Compilador.

Trabajo Diseño de Compiladores

Etapas 2 y 3: Generación de Código Intermedio y Generación de Código de Máquina

Universidad Nacional del Centro de la Provincia de Buenos Aires.

23/11/2024

Alumnos: Girardi Emiliana; Lapenta Nicolas; Román Nahuel.

Tutor: José Fernández León

Índice.

Introducción

En este informe se lleva a cabo la explicación de cómo implementamos en lenguaje Java la etapa de Generación de Código del compilador propuesto por la cátedra.
La generación de código de salida fue dividida en dos etapas intermedias que son:

- Generación de código intermedio
- Generación de código de máquina

para la generación de código intermedio se nos asignaron Tercetos, mientras que para la generación de código de máquina elegimos Assembler de Pentium x86.
A continuación se realiza una explicación detallada de cómo fue llevada a cabo cada etapa.

Generador de código intermedio.

El código intermedio es una representación previa a la traducción del programa a un código máquina.

En esta etapa se generará el código de las sentencias en formato de Tercetos. Este código intermedio permite realizar el chequeo semántico de:

- Comprobación de tipos.
- Comprobación de flujo de control.
- Comprobación de unicidad.

Tercetos.

Los *tercetos* son una forma de representación intermedia del código que consta de tres partes. La primer parte almacena el operando u operación, mientras que la segunda y tercera pueden almacenar una referencia a otro terceto o un lexema, así como también, en operaciones unarias el segundo operando podría ser nulo. Visualmente el terceto tiene la siguiente forma:

$$T1.\ (\ +,\ 1,\ 8\ )$$

$$T2.\ (\ :=,\ a,\ [T1]\ )$$

Para la representación en código de esta estructura se planteó la creación de una

clase terceto que tiene como atributos tres variables de tipo String que representan los operandos, y una variable de tipo Integer que representa el tipo.

```
public class Terceto {
    private String operador, operando1, operando2;
    private Integer tipo;
    public Terceto(String operador, String operando1, String operando2){
        this.operador=operador;
        this.operando1=operando1;
        this.operando2=operando2;
        this.aux = null;
    }

}
```

Notación Posicional YACC

Para obtener en cada regla de la gramática, los valores de los lexemas, se debe utilizar la lógica posicional $$, $$n. Estos valores que provee YACC, contienen la clave a la Tabla de Símbolos de cada token, el cual en nuestro caso es el valor del lexema que es de tipo String.

Esta notación fue imprescindible a la hora de automatizar la creación de los tercetos de cada regla, ya que nos permitió obtener el lexema de los operandos.

$$.sval: contiene el lexema del lado izquierdo de la regla.

$n.sval; contiene el lexema del token que se encuentra en la posición n del lado derecho de la regla.

Generador.

A medida que se recorre el programa y se genera la lista de reglas, se deben ir creando los tercetos correspondientes y tienen que almacenarse de forma tal que luego puedan volver a ser accedidos cuando se necesite.

Para lograr este cometido se implementó la clase *Generador* la cual contiene:

- El arreglo de tercetos, donde cada terceto se identifica por su posición en el mismo.
- Una instancia de la clase traductor que será el encargado de la traducción a código de máquina en la siguiente etapa.

- Un atributo booleano de error, cuyo valor es 1 cuando hay un error de sintaxis o de semántica en el programa de entrada.
- Una pila que se utiliza para la creación de Tercetos de bifurcación.
- Dos estructuras de almacenamiento para etiquetas y sentencias GOTO para implementar la lógica de etiquetas en Tercetos.
- La tabla de compatibilidad de tipos para el chequeo semantico.

```
public class Generador {
    private ArrayList<Terceto> tercetos;
    private TraductorAssembler traductor;
    private static volatile Generador instance;
    private boolean error;
    private Stack<String> pila;
    private ArrayList<String> etiquetas;
     private HashMap<Integer, String> mapGoto;
     //Key: numero terceto, Value: etiqueta
     private String[][] tablaComp;
     ...
}
```

Dicha clase debe tener una instancia única en todo el programa, por este motivo se utilizó el patrón de diseño *singleton*.

Creación de Tercetos.

Los tercetos deben ser creados en las siguientes reglas:

- Invocación a función.
- Declaración de función.
- Asignación.
- Goto.
- Sentencia if.
- Sentencia REPEAT_UNTIL.

- Etiqueta.
- Salida.

Chequeo Semántico de Tipos.

