Trabajo Práctico de Cursada
Segunda Entrega

Diseño de Compiladores

Grupo Número 14

Bianco, Martin (martinbianco995@gmail.com)

Di Pietro, Esteban (dipietroesteban@gmail.com)

Serrano, Francisco (francisco.serrano372@gmail.com)

Ayudante asignado: Antonela Tommasel

Introducción

En el siguiente informe se detallaran las decisiones tomadas para llevar a cabo la generación de código intermedio para todas las sentencias ejecutables teniendo en cuenta criterios establecidos por la cátedra. Las sentencias a considerar son asignaciones, selecciones, sentencias de control asignadas, sentencias OUT y funciones.

Con respecto al trabajo práctico número 4, se describirán las acciones llevadas a cabo para lograr generar código assembler a partir de la base del código intermedio construido previamente. Se explicaran los mecanismos utilizados para realizar cada una de las operaciones aritméticas, la generación de etiquetas, entre otras funciones implementadas en el trabajo.

También se hará referencia a las decisiones tomadas para realizar la detección e informe de chequeos semánticos y errores en tiempos de ejecución.

Temas particulares asignados

A continuación, se detallan los temas asignados por la cátedra para la Generación de código intermedio:

- Tercetos para sentencia WHILE-DO, IF, IF-ELSE, ASIGNACIONES, EXPRESIONES, sentencias OUT.
- En tema 13 (Funciones/Funciones Move) tener en cuenta que cuando se declara la función con Move, solo se podrán usar variables del ámbito local, y no se podrán usar variables de ámbito global.
- En tema 16: para el chequeo de compatibilidad de tipos, este tema es sin conversiones de ningún tipo.

En cuanto al mecanismo de generación de código assembler, el tema asignado fue seguimiento de registros. Los controles en tiempo de ejecución considerados fueron los siguientes:

- División por cero
- Overflow en productos

Decisiones de Diseño e Implementación

Para poder desarrollar de una mejor manera esta entrega, lo primero que se hizo fue una corrección bastante importante sobre la gramática para evitar complicaciones futuras. En el siguiente cuadro se exponen las listas de reglas de la gramática antigua contra la gramática nueva:

Gramática vieja	Gramática nueva

programa := bloque;
 	
declaracion := declaracion_variables | declaracion_funcion;
 	
declaracion_variables := lista_variables COLON tipo DOT;
 	
lista_variables := ID COMMA lista_variables| ID;
 	
tipo := UINT | ULONG;
 	
declaracion_funcion := funcion;
 	
funcion := tipo FUNCTION ID cuerpo_funcion | tipo MOVE FUNCTION ID cuerpo_funcion;
 	
cuerpo_funcion := OPEN_BRACE bloque_funcion RETURN OPEN_PAR expresion CLOSE_PAR DOT CLOSE_BRACE | OPEN_BRACE RETURN OPEN_PAR expresion CLOSE_PAR DOT CLOSE_BRACE;
 	
sentencias := sentencia sentencias | sentencia;
 	
sentencia := asignacion | ejecutable | control;
 	
ejecutable := IF OPEN_PAR condicion CLOSE_PAR THEN bloque_control ELSE bloque_control END_IF | IF OPEN_PAR condicion CLOSE_PAR THEN bloque_control END_IF | OUT OPEN_PAR CADENA CLOSE_PAR DOT;
 	
asignacion := ID ASIGN expresion DOT;
 	
bloque_funcion := sentencias | declaracion_variables bloque_funcion;
	programa := sentencias;
	
	declaracion := declaracion_funcion | declaracion_variables;
	
	declaracion_variables := lista_var COLON tipo DOT;
	
	lista_var := lista_var COMMA ID | ID;
	
	tipo := ULONG | UINT;
	
	declaracion_funcion := tipo FUNCTION ID cuerpo_funcion | tipo MOVE FUNCTION ID cuerpo_funcion;
	
