FACULTAD DE CIENCIAS EXACTAS UNICEN

Diseño de Compiladores

Trabajo Práctico Especial - Segunda Parte

Universidad Nacional del Centro de la Provincia de Buenos Aires
Facultad de Ciencias Exactas

Profesor Designado:
José A. Fernández León

Integrantes:
Vibart Eduardo
Elordi Tomas
González Matarras Felipe

Parte Tres: Generacion de Codigo Intermedio

Consigna TP3

Generación de código intermedio para las sentencias ejecutables. Es requisito para la aprobación del trabajo, que el código intermedio sea almacenado en una estructura dinámica. La representación puede ser, según la notación asignada a cada grupo. Para este grupo mediante un **Árbol Sintáctico**.

Temas particulares TP3

- Tema 10: **Cláusulas de compilación**

Ante una sentencia de declaración de constantes:
const <lista_de_constantes>; el compilador deberá registrar en la Tabla de Símbolos, la información de las constantes declaradas, registrando que se trata de constantes, el valor y el tipo que corresponda.
Ante una sentencia: when (<condicion>) then <bloque_de_sentencias>
Las sentencias del bloque sólo deberán ser consideradas en la compilación, cuando la condición pueda evaluarse como verdadera en tiempo de compilación.

- Tema 12: **while con expresión**

while (<condicion>) : (<asignacion>) <bloque_de_sentencias_ejecutables> ;
El bloque se ejecutará mientras la condición sea verdadera.
Si se detecta una sentencia break, se debe salir de la iteración en la que se encuentra.
La asignación del encabezado, en caso de estar presente, se deberá ejecutar al final de cada ciclo.

- Tema 18: **Continue con etiquetado**

Al detectar una etiqueta precediendo a la sentencia de control asignada (temas 11 al 16), y un continue con una etiqueta, se deberá proceder de la siguiente forma:
Se debe saltar al próximo ciclo de la iteración etiquetada con la etiqueta indicada en el continue.
Se deberá chequear que exista tal etiqueta, y que la sentencia continue se encuentre en la sentencia de control que lleva esa etiqueta, o en una sentencia anidada en la misma.
Por ejemplo:

```
outer: while (i < 10) : (i =: i + 1) { while (j < 5) : (j := j + 1) {
        continue :outer; // provocará el salto al próx. ciclo del while etiquetado con outer.
 };
};
```

- Tema 20: **Sentencia de control como expresión**

Incorporar, a la sentencia de control asignada en los temas 11 a 16, la posibilidad de devolver un valor utilizando la sentencia break, o un valor por defecto en la finalización normal de la sentencia.
Al detectarse la utilización de una sentencia de control (temas 11 a 16) como expresión, se deberá proceder de la siguiente manera:
Si se produce la ejecución del break, el valor retornado por la sentencia de control, que se utilizará como expresión, será el indicado en la sentencia break.
Si no se produce la ejecución del break, el valor retornado por la sentencia de control, que se utilizará como expresión, será el indicado en el else al final de la sentencia.
Para todos los ejemplos, si se ejecuta el break, el valor asignado a la variable a será 1. En caso contrario, será 3.
Dado que el valor retornado participa en una expresión aritmética, se deberá chequear que el tipo del valor devuelto sea compatible con el resto de la expresión en la que participa.

- Tema 23: **Sin conversiones**

Los grupos que tienen asignado el tema 23, deben prohibir cualquier operación (expresión, asignación, comparación, etc.) entre operandos de tipos diferentes.

Introducción

En esta etapa se va a realizar la generación de código intermedio en algunas de las estructuras vistas en la teoría.
Dado el enunciado la estructura a realizar es Árbol Sintactico, esta estructura será generada a partir de las reglas de la gramática y la detección de tokens, estos dos pasos ya descritos en las etapas anteriores.
También en esta etapa se deberá chequear todos los posibles errores sintácticos, estos errores pueden ser errores de no declaración, declaración de variables, función y etiquetas. Se deberá chequear que los tipos de la operación, comparación, llamada a funciones sean los que corresponde, también se chequeará que la variable, función, etiquetas estén declarada en el ámbito en que se las está invocando. Y luego para cada uno de estos errores se deberá informar al usuario en qué línea se encuentran los mismos y cuales fueron.

Descripción de la Generación de Código Intermedio

El Árbol Sintactico es una estructura la cual puede contar con NodoComun , NodoControl o NodoHoja. Los tres tipos de nodos tienen en común un Lexema, Línea y Tipo, luego cada tipo cuenta con algunas particularidades.
El NodoComun cuenta con dos hijos uno a izquierda y otro a derecha.
El NodoControl es el nodo que se va a encargar en un futuro del tratado de la bifurcaciones, este cuenta con un solo hijo a la izquierda.
El NodoHoja no cuenta con hijos y se denomina un nodo de finalización de rama.
Una vez explicado a la brevedad como será la estructura utilizada, pasaremos a explicar cómo se generara el Árbol Sintáctico.
La forma en la que se genera el árbol es mediante la detección de las reglas de la gramática. Cada vez que se detecta una regla en particular se genera el nodo correspondiente a la regla, ya sea este un NodoComun, NodoControl o NodoHoja, y luego de manera recursiva se van incluyendo en las reglas más abarcativas hasta formar el árbol.
En el caso de las sentencias de control lo que se hace es, se generan nodos de control para el manejo de la bifurcación, ya que el código se debe ejecutar dependiendo de una condición que será verificada en ejecución. Estos nodos de control se deberán reconocer en la generación de código y generar las instrucciones de assembler correspondientes.

La raíz del árbol generado será un Nodo Control el cual contará con un solo hijo que serán todas la árboles generados para las diferentes sentencia que se detectaron en la entrada ingresada por el usuario. Entonces el árbol generado se verá de la siguiente manera.

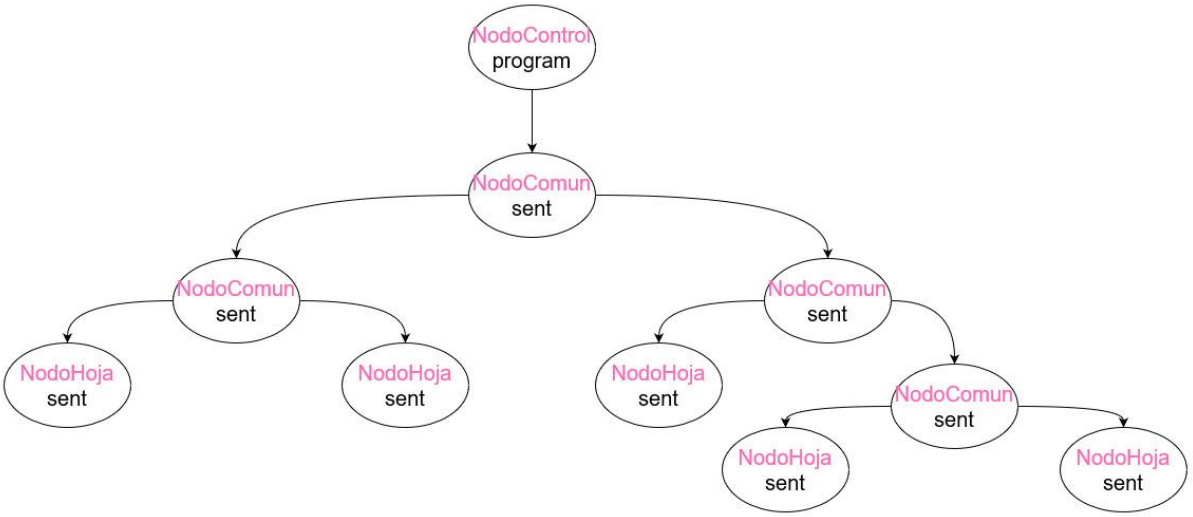


Imagen representativa del árbol generado para una entrada.

Luego cada una de esas sentencias que se ven en el árbol puede ser alguna de la sentencias permitidas para la gramática del lenguaje. Para las que notamos que son las más importantes o con una complejidad mayor explicaremos cómo se genera el árbol correspondiente.

Ejemplo de IF

En la sentencia de control IF se genera un NodoComun que cuenta por un lado con la condición y por el otro con otro NodoComun en el cual se encuentran dos NodoControl que van a ser la rama del THEN por la izquierda y el ELSE por la derecha. A continuación se verá una imagen explicativa.