El lenguaje asignado para nuestro compilador, no admite operaciones entre operandos de distintos tipos, por lo tanto es de suma importancia hacer verificaciones de compatibilidad en cada sentencia ejecutable.
En el caso de los parámetros, se nos ha especificado que puede haber conversiones explícitas de parte del usuario que permita pasar como parámetro real, un elemento que tiene un tipo distinto al que se espera en el parámetro formal.

Dentro del trabajo los tipos son representados numéricamente de la siguiente manera:

- **Palabra Reserv:** 0.
- **Unsigned :** 1.
- **Single :** 2.
- **Octal :** 3.
- **Multilínea :** 4.
- **Triple Unsigned:** 5.
- **Triple Single:** 6.
- **Triple Octal:** 7.
- **Etiqueta:** 8.
- **Salto:** 9.
- **Función:** 10.
- **Desconocido :** 50.
- **Símbolo:** 60.

Los tipos de Triple son los que se admiten para definir tipos propios del usuario con la sentencia TYPEDEF.

Luego, cuando se define una variable de un tipo definido por el usuario, en la tabla de

símbolos se crean las 3 variables individuales e independientes que contiene el terceto. A dichas variables se les coloca el tipo que corresponde al tipo primitivo sobre el cual se definió el tipo del usuario.
Por ejemplo:
Si el usuario declara un tipo “tipoTriple” de TRIPLE_UNSIGNED, y luego define una variable X con este tipo, en la regla de la gramática donde se detecta esta declaración, se verificará primero que el tipo sobre el cual se define X sea un tipo definido por el usuario y como segunda instancia se agregarán a la tabla de símbolos las variables:

- X{1} (tipo TIPO_UNSIGNED)
- X{2} (tipo TIPO_UNSIGNED)
- X{3} (tipo TIPO_UNSIGNED).

Sentencias Declarativas

Es necesario que cada terceto tenga como característica de qué tipo será el resultado de la operación. Sin embargo, al momento en que el analizador léxico detecta un identificador, o variable, y su lexema es almacenado en la tabla de símbolos, no siempre será posible saber el tipo de dicho identificador.
Por este motivo dentro de la gramática en cada regla que represente una declaración de variable se deberá cargar, al lexema en la tabla de símbolos como atributo el tipo correspondiente según la declaración.

Sentencias Ejecutables

Para asegurar la integridad del programa, que no permite operaciones entre operandos de tipos distintos (excepto en pasaje de parámetros), se debe verificar que los tipos de los operandos de cada sentencia ejecutable sean el mismo.
Dichos operandos pueden ser lexemas de la tabla de símbolos o pueden ser referencias a otros tercetos, en cuyo caso se debe verificar el atributo “tipo” de terceto referenciado.
En el caso del retorno de función RET, que es una sentencia ejecutable, se verifica que el tipo de dato retornado sea el mismo que posee el identificador de la función.

Conversiones Explícitas en Parámetros

En el caso de los parámetros, se debe consultar la tabla de compatibilidad de tipos del Generador para evaluar si los tipos son compatibles en la sentencia de Invocación a Función.

Esta tabla se implementó como una matriz de String que puede contener NULL si los tipos consultados son incompatibles o la conversión indicada para que el pasaje sea

válido. En caso de que el resultado de la consulta a la tabla contenga NULL, se arroja error de compatibilidad de tipos, en cambio, si se recibe un valor String, se genera un terceto de conversión para el parámetro real, que contiene el tipo de conversión que debe realizarse.

	Unsigned	**Single**	**Octal**
Unsigned	*itoI*	*stoI*	*null*
Single	*itoS*	*stoS*	*null*
Octal	*null*	*null*	*oto0*

Fig. 1. Tabla de Compatibilidad de Tipos.

Chequeo Semántico de Uso.

Se agregó a cada lexema de tipo identificador en la Tabla de Símbolos el atributo de Uso. Esta información es de gran utilidad al momento de verificar si las sentencias ejecutables son válidas.

Los tipos de uso están diferenciados por valores numéricos y son diferenciados de la siguiente manera:

- **Nombre de variable:** 101.
- **Nombre de función:** 102.
- **Nombre de tipo:** 103.
- **Nombre de parámetro:** 104.
- **Nombre de etiqueta:** 105.
- **Nombre de programa:** 106.

En la gramática, cuando se detecta una declaración de variable se le asigna el uso 101, mientras que cuando se declara una función se le asigna al identificador del nombre de la misma el uso 102 y al de su parámetro el uso 104.

