Trabajo Práctico Nº 3 y 4

Generación de código intermedio y assembler

Diseño de Compiladores I

Docente asignado
Nicolás Dazeo

Grupo 6
Maria Carolina Diaz
Ezequiel Carbajo
José Noguera

Introducción

El siguiente informe se enfocará en detallar aspectos generales y en concreto de las decisiones tomadas para el desarrollo de un compilador con las especificaciones solicitadas por la cátedra de Diseño de Compiladores I de la facultad de UNICEN Tandil en el año 2019.

Desarrollo

Esta segunda parte del trabajo de cátedra continúa con las siguientes fases de la compilación, con la gramática permitida por el compilador como entrada. A su vez, se divide en dos, la primera es la generacion de codigo intermedio, en la cual se utiliza una estructura propuesta para contener las sentencias reconocidas por la gramática del compilador y esto llevarlo a la segunda división, que es la de leer esta estructura para traducir a código assembler para que pueda ser reconocido por la máquina.

Para la primer parte del trabajo, se tuvo que corregir la gramática para agregar los chequeos semánticos que no fueron considerados en la primer parte de este trabajo y así pueda reconocer chequeos de tipos, conversiones, entre otros. Además, de generar el código intermedio específico para luego poder ser traducido a código assembler.

Información contenida en la tabla de símbolos

- Número de token (*value*): El número asignado por el parser para reconocer cada token como un terminal en específico.
- Tipo de Token (*tokenClass*): Un string indicando si el token se trata de una palabra reservada, un ID o una constante.
- Tipo de atributo (*variableType*): Un string indicando el tipo del que será la variable, ya sea INT o LONG. Esto se define en la etapa sintáctica con la gramática.
- *variableRepetitions*: Indica la cantidad de veces que la constante aparece en el código. Esto sirve para casos en los que aparecen constantes negativas (ej: -1) sin su opuesto (en este caso 1). En esos casos la gramática deberá detectar que la constante positiva detectada en el léxico no tiene apariciones, por ende debe eliminarse de la tabla de símbolos.
- *declared*: Indica si fue declarada la variable (al terminar la compilación todas las variables deberían tener este casillero en verdadero).
- Estructura usada (*identifierClass*): Indica si es una variable o una colección, en caso de detectarse como un ID en la etapa sintáctica.
- Tamaño de arreglo (*cSize*): Indica el tamaño que tendrá la colección en base a su declaración.
- Tamaño de arreglo en bytes (*cSizeBytes*): Indica el tamaño que tendrá la colección en bytes, multiplicando la cantidad de elementos que contiene con el tamaño de los tipos que contiene.
- Elementos de la colección (*elemsColection*): Almacena los datos que tendrá la colección, separados por “,”.
- “*idCounter*”: Un atributo entero que permite generar identificadores distintos para acceder a cada cadena.
- “*idCadena*”: representa el id asignado para el token cadena.

- "*isPointer*": Este atributo booleano será verdadero si el dato que almacena es un valor o una dirección de memoria (usado por variables de iteración en el FOREACH).
- "*foreachControlCounter*": Un entero que variará su valor a medida que se recorre el árbol sintáctico para la generación de assembler. Permite identificar si la variable está dentro de un foreach (o múltiples foreach) donde se lo utiliza como puntero a una parte del arreglo. En caso tal, se le restringe la posibilidad de asignarle valor. En caso contrario, funcionará como una variable común de su tipo.

Decisiones de diseño e implementación del código intermedio

Para la generación de código intermedio, se empleó el uso de un árbol sintáctico, el cual genera sus nodos a medida que va resolviendo la gramática encontrada.

Usando la notación posicional de Yacc, se fue creando nodos desde las hojas representando terminales y no terminales clave, los cuales fueron siendo devueltos a los no terminales superiores para conectarlos con los nuevos que se generen. De esta forma se generan todos los nodos desde las hojas hasta la raíz, la cual se devolverá para ser resuelta y generar el código assembler.

