Universidad Nacional del Centro de la Provincia de Buenos Aires.

Trabajo Práctico

Diseño de Compiladores

Generación de Código Intermedio y Assembler

Re-entrega

Grupo 8

INTEGRANTES:	CORREO:
Chiozza, Juan Ignacio	chiozzajuani@gmail.com
González, Cristian	cris_mdq22@hotmail.com
Rodríguez, Facundo Hernán	farodriguez@alumnos.exa.unicen.edu.ar

Contenido

INTRODUCCION

A lo largo del presente trabajo se trabajará con la generación de código intermedio del compilador y la traducción de este a Assembler. En la generación de código se usará la representación en Tercetos.

El mecanismo de generación de Código Assembler, será mediante creación de variables auxiliares. Además, se controlará distintas situaciones en tiempo de ejecución:

- División por cero,
- Overflow en Productos,
- Chequeo de subíndices en referencias a elementos de colección.

GENERACION DE CODIGO

DECISIONES DE DISEÑO

Para la implementación de los tercetos, se creó una clase Terceto abstracta general de la cual heredan los tercetos correspondientes de Expresión, Asignación, Comparación, Terceto de Colecciones, Terceto IF, Print y Label.

A su vez, las clases ExpresionDiv y ExpresionMult heredan de la clase Expresion que se encargan de la división y multiplicación respectivamente, ya que el terceto Expresión se encarga de la suma y resta. De esta manera, se logra una mejor abstracción entre los tercetos, ya que en los tercetos de multiplicación y división se deben hacer controles adicionales únicos a los mismos (Overflow en productos y división por cero).

También se añadió el atributo Tipo_Uso a ciertas entradas de la tabla de símbolos, particularmente en los casos de variables y colecciones, denotando el tipo de datos sobre el cual fueron declaradas, sumamente útil para la comprobación de operaciones sobre tipos.

Cada clase que hereda de Terceto debe implementar el método abstracto getAssembler(), para que cada uno de los tercetos puedan generar el código assembler correspondiente.

GENERACION DE TERCETOS

Para llevar a cabo la generación de código intermedio se implementaron diferentes tipos de tercetos, de acuerdo con la funcionalidad requerida:

CLASE	FORMATO	DESCRIPCIÓN
Terceto Multiplicación	(*, operando1, operando2)	Los operandos pueden ser una constante, variable u otro terceto (expresiones)
Terceto División	(/, operando1, operando2)	Ídem anterior.
Terceto Expresión	(+, operando1, operando2) (-, operando1, operando2)	Operandos con mismas características que los de la multiplicación y división. Se unificó la suma y resta porque contienen un formato de assembler muy similar.
Terceto Comparación	(comparador, operador1, operador2)	Los operandos contienen las características ya mencionadas. No permite operadores de distinto tipo.
Terceto Asignación	(:=, lado izquierdo, expresión)	lado izquierdo se ve limitado a una variable y lado derecho a una expresión.
Terceto IF	(BF, Condicion, Destino)	Similar al terceto DO pero con salto por falso a una posición posterior al terceto actual. Es utilizado luego de

		evaluar una condición y permite generar el assembler para saltar un bloque en caso de branch por falso, o saltar incondicionalmente luego de ejecutar el bloque correcto en caso de poseer algún Else en la bifurcación. También se es reusado para el bloque do – until, pero con solo salto por falso a un terceto previo a la condición.
Terceto Print	(print, valor, -)	El primer elemento del terceto tiene la palabra reservada print, mientras que valor se refiere al contenido que se quiere mostrar por pantalla, pudiendo ser el mismo una cadena multilínea, variable o colección indexada.
Terceto Label	(-,-,-)	No contiene elementos. Fue creado para la inserción del label correspondiente al salto del Do al momento de implementar el mismo en assembler.
Terceto Colección Índice	(CGI, ID, subíndice)	El primer elemento del terceto indica que es un índice de la colección, el segundo indica el identificador de la colección y el tercero el subíndice de esta que se quiera extraer.
Terceto Colección Length	(CLE, Colección, -)	El primer elemento del terceto indica que es un método de la colección, y el segundo representa a la colección a la cual se aplica el método.
Terceto Colección First	(CFI, Colección, -)	Ídem anterior.
Terceto Colección Last	(CLA, colección, -)	Ídem anterior.
Terceto Colección Asignación	(:=, lado izquierdo, expresión)	Es utilizado para asignaciones en la cual intervienen colecciones, específicamente el mecanismo de rowing para las mismas.