En las sentencia ejecutable de invocación a función se verifica que el identificador al cual se está refiriendo como función, sea realmente un nombre de función, buscando su uso en la tabla de símbolos.

Chequeo Semántico de Ámbitos y Alcance.

El ámbito podemos definirlo como el contexto dentro del cual una variable o función o etiqueta es reconocible y accesible. Determina dónde una entidad definida en el código puede ser utilizada y dónde no es visible o accesible.

El lenguaje de programación utilizado en este trabajo tiene un binding de alcance estático, por ende el compilador debe poder reconocer y diferenciar los ámbitos a donde pertenecen las variables y arrojar error cuando se intenta acceder a un identificador que no está al alcance.
Este error se considera como intento de acceso a un identificador inexistente, debido a que el identificador solicitado para el alcance de ese ámbito no existe en la tabla de símbolos.

Para lograr este objetivo se implementó el método ***Name Mangling*** que consiste en tener una variable global de tipo String en la Tabla de Símbolos que almacena y actualiza el ámbito en función del programa que se esté ejecutando.

```
String ambitos = ".global";
```

Esta variable de ámbito comienza con el ámbito ".global", el cual es el ámbito mínimo que puede existir en el programa. Luego, al momento que se detecta la declaración de una función, se actualiza el ámbito concatenando lo que ya tenía con el nombre de la función encontrada.
Entonces, por cada declaración de variable que encuentre dentro de dicha función se cambiará el lexema del identificador en la Tabla de Símbolos, concatenando su ámbito correspondiente.

En las sentencias ejecutables se verifica el alcance de los identificadores que se van a utilizar, mediante el metodo de la Tabla de Simbolos: buscarVariable(String lexema).

A este método se le pasa por parámetro el lexema que se quiere encontrar en el alcance actual, debido a que los operandos de una sentencia ejecutable son Strings que pueden contener el valor del token con el cual se definió en el programa la sentencia o la referencia a un terceto que se quiere utilizar.
De esta forma, buscarVariable puede devolver:

- NULL si no se encuentra el lexema en el alcance,
- el propio lexema si se trata de un identificador al alcance o de una constante,
- el String "Terceto" si se trata de una referencia a terceto.

Esta última opción fue agregada con la finalidad de facilitar el chequeo semántico de tipos, para la cual se requiere saber el operando es un identificador al alcance cuyo tipo se puede obtener de la Tabla de Símbolos o un terceto cuyo tipo se obtiene mediante el Generador.

Búsqueda de los Identificadores

Para buscar un identificador, el método buscarVariable verifica primero si se trata de un terceto o una constante. Si no es así, concatena al lexema el ámbito actual, que es el ámbito cuyo alcance se quiere verificar. Dicho ámbito puede corresponder a la concatenación de varios identificadores de funciones separados por puntos, o como mínimo el ámbito global.
De esta forma, el método comienza a buscar el lexema en la Tabla de Símbolos, desconcatenando cada ámbito hasta que lo encuentra. Al momento que lo encuentra, retorna ese valor, que es el que será utilizado para generar el código intermedio.

Sentencias Condicionales.

En el manejo de sentencias condicionales hay que tener especial cuidado, ya que realmente lo que se debe lograr es hacer un salto por falso a una etiqueta específica en caso que la condición sea falsa, y un salto incondicional a otra etiqueta en caso de verdadero.

El problema surge que al momento de crear el terceto que genera el salto no es posible determinar a qué etiqueta hay que saltar ya que aún no ha sido creada.

Para lograr este cometido se requirió implementar una pila de tipo String en la clase *generador* que almacenará posiciones de tercetos que aún no tengan definido los saltos, para posteriormente poder completarlos.

Sentencia IF.

La sentencia IF está constituida de la siguiente manera:

```
Sentencia_if : condicion_if  bloque_sentencias_ejecutables
   condicion_ELSE  bloque_sentencias_ejecutables END_IF
```

Donde

condicion_if : **IF** condicion **THEN**

condicion_else : ELSE

Se optó por separar la regla de esta manera ya que es necesario distinguir el momento en que termino una condición para poder crear un terceto de salto por falso y almacenarlo en la pila, para después al momento de que se lea un *ELSE* poder crear la etiqueta, desapilar el terceto, completarlo, crear otro terceto de salto incondicional y almacenarlo en la pila. Sobre el final de la *sentencia_if* se podrá crear una etiqueta de salto y desapilar, y completar el terceto faltante.