Para sentencias especiales como las que contienen el comando "IF" o "FOREACH" se requiere un método que permita indicar que existen posibles saltos, aunque aún no se sepa a donde. Para resolver esto con el desarrollo del árbol sintáctico, se empleó el uso de nodos especiales de control, los cuales tienen un solo hijo y una funcionalidad especial a la hora de generar el código assembler que agregue instrucciones de salto donde corresponda.

Finalmente, para mostrar una representación del árbol como salida del compilador, se aprovechó el recorrido realizado por el compilador sobre el árbol para imprimir su contenido en preorden, agregando una tabulación con cada bajada de nivel en el mismo.

Pseudocódigo de bifurcación de IF:

- Se genera el subárbol representando el cuerpo de la condición.
- Se genera el subárbol representando el cuerpo "*Then*".
- En caso de existir, se genera el subárbol representando el cuerpo "*Else*".
- Se crean nodos de control como padres de cada subárbol creado (*Cond*,*Then*,*Else*).
- En caso de no haber *Else*, se crea el nodo común "*If*" y se le define *Cond* como hijo izquierdo y *Then* como nodo derecho.
- En caso de haber *Else*, se crea un nodo común "*Cuerpo*" que conecte *Then* y *Else* y se crea el nodo común "*If*" que conecte *Cond* y "*Cuerpo*".

Pseudocódigo de bifurcación de FOREACH:

- Se generan múltiples veces nodos hoja que representen ambos parámetros del FOREACH: la variable que itera (la llamaremos "*i*" para el ejemplo) y el arreglo a recorrer (lo llamaremos "*arr*" para el ejemplo).
- Se crea el nodo de operaciones de memoria "<<" para controlar si se llegó al final del arreglo, y se le asignan por hijos a "*i*" y "*arr*".

- Se crea un nodo de operaciones de memoria "::=" para asignarle al iterador la dirección de inicio del arreglo y se le asignan por hijos a "*i*" y "*arr*".
- Se crea un nodo de operaciones de memoria "+=" para hacer que el iterador avance sobre el arreglo (2 bytes si es de tipo int, 4 bytes si es de tipo long) y se le asignan por hijos a "*i*" y "*arr*".
- Se crea el nodo común "*CUERPO*" y se le asigna como hijo a la izquierda el conjunto de sentencias ejecutables del cuerpo del FOREACH, y a la derecha el nodo "+=".
- Se crea el nodo de control "*CUERPOAVANCE*" y se le asigna por hijo al nodo "*CUERPO*".
- Se crea el nodo de control "*COND_FOREACH*" y se le asigna por hijo al nodo "<<".
- Se crea el nodo común "*BUCLE*" y se le asigna por hijo izquierdo al nodo "::=" y por hijo derecho al nodo de control "*COND_FOREACH*".
- Se crea el nodo común "*FOREACH*" y se le asigna como hijo izquierdo al nodo "*BUCLE*" y como hijo derecho al "*CUERPOAVANCE*".

Consideraciones

● Al no tomar en cuenta el concepto de "ámbitos" para este lenguaje, resulta innecesaria la posibilidad de generar bloques begin-end dentro de otro bloque begin-end, por lo que se prohibió su uso en el lenguaje.

● Para que no haya ambigüedad en la declaración de una colección, se requirió que para definir su tamaño de forma directa se deba agregar al principio de lo que está entre corchetes un token "*size:* ". De esta forma, puede definirse un arreglo de tipo int de tamaño 5 de la siguiente forma:

int arr[size: 5];

● Para este trabajo se incorporó conversiones implícitas, lo cual implica que el compilador se encargará de detectar las conversiones que se requieran, desligando al usuario de dicha responsabilidad. En este caso, el compilador realiza una conversión del tipo int a long en caso de requerirse.

● Cabe resaltar que solo se permite realizar conversiones de un tipo más pequeño a uno más grande y solo del lado de la derecha, por ende si en una asignación el lado izquierdo queda con un tipo más chico que el lado derecho, se indicará como error por incompatibilidad de tipos.

● La operación de multiplicación, al devolver siempre resultados de tipo long, no puede ser almacenado en una variable de tipo int, así que se consideraría un caso error.