Cada estructura mencionada tiene su correspondiente traducción a código assembler.

Algunas referencias importantes para destacar son las siguientes:

1 (:=, a, 16)

2 (/, a, 8)

3 (/, [2], -1)

4 (:=, c, [3])

En los tercetos anteriores expresados como ejemplos podemos ver que las variables y constantes son expresadas por su nombre en la tabla de símbolos. Por otro lado, los números encerrados entre corchetes (como por ejemplo en el terceto 3) hacen referencia a otro terceto (en este caso al resultado de la división detallada en el terceto 2).

Dicho ejemplo surge de las siguientes sentencias en el lenguaje implementado:

a := 16;

c := a /8 /-1;

USO DE NOTACION POSICIONAL DE YACC

Yacc provee dos tipo de notación posicional ($$ y $n). Su principal uso fue aquellas tareas pertenecientes a la generación de código.

Principalmente, fue muy utilizado para crear los tercetos, ya que la notación $n nos permitía obtener la referencia a la tabla de símbolos del terminal/ no terminal que se encontraba en la posición n de la regla. En cambio, la notación $$ fue de utilidad para asignar dicha referencia en el no terminal y luego poderlo utilizar como operando en otras reglas.

En segundo lugar, fue utilizado para los chequeos semánticos como, por ejemplo, determinar si una variable ya fue declarada o si se quiere utilizar sin haberla declarado previamente, o si se quiere utilizar el mismo nombre en una variable que en una colección o viceversa.

ERRORES DE GENERACION DE CODIGO

Seguidamente, se presentarán los errores identificados en la generación de código:

- errorNoExisteVariable: Esta variable no fue declarada. Se da cuando el programador utiliza una variable que no fue declarada con anterioridad.
- errorVariableRedeclarada: Ya se declaró una variable con este nombre. Se produce cuando se encuentra otra variable con el mismo nombre en la tabla de símbolos.
- errorFaltaIndiceColeccion: Este error se produce cuando no se declara explícitamente el tamaño de la colección.

- errorColeccionPorVariable: Este error se produce cuando se pretende usar una colección como variable.
- errorVariablePorColeccion: Opuesto al anterior. Se pretende usar una variable como colección.
- errorTiposDiferentes: Se produce cuando se realizan operaciones con tipos diferentes.

ESTRUCTURAS DE CONTROL

Para llevar a cabo las estructuras del control, en nuestro caso las estructuras IF y DO, se utilizaron reglas diferentes para las mismas, pero reutilizando en ambos casos el terceto IF para su creación. La diferencia radica en que Do utiliza auxiliarmente un terceto label para la posterior ubicación del label correspondiente al salto al inicio del bloque, además de apilar y desapilarse de manera diferente al IF, mientras que los labels necesarios para el IF siempre se insertarán posteriormente al IF en cuestión.

De forma auxiliar se utiliza una pila que nos permite almacenar los índices de los tercetos donde se indica la bifurcación, de esta manera una vez identificada la dirección de salto se desapila el valor del índice del terceto y se le modifica la dirección de salto en la misma pasada de código.

Particularmente para el caso del IF, en cuanto se detecte el inicio de la sentencia junto a la condición asociada, se apilara el terceto en dicha pila. La misma se desapilará al encontrar el fin del bloque IF, o el inicio del bloque ELSE en caso de existir, indicando la dirección de salto para el branch por falso. En caso de tratarse de un ELSE, se apilará de nuevo un terceto de tipo salto incondicional, el cual se desapilará al finalizar el bloque IF, indicando la dirección de salto en el caso de que haya entrado en el primer bloque, con tal de no entrar en el segundo.