Sentencia REPEAT UNTIL.

La sentencia Repeat Until esta formada por:

```
repeat_until : sentencia_repeat bloque_sentencias_ejecutables UNTIL
                                    condicion
```

donde

```
sentencia_repeat: REPEAT
```

El propósito es similar al de la *sentencia_if* antes mencionada. Se debe reconocer primero la palabra reservada REPEAT para poder crear una etiqueta y luego almacenar dicha etiqueta en el terceto de salto incondicional que se creará después de la condición.

Saltos Incondicionales.

El lenguaje presentado cuenta con la función **GOTO** que permite saltar a una etiqueta dentro del programa, siempre y cuando se encuentre en el mismo ámbito.

El punto clave se encuentra en que la etiqueta puede estar declarada antes o después de el llamado a **GOTO**, por ende se debe tener un especial cuidado con dicho manejo.

Para la implementación de esta funcionalidad se implementó dentro de la clase *generador* una estructura de lista en donde se almacenan las etiquetas creadas(concatenadas con su ámbito). Por otra parte se utilizó una estructura de tipo hash con una clave Integer y un valor de tipo String, en la cual se almacenarán los tercetos que contengan un salto a una etiqueta aún inexistente. En está última estructura la clave representa el número de terceto y el valor de la etiqueta aún no creada.

Al momento en que se detecta una etiqueta se verifica si existe. Si la misma no se encuentra en la lista, se agrega a la misma y se recorre el hash map buscando algún terceto que la haya referenciado. En el caso de encontrar un terceto así se procede a eliminarlo de la estructura.

Por otro lado, siempre que se realice un salto se almacenará el terceto de salto con la etiqueta detectada, y en el caso que no se encuentre la misma se procede a almacenar el terceto en la estructura de hash.

Funciones.

Para el tratamiento de la declaración de una función se optó por separar el encabezado del cuerpo de la siguiente manera:

```
encabezadoFun : tipo FUN ID '(' tipo ID ')'
```

El motivo de esta implementación se debe a la necesidad de crear un terceto función antes de comenzar a generar los tercetos del cuerpo de dicha función. Esto nos permite poder diferenciar donde comienza la función para posteriormente utilizar este recurso en la traducción a código assembler.

Manejo de errores.

Dentro del código dentro de la gramática pueden surgir varios errores tales como:

- Falta de punto y coma.
- Errores de sintaxis.
- Errores de tipos.
- Errores de ámbito.
- Errores de conversión.
- Errores de recursión.

Otro error puede surgir al momento de hacer un salto GOTO a una etiqueta y que esta nunca se cree, esto genera que el map de goto mencionado anteriormente nunca se vacíe y genere un error al final del programa.

Dichos errores no detienen la ejecución del programa y tampoco la creación de tercetos, pero sí arrojan un mensaje por pantalla y no se permitirá que se genere código assembler.

Generador de código Assembler.

Para la traducción a Assembler se utilizó la técnica de Variables Auxiliares, que consiste en guardar los resultados intermedios de las operaciones en variables creadas por el compilador de forma automática. Dichas variables auxiliares son asignadas a cada terceto para indicar que el resultado de la operación traducida se encuentra ahí.

El código traducido se coloca en el archivo de salida obtenido mediante argumentos del programa.

Formato de un programa Assembler:

```
.MODEL small ; Indica el tamaño de programa
.STACK 200h ; Inicializa Stack en dir. indicada
.DATA ; Indica zona de datos ; Aquí se definen variables y datos
.CODE ; Indica inicio zona de código
  START:
  ; Código del programa
```

La lógica de la traducción fue implementada en la clase TraductorAssembler que se encarga de mapear cada terceto con su correspondiente método de creación de Assembler y posee los métodos que declaran las variables auxiliares y las constantes de tipo TIPO_SINGLE (debido a que el manejo en assembler de constantes float requiere que se declaren previamente en la sección de .DATA) y los métodos que escriben el programa en el formato esperado de archivo Assembler.

Flujo de Ejecución.

Cuando se termina de ejecutar el Parser, se llama al método del Generador: generarCodigoMaquina(). En este punto, los tercetos ya se generaron en base al código contenido en cada regla de la gramática.

