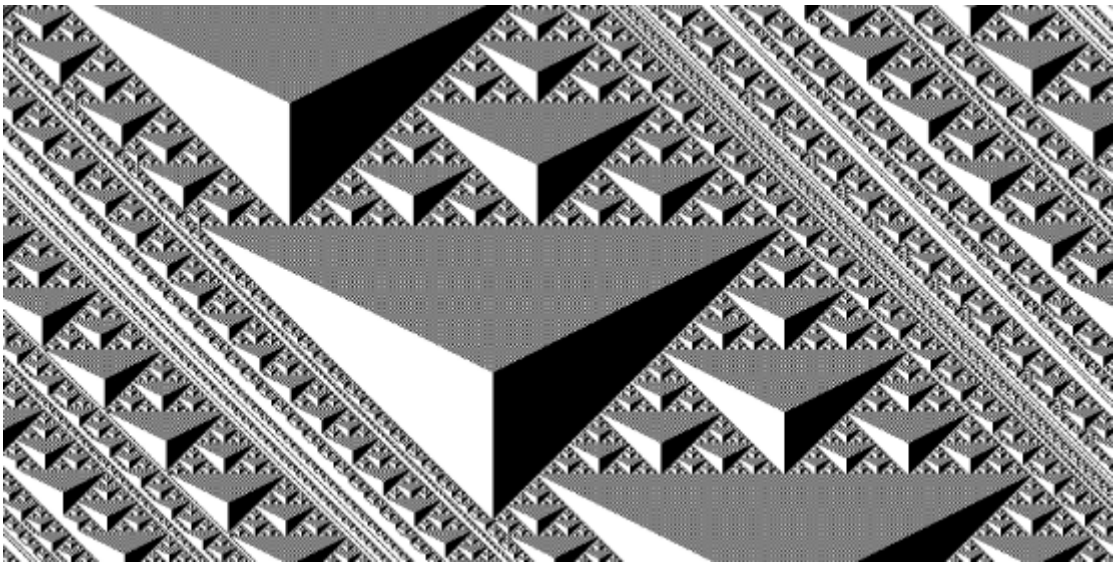

CAL 9000

Cellular Automata Language 9000

V1.0.0

Nombre	Apellido	Legajo	E-mail
Nicolas	Casella	62.311	ncasella@itba.edu.ar
Micaela	Perillo	62.625	miperillo@itba.edu.ar
Timoteo	Smart	62.844	tsmart@itba.edu.ar

Tabla de Contenidos

Introducción

En la web hay muchos juegos que permiten interactuar con un autómata celular famoso como el Juego de la Vida, pero ¿qué pasa si el usuario quiere probar sus propias reglas?

En este informe se presenta CAL 9000, un *Domain Specific Language* diseñado para facilitar la generación de Autómatas Celulares interactivos. El objetivo principal del lenguaje es abstraer al máximo posible la generación gráfica de los autómatas, ofreciéndole al usuario la posibilidad de implementar sus propias reglas de manera sencilla y rápida.

El resultado final se construye utilizando la librería *PyGame* de *Python* para ofrecer al usuario una interfaz gráfica para poder interactuar y experimentar con el autómata creado.

En el presente informe se detallará el proceso de desarrollo del lenguaje, además de los problemas y limitaciones encontrados en el transcurso del cuatrimestre.

Modelo Computacional

Dominio

Un autómata celular es un modelo matemático para un sistema dinámico que evoluciona en pasos discretos, según un conjunto finito de reglas. Para definirlo, se necesitan los siguientes elementos básicos

- Un **espacio**, ya sea una línea, un plano de dos dimensiones o un espacio de n dimensiones. Para los autómatas de este trabajo, consideraremos solo los que viven en el plano bidimensional.
- Un **conjunto de estados**, que en este caso serán vivos o muertos.
- **Vecinos**, conjunto de células cercanas a una dada. En este caso, las células a considerar para el cálculo de la vecindad de una serán a elección del usuario. Algunas a considerar (no implementadas, pero pueden ser de ayuda para quien desee diseñar un autómata), pueden ser la Vecindad de Moore y la Vecindad de Von Neumann.
- **Reglas**, las cuales definirán la cantidad de celular en la vecindad de una dada para que esta pase de un estado a otro.

La nomenclatura utilizada para las reglas en CAL 9000 se define de la siguiente manera: **A/D/B**, donde cada letra representa

A: Cantidad de vecinos vivos para que la célula siga viva, sino muere de soledad

D: Cantidad de vecinos vivos para que la célula muera por sobrepoblación

B: Cantidad de vecinos vivos para que una célula nazca en una celda vacía

Algunos autómatas celulares famosos son el Juego de la Vida de Conway o la Regla 32 de Stephen Wolfram.

Lenguaje

CAL 9000 es un lenguaje diseñado para generar autómatas celulares bidimensionales.

Se pueden especificar las reglas a seguir, dando como parámetro la cantidad de vecinos que debe tener una célula para morir, vivir o nacer. Además, se puede especificar la definición de vecino, para que el cálculo de vecindad solo considere ciertas celdas.

Por otro lado, se puede personalizar la parte gráfica de la interfaz. Se permite cambiar la cantidad de celdas en la grilla, generando un autómata mas complejo, y también la parte estética, haciendo que el automata cambie de color en cada generacion.

