UBP UNIVERSIDAD Blas Pascal

Informe Proyecto Final Técnicas de Compilación

Fecha entrega: **14/12/2023**

Docente: Ing. **Maximiliano Andrés Eschoyez**

Integrantes:

- **Bobadilla Barceló, Daniel Agustin**
- **Genaro, Kevin Luis**

Institución: **Universidad Blas Pascal**

Carrera: **Ingeniería en Informática**

Materia: **Técnicas de compilación**

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

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. 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.

Solución

Para resolver este TP Final partimos del TP2 entregado durante la cursada de la materia. Como primera medida modificamos algunas reglas gramaticales en nuestro archivo tp2.g4 para poder cumplir con todos los puntos de la consigna.
También, esto nos obligó a modificar el Listener en archivo MiListener.java para detectar todos los errores que se puedan presentar antes de pasar a elaborar el código intermedio.
Una vez que tuvimos esto ya refinado pasamos a crear clases que nos iban a servir para generar el código de tres direcciones. La más importante y en la que tuvimos que desarrollar el recorrido sobre el árbol generado por el Parser fue la clase MyVisitor.java. Esta clase es la encargada de recorrer este árbol e ir generando código intermedio a su paso.
Dentro de esta clase utilizamos:

- **pilaVariablesTemporales**: pila en donde hacemos pop y push en orden de las variables intermedias que vamos a utilizar.
- **pilaCodigo:** pila en donde hacemos push y pop de variables temporales, labels temporales y caracteres.
- **pilaLabelsTemporales**: pila en donde hacemos pop y push de los labels utilizados en la generación de código intermedio.
- **mapaFunciones**: mapa en donde se almacena Id y el label asociado a esa función.
- **Función generadorVariablesTemporales**: proceso que genera los nombres de las variables temporales que vamos a utilizar.

- **Función generadorLabelsTemporales**: proceso que genera los nombres de los labels temporales que vamos a utilizar.

Nuestro visitor inicializado una vez que verificamos que no existe ningún error crítico con nuestro Listener, procese a recorrer el árbol visitando cada regla (nodo) y su respectivo subárbol determinando que si se encuentran una determinada cantidad de nodos con la regla Factor, procedemos a generar una variable temporal o intermedia.

Para esto último, usamos las dos pilas correspondientes, en la pilaCodigo en donde recorremos hasta encontrar la variable temporal en la pilaVariablesTemporales guardando todo lo que se encuentre antes del matcheo, para luego imprimirlo en nuestro archivo de código intermedio. En la pila de labelsTemporales realizamos algo muy parecido, pero para poder reconocer los distintos labels y saltos dentro de bucles, condicionales o funciones.

Para las funciones, utilizamos el mapaFunciones que nos permite reconocer gracias al label almacenado, a donde tiene que saltar el código ante el llamado de una función.

Por último, una vez que imprimimos el código intermedio en el archivo código-intermedio.txt, gracias al método optimizarCodigo logramos optimizarlo e imprimirlo en el archivo código-intermedio-optimizado.txt. Para entender más este proceso, lo que hicimos fue ir iterando sobre cada línea del archivo y si matcheabamos una variable temporal, saltábamos a la próxima línea para chequear si esa misma variable se asignaba a otra no temporal. Así pudimos eliminar estas variables intermedias e imprimir un código más limpio.

Conclusión

Como mencionamos anteriormente, para este trabajo final tuvimos que optimizar las reglas en nuestro tp2.g4 para poder ir desde los nodos superiores hasta los más chicos en los respectivos subárboles.

En principio intentamos realizar la solución sin utilizar pilas, pero se hacía muy difícil a la hora de generar el código intermedio de una manera correcta, por ende creemos que fue de gran ayuda implementar el visitor de esta manera.

Con respecto al trato de funciones, el mapa nos ayudó mucho a guardar estos labels y a la hora de su llamado detectar fácilmente la dirección a donde continuar para la generación de código.

Por último, el optimizar el código intermedio, nos permitió eliminar variables temporales con operaciones redundantes, para lograr un código final con menos líneas y más limpio.