Este método se encarga de verificar que el flag de errores este en 0, es decir, que no haya habido errores sintácticos y semánticos, y además verifica que la lista de GOTOs pendientes esté vacía. Luego de ambas verificaciones se procede a recorrer los tercetos y la tabla de símbolos para reemplazar aquellos caracteres que generan conflicto con la sintaxis del código Assembler. Una vez finalizado el proceso se llama al Traductor para generar el encabezado y luego para que genere la traducción de cada terceto a código Assembler, pasándole como parámetro la lista de tercetos generada.

```
public void traducir(Terceto t)
```

Como se mencionó con anterioridad, cada tipo de terceto posee su método de traducción. El método traducir(Terceto t) se encarga de mapear cada terceto a su correspondiente traducción.
Para ciertas operaciones como las aritméticas o las comparaciones, se hizo la distinción entre operandos de tipo Single y de tipo Octal o Unsigned, debido a que el manejo de registros Assembler no es el mismo para datos de tipo float.

Mapeo de las Traducciones

En la siguiente tabla se muestra para cada tipo de terceto, el método que le corresponde para traducir su código intermedio a código de máquina.

Operador del Terceto	Método de Traducción
Etiqueta	etiqueta(Terceto terceto)
Declaración a Función	funcion(Terceto terceto)
Suma	Para tercetos de tipo TIPO_SINGLE: sumaPuntoFlotante(Terceto terceto) Para tercetos de tipo TIPO_OCTAL o TIPO_UNSIGNED: suma(Terceto terceto)
Resta	Para tercetos de tipo TIPO_SINGLE: restaPuntoFlotante(Terceto terceto) Para tercetos de tipo TIPO_OCTAL o TIPO_UNSIGNED: resta(Terceto terceto)
Multiplicación	Para tercetos de tipo TIPO_SINGLE: multiplicacionPuntoFlotante(Terceto terceto) Para tercetos de tipo TIPO_OCTAL o TIPO_UNSIGNED: multiplicacion(Terceto terceto)
División	Para tercetos de tipo TIPO_SINGLE: divisionPuntoFlotante(Terceto terceto) Para tercetos de tipo TIPO_OCTAL o TIPO_UNSIGNED: division(Terceto terceto)
Asignación	Para tercetos de tipo TIPO_SINGLE: asignacionPuntoFlotante(Terceto terceto). Para tercetos de tipo TIPO_OCTAL o TIPO_UNSIGNED: asignacion(Terceto terceto)
Salida	impresion(Terceto terceto)

Retorno	ret(Terceto terceto)
Invocación a Función	call(Terceto terceto)
Branch Incondicional (BI)	branchIncondicional(Terceto terceto)
Branch True (BT)	branchTrue(Terceto terceto)
Branch False (BF)	brachFalse(Terceto terceto)
And	and(Terceto terceto)
!=	distinto(Terceto terceto) Dentro de este método se verifica si el tipo de los operandos es TIPO_SINGLE y se realiza una traducción distinta para este caso.
=	igual(Terceto terceto) Dentro de este método se verifica si el tipo de los operandos es TIPO_SINGLE y se realiza una traducción distinta para este caso.
>	mayor(Terceto terceto) Dentro de este método se verifica si el tipo de los operandos es TIPO_SINGLE y se realiza una traducción distinta para este caso.
<	menor(Terceto terceto) Dentro de este método se verifica si el tipo de los operandos es TIPO_SINGLE y se realiza una traducción distinta para este caso.
>=	mayorIgual(Terceto terceto) Dentro de este método se verifica si el tipo de los operandos es TIPO_SINGLE y se realiza una traducción distinta para este caso.
<=	menorIgual(Terceto terceto) Dentro de este método se verifica si el tipo de los operandos es TIPO_SINGLE y se realiza una traducción distinta para este caso.
itoS (conversión de unsigned a single)	conversionItoS(Terceto terceto)
stoI (conversión de single a unsigned)	conversionStoI(Terceto terceto)

Instrucciones utilizadas.

Las instrucciones que se van cargando en el archivo assembler y van generando el código son aquellas provistas por la cátedra y mencionadas en el material adicional, a excepción de puntuales casos que explicaremos a continuación.

