



Instituto Tecnológico de Buenos Aires

CAL 9000

72.39 - Autómatas, Teoría de Lenguajes y Compiladores

Perillo, Micaela - 62625 - miperillo@itba.edu.ar

Segundo Cuatrimestre, 2024

Tabla de Contenidos

Tabla de Contenidos.....	1
1. Introducción.....	2
2. Modelo Computacional.....	3
2.1. Dominio.....	3
2.2. Lenguaje.....	4
2.3. Ejemplos de aceptación.....	4
2.4. Ejemplos de rechazo.....	9
3. Implementación.....	13
3.1. Frontend.....	13
3.2. Backend.....	14
3.3. Extensión de VSCode.....	15
3.4. Dificultades Encontradas.....	16
4. Futuras Extensiones.....	17
5. Conclusiones.....	18
6. Bibliografía.....	19
7. Anexo.....	20
7.1. Implementación de la Vecindad de Von Neumann.....	20
7.2. Implementación de la Vecindad de Moore.....	20

1. Introducción

En la web hay muchos juegos que permiten interactuar con autómatas celulares famosos como el Juego de la Vida, pero ¿qué pasa si el usuario quiere probar sus propias reglas? El objetivo principal de este trabajo es el diseño e implementación de **CAL 9000**, un lenguaje específico de dominio para facilitar la generación de Autómatas Celulares interactivos y personalizados. El mismo busca 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 compilador se implementó en el lenguaje C, utilizando las herramientas Flex y Bison para el análisis léxico y sintáctico. El resultado final se construye utilizando la librería PyGame, de Python, que permitirá que el usuario pueda 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.

2. Modelo Computacional

2.1. Dominio

CAL 9000 es un lenguaje de dominio específico para crear autómatas celulares. Una vez compilado un archivo en dicho lenguaje se devuelve una interfaz gráfica hecha en *PyGame* con las reglas del autómata creado.

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
- Un conjunto de estados
- Una regla de vecindad

El espacio elegido puede ser una línea, un plano de dos dimensiones o incluso un espacio de N dimensiones. Los autómatas considerados en el presente trabajo serán solamente los que viven en el plano bidimensional.

El conjunto de estados son las posibles disposiciones de cada celda. Este mismo debe ser finito. Para este trabajo, se consideran sólo dos estados, por lo que una celda solamente podrá estar viva o muerta.

Los vecinos son, dado una celda, el conjunto de celdas cercanas a la misma. La forma de contar vecinos puede variar según el autómata. Siguiendo el objetivo principal del lenguaje, facilitar la definición de autómatas celulares a elección del usuario, esta regla es elegida por el programador. Algunas reglas famosas que pueden ser interesantes para comenzar a implementar son la Vecindad de Moore y la Vecindad de Von Neumann. Las implementaciones de las mismas se encuentran en el [Anexo](#).

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

2.2. Lenguaje

El lenguaje desarrollado ofrece las siguientes funcionalidades

- Se podrán generar interfaces gráficas de autómatas celulares en 2D en PyGame.
- Se podrán definir las reglas de vecindad en `automata`.
- Se podrá definir el tamaño de la grilla de celdas `grid`.
- Se podrá definir el color de fondo de la grilla en `prop:bg_color`.
- Se podrán definir varios colores para las generaciones generadas en `prop:color`.
- Se podrá definir si se considera que la grilla tiene un buffer circular o no en `prop:wrapping`.
- Se podrá definir el tamaño de la ventana a generar en `prop>window_width` y `prop>window_height`.
- Se podrá definir el tiempo que pasa entre que se genera una nueva generación en `prop:min_time_between_updates`.

A modo de ejemplo, se presenta un autómata que necesita tres vecinos vivos para que la célula siga viva, sino muere de soledad, cuatro vecinos vivos para que la célula muera por sobrepoblación y tres vecinos vivos para que una célula nazca en una celda vacía.

```
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 extensión de los archivos será **.cal9k**.

2.3. Ejemplos de aceptación

Los siguientes programas de prueba deben resultar exitosos al momento de ser compilados.

1. Un programa que define primero automata y después rule.

```

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

rule:
    prop:color = (#FFFFFF, #000FFD, #12DCFF);
    prop:bg_color = (#000000);
    prop:wrapping = (true);
rulen't

```

2. Un programa que define primero rule y después automata.