	cuerpo_funcion := OPEN_BRACE bloque_funcion RETURN OPEN_PAR expresion CLOSE_PAR DOT CLOSE_BRACE | OPEN_BRACE RETURN OPEN_PAR expresion CLOSE_PAR DOT CLOSE_BRACE;
	
	sentencias := sentencias sentencia | sentencia;
	
	sentencia := asignacion | print | seleccion | iteracion | declaracion;
	
	print := OUT OPEN_PAR CADENA CLOSE_PAR DOT;
	
	asignacion := ID ASIGN expresion DOT;
	
	bloque_funcion := bloque_funcion bloque | bloque;
	
	bloque := declaracion_variables | asignacion | print | seleccion | iteracion;
	
	seleccion := seleccion_simple else bloque_sentencias END_IF | seleccion_simple END_IF;

condicion := expresion comparador expresion;
 	
bloque := sentencia_bloque bloque | sentencia_bloque;
 	
bloque_control := bloque_simple | bloque_compuesto;
 	
bloque_compuesto := BEGIN sentencias END;
 	
bloque_simple := sentencia;
 	
control := WHILE OPEN_PAR condicion CLOSE_PAR DO bloque_control;
 	
expresion := expresion ADD termino | expresion SUB termino | termino;
 	
termino := termino MULT factor | termino DIV factor | factor;
 	
factor := ID | CTE |invocacion_funcion;
 	
invocacion_funcion := ID OPEN_PAR CLOSE_PAR;
 	
comparador := LEQ | GEQ | LT | GT | EQ | NEQ;
	else := ELSE;
	
	seleccion_simple := IF condicion_if THEN bloque_sentencias;
	
	condicion_if := condicion;
	
	condicion := OPEN_PAR expresion comparador expresion CLOSE_PAR;
	
	bloque_sentencias := bloque_simple | BEGIN bloque_compuesto END;
	
	bloque_simple := asignacion | seleccion | iteracion | print;
	
	bloque_compuesto := bloque_compuesto bloque_simple | bloque_simple;
	
	iteracion := while condicion_while DO bloque_sentencias;
	
	while := WHILE;
	
	condicion_while := condicion;
	
	expresion := expresion ADD termino | expresion SUB termino | termino;
	
	termino := termino MULT factor | termino DIV factor | factor;
	
	factor := ID | CTE |invocacion_funcion;
	
	invocacion_funcion := ID OPEN_PAR CLOSE_PAR;
	
	comparador := LEQ | GEQ | LT | GT | EQ | NEQ;

Además de las correcciones a la gramática, se agregaron algunos atributos a los Tokens que se guardan en la Tabla de Símbolos. Los atributos agregados fueron los siguientes:

- Uso
- Ámbito

El primero de ellos se usa para diferenciar entre los identificadores que corresponden a un nombre de función y los que corresponden a nombres de variables. El segundo atributo sirve para identificar en qué ámbito fueron declarados los identificadores. Esta última diferenciación nos ayuda a resolver el problema de los ámbitos de funciones.

Para implementar los tercetos, se tomó la decisión de crear algunas clases nuevas. Esto fue debido a que los tercetos tienen un patrón en el cual el primer lugar del terceto corresponde siempre a un operador, y el segundo y tercer lugar pueden ser otros tercetos o identificadores/ constantes/invocaciones de función. Por ello se crearon las clases Terceto, Item (clase interfaz), ítemTerceto e ítemString. El itemTerceto se utiliza cuando uno de los argumentos es un terceto anterior, mientras que itemString se utiliza si el argumento es un identificador o constante o invocación a función. La clase Terceto es el objeto propiamente dicho.

