Trabajo práctico
Teoría del Lenguaje, Automatas y Compiladores

Primer cuatrimestre de 2025

Grupo: Unical

Integrantes:

Tiziano Fuchinecco	64191
Nicolas Revale	64227
Santiago Devesa	64223
Tomás Balboa	64237

Fecha de entrega: Domingo 22 de Junio 2025

Tabla de Contenidos:

1. Introducción

Este trabajo práctico, desarrollado para la materia *Teoría de Lenguajes y Autómatas*, consiste en el diseño e implementación de un compilador para un lenguaje específico de dominio (DSL) orientado a la planificación académica universitaria. El lenguaje permite definir materias, aulas, profesores, disponibilidad horaria, restricciones y preferencias, con el objetivo de generar automáticamente un cronograma completo en formato HTML.

El desarrollo se dividió en tres etapas: diseño del lenguaje, implementación del frontend (análisis léxico y sintáctico con Flex y Bison), y desarrollo del backend, encargado de validar semánticamente el programa y generar la salida final. Este informe describe cada una de esas etapas y el proceso seguido para construir la solución.

2. Modelo Computacional

2.1. Dominio

El dominio seleccionado para este trabajo práctico es la planificación académica cuatrimestral en universidades. El objetivo es facilitar la organización de horarios de cursada teniendo en cuenta materias, profesores, aulas, restricciones y preferencias específicas.

Actualmente, este proceso suele realizarse de forma manual o mediante combinadores de horarios con estructuras rígidas y poco flexibles. Por eso, se propone un lenguaje declarativo que permita describir de manera estructurada los elementos involucrados y sus relaciones, dejando en el compilador la generación automática de un cronograma viable.

El hecho de que se trate de un lenguaje declarativo le otorga al usuario una mayor libertad y flexibilidad a la hora de expresar sus intenciones, sin necesidad de especificar paso a paso cómo deben resolverse las asignaciones.

Consecuentemente, este enfoque busca reducir el esfuerzo manual del usuario, simplificando la logística y minimizando errores organizativos.

2.2. Lenguaje

Adentrándonos más específicamente al lenguaje, como se mencionó anteriormente, se optó por una sintaxis declarativa, evitando estructuras más rígidas como las de estilo JSON. Esto con el objetivo de lograr un estilo claro y amigable, orientado a una redacción que resulte intuitiva.

El lenguaje está enfocado en describir de manera clara y concisa, todos los elementos y aspectos relevantes que surgen en la organización de horarios académicos (dentro de una institución educativa). En resumen, el lenguaje se centra simplemente en declarar que se quiere lograr.

Profundizando un poco más, el DSL permite definir las siguientes entidades principales:

- Configuración Global: Permite establecer los horarios de apertura y cierre de la universidad, como también un rango de la duración de las clases.
- Aulas: Define el nombre de las aulas, especificando también la sede, la capacidad y el equipamiento.

- Materias: Los distintos cursos que se ofrecen, aclarando el nombre, la cantidad de horas semanales y si la materia requiere algún equipamiento específico para ser dada.
- Profesores: Detalla los profesores que forman parte del cuerpo docente, individualizandolo con su nombre, su horario de disponibilidad y las materias que enseña.
- Asignaciones: Indica que un profesor dicta un curso en un horario y una clase determinada. Puede ser que también lo dicte tanto en un día particular como en varios.
- Demandas: Exhibe la cantidad de estudiantes anotados en una materia específica.

- Preferencias: Se permite también que se agreguen preferencias en relación a horarios o días que un profesor encuentre más cómodos. Sin embargo, las preferencias no son restricciones obligatorias al momento de generar el cronograma.

3. Implementación

3.1. Frontend

Para esta parte se usaron Flex y Bison. Con Flex se armó el lexer que reconoce palabras clave (PROFESSOR, COURSE, etc.), identificadores, strings, números, horas, y otras cosas básicas. Después con Bison se hizo el parser, que se encarga de ver que el programa tenga una estructura válida según nuestra gramática, y de ahí se construyó un AST (árbol de sintaxis abstracta).

- Una configuración global (Configuration), que incluye datos como el horario de apertura de la universidad y la duración mínima y máxima de las clases.
- Una lista de declaraciones (DeclarationList), que puede contener definiciones de profesores, cursos, aulas, demandas y preferencias duras o suaves.

Cada declaración se representa con nodos específicos:

- Entity para profesores, cursos y aulas, con sus respectivos atributos.
- Preference y Demand para guardar restricciones duras o suaves.

- Cada Entity contiene una lista de Attribute, que encapsula propiedades como name, hours, available, canTeach, entre otras. Las propiedades que puede tener cada Entity dependen de su tipo.
- Los atributos tienen tipos heterogéneos, permitiendo representar distintos dominios: enteros, cadenas, intervalos horarios y días de la semana.