- **SETGE y SETAE**: Cada instrucción usa los flags de comparación para determinar si una operación entre dos valores cumplió la condición "mayor o igual", almacenando 1 si es verdadero, y 0 si es falso. La primer instrucción es para comparaciones entre valores *con signo* mientras que la segunda es para valores sin signo.
  Esta instrucción es utilizada en la función mayorIgual() dentro de la clase TraductorAssembler. Ejemplo:

```
salida.append("MOV CX, 0 " + saltoLinea);
salida.append("FLD " + op1 + saltoLinea);  //Cargar op1 en ST(0)
salida.append("FLD " + op2 + saltoLinea);  //Cargar op2 en ST(1)
salida.append("FCOM ST(1)" + saltoLinea);  //Comparar ST(0) y ST(1)
salida.append("FSTSW AX"+saltoLinea);
salida.append("SAHF"+saltoLinea);
salida.append("SETGE  CL" + saltoLinea);
```

- **SETG** y **SETA::** Cada instrucción usa los flags de comparación para determinar si una operación entre dos valores cumplió la condición "mayor", almacenando 1 si es verdadero, y 0 si es falso. La primer instrucción es para comparaciones entre valores *con signo* mientras que la segunda es para valores sin signo.
  Esta instrucción es utilizada en la función mayor() dentro de la clase TraductorAssembler. Ejemplo:

```
salida.append("MOV CX, 0 " + saltoLinea); //inicializamos en false
salida.append("MOV  AX, " + op1 + saltoLinea);
salida.append("CMP AX, " + op2 + saltoLinea);
salida.append("SETA CL" + saltoLinea);
salida.append("MOV CH, 0 " + saltoLinea); //inicializamos en false
```

- **SETLE** y **SETBE:** Cada instrucción usa los flags de comparación para determinar si una operación entre dos valores cumplió la condición "menor o igual", almacenando 1 si es verdadero, y 0 si es falso. La primer instrucción es para comparaciones entre valores *con signo* mientras que la segunda es para

valores sin signo. Esta instrucción es utilizada en la función menorIgual() dentro de la clase TraductorAssembler. Ejemplo:

```
salida.append("MOV CX, 0 " + saltoLinea); //inicializamos en false
salida.append("MOV AX, " + op1  + saltoLinea);
salida.append("CMP AX, " + op2 + saltoLinea);
salida.append("SETBE CL" + saltoLinea);
salida.append("MOV CH, 0 " + saltoLinea); //inicializamos en false
```

- **SETL** y **SETB:** Cada instrucción usa los flags de comparación para determinar si una operación entre dos valores cumplió la condición "menor", almacenando 1 si es verdadero, y 0 si es falso. La primer instrucción es para comparaciones entre valores *con signo* mientras que la segunda es para valores sin signo.
  Esta instrucción es utilizada en la función menor() dentro de la clase TraductorAssembler. Ejemplo:

```
salida.append("MOV CX, 0 " + saltoLinea); //inicializamos en false
salida.append("FLD " + op1 + saltoLinea); // Cargar op1 en ST(0)
salida.append("FLD " + op2 + saltoLinea); // Cargar op2 en ST(1)
salida.append("FCOM ST(1)" + saltoLinea); //Comparar ST(0)con ST(1)
salida.append("FSTSW AX"+saltoLinea);
salida.append("SAHF"+saltoLinea);
salida.append("SETL  CL" + saltoLinea);
```

- **SETZ** y **SETE:** Estas instrucciones evalúan el estado de los flags y determina si la comparación anterior entre dos valores es la de igual, en caso de verdadero almacena un 1 en el byte de destino. Esta instrucción es utilizada en la función menor() dentro de la clase TraductorAssembler. Ejemplo:

```
salida.append("MOV CX, 0 " + saltoLinea); //inicializamos en false
salida.append("MOV AX, " + op1  + saltoLinea);
salida.append("CMP AX, " + op2 + saltoLinea);
salida.append("SETE CL" + saltoLinea);
```

Manejo de archivos.

Para lograr la traducción de código intermedio a código Assembler se implementó la utilización de dos archivos uno para construir el encabezado y otro para el código en sí. Se optó por esta alternativa debido a que para poder utilizar las variables auxiliares es necesario que estén declaradas, pero dichas variables no se declaran hasta luego de armar las instrucciones del cuerpo del código.

El funcionamiento consta de primero generar el código intermedio del encabezado, recorriendo la tabla de símbolos y registrando todas las variables, cadenas y constantes de punto flotante en la sección *.DATA*. Luego se genera el código para el cuerpo del programa, y en el momento que se necesiten crear variables auxiliares las mismas se almacenarán en la tabla de símbolos. Posteriormente se recorre nuevamente la tabla de símbolos con el fin de encontrar aquellas variables que no estaban en un inicio y cargarlas en el archivo de encabezado.
Una vez finalizado el proceso se procede a copiar el encabezado y el cuerpo en el archivo indicado por el usuario.

Controles en Tiempo de Ejecución.

