Instituto Tecnológico de Buenos Aires

Trabajo Práctico: GANTT

72.39 - Autómatas, Teoría de Lenguajes y Compiladores

Integrantes

Bloise, Luca - 63004

Tepedino, Cristian - 62830

Madero Torres, Eduardo Federico - 59494

Mendonca, Juana - 61185

Primer Cuatrimestre del 2024

Índice

1. Introducción

El proyecto presentado en este informe se enmarca dentro de la asignatura "Autómatas, Teoría de Lenguajes y Compiladores"de la carrera de Ingeniería Informática en ITBA. El objetivo principal es el diseño e implementación de un lenguaje específico de dominio para la creación y graficación de diagramas de Gantt, una herramienta fundamental en la gestión de proyectos.

El desarrollo de este proyecto abarca desde la especificación inicial del lenguaje hasta la implementación de un compilador completo, incluyendo pruebas de aceptación y rechazo para validar su correcto funcionamiento. La documentación y el código fuente se gestionarán y versionarán en el repositorio TP-GANTT.

Este informe detallará el modelo computacional, la implementación del lenguaje, las dificultades encontradas, y las posibles extensiones futuras, proporcionando una visión integral del proceso y los resultados obtenidos.

2. Modelo Computacional

2.1. Dominio

Desarrollar un lenguaje que permita crear y graficar diagramas de Gantt. El lenguaje debe permitir crear proyectos y definir su estructura, donde se pueden agregar tareas especificando su duración o inicio y fin, sus dependencias y asignarles un puntaje. Además, debe permitir especificar máxima cantidad de tareas o puntos que se puedan realizar en simultáneo. Debe permitir clasificar las distintas tareas en categorías definidas en la estructura del proyecto.

Como resultado, se debe poder obtener un programa en python que utiliza Plotly para ilustrar los diagramas de Gantt. La implementación satisfactoria de este lenguaje permitiría evidenciar diagramas de Gantt con la organización más eficiente de las tareas, según los requerimientos del proyecto.

2.2. Lenguaje

El lenguaje desarrollado ofrece las siguientes construcciones, prestaciones y funcionalidades:

1. Se podrá crear un **project** con su nombre y tipo de formato (con duración o inicio y fin de tarea).
2. El formato del proyecto podrá ser del tipo **hour**, **day**, **week**, **month** o **date** (se especifica inicio y fin de cada tarea).
3. Se podrán crear una o varias **task** dentro de cada **project** y cada una tendrá un nombre y características propias.
4. Se podrá especificar las **categorías** disponibles para cada **task** en la estructura del **project**.
5. Se podrá especificar la **máxima cantidad de tasks** que pueden ocurrir en simultáneo.
6. Se podrá especificar la **fecha de inicio** del **project**.
7. Se podrán agregar las **dependency** entre **tasks**, (qué tareas se deben finalizar para comenzar).
8. Se le podrán asignar **points** a cada **task**.
9. Se podrá especificar la **máxima cantidad de points** que pueden ocurrir en simultáneo.

10. En caso de formato **date** se deberá ingresar **start** y **finish** de cada **task**. En caso contrario el **length** representado como un intervalo.

11. En caso de definir **categories** con sus respectivos id, se puede asignar una **category** a cada **task** mediante el id de la misma.

12. Se podrá definir una **task** como **unique** para indicar que no se pueden realizar otras tareas en simultáneo.

2.3. Ejemplos de aceptación

A continuación se presentan diferentes programas de prueba que deberían resultar exitosos a la hora de realizarse la traducción en el compilador.

- **Proyecto 1:** Crear un nuevo proyecto con formato hora y una tarea.

```
project 1 "Proyecto1" format hour {
    task 1 "Tarea1" length [3-5]
}
```

- **Proyecto 2:** Crear un nuevo proyecto con formato fecha y una tarea.

```
project 2 "Proyecto2" format date {
    task 1 "Tarea2" start 2024-04-01 finish 2024-04-05
}
```

- **Proyecto 3:** Crear un nuevo proyecto con formato semana y múltiples tareas.

```
project 3 "Proyecto3" format week {
    task 1 "Tarea3" length [2-4]
    task 2 "Tarea4" length [3-6]
    task 3 "Tarea5" length [1-2]
}
```

- **Proyecto 4:** Crear un nuevo proyecto con formato día, categorías específicas y múltiples tareas.

```
project 4 "Proyecto4" format day {
    categories 1 "cat", 2 "catb", 3 "catc"
    task 1 "Tarea6" length [1-3] category 1
    task 2 "Tarea7" length [2-4] category 2
}
```

- **Proyecto 5:** Crear un nuevo proyecto con formato mes, tareas y dependencias.

```
project 5 "Proyecto5" format month {
task 1 "Tarea8" length [2-3]
task 2 "Tarea9" length [3-5] depends_on 5.1
}
```