Desde el parser ya se realiza una validación estructural y semántica temprana del programa:

- Se verifica que cada entidad tenga los atributos correctos según su tipo. Por ejemplo:
    - Solo los profesores pueden tener atributos como available o canTeach.
    - Los cursos pueden tener hours, pero no available.

- Se valida que los atributos estén bien formados, por ejemplo:
    - Que HOURS esté seguido de un número entero.
    - Que los intervalos horarios estén correctamente expresados con FROM, TO y ON.

Si bien muchas validaciones se resolvieron directamente desde el parser gracias a la expresividad de Bison, hubo ciertos controles que, por la naturaleza del autómata de pila y sus restricciones estructurales, no se pudieron implementar en el frontend. Estas validaciones fueron delegadas al backend.

3.2. Backend

Con el AST ya creado y estructurado, el back se va a encargar de recorrer sus nodos para interpretar y discriminar correctamente las declaraciones hechas por el usuario en el programa fuente.

El recorrido del AST sucede en *validateSemantic* en *Generator.c* que recibe el AST como argumento (*Program *program*). En primer lugar se inicializa la configuración global de donde se extrae los horarios de apertura, cierre y el rango de duración de clases. Más adelante, iterando sobre la lista de declaraciones, para cada declaración el compilador “se detiene” a fijarse su contenido y tipo. Si es una entidad, se accede a su identificador y tipo. Luego, dentro de la entidad se recorre la lista de atributos para extraer información como: nombre, disponibilidad horaria (para profesores), capacidad, etc. Los datos se dividen en estructuras específicas (ej: Professor Data) y se agregan a la tabla de símbolos.

Por otro lado, si la declaración es una preferencia, primero la discrimina entre “HARD” y “SOFT”. Para cada preferencia dura, se extraen los identificadores que se hayan especificado (profesor, materia, día y horario) y posteriormente se realizan diversas validaciones, como por ejemplo: la existencia de las entidades, la duración de la

clase, la disponibilidad del profesor, entre otras. Luego, si supera todas las validaciones, se agrega a la lista de horarios. En el caso de las blandas, se validan las mismas restricciones que las duras pero no se agrega a la lista de horarios.

Finalmente, si se trata de una demanda se verifica que los cursos existan y que efectivamente el aula tenga la capacidad suficiente para la cantidad de estudiantes.

En resumen, el recorrido está dividido en fases para construir la tabla de símbolos, validar preferencias y verificar demandas. Aunque es cierto que el código recorre la lista varias veces, se decidió tomar esta decisión (envés de hacer todo en una pasada) con el objetivo de reflejar de forma más clara el propósito específico de cada fase.

Como se mencionó anteriormente, la validación de la semántica resulta un aspecto clave del proyecto. Se trató de tener en cuenta la mayor cantidad de casos posibles, incluyendo tanto situaciones válidas como errores. Entre algunas de las validaciones implementadas están: restricciones y superposiciones horarias (asegurando que los horarios están en el rango de apertura y cierre de la facultad y que además no se superpongan entre materias), superposición de clases dictadas por el mismo docente, requisitos de recursos (ej: si la materia necesita un proyector, el aula debe tener un proyector), capacidad de aulas, entre otras.

Profundizando un poco más en la tabla de símbolos, cada entrada en la tabla está compuesta por un identificador, un tipo de entidad (profesor, aula o materia) y un puntero a datos específicos. Cada entidad organiza sus datos de forma particular. Por ejemplo, Profesor almacena: nombre , disponibilidad horaria y las materias que enseña.

Para mantenerlo simple, se utilizó un array dinámico que duplica su capacidad cuando resulta necesario. Por otro lado no se utilizaron scopes pues todas las declaraciones tienen alcance global.

Una situación que se debe aclarar es que en caso que se declare más de una vez un aula con el mismo identificador, siempre que un aula con ese id se quiera utilizar para una materia, se asignará la que fue instanciada primera. Lo mismo sucede para los profesores, si Agustin se define más de una vez, cuando Agustin sea requerido (indicado por el programador o automáticamente por el back) para dar clase, se tomarán los horarios indicados en la primera instancia del profesor.

En relación a la etapa final del desarrollo del backend, la función *generateCode* crea un documento HTML con una tabla que detalla cursos, profesores, aulas, edificios, características, días y horarios, asegurando que las asignaciones respeten restricciones como disponibilidad de profesores y requisitos de aulas. También incorpora estilos CSS y un script de JavaScript que habilita el filtrado por materia, profesor y aula. Además agregamos la funcionalidad de ordenamiento por día, lo que permite que el calendario se vea en orden cronológico de Lunes a Viernes. Y en caso que no se opte por este ordenamiento, siempre se puede dejar en el orden de salida del programa. Este proceso

es validado por *validateSemantic* que garantiza un horario libre de conflictos, con manejo de errores y asignación automática de aulas cuando es necesario.

