Treeggered

Diseño e Implementación de un Lenguaje

v1.0.0

Trabajo Práctico Especial
Autómatas, Teoría de Lenguajes y Compiladores 72.39
Grupo G-67: Treeggered
2C - 2024

Tabla de Contenidos

1. Equipo

Nombre	Apellido	Legajo	E-mail
Pedro	Seggiaro	63337	pseggiaro@itba.edu.ar
Santiago Joaquín	Nartallo Galvagno	62208	snartallogalvagno@itba.edu.ar
Martina	Schvartz Tallone	62560	mschvartztallone@itba.edu.ar
Candela	Silva Diniz	63402	csilvadiniz@itba.edu.ar

2. Repositorio

La solución y su documentación serán versionadas en: GitHub

3. Introducción

Este informe describe el trabajo práctico de la materia que consiste en el desarrollo de un compilador diseñado para la generación de árboles, que permite a los usuarios crear escenarios personalizados con múltiples figuras y/o estructuras de diversas formas y tamaños. El sistema está basado en un lenguaje de dominio específico (DSL) que facilita la definición y configuración de las características de éstos, ofreciendo una herramienta sencilla y flexible de usar para poder modelar dichos espacios/mundos naturales.

El objetivo principal del compilador es procesar archivos escritos en este lenguaje y generar una representación visual detallada de los árboles definidos por el usuario, permitiéndole a éste utilizar un entorno adaptable a sus necesidades creativas. La implementación se llevó a cabo usando el lenguaje de programación C, complementado por las herramientas Flex y Bison para el análisis léxico y sintáctico mediante el uso de un Flex-Bison Compiler que nos fue dado por el profesor de la materia, Agustín Golmar. Éste informe detalla todo el proceso de desarrollo, los desafíos encontrados y las soluciones implementadas, así como también muestra los aspectos técnicos más destacados de la construcción y componentes del compilador.

4. Modelo Computacional

4.1. Dominio

Se desarrolló un lenguaje de dominio específico (DSL) orientado a la creación de árboles, permitiendo que, al compilar un archivo escrito en este lenguaje, se genere un escenario personalizado compuesto por estructuras arbóreas de distintas formas y tamaños, representado gráficamente en un formato visual intuitivo.

El lenguaje le da la posibilidad a los usuarios de definir uno o más árboles, especificando características como la altura, el grosor del tronco, el tipo de hojas y su distribución en el espacio. Además, se pueden combinar elementos como patrones de ramificación y la densidad del follaje para poder generar configuraciones únicas y reutilizables en múltiples escenarios.

Si bien sería posible manejar formaciones ramificadas sin una categorización explícita, el uso de tipos default o predeterminados de éstas estructuras permite reducir la complejidad del proyecto inicial para aquellos usuarios que sean principiantes y no sepan bien cómo funciona el lenguaje, de todas formas, se pueden aprovechar los errores de compilación como guía para aquellos que se encuentren en esa situación.

Asimismo, el lenguaje incluye el manejo de conjuntos reutilizables, que permiten definir atributos comunes entre configuraciones arborescentes similares, como patrones de crecimiento, colores o densidad de ramas. Estos conjuntos se manejan a través de múltiples operaciones básicas como la unión, la intersección y la diferencia, haciendo de esta configuración una más flexible y eficiente.

Una vez implementado con éxito, éste lenguaje le otorga al usuario una herramienta para crear escenarios naturales, permitiéndole diseñar, personalizar y optimizar entornos de una forma más intuitiva, sin necesidad de preocuparse por aspectos técnicos complejos o recordar la sintaxis de representación gráfica específica.

4.2. Lenguaje

1. Comentarios

Los comentarios están basados en el formato de comentario de múltiples líneas provisto por la sintaxis del lenguaje de programación C.

```
/* Comentario de una sola línea */

/* Comentario
de múltiples líneas
*/
```

2. Generar un mundo

El componente ***world*** en el lenguaje permite definir el escenario global donde se ubicarán los árboles. Este componente es opcional; si no se define un ***world***, el escenario se considera ***worldless***. En este caso, los árboles se posicionarán en un espacio abstracto con configuraciones específicas (valores default/predeterminados).
Un componente ***world*** contiene las siguientes propiedades clave:

- **height**: Define la altura del mundo en unidades. Este valor determina el límite vertical del escenario. Por ejemplo, *height = 100* establece que el mundo tiene 100 unidades de altura.
- **width**: Define el ancho del mundo en unidades. Este valor determina el límite horizontal del escenario. Por ejemplo, *width = 100* configura un mundo de 100 unidades de ancho.
- **uneveness**: Representa la irregularidad o rugosidad del terreno. Este parámetro puede influir en la distribución o posición de los árboles dentro del mundo. Por ejemplo, *uneveness = 1* indica un terreno levemente desigual.
- **message**: Permite incluir un mensaje personalizado para describir o anotar características del mundo. Este mensaje podría mostrarse en la visualización final del escenario. Por ejemplo, *message = 'something'* agrega un comentario genérico.

```
/* Definimos un mundo */
world {
    height = 100, /* Altura del mundo en unidades */
    width = 100, /* Anchura del mundo en unidades */
    uneveness = 1, /* Nivel de irregularidad del terreno */
    message = 'A beautiful forest' /* Mensaje descriptivo */
}
```

