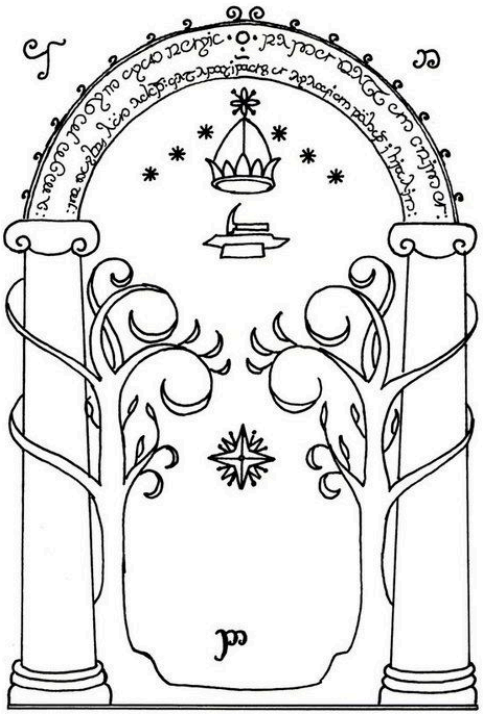


Proyecto Especial

Diseño e Implementación de un Lenguaje

v3.0.6

1. Introducción

Este trabajo presenta el diseño e implementación de un DSL (Lenguaje Específico de Dominio) junto con su compilador. Para ello se utilizaron las herramientas Flex, el cual es un analizador léxico Flex y un analizador sintáctico Bison, tomando como base el proyecto Flex-Bison-Compiler provisto por la cátedra.

El lenguaje propuesto permite definir y simular autómatas celulares 2D, el objetivo principal del proyecto es dar una herramienta flexible y fácil de usar que permita a los usuarios modelar sistemas complejos sin necesidad de avanzados conocimientos en el tema. Se pueden definir desde autómatas clásicos como lo es "El Juego de la Vida" de Conway hasta modelos con múltiples estados, vecindarios customizados y reglas de transiciones no convencionales.

En este documento se expresa con detalle el lenguaje, decisiones de diseño, implementación y desafíos encontrados durante el desarrollo.

2. Equipo

Nombre	Apellido	Legajo	E-mail
Filipo	Ardenghi	64306	FILIPO ARDENGHI
Agustín Tomás	Romero	64268	AGUSTÍN TOMÁS ROMERO
Federico Ignacio	Ruckauf	64356	FEDERICO IGNACIO RUCKAUF
Pedro	Salinas	64388	PEDRO SALINAS

3. Modelo Computacional

3.1. Dominio

Como ya se mencionó anteriormente el dominio de este lenguaje son los autómatas celulares en dos dimensiones, un sistema matemático y computacional el cual consta de una rejilla formada por celdas (células) que cambian o no de estado dependiendo de leyes establecidas previamente. Para esto un autómata celular necesita una serie de elementos básicos para su funcionamiento, además de la grilla, utiliza un conjunto de estados que son los "valores" que pueden tomar las celdas, una vecindad (Neighborhood) la cual define qué celdas se consideran adyacentes a una dada y finalmente la Función de Transición que define cómo va a cambiar la célula respecto a su entorno. También se incluyó la opción de asignar colores personalizados a cada estado definido y poder definir cómo interactúan las celdas cuando se encuentran con una frontera (fin de la grilla)

El lenguaje permite definir altura (HEIGHT) y ancho (WIDTH), tantos estados como sean necesarios y los colores correspondientes a cada estado. Para definir la vecindad se pueden usar configuraciones preexistentes como MOORE, VON NEWMANN y K NEIGHBORHOOD pero también se permite definir uno personalizado. Las fronteras permitidas son MIRROR, OPEN y PERIODIC. Finalmente para el cambio de estado de las

células se define una función declarativa en formato (Evolución) o también está permitido una función imperativa (Transición)

Este lenguaje permite simular una gran variedad de fenómenos físicos, computacionales y hasta biológicos como lo pueden ser propagación de epidemias, expansión de incendios, simulación de flujos de tránsito, etc.

3.2. Lenguaje

La forma de definir la configuración en el modelo presentado es mediante el uso de tres bloques: **configuration**, **neighborhood** y **transition**.