● El foreach requiere como parámetros un arreglo y una variable que acceda a sus datos. Para evitar conflictos de lectura, ambos identificadores deben ser del mismo tipo.

● A la variable encargada de iterar en una colección dentro de un FOREACH no puede asignarle un valor nuevo manualmente hasta que el FOREACH que lo utiliza como puntero al arreglo termine su ejecución.

Proceso de generación de código assembler

Funcionalidades de los nodos del árbol sintáctico y generación de código assembler

Como se detalló en el apartado anterior, se empleó el uso de un árbol sintáctico para esta etapa de la generación de código. Este árbol será leído y resuelto desde el nodo con hijos hoja más hacia la izquierda hasta el más a la derecha, generando instrucciones assembler a medida que sube hasta la raíz.

Se crearon varios tipos de nodos cuyas funcionalidades se verán reflejadas en esta etapa:

- Los nodos hoja (*ST_Leaf*) representan constantes o identificadores, los cuales se devolverán a su nodo padre para la operación que se requieran. En el caso de las posiciones de arreglo, primeramente se agregan comandos de acceso al valor puntual y se retorna el registro con la dirección del valor indicado.
- Los nodos colección (*ST_LeafCollection*) guardan el lexema de una colección, siendo el hijo quién referencia el valor que se debe devolver. A diferencia de en los nodos hoja, se requiere este nodo intermedio para aquellos casos especiales en que el id indicando la posición es un puntero (por lo que almacena una dirección y no un valor en concreto).
- Los nodos operacionales binarios (*ST_Common*) recibirán los resultados de sus nodos hijos, ya sean valores, identificadores o registros, y generarán código assembler a partir de estos. En el proceso se requerirán ciertos chequeos según el tipo de operación que sea, el tipo de los resultados, los registros que haya disponibles y el lugar en el que los datos estén guardados (no es lo mismo para el assembler que estén en un registro que en una variable). También pueden usarse simplemente como nodos para conectar sentencias ejecutables (en caso tal no requieren datos de sus hijos).
- Los nodos operacionales unarios (*ST_Unary*) fueron creados en este trabajo en particular específicamente para la operación de PRINT, la cual solo necesita por parámetro una cadena. El nodo generará una sentencia assembler de impresión por pantalla.
- Los nodos de control (*ST_Control*) son los encargados de generar las instrucciones de salto para instrucciones de condición (if y foreach) y sus respectivas etiquetas donde corresponda. Tienen incluida una lógica para generar etiquetas únicas de manera de saber siempre a qué dirección saltar.
- Los nodos de conversión (*ST_Convertion*) son los encargados de recibir algún dato de tipo INT en un registro, constante o variable y devolver su valor en tipo LONG para la siguiente operación en la que se lo requiera. Esto devolverá código assembler para la conversión de variables de un tipo a otro. También se utilizan en las variables puntero (usadas en el foreach para referenciar un valor de colección) para extraer y devolver el valor de la dirección indicada.
- Los nodos operacionales para instrucciones enfocadas a memoria (*ST_Memory*) son usados por las sentencias foreach, para las asignaciones en tiempo de ejecución. Al ejecutarse se encargan de crear instrucciones assembler para avanzar sobre el arreglo asignado hasta el final y acceder a sus valores uno por uno.