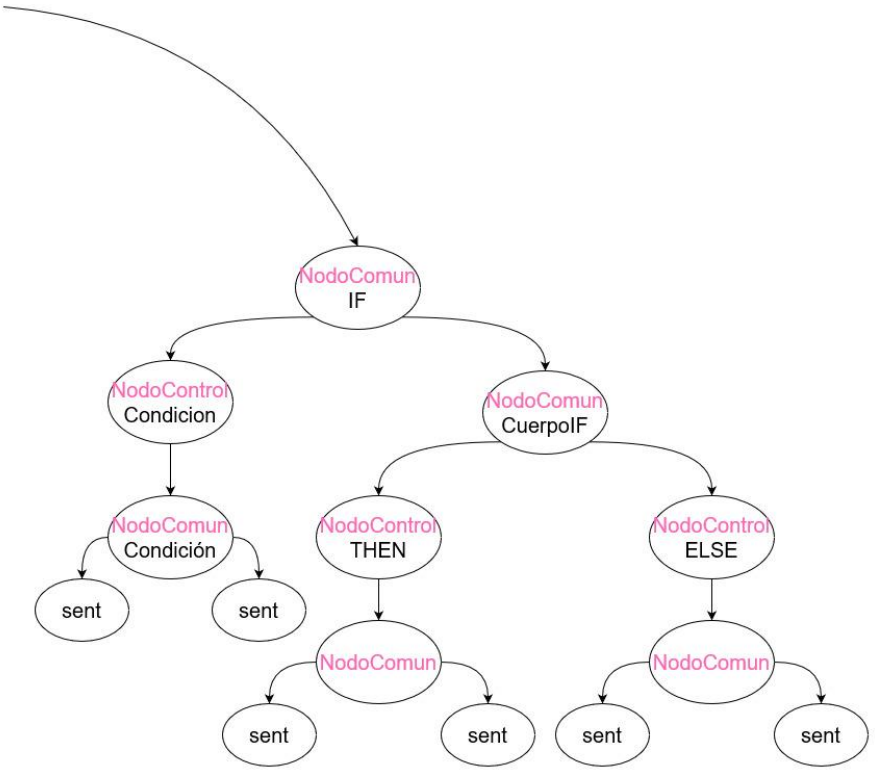


Imagen representativa de árbol generado para la sentencia if.

Ejemplo WHILE

En el caso del While lo que se hace es separar la condición del cuerpo propio. Dejando paso así para el control de los saltos a la hora de realizar el código assembler. Es entonces que se decide implementar al While como un NodoComun.
El mismo cuenta con un NodoComun que por una lado cuenta con la condición y por otro con otro NodoComun en el contiene el Cuerpo del while y la asignación. En la siguiente imagen se podrá ver como quedaría en forma de árbol.

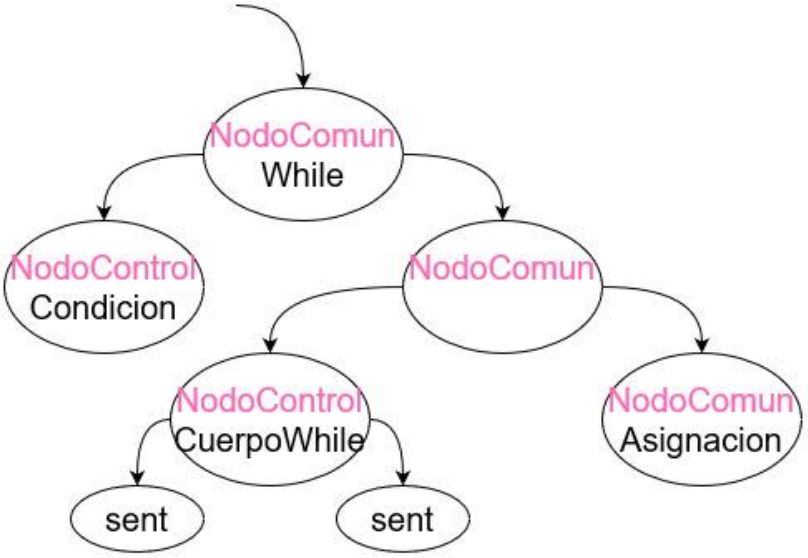


Imagen representativa del árbol generado para la sentencia while.

Ejemplo WHILE con Etiqueta

En esta sentencia se generará un NodoComun el cual contará con la etiqueta a la izquierda y el árbol del while en sí a la derecha, y el árbol se verá de la siguiente manera.

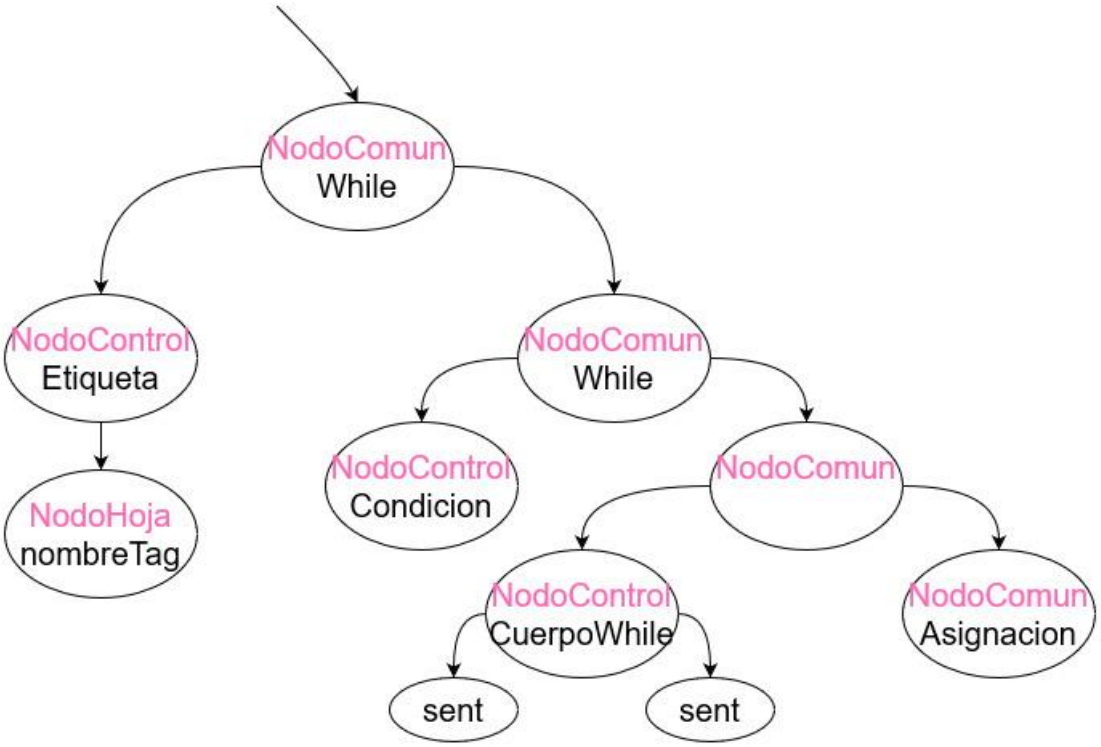


Imagen representativa del árbol generado para la sentencia while con etiqueta.

Ejemplo WHEN

El árbol que se genera en la sentencia when dependerá de si la condición del when sea verdadera o falsa. Para poder realizar esto en tiempo de compilación se optó por que en la condición solo se podrá tener constantes ya que estas son las únicas variables para las que se sabe su valor en tiempo de compilación luego una vez que se sabe que la condición fue verdadera de agrega el árbol correspondiente el cual será un Nodo Control que tendrá como hijo a las sentencias incluidas en el when, viéndose el árbol de la siguiente manera.

