Universidad Nacional de Río Cuarto

Facultad de Ciencias Exactas Físico Químicas y Naturales

Departamento de Computación

Taller de Diseño de Software

Proyecto
Segunda entrega

Campregher Bruno
Machuca Franco
Natali Valentino

Análisis Semántico

1. Introducción

El actual documento presenta todos los avances realizados en la generación de árboles abstractos sintácticos y de tablas de símbolos y el posterior análisis semántico.

La finalidad de esta entrega abarca la creación de las estructuras utilizadas para la representación de ambas abstracciones y la elaboración de herramientas para la construcción de dichas estructuras, en conjunto con la verificación de errores semánticos al momento de la construcción de la tabla de símbolos.

Repositorio del proyecto

El **código fuente completo** del proyecto está disponible en nuestro **repositorio de GitHub**.

La rama `Entrega-01/10` contiene la versión correspondiente a la **entrega actual**.

2. AST

Para la representación estructural del programa, se implementó un **Árbol de Sintaxis Abstracta (AST, por sus siglas en inglés)**. El AST es una estructura jerárquica que modela la organización lógica del código fuente, permitiendo separar la etapa de análisis sintáctico de las posteriores fases de compilación.

Diseño de la estructura

La implementación del AST se define en el archivo `ast.h`, en conjunto con los módulos de apoyo `info.h`, `params.h` y `enums.h`.
Cada nodo del árbol contiene la información necesaria para representar una construcción del lenguaje, la cual se organiza en dos componentes principales:

- Tipo de nodo: definido mediante un enumerado en `enums.h`, que identifica la categoría del nodo (por ejemplo, operaciones binarias, literales, identificadores, declaraciones, etc.).
- Información asociada: encapsulada en estructuras declaradas en `info.h` y `params.h`, que permiten almacenar valores, nombres de variables o listas de parámetros dependiendo del caso.

Las definiciones encontradas en los headers están implementadas en sus respectivos archivos .c.

Creación del Árbol

La construcción del AST se lleva a cabo durante la etapa de análisis sintáctico, implementada en el archivo `parser.y` mediante la herramienta **Bison**.
En esta etapa, cada regla gramatical no solo reconoce la estructura del lenguaje, sino que además **genera nodos del árbol** que representan las construcciones sintácticas correspondientes.

El mecanismo funciona de la siguiente manera:

1. **Acciones semánticas en Bison**
   Cada producción de la gramática está asociada a una acción en C. En dichas acciones, se invocan funciones de creación de nodos (definidas en `ast.h` y `ast.c`), que reciben como parámetros:
   - El tipo de nodo a construir (según el enumerado en enums.h).
   - La información asociada (constantes, identificadores, operadores).
   - Referencias a subárboles ya construidos.
2. **Composición jerárquica**
   El árbol se construye de manera **recursiva**: cuando una producción combina varias subproducciones, sus acciones unen los nodos parciales en un único nodo padre.
   Por ejemplo:
   - Una expresión binaria como a + b se convierte en un nodo `OPERADOR_BINARIA` con dos hijos: el nodo que representa a y el que representa b.
   - Una llamada a función se convierte en un nodo `CALL_FUNCION` que enlaza el identificador de la función con la lista de parámetros.
3. **Árbol raíz**
   La producción inicial de la gramática genera el nodo raíz del AST. Este nodo representa el programa completo, y desde él es posible recorrer toda la estructura.

De esta forma, el archivo `parser.y` actúa como el **puente** entre el análisis sintáctico y la representación abstracta del programa:

- El análisis gramatical asegura que la entrada cumple las reglas del lenguaje.
- Las acciones asociadas construyen progresivamente el AST, que será utilizado en las fases posteriores del sistema.

Asunciones realizadas

- Los programas vacíos son ignorados
- Los bloques vacíos o sin sentencias son ignorados

3. Tabla de Símbolos

La **tabla de símbolos** es una estructura fundamental en todo compilador o intérprete, ya que permite registrar y acceder a la información asociada a los identificadores declarados en el programa (variables, funciones, parámetros, etc.). En este proyecto, la implementación de la tabla de símbolos se encuentra en los archivos `tsim.c` y `tsim.h`.

Diseño y estructura

Símbolo: representa una entrada concreta en la tabla de símbolos.