- **Proyecto 6:** Crear un nuevo proyecto con formato día, tareas y máximo de tareas a realizar simultáneamente.

```
project 6 "Proyecto6" format day {
    max_points 10
    task 1 "Tarea1" length [2-4] points 3
    task 2 "Tarea2" length [3-5] points 4
    task 3 "Tarea3" length [1-2] points 2
    task 4 "Tarea4" length [4-6] points 5
}
```

- **Proyecto 7:** Crear un nuevo proyecto con formato mes, tareas y máximo de tareas a realizar simultáneamente.

```
project 7 "Proyecto7" format month {
    task 1 "Tarea1" length [2-3]
    task 2 "Tarea2" length [3-4] unique
    task 3 "Tarea3" length [1-2]
}
```

- **Proyecto 8:** Crear un nuevo proyecto con formato semana, tareas, duración y dependencias con la característica unique.

```
project 8 "ProyectoDependenciasUnica" format week {
    task 1 "Conf de la base de datos" length 2
    task 2 "Desarrollo del backend" length [3] depends_on
       8.1
    task 3 "Despliegue del servidor" length [2] depends_on
       8.2
    task 4 "Conf del frontend" length [2] unique
}
```

- **Proyecto 9:** Crear un nuevo proyecto complejo con múltiples tareas y características.

```
project 9 "ProyectoComplejo" format day {
    max_tasks 5
    project_start 2024-04-01
    categories 1 "Desarrollo", 2 "Pruebas", 3 "Implement"
    task 1 "Desarrollo de la interfaz" length [2] category
       1 points 1
    task 2 "Desarrollo del backend" length [3] category 2
       points 2
    task 3 "Conf del entorno de pruebas" length [1]
       category 2 points 2
    task 4 "Ejecucion de pruebas unitarias" length [2]
       category 2 points 4 depends_on 3
    task 5 "Implem de la funcionalidad principal" length
       [4] category 3 points 6 depends_on 9.2
    task 6 "Pruebas de integration" length [2] category 2
       points 4 depends_on 4, 5
}
```

- **Dependencia de tarea externa:** Crear un nuevo proyecto con una tarea que dependa de otra ya definida en otro proyecto.

```
project 10 "ProyectoA" format day {
    task 1 "TareaInicial" length [1-3]
}

project 11 "ProyectoB" format day {
    task 1 "TareaDependiente" length [2-4] depends_on 10.1
}
```

- **Dependencia entre proyectos:** Crear un nuevo proyecto con formato días y que dependa de otro proyecto ya definido.

```
project 12 "ProyectoPadre" format day {
    task 1 "TareaInicial" length [2-3]
    task 2 "TareaIntermedia" length [3-5] depends_on 12.1
    task 3 "TareaFinal" length [1-2] depends_on 12.1, 12.2
}

project 13 "ProyectoHijo" format day depends_on_project 12
   {
    task 1 "TareaHijo1" length [2-3]
    task 2 "TareaHijo2" length [3-5] depends_on 13.1
    task 3 "TareaHijo3" length [1-2] depends_on 13.2
}
```

- **Unión de proyectos:** Unir proyectos y tareas especificadas anteriormente.

```
project 14 "UnionDeProyectos" format day with 12, 13 { }
```

2.4. Ejemplos de rechazo

A continuación se presentan diferentes programas de prueba que deberían ser rechazados a la hora de realizarse la traducción en el compilador. Algunas reglas de nuestro lenguaje son las siguientes:

- Si se definen proyectos, tareas o categorías, todas deben tener un ID y un NOMBRE asociado.
- En el caso de utilizar el formato 'date', la fecha de comienzo de una tarea no debe ser posterior a la fecha de finalización de la misma.
- Si se definen categorías y se clasifican tareas, las categorías referidas mediante el ID deben estar previamente definidas.

- Si se declaran dependencias, deben existir tanto el proyecto (se especifica en el primer número antes del punto) como la tarea (después del primer punto).
- Si se definen puntos máximos a asignarse a una tarea, estos no deben ser mayores a 10.
- Si se definen dependencias entre proyectos, el proyecto debe existir.
- Si se realiza una unión de proyectos, entonces los proyectos unidos deben existir.