Pseudo-código de bifurcaciones - Asociado al método recorreArbol(...) implementado en la clase SyntacticTreeCtrl.

```
recorreArbol(TablaDeRegistros registros, PilaDeMensajes codigoAssembler, TablaDeSimbolos symbolTable) {
        if(getNombreControl() == "COND_FOREACH") {
                contEtiquetas++;
                codigoAssembler.agregarMensaje("etiquetaInicioCondForeach:");
                etiquetas.agregarEtiqueta("etiquetaInicioCondForeach");
        }
        operacion = NodoHijo().getAlmacenamiento();
        Si(getNombreControl()){
        es igual a "COND_IF":{
                contEtiquetas++;
                Si(operacion) {
                es igual a "<":{
                        codigoAssembler.agregarMensaje("JGE etiquetaPorMenor");
                        etiquetas.agregarEtiqueta("etiquetaPorMenor");
                }
                es igual a ">":{
                        codigoAssembler.agregarMensaje("JLE etiquetaPorMayor");
                        etiquetas.agregarEtiqueta("etiquetaPorMayor");
                }
                es igual a "LET":{
                        codigoAssembler.agregarMensaje("JG etiquetaPorMayorIgual");
                        etiquetas.agregarEtiqueta("etiquetaPorMayorIgual");
                }
                es igual a "GET":{
                        codigoAssembler.agregarMensaje("JL etiquetaPorMenorIgual");
                        etiquetas.agregarEtiqueta("etiquetaPorMenorIgual");
                }
                es igual a "DIF":{
                        codigoAssembler.agregarMensaje("JE etiquetaPorIgual");
                        etiquetas.agregarEtiqueta("etiquetaPorIgual");
                }
                es igual a "EQ":{
                        codigoAssembler.agregarMensaje("JNE etiquetaPorDistinto");
                        etiquetas.agregarEtiqueta(etiquetaPorDistinto);
                }}
        }
        es igual a "COND_FOREACH": {
                contEtiquetas++;
                codigoAssembler.agregarMensaje("JGE _label" + contEtiquetas);
                etiquetas.agregarEtiqueta("etiquetaFinForeach");
        }
```

```
                es igual a "THEN_ELSE":{
                        codigoAssembler.agregarMensaje("etiquetaFinIF:");
                        etiquetas.removerUltimaEtiqueta();
                }
                es igual a "THEN":{
                        contEtiquetas++;
                        codigoAssembler.agregarMensaje("JMP etiquetaFinIF");
                        codigoAssembler.agregarMensaje("EtiquetaInicioELSE:");
                        etiquetas.removerUltimaEtiqueta();
                        etiquetas.agregarEtiqueta();
                }
                es igual a "ELSE":{
                                codigoAssembler.agregarMensaje("etiquetaFinIF:");
                                etiquetas.removerUltimaEtiqueta();
                }
                es igual a "CUERPOAVANCE":{
                        codigoAssembler.agregarMensaje("JMP etiquetaPrincipioForeach");
                        codigoAssembler.agregarMensaje("finDeForeach:");
                        etiquetas.removerUltimaEtiqueta();
                        etiquetas.removerUltimaEtiqueta();
}}}
```

Tabla de ocupación de registros

Todos estos nodos generan código trabajando en sincronía con una tabla ocupación de registros que posee asociados cuatro registros específicos (o variables auxiliares en caso de quedarse sin registros libres). Al solicitarse un espacio de memoria (de 16 o 32 bits), la tabla devolverá el primer registro que encuentre libre asignándoselo al nodo que lo solicitó. También está la opción de solicitar un registro en específico, requiriendo hacer un cambio de registro en caso de que esté siendo usado por otro nodo.

Esta tabla fue resuelta mediante la clase RegisterTable, la cual posee una clase privada Registro con tres atributos:

- *nombre*: que indica a qué registro hace referencia (EAX, AX, ECX, etc).
- *is_busy*: atributo booleano que se setea en “true” si está ocupado.
- *nodoAsociado*: el cual indica a qué instancia de la clase SyntacticTree hace referencia.

La clase posee una lista de tamaño igual a la cantidad de registros del procesador y cada posicion del arreglo hará referencia a un registro especifico. EAX y AX asociados a la posicion 0, EBX y BX a 1, ECX y CX a 2 y EDX y DX a 3.

Para solicitar algún registro, se deberá llamar a alguno de los 3 métodos específicos para ello:

- *getRegFreeLong()*: Busca un registro disponible de 32 bits, y en caso de no encontrarlo asigna una variable auxiliar.
- *getRegFreeInt()*: Busca un registro disponible de 16 bits, y en caso de no encontrarlo asigna una variable auxiliar.

