Informe Final

Técnicas de Compilación

Docente: Maximiliano A. Eschoyez.

Alumnos: BOZZANO, Felipe Gabriel.
OLMOS BOSSA, Francisco Augusto.

Institución: Universidad Blas Pascal.

Carrera: Ingeniería Informática.

Asignatura: Técnicas de Compilación.

Fecha entrega: 22/07/2022

Técnicas de Compilación
BOZZANO, Felipe
OLMOS, Francisco

Problemática

Resumen

El objetivo de este Trabajo Final es extender las funcionalidades del programa realizado como Trabajo Práctico Nro. 2 . El programa a 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:

- 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. Realizar optimizaciones simples como propagación de constantes y eliminación de operaciones repetidas.

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

1. La tabla de símbolos para todos los contextos.
2. La versión en código de tres direcciones del código fuente.
3. La versión optimizada del código de tres direcciones.

Técnicas de Compilación
BOZZANO, Felipe
OLMOS, Francisco

Solución

Partimos del TP-2, donde tuvimos que redefinir algunas reglas debido a que no podíamos tratarlas correctamente con el Visitor.

El Visitor recorre en profundidad el árbol generado por el Parser, y genera el código intermedio a su paso. Para ello utiliza:

- Una pila de variables temporales asociada a la variable pilaVariablesTemporales, donde almacenamos y quitamos en orden las variables intermedias necesarias para crear el código intermedio.
- Una pila de labels temporales asociada a la variable pilaLabelsTemporales, donde almacenamos y quitamos en orden los labels necesarios para crear el código intermedio.
- Una pila de código, asociada a la variable pilaCodigo, donde agregamos y quitamos variables temporales, labels temporales, y los caracteres del tipo ‘=’, ‘*’, ‘+’, ‘<’, ‘&&’, ‘||’, etc.
- Un mapa asociado a la variable funciones que, dado el nombre de una función devuelve el label asignado a la misma.
- Un generador de variables temporales, asociado a la función generadorNombresTemporales(), que crea variables temporales y las agrega a la pila de variables temporales y a la pila de código. Además, en la pila de código agrega un carácter ‘=’.
- Un generador de labels temporales, asociado a la funcion nuevoLabel(), que devuelve el nombre del siguiente label a crear.

A medida que el Visitor recorre el arbol, encuentra determinadas reglas (nodos) en las que tiene que decidir que acción tomar. Cada regla es un sub-árbol que nosotros aprovechamos para recorrer y obtener la cantidad de nodos del tipo de regla Factor que tienen sus hijos, y a partir de ese dato decidimos si debe generar una variable intermedia o no. El hecho de llegar hasta la regla Factor se debe a que es la única regla que tiene un valor real, ya sea un número o un ID.

La idea de utilizar una pila donde guardamos el código y una en la que guardamos variables temporales, se basa en mostrar los caracteres guardados en la pila de código hasta que el carácter actual sea igual al que tenemos en la pila de variables temporales. De esta manera siempre imprimimos las variables temporales en orden y con su contenido correcto. Sólo las reglas que agregan una variable temporal a la pila son las que las muestran al finalizar su ejecución.Sucede lo mismo con la pila de labels temporales.

Si necesitamos invocar a una función, utilizamos el mapa mencionado anteriormente, el cual nos va a devolver el label que le fue asignado a la función que estamos llamando, para poder saltar a ese label, ejecutar la rutina, y volver con el resultado de la misma.

En el siguiente enlace se puede ver un ejemplo.

Al finalizar, el programa escribe el código intermedio en codigo-intermedio.txt y realiza una optimización sobre el mismo generando el archivo codigo-intermedio-optimizado.txt.

Conclusión

A modo de conclusión, notamos que la implementación del Visitor nos obligó a optimizar las reglas de compiladores.g4. Al tener que hacer búsquedas en árboles en donde cada nodo era una regla, tuvimos que analizar hasta el nodo más chico y eso nos llevó a optimizar las reglas para facilitar este trabajo.

La utilización de la pila nos fue muy conveniente porque nos permitió guardar parte del código sin necesidad de que las reglas se vayan comunicando entre sí.

Implementar un mapa para almacenar las direcciones de memoria de las funciones nos facilitó bastante al momento de generar el código de tres direcciones.

La optimización consigue decrementar la cantidad de líneas de código del código intermedio, y facilitar la lectura del mismo. La implementada en este trabajo consigue evitar el uso de variables intermedias sin sentido, y asignar un valor directo a las variables creadas por el usuario.