```

rule:
    prop:color = (#FFFFFF, #000FFD, #12DCFF);
    prop:bg_color = (#000000);
    prop:wrapping = (true);
rulen't

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

```

3. Un programa que solo define automata

```

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

```

4. Un programa que simule un autómata tipo 1/2/1 en una grilla de 10 x 10 con 1 chequeo positivo y 1 regla con 1 parámetro (int)

```
automata (1,2,1), grid (10,10):  
    check(1,1);  
automatan't  
  
rule:  
    prop>window_width = (1280);  
rulen't
```

5. Un programa que simule una regla tipo 5/6/3 en una grilla de 100 x 100 con 6 chequeos negativos y 5 reglas con hasta 3 parámetros (int, boolean, color).

```
automata (5,6,3), grid (100,100):  
    check(0,-1);  
    check(-1,0);  
    check(-2,-1);  
    check(-3,-2);  
    check(-2,-3);  
    check(-1,-2);  
automatan't  
  
rule:  
    prop:color = (#FFFFFF, #000FFD, #12DCFF);  
    prop:bg_color = (#000000);  
    prop>wrapping = (true);  
    prop>window_width = (1280);  
    prop>window_height = (768);  
rulen't
```

6. Un programa que simule una regla tipo 8/9/7 en una grilla de 500 x 500 con 9 chequeos positivos/negativos y 7 reglas con hasta 3 parámetros o más (int, boolean, color).

```
automata (8,9,7), grid (500,500):
    check(1,0);
    check(-1,-1);
    check(2,-1);
    check(-1,0);
    check(-1,1);
    check(-2,-1);
    check(1,3);
    check(-3,-1);
    check(2,-3);
automatan't

rule:
    prop:color = (FFFFFF, #00FFD, #12DCFF, #010203, #FFEEDD);
    prop:bg_color = (000000);
    prop:wrapping = (true);
    prop>window_width = (1280);
    prop>window_height = (768);
    prop:min_time_between_updates = (1);
rulen't
```

7. Un programa con líneas vacías dentro de automata y rule.

```
automata (3,4,3), grid (100,100):

    check(1,0);

    check(-1,-1);
    check(2,-1);

automatan't

rule:
    prop:color = (FFFFFF, #00FFD, #12DCFF);
    prop:bg_color = (000000);

    prop:wrapping = (true);

rulen't
```


8. Un programa con espacios extra entre cada string.

```

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

rule :
    prop : color = ( #FFFFFF , #000FFD , #12DCFF ) ;
    prop : bg_color = ( #000000 ) ;
    prop : wrapping = ( true ) ;
    rulen't

```

9. Un programa que utilice comentarios multilínea dentro de automata y rule

```

automata (3,4,3), grid (100,100):
    /* this is a comment
    hello
    */
    check(1,0);
    check(-1,-1);
    check(2,-1);
    automatan't

rule:
    prop:color = (#FFFFFF, #000FFD, #12DCFF);
    /* this is another comment
    goodbye
    */
    prop:bg_color = (#000000);
    prop:wrapping = (true);
    rulen't

```

10. Un programa que utilice comentarios multilínea fuera de automata y rule.

```

/* this is a comment
I'm outside this time
*/

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

/* this time I'm in the middle
*/

rule:
    prop:color = (#FFFFFF, #000FFD, #12DCFF);
    prop:bg_color = (#000000);
    prop:wrapping = (true);
rulen't

/* yet another comment also outside */

```

11. Un programa que crea una grilla con x distinto de y.

```

automata (1,2,1), grid (100,10):
    check(1,1);
automatan't

rule:
    prop>window_width = (1280);
rulen't

```

2.4. Ejemplos de rechazo

En particular, para que un programa sea válido debe cumplir ciertas reglas.

- En las funciones `automata` y `rule` la capitalización debe ser en minúsculas: no es válido definir `Automata` pero si `automata`.
- La definición de `automata` debe aparecer obligatoriamente, así como también la definición de la regla de vecindad utilizando `check`.

- Las funciones `automata` y `rule` deben ser únicas, no puede definirse más de una en un mismo programa.
- Las propiedades de `automata` y `rule` deben ser las especificadas
- Los números de A, D y B deben ser positivos
- La definición de grilla debe hacerse con números positivos.

Los siguientes programas de prueba no deben resultar exitosos al momento de ser compilados.

1. Un programa con capitalización invalida

```
Automata (3,4,3), grid (100,100):  
    check(1,0);  
    check(-1,-1);  
    check(2,-1);  
Automatan't  
  
