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 Parser 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.

Segunda Etapa: Tabla de Símbolos y AST

División del trabajo

Se trabajó de la misma manera que en la etapa anterior, colaborando en conjunto durante las clases presenciales y en horario fuera de estas.

Descripción del diseño:

La segunda etapa del diseño e implementación del compilador para el lenguaje de programación TDS24 se centró en el desarrollo del árbol sintáctico abstracto (AST) y la tabla de símbolos (TS) correspondiente para la gramática, que luego será usada para el desarrollo del análisis semántico de la etapa siguiente.

La tabla de símbolos se implementó con el objetivo de mantener la información de los símbolos (identificadores) del programa. Utilizando el árbol de parsing, se realizará el análisis semántico en la siguiente etapa haciendo uso de la información almacenada en la tabla de símbolos.

Las decisiones clave en el diseño del Árbol Sintáctico Abstracto (AST):

1. Modularidad de nodos: se decidió trabajar con una estructura de árbol que contiene nodos de diferentes tipos incluyendo nodos terminales para variables, literales y operadores, y nodos no terminales para estructuras de control, declaraciones y expresiones. Esto permite representar tanto elementos simples como complejos en el programa fuente.
2. Uso de subárboles: cada nodo del árbol puede tener subárboles para representar componentes del programa, como listas de declaraciones, bloques de código o expresiones compuestas. La implementación de un árbol binario fue suficiente ya que con dos hijos para una raíz se logró armar el AST sin ningún inconveniente.

Las decisiones clave en el diseño de la Tabla de Símbolos:

1. Estructura en forma de pila: la tabla de símbolos fue implementada como una estructura en forma de pila, compuesta por distintos niveles que van a ser insertados y eliminados a medida que se vaya construyendo en base al AST. Cada nivel de la pila corresponde a un bloque de alcance (“scope”) diferente y cuando este se elimina, toda la información correspondiente a ese ámbito es descartada, emulando así un comportamiento de pila. De esta manera es posible la representación de ámbitos diferentes dentro de un mismo programa sin que interfieran incorrectamente entre ellos.
2. Métodos de búsqueda: se implementaron métodos para llevar a cabo la búsqueda de un identificador dentro de un único nivel específico, como también la búsqueda de un elemento *a partir* de un nivel específico en la jerarquía de ámbitos. Esto último realiza una búsqueda descendente a partir del nivel proporcionado, si el símbolo no está presente en dicho nivel, se procede a buscar en el nivel inmediatamente anterior, continuando así el proceso hasta encontrar el identificador o llegar hasta el nivel global. De esta manera, se asegura que el método respete las reglas de visibilidad de los identificadores, priorizando los símbolos más cercanos al ámbito actual antes de buscar en niveles más globales.

Decisiones de diseño:

En el diseño de la tabla de símbolos, se estableció que el nivel 0 representa el ámbito global del programa. Este nivel está presente de manera permanente en la tabla y actúa como el nivel base que contiene las definiciones de los símbolos globales. A diferencia de los niveles superiores, que pueden crearse

y eliminarse dinámicamente conforme se entra y sale de diferentes ámbitos (como funciones o bloques de código).

Tercera Etapa: Generación de código Intermedio

En esta etapa se generó una representación intermedia (IR) del código, la cual es retornada por el compilador. Esto es necesario para luego generar código objeto a partir de esta. Se utilizó Código de Tres Direcciones para manejar operaciones aritméticas, lógicas y de control de flujo. Este código sirve como un puente entre el análisis semántico y la generación de código objeto, marcando la finalización del front-end del compilador.

Descripción del diseño:

El Código de Tres Direcciones es particularmente útil para las operaciones aritméticas con tipos enteros y lógicos, así como para las estructuras de control de flujo. Cada instrucción en esta representación se descompone en pasos simples que involucran operandos. Por ejemplo, una operación compleja puede dividirse en múltiples instrucciones que asignan los resultados intermedios a variables temporales.

Como el objetivo principal de este módulo es traducir el árbol sintáctico abstracto (AST) a un conjunto de instrucciones de tres direcciones que pueden ser utilizadas posteriormente para optimizar el código o generar código objeto, se llevó a cabo el diseño de un generador de código de tres direcciones (`threeAddresGenerator`). Este, se encarga de generar una lista enlazada de cuádruples en base al AST.