- **Proyecto 1:** Se crea un proyecto cuyo formato no es válido.

```
project a "Proyecto1" format years {
    task a "Tarea1" length [3-5]
}
```

- **Proyecto 2:** Proyecto con formato 'date' cuyo orden de construcción es inverso (debería comenzar indicando start primero, y luego finish).

```
 project a "Proyecto2" format date {
    task a "Tarea2" finish 2024-04-01 start 2024-04-05
}
```

- **Proyecto 3:** Proyecto con formato 'date' cuya fecha de comienzo es posterior a la de finalización.

```
 project a "Proyecto2" format date {
    task a "Tarea2" start 2028-04-01 finish 2024-04-05
}
```

- **Proyecto 4:** Proyecto sin nada definido.

```
EMPTY
```

- **Proyecto 5:** Proyecto definido pero con los atributos de las tareas en el orden incorrecto.

```
project a "Proyecto3" format day {
    max_points 10
    task a "Tarea1" length [2-4] points 3
    task b "Tarea2" length [3-5] points 4
    task c "Tarea3" length [1-2] points 2
    task d "Tarea4" points 5 length [4-6]
}
```

- **Proyecto 6:** Proyecto con tareas dependientes no definidas previamente.

```
project a "Proyecto4" format month {
    task a "Tarea8" length [2-3]
    task b "Tarea9" length [3-5] depends_on a.c
}
```

3. Implementación

Los compiladores e intérpretes son programas que, a través de diferentes fases, analizan un código de entrada con el principal objetivo de generar un código de salida. Estas fases a menudo requieren determinar el significado de los identificadores, incluyendo su tipo y atributos asociados.

En general, se pueden identificar dos fases principales:

Fase de Análisis: En esta fase, partiendo del código de entrada, se emplean los siguientes analizadores:

- **Analizador Léxico:** Se encarga de leer el código fuente y convertirlo en una serie de tokens, que son unidades léxicas que representan las palabras clave, identificadores, literales y otros elementos sintácticos.
- **Analizador Sintáctico:** Utiliza los tokens generados por el analizador léxico para construir una estructura jerárquica llamada árbol de sintaxis, que representa la estructura gramatical del código fuente.
- **Analizador Semántico:** Verifica la corrección semántica del árbol de sintaxis, asegurándose de que las operaciones y las declaraciones en el código sean válidas y tengan sentido en el contexto del lenguaje de programación. También realiza la resolución de tipos y la vinculación de identificadores.

Fase de Síntesis: En esta fase intervienen los siguientes componentes para producir el código de salida:

- **Generador de Código:** Transforma el árbol de sintaxis anotado en un código intermedio o código máquina, que puede ser ejecutado por la computadora.
- **Optimizador de Código:** Mejora el código generado para hacerlo más eficiente en términos de tiempo de ejecución y uso de recursos, aplicando diversas técnicas de optimización.

En nuestro proyecto, hemos dividido el proceso en dos etapas: **frontend** y **backend**. En el frontend se describen el analizador léxico y el analizador sintáctico, mientras que en el backend se abordan la generación de código y el chequeo de tipos (type-checking) mediante el desarrollo de una tabla de símbolos.

3.1. Frontend

3.1.1. Analizador Léxico

En la etapa del frontend, se ha implementado en primer lugar el analizador léxico utilizando Flex, una vez conocido el lenguaje y su dominio correspondiente. Flex utiliza expresiones regulares que sirven para armar los tokens necesarios, permitiendo la tokenización mediante la lectura de estas expresiones, que harán match con el texto encontrado en los archivos de entrada.

Se han empleado pilas de estados para manejar contextos anidados como comentarios multilínea y cadenas de texto que representan los nombres de los projectos, cateogorías y tareas. Este enfoque es útil para reconocer diferentes componentes léxicos del lenguaje específico de dominio y definir qué acciones tomar cuando se encuentran estos componentes. Las acciones correspondientes crean los tokens que serán utilizados posteriormente por el analizador sintáctico.

Se han definido los siguientes identificadores como palabras reservadas dentro de nuestro lenguaje: `"project"`, `"format"`, `"date"`, `"day"`, `"week"`, `"month"`, `"task"`, `"length"`, `"start"`, `"finish"`, `çategories"`, `çategory"`, `"depends_on_project"`, `"depends_on"`, `"points"`, `ünique"`, `"max_tasks"`, `"project_start`, y `"with"`. Además, establecimos reglas específicas para reconocer fechas en el formato `YYYY-MM-DD`. Estas palabras clave y reglas permiten una estructura clara y organizada del lenguaje, facilitando el análisis y procesamiento del código fuente.

La implementación cuidadosa de estas reglas y palabras reservadas asegura que el analizador léxico pueda manejar adecuadamente los distintos elementos y sintaxis del lenguaje, proporcionando una base sólida para las siguientes etapas del compilador.

3.1.2. Analizador Sintáctico

Para poder generar nuestro analizador sintáctico, usamos BISON. Bison permite definir gramáticas libres de contexto, las cuales son utilizadas para analizar la estructura sintáctica del código fuente. Esto se logra a través de la especificación de terminales (tokens) y no terminales, así como las reglas de producción que describen cómo los terminales y no terminales se combinan para formar estructuras válidas en el lenguaje.