- El campo `info` enlaza con una unión definida en `info.h`, que almacena los datos específicos del identificador (nombre, valor, tipo de dato, etc.).
- `next` permite organizar los símbolos en forma de **lista enlazada** dentro de un mismo nivel.
- `flag` indica el tipo de símbolo (variable, parámetro, función, etc.), facilitando la validación semántica.

Nivel: representa un **ámbito o contexto** del programa.

- Cada nivel contiene una lista de símbolos (`head`).
- El campo `parent` enlaza con el nivel inmediatamente superior, permitiendo modelar la **jerarquía de scopes**.

Funcionalidades principales

El módulo de tabla de símbolos implementa las siguientes operaciones:

1. **Creación e inicialización**
   Se proveen funciones para inicializar una nueva tabla, reservando el espacio necesario para su uso en el análisis semántico.
2. **Inserción de símbolos**
   Cuando se detecta una declaración en el código (por ejemplo, la creación de una variable o función), se agrega el símbolo a la tabla.

3. **Búsqueda de símbolos**
   Existen tres métodos para la busqueda de símbolos:
   - buscar_simbolo: Utilizado para buscar un símbolo en todos los ambientes de la tabla(niveles).
   - buscar_simbolo_en_nivell: Empleado para realizar búsquedas en el nivel actual de anidamiento.
   - buscar_ultimo_metodo: Busca el último método declarado en la tabla.

4. **Gestión de alcance (scope)**
   La tabla admite la diferenciación entre distintos contextos (por ejemplo, variables locales y globales). Esto permite reutilizar nombres en diferentes bloques del programa sin generar conflictos.

4. Análisis Semántico

La etapa de análisis semántico constituye el siguiente paso tras la construcción del AST y la tabla de símbolos. Su finalidad es verificar que el código escrito del lenguaje, analizado desde el AST, esté exento de errores semánticos referidos al lenguaje. A su vez, es por este recorrido del árbol que se desarrolla la tabla de símbolos.

Diseño e implementación

El análisis semántico se implementa en los archivos semantico.c y semantico.h.

La función principal analisis_semantico recibe como entrada el árbol raíz y el nivel global de la tabla de símbolos. Desde allí se inicia el procesamiento de:

- **Declaraciones de variables y funciones**:
  - Se validan las declaraciones de variables a través de procesar_declaracion_variable y declarar_variable, que comprueban los tipos asociados y vinculan los identificadores a la tabla de símbolos.
  - En el caso de funciones, procesar_declaracion_metodo y declarar_metodo registran cada método, controlan sus parámetros y gestionan los bloques de código asociados a su cuerpo.
- **Gestión de ámbitos**:
  - Cada bloque de código abre un nuevo nivel en la tabla de símbolos mediante procesar_bloque, lo que permite modelar adecuadamente el alcance de las variables y parámetros.

    - Al finalizar el bloque, dicho nivel es cerrado, garantizando la correcta visibilidad y vida útil de los símbolos.

- **Procesamiento de sentencias**:

    - La función procesar_statement actúa como punto central para recorrer estructuras de control (if, while), bloques anidados, asignaciones, sentencias de retorno y llamadas a función.

    - En cada caso, se aplican verificaciones de tipo y consistencia semántica mediante funciones especializadas (procesar_if, procesar_while, procesar_asignacion, procesar_return, procesar_metodo).

- **Expresiones y operadores**:

    - El análisis de expresiones se realiza con procesar_expresion, que distingue entre identificadores, literales, operadores unarios/binarios y llamadas a funciones.

    - Para los operadores, procesar_operador valida la compatibilidad de tipos entre operandos, determinando el tipo resultante de la operación.

    - El método obtener_tipo permite inferir el tipo de cualquier subárbol del AST, lo cual resulta fundamental para las verificaciones posteriores.

- **Reglas específicas del lenguaje**:

    - Se introduce un control especial para la función main, asegurando que exista exactamente una definición y gestionando el caso de parámetros no permitidos.

    - Mediante buscar_return, se comprueba que las funciones que declaran un tipo de retorno distinto a void efectivamente devuelvan un valor en todos sus caminos de ejecución.

5. Manejo de Errores en el Análisis Semántico

Diseño e implementación

El módulo de errores se implementa en los archivos errores.h y errores.c.
Su función principal es servir de intermediario entre las verificaciones semánticas realizadas en el recorrido del AST y la presentación de mensajes descriptivos al usuario.