Rule:  
    prop:color = (#FFFFFF, #000FFD, #12DCFF);  
    prop:bg_color = (#000000);  
    prop:wrapping = (true);  
Rulen't
```

2. Un programa que no define automata

```
rule:  
    prop:color = (#FFFFFF, #000FFD, #12DCFF);  
    prop:bg_color = (#000000);  
    prop:wrapping = (true);  
rulen't
```

3. Un programa que define múltiples rule

```

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

rule:
    prop:bg_color = (#000000);
    prop:wrapping = (true);
rulen't

rule:
    prop:color = (FFFFFF, #00FFD, #12DCFF);
rulen't

```

4. Un programa con un valor invalido en A

```

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

rule:
    prop:color = (FFFFFF, #00FFD, #12DCFF);
    prop:bg_color = (#000000);
    prop:wrapping = (true);
    prop:color = (FFFFFF);
rulen't

```

5. Un programa con un número invalido (negativo) en la definición de grilla

```

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

rule:
    prop:color = (FFFFFF, #00FFD, #12DCFF);
    prop:bg_color = (#000000);
    prop:wrapping = (true);

```

```
    prop:color = (#FFFFFF);  
rulen't
```

3. Implementación

El proyecto fue dividido en *frontend* y *backend*. La primera etapa es responsable del análisis léxico y sintáctico del programa escrito. En la segunda, se genera el código y se chequean los tipos.

3.1. Frontend

Lo primero que se implementó en esta etapa fue el analizador léxico. Este mismo, mediante el uso de Flex, se encarga de convertir el código fuente en tokens según expresiones regulares definidas.

En el archivo `FlexPatterns.l` se definen las funciones que corresponden a cada token reconocido. Por otro lado, en `FlexActions.c` se encuentran las definiciones de las funciones que son llamadas en el archivo mencionado. Las palabras reservadas del lenguaje son las siguientes: `automata` y `rule` para comenzar a definir la función correspondiente, `automatan't` y `rulen't` para terminar la definición de la misma, `/* ... */` para comentarios, `grid` para definir el tamaño de la grilla de celdas, `check` para definir los vecinos que se verifican en la regla de vecindad, `prop` para, junto a `“:”`, agregar propiedades a `rule`, `true` y `false`. Además, se definió que los colores se pasen en hexadecimal, con una expresión regular que lo verifica.

Por otro lado, para el análisis sintáctico, se empleó Bison. En el mismo se definen las gramáticas libres de contexto utilizadas para analizar la estructura sintáctica del código fuente.

En el archivo `BisonGrammar.y` se definió la gramática del lenguaje propuesto. En el mismo, se encuentran los terminales y no terminales definidos.

Los terminales definidos son: `INTEGER`, `COMMA`, `BOOLEAN`, `CLOSE_PARENTHESIS`, `OPEN_PARENTHESIS`, `AUTOMATA`, `AUTOMATAN_T`, `RULE`, `RULEN_T`, `UNKNOWN`, `COLON`, `SEMICOLON`, `EQUAL`, `CHECK`, `COLOR_HANDLER`, `GRID`, `PROPERTY` y `KEYWORD`.

Los no-terminales definidos son: `automata`, `program`, `check_list`, `check`, `rule_number`, `rule`, `grid`, `data_type`, `parameter_list`, `property_list`, `property`.

La gramática libre de contexto definida es la siguiente

```

program → automata rule | rule automata | automata

rule_number → OPEN_PARENTHESIS INTEGER COMMA INTEGER COMMA INTEGER
              CLOSE_PARENTHESIS

grid → GRID OPEN_PARENTHESIS INTEGER COMMA INTEGER CLOSE_PARENTHESIS

check_list → check | check_list check

check → CHECK OPEN_PARENTHESIS INTEGER COMMA INTEGER CLOSE_PARENTHESIS
        SEMICOLON

automata → AUTOMATA rule_number COMMA grid COLON check_list AUTOMATA_NT

data_type → BOOLEAN | COLOR_HANDLER | INTEGER

parameter_list → data_type | parameter_list COMMA data_type

property → PROPERTY COLON KEYWORD EQUAL OPEN_PARENTHESIS parameter_list
          CLOSE_PARENTHESIS SEMICOLON

