Taller de Diseño de Software

Integrantes: Balestra, Edgar Agustín
Buil, Delfina
Compagnucci Bruno, José Ignacio

Primera Etapa: Análisis Sintáctico y Léxico

División del trabajo en la etapa actual

Durante esta etapa del proyecto, tanto el desarrollo del Scanner como del Parser se llevó a cabo de manera colaborativa entre los integrantes del grupo. Para facilitar el trabajo en equipo, se utilizó la herramienta *Live Share* de Visual Studio Code, la cual permitió a todos los miembros conectarse a una misma sesión y realizar cambios en los archivos de forma simultánea y sincronizada. Este enfoque fomentó una comunicación constante y un trabajo colaborativo eficiente, asegurando que todos los integrantes pudieran contribuir activamente al desarrollo del proyecto.

Además, se aprovecharon las clases prácticas presenciales para trabajar juntos en persona, lo que permitió coordinar mejor las tareas y resolver dudas de manera inmediata durante los horarios de consulta. Esta combinación de trabajo presencial ayudó a maximizar el aprovechamiento del tiempo y a mejorar la calidad del trabajo en equipo.

Descripción del diseño:

La primera etapa del diseño e implementación del compilador para el lenguaje de programación TDS24 se centró en la creación de dos componentes principales: el Scanner y Parser. Estos componentes fueron desarrollados teniendo en cuenta la gramática proporcionada, que define un lenguaje de programación simple con estructuras de control, operaciones aritméticas y lógicas, como también la gestión de tipos de datos como `integer` y `bool`.

El Scanner, o analizador léxico, fue diseñado para reconocer los diferentes elementos del lenguaje. Esto incluye palabras clave, operadores, identificadores y literales, entre otros. Este componente toma como entrada un archivo con código fuente TDS24 y retorna tokens que son válidos dentro de la gramática del lenguaje; aquellos que no lo son son reportados como errores. El primer paso en la implementación de este componente fue definir una secuencia de tokens a los que el código fuente será convertido, permitiendo así que el Parser procese estos tokens de manera efectiva.

Las decisiones clave en el diseño del Scanner incluyen:

1. Definición de Tokens: se crearon tokens para las palabras clave (`if`, `while`, `return`, `Program`, etc), los operadores (`+`, `-`, `==`, `&&`, `||`, `!`, etc), y los tipos de datos (`integer`, `bool`). También se incluyeron tokens para los literales numéricos y booleanos, como `TINTEGER_LITERAL` y `TBOOL_LITERAL`, y los identificadores del lenguaje `TID`.
2. Comentarios: se implementaron dos tipos de comentarios que son ignorados por el Scanner, permitiendo así la inclusión de documentación en el código sin afectar el análisis léxico. Estos son los siguientes: comentarios de una sola línea (`// comentario`) y comentarios multilínea (`/* comentario */`).
3. Expresiones Regulares: se emplearon expresiones regulares para identificar ciertos patrones correspondientes a los distintos tokens del lenguaje. Por ejemplo, la expresión `digit` se define para reconocer dígitos del 0 al 9, `alpha` se utiliza para identificar letras y `alpha_num` para reconocer identificadores compuestos de letras, números y guiones bajos. Estas expresiones regulares sirvieron de base para la definición de tokens como `TINTEGER_LITERAL`, que corresponde a literales enteros, y `TID`, que representa identificadores dentro del código.

Por otro lado, el Parser, o analizador sintáctico, fue diseñado para recibir una secuencia de tokens como entrada y procesarla, verificando que esta sea una secuencia válida, es decir, que cumpla con la especificación sintáctica del lenguaje. Esta verificación asegura que los paréntesis y llaves estén correctamente balanceados, que la asociación de operadores sea adecuada, que la declaración de variables y métodos sea correcta, entre otros. La salida de esta etapa es simplemente si la entrada es sintácticamente correcta o no.

Las decisiones clave en el diseño del Scanner incluyen:

1. Definición de la Gramática: la gramática se definió utilizando la herramienta Bison, en donde se describe cómo los tokens se combinan para formar expresiones y declaraciones válidas en el lenguaje. Dentro de estos se incluyen sentencias condicionales (`if`), bucles (`while`), declaraciones de variables y métodos con sus respectivos parámetros, llamadas a métodos, definición de bloques de código con sus respectivas sentencias y expresiones, entre otros.
2. Precedencia de Operadores: se estableció un orden de precedencia para los operadores aritméticos y lógicos, lo que asegura que las expresiones sean evaluadas correctamente según las reglas establecidas. Además, se definió la asociatividad simple para ciertos operadores, como en el caso de ==, < y >, lo que implica que expresiones como `a < b < c` no estén permitidas.
3. Manejo de Errores: se incorporó un manejo básico de errores en el Parser. Cuando se encuentra un error de sintaxis, el programa imprime un mensaje que indica la línea donde ocurrió el error, ayudando en la depuración del código.

Decisiones de diseño

El diseño tanto del Scanner como del Parser se llevó a cabo teniendo en cuenta la gramática original y sus restricciones. No se realizaron grandes extensiones ni asumpciones significativas ya que se intentó, en la medida de lo posible, seguir fielmente la gramática presentada en la consigna. No obstante, las pocas decisiones adicionales que fueron tomadas se detallan a continuación:

- Modificación en la sentencia `while`: se añadieron paréntesis a la sentencia `while` ⟨`expr`⟩ ⟨`block`⟩ con el fin de delimitar claramente la expresión de control y el bloque de código asociado. Dicha modificación se implementó con el objetivo de mantener la convención utilizada en la sentencia `if`.

Detalles de implementación interesantes:

En la implementación del Scanner y del Parser, se optó por utilizar enfoques sencillos y directos, evitando la incorporación de algoritmos complejos y no triviales o estructuras de datos no usuales. La estrategia adoptada se centró en la simplicidad y en la aplicación de buenas prácticas de programación, con el objetivo de contar a futuro con un desarrollo modular y un mantenimiento eficiente del código.

Problemas conocidos en el proyecto:

Durante el desarrollo de esta etapa no se identificaron problemas significativos o errores en la implementación final. Para proporcionar robustez y corroborar la corrección del diseño, se proporcionaron casos de tests en donde se abarcan una variedad de escenarios del programa.