3. Generar un árbol

El componente ***tree*** permite la creación de árboles individuales con características específicas. Cada árbol definido en el lenguaje puede incluir una amplia gama de atributos para personalizar su apariencia y comportamiento dentro del escenario.

Un árbol se define mediante el identificador ***tree***, seguido de un nombre único para el árbol, y las propiedades deseadas, agrupadas en una declaración ***with*** entre paréntesis.

Propiedades disponibles para un árbol

1. **x**:
   - ➢ Define la posición horizontal del árbol en el escenario.
   - ➢ Ejemplo: *x = 1* posiciona el árbol en la coordenada 1 en el eje horizontal.
2. **height**:
   - ➢ Especifica la altura del árbol en unidades.
   - ➢ Ejemplo: *height = 1* indica que el árbol tiene una altura de una unidad.
3. **leaf**:
   - ➢ Representa el carácter utilizado para simbolizar las hojas del árbol en su representación gráfica.
   - ➢ Ejemplo: *leaf = '&'* utiliza el símbolo **&** como hoja.
4. **color**:
   - ➢ Define el color del árbol en formato hexadecimal (RGB).
   - ➢ Ejemplo: *color = #FFFFFF* asigna al árbol el color blanco.
5. **depth**:
   - ➢ Indica la profundidad del árbol, que puede influir en la complejidad de su estructura ramificada y en la posición en la que se encuentre llegado el caso de generar un bosque.
   - ➢ Ejemplo: *depth = 1* establece una profundidad básica.
6. **density**:
   - ➢ Controla la densidad del follaje, es decir, cuán llenas de hojas están las ramas del árbol.
   - ➢ Ejemplo: *density = 1* genera un follaje poco denso.
7. **bark**:
   - ➢ Especifica el grosor del tronco del árbol.
   - ➢ Ejemplo: *bark = 2* configura un tronco con grosor 2.
8. **snowed**:
   - ➢ Determina si el árbol está cubierto de nieve (*booleano*).
   - ➢ Ejemplo: *snowed = TRUE* indica que el árbol está nevado.

```
/* Definimos un árbol con características específicas */
tree t1 with (
   x = 1, /* Posición horizontal en el escenario */
   height = 1, /* Altura del árbol */
   leaf = '&', /* Símbolo para las hojas */
   color = #FFFFFF, /* Color blanco para las hojas */
   depth = 1, /* Profundidad básica */
   density = 1, /* Follaje poco denso */
   bark = 2, /* Tronco con grosor 2 */
   snowed = TRUE /* El árbol está nevado */
);
```

4. Generar un bosque

El componente *forest* permite la agrupación y manipulación de múltiples árboles dentro de un rango específico del mundo. Este elemento facilita la organización de árboles en bloques coherentes y la aplicación de operaciones en conjunto sobre los mismos.

Un bosque se define mediante el identificador *forest*, seguido de un nombre único y las propiedades deseadas, agrupadas en una declaración *with* entre paréntesis.

Propiedades del bosque

1. **start**:
   - ➢ Define el inicio del rango horizontal que ocupará el bosque en el mundo.
   - ➢ Ejemplo: *start = 0* indica que el bosque arranca en la coordenada 0 horizontal.
2. **end**:
   - ➢ Define el final del rango horizontal que ocupará el bosque en el mundo.
   - ➢ Ejemplo: *end = 20* marca que el bosque termina en la coordenada 20 del eje horizontal.

```
/* Definimos un bosque que ocupa un rango de 20 unidades */
forest f1 with (
    start = 0,    /* Coordenada inicial del bosque */
    end = 20      /* Coordenada final del bosque */
);
```

5. Agregar árboles a un bosque

Los árboles pueden añadirse al bosque mediante la operación de incremento +=, que asocia un árbol existente al bosque previamente definido.

```
/* Agregamos árboles al bosque */
f1 += t1; /* Añade el árbol t1 al bosque f1 */
f1 += t2; /* Añade el árbol t2 al bosque f1 */
```

6. Iteración y manipulación de árboles en un bosque

Los árboles pertenecientes a un bosque pueden iterarse utilizando una sintaxis del estilo *for-in*. Esto permite realizar operaciones en conjunto, como modificar atributos individuales o aplicar reglas específicas.

```
/* Incrementamos la altura de todos los árboles en el bosque */
for t in f1 {
    t->height += 1; /* Incrementa la altura de cada árbol en el bosque f1 en 1
                       unidad */
}
```

7. Condicionales

Los condicionales permiten ejecutar ciertas instrucciones basadas en condiciones específicas. En el lenguaje que estás desarrollando, puedes definir condicionales utilizando la estructura ***if***, que evalúa una condición booleana y ejecuta un bloque de código si esa condición es verdadera. Además, puedes utilizar operadores lógicos y de comparación dentro de las condiciones para hacer evaluaciones más complejas.