Para implementar la funcionalidad requerida para la generación de código intermedio, en la gramática se utilizó el mecanismo que ofrece Yacc para retornar valores de una acción específica ($$). Más concretamente, se instancio el $$.obj con objetos de las clases que heredan de la interfaz Item. Con esto lo que se logró es simular el pasaje de punteros que tiene lugar en la creación de tercetos a partir de una lista de reglas. La creación de nuevos tercetos ocurre durante asignaciones y expresiones de suma, resta, multiplicacion, o division. A esto último se le agregó además la creación de tercetos en invocaciones a funciones, declaraciones de funciones, sentencias de Return en funciones, sentencias out, condiciones tanto de if como de whiles para indicar saltos, sentencias else e inicios de sentencias while también para indicar saltos.

Con respecto a los últimos tercetos mencionados, a continuación se detallarán aquellos que hacen referencia a las sentencias if y while. En ambas sentencias se utilizó una pila para luego poder completar los tercetos de salto que en su creación quedan incompletos. Se utilizaron dos operaciones distintas de salto para los tercetos, una BI que significa salto incondicional, y otra BF que es un salto en caso de que la condición sea falsa.

En el caso del if, el salto BF se utiliza bien para saltar al final de la sentencia, en caso de que la misma sea simple (sin bloque else), o bien para saltar directamente a la rama del else luego de la condición. Para poder completar el terceto BF, dado que al ser un salto todavía no se tiene determinado a que terceto se debe ir, se deja incompleto anotando el número de terceto en la pila antes mencionada hasta que se termina de reducir la regla y se llega al END_IF, o bien hasta que se entre en la rama del ELSE, según sea el caso. En cuanto al salto BI, solo se hace en los ifs que tienen bloque ELSE. En este caso, el terceto BI es creado cuando se lee el ELSE, y luego es completado al llegar al END_IF y haber reducido la sentencia if. Como en el caso anterior, el número de terceto BI queda en la pila para cuando se llega al final de la sentencia se busca en la lista de tercetos el número que indica la pila y se completa.

En el caso del while, lo que primero se hace cuando se lee el token WHILE es poner en la pila el número del siguiente terceto, ya que sabemos que ese siguiente terceto va a ser parte de la condición del while y va a ser el terceto al que va a tener que saltar el BI más tarde. Luego de hacer esto y crear los tercetos relacionados con la condición del while, se crea el terceto BF incompleto, y se apila el número de este nuevo terceto. Cuando se terminan de reducir las reglas de la sentencia while y se llega al final ocurren varias cosas. La primera de ellas es que se completa el terceto BF, cuyo número se tiene guardado en el tope de la pila, y se completa con el número del siguiente terceto más uno, ya que ese terceto estará fuera de la sentencia while. La segunda cosa que pasa es que se crea el BI, y se completa en este mismo lugar con el número de terceto que indica el tope de pila. Es decir, para cuando llega el final de la sentencia

WHILE, la pila tiene dos valores, el tope que indica el terceto que hay que completar(BF), y el otro valor que indica a que terceto tiene que hacerse el salto del terceto BI.

La estructura utilizada para el almacenamiento del código intermedio es un ArrayList de la clase Terceto. En el mismo se insertan los tercetos a medida que se generan permitiendo en el futuro poder acceder a cada uno de ellos de manera sencilla. También se decidió implementar una clase CustomStack cuyo objetivo consiste en llevar un registro de los ámbitos que poseen cada una de las variables definidas. Su sencillo funcionamiento se basa en agregar a la pila el ámbito en el momento que se inicia la declaración de una función y retirarlo al finalizar la misma.

Para la salida del compilador en la Generación de código se hizo lo siguiente:

numero de terceto. (*operador, argumento1, argumento2*) *tipo terceto*

Ejemplo:

20. (/, var2@main, 5) UINT

21. (+, var1@main, [20]) UINT

22. (=, var3@main, [21]) NULL

Cada terceto se le asigna un número y se lo muestra para que se más fácil leer el código. Además, se puede apreciar que los números que aparecen entre corchetes son los correspondientes a un terceto anterior, mientras que los demás argumentos son variables o constantes. Como se dijo antes, siempre los primeros argumentos son los operadores del terceto.