- *getReg(String registro)*: Busca el registro específico solicitado y lo asigna. En caso de encontrarlo ocupado primeramente debe buscar otro registro (o variable auxiliar en su defecto) para el nodo que inicialmente posee dicho registro solicitado.

Por último, algunas operaciones requieren una extensión o reducción de registro, y para mantener registro de esa extensión se utiliza la siguiente función:

- *extendTo32bits()*: Se modifica el nombre del registro guardado como de 16 bits a 32 bits.
- *reduceTo16b()*: Se modifica el nombre del registro guardado como de 32 bits a 16 bits.

Chequeos en tiempo de ejecución

Se nos asignó dos tipos de chequeos en tiempo de ejecución: La división por cero y el acceso a una parte inexistente del arreglo.

Para su control se creó una sección especial dentro del código assembler (indicada con su propia etiqueta) donde se indicará sobre el error detectado y se saltará al final del programa. Las etiquetas son "*_DivisionPorCero:*" y "*_ArregloFueraDeRango:*".

Cada división y acceso a un arreglo tendrá su respectivo chequeo, y en caso de ser necesario, saltará a la sección correspondiente que da aviso del error y terminará el programa. En caso de no haber error, no saltará y continuará el resto del programa sin problemas.

Colecciones inferidas

La inclusión de manejo de colecciones dentro del compilador requirió agregar ciertos atributos a la tabla de símbolos: La identifierClass, cSize, cSizeBytes y elemsCollection.

Estos datos sirven tanto para generar el archivo final como para acceder a ciertas características que conviene tener almacenadas (en vez de calcularse cada vez que se necesitan). La mayoría de sus atributos son definidos en la ejecución del parser para utilizarse en la generación de código Assembler con mayor practicidad.

Un buen ejemplo de esto es cSizeBytes. Este atributo facilita los cálculos que de otro modo podría hacer el Assembler, requiriendo registros extra para los cálculos (los cuales son considerablemente limitados en este trabajo).

Ya desde la declaración se nos permite siempre obtener el tamaño del arreglo, ya sea por definición de sus valores internos o seteo directo del tamaño. El parser aprovecha esto para ya dejar definidos la mayor cantidad de atributos en la tabla de símbolos.

Uso de interfaz

La pantalla principal al ejecutar el compilador brinda la posibilidad de crear un archivo de texto con código en el cuadro grande, guardarlo o cargar uno ya hecho.

Se brinda la posibilidad de realizar la acción "Deshacer" presionando "Ctrl + Z" y "Rehacer" con "Ctrl + Shift + Z".

Por último, con el botón "Compilar" se compila el código, devolviendose el .txt con toda la información necesaria para ejecutar el código assembler en la misma carpeta donde el código original se almacena. A continuación se dará la opción de crear el ejecutable seleccionando el archivo "...\masm32\bin\ml.exe".

Conclusiones

El diseño de esta segunda parte del compilador requirió un conocimiento importante sobre la estructura que usamos para generar el código intermedio (el árbol sintáctico) junto con la forma en la que la gramática se comporta para ir retornando los datos a medida que lee los tokens.

El diseño de distintos nodos con las funcionalidades de generación de assembler ya definidas permitió tener un mayor control de la distribución de responsabilidades.

Aún así, ciertas situaciones presentaron cierta dificultad. Por ejemplo, el pasaje de nodos de un no terminal a otro, ya que estos requerían un casteo de tipo ParserVal a SyntacticTree en el proceso. También, el uso de un árbol sintáctico para generar el código intermedio requirió diseñar la gramática de un modo muy específico que permitiera armar la estructura sin el uso de funciones especiales, todo para no perder los nodos “cabecera”.

En pocas palabras, se notó una limitación importante empleando árbol sintáctico como código intermedio a la hora de realizar las modificaciones necesarias en la generación de código assembler.

También, comprendiendo mejor toda la funcionalidad de la tabla de símbolos, se agregaron muchísimos atributos que permitan generar el “*.data*” del archivo de salida correctamente, junto con otros que detallaron cosas muy en específico de los datos que no necesitamos considerar en la entrega anterior.