Para lograr esto, se implementó una lista enlazada (`QuadrupleLinkedList`) de cuádruples (`Quadruples`) para almacenar las instrucciones generadas, la cual será utilizada por el generador de código intermedio. Cada cuádruple contiene información sobre la operación que se llevará a cabo (`op`), sus operandos (`arg1` y `arg2`) y el destino en donde se almacenará el resultado (`result`). Esto se puede observar ilustrado a continuación:

```
result = argument1 operator argument2
```

Cabe aclarar, que no todos los cuádruples estarán siempre conformados por estos cuatro elementos, ya que el lenguaje permite operaciones unarias como los son la negación lógica por ejemplo, o realizar declaraciones de variables sin asignación.

Una vez implementada la estructura en donde se almacenarán las instrucciones generadas, se prosiguió a implementar el generador de código de tres direcciones. Este se encarga de recorrer el AST, generando instrucciones específicas para cada operación, cada estructura de control, cada pasaje de parámetros, etc.

Uno de los aspectos clave fue la gestión de variables temporales y etiquetas, las cuales se crean dinámicamente según sea necesario para representar expresiones y estructuras de control. Esto asegura que el código generado sea modular y fácil de interpretar en etapas posteriores del compilador.

Además, se diseñaron mecanismos para gestionar el contexto de métodos y parámetros. Por ejemplo, durante una llamada a un método, los parámetros se procesan en un orden específico y se generan cuádruples que reflejan esta jerarquía. También se distinguieron las variables globales de las locales mediante

indicadores específicos, asignándoles offsets a estas últimas, lo cual permite su correcta ubicación en memoria.

Cuarta Etapa: Generador de Código Objeto

En la cuarta y última etapa, se generó código assembly x86-64 (sin optimizaciones) a partir del código intermedio generado en la etapa anterior.

Descripción del diseño:

Se llevó a cabo la implementación de un generador de código assembly (`assemblyCodeGenerator`) que, a partir de la lista de cuádruplos generada por el `threeAddressGenerator`, produce código en lenguaje assembly. Este se almacena en un archivo con extensión `.s`, el cual luego se compila utilizando GCC y, finalmente, se ejecuta. Dado que el lenguaje admite funciones externas, aquellas que se utilicen en el código serán compiladas y vinculadas con el programa principal durante el proceso de compilación.

Este generador de código assembly se encarga de iterar sobre cada cuádruple perteneciente a la lista de cuádruples, analizando su contenido y generando una instrucción adecuada según el tipo de operación que se trate. Por ejemplo, cuando se trata de una operación de suma (`PLUS`), se genera el código assembly para sumar dos operandos y almacenar su resultado en una variable. En operaciones de comparación, como `EQUALS` o `LESS_THAN`, se comparan sus operandos y se generan instrucciones para almacenar el resultado de la comparación. Para llamadas a métodos (`CALL`), el código genera una instrucción para invocar una función y almacenar su valor de retorno.

La implementación de este generador también incluye el manejo de saltos condicionales, como `JMPF` (salto por falso) o `GOTO`, generando las instrucciones correspondientes para la manipulación del flujo de control. Cada tipo de cuádruplo tiene una función asociada que se encarga de emitir las instrucciones necesarias para esa operación específica, como `generateAddition`, `generateMultiplication`, `generateMethodCall`, entre otras.

El generador también se encarga de gestionar correctamente las variables locales y globales, las cuales tienen diferentes ámbitos de visibilidad y ciclo de vida dentro del programa. Cuando el generador de código assembly procesa una variable local, se realiza la asignación de un espacio adecuado en la pila de ejecución (stack). Las variables globales, en cambio, se almacenan en una región especial de memoria, fuera de la pila. El generador de código asigna una dirección de memoria fija para estas variables, permitiendo que sean accesibles desde cualquier punto del programa.

El generador de código también se asegura de que se respeten las convenciones de llamada para funciones externas, garantizando que los parámetros sean pasados de la manera correcta y que el valor de retorno sea gestionado adecuadamente.