Para el caso del DO, se utilizó un apilado y desapilado diferente, el cual se apila al comienzo de la sentencia, y desapila al encontrar la condición, generando un branch por falso con dirección de destino hacia el inicio del bloque.

A su vez, es necesario indicar al terceto que tipo de salto es utilizado dentro de la estructura para su correcta ejecución, junto con un indicador al terceto condición asociado (en caso de branch por falso).

ERRORES DE EJECUCION

Dentro de los errores de ejecución contemplados para el compilador diseñado se tuvieron en cuenta:

- Chequeo de subíndices en referencias a elementos de colección
- División por cero
- Overflow en Productos

Con respecto a los chequeos de subíndices en las colecciones, se creó el assembler mediante el uso de comparaciones entre el índice a buscar y el largo asociado a la colección para saber si se accede a una posición válida de la misma o no, y chequeo del índice y 0 para saber si no se trata de un índice negativo.

Para evitar que el divisor sea cero, se verificó antes de hacer la división mediante una comparación entre cero y el divisor en cuestión. De ser verdadero, se mostrará un mensaje informando el error y el programa finalizará.

Por otro lado, para el chequeo de Overflow en productos entre operandos de tipo entero, se utilizó el flag OF. En caso de que el valor de este sea 1 se abortará la ejecución del programa y se mostrará el mensaje por pantalla informando el error. Para el caso de producto entre operandos de tipo Double, se creó una función auxiliar (con el fin de no insertar todo el código asociado cada vez que se produzca una multiplicación entre Doubles), la cual en primera instancia chequea si el producto es 0 (ya que posteriormente se hará una comprobación de que el producto no es menor al límite menor positivo, lo cual debe excluir solamente al 0 en caso de serlo). Si lo es, salta al final del bloque sin errores, y limpia la pila de registros del coprocesador x87, siendo entonces el 0 un resultado válido. Caso contrario, se debe chequear que se encuentre entre los límites impuestos a los Doubles, siendo los mismos 2 limites positivos y 2 limites negativos opuestos. Para ello, se verificó que el valor absoluto del producto en cuestión se encuentre entre el limite mayor y menor positivo, en caso de no serlo se termina la ejecución y se informa del error.

GENERACION DE CODIGO ASSEMBLER

Los mecanismos que utilizamos para la generación de código assembler se basaron en la traducción de la información almacenada en cada terceto de manera de generar a partir del mismo, con la ayuda del mecanismo de variables auxiliares para guardar valores intermedios, las instrucciones requeridas. Para el caso de sentencias de tipo Int, se utilizó siempre los 16 bits menos significativos de los registros EAX, ECX y EDX, junto a las correspondientes variables declaradas de tipo DW necesarias. Para el caso de Double, se utilizó la pila, junto a las sentencias asociadas, del coprocesador 80x87, junto a variables QWORD necesarias, siempre liberando posteriormente la pila de registros al final de cada operación.

Particularmente, se utilizaron las directivas MOV, ADD, SUB, IMUL, IDIV, MOVZX, CMP y todas las condiciones de salto asociadas para las operaciones con enteros.

Para el caso de operaciones con Double, se utilizo FLD, FADD, FSUB, FSTP, FST, FCOMPP, FDIV, FTST, FSTSW (junto a SAHF para guardar los flags en el registro AH) y FMUL.

DECISIONES DE DISEÑO PARA LA TRADUCCION A ASSEMBLER

Para la traducción del código intermedio a Assembler se tuvieron en cuenta ciertas decisiones sobre cómo implementar el funcionamiento, siempre teniendo en cuenta los errores a contemplar, y la diferencia de operaciones entre datos de tipo Int y Double.