3.3. Adicionales (extra)

En caso que una materia sea por ejemplo de 6 créditos y se defina que esa clase se dicta por 4 horas, las horas faltantes para completar lo pedido inicialmente (6 créditos) se calculará automáticamente, en función de la disponibilidad de profesores que estén habilitados a dictar dicha materia y las aulas. En caso que se pueda hará que quede contiguo el horario y en el mismo aula de la primera clase dictada, aunque en la salida se verá en una fila diferente al horario que se declaró en la entrada. Por ejemplo, si se indica que Agustin da la clase de 14:00 a 18:00, luego faltarían 2 horas más para completar los créditos de la materia (en este ejemplo sólo Agustin dicta la materia y tiene disponibilidad horaria), entonces lo que hace el programa es indicar que Agustin dará las 2 horas restantes de 18:00 a 20:00. Y en la salida esto se verá porque habrá dos filas: una que indica que Agustin dicta de 14:00 a 18:00 y en otra que Agustin dicta de 18:00 a 20:00.

En caso que se instancie un aula, si luego se define una materia y un profesor que puede dictarla, el programador podrá dejar que el back decida el aula en la que se dictara dicha clase (no debe ser explícitamente declarado en la sentencia el aula elegida, debiendo esta ser seleccionada por el back), por ejemplo se permitirán instrucciones como la siguiente: *Ana teaches ATLyC from 16:00 to 19:00 on WEDNESDAY.* El programa siempre optará por elegir el aula que antes se haya instanciado y que además tenga disponibilidad horaria y los requisitos de la materia.

3.4. Dificultades Encontradas

La realidad es que el inicio del trabajo práctico, quizás fue el momento en el que el grupo se sintió más desorientado. Principalmente, por no haber tenido un trabajo práctico de índole similar y no terminar de entender concretamente cuál era el objetivo del proyecto. En consecuencia, el momento de definir posibles ideas a implementar fue el que generó más dudas. Sin embargo, una vez aclarado con la cátedra y entendiendo mejor el enfoque que tenía el trabajo, la continuación del mismo (teniendo en cuenta las clases que se iban dictando) resultó muy llevadero.

4. Futuras Extensiones

Existen varias extensiones que podrían incorporarse en versiones futuras de Unical. En primer lugar, sería ideal lograr que el sistema soporte múltiples tipos de

clases, es decir permitir que una materia pueda definir diferentes modalidades (práctica, teórica, presencial o virtual). Esto ampliará considerablemente la flexibilidad del sistema. Más adelante, otra extensión necesaria sería poder establecer un límite de horas semanales para los docentes, para así poder evitar asignaciones excesivas y distribuir mejor la carga horaria entre el cuerpo docente. También, tendría mucho sentido contemplar los tiempos de traslado entre sedes al momento de definir los horarios de materias consecutivas, para así evitar cronogramas poco realistas, tanto para los profesores como para los alumnos.

Abordando una extensión más concreta, hoy en día una limitación que se tiene es: si una materia es dictada por dos profesores y se especifica que ambos la dictan el mismo día en horarios superpuestos, al instanciar esto de esta forma, se completa la cantidad de créditos de la materia, aunque sería imposible que alguien curse la cantidad de créditos totales de esa forma pues no se puede estar en dos clases en simultáneo. Esto viene de querer instanciar dos clases en horarios superpuestos el mismo día, el programador deberá definirlas en horarios/días disjuntos o directamente instanciar una clase y dejar que Unical defina la otra en base a la disponibilidad.

5. Conclusiones

En primer lugar, como conclusión inicial, el trabajo práctico resultó ser de gran ayuda como para poder establecer una conexión más clara entre la parte teórica de la materia y su aplicación en el mundo real. Al principio muchas ideas de la materia resultan bastante abstractas (como las gramáticas libres de contexto o los autómatas de pila), pero tener que implementarlas en código para una tarea tangible (en este caso la organización de horarios académicos) ayudo muchisimo a entender mejor, tanto su utilidad como su funcionamiento. Es decir, en otras palabras, el trabajo práctico no solo sirvió para entender mejor algunos conceptos de la materia, sino para también tener más herramientas al momento de construir analizadores léxicos y sintácticos en otros contextos más adelante.

Por otra parte, en el trabajo el grupo se sintió satisfecho con el resultado obtenido al haber logrado desarrollar una solución a un problema real que tienen las instituciones educativas.

Sabemos que el poder asignar aulas y clases a los profesores en base a disponibilidad, preferencias y obligaciones es un proceso muy complejo por la gran variedad de combinaciones y suele llevarle a los directivos de las instituciones educativas mucho tiempo y esfuerzo mental, que con esta solución, podría ser destinado a otros asuntos.

6. Referencias