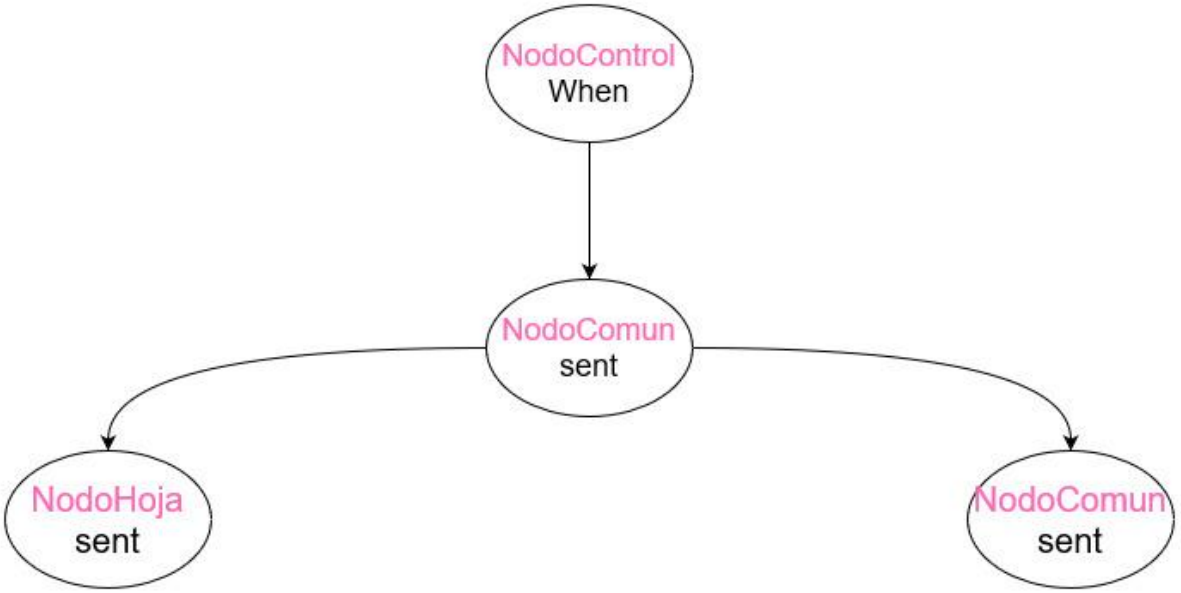


Imagen representativa del árbol generador para la sentencias when

Ejemplo Funciones

En el caso de las funciones se optó por generar árboles aparte para cada una de las funciones que se encuentran en la entrada, esta decisión se debe a que para cada una de las funciones se debe generar el código assembler correspondiente y ubicarlo en una sección diferente dentro de lo que sería el código assembler final. Entonces se tendrá una lista con los árboles de cada una de las funciones, cada uno de esos árboles se generará de la siguiente manera, será simplemente un NodoControl el cual tendrá como hijo todas las sentencias que se realizan dentro de la función.

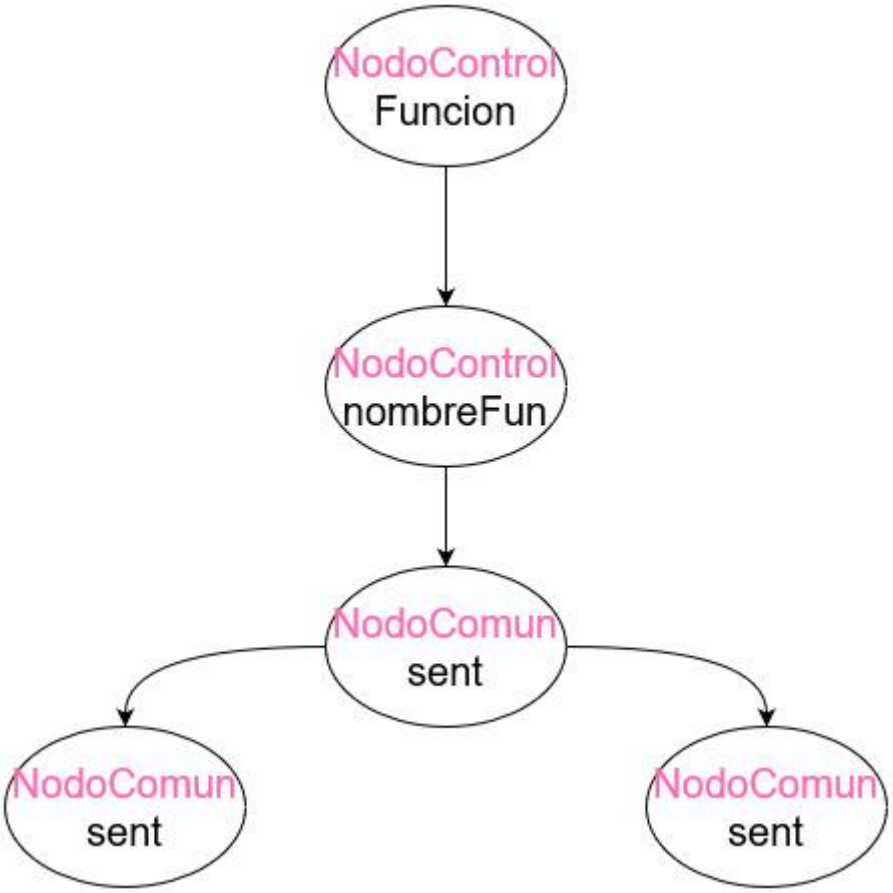
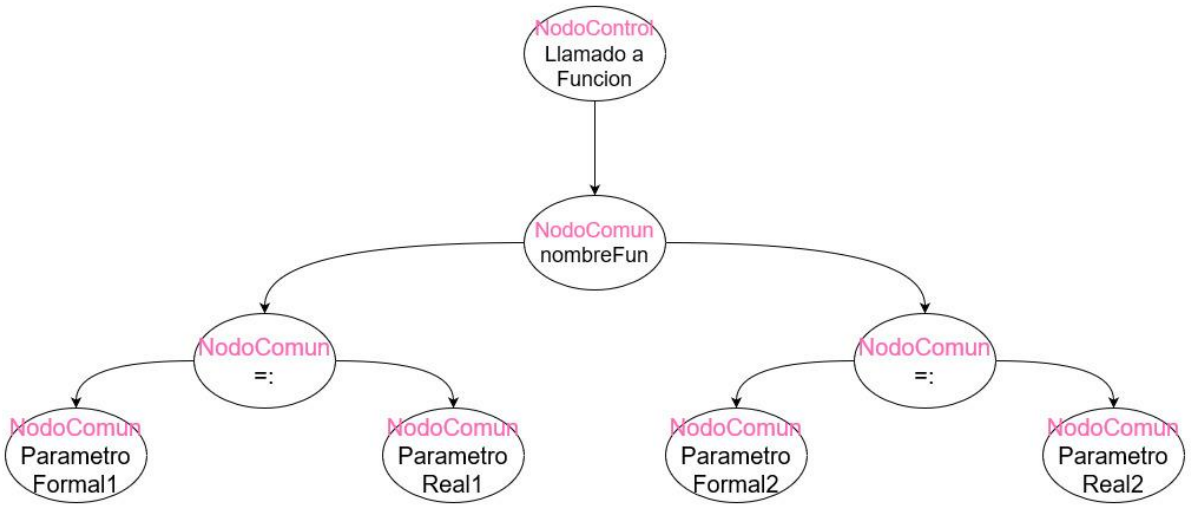


Imagen representativa del árbol generado para la sentencia funcion.

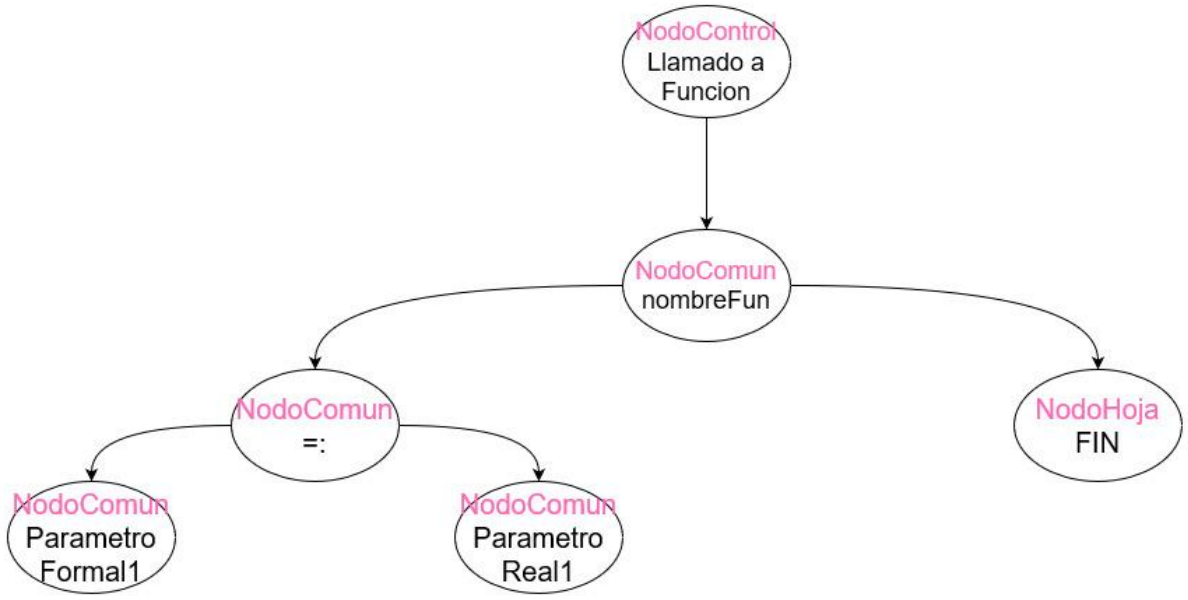
Ejemplo Llamado a Funcion

Para generar el árbol para la sentencia de llamado a función, se optó por generar un NodoControl el cual cuenta con el nombre de la función a la que se está llamado y en el caso de la se llame a una función con dos/un parámetro/s se optó por generar el árbol para las asignaciones correspondientes a la copia de los parámetros reales en los parámetros formales, se realizó así para que a posteriori se facilitará la generación de código assembler.
Entonces estos serían los tres posibles árboles para el llamado a funcion