El bloque **configuration** se puede tomar como la base del programa; en este se definen los atributos obligatorios, como por ejemplo, las dimensiones de la grilla, las condiciones frontera, etc. El formato del bloque es:

```
configuration:
    Height: 100;
    Width: 100;
    Frontier: Open;
    States: {a, b, c};
    Colors: {#000000, #FF0000, #00FFFF};
```

En este ejemplo se ven:

- **Height y Width**: Estos atributos especifican las dimensiones de la grilla. Son enteros mayores a 0.
- **Frontier**: Es el atributo que marca el comportamiento de las células con los bordes. Hay tres opciones:
    - **MIRROR**: Las posiciones de la grilla toman el mismo valor que la celda que lo enfrenta.
    - **PERIODIC**: Forma una grilla 'infinita' ya que conecta el lado izquierdo con el derecho y conecta el lado superior con el inferior.
    - **OPEN**: Las celdas que están fuera de la grilla no afectan al cambio de estados de las células.
- **States**: Es el conjunto de estados que identifican todos los valores que pueden tomar las células.
- **Colors**: Es una lista de colores en notación hexadecimal; cada uno representa uno de los estados. Debe coincidir en cantidad y orden con los elementos definidos en el conjunto de estados.

Después de definir la configuración básica, el usuario deberá elegir entre dos opciones para definir la evolución de las células.

La primera opción es incorporar los atributos **Neighborhood** y **Evolution** al bloque **configuration** que fue definido anteriormente. Esta incorporación es de la forma:

```
configuration:
    Height: 100;
```

```
Width: 100;
Frontier: Open;
States: {a, b, c};
Colors: {#000000, #FF0000, #00FFFF};
Neighborhood: MOORE;
Evolution: CONWAY;
```

Los atributos definidos pueden tener ciertos valores:

- **Evolution**: Define la lógica que usarán las células para cambiar de estado. Este atributo puede tomar tres formas:
    - **CONWAY**: Aplica las reglas del "Juego de la Vida" de Conway.
    - **SEEDS**: Aplica la regla /2 en formato S/B.
    - **S/B**: Tiene un formato personalizado como por ejemplo 1,2,3,4/6,7.
- **Neighborhood**: Define qué celdas se consideran vecinas. Puede tomar los valores:
    - **MOORE**: Las 8 células adyacentes son las que conforman el vecindario.
    - **VON_NEUMANN**: Las 4 células más cercanas son parte del vecindario.
    - **CUSTOM**: Permite al usuario definir un vecindario propio utilizando el bloque **neighborhood**. Este bloque se define luego de **configuration** de la forma:

```
neighborhood:
    for i in [-3, 3] do
        add((i, 0));
        add((0, i));
    end
```

En este bloque la función "add" agrega esa celda relativa al neighborhood.

La segunda opción reemplaza los dos atributos explicados anteriormente y abarca una personalización completa de la lógica de evolución.

```
transition:
    for i in [1,3] do
        if (0,i) == first && (i,0) == second then
            ->first;
        else
            ->second;
        end
    end
```

Al igual que en **Neighborhood** se pueden utilizar estructuras como **`if`**, **`else`** y **`for`**. Para indicar el cambio de estado se realiza de la forma `->estado` y las expresiones `(x,y)` hacen referencia al estado de una celda relativa a la posición actual.

4. Implementación

Para poder procesar este lenguaje de manera efectiva, y luego generar en base a él código python funcional para ejecutar el autómata, se llevaron a cabo dos etapas de desarrollo diferentes.

4.1. Frontend

En esta etapa del proyecto se completó el diseño final de la sintaxis del lenguaje elegido y se comenzó a construir la solución. Se definió una gramática con un conjunto de producciones tipo 2 (según la Jerarquía de Chomsky). La etapa se dividió en 3 partes:

1. **Análisis Léxico:** Transformación de un programa de entrada en un stream de tokens.
2. **Análisis Sintáctico:** Transformación del stream de tokens en un Árbol de Sintaxis Abstracta.
3. **Testing:** Ejecución del script de testing provisto en el repositorio base, aceptando o rechazando cada uno de los casos de prueba.

Para el análisis léxico, lo que se hizo fue transformar los programas de entrada en un stream de tokens a través del archivo `FlexPatterns.l`, desarrollando las equivalencias entre cada potencial lexema y tokens a procesar por nuestro analizador sintáctico. Además, haciendo uso de "`Lexeme Actions`" en el archivo `FlexActions.c`, almacenamos información necesaria para los análisis posteriores y la generación del código final. Por ejemplo, si se recibe un número entero, además de corresponderlo con el token "`INTEGER`", almacenamos su valor numérico para procesarlo más adelante.

Habiendo tokenizado la entrada, pasamos al análisis sintáctico. Para esta etapa, se confeccionó una gramática libre de contexto que representa el lenguaje aceptado por el compilador. Las producciones de esta gramática se desarrollaron en el archivo `BisonGrammar.y`, donde para cada producción se ejecuta la "`Semantic Action`" correspondiente, las cuales están implementadas en el archivo `BisonActions.c`. Estas funciones transforman las cadenas de tokens de cada producción en información pertinente para el análisis semántico y la generación de código de la siguiente etapa. Por ejemplo, en una estructura "if", la producción sería:

```
IF arithmetic_expression THEN expresion_sequence END
```

para la cual se llama a la función

```
IfExpressionSemanticAction(arithmetic_expression, expresion_sequence)
```

que procesa y almacena la condición del if y el cuerpo del mismo en una estructura que lo representa, lo cual es lo que se precisa analizar en etapas posteriores y luego se utiliza para la generación de código.

Por último, en esta etapa se desarrollaron tests con casos de aceptación y rechazo para potenciales programas del lenguaje, de los cuales no todos podían ser correctamente verificados ya que todavía faltaba la implementación del análisis semántico.

4.2. Backend

En esta etapa final se desarrolló la última etapa de análisis y se implementó la generación efectiva de código python en base a la entrada del compilador:

1. **Análisis Semántico:** Extracción de información y validación en base al AST.
2. **Generación de Código:** Producción de los artefactos finales de la solución.

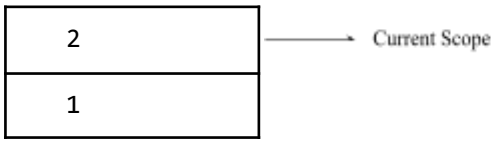
Para complementar las previas etapas de análisis, y poder verificar los casos de aceptación y rechazo faltantes, se implementó una fase de análisis semántico con el uso del AST previamente construido.

Esta fase es la encargada de verificar la coherencia del programa validando las reglas que no se pueden validar haciendo uso exclusivo de la gramática libre de contexto. La lógica de toda esta etapa se encuentra principalmente en el archivo `SemanticAnalyzer.c` cuya única función global es checkSemantic la cual recibe el AST generado en la etapa anterior y lo va validando de a sectores. Para lograr esa validación hay una serie de funciones estáticas como por ejemplo `validateDefaultConfiguration` que van validando el AST de forma recursiva con ayuda de funciones auxiliares.

Un ejemplo de las validaciones que se hicieron es que para un tipo de programa `DEFAULT` (es decir que no tiene ni neighborhood personalizado ni bloque transición) con vecindad `VON_NEUMANN`, las reglas de evolución no deben superar el número máximo de vecinos que establece la vecindad elegida, es decir, cuatro. Si se define una regla de nacimiento o supervivencia que contengan un número mayor a 4 el analizador semántico va a detectar esto y va a reportar un error ya que si bien la sintaxis es correcta el programa es inconsistente.

Para poder verificar la validez del uso de las variables en el código, incorporamos una tabla de símbolos, junto con un stack de scopes de la siguiente manera:

Name	ReadOnly	Scope
`alive`	`true`	`1`
`dead`	`true`	`1`
`x`	`false`	`1`
`y`	`false`	`1`
`x`	`false`	`2`



De esta forma podemos manejar variables definidas en diferentes bloques de código, como lo son los cuerpos de las estructuras `for`, `if`, y `else` en las funciones `transition` y `neighborhood`. Cada vez que se ingresa en un bloque nuevo se hace un `push` en el stack de

scopes, para cada nueva variable declarada se marca con el scope actual. Al buscar una variable se busca entre las del scope actual y si no se encuentra se busca en el scope anterior, y así sucesivamente hasta encontrarla. Al salir de un bloque de se hace un `pop` en el stack de scopes y se borran las entradas de la tabla de símbolos asociadas a ese scope.

Si todas las validaciones que se realizan en esta etapa son exitosas el programa se lo considera semánticamente correcto y se habilita la siguiente etapa de compilación.

Finalmente, habiendo atravesado todas las etapas de análisis necesarias, llevamos a cabo la generación de código. En esta instancia, se asume que todas las entradas que no pertenecen al lenguaje fueron filtradas por los correspondientes análisis, por lo que la generación sólo se efectuará sobre programas válidos para el lenguaje definido. Partiendo de la estructura `Program`, el generador procesa cada una de sus subestructuras, generando código python para cada una de ellas. Para esta implementación se tuvo en cuenta que el código python utiliza la indentación como parte esencial de su estructuración. Por ejemplo, para el caso de un "if" dentro de una función de transición, se generaría:

```
_output(indentation, "if ");
_generateArithmeticExpression(transitionExpression->ifCondition);
_output(0, ":\n");
_generateTransitionSequence(indentation + 1, transitionExpression->ifBody)
```

haciendo uso de la condición y el cuerpo del if que habían sido previamente almacenadas en la etapa de análisis sintáctico, los cuales son enviados a las funciones correspondientes para su generación. La condición es enviada a "`_generateArithmeticExpression`" ya que es una expresión aritmética, y el cuerpo es enviado a "`_generateTransitionSequence`" ya que es una secuencia de expresiones, la cual podría contener dentro de sí misma otro if, o cualquier otra estructura de control, por lo que cada expresión de esa secuencia debe ser procesada. La función se encarga de esto.

