Proyecto Final - TC

Asignatura:
Técnicas de Compilación
Profesor:
Eschoyez, Maximiliano A.

Escrito por:
Ventura, Gino

Fecha de entrega: 02/10/2024

UBP UNIVERSIDAD Blas Pascal

CONTENIDO

INTRODUCCIÓN

OBJETIVO

El objetivo de este trabajo final es extender las funcionalidades del programa realizado como trabajo práctico número 2. El programa por desarrollar tiene como objetivo tomar un archivo de código fuente en C, generar como salida una verificación gramatical, reportando errores en caso de existir, generar código intermedio y realizar alguna optimización al código intermedio.

CONSIGNA

Dado un archivo de entrada en C, se debe generar como salida el reporte de errores en caso de existir. Para lograr esto se debe construir un ***parser*** que tenga como mínimo la implementación de los siguientes puntos:

1. Reconocimiento de bloques de código delimitados por llaves y controlar el balance de apertura y cierre.
2. Verificación de la estructura de las operaciones aritmético/lógicas y las variables o números afectados.
3. Verificación de la correcta utilización del punto y coma para la terminación de instrucciones.
4. Balance de llaves, corchetes y paréntesis.
5. Tabla de símbolos.
6. Llamada a funciones de usuario.

Si la fase de verificación gramatical no ha encontrado errores, se debe proceder a:

1. Detectar variables y funciones declaradas, pero no utilizadas y viceversa.
2. Generar la versión en código intermedio utilizando ***código de tres direcciones***.
3. Realizar optimizaciones simples como propagación de constantes y eliminación de operaciones repetidas.

En resumen, dado un código fuente de entrada, el programa deberá generar la siguiente salida:

- Tabla de símbolos para todos los contextos.
- La versión en código de tres direcciones del código fuente.
- La versión optimizada del código de tres direcciones.

DESARROLLO DEL PROYECTO

DEFINICIÓN DE LA GRAMÁTICA (.G4)

La gramática del compilador fue implementada utilizando ANTLR4, definiendo la sintaxis del lenguaje que maneja operadores aritméticos, comparativos, lógicos, control de flujo y funciones.

El programa se compone de una serie de instrucciones que pueden ser declaraciones de variables, asignaciones, llamadas a funciones, operaciones lógicas y aritméticas, así como estructuras de control como if, while y for. La gramática está diseñada para permitir la combinación de estas instrucciones dentro de bloques de código, lo que otorga flexibilidad.

LISTENER: GENERACIÓN DE LA TABLA DE SÍMBOLOS Y MANEJO DE ERRORES

La clase ***"MiListener"*** es una implementación que se encarga de gestionar el análisis semántico del código durante el proceso de compilación. Utiliza una pila de contextos para manejar los ámbitos de las variables y funciones, permitiendo la declaración y el uso de variables locales y globales.

Al ingresar a bloques de código o funciones, el ***listener*** crea nuevos contextos en la pila, lo que facilita la verificación de la existencia y el estado de las variables.

Dentro de la clase ***"MiListener"***, utilizamos métodos de la clase ***"MisErrores"*** la cual se encarga de centralizar la captura y reporte de los distintos tipos errores semánticos que pueden ocurrir durante el análisis de un programa, como el uso de variables no declaradas, funciones no implementadas o faltas de retorno en las funciones *(Imagen 1)*. Además de detectar errores, también permite emitir advertencias sobre prácticas incorrectas, como variables o funciones que se declaran, pero nunca se utilizan en el programa *(Imagen 2)*.

Además, al final del análisis, se genera un informe que muestra la tabla de símbolos *(Imagen 3)*, indicando el estado de cada variable y función, y se destacan los errores o advertencias en caso de que hubiera.

```
Error en línea [7:20]: La función 'restar' no ha sido declarada
Error en línea [7:20]: La función 'restar' no ha sido declarada
Error en línea [10:4]: 'resultado' no ha sido declarado.
Error en línea [13:12]: 'resultado' no ha sido declarado.
```

Imagen 1 - Errores

```
Advertencia: La variable global 'final' fue declarada pero nunca usada.
Advertencia: La variable global 'z' fue declarada pero nunca usada.
Advertencia: La función 'restar' fue declarada pero nunca usada.
Advertencia: La variable local 'h' en la función 'restar' fue declarada pero nunca usada.
Advertencia: La función 'dividir' fue declarada pero no implementada.
Advertencia: La función 'dividir' fue declarada pero nunca usada.
```

