Diseño de Compiladores I

Trabajo Práctico Especial Partes 3 y 4.

Integrantes:

- Marín, Cynthia Romina (cynthia.romina.marin@gmail.com)
- Sarti, Mauro (msarti@alumnos.exa.unicen.edu.ar)

Ayudante Asignado: Tommasel, Antonela

Número de Grupo: 10

Parte 3 – Generación de código Intermedio

Introducción y Objetivos

El objetivo de este trabajo práctico es la generación del código intermedio para las sentencias ejecutables. En nuestro caso nos tocó la representacion Terceto.
Se deberán generar tercetos para asignaciones, selecciones, sentencias de control asignadas al grupo, y sentencias OUT. En nuestro caso puntual, las sentecias de control asignadas fueron, IF y SWITCH.
En el caso de la sentencias SWITCH se incorporaron los caracteres “{” y “}” como caracteres válidos.

También se necesitó agregar información adicional a la tabla de símbolos como el tipo de dato que retornan las funciones, el significado de un identificador (variable o función), el tipo de cada variable. Al poder definir variables dentro de las funciones tambien nos vimos obligados a establecer ambitos para las variables, de esta forma podemos ver si una variable es alcanzable o no desde un punto específico del programa.

En cuanto a los chequeos semánticos, se verifica el uso de variables o funciones no declaradas, la redeclaracion de variables o funciones, los chequeos de compatibilidad de tipos, en nuestro caso no se permiten conversiones por lo tanto en caso de que los tipos no sean iguales no se permitirá ninguna operación.

Decisiones de Diseño e Implementación

Para este trabajo se decidió hacer una clase Sintáctico, la cual interactúa con las clases ya existentes en el proyecto y con la clase Parser, generada por el Yacc. La clase Parser es la que realiza los chequeos sintácticos recorriendo la gramática realizada y aplicando las distintas acciones semánticas que se asignaron a las reglas. También se realizó la implementación de una clase abstracta Terceto de la cual se extendió para realizar todos los tercetos necesarios, por ejemplo, TercetoAsignacion, TercetoSuma, TercetoResta, etc.

En la clase Sintáctico se encuentra, entre otras cosas, una lista de Tercetos, la cual contiene todos los tercetos en el orden en el que se crearon. Cada Terceto se crea en función de la gramática, por lo tanto, en función del código ingresado.

Junto con la clase Parser, Yacc genera otra clase, ParserVal, que es utilizada para acceder a los elementos reconocidos por la gramática.

La clase terceto tiene tres elementos principales, un operando de tipo String y dos elementos, primero y segundo de tipo String, los cuales representan los operadores del terceto o la posición del terceto al que referencian. También se agregaron dos atributos más, una posición, que representa el número de terceto y un tipo de dato, que representa el tipo de dato del terceto. Este último atributo se utiliza para los tercetos se operaciones como suma, resta, multiplicación y división, pero también para el terceto asignación y comparación ya que se debe revisar que sean del mismo tipo ambos lados de la operación.

Parte 2 – Generación de código Assembler

Introducción y objetivos

Esta cuarta parte del trabajo se trabajó en la generación de código Assembler para Pentium de 32 bits, a partir del código intermedio generado en el práctico anterior.

Como mecanismo de generación de código Assembler utilizamos variables auxiliares.

Los controles en tiempo de ejecucion que implementados son división por cero y overflow en sumas y multiplicaciones.

Decisiones de Diseño e Implementación

Para este trabajo se decidió hacer una clase GenCodigo que es la que se encarga de generar el archivo .asm con el código assembler correspondiente a esa ejecución. La generación del archivo está separada en dos partes, esto se debe a la necesidad de crear variables auxiliares que luego necesitamos que estén definidas en el encabezado del código assembler. En la primera parte se genera el “encabezado”, allí se realizan todos los imports necesarios y se vuelca el contenido de la tabla de símbolos, las variables son definidas según el tipo de contenido que almacenan. Esto se hace luego de haber generado todas las variables auxiliares que se utilizan en el código intermedio.
En la segunda parte se genera el código propiamente dicho, se traduce todo el contenido de los tercetos, el código intermedio, a instrucciones assembler de Pentium 4.
Anteriormente, en el práctico 3 se decidió implementar la clase terceto, de manera abstracta. Esto fue planteado así pensando en la futura generación del código assembler. Los tercetos son definidos como clases separadas donde todas heredan de la clase abstracta Terceto. De esta manera podemos tener un método en cada clase, el método getCodigo() de modo que cada terceto sabe cómo “traducirse” a assembler. Esto método getCodigo(), devuelve un String a quien lo llama (el GenCodigo) y de esta manera vamos formando parcialmente las instrucciones assembler.
Respecto al manejo del Yacc, se utilizó la notación posicional en todo momento, para contener la información que el usuario ingresaba en su código. Se utilizó para saber si una variable o función se encontraba declarada, si en una operación los ámbitos estaban correctos, para pasar el nombre de la función al momento de realizar marcas para la posterior implementación de las etiquetas en el assembler, entre otros.

Los errores que se consideraron en esta etapa fueron las re-declaraciones de variables o funciones, la no declaración de variables o funciones, las divisiones por cero y el overflow en sumas y multiplicaciones.
Con respecto a las funciones y funciones MOVE (la función MOVE solo permite utilizar variables propias) el manejo de ámbitos consistió en asignarle a cada variable, constante o función el ámbito en el que fue creada. Si se entraba a una función aumentaba el ámbito (se le concatenaba ":X", donde X es la letra que le sigue al ámbito actual) y se decrementaba cuando se salía de la misma (quitándole los últimos dos caracteres). La mayor letra de ámbito se guarda en una variable para no repetir ámbitos.

Por ejemplo, una variable global tenía el ámbito "A", una variable dentro de una función ubicada en el programa general tenía un ámbito "A:B". Si se declaraba otra función, al mismo nivel que la función anterior, tendría un ámbito "A:C".

Conclusiones

A lo largo del trabajo pudimos ver cómo, etapa a etapa, se desarrolla un compilador, desde la definición de la gramática hasta el reconocimiento de las mismas y la generación de código assembler. Comenzamos solo con algunas reglas gramaticales, caracteres que permite el lenguaje, el reconocimiento de tokens, y poco a poco se le fue dando un contexto a todo el trabajo anterior, chequeos semánticos, sintácticos, generación de código intermedio, etc. En su conjunto se logró llegar a lo que finalmente sería un compilador completo.

Hubo varios inconvenientes a lo largo del proceso de desarrollo, algunas cuestiones surgieron por falta de entendimiento y otras por problemas de implementación de código. Cabe destacar que por parte de los docentes y ayudantes de la materia la predisposición a evacuar nuestras dudas fue siempre la mejor, todos disponibles y dispuestos a contestar preguntas y aclarar dudas.

En nuestra opinión personal consideramos muy interesante saber cómo funcionan más a bajo nivel las herramientas que usamos constantemente en nuestro día a día.