Para llevar a cabo la generación de código assembler se desarrollaron dos clases nuevas. La primera, TablaRegistros permite manejar el uso de los registros disponibles para realizar las operaciones. Se trata de un arreglo de booleanos que mediante las funciones occupyRegister y freeRegister, se ocupan y liberan respectivamente los registros a medida que se van utilizando. También se definió una función en dicha clase, que retorna un registro libre cada vez que se la invoca, y en caso de no tener uno libre se retorna una constante con un valor inválido.

En la segunda clase, Generador, se genera el código assembler a partir de los tercetos obtenidos previamente en la gramática. Esta clase posee, además de los tercetos, la tabla de símbolos y los registros con los que se van a generar las instrucciones. El código que se va generando es almacenado en una variable del tipo StringBuilder. Inicia con la incorporación de librerías y continúa con la declaración de las variables que se encuentran almacenadas en la tabla de símbolos, ya sean constantes, variables o cadenas. Luego se prosigue con la lectura del arreglo que contiene a los tercetos generando el código correspondiente para cada uno de ellos. Para cada terceto se analiza el operando, distinguiendo entre asignaciones, sentencias out, comparaciones, funciones, entre otros, y a partir de eso se obtiene el assembler correspondiente. Algo a resaltar son el trabajo que se hizo con las operaciones matemáticas, donde se debió agregar comportamiento para considerar los diferentes casos de operandos, ya que podían aparecer tanto registros como variables/constantes.

Otras operaciones a las que se les puso especial atención fueron las de salto (ya sea que se trate de BI o BF) donde se optó por utilizar una lista nueva para almacenar los índices

de los tercetos a los cuales realizar el salto. Una vez realizado el código para todos los tercetos, se realiza una segunda pasada incorporando por cada uno de los índices que aparece en la lista el código label correspondiente.

Para la generación de código assembler de las funciones, se procedió a ir almacenando la traducción de manera independiente del código del main pero utilizando el mismo formato de estructura. El mecanismo consiste en que cada vez que se encuentra con el terceto que posee el operando “FUNCTION” se indica en un boolean que los tercetos a continuación van a ser propios de la función. Por lo tanto ante cada nueva operación que se trabaja, se contempla el boolean mencionado anteriormente para analizar si la operación pertenece al contenido de una función o simplemente es código del main. Cuando finaliza el contenido de la función, con un return, se vuelve a cambiar el boolean indicando la salida de la definición y se agrega el código assembler correspondiente dependiendo del tipo de dato que se deba retornar.

Para poder identificar un llamado a una función determinada previamente definida, se debe reconocer el operador “CALL” en un terceto. En caso de que esta situación ocurra se procede a almacenar el contenido de los cuatro registros principales a un conjunto de variables auxiliares. El conjunto donde se va a disponer el almacenamiento va a depender del tipo de dato que retorna la función correspondiente, ya sea de formato ULONG o UINT.

Errores Semánticos

Los errores semánticos considerados fueron los siguientes:

- Re declaración de variables dentro de un mismo ámbito.
- Variables no definidas dentro de un ámbito en particular.
- Re declaración de funciones.
- Llamados a funciones que no fueron declaradas.
- Nombres de funciones como operando de expresiones sin los paréntesis que indiquen su correspondiente invocación.
- Asignaciones y comparaciones entre expresiones de distinto tipo.
- Retornos de funciones que sean de distinto tipo que la declaración de la función.

Errores en tiempo de compilación

Los errores que se consideraron para que se controlen en tiempo de ejecución fueron los siguientes:

- Overflow en productos.
- División por 0.

Conclusión

Para dar cierre al informe podemos concluir que con la realización de este trabajo pudimos aprender nuevas tecnologías como lo fue YACC y MASM32, además de estudiar el proceso de programación que se lleva a cabo para construir un compilador y todas las difíciles decisiones de implementación que hay que tomar por el camino.