Imagen 2 - Advertencias

```
--------------- Tabla de Símbolos ---------------

Variables Globales:
  Nombre: a, Inicializado: Sí, Usado: Sí
  Nombre: b, Inicializado: Sí, Usado: Sí
  Nombre: final, Inicializado: Sí, Usado: No
  Nombre: z, Inicializado: No, Usado: No
  Nombre: resultadoMult, Inicializado: Sí, Usado: Sí
  Nombre: resultadoSuma, Inicializado: Sí, Usado: Sí

Funciones:
  Funcion: sumar, Implementada: Sí, Usada: Sí
    Parámetros: [a, b]
    No hay variables locales declaradas.
  Funcion: multiplicar, Implementada: Sí, Usada: Sí
    Parámetros: [e, f]
    Variables Locales:
      Nombre: resultado, Inicializado: Sí, Usado: Sí
  Funcion: restar, Implementada: Sí, Usada: No
    Parámetros: []
    Variables Locales:
      Nombre: c, Inicializado: Sí, Usado: Sí
      Nombre: d, Inicializado: Sí, Usado: Sí
```

Imagen 3 - Tabla de símbolos

VISITOR: GENERACIÓN DEL CÓDIGO DE TRES DIRECCIONES

La clase ***"MiVisitor"*** es responsable de generar código intermedio *(Imagen 4)* en forma de tres direcciones a partir de la representación abstracta del código fuente analizado. Esta clase extiende de ***"compiladoresBaseVisitor"*** y utiliza listas para almacenar las instrucciones generadas, representando el código de entrada ingresado.

Al visitar nodos en el árbol de sintaxis, el ***visitor*** crea variables temporales y etiquetas para las operaciones, lo que permite manejar expresiones complejas. Además, incluye métodos para el manejo de declaraciones de variables, asignaciones, definiciones y llamadas de funciones, así como estructuras de control como ***if, while*** y ***for***.

El ***visitor*** también implementa una optimización simple que es eliminar variables no utilizadas mediante una comparación con la tabla de símbolos, lo que resulta en un código intermedio mas limpio. Al final, se puede imprimir por consola o guardar el código generado en archivos de texto, tanto en su forma original como optimizada, facilitando su análisis y comprensión.

```
1: a = 3
2: b = 2
3: z = 0

4: param1 = a
5: param2 = b
6: t0 = call sumar

7: resultadoSuma = t0
8: resultadoMult = 0

9: t1 = resultadoSuma >= 5
10: if t1 goto L0
11: goto L1
12: L0:

13: param1 = a
14: param2 = resultadoSuma
15: t2 = call multiplicar

16: resultadoMult = t2
17: goto L2
18: L1:
19: resultadoMult = 0
20: L2:
21: final = resultadoMult

22: sumar:
23: a = param1
24: b = param2
25: t3 = a + b
26: return t3

27: multiplicar:
28: e = param1
29: f = param2
30: t4 = e * f
31: resultado = t4
32: return resultado
```

Imagen 4 – Código de tres direcciones

CONCLUSIÓN

El desarrollo de este compilador utilizando ANTLR4 permitió la implementación de una gramática robusta y flexible que abarca varios aspectos del lenguaje C, como operadores aritméticos, comparativos, lógicos y algunas estructuras de control.

La utilización del ***listener*** para la gestión del análisis semántico fue crucial a la hora de detectar errores y advertencias, asegurando que las variables y funciones sean utilizadas adecuadamente en el contexto correcto. Además, facilitó la generación de una tabla de símbolos para proporcionar información sobre el estado de las variables y funciones.

Por otro lado, la implementación del ***visitor***, ayudó a realizar la generación del código intermedio en forma de tres direcciones, añadiendo también la optimización de la eliminación de variables no utilizadas.

Este proyecto, no solo ha logrado el objetivo de analizar y compilar código en C, sino que también me permitió entender mejor como función un compilador y las etapas del análisis sintáctico y semántico, la importancia de la tabla de símbolos y cómo se genera el código intermedio.

LINK A REPOSITORIO DE GITHUB

https://github.com/ginoventura/TPFINAL-TC