El lenguaje generará una salida en Python, utilizando la librería PyGame para poder visualizar el autómata y que el usuario interactúe con el mismo. La extensión de los archivos será .cal9k.

Implementación

En primer lugar, en el proceso de compilación, se realizó un análisis sintáctico en Flex, con los símbolos terminales reconocidos del lenguaje definidos allí. Después, se utilizó Bison. Allí se definieron las producciones a reconocer y la forma en la que se guardan los nodos en memoria. Con ello se realizó el análisis sintáctico y la construcción del *Abstract Syntax Tree*. Finalmente, se genera el código. El resultado es un código en Python que, instalando las librerías necesarias, se puede ejecutar por el usuario.

Previamente a ejecutar el código generado, se deben tener instaladas las siguientes librerías: *PyGame*, *Numpy* y *Tkinter*. Caso contrario, se pueden instalar corriendo el siguiente comando

```
pip install pygame
pip install numpy
pip install tk
```

Para ejecutar en código generado, el usuario se debe ubicar en la misma carpeta que el archivo y correr

```
python cal9k.py
```

Ejemplo de uso

La sintaxis del lenguaje para generar un autómata con una regla 3/4/3 es la siguiente

```
automata (3,4,3), grid (100,100):
     check(1,0);
     check(-1,-1);
     check(2,-1);
     check(-2,0);
automatan't

rule:
     prop:color = (#FFFFFF, #000FFD, #12DCFF);
     prop:bg_color = (#000000);
     prop:wrapping = (true);
     prop:window_width = (400);
     prop:window_height = (300);
     prop:min_time_between_updates = (2);
rulen't
```

La primera función define el autómata. En este caso, el autómata 3/4/3 según la nomenclatura previamente mencionada. La grilla tendrá una dimensión de cien celdas por cien celdas. Además, se define la regla de vecindad, esto es los vecinos que se considerarán en el cálculo de los mismos. En este caso, siendo (0, 0) la celda, se considerarán las celdas resaltadas a continuación

(-2, -2)	(-2, -1)	(-2, 0)	(-2, 1)	(-2, 2)
(-1, -2)	(-1, -1)	(-1, 0)	(-1, 1)	(-1, 2)
(0, -2)	(0, -1)	(0, 0)	(0, 1)	(0, 2)
(1, -2)	(1, -1)	(1, 0)	(1, 1)	(1, 2)
(2, -2)	(2, -1)	(2, 0)	(2, 1)	(2, 2)

Tabla 1: Representación gráfica del cálculo de vecindad

Por otro lado, en rule, se definen propiedades relacionadas al estilo de la interfaz. La lista de colores *prop:color* es el patrón de colores que cambia según la generación. La prop:bg_color define el color para el fondo de la grilla, es decir, las células muertas. La prop:wrapping refiere a la forma en la que los bordes de la grilla. Si se quiere que no existan los bordes, es decir que las células que salgan por el borde derecho entren por el izquierdo se debe definir como True, caso contrario, como False. Las propiedades *prop:window_height* y *prop:window_width* definen el tamaño de la interfaz gráfica. Por último, *prop:min_time_between_updates* define el tiempo (en milésimas de segundo) a esperar desde que se calcula una generación hasta empezar a producir la siguiente.

Dificultades encontradas

Si bien no se encontraron mayores dificultades en el desarrollo del frontend y del backend, la mayoría de ellos ocurrieron en el proceso de definir el tema del trabajo práctico. Se buscó equilibrar que el dominio del lenguaje sea interesante y que la dificultad del lenguaje sea tal que sea posible terminar el trabajo en tiempo y forma.

En la primera entrega, se planteó la posibilidad de que los autómatas pudieran ser generados en 1D, 2D y 3D en el entorno de Godot. Debido a la dificultad que traía esta idea y por sugerencia de la cátedra, se cambió a generar autómatas 2D en PyGame.

También surgieron problemas a la hora de definir de qué manera se iban a especificar las propiedades del autómata dentro del lenguaje. Llevando a múltiples modificaciones e iteraciones de la sintaxis del lenguaje a lo largo del proyecto.

Una vez creado el compilador, se encontraron problemas al intentar correr el programa en Windows. Debido a una restricción de tiempo, se decidió depreciar la compatibilidad con el sistema operativo.

Futuras Extensiones

Algunas features que no se llegó a implementar en el transcurso de este trabajo es la de agregar los operadores =, <, >, <= y >= para que se puedan generar reglas de la forma “La célula muere si tiene menos de dos vecinos”. Por ejemplo, esto podría convertir la sintaxis de definición de autómata en `automata (>3, =2, <4)`.

Además, se podría haber implementado un *Syntax Highlighter* para VSCode para complementar el trabajo y facilitar la escritura de código CAL 9000.

Conclusiones

Se logró aplicar correctamente el conocimiento teórico adquirido a lo largo de la cursada en el trabajo práctico. Esto ayudó a que el grupo se acerque al mundo de los compiladores, que parecía lejano. El trabajo fue interesante ya que permitió desarrollar un tema que nos interesaba, como lo son los autómatas celulares.

Consideramos que fue un buen trabajo y que se cumplieron los objetivos esperados.

Bibliografía

- https://natureofcode.com/cellular-automata/
- https://playgameoflife.com/
- https://github.com/agustin-golmar/Flex-Bison-Compiler