UNIVERSIDAD BLAS PASCAL

TÉCNICAS DE COMPILACIÓN

MAXIMILIANO A. ESCHOYEZ

TRABAJO FINAL

26 DE NOVIEMBRE DE 2021

IBAZETA, JIMENA

SPECIA, SANTIAGO.

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.
3. etiquetar los bloques de código intermedio a los cuales podrían aplicarse optimizaciones
4. implementar una optimización independiente del hardware.

En resumen, dado un código fuente de entrada el programa deberá generar 2 archivos de salida:

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

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 el Aula Virtual del curso. 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

En el siguiente informe abordaremos la solución a la problemática planteada por el profesor en la materia “Técnicas de compilación”. Para ello explicaremos de manera detallada los pasos a seguir para cumplir con el objetivo de la consigna.

La forma de trabajar será en equipo de dos personas, donde se realizarán reuniones de manera cotidiana y al final del informe analizaremos el cumplimiento de los objetivos.

SOLUCIÓN:

Analizaremos el código fuente de un programa escrito en lenguaje C, el mismo se encuentra dentro de la carpeta del proyecto con el nombre “CodigoPrueba.txt”, éste es un simple programa que define funciones, declara variables, hace operaciones lógicas, aritméticas y demás.

En base a dicho archivo procedemos a implementar la solución:

1. ANÁLISIS LÉXICO:

   Es la primera fase del compilador, lo que hace es tomar como entrada el código fuente que mencionamos anteriormente y devolver Tokens.

   Para obtener dichos tokens contamos con un archivo llamado “id.g4”, donde encontraremos expresiones regulares y reglas definidas. Las reglas se componen de expresiones regulares u de otras reglas, permitiéndonos utilizar patrones recursivos.

   Dentro del archivo podremos notar el tipo token SKIP que quiere decir que el token no debe ser pasado al nivel sintáctico.

2. ANÁLISIS SINTÁCTICO:

   El objetivo del analizador sintáctico es reconocer patrones en el flujo de tokens que le llega del nivel léxico y organizar la información en un árbol de sintaxis.

   Es decir que podemos generar el árbol a partir de los token obtenidos anteriormente. En este punto debemos verificar que el árbol quede de manera correcta, realizando correcciones en el archivo .g4 si fuese necesario.

3. ANÁLISIS SEMÁNTICO

   En este punto, con la herramienta de ANTLR y luego de haber procesado el archivo .g4 correspondiente, se generan las clases en java con los métodos listos para ser re implementados de acuerdo a la estructura (parser) que tengamos.

   Entonces, con la posibilidad de agregar comportamiento durante el análisis sintáctico, se implementaron estructuras de datos tales que representen las tablas de símbolos de un programa y los contextos del mismo. En ellos se fue almacenando las funciones y variables que el programa utiliza, también así su tipo de dato, estado y uso.

   La misión de la tabla de símbolos es colaborar en las comprobaciones semánticas y facilitar la generación de código. Las operaciones que podemos necesitar para utilizarla son la de inserción, búsqueda o consulta, actualización y eliminación. La eficiencia de estas

operaciones depende de la estructura de datos utilizada, y puede hacer que el compilador consuma mucho tiempo en los accesos a la misma.

Esta información que la tabla almacena está disponible en tiempo de compilación, puesto que se utiliza para construir el compilador.

Esto nos permitió detectar errores de compilación:

- ✓ Declarar dos variables o funciones con el mismo nombre.
- ✓ Usar variables o funciones no declaradas.
- ✓ Implementar una función más de una vez.

Errores gramaticales:

- ✓ Variables no inicializadas.
- ✓ Funciones sin implementar

Problemas de optimización como:

- ✓ Variables y funciones no utilizadas.

4. CÓDIGO INTERMEDIO

A partir del paso anterior, se genera el código intermedio. Éste nos permite pasar de un lenguaje expresivo a un lenguaje cercano al código máquina.
Si bien existen diversos tipos de códigos intermedios que varían en cuanto a su sencillez, nosotros nos centraremos en un tipo de código que se parece bastante al lenguaje ensamblador, permitiéndonos realizar optimizaciones.

El formato que usaremos para las operaciones binarias es:

T1: = T2 op T3

Donde:

- T1, T2 y T3 son registros, de los que tenemos un número infinito. Pueden ser un nombre, una constante o una variable temporal generada por el compilador.
- op es un operador.

En este ejemplo podemos ver que *x:= y + z – (u * w)* se transforma en :

Tmp1:= u*w

Tmp2:= z – Tmp1

X:= y + tmp2

En el siguiente ejemplo tomado de nuestro programa podemos ver como se evalúa la condición de la expresión booleana (a < 10), donde si es cierta ejecuta el código correspondiente a la sentencia t13 y si es falsa salta a t15 , que en este caso equivale a continuar con el resto del código del programa

```
t11: a < 10
t12: if not t11 goto t15
t13: c - While-context = 100
t14: goto t12
t15: a > 2
```

Luego de generar el código intermedio, se aplicaron las siguientes optimizaciones:

- Método removeUninitializedVariables nos permite remover variables que no están inicializadas.
- Método replaceOperationsWithConstants nos permite reemplazar las operaciones matemáticas por constantes para evitar el cálculo en tiempo de ejecución
- Método removeUnusedVariables nos permite eliminar variables sin uso.
- Método removeUnusedFunctions nos permite eliminar funciones sin uso.
- Método removeEmptyFunctions() para eliminar funciones vacías.

CONCLUSIÓN:

El proyecto nos permitió aprender sobre el proceso de compilación, la generación de código intermedio y los métodos de optimización que pueden ser aplicados. Si bien durante el desarrollo del programa tuvimos varias complicaciones pudimos resolverlas. Dentro de las principales dificultades que encontramos fue tener que modificar muchas veces el archivo id.g4 ya que los métodos generados no servían y necesitaba de cambios para que sea más fácil de utilizar.

La segunda gran dificultad fue la generación de código intermedio, lo cual pudimos resolver con el uso de la bibliografía entregada en la materia y con documentación/ejemplos que pudimos ir encontrando en internet. Esto nos llevó mucho tiempo, pero no fue un impedimento ya que comenzamos el proyecto con mucho tiempo de anticipación.

Repositorio Github: https://github.com/Santiagospecia/FinalTC-2021.git