En primer medida se puede contemplar la adición de un cierto campo a la tabla de símbolos, el campo Dirección, con el fin de proveer mayor información sobre los datos que manejan los tercetos, y así poder realizar la conversión de manera eficiente y efectiva. Este campo Dirección se usó para los tercetos de tipo ColecciónGetIndice específicamente, con el fin de poder declarar en la sección de variables del assembler datos con la forma @aux y numero de terceto, de tipo DWORD, ya que apuntarán a direcciones de memoria de 32 bits. Esta información añadida fue de suma utilidad para toda operación que pudiese tener como operando un elemento de una colección, entre otros, pudiendo así acceder de manera efectiva a la información requerida en caso de tratarse del mismo. A su vez, esto nos permitió el uso de elementos de colección como operando del lado izquierdo en asignaciones.

Cabe destacar también las funciones que se declaran dentro del assembler, particularmente las correspondientes a los métodos de colecciones. Las mismas trabajan con un pasaje de parámetro mediante variables auxiliares, las cuales son declaradas en la sección de variables. Las mismas son:

- @param de tipo DWORD, ya que todas las funciones trabajan con la dirección de memoria de la primera posición del arreglo.
- @retI de tipo DW, utilizado para el retorno de todos los métodos sobre colecciones de tipo Int, asi como el método Length para colecciones de tipo Double.
- @retD de tipo QWORD, el cual se utiliza para el almacenamiento del retorno de las funciones First y Last sobre colecciones de tipo Double.

Es necesario también aclarar cómo se implementaron las colecciones en sí, siendo las mismas de largo igual al declarado más uno. Esto se debe a que el largo de las mismas se almacena en la primera posición del arreglo, en formato Int o Double según corresponda. Existen dos funciones para cada tipo de método, siendo una para colecciones de tipo Int y otra para Double. El método length simplemente devuelve el valor de esta primera posición, convirtiendo a Int en caso de ser necesario. Luego el método First simplemente devuelve la posición siguiente a la primera, utilizando un offset de 2 para colecciones de tipo Int y 8 para Double. Por último, para el método Last, se procedió a utilizar el método Length, multiplicar el valor devuelto por el offset correspondiente y sumarlo al valor guardado en @param, extrayendo finalmente el valor de la última celda de la colección.

Adicionalmente, se agregó la funcionalidad necesaria para poder imprimir tanto cadenas multilínea, como variables y elementos de colecciones. El diseño de la misma recae en la inserción de comandos “invoke MessageBox” para cadenas, y comandos “invoke printf” con argumento “d” o “f” según el tipo de dato a imprimir.

REENTREGA

Se corrigieron distintos erros y consideraciones particulares que se encontraron en la primera entrega:

- Correcta detección de errores de overflow en multiplicación, particularmente en Doubles.
- Incorporación del mecanismo utilizado para los métodos de colecciones.
- Correcta indirección de elementos de las colecciones.
- Corrección de la declaración de variables en assembler, particularmente las constantes Doubles.
- Se corrigieron los saltos para operaciones de tipo Double.
- Comparaciones de elementos de tipo Double arreglada.
- Nombre y ubicación del archivo .asm generado corregida.
- Correcta impresión de enteros negativos.
- Mayores controles en la etapa de análisis sintáctico.

CONCLUSIONES

Mediante la implementación de la etapa de generación de código intermedio se logró entender la importancia de este ya que no solo permite una mejora a la hora de abstraer el código fuente, sino que también reduce la complejidad en la traducción al código assembler, pudiendo realizar ciertos chequeos necesarios ya en esta etapa.

Por otro lado, la implementación a nivel assembler permite entender con mayor profundidad los mecanismos utilizados por un compilador, asi como el funcionamiento del lenguaje assembler en si, junto con todas las consideraciones que hay que tener a la hora de implementar algo de nivel tan bajo.

De esta manera, entender estos aspectos del compilador y del lenguaje assembler, ayuda a limitar las situaciones descritas anteriormente y lograr entonces mejores prácticas de programación y comprensión sobre el funcionamiento interno de la computadora.