciones mencionadas (background, filter, etc) en el orden establecido en el mismo programa.
- Se podrá setear un foco (`.addFoco(...)`) al cual se le aplicarán todas las operaciones o un conjunto de focos (`.foreachFoco(..., ..., ...)`) a los cuales se aplicarán las operaciones generando múltiples combinaciones.
- Se proveerá una serie de "flavours" (texturas, fondos, etc) que podrán ser identificados viendo la documentación.
- El método `addFlavour(...)` añadirá un flavour al render final.
- El método `pickFlavour(..., ..., ...)` recibirá un conjunto de flavours, que se podrán añadir o no al render final, según decida `imaginate` y el usuario en la renderización final.
- Adicionalmente, se podrán agregar backgrounds para las imágenes.
- Habrá una cantidad determinada de filtros (blur, contraste, etc) que se podrán aplicar en el orden que se desee y cuantas veces quieran. Y dicho filtro se aplicará sobre los elementos previos (ver ejemplos).
- Se proveerá el símbolo "?" antes de un método para decirle que se puede incluir o no en la renderización final, así dejando a `imaginate` elegir si se incluye o no.
- Se proveerá el método `render`, que será el cual finaliza el código y renderiza una única versión del `imaginate` especificado.
- Se proveerá el método `renderAll`, que será el cual finaliza el código y renderiza todas las versiones del `imaginate` especificado. Es decir, genera todas las combinaciones de las opciones definidas a través de `pickFlavour` y los métodos o símbolo "?".
- El background puede ser tanto una imagen determinada como un color sólido.
- Se pueden agregar definiciones, las cuales podrán parametrizar la aplicación de ciertas operaciones. Se deben declarar previo al `Imaginate` y pueden tomar parámetros.
- Se pueden declarar valores, para poder parametrizar en las operaciones a aplicar. El objetivo es evitar los magic numbers.

III. Ejemplos

```
val grayScale: 20

def bAndw():
        imaginate.addContrast(10,
            20).addGrayScale(grayScale)

Imaginate
        .foreachFoco( ./luna. png ,
             ./zapatillas. png ,
             ./reloj. png )
        .addBackground( ./darkblue. png )
        .addFlavour( ./nubes. png )
        .pickFlavour( ./estrellas. png ,
             ./truenos. png ,
             ./lluvia. png )w
        .addFlavour( ./niebla.png)
        .bAndw()
        .renderAll()
```

```
Imaginate
        .addFoco( ./luna. png )
        .addBackground( ./darkblue. png )
        .addFlavour( ./nubes. png )
        .pickFlavour( ./estrellas. png ,
             ./truenos. png ,
             ./lluvia. png )
        .addFlavour( ./niebla.png)
        .?addContrast(20)
        .addGrayScale(20)
        .renderAll()
```