property_list → property | property_list property

rule → RULE COLON property_list RULE_NT

```

3.2. Backend

La idea del proyecto es utilizar un *script* hecho en *Python* en el cual define un autómata celular completamente parametrizable. Estas configuraciones se pasan por medio de un diccionario al comienzo del programa denominado `modified_params`. El *script* base de *Python* se puede encontrar en la carpeta: `/src/main/python/cal9k.py`.


De esta manera, al generar el código, se agregan los `includes` necesarios, luego se arma un diccionario con los parámetros que se le pasaron y se sigue con el código ya creado. Eso ocurre en `Generator.c`, en donde, como su nombre indica, va generando las distintas partes del código final: el prólogo con los `includes`, el programa con el diccionario antes mencionado, la regla, etc. hasta llegar a la generación del epílogo, que es la llamada del programa a la función `main`.

Por otro lado, en `Program.c`, se define la función que será llamada directamente desde el `main` del compilador: `computeProgram`. Esta misma es la encargada de llamar a las funciones necesarias, `computeRule` y `computeAutomata` para que se compilen los mismos. Las funciones de `Program.c` utilizan de `Properties.c` para verificar que una propiedad sea válida.

3.3. Extensión de VSCode

A modo de complementar el proyecto y facilitar la escritura de código en el lenguaje propuesto, se implementó la extensión: [CAL 9000 Syntax Highlighting](#). El código de la misma se puede encontrar [aquí](#).

Utilizando la misma, los archivos con extensión `.cal9k` se ven de la siguiente manera



```
automata (3,8,3), grid (100, 100):
    check(0, -1);
    check(0, 1);
    check(1, 0);
    check(1, 1);
    check(1, -1);
    check(-1, 0);
    check(-1, 1);
    check(-1, -1);
    automatan't

rule:
    prop:color = (#00BB00);
    prop:bg_color = (#000000);
    prop:wrapping = (true);
    prop>window_width = (800);
    prop>window_height = (600);
    prop:min_time_between_updates = (1);
    rulen't
```

Imagen 1: Archivo `moss.cal9k` disponible en `/src/sample/c`

3.4. Dificultades Encontradas

Si bien no se encontraron mayores dificultades en el desarrollo del frontend y del backend, la mayoría de ellas 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 deprecia la compatibilidad con el sistema operativo y se recomienda ejecutar el mismo en Ubuntu (o WSL en su defecto).

4. Futuras Extensiones

Como una próxima mejora, se buscaría implementar en el proyecto los operadores: $=$, $<$, $>$, \leq y \geq . Este cambio daría lugar a un gran número de autómatas celulares nuevos, ya que permitiría que se generen autómatas con características de la forma “La célula muere si tiene menos de dos vecinos”. Esto convertiría la sintaxis de definición de autómata en automata (>3 , $=2$, <4) que, por como se programó el compilador, no sería complicado de implementar.

Otra limitación que se podría eliminar es la de tener solo dos estados, vivo o muerto. Se debería permitir que el usuario ingrese N estados y que a cada uno le corresponda un color ingresado.

5. Conclusiones

En el presente informe se detalló el proceso de diseño y desarrollo de un lenguaje y su respectivo compilador para poder programar autómatas celulares. Consideramos que nuestro lenguaje cumple el objetivo principal que nos planteamos, provee una forma muy sencilla de crear una interfaz gráfica interactiva de un autómata celular con parámetros completamente personalizables.

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. Una parte muy destacada del trabajo es que permitió que mientras se reforzaron conceptos vistos en la materia, a la vez, se pueda seguir un interés personal como lo son los autómatas celulares.

6. Bibliografía

- Shiffman, D. [The Nature of Code](#), 2024 <Revisado por última vez: 31/10/2024>
- <https://playgameoflife.com/> <Revisado por ultima vez: 31/10/2024>
- <https://github.com/agustin-golmar/Flex-Bison-Compiler> <Revisado por ultima vez: 31/10/2024>
- Apuntes de la cátedra

7. Anexo

Para ambas implementaciones se tomó el autómata 3/4/3

7.1. Implementación de la Vecindad de Von Neumann

```
automata (3,4,3), grid (100,100):
    check(0,-1);
    check(0,1);
    check(-1,0);
    check(1,0);
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
```

7.2. Implementación de la Vecindad de Moore

```
automata (3,4,3), grid (100,100):
    check(0,-1);
    check(0,1);
    check(-1,-1);
    check(-1,0);
    check(-1,1);
    check(1,-1);
    check(1,0);
    check(1,1);
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
```