Los errores identificados se almacenan en un arreglo global de estructuras Error, cada una compuesta por:

- **Código de error (CodigoError)**: definido como un enumerado en enums.h, que categoriza el tipo de error (por ejemplo, VAR_NO_DECLARADA, TIPO_INCOMPATIBLE, FUN_SIN_RETURN, entre otros).

- **Mensaje**: cadena de texto que describe la naturaleza del error, la posición en el código (línea y columna) y los detalles contextuales.

La cantidad de errores registrados se controla mediante un contador global, limitado a un máximo predefinido.

Reporte y acumulación de errores

- La función principal para el registro es reportarError, la cual recibe el código de error, la posición (línea y columna) y argumentos adicionales según el tipo de error.

- A partir de esta información, se construye un mensaje detallado que se almacena en el arreglo de errores.

- El sistema permite acumular múltiples errores durante una misma ejecución, sin detener prematuramente el análisis semántico.

Visualización de errores

El módulo ofrece la función mostrarErrores, que recorre el arreglo global e imprime los mensajes generados. Esto facilita la inspección de todas las inconsistencias detectadas de manera centralizada, permitiendo al usuario corregirlas en el código fuente.

Integración con el análisis semántico

Durante la ejecución de las funciones del módulo semántico (declarar_variable, procesar_if, procesar_asignacion, entre otras), se invoca reportarError en los casos en que se detectan violaciones a las reglas del lenguaje.

6. Ejecución del compilador

El archivo principal del proyecto (main.c) es reutilizado de la entrega anterior. Se agrega la etapa para la generación de un **.dot** del árbol y la posterior generación de una imágen del mismo

Uso del programa

```
programa [-target etapa] archivo.ctds
```

Donde:

- -target scan ejecuta únicamente el analizador léxico y genera como salida un archivo con extensión .lex.

- -target parse ejecuta el analizador sintáctico (por defecto si no se especifica -target) y genera un archivo con extensión .sint.
- -target ast realiza la creación del archivo dot en base al árbol, aparte de todas las etapas anteriores (scan y parse).
- **-target sem**: ejecuta el análisis semántico del programa. En esta etapa se generan los archivos .lex y .sint como en las fases previas, y adicionalmente un archivo con extensión .sem que contiene todos los errores semánticos detectados, gracias al uso del módulo de manejo de errores.

- archivo.ctds corresponde al programa fuente a compilar. Debe finalizar con la extensión .ctds.

Para la generación del png en base al archivo dot se necesita la anterior instalación de Graphviz. Luego se ejecuta el comando **dot -Tpng arbol.dot -o arbol.png** para generar la imágen.

Manejo de archivos

- El compilador valida que el archivo fuente sea correcto (no debe comenzar con - y debe terminar en .ctds).

- En caso de error al abrir el archivo fuente o crear el archivo de salida, se informa al usuario mediante un mensaje descriptivo.

- El nombre del archivo de salida se deriva automáticamente del archivo de entrada, reemplazando la extensión .ctds por .lex, .sint o .sem según corresponda.

Asunciones realizadas

- El parámetro -target solo admite los valores scan, parse, ast y sem.

- Si no se especifica -target, la etapa ejecutada por defecto es sem.

- Se asume que el archivo de entrada existe y es accesible para lectura.

7. Proceso de compilación

En esta entrega, el archivo **Makefile** fue extendido para mejorar la organización del proyecto y facilitar el mantenimiento del proceso de compilación. Los principales cambios son:

- **Separación en directorios**:

    - Se agregó la carpeta `includes` para los headers(`.h`).
    - Se centralizaron los archivos fuente auxiliares en `src/utils`.
    - Se definió la carpeta `build` como destino de los archivos objeto (`.o`).

- **Variables adicionales**:
  Ahora se emplean variables para definir rutas y archivos, lo que permite modificar fácilmente la estructura del proyecto sin tener que cambiar múltiples líneas del Makefile. Ejemplo:
    - `INCLUDES`, `SRC` y `BUILD` para directorios.
    - `LEXER`, `PARSER`, `SRCS` para los archivos fuente.

- **Compilación diferenciada de archivos especiales**:
  Se definieron reglas independientes para:
    - `parser.tab.o` generado por **Bison**.
    - `lex.yy.o` generado por **Flex**.
    - `main.o` compilado directamente desde el archivo principal.