El código assembler resultante incorpora chequeos en tiempo de ejecución y ante una situación de error se emite un mensaje y finaliza la ejecución.

Overflow en sumas de enteros.

Se chequea en ejecución que una suma entre enteros positivos no exceda el rango del tipo resultado. En nuestro caso como las operaciones son entre enteros positivos y no se permiten conversiones, el resultado será entero positivo.
Para manejar el overflow implementamos la instrucción **JC**. Esta instrucción tomá el flag **CF**(Carry Flag) que luego de una operación aritmética y nos indica si el resultado de la misma es mayor que lo que puede almacenarse en el tamaño del registro utilizado, tendrá un 1 si hubo desbordamiento o un 0 si no lo hubo. Luego la instrucción **JC** observa ese flag y en caso de ser verdadero(1) salta a una etiqueta, en caso de ser falso sigue el flujo normal del programa.
Para nuestro programa se creó una etiqueta a la cual se saltará en caso de un desbordamiento cuyo comportamiento posterior será mostrar un mensaje de error por pantalla y terminarán la ejecución. Ejemplo:

```
MOV AX, l_global_funcion
ADD AX, @aux3
JC ??errorOverflow
MOV @aux4, AX
...
??errorOverflow:
  invoke StdOut, addr errorMsgOverflow
  JMP END_START
```

Resultados Negativos en restas de enteros sin signo.

El código resultante controla el resultado de una resta entre enteros positivos y si la misma es negativa arroja error por pantalla y cierra la ejecución.
Este manejo se hizo a través de la instrucción **JS** la cual hace un salto a una etiqueta definida observando el estado del flag **SF**(Sign Flag) el cual es uno si el resultado de la operación aritmética previa es negativo. En caso de no serlo se continúa con el flujo normal del programa. Ejemplo:

```
MOV AX, x_global
SUB AX, y_global
JS ??errorRestaNegativa
MOV @aux13, AX

...
??errorRestaNegativa:
  invoke StdOut, addr errorMsgRestaNegativa
```

Conversión de un número negativo a un entero.

Puede darse el caso en que el programa indique hacer una conversión de tipo *single* a *unsigned* en un pasaje de parámetros. El problema sucede cuando ese valor tipo *single* es negativo, en cuyo caso la ejecución del programa terminará y arrojará error de conversión.
Esto lo logramos controlar con la instrucción **JL** que realiza un salto condicional observando una comparación previa. El salto se efectuará si el resultado de la comparación indica que un número es menor que otro. En nuestro caso:

```
MOV EAX, z_global
CMP EAX, 0
JL ??errorConversionNegativo
...
??errorConversionNegativo:
  invoke StdOut, addr errorMsgConversionNegativa
  JMP END_START
```

Control de recursividad.

Para el manejo de recursividad optamos por tratarlo en tiempo de compilación. Esto lo hicimos debido al manejo tercetos implementado en donde podemos verificar al momento de una invocación a una función que no sea igual a la función actual utilizando el ámbito.

Problemas Surgidos

Pasaje de Parámetros

Uno de los problemas más difíciles de solucionar fue el surgido con los parámetros de las funciones. Debido a que Assembler provee una pila de variables donde se colocan los parámetros reales antes de llamar a cada función, al utilizarla nos dimos cuenta de que traía problemas cuando se declaran funciones anidadas, debido a que si las funciones de ámbito más profundo quieren utilizar un parámetro de su función padre, no saben en qué registro encontrarlo.

Solución: para solucionar este inconveniente se ideó que el terceto de la declaración de función tenga como un operando al parámetro real, y a su vez tenga una etiqueta de tipo *función* que al ser detectada en el traductor se genera la instrucción de assembler para generar una etiqueta y posteriormente las instrucciones para el manejo de la pila. Esto nos permite poder sacar el primer elemento de la pila y poder almacenarlo en el parámetro de la función.

Declaración de variables auxiliares.

Cada resultado de una operación, retorno o pasaje de parámetro fue tratado dentro del código assembler con variables auxiliares, que se iban generando en el momento en que se precisaban. Esto genera una facilidad a la hora de crear programas extensos, pero trae un inconveniente. Para poder utilizar las variables auxiliares es necesario declararlas previamente.

Solución: tal y como se mencionó en una sección anterior, se implementó la utilización de un archivo de encabezado el cual se crearía y completaría con el contenido de la tabla de símbolos, y al finalizar la creación de instrucciones que componen el cuerpo del programa se procedería a finalizar el encabezado con la declaración de las variables auxiliares creadas.