Estructura básica de un condicional

```
if (condición) {
    /* Bloque de código ejecutado si la condición es verdadera */
}
```

8. Operadores Comparativos

Dentro de los condicionales, puedes realizar comparaciones entre valores numéricos (tipo *integer*) o valores booleanos (tipo *boolean*). Estos operadores permiten que el bloque de código se ejecute dependiendo del resultado de la comparación.

- **==**: Igualdad.
- **!=**: Desigualdad.
- **<**: Menor que.
- **<=**: Menor o igual que.
- **>**: Mayor que.
- **>=**: Mayor o igual que.

Ejemplo con operaciones comparativas

```
if (f1->start < 5) {
    f1->start += 2; /* Si el valor de 'start' en el bosque f1 es menor que 5, lo
                       incrementamos en 2 */
}
```

9. Acceso y uso de atributos de componentes

El lenguaje permite acceder a los valores de los atributos de los componentes (*tree*, *forest*, *world*, etc.) para reutilizarlos en otras definiciones o modificarlos directamente. Esto incluye:

1. **Asignar valores basados en atributos existentes** al crear nuevos componentes.
2. **Modificar los valores de los atributos** de componentes previamente definidos.
3. **Utilizar atributos como parte de expresiones matemáticas o lógicas** en condicionales o cálculos dinámicos.

1. Uso de atributos en nuevas definiciones

Es posible acceder a los atributos de un componente existente para definir o calcular valores en nuevos componentes. Esto permite establecer relaciones dinámicas entre los elementos del escenario.

Ejemplo:

```
tree t2 with (
    x = world->width * (3/4), /* Posición horizontal como 3/4 del ancho del
                                 mundo */
    height = world->height/2  /* Altura como la mitad de la altura del mundo */
);
```

2. Modificación de atributos de componentes existentes

Puedes modificar los valores de los atributos de un componente ya definido, permitiendo actualizaciones dinámicas según las necesidades del escenario.

Ejemplo:

```
t1->snowed = FALSE; // Cambia el atributo "snowed" de t1 a falso
```

3. Acceso a atributos en condicionales y operaciones

Los atributos pueden ser utilizados en:

- **Condicionales**, para evaluar y tomar decisiones.
- **Operaciones matemáticas**, para realizar cálculos dinámicos.
- **Asignaciones**, para reutilizar valores en otros componentes.

Ejemplo en un condicional:

```
if (t1->height > world->height / 2) {
    t1->bark = 3; /* Si el árbol es más alto que la mitad de la altura del
                     mundo, aumenta el grosor del tronco */
}
```

10. Creación de elementos en el mundo

El comando ***grow*** es la instrucción utilizada para materializar en el mundo los elementos definidos, ya sean bosques (*forest*) o árboles individuales (*tree*). Este comando toma los

componentes previamente configurados con sus atributos y los inserta en el escenario, respetando sus características y posiciones definidas.

Sintaxis y uso básico del comando grow:

```
grow(f1); /* Inserta el bosque f1 en el mundo */

grow(t1); /* Inserta el árbol t1 en el mundo */
```

5. Implementación

5.1. Frontend

En esta sección, se detalla el desarrollo del frontend del compilador, implementado con Flex y Bison. El frontend tiene como objetivo realizar el análisis léxico y sintáctico del código fuente escrito en el lenguaje de dominio específico (DSL) diseñado para la creación y manipulación de mundos, bosques y árboles.

El **analizador léxico** se encarga de dividir el código fuente en tokens, es decir, unidades léxicas significativas para el compilador. Para esto, se creó el archivo ***FlexPatterns.l***, que contiene las definiciones de patrones como palabras reservadas, operadores y símbolos específicos del lenguaje (como ***tree***, ***forest***, ***world***, ***grow***, etc.). Además, el archivo ***FlexActions.c*** define las funciones correspondientes a las acciones asociadas con cada token reconocido, tales como registrar el token, ignorar ciertos caracteres o devolver el token al analizador sintáctico.

Por otro lado, el archivo ***BisonGrammar.y*** define la gramática del lenguaje. Esta gramática, utilizada por Bison, permite construir un analizador sintáctico que convierte el código fuente en una representación estructurada llamada Árbol de Sintaxis Abstracta (AST). La gramática diferencia entre tokens **terminales** (palabras clave, operadores, etc.) y **no terminales** (estructuras complejas como definiciones de árboles, bosques o mundos). Las reglas que describen las construcciones válidas del lenguaje se especifican en este archivo y están asociadas a acciones que se ejecutan cuando se reconocen dichas reglas. Estas acciones se implementan en el archivo ***BisonActions.c***.