Definimos nuestra gramática, especificando tanto los terminales como los no terminales. Los terminales son las palabras clave y símbolos específicos de nuestro lenguaje, mientras que los no terminales representan las estructuras sintácticas compuestas por uno o más terminales.

A continuación se presenta la definición de nuestra gramática en forma de conjunto:

Terminales:

INTEGER, NAME, ID, LEFT_INTERVAL, RIGHT_INTERVAL, SPECIFIC_DATE, PROJECT, FORMAT, DATE, HOUR, DAY, WEEK, MONTH, OPEN_BRACES, CLOSE_BRACES, TASK, LENGTH, START, FINISH, CATEGORIES, CATEGORY, DEPENDS_ON, DOT, MAX_POINTS, POINTS, UNIQUE, MAX_TASKS, PROJECT_START, COMMA, DEPENDS_ON_PROJECT, WITH, UNKNOWN

No Terminales:

projectStructure, projectOptionals, projectUnion, projectBody, taskStructure, projectBodyOptionals, bodyCategoriesOption, taskList, taskLengthFormat, taskOptionals, taskOptionDependsOn, program, projectStructureCommon, timeUnit, categoryId, pointsInteger, dependsOnId, unique, projectStart, maxPoints, categoriesId, maxTasks, projectId, taskId

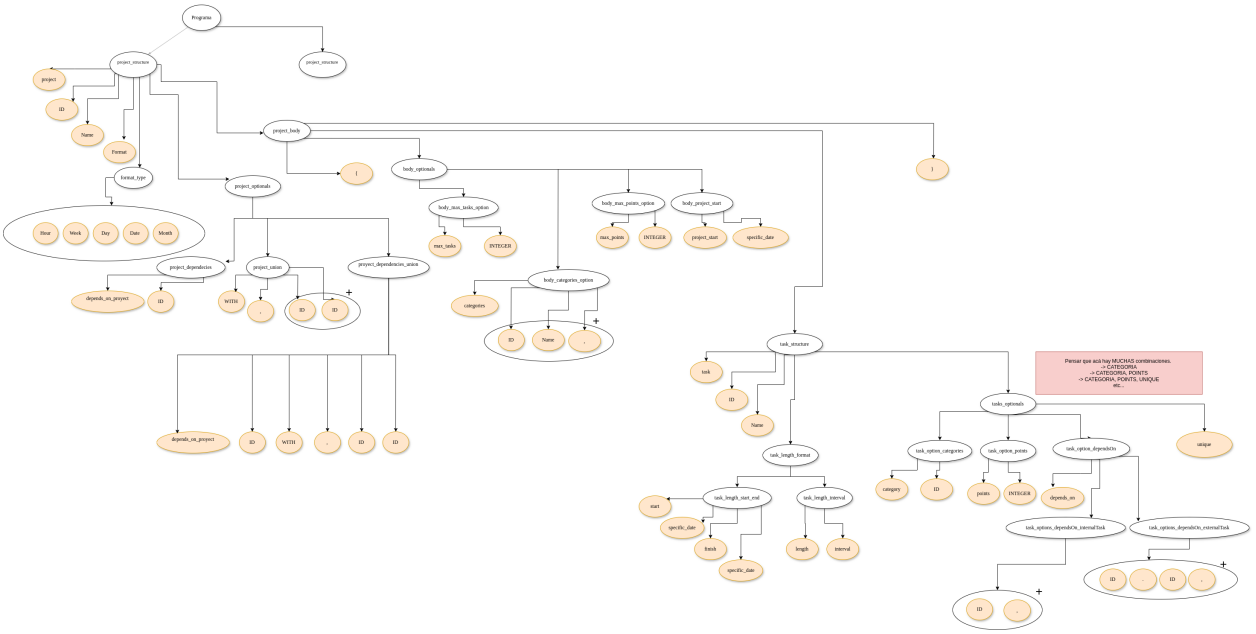

Figura 1: Árbol de sintaxis abstracta genérico.

(a) Para visualizar mejor el árbol pensado en la primera iteración (es decir, se han modificado algunos símbolos no terminales, pero ha sido útil para ayudarnos a definir nuestra gramática completa. Se puede acceder desde aquí.

Producciones:

$$
\begin{aligned}
\text{program} &\rightarrow \text{projectStructure} \\
\text{projectStructure} &\rightarrow \text{projectStructureCommon projectOptionals projectBody} \\
\text{projectStructureCommon} &\rightarrow \text{PROJECT projectId NAME FORMAT timeUnit} \\
\text{projectId} &\rightarrow \text{ID} \\
\text{timeUnit} &\rightarrow \text{HOUR} \\
&\quad |\text{DAY} \\
&\quad |\text{WEEK} \\
&\quad |\text{MONTH} \\
&\quad |\text{DATE} \\
\text{projectOptionals} &\rightarrow \epsilon \\
&\quad |\text{DEPENDS\_ON\_PROJECT ID} \\
&\quad |\text{WITH ID projectUnion} \\
&\quad |\text{DEPENDS\_ON\_PROJECT ID WITH ID projectUnion} \\
\text{projectUnion} &\rightarrow \epsilon \\
&\quad |\text{projectUnion COMMA ID} \\
\text{projectBody} &\rightarrow \text{OPEN\_BRACES projectBodyOptionals taskList CLOSE\_BRACES} \\
\text{taskList} &\rightarrow \text{taskStructure} \\
&\quad |\text{taskList taskStructure} \\
\text{taskStructure} &\rightarrow \text{TASK taskId NAME taskLengthFormat taskOptionals} \\
\text{taskId} &\rightarrow \text{ID} \\
\text{taskLengthFormat} &\rightarrow \text{START SPECIFIC\_DATE FINISH SPECIFIC\_DATE} \\
&\quad |\text{LENGTH LEFT\_INTERVAL RIGHT\_INTERVAL} \\
&\quad |\text{LENGTH LEFT\_INTERVAL} \\
\text{taskOptionals} &\rightarrow \text{categoryId pointsInteger dependsOnId unique} \\
\text{categoryId} &\rightarrow \epsilon \\
&\quad |\text{CATEGORY ID} \\
\text{pointsInteger} &\rightarrow \epsilon \\
&\quad |\text{POINTS INTEGER} \\
\text{dependsOnId} &\rightarrow \epsilon \\
&\quad |\text{DEPENDS\_ON ID DOT ID taskOptionDependsOn} \\
\text{unique} &\rightarrow \epsilon \\
&\quad |\text{UNIQUE}
\end{aligned}
$$

taskOptionDependsOn $\rightarrow \epsilon$

|taskOptionDependsOn COMMA ID DOT ID

projectBodyOptionals $\rightarrow$ maxTasks categoriesId maxPoints projectStart

maxTasks $\rightarrow \epsilon$

|MAX_TASKS INTEGER

categoriesId $\rightarrow \epsilon$

|CATEGORIES ID NAME bodyCategoriesOption

maxPoints $\rightarrow \epsilon$

|MAX_POINTS INTEGER

projectStart $\rightarrow \epsilon$

|PROJECT_START SPECIFIC_DATE

bodyCategoriesOption $\rightarrow \epsilon$

|bodyCategoriesOption COMMA ID NAME

3.2. Backend

En el backend del proyecto se ha implementado la generación de código, la construcción de una tabla de símbolos y un proceso verificación de tipos (typechecking) para nuestro lenguaje. Esta etapa es crucial, ya que permite detectar y prevenir errores semánticos en el código fuente, garantizando así la coherencia y la corrección del programa antes de su ejecución.

3.2.1. Tabla de símbolos

Las Tablas de Símbolos (TS) actúan como almacenes de información esencial sobre los identificadores presentes en el código fuente de un lenguaje de programación. Estas estructuras de datos desempeñan un papel fundamental en el proceso de traducción, ya que permiten realizar comprobaciones semánticas para asegurar la coherencia del código y proporcionan información clave para la generación eficiente de código ejecutable. La naturaleza de los datos almacenados en una TS varía según las características y elementos específicos del lenguaje en cuestión, pero generalmente incluye detalles sobre los distintos tipos de identificadores, como nombres de variables, funciones, clases y métodos, entre otros.

Durante el proceso de compilación, se deberán ejecutar las siguientes operaciones sobre la tabla de símbolos:

- **Búsqueda/Consulta**: Esta es la operación más frecuente y se utiliza para verificar si un símbolo ya existe, encontrar y modificar sus atributos, o eliminar un símbolo de la tabla. Las demás operaciones generalmente implican una búsqueda previa.
- **Actualización**: En compiladores de una sola pasada, el análisis léxico crea la tabla de símbolos e introduce los identificadores. Sin embargo, la mayoría de los atributos se asocian durante el análisis semántico, que debe localizar y actualizar cada símbolo.
- **Inserción**: A medida que se analiza el código fuente, se introducen en la tabla de símbolos las parejas (identificador, atributos). Esta tarea puede ser realizada por el analizador léxico o el semántico, dependiendo del lenguaje. En algunos lenguajes, puede haber varios símbolos con el mismo identificador pero con atributos diferentes.
- **Borrado**: Se produce cuando un símbolo queda fuera de ámbito y deja de existir.

La estructura que se ha elegido para la tabla de símbolos es una tabla HASH. Dado que cada proyecto, tarea y categoría tienen IDs únicos, estos funcionarán como claves en nuestra tabla. Los atributos asociados a cada uno de estos identificadores se almacenan en la tabla, lo que permite una gestión eficiente y rápida de las operaciones de búsqueda, actualización, inserción y borrado.

Para implementar esta tabla de símbolos, se ha optado por utilizar un archivo de cabecera que implementa hashing mediante tablas, definiendo qué estructuras se desean almacenar mediante hash. UTHASH es una herramienta que proporciona una tabla hash para estructuras, permitiendo la implementación eficiente de operaciones como adición, búsqueda, eliminación, conteo, iteración y ordenamiento de elementos utilizando claves únicas. Esta herramienta es especialmente útil en contextos donde se necesita rapidez y eficiencia, ya que estas operaciones son de tiempo constante en la mayoría de los casos.

UTHASH está implementado como un único archivo de cabecera (uthash.h), lo que facilita su integración en proyectos de C y C++. Para utilizar UTHASH, simplemente se ha incluido este archivo en el proyecto y se han declarado las estructuras con los campos necesarios. Las principales funcionalidades de UTHASH incluyen la adición y reemplazo de elementos, búsqueda eficiente mediante claves únicas, eliminación de elementos, conteo de elementos presentes en la tabla y la capacidad de iterar y ordenar elementos según criterios específicos.

Las funciones disponibles han sido utilizadas desde la generación de código y desde el análisis semántico.

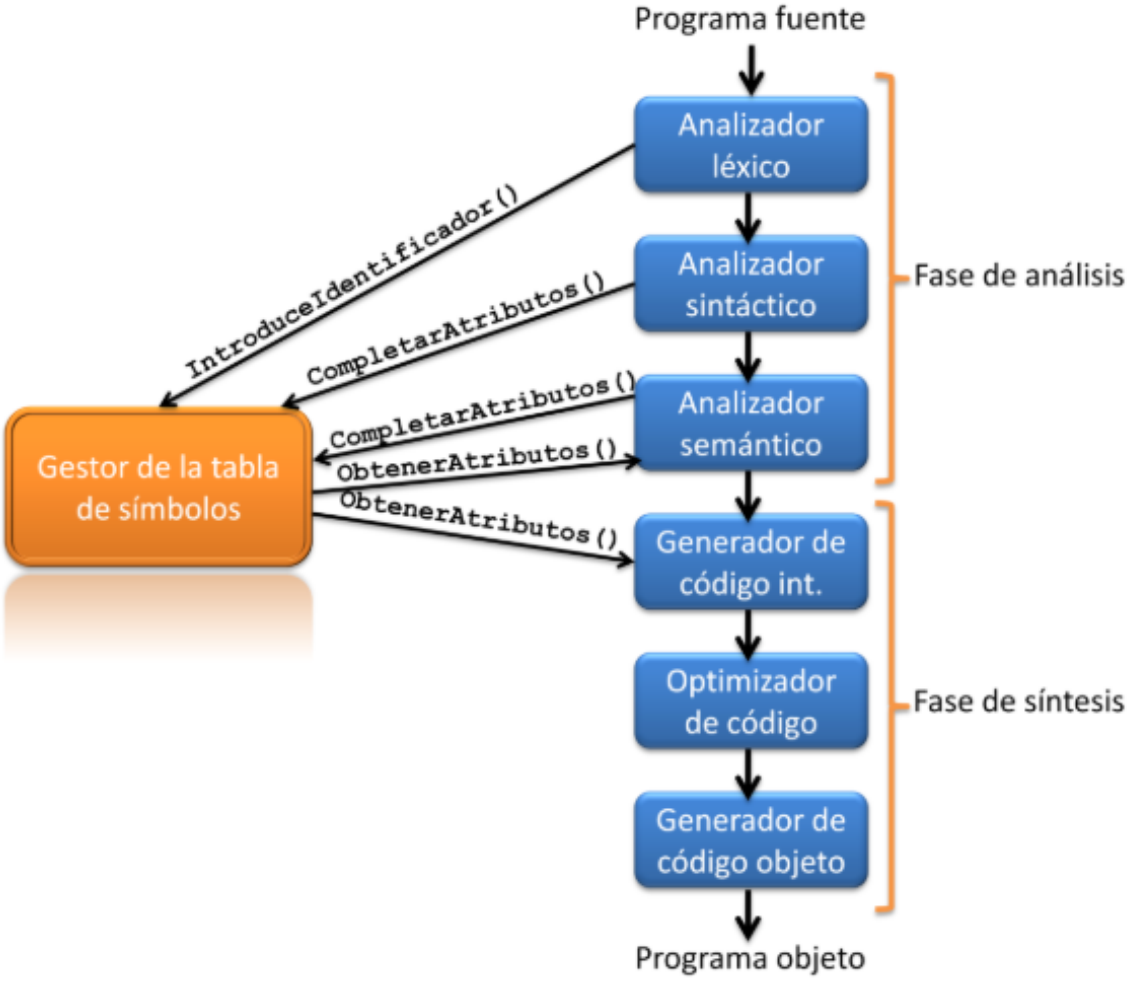

Figura 2: Gestor de una tabla de símbolos.

3.2.2. Tabla de símbolos

Se ha implementado un sistema de verificación de tipos (type checking). Esta verificación asegura que las estructuras y relaciones definidas en el programa sean válidas y coherentes antes de proceder con la generación de código o ejecución del programa.

El proceso de verificación comienza con la función typecheckProgram, que valida el programa completo, que se le es pasado desde el entrypoint. Si el programa contiene múltiples estructuras de proyectos, se verifica cada una de ellas recursivamente. En caso de encontrar una estructura de proyecto inválida, se reporta un error y se termina la ejecución.

La función typecheckProjectStructure se encarga de validar la estructura de un proyecto específico. Primero, verifica que la estructura no esté vacía y que el proyecto referido exista en la tabla de símbolos en el hash de proyectos. Luego, valida el cuerpo del proyecto y los opcionales, asegurándose de que ambos sean válidos para considerar la estructura del proyecto como válida.

Dentro del cuerpo del proyecto, typecheckProjectBody valida tanto los opcionales del cuerpo del proyecto como la lista de tareas. Los opcionales del cuerpo pueden incluir restricciones como el número máximo de tareas y puntos, así como la definición de categorías. La lista de tareas se valida recursivamente, verificando cada tarea individualmente.

La verificación de las tareas individuales se realiza en la función typecheckTaskStructure. Esta función valida que la estructura de la tarea no esté vacía, que la tarea referida exista y que su formato de duración y opcionales sean válidos. Las verificaciones incluyen asegurarse de que las fechas de inicio y fin sean coherentes y que los intervalos de duración estén correctamente definidos.

Las dependencias de las tareas son validadas en typecheckDependsOnId y typecheckTaskOptionDependsOn. Estas funciones aseguran que los proyectos y tareas referenciados existan y que las tareas no dependan de sí mismas. Además, se verifica que las tareas dependientes comiencen después de la finalización de las tareas de las que dependen, especialmente en proyectos con formato de fecha.

Por último, la función validateProjectReference verifica que los proyectos referenciados en los opcionales del proyecto existan. Las funciones auxiliares validateMaxTasks y validateMaxPoints aseguran que no se excedan las restricciones de número máximo de tareas y puntos en un proyecto. En caso de encontrar cualquier inconsistencia o violación de las reglas, se reporta un error y se termina la ejecución para evitar la generación de código incorrecto.

Este sistema de verificación de tipos garantiza que el programa esté estructurado correctamente y que todas las dependencias y restricciones estén bien definidas, asegurando así la integridad del proyecto antes de proceder con su procesamiento o visualización.

3.2.3. Generación de código

En la generación de código, se toma la representación interna del proyecto y la tabla de símbolos, y se traduce en un conjunto de instrucciones en lenguaje Python que utiliza la librería Plotly. Estas instrucciones son responsables de crear un diagrama de Gantt interactivo que visualiza el proyecto.

La generación de código realiza varias tareas específicas:

- **Creación de un DataFrame de Pandas:** Se construye un DataFrame, una estructura de datos tabular, para almacenar la información de las tareas del proyecto. Esta información incluye el nombre de la tarea, duración, fechas de inicio y fin, categorías y dependencias.
- **Generación de código Plotly:** Se generan las instrucciones en Python utilizando la librería Plotly para crear el diagrama de Gantt a partir del DataFrame. Esto abarca la elección del tipo de gráfico (barras o línea de tiempo) según la unidad de tiempo utilizada en el proyecto, la configuración de los ejes, el título y las opciones de visualización como colores y leyendas.
- **Manejo de dependencias:** Si el proyecto incluye dependencias entre tareas o proyectos, se genera el código necesario para representar estas relaciones en el diagrama de Gantt. Esto puede implicar ajustar las fechas de inicio y fin de las tareas para reflejar las dependencias, así como agregar elementos visuales como flechas o conectores.
- **Formato de salida:** El código Python generado se escribe en un archivo (en este caso, `gantt.py`) con el formato adecuado para que pueda ser ejecutado y producir el diagrama de Gantt.

Estas tareas aseguran que el diagrama de Gantt final no solo refleje con precisión la estructura y programación del proyecto, sino que también sea interactivo y visualmente informativo, facilitando así la comprensión y el seguimiento del progreso del proyecto.

3.3. Dificultades Encontradas

Al comienzo, la inexperiencia en el uso de Flex y Bison retrasó el desarrollo hasta que se pudo alcanzar un dominio adecuado de estas herramientas. La principal dificultad encontrada fue la implementación de la tabla de símbolos.

Se ha realizado una investigación que nos ha ayudado en su implementación, pero la definición de la estructura elegida fue responsabilidad del equipo, tal como se ha detallado anteriormente. No solo fue difícil la implementación, sino también entender cómo la tabla de símbolos intervenía a lo largo de los procesos.

4. Futuras Extensiones

Actualmente, la característica ‘unique‘, que asegura que una tarea no se superponga con otras tareas, se verifica solo para las tareas definidas antes de la tarea ‘unique‘. Sin embargo, no se verifica la superposición con las tareas definidas después de la tarea ‘unique‘. Esta limitación podría llevar a inconsistencias si las tareas posteriores tienen solapamientos no deseados con la tarea marcada como ‘unique‘.

Para futuras mejoras, se debería implementar una verificación más robusta que asegure que la tarea ‘unique‘ no se superponga con ninguna otra tarea del proyecto, independientemente de si estas tareas están definidas antes o después de la tarea ‘unique‘.

Se propone para una futura extensión agregar diferentes atributos para las tareas tales como:

- Incluir un atributo de prioridad para cada tarea, permitiendo especificar el orden de importancia o urgencia de las tareas.
- Añadir la capacidad de especificar costos asociados a cada tarea para el seguimiento presupuestario del proyecto.
- Implementar la generación de un diagrama de Gantt que resalte el camino crítico del proyecto, mostrando las tareas que determinan la duración total del proyecto.
- Implementar la generación de un diagrama de Gantt que resalte el camino crítico del proyecto, mostrando las tareas que determinan la duración total del proyecto.
- Permitir la asignación de recursos (por ejemplo: personal, equipamiento) a cada tarea. Esto incluiría definir recursos a nivel de proyecto y asignarlos a tareas específicas.
- Incluir un atributo de estado para cada tarea, habilitando especificar el estado de la misma, por ejemplo: complete, pendiente, empezada.

5. Conclusiones

El desarrollo del lenguaje específico de dominio para la creación y graficación de diagramas de Gantt ha sido un proceso integral que ha permitido aplicar y consolidar conocimientos teóricos y prácticos adquiridos en la presente materia (Autómatas, Teoría de Lenguajes y Compiladores). A lo largo de este proyecto, se han alcanzado varios objetivos clave, cada uno contribuyendo significativamente al resultado obtenido..

En primer lugar, la creación del lenguaje y su correspondiente compilador ha demostrado ser un desafío técnico significativo. La implementación de un analizador léxico utilizando Flex y un analizador sintáctico con Bison ha permitido estructurar y validar la sintaxis del lenguaje desarrollado. Esto ha requerido una comprensión profunda de las expresiones regulares y las gramáticas libres de contexto, así como la capacidad de manejar contextos anidados y diversas construcciones sintácticas.

Por otro lado la elección y comprender diferentes estructuras de datos utilizadas, como la tabla hash para la implementación de la tabla de símbolos ha sido importante para optimizar las consultas, y la complejidad de las diferentes operaciones dentro del lenguaje. Utilizando UTHASH, se ha logrado una gestión rápida y efectiva de las operaciones de búsqueda, inserción, actualización lo que ha facilitado la implementación del análisis semántico y la generación de código. Esta decisión ha sido particularmente útil dado el carácter único de los IDs utilizados para proyectos, tareas y categorías.

La generación de código en Python, utilizando la librería Plotly para visualizar diagramas de Gantt, ha sido un aspecto innovador del proyecto. La creación de DataFrames de Pandas y la generación de gráficos interactivos han proporcionado una herramienta visual potente para la gestión de proyectos, demostrando la aplicabilidad práctica del lenguaje desarrollado.

Además, se ha enfrentado y superado diversas dificultades, especialmente en la implementación de la tabla de símbolos y la integración de las diferentes fases del compilador. Este proceso ha permitido adquirir una comprensión más profunda de cómo interactúan los componentes de un compilador y la importancia de una implementación coherente y bien estructurada.

En conclusión, el proyecto ha sido un desafío significativo que nos ha permitido interpretar, investigar y desarrollar un lenguaje propio.

6. Bibliografía

- Cueva Lovelle, J. M. *Tablas de Símbolos en Procesadores de Lenguaje*. Cuaderno didáctico nº 54, Depto. De Matemáticas, Universidad de Oviedo, 1995.
- Cueva, J.M., Izquierdo, R., Juan, A.A., Luengo, M.C., Ortín, F., Labra, J.E. *Lenguajes, Gramáticas y Autómatas en Procesadores de Lenguaje*. Servitec, 2003.
- Hibbard, T.N. *Some Combinatorial Properties of Certain Trees With Applications to Searching and Sorting*. Journal of the ACM (JACM), vol. 9, no. 1, pp. 13-28, Jan. 1962.
- Apuntes de la cátedra.
- Hanson, T.D. *Uthash*. Disponible en: `http://troydhanson.github.com/uthash/`. Todos los derechos reservados. 2005-2018.