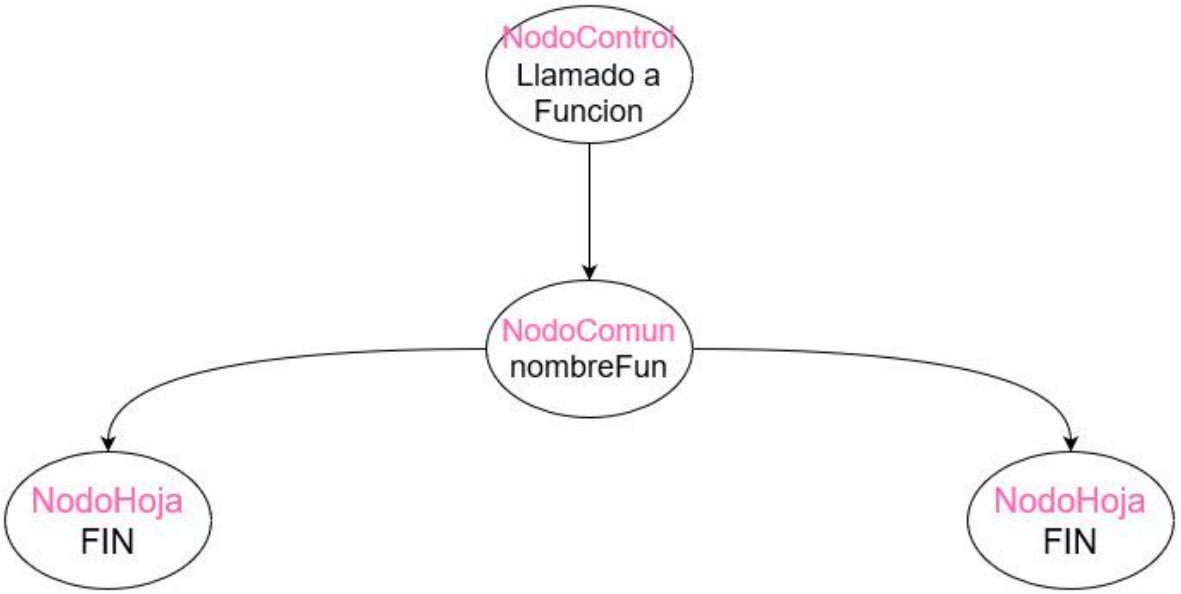
Llamado a función con dos parámetros



Llamado a función con un parámetro



Llamado a función sin parámetros



Errores semánticos considerados

Los errores considerados por nosotros fueron:

- Funciones no declaradas, funciones re declaradas , variables no declaradas, variables redeclaradas, etiquetas no declaradas,etiquetas redeclaradas, todos estos errores son tenidos en cuenta en el ámbito en el que se intenta invocar una función o utilizar una variable o etiqueta.
- En el caso de los retornos de la funciones debe ser del mismo tipo que el retorno que debe devolver la función y no se puede escribir una sentencia de retorno en el ámbito global.
- En el caso de las asignaciones, expresiones, términos y comparaciones no se permite realizar operaciones entre tipos diferentes, ya que contamos con la consigna sin conversiones
- Luego en las llamadas a funciones los tipos de los parámetros reales deben ser los mismos que cuando se declaró la función y la cantidad también debe ser la misma.

Implementación de errores semánticos considerados

A continuación se pasará a explicar cómo fueron detectados a comunicados cada uno de los errores considerados antes planteados

No declaracion, declaracion de etiquetas, función y variables.

Para poder verificar esto primero se debe saber en ámbito se está intentando llamar una función o utilizar una variable o una etiqueta, entonces contamos con una variable global dentro del parser que cuenta con el ámbito actual, está variables se modifica cada vez que se ingresa en una declaración a una función y cuando se sale del mismo. Luego cuando se detecta una declaración ya sea de función, etiqueta o variable, se la agrega a la tabla de símbolos utilizando name mangling, esto es cambiarle el nombre a la función, etiqueta o variable por Nombre@ambito.
Luego para el caso de un llamado a una función se chequea que la misma está declarada en el ámbito desde donde se está llamando o en los ámbito padres.
Cuando se intenta declarar una función se chequea que la misma no esté declarada en ese ámbito
Cuando se utiliza una etiqueta o variable se chequea que la misma esté declarada en ese ámbito o en los ámbitos padres.
Cuando se declara una variable o etiqueta se chequea que la misma no esté declarada en ese ámbito.

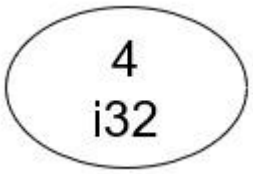
Funciones con retornos

Para este chequeo se cuenta con una pila de variables boolean. Lo que se hace es cada vez que se declara una función se apila en el tope de la pila un false, luego en el caso de que se encuentre una sentencia de retorno, se cambia el tope de la pila por true, para luego cuando se cierre la funcion se podra chequear preguntando si el tope de la pila es true y se sabre si hubo o no retorno. En este caso es de utilidad el uso de la pila ya que se puede contar con una anidamiento de función por lo que los booleanos se pisarían, en cambio utilizando una pila los boolean solo se agregan el tope y una vez termina la función se desapilan.

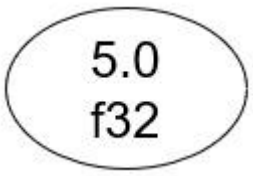
Chequeo de tipos en asignaciones, expresión y términos.

Para este chequeo lo que se utilizó es la estructura con la cual se genera el árbol, ya que el árbol se genera desde la reglas más básicas a la más complejas, esto puede ser utilizado para

el chequeo de tipo. En cada uno de los nodos intermedio de ira guarda el tipo entonces en las reglas más abarcativas se chequea que ambos nodos son del mismo tipo.
Se puede ver mejor en las siguientes imágenes.
Primero se genera el nodo para el factor y se le agrega el tipo



Luego se genera el nodo para el siguiente otro factor



Después cada factor se vuelve término y se pasa el mismo nodo y luego cuando se tiene la regla Expresion : termino SUMA término
se chequea que los tipos de ambos términos sean iguales en este caso no pasaría.
Este tipo de chequeo se realiza de la misma manera para Asignaciones, expresiones, términos y comparaciones

Chequeo de tipos en llamado a funcion

Para realizar este chequeo lo que se hace es cuando se declara una función se agrega a la misma entrada que tiene la función en la tabla de símbolos dos atributos los cuales son las dos entradas a la tabla de símbolos de los parámetros de la función en el caso de que la función tenga dos parámetros, en el caso que tengo uno parámetro solo se agrega ese solo y el caso que no tenga no se agrega nada.
Luego en el llamado a función se chequea que el llamado que se está realizando sea la misma cantidad que los parámetros agregados en la tabla de símbolos para la función específica. Una vez que se cumple esto se chequea que es tipo de los parámetros esté correcto, ya que en la tabla de símbolo se encuentra el tipo de los parámetros formales.

