UNIVERSIDAD
Blas Pascal

Técnicas de Compilación

Profesor: Maximiliano A. Eschoyez
Alumno/s: Javier Ismael Figueroa Rivarola
Materia: Técnicas de Compilación.
Fecha:25/02/2022

Consigna

El objetivo de este Trabajo Final es mejorar el filtro generado en el Trabajo Práctico Nro. 2. 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:

- Reconocimiento de bloques de código delimitados por llaves y controlar balance de apertura y cierre.
- Verificación de la estructura de las operaciones aritmético/lógicas y las variables o números afectadas.
- Verificación de la correcta utilización del punto y coma para la terminación de instrucciones.
- Balance de llaves, corchetes y paréntesis.
- Tabla de símbolos.
- Llamado 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, el cual fue abordado en clases y se encuentra explicado con mayor profundidad en la bibliografía de la materia,

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

1. La versión en código de tres direcciones con las etiquetas de bloque.

Presentación del Trabajo Final

Código Fuente

El código fuente generado para este proyecto y la versión digital del informe en PDF deberán entregarse a través del enlace correspondiente en la plataforma MiUBP. En dicho enlace se deberá subir un único archivo en formato ZIP conteniendo todos los código fuente que se requieran para la realización del trabajo final y el informe.

Informe Escrito

Se entregará al profesor un informe escrito (en versión digital formato PDF) donde se debe describir la problemática abordada en el trabajo final, el desarrollo de la solución propuesta y una conclusión. El texto deberá ser conciso y con descripciones apropiadas. No se debe incluir el código fuente, sino los textos necesarios para realizar las explicaciones pertinentes.

Introducción

Este trabajo consiste en extender las funcionalidades que se realizaron en la entrega del Trabajo Practico N.º 2. El objetivo es desarrollar un programa que a partir de un archivo con extensión “.C” generar una salida en la cual exista una verificación gramatical y reporte los errores encontrados durante la ejecución en caso de que existan, generar código intermedio y realizar el etiquetado del bloque de código intermedio para futuras optimizaciones.

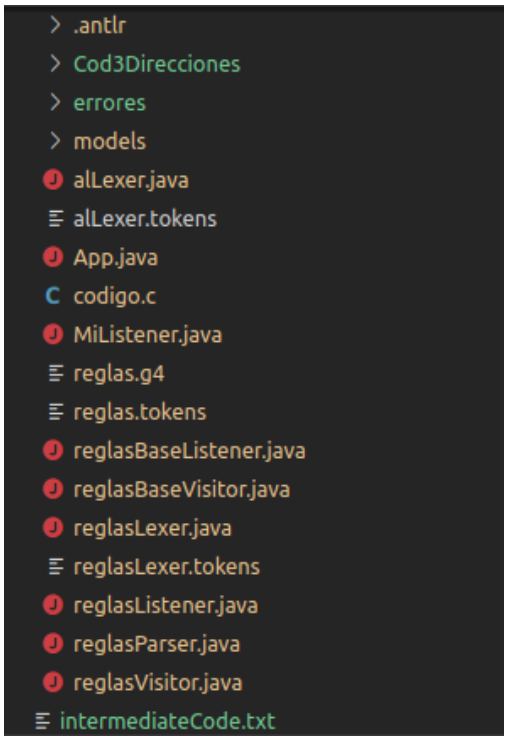
Desarrollo

Este trabajo se utilizando como IDE “VISUAL STUDIO CODE” , Java y como complemento/Plug-in ANTLR4(es un potente generador de analizadores para leer, procesar, ejecutar o traducir texto estructurado o archivos binarios. Es ampliamente utilizado para construir lenguajes, herramientas y marcos. A partir de una gramática, ANTLR genera un analizador que puede construir y analizar árboles).

Lo que se realizo fue un parser o analizador sintáctico para el lenguaje C, que cumpla las siguientes condiciones:

1. Reconocimiento de bloques de código delimitados por llaves y controlar balance de apertura y cierre.
2. Verificación de la estructura de las operaciones aritmético/lógicas y las variables o números afectadas.
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. Llamado a funciones de usuario.

La estructura principal del proyecto es :



Cod3Direcciones

Dentro de la carpeta Cod3Direcciones se encuentra el archivo “Cod3Direcciones.java”, su función principal es recorrer el árbol y visitar cada uno de los nodos de los subarboles; una vez realizado se guarda el recorrido dentro del archivo intermediateCode.txt. Esto sirve para optimizar nuestro compilador y mejorar el código.

Errores

En la carpeta de errores se encuentran los archivos “ErroresListeners.java” y “ErroresEncontrados.java”. Dentro de “ErroresEncontrados,java” su función es imprimir los errores sintácticos mientras que “ErroresListeners.java” lo que hace es imprimir la posición del error y el formato del mismo.

Models

Aquí se encuentran los archivos “ID.java” , “Funcion.java” , “TablaSimbolos.java” y “Variable.java”.
El archivo “Funcion.java” es guardar todas las características de una variable dentro de un arraylist. Dentro de “ID.java” se crean todas las características de un identificador, nombre,tipo de dato, si fue usado o inicializado. Dentro de “Variable.java” lo que se hace verificar sino se repiten los identificadores.
“TablaSimbolos.java” Esta clase mantiene la información asociada con los identificadores: funciones, variables, constantes y tipos de datos. Lo que hace es interactuar con casi todas las fases del compilador esto se debe a que pueden introducir identificadores dentro de la tabla; el analizador semántico agregará tipos de datos y otra información; y las fases de optimización y generación de código utilizarán la información proporcionada por la tabla de símbolos para efectuar selecciones apropiadas de código objeto.

App.java

Esta es la clase principal del proyecto, donde se ejecuta el proyecto.

Codigo.c

Es el archivo que va a ejecutar la clase principal, para mostrar los resultados de la compilación.

MiListener.java

Contiene toda la lógica del programa, desde encontrar los errores y mostrarlos, hasta las llamadas a funciones, return, entrar a los bloques, los prototipos de las funciones, declaración de variables.

Reglas.g4

El archivo reglas.g4 contiene la gramática para el analizador llamado ANTLR 4.

Solución

Una vez hecho esta estructura, la funcion utilizada para ejecutar el codigo.c es:

```
int funcionM(int c);
int funcionM(int c){
   c = c + 10;
   return c;
}
int main(){
   int x;
   x = funcionM(10);
   int h;
   int i;
   for(i = 0; i < 18; i++){
      h = 3 * 10;
   }
}
```

Una vez ejecutado esa porción de código el resultado es el siguiente mediante la terminal:

```
=== SYMBOL TABLE ===
Contexto: 1 {
   int funcionM([int c true true])
   int main([])
}
Contexto: 2 {
   int c true true
}
Contexto: 3 {
   int x true false
   int h true false
   int i true false
}
Contexto: 4 {
}
funcion comienza funcionM
t0 = c + 10
c = t0
return c
funcionM termina
funcion comienza main
2
param 10
t1 = call funcionM, 1
x = t1
i = 0
L1:
t2 = ;i < 18
ifnot t2, goto L2
t3 = 3 * 10
h = t3
i++
goto L1
```

L2:
main termina

Conclusión

Desde mi punto de vista durante el transcurso de la materia y las diferentes situaciones que abordamos durante los trabajos prácticos entendí como es la estructura de un compilador, sus diferentes fases desde el análisis léxico hasta las optimizaciones.