Por ejemplo:

- Un token puede representar alguna de las palabras clave como ***tree*** o ***grow***, mientras que una construcción no terminal puede definir la sintaxis completa para declarar un árbol o agregarlo a un bosque.
- Una acción asociada a una regla puede llevar a la creación de nodos en el AST para poder representar árboles, bosques y/o configuraciones de mundos.

A continuación, se muestran algunas de las estructuras de datos utilizadas, fundamentales para organizar y manipular las entidades del lenguaje:

- **Componentes:**

  Estas estructuras representan los elementos fundamentales en el sistema, como árboles, bosques y mundos. Cada componente tiene atributos específicos que definen su comportamiento y características, como tamaño, color, forma, posición y otros parámetros físicos o lógicos.

  Ejemplo:

```
struct _TREE{
   int x;
   int height;
   char leaf;
   Hexcolor color;
   int depth;
   int density;
   int bark;
   boolean snowed
};
```

- **Expression:**

  Las expresiones son estructuras que representan diferentes tipos de operaciones o acciones dentro del lenguaje. Pueden ser asignaciones, condiciones, operaciones aritméticas, o referencias a componentes como árboles o bosques. Las expresiones son evaluadas durante la ejecución para determinar el estado del programa.

  Ejemplo:

```
struct ConditionalExpression{
   ConditionalClause * conditionalClause;
   MainExpressions * ifMainExpressions;
   MainExpressions * elseMainExpressions;
   ConditionalType type;
};
```

- **Clause**

  Las cláusulas son unidades dentro de las expresiones condicionales. Definen las condiciones a evaluar, como comparaciones entre valores o la relación entre componentes, y pueden combinarse con otras cláusulas para formar condiciones más complejas.

Ejemplo:

```
struct ConditionalClause{
    union{
        struct{
            DeclarationValue * leftValueDeclare;
            ConditionalClause * rightConditionalClause;
        };
        struct{
            DeclarationValue * rightValueDeclare;
            ConditionalClause * leftConditionalClause;
        };
        struct{
            DeclarationValue * leftValue;
            DeclarationValue * rightValue;
        };
        struct{
            ConditionalClause * leftConditional;
            ConditionalClause * rightConditional;
        };
        ConditionalClause * conditionalClause;
    };
  ComparissonType comparissonType;
ConditionalClauseType conditionalType;
};
```

- **Assignment**

Las asignaciones representan la acción de asignar valores a variables o atributos de componentes. Pueden involucrar simples valores, operaciones aritméticas o incluso la modificación de componentes completos (árboles, bosques o mundos) según las reglas definidas.

Ejemplo:

```
struct ForestAssignment{
  _ID * id;
  union{
    DeclarationValue * value;
    ArithmeticOperation * arithmeticOperation;
  };
  AssignationType type;
};
```

- **DeclarationValue:**

Esta estructura es una de las más importantes porque permite representar valores individuales o atributos de diversas entidades del lenguaje, como árboles, bosques o el mundo. Es utilizada en asignaciones, comparaciones y operaciones aritméticas.

```
struct DeclarationValue{
  union{
    _ID * idValue;
    _STRING * charValue;
    _BOOLEAN * booleanValue;
    _HEXCOLOR * hexcolorValue;
    _INTEGER * intValue;
    AttributeValue * attValue;
    DeclarationValue * declareValue;
  };
  DeclarationValueType type;
};
```

- **MainExpression:**

  Esta estructura representa cualquier expresión que puede aparecer en el programa. Es la base para procesar las operaciones principales del lenguaje.

```
struct MainExpression {
  union{
    TreeExpression * treeExpression;
    ForestExpression * forestExpression;
    GrowExpression * growExpression;
    ForExpression * forExpression;
    ArithmeticAssignation * arithmeticAssignation;
    GeneralAssignation * generalAssignation;
    ConditionalExpression * conditionalExpression;
  };
  MainExpressionType type;
};
```

- **Program:**

  La estructura del programa y su expresión es el contenedor principal que organiza todas las expresiones, asignaciones y declaraciones dentro de un programa completo. Puede incluir un mundo o ser "sin mundo" (*worldless*), y gestiona cómo se ejecutan y relacionan las distintas expresiones.

```
struct ProgramExpression {
  union {
    MainExpressions * worldlessMainExpressions;
    struct {
      WorldExpression * worldExpression;
      MainExpressions * mainExpressions;
    };
  };
  ProgramType type;
};

struct Program {
  ProgramExpression * programExpression;
};
```

5.2. Backend

❖ *Table.c*:

El componente ***Table.c*** implementa una tabla de símbolos diseñada para almacenar y gestionar constantes definidas de manera eficiente. Para ello, utiliza un hashmap basado en la librería ***klib***, específicamente en la implementación definida en ***khash.h***. Este hashmap asocia cadenas de caracteres (***char * identifier***) con entradas del tipo ***Entry***, las cuales contienen un puntero al valor original y el tipo de dato correspondiente. Esta estructura permite realizar búsquedas rápidas y eficientes, así como una gestión clara de los diferentes tipos de valores almacenados.

Cada entrada en la tabla está definida por la estructura ***Entry***, que combina un valor de tipo ***EntryValue*** y un tipo de dato especificado por ***EntryType***. El campo ***EntryValue*** es una unión que puede contener un puntero a cualquiera de los tipos soportados: enteros, booleanos, cadenas, colores en formato hexadecimal, o estructuras complejas como árboles, bosques y mundos. El tipo de dato, representado por ***EntryType***, permite distinguir entre los distintos tipos de valores almacenados y asegura que solo se realicen operaciones válidas sobre ellos. Además, el sistema no permite duplicados en las claves, es decir, no se pueden insertar dos entradas con el mismo identificador, incluso si son de diferentes tipos.

La inicialización de la tabla se realiza mediante la función ***initializeTable***(), que configura el hashmap y asocia un logger para registrar eventos relevantes durante el ciclo de vida del componente. Por otro lado, ***destroyTable***() libera los recursos asociados al hashmap y destruye el logger, asegurando que no queden recursos sin liberar al finalizar su uso.

Para insertar valores en la tabla, se utiliza la función genérica ***insert***(), que recibe un identificador, un tipo y un valor, almacenándolos en la estructura interna del hashmap. Adicionalmente, se proporcionan funciones específicas como ***insertInteger***() o ***insertBoolean***() que encapsulan ***insert***() para manejar diferentes tipos de valores de manera más conveniente. Estas funciones aseguran que los valores se almacenen correctamente junto con su tipo correspondiente.

En cuanto a la recuperación de datos, el componente ofrece la función genérica ***getEntry***() que busca un identificador y verifica si el tipo asociado coincide con el solicitado. Si la búsqueda es exitosa, retorna un resultado que contiene tanto el valor como un indicador de éxito. Para facilitar el acceso a tipos específicos, se han definido funciones como ***getInteger***() o ***getBoolean***(), que simplifican la recuperación de valores de un tipo determinado. Por su parte, ***getType***() permite obtener únicamente el tipo asociado a un identificador dado, devolviendo **EMPTY_TYPE** si el identificador no existe.

La función ***exists***() facilita la verificación de la existencia de un identificador en la tabla, devolviendo un valor booleano según corresponda. Esta funcionalidad resulta útil para prevenir intentos de acceso a identificadores no definidos.

❖ *Generator.c*:

El componente **Generator.c** es responsable de producir la salida final del programa a partir de las estructuras jerárquicas creadas y modificadas durante la ejecución. Su propósito principal es interpretar expresiones definidas por el usuario y

generar un archivo de texto que represente los datos mediante configuraciones de mundo, árboles y bosques. Para ello, utiliza valores predeterminados como dimensiones del mundo y atributos básicos de árboles y bosques, que pueden ser ajustados por las definiciones del usuario.

El módulo inicializa su estado por medio del uso de la función ***initializeGeneratorModule***(), que configura un logger y abre un archivo para escribir la salida generada. Las estructuras principales que están involucradas para la generación del archivo son **_TREE**, **_FOREST** y **_WORLD**, utilizadas para modelar entidades como árboles con altura, densidad y color, o bosques con rangos específicos. La función ***generate***() actúa como el punto de partida para el proceso, delegando la ejecución a ***_generateProgram***(), que recorre de manera recursiva las expresiones del programa y utiliza algunas funciones específicas como por ejemplo ***_generateTreeExpression***() o ***_generateForestExpression***() para procesar árboles y bosques.

La generación de la salida se realiza mediante la función ***_output***(), que organiza el contenido con una estructura clara y aplicando formatos adecuados. Además, se incluye un manejo de errores a través del estado global **ERROR_OCCURED**, que detiene el procesamiento en caso de fallas y registra mensajes útiles para la depuración. Esto garantiza que la salida sea consistente y no contenga resultados parciales.

5.3. Dificultades Encontradas

- ***Orden de generación:***

A la hora de ir recorriendo el Abstract Syntax Tree, hay expresiones las cuales pueden ser únicas, o una concatenación de múltiples ocurrencias. Un ejemplo de esto son las ***MainExpressions***. Un problema que surgió fue que si se generaba primero la ***mainExpression*** y luego ***mainExpressions***, al ser ***mainExpression*** la última expresión en orden, ciertos test fallaban, indicando que no encontraban una variable claramente definida anteriormente. Esto se debe a que primero procesaba, por ejemplo: ***a* = 5;** antes que ***int a* = 2;**. Una vez cambiado el orden donde se presentaba esta circunstancia los errores de esta índole terminaron.

- ***Errores de Front:***

Se ve que durante la primera entrega, se pasaron por alto algunas asignaciones y errores de tipeo que, si bien en la primera entrega funcionaba bien, al añadir el back en la ecuación saltaron a la vista. Un ejemplo de estos fue un error de tipeo en el cual en vez de asignar ***variable->atributo*** correspondientemente, primero asignaba la variable en la variable y luego volvía a asignar en variable pero el atributo. Otro ejemplo sería la creación de un nodo sin asignarle su tipo. Igualmente que en el problema anterior, solo basto con un buen manejo de logs para identificar estos errores.

- ***Clases de datos:***

Durante el desarrollo del back surgió la necesidad de elaborar una tabla de símbolos, y con ello tener clases de datos. Tuvimos que modificar un poco el flex y bison de la primera entrega. Se añadió el token ***type*** para identificar correctamente los tipos necesarios para nuestro proyecto.

- ***Flujo de generación:***

  Nuestra problemática principal fue plantear cómo procesar la entrada del *front* al *back*. Nos tomó demasiado tiempo tener una tabla de símbolos y una estructura de validación general. A esto se le sumó que recién al incorporar ambas entre sí pudimos empezar a identificar los errores mencionados anteriormente. Una vez estuvo incorporado el proceso de generación fue bastante ameno, pero el tiempo perdido en entender esa lógica fue realmente problemático en nuestra organización.

6. Futuras Extensiones

El compilador fue diseñado con la suficiente flexibilidad para poder permitirnos incorporar nuevas funcionalidades y poder escalar según las necesidades del usuario a futuro. Debajo de éste párrafo, se encuentran distintas sugerencias con respecto a varios cambios y extensiones que podrían hacer del lenguaje uno más completo y con una mejor experiencia de usuario.

Ampliación de Tipos de Árboles Default o Predeterminados

Por el momento, nuestra implementación permite la creación de árboles personalizados con atributos específicos, pero se podría ampliar la selección de especies predeterminadas, añadiendo nuevos tipos como bonsáis, pinos o árboles con flores. Esto le daría la posibilidad a los usuarios de seleccionar rápidamente entre una diversidad de tipos de estructuras con características ya predefinidas, haciendo del proceso de creación, uno más ágil y accesible.

Implementación de Ciclos y Repeticiones para Componentes

Se podría incluir la posibilidad de crear bosques o escenarios completos por medio del uso de bucles, permitiendo a los usuarios generar múltiples instancias de árboles con variaciones específicas (por ejemplo, diferentes tamaños, alturas, colores) a partir de un solo modelo. Esto podría ser útil para escenarios grandes o cuando se requiere el uso de muchos elementos similares, ahorrando tiempo y reduciendo la repetición de código.

Cobertura del Cielo y Condiciones Climáticas

Otra extensión a tener en cuenta a futuro sería incorporar una sección de clima y cielo, permitiendo la configuración de condiciones meteorológicas como lluvia, nieve, sol, o nubes que afecten la apariencia de los árboles o del mundo en general. Asimismo, se podría añadir la capacidad de definir ciclos de día y noche, con variaciones en la iluminación o el color del cielo, lo que permitiría crear escenarios/paisajes mucho más detallados y realistas.

Personalización de la Topografía del Mundo

En lugar de limitarse a mundos rectangulares o planos, podría ser interesante permitirle a los usuarios personalizar la topografía de su mundo, creando terrenos irregulares, montañas, valles, lagos o ríos. Esto podría lograrse mediante una mayor parametrización en la sección de world, añadiendo atributos para definir el relieve o el tipo de superficie del mundo.

7. Conclusiones

En este informe se detalló el desarrollo de un compilador para la creación y simulación de árboles y bosques dentro de un mundo tridimensional mediante un lenguaje de dominio específico (DSL). El objetivo principal fue permitir a los usuarios definir y modificar atributos de árboles, bosques y el mundo, y generar representaciones visuales personalizadas que se ajusten a sus necesidades, ya sea en términos de características de los árboles (como altura, color, o tipo de hoja) o en la disposición del mundo y su clima.

Durante el desarrollo, se presentaron desafíos técnicos, tales como la creación de una estructura flexible que permitiera modificar dinámicamente los atributos de los árboles, bosques y el mundo, así como la gestión de condiciones complejas como el clima y el paso del tiempo. Además, la integración de funcionalidades como la iteración sobre árboles o el uso de condicionales dentro del lenguaje para modificar valores de los atributos generó dificultades adicionales. Sin embargo, el grupo pudo superar estos obstáculos, y, a pesar de las complicaciones relacionadas a los tiempos de entrega, el compilador ahora es capaz de generar escenarios completos a partir de las especificaciones dadas en el lenguaje.

Este proyecto definitivamente nos dió la posibilidad de aplicar los conceptos aprendidos a lo largo del cuatrimestre en la materia, sobre todo aquello vinculado con el desarrollo de un compilador propio.

8. Apéndices

8.1. Programa de salida exitoso

Dentro del proyecto, en la sección de aceptación de tests, hay uno titulado ***000-generalTest***. Este es usado a modo de prueba de una salida exitosa. Para testearlo en la terminal escribimos:

script/ubuntu/start.sh src/test/c/accept/000-generalTest

Esto generará en la carpeta general del proyecto un archivo **treegered.html** que lucirá de la siguiente forma:

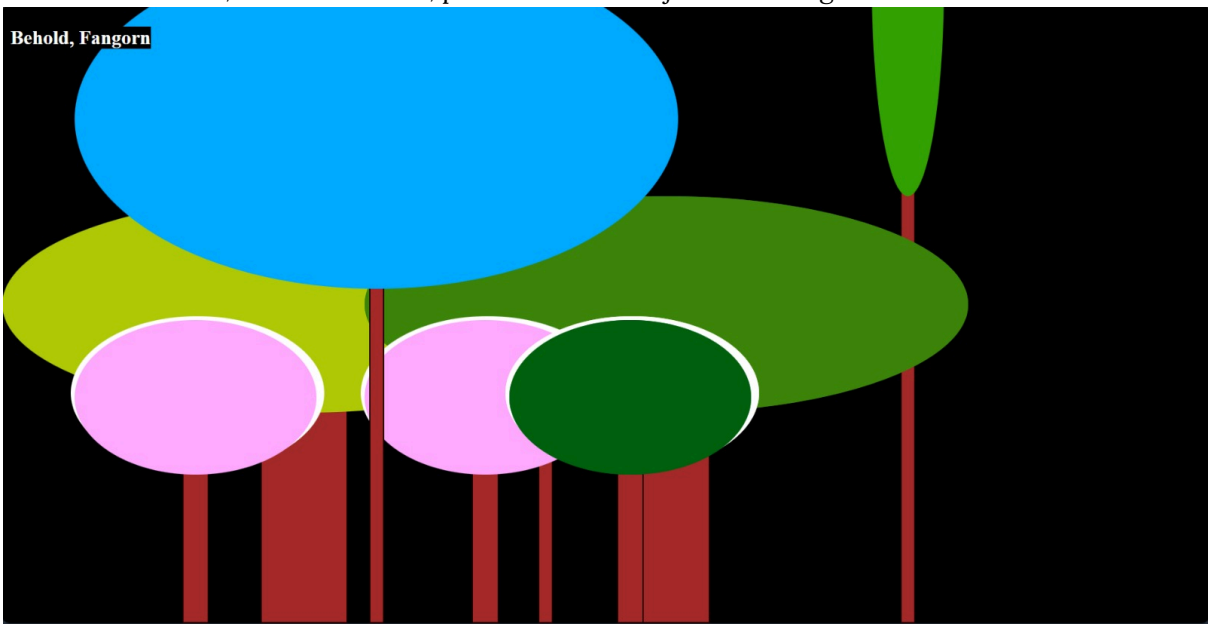
```
<!DOCTYPE html>
<html>
<head>
<title>Tree Visualization</title>
<style>
body {
  margin: 0;
  overflow: hidden;
```

```
}
.tree {
  position: absolute;
  display: flex;
  flex-direction: column;
  align-items: center;
}
.trunk {
 background-color: brown; border: 2px solid #1B0000;
}
.leaves {
  border-radius: 50%;
  text-align: center;
  color: white;
}
</style>
<script>
window.onload = function() {
  document.body.style.height = window.innerHeight + 'px';
  document.body.style.width = window.innerWidth + 'px';
  const worldHeight = 100;
  const worldWidth = 100;
  const scaleHeight = window.innerHeight / worldHeight;
  const scaleWidth = window.innerWidth / worldWidth;
  document.querySelectorAll('.tree').forEach(tree => {
    tree.style.left = (parseFloat(tree.style.left) * scaleWidth) + 'px';
      tree.style.bottom = (parseFloat(tree.style.bottom) * scaleHeight) +
'px';
  });
  document.querySelectorAll('.leaves, .trunk').forEach(part => {
    part.style.width = (parseFloat(part.style.width) * scaleWidth) + 'px';
      part.style.height = (parseFloat(part.style.height) * scaleHeight) +
'px';
  });
};
</script>
</head>
<body style='background-color: black'>
<h1    style='position:absolute;    top:10px;    left:10px;    color:white;
background-color:black; z-index:99999' >Behold, Fangorn</h1>
<div class='tree' style='left: 30%; bottom: 0;'>
```

```
<div class='leaves' style='width: 21px; height: 25px; background-color:
white; z-index:9; position:absolute; top: 5px; left:-5px'></div>
<div class='leaves' style='width: 20px; height: 25px; background-color:
#FFAAFF; z-index:10; position:relative; top:10px'></div>
<div class='trunk' style='height: 25px; width: 2px; z-index:9;'></div>
</div>
<div class='tree' style='left: 6%; bottom: 0;'>
<div class='leaves' style='width: 21px; height: 25px; background-color:
white; z-index:9; position:absolute; top: 5px; left:-5px'></div>
<div class='leaves' style='width: 20px; height: 25px; background-color:
#FFAAFF; z-index:10; position:relative; top:10px'></div>
<div class='trunk' style='height: 25px; width: 2px; z-index:9;'></div>
</div>
<div class='tree' style='left: 0%; bottom: 0;'>
<div class='leaves' style='width: 50px; height: 35px; background-color:
#B0CA07; z-index:8; position:relative; top:10px'></div>
<div class='trunk' style='height: 35px; width: 7px; z-index:7;'></div>
</div>
<div class='tree' style='left: 6%; bottom: 0;'>
<div class='leaves' style='width: 50px; height: 55px; background-color:
#00AAFF; z-index:14; position:relative; top:10px'></div>
<div class='trunk' style='height: 55px; width: 1px; z-index:13;'></div>
</div>
<div class='tree' style='left: 42%; bottom: 0;'>
<div class='leaves' style='width: 21px; height: 25px; background-color:
white; z-index:15; position:absolute; top: 5px; left:-5px'></div>
<div class='leaves' style='width: 20px; height: 25px; background-color:
#00610E; z-index:16; position:relative; top:10px'></div>
<div class='trunk' style='height: 25px; width: 2px; z-index:15;'></div>
</div>
<div class='tree' style='left: 30%; bottom: 0;'>
<div class='leaves' style='width: 50px; height: 35px; background-color:
#3D860B; z-index:8; position:relative; top:10px'></div>
<div class='trunk' style='height: 35px; width: 7px; z-index:7;'></div>
</div>
<div class='tree' style='left: 42%; bottom: 0;'>
<div class='leaves' style='width: 6px; height: 70px; background-color:
#34A203; z-index:4; position:relative; top:10px'></div>
<div class='trunk' style='height: 70px; width: 1px; z-index:3;'></div>
</div>
<div class='tree' style='left: 42%; bottom: 0;'>
```

```
<div class='leaves' style='width: 21px; height: 25px; background-color:
white; z-index:15; position:absolute; top: 5px; left:-5px'></div>
<div class='leaves' style='width: 20px; height: 25px; background-color:
#00610E; z-index:16; position:relative; top:10px'></div>
<div class='trunk' style='height: 25px; width: 2px; z-index:15;'></div>
</div>
<div class='tree' style='left: 30%; bottom: 0;'>
<div class='leaves' style='width: 50px; height: 35px; background-color:
#3D860B; z-index:8; position:relative; top:10px'></div>
<div class='trunk' style='height: 35px; width: 7px; z-index:7;'></div>
</div>
<div class='tree' style='left: 42%; bottom: 0;'>
<div class='leaves' style='width: 6px; height: 70px; background-color:
#34A203; z-index:4; position:relative; top:10px'></div>
<div class='trunk' style='height: 70px; width: 1px; z-index:3;'></div>
</div>
<div class='tree' style='left: 72%; bottom: 0;'>
<div class='leaves' style='width: 6px; height: 70px; background-color:
#34A203; z-index:4; position:relative; top:10px'></div>
<div class='trunk' style='height: 70px; width: 1px; z-index:3;'></div>
</div>
</body>
</html>
```

Y ese html, una vez abierto, producirá un dibujo como el siguiente:



8.1. Programa malformado

Dentro del proyecto, en la sección de rechazo de tests, hay uno titulado ***001-emptyProgram***. Este es usado a modo de prueba de una salida exitosa. Para testearlo en la terminal escribimos

script/ubuntu/start.sh src/test/c/reject/001-emptyProgram

Esto generará la siguiente salida en terminal

[ERROR][SyntacticAnalyzer] Syntax error (on line 1).

[ERROR][EntryPoint] The syntactic-analysis phase rejects the input program.

y el archivo estará **treegered.html** vacío.

9. Referencias

1. khash.h (librería utilizada para la implementación de la tabla hash) https://github.com/attractivechaos/klib/blob/master/khash.h

10. Bibliografía

1. Material y apuntes teórico-prácticos otorgados por la cátedra de la materia Autómatas, Teoría de Lenguajes y Compiladores en el segundo cuatrimestre de 2024.