Parte Cuatro : Generación de Código

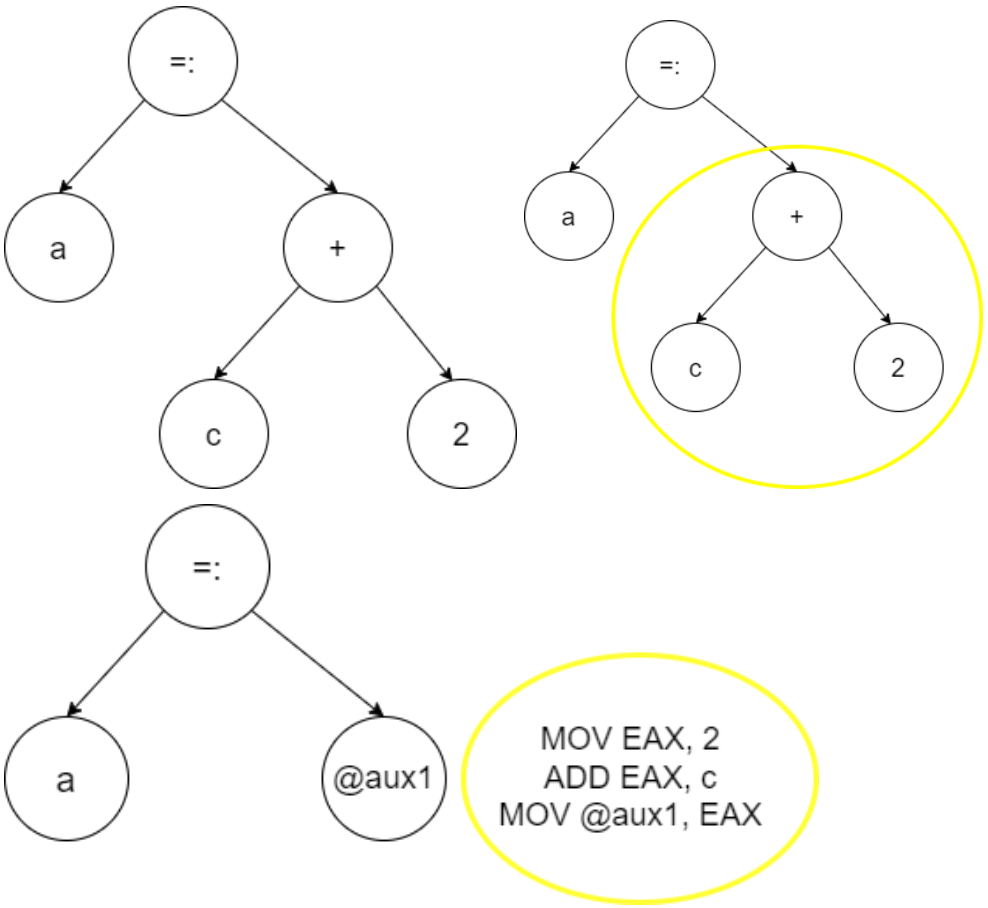
Controles en Tiempo de Ejecución

Objetivo: Generar código Assembler, a partir del código intermedio generado en el Trabajo Práctico Nº 3. El mecanismo para generar código será el de variables auxiliares. El código Assembler generado debe poder ser ensamblado y ejecutado sin errores.

Los temas particulares a cumplir son los siguientes:
a) División por cero para datos enteros y de punto flotante: El código Assembler deberá chequear que el divisor sea diferente de cero antes de efectuar una división. Este chequeo deberá efectuarse para los dos tipos de datos asignados al grupo.
d) Overflow en productos de enteros: El código Assembler deberá controlar el resultado de la operación indicada, para el tipo de datos entero asignado al grupo. Si el mismo excede el rango del tipo del resultado, deberá emitir un mensaje de error y terminar.
h) Recursión mutua en invocaciones de funciones: El código Assembler deberá controlar que si la función A llama a la función B, ésta no podrá invocar a A.

Introducción

La cuarta parte del trabajo consiste en la generación de código Assembler a partir de código intermedio realizado en la etapa anterior según la estructura indicada (en este caso árbol sintáctico).
Para poder generar el código Assembler se utilizará el mecanismo de variables auxiliares, el cual consiste en el guardado de resultados de operaciones en las propias variables auxiliares pudiendo generar sin límite las requeridas. Por consiguiente el proceso de generación de código será recorrer la estructura de código intermedio antes mencionada, buscando el subárbol izquierdo inferior y a partir de él crear una variable auxiliar conteniendo el resultado de la operación. Ésta variable reemplazará el propio subárbol, y a su vez generará las debidas instrucciones de código assembler descriptoras de la operación que se genera para almacenar el resultado en la propia variable.
A continuación se ejemplifica la aplicación de variables auxiliares gráficamente.



Por último, anteriormente se mencionó la traducción a ‘assembler’ ya que se seleccionó la opción de X86-Assembly o Assembler para Pentium como lenguaje para la salida. Se tenía la opción también de utilizar Web-Assembly, sin embargo el acuerdo interno decantó por Assembler para Pentium, cuya sintaxis se encuentra brindada por apuntes de la cátedra además de material adicional encontrado en el siguiente link (https://www.aldeid.com/wiki/Category:Architecture/x86-assembly).

Desarrollo

El código assembler cuenta con tres partes principales:
-Bibliotecas: en esta sección se incluyen todas las bibliotecas requeridas para correr las directivas de assembler.
-Datos(.data): incluye todos los datos inicializados, lo que se hace en esta sección es volcar el contenido recolectado en la tabla de símbolos. Si un dato no posee valor entonces se inicializa con ‘?’.
-Código (.code): incluye todo el código generado a partir de la estructura contenedora de la generación intermedia de código. El código principal va seguido de una etiqueta ‘start’ (en nuestro caso ‘main’) y finaliza con ‘end start’ (nuestro caso ‘end main’).

Para ésto se creó una nueva clase encargada de generar todas las partes del código, ensamblarlas y devolverlo. Esta clase es llamada 'GeneradorAssembler', al cual se le pasa para su construcción el parser, y a partir de éste toma el árbol sintáctico ya generado junto con los respectivos árboles sintácticos de las diversas funciones si es que las hay en la entrada.

Para comenzar, la parte de las bibliotecas es tan simple como una concatenación en forma de 'string' de las bibliotecas a utilizar. Éstas fueron suministradas por el apunte brindado por la cátedra "Windows Assembly Programming Tutorial", el cual indica la funcionalidad de cada una de éstas.

En la generación de los datos (.data), con la tabla de símbolos ya generada facilitó mucho porque consistió en una transcripción de cada dato de la tabla a ésta parte, los datos que no poseen valor inicial son inicializados con el signo '?', indicando que aún no poseen valor.
La cualidad que favorece todo fue el hecho de poseer todos datos de 32 bits, ya que simplificó mucho la definición como también la manipulación de estos datos en registros de mismos tamaños. Utilizando 'dd' (define double word - dw, double word contiene 16 bits) para la definición de datos de 32 bits, y utilizando 'db' (define byte) para los mensajes de error en la división por cero, overflow, etc.
Además, en esta parte se incluyen

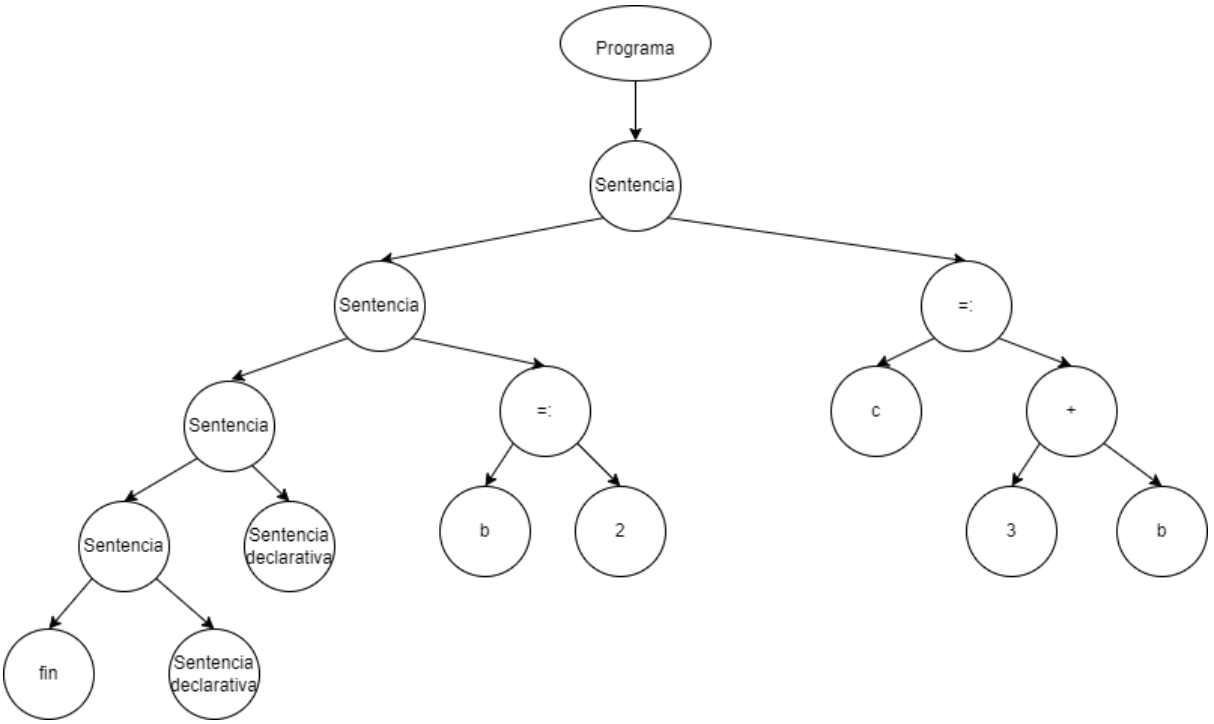
Para lograr la parte de código (.code), la idea principal consiste en el recorrido de la estructura del árbol sintáctico la cual contiene generado el código intermedio.
El recorrido es post-order, esto quiere decir que se buscará descender en búsqueda del subárbol izquierdo inferior. Esto para poder aprovechar la estructura recursiva en beneficio para la generación de código.
A medida que se visiten los nodos se irá generando el código debido respecto del lexema que contenga el nodo en cuestión (por ejemplo, en caso de ser una suma se generará la respectiva instrucción de 'ADD' entre los resultados de sus dos hijos, en cambio si es una asignación se generará la respectiva instrucción 'MOV' para asignar el resultado del nodo derecho, al nodo izquierdo).
Como antes se mencionó, en la estructura del árbol sintáctico, se cuenta con tres tipos de nodo con mayor importancia que heredan de la clase 'ArbolSintactico': 'NodoComun' para la representación de nodos intermedios con dos hijos; 'NodoControl' para la representación de nodos intermedios con el objetivos de anunciar la presencia de sentencias específicas como lo puede ser un continue, o en el caso del 'If' su respectivo 'then' y 'else' si es que los posee; por último el tercer tipo de nodo es 'NodoHoja' encargado de indicar los finales de las "ramas" de los árboles ya sea por ejemplo con una variable, constante o con un lexema "Fin" indicando que la rama finaliza.
Partiendo de esta base, el planteo de la generación de código se hace principalmente desde los nodos intermedios, principalmente todos aquellos que sean 'NodoComun' ya que estos van a contener el mayor porcentaje de acciones a resolver como lo puede ser por ejemplo asignaciones, expresiones aritméticas, sentencias if, sentencias while, etc. Sin perder de vista los nodos intermedios que sean 'NodoControl' ya que estos también advierten posibles