Funciones.

Las funciones deben ir declaradas por fuera de la etiqueta *START* del código assembler. Lo que sucede es que la lectura de tercetos se hará secuencial por ende no se podrá diferenciar las funciones previamente como lo hacemos con las variables.

Solución: se implementa un nuevo archivo de funciones el cual será escrito cuando se detecte una declaración de función, y dejará de ser escrito al momento que llega el último retorno.

Salida del Compilador.

El compilador genera cinco salidas:

1. Los Warnings del programa que serán generados en tiempo de compilación. Estos se mostraran por consola y en un color amarillo.
2. Los errores generados en tiempo de compilación (Errores Léxicos, Sintácticos y Semánticos), indicando la descripción de cada error y la línea en la que fue detectado. Estos se mostrarán por consola y en un color rojo..
3. Contenidos de la Tabla de Símbolos. Se visualizará por consola todos los lexemas junto con sus atributos(token, tipo y uso).
4. Lista de Tercetos generados en la etapa de generación de código intermedio. Se visualizarán por consola y en ellos se podrá ver el número de terceto y su contenido en el formato presentado en secciones anteriores.
5. El Código Assembler creado se verá reflejado un archivo en la dirección pasada por el usuario a través del argumento del programa.

Ejecución del Compilador.

Se adjunta el archivo Compilador_grupo16.jar compilado desde la versión de Java 11.
Para ejecutar dicho archivo se le debe pasar como argumento del programa un dirección archivo de texto que contenga el código fuente a compilar y una dirección destino indicando el archivo a donde se desea escribir el código assembler. Esto se debe realizar mediante **consola** con el siguiente comando:

```
java -jar Compilador.jar <Dirección Archivo>.txt <Dirección archivo>.asm
```

El segundo archivo que se debe indicar puede o no existir dentro de la dirección, pero si es necesario colocar el nombre y la extensión.
Acto seguido se debe ejecutar las instrucciones de **MAMS** para poder compilar y ejecutar el archivo assembler generado.

Compilar .ASM:

```
ml /c /coff <codigo assembler>.asm
```

Enlazar el archivo objeto y generar un ejecutable:

```
link /SUBSYSTEM:CONSOLE <codigo assembler>.obj
```

Ejecutar programa:

```
<codigo assembler>.exe
```

Los códigos de prueba se adjuntan en una carpeta llamada “Archivos de Prueba”

junto con el **.jar**. Dentro de dicha carpeta se encontrará un archivo de prueba llamado “general1” cuyo programa aborda todas las instrucciones que toma nuestro programa y no contiene errores; por otro lado habrá carpetas que contendrán archivos que están separados por las situaciones que abordan. Las carpetas son:

- **Aritméticas:** aborda situaciones de sumas, restas, multiplicación, división y manejo de subtipos. A su vez contiene la carpeta *Overflows Flotantes* cuyo contenido son dos archivos que abordan los overflows de los tipos flotantes tanto en la suma como en la resta.
- **Funciones:** contiene tres archivos en los cuales se aborda el anidamiento, el casteo en el pasaje de parámetros y el manejo de funciones.
- **GOTO:** contiene un único archivo que tiene la instrucción de salto GOTO.
- **Incompatibilidades:** contiene dos archivos que manejan situaciones de incompatibilidad en el retorno e incompatibilidad en operaciones aritméticas.
- **Overflow Enteros:** aborda la situación en que una suma genera un overflow.
- **Pattern Matching:** tiene un archivo que contiene declaraciones y comparaciones múltiples.
- **Problemas de Alcance:** aborda las situaciones en que no se tiene alcance a una función o a variables.
- **Recursión:** situación en la que se intenta hacer un llamado recursivo.
- **Resta Enteros:** situación en que la resta de dos enteros es negativa.

Por último, junto al ejecutable se encontrará un archivo **.zip** con el código fuente.

CONCLUSIONES

En esta etapa se realizó la generación de código intermedio y a partir de la misma una generación de código assembler. Se trabajó con una estructura de tercetos y a partir de cada uno de los mismo y sus características se realizó la traducción a código assembler respetando la sintaxis y creando un código funcional.

Durante todo el desarrollo se aspiró, no solo a tener un compilador funcionando, sino también a trabajar de una manera ordenada y escalable, aplicando todo lo visto en la cursada 2024. Fue un trabajo desafiante el cual contiene mucho esfuerzo y horas de dedicación.