Finalmente, para que el código generado en base al programa de entrada sea efectivamente un código python válido, el cual al ejecutarse da lugar a un autómata celular interactivo de dos dimensiones, se le deben incorporar un prólogo y un epílogo. En el prólogo se le anexan los "imports" necesarios, como pygame y numpy, librerías utilizadas para el desarrollo de la aplicación python, además de definirse ciertas constantes. En el epílogo se incorpora el resto de la aplicación. Se definen las funciones que arman la ventana de pygame, junto con la grilla y la interfaz gráfica de usuario (GUI), la función update que es la que actualiza la pantalla en cada iteración del autómata, y finalmente la función main que se encarga de incorporar todas las etapas del código y generar el flujo de la aplicación.

4.3. Dificultades Encontradas

Al implementar scopes en la tabla de símbolos nos encontramos con el problema de que desde el análisis sintáctico no podíamos identificar el comienzo y el final de un bloque de código. Esto nos llevó a pensar que habíamos cometido algún error grave en la etapa de frontend, ya que por el diseño de la gramática no podíamos identificar estos delimitadores. Finalmente, le realizamos la consulta al profesor del trabajo práctico, quién nos indicó que Bison tiene una

feature llamada `mid-rule-action` que permite realizar acciones entre símbolos del lado derecho de una producción, y no solo al final como nos habían enseñado.

5. Futuras Extensiones

Al experimentar con nuestro compilador funcional, nos dimos cuenta que sería útil para el desarrollo de ciertos autómatas la posibilidad de verificar la cantidad de iteraciones que lleva cada celda en el estado actual. Esto sería especialmente útil por ejemplo al generar un autómata que simula la difusión de una enfermedad. Esto permitiría modelar por ejemplo que cada cierta cantidad de iteraciones las células infectadas se recuperan. Para que esto sea posible, se debería agregar al lenguaje la posibilidad de acceder no solo al estado actual de las celdas sino también el número de iteraciones que lleva en ese estado. Se podría plantear una sintaxis de la siguiente forma:

```
if (0,0) == infected && (0,0).age >= 10
    ...
```

siendo age la cantidad de iteraciones que lleva esa celda infectada.

A su vez, creemos que la aplicación de python generada podría tener ciertas mejoras de GUI, como un selector de estados a la hora de marcar cada celda. De esta forma se haría más ameno el uso para autómatas de más de dos estados, ya que actualmente solo se puede ciclar sobre los estados con click izquierdo, o volver al estado 0 con click derecho. Además creemos que sería útil tener varios “grosores” para marcar. En vez de que cada click marque una celda, que marque un cuadrado de 2x2 celdas, uno de 3x3 y así sucesivamente.

6. Conclusiones

Como conclusión, este trabajo finaliza con el completo diseño e implementación de un Lenguaje Específico de Dominio (DSL) para la simulación de autómatas celulares. Se desarrolló un compilador que transforma programas escritos en nuestro lenguaje a código Python, a través del análisis sintáctico, léxico y semántico, hasta la generación de código. Este compilador realiza las validaciones necesarias para aceptar y rechazar, en caso de errores, los programas analizados. Finalmente, devuelve un código Python listo para ser ejecutado en un entorno que poseea las respectivas herramientas y librerías necesarias instaladas.

Consideramos que este trabajo fue de gran utilidad para la materia ya que involucra de manera directa los temas abordados en la parte teórica y práctica de la misma, aún más en nuestro caso que elegimos desarrollar un lenguaje justamente para la simulación de autómatas.

7. Referencias

https://github.com/agustin-golmar/Flex-Bison-Compiler/tree/v1.1.0
https://es.wikipedia.org/wiki/Aut%C3%B3mata_celular
https://en.wikipedia.org/wiki/Seeds_(cellular_automaton)#cite_ref-msz_1-0

8. Bibliografía

Introducción a la teoría de autómatas, lenguajes y computación
Jeffrey Ullman, John Hopcroft y Rajeev Motwan