variaciones como por ejemplo el caso que una sentencia ‘If’ posea la rama ‘else’ o también el caso en el que exista una sentencia ‘break’ o ‘continue’ derivando en diversas acciones respectivas a la sentencia en cuestión. Limitando así a las hojas creadas con ‘NodoHoja’ a únicamente brindar su contenido (lexema) a su padre mediante un método (getLexema()) para que éste último pueda realizar el código assembler según el contexto en el que se encuentre.
De esta forma basta con generar dentro del nodo una función encargada de generar su respectivo código assembler según su propio lexema. Por ejemplo a continuación se explica con acompañamiento gráfico la generación de una asignación a medida que se recorre el árbol:

Supongamos que se cuenta con el siguiente programa:

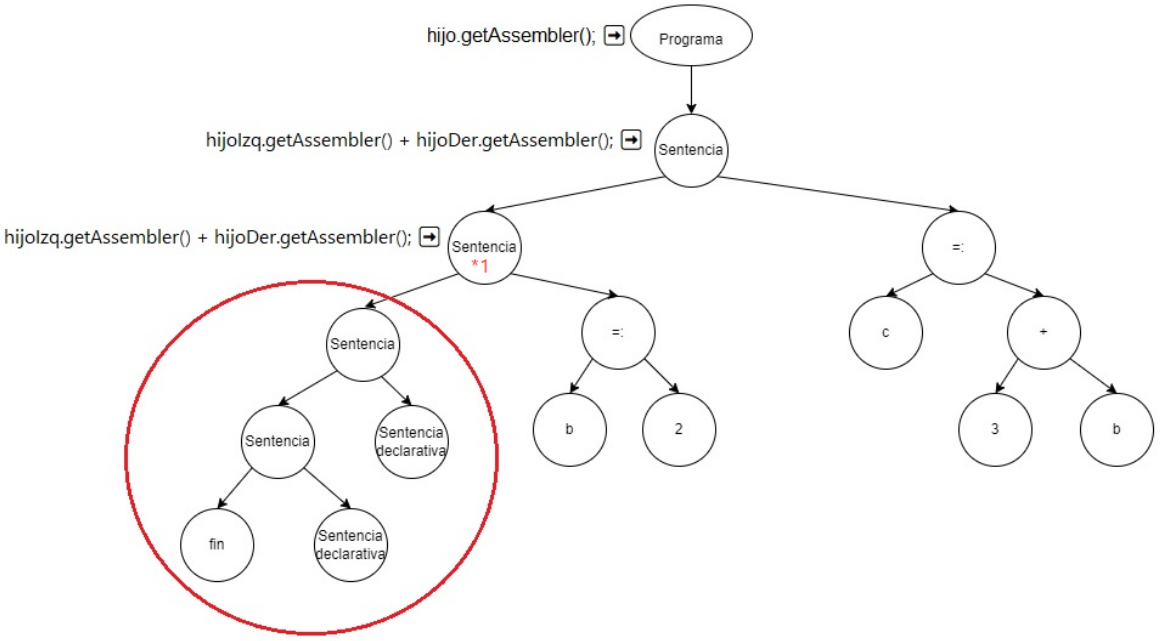
```
program {
	i32 b;
	i32 c;
	b =: 2;
	c=: 3 + b;
}
```

El árbol sintáctico generado, según se explica en la etapa 3 será el siguiente:

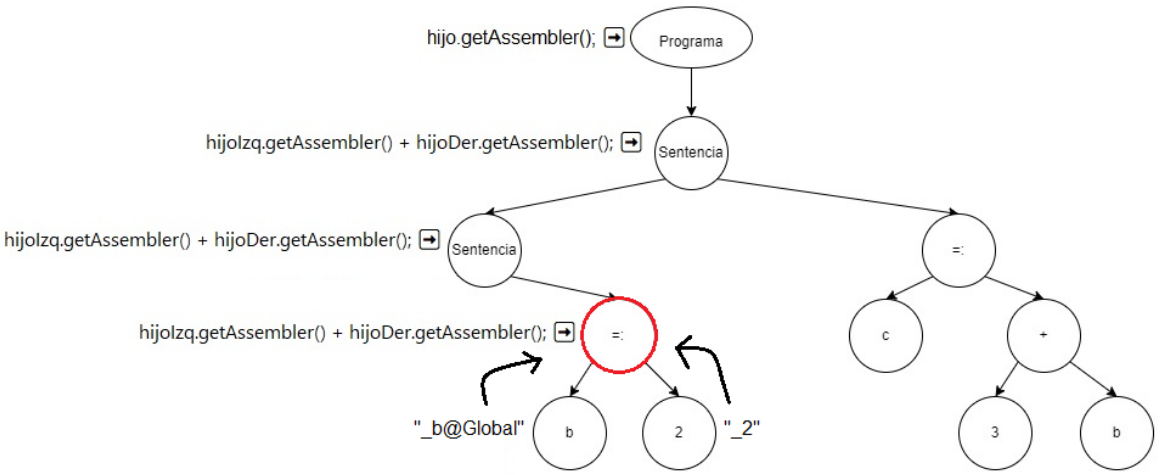


Como se mencionó anteriormente, se recorrerá el árbol de manera tal que cada nodo solicitará a su/s hijo/s su respectivo assembler, por lo que en primer lugar el nodo “Programa” cuyo tipo es NodoControl, se encargará de requerir a su hijo “Sentencia” que le devuelva su assembler. Por su parte, “Sentencia” hará lo mismo con sus dos hijos, primero requiriendo a

su hijo izquierdo “Sentencia” y luego a su hijo derecho “=:”. De esta forma se logrará el recorrido del árbol ‘de izquierda a derecha’:



Como se puede observar, el recorrido natural a izquierda llega al punto en el cual se llega a los nodos de declaración de las variables “a” y “b”, como éstas no son parte de la generación de assembler, su retorno será nulo. Ésto se logra gracias a que los nodos que no poseen un lexema determinado, la devolución de código assembler es naturalmente vacía.
Por lo tanto la devolución de todo el circulo contorneado de color rojizo, será nula y el nodo padre “Sentencia”(*1) una vez que reciba ésto, procederá a sumarselo a la devolución que obtenga posteriormente de la asignación por parte de su hijo derecho:



Como se puede observar, el NodoComun con el lexema de “=:” (asignación) tendrá el compromiso de devolver a su respectivo padre el assembler para asignar a la variable “b” un “2”.

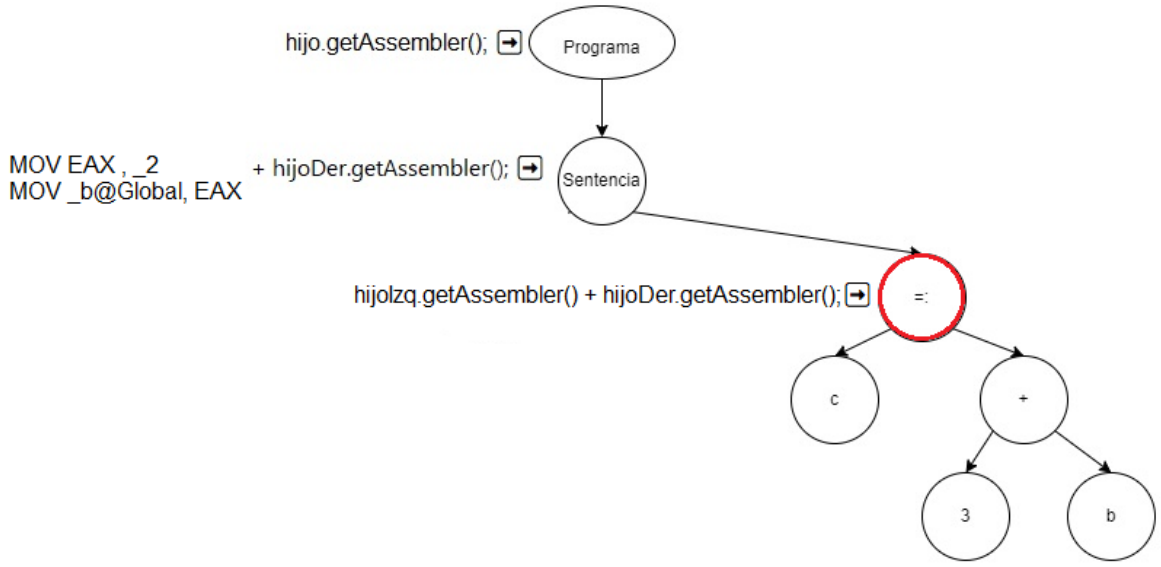
Ésto se hace gracias a que antes de solicitar el código assembler a sus hijos, se verifica el lexema que contiene el nodo en cuestión.
Como es una asignación lo que se procede a hacer es, primero que nada verificar el tipo con el que se está tratando (‘Entero’ o ‘Doble’) ya que dependiendo de éste, las instrucciones van a ser de una forma u de otra. Como la solución a intentos de asignaciones entre tipos distintos se resuelve en la gramática, únicamente basta con saber el tipo de uno de los dos operadores de la asignación.
En éste caso es de tipo entero por lo que las instrucciones que se generarán serán para enteros, en caso contrario se deberán utilizar instrucciones para flotantes descriptas también en la biografía dada por la cátedra.
En éste punto solo queda aclarar una duda. ¿Cómo sabe el nodo en cuestión que a la variable “b” lo que se le asigne está ya definido y listo para ser devuelto por su hijo derecho?. Para solucionar este inconveniente lo que se hizo es adicionar a la clase ‘ArbolSintactico’ un nodo “extra” además de sus dos nodos hijos, el cual iba a contener el resultado de su subárbol para que a la hora de efectuarse la asignación, el nodo encargado únicamente consulte por el lexema de ese nodo “extra” llamado ‘HojaPropia’. En caso de ser una Hoja el lexema que devolverá su hoja propia será el propio de la hoja, en cambio si es un nodo intermedio, para devolver el lexema de su hoja propia deberá esperar a que sus subárboles retornen sus resultados a partir de sus debidas hojas propias.
De esta forma el nodo de asignación lo que hace es devolver a su nodo padre instrucciones para el tratado de Enteros de la siguiente forma:

```
retorno = "MOV EAX , " + getDer().getHojaPropia().getLex() + "\n";
retorno = "MOV " + getIzq().getHojaPropia().getLex() + ", " + "EAX" + "\n";
```

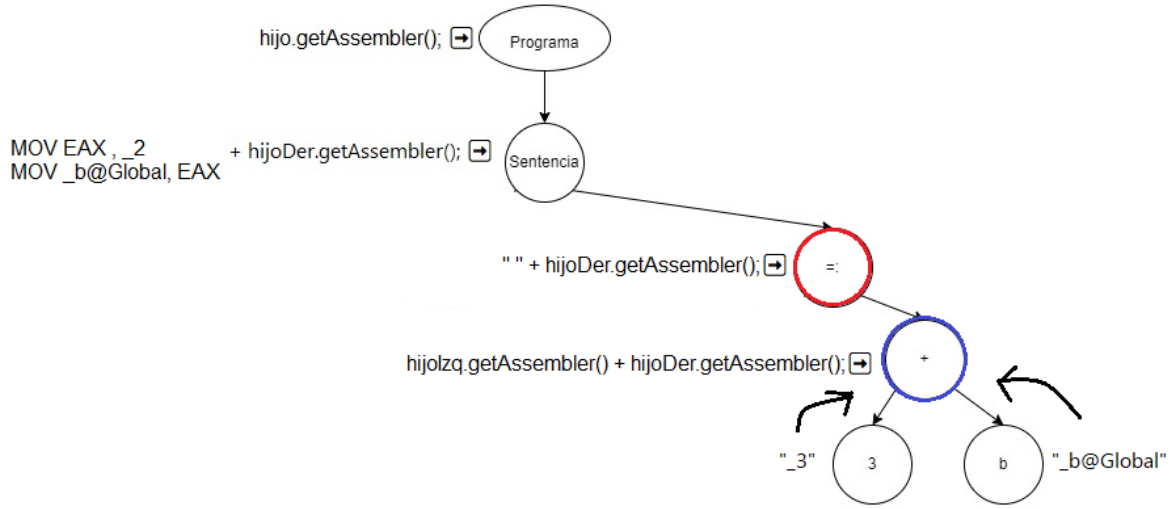
Quedando el retorno así:

```
MOV EAX , _2
MOV _b@Global, EAX
```

Una vez retornado el assembler de la asignación, la recursión seguirá su curso:



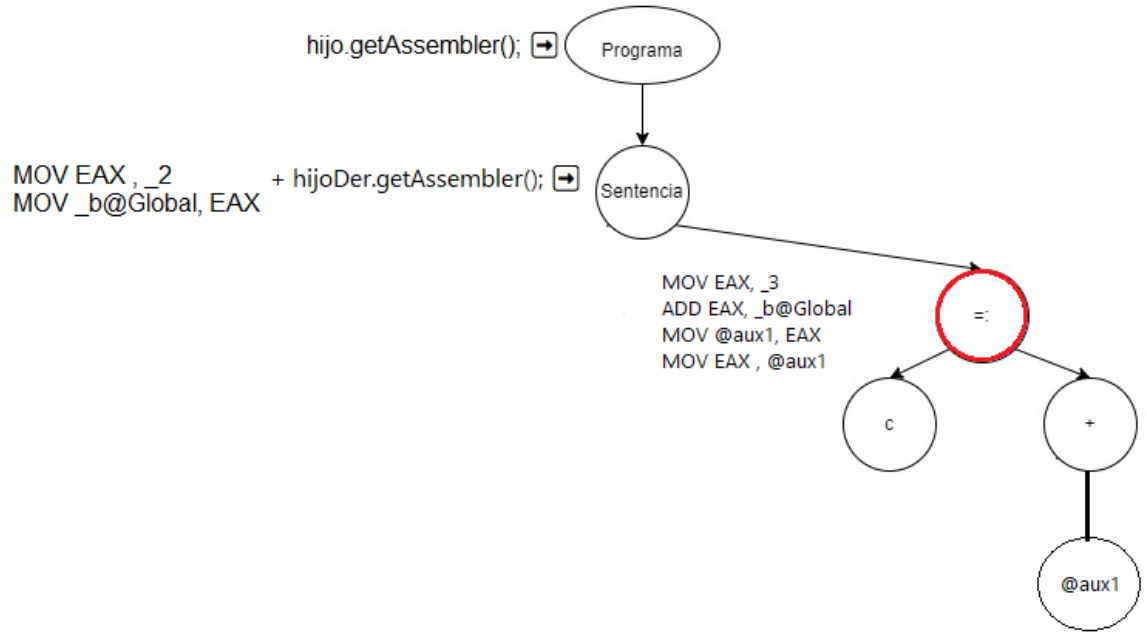
De la misma forma que la asignación anterior, el nodo en cuestión seguirá la recursión a izquierda y luego a derecha para que cada nodo que no es hoja genere su 'HojaPropia' y la asignación pueda generar sus instrucciones:



Como se puede observar, de la misma forma pasa con la suma, pidiendo el tipo a uno de sus hijos y luego solicitando el lexema de las hojas propias de sus hijos. De esta forma el NodoComun de la suma generará el siguiente assembler:

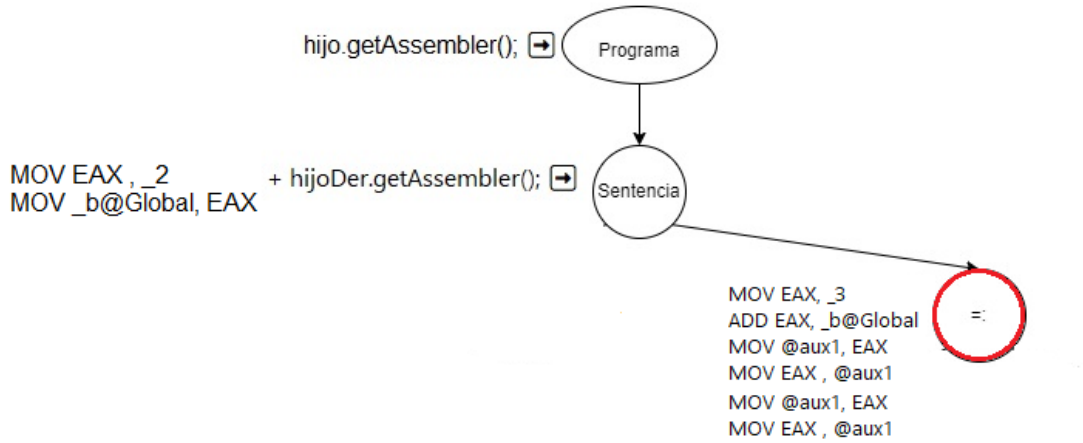
```
MOV EAX, _3
ADD EAX, _b@Global
MOV @aux1, EAX
```

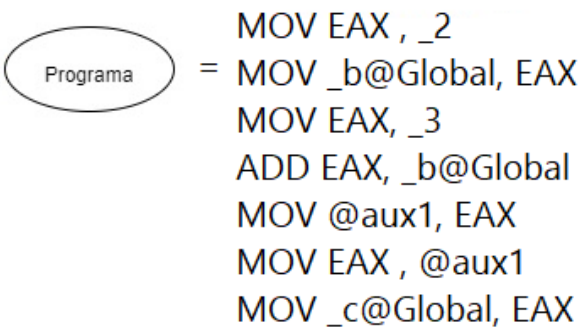
Además de esto colocará en su nodo hoja el lexema a una variable auxiliar, para que luego la asignación pueda retomar el valor de la suma efectuada en el registro EAX.



De esta forma lo que le queda efectuar a la asignación es con el lexema de su hijo izquierdo y el lexema del nodo hoja de su hijo derecho, generar las instrucciones de siempre:

```
MOV EAX , @aux1
MOV _c@Global, EAX
```





Como se explicó en el ejemplo la forma de guardado en variables auxiliares consta de la adición de un nodo extra llamado “hojaPropia”, el cual contendrá como lexema el nombre de la variable auxiliar propia y a su vez no poseerá hijos.
De esta forma cada vez que se solicite a un nodoHoja que retorne su “hojaPropia”, se retornará a ella misma. Por lo tanto cuando se quiera saber el lexema del hojaPropia de una hoja, básicamente se estará solicitando el lexema a la hoja misma.
Así es que el manejo de solicitud de lexemas de hojas hijo, se administra de igual forma para todos los nodos. Pudiendo también, ampliar el comportamiento a la utilización de variables auxiliares como se explicó en el anterior ejemplo dentro de los nodo hojaPropia.

En resumen, la generación de código assembler se basa en el recorrido del árbol sintáctico, evaluando el lexema del nodo tratado, seguido de la solicitud de código assembler de sus hijos, para luego adicionar las debidas instrucciones assembler dependiendo del nodo en cuestión.

Una vez resuelta la generación a partir del árbol resta generar las funciones, solucionando los respectivos saltos a ellas.
Para la solución únicamente basta con recorrer la lista contenedora de Árboles que describen las funciones, dejando su representación assembler previo al código “main”

Para la resolución de temas particulares se implementaron saltos a diversas etiquetas contenedoras del error detectado.
En caso de la división por cero ya sean flotantes o enteros, simplemente bastó con comparar el divisor con 0, para los enteros utilizando la instrucción CMP, el divisor y el valor inmediato ‘0’; para los flotantes en cambio, se debió comparar el tope de la pila con 0 utilizando la instrucción FTST. Si el divisor es 0, entonces se saltará a la etiqueta “errorDivisionPorCero”.
En caso de el producto de enteros con overflow, lo que se hace simplemente es saltar a la etiqueta de “errorProductoEneros” mediante la instrucción ‘JO’ que salta si es que existe un overflow.
Para el tercer caso, se debía controlar la imposibilidad de la recursión mutua. Se logró gracias a la ventaja de las variables de retorno de las funciones, ya que cuando se invoca una función propia, la variable auxiliar de retorno se almacena en una pila. Entonces la mecánica consta de verificar si hay variables en esa pila, si las hay quiere decir que actualmente nos encontramos dentro de una función, por lo que debemos verificar que a la hora de invocar una próxima función se verifique que no sea la misma que la anterior. Por lo que, solo queda utilizar una variables llamada “tagAnterior” la cual contiene el contenido de la función visitada previamente, al comparar el nombre de una función invocada con esta variable, si es que son diferentes, entonces se permite y actualiza el contenido de la variable “tagAnterior”. De lo contrario se saltará a una etiqueta llamada “errorRecursionMutua”.
Todas estas etiquetas de error lo que hacen es finalizar el programa indicando la finalización no exitosa.

Complicaciones

Las primeras complicaciones surgen con los saltos, ya que siempre que se encuentre una condición (excepto en el when) se debe evaluar en tiempo de ejecución y verificar si se toma el camino por cumplimiento o caso contrario otro camino por incumplimiento.
Para solucionar esto lo que se decidió utilizar un sistema de pilas de tags o etiquetas. En principio la idea fue que el nodo encargado de tomar la decisión generaría la etiqueta, la utilizaría para saltar allí y luego el nodo padre eventualmente se encargaría de extraer la etiqueta de la pila y colocarla previa al código de sus hijos. Sin embargo esta metodología no tardó mucho en expirar, ya que requería afectar el orden en el cual se efectúa la recursión (por ejemplo realizando primero la recursión a derecha para la generación de etiquetas almacenando su resultado en una variable; luego realizando la recursión a izquierda, adicionando etiquetas si fuera necesario y finalmente agregando el resultado de la previa recursión a derecha al final). Cabe aclarar que la única sentencia que se realizó con esta

estructura es el ‘If’ ya que se requiere saber previamente si en su cuerpo existe un else o no, porque dependiendo de esto, el salto condicional se efectuará, en caso de haber else al inicio de la rama del else; en caso contrario al final del cuerpo del if.
La solución es simple y hasta mucho más natural que la antes planteada, los nodos encargados de la generación de etiquetas son los padres (por ejemplo el nodo con el lexema ‘while’, etc) de ésta manera aquel nodo que requiera el conocimiento de una etiqueta para identificar dónde saltar, bastará simplemente con consultar el tope de la pila destinada a ello (pila de labels, pila de variables auxiliares, pila de labels de breaks, pila de labels de continues, etc).

Continuando con problemáticas, los saltos en caso de encontrar breaks y continues.
Para el caso de los breaks, si la sentencia no estaba en un contexto del lado izquierdo de una asignación, no hay mucha dificultad ya que es simplemente hacer un salto al final de la sentencia. La complicación surge cuando se está dentro de una asignación, por ejemplo el caso de un while, en el cual deben dar un retorno conteniendo un posible break en su cuerpo principal y siempre en su rama ‘else’.
La solución surge partiendo de la gramática, ya que se tuvo que diferenciar el contexto en el que esté la sentencia (para la sentencia while), indicando en el árbol sintáctico diferentes nombres, es por eso que se cuenta con por ejemplo los nodos con lexema “while como expresión”, indicando que en este caso el contexto es si o si dentro de una asignación y se contiene una rama “else” seguida de un retorno en caso de no cumplir la condición.
Para poder dar retorno lo que se hizo es generar dentro de estos nodos padre (“while como expresión”) una variable auxiliar la cual va a contener el valor de retorno, de esta forma si existe un break en el cuerpo, la variable auxiliar se utilizará para almacenar este valor. De igual forma cuando se toma el camino del ‘else’.
Para el caso de los ‘continue’, la complicación surgió al observar que si se tiene una etiqueta, el iterador que debe incrementar/decrementar su valor es el propio de la etiqueta con la que se quiere continuar. La solución parte, de nuevo, en un replanteo desde la gramática, incluyendo ahora el nodo ‘continue’ como un ‘NodoComun’ (previamente era un nodo de control únicamente con la finalidad de indicar que existía un continue), de esta forma se podrá colocar a su izquierda el iterador a incrementar y a su derecha el tag de la etiqueta a la que se quiere saltar si así se desea, caso contrario la hoja de tag contendrá un lexema ‘Fin’ y el iterador en la derecha del ‘continue’ será el propio de su estructura.
De esta forma el problema queda mitigado, ya que el flujo natural será recorrer hacia la izquierda, aumentando naturalmente el iterador que se encuentre, siguiendo con su rama derecha saltando a la etiqueta indicada en caso de que el lexema sea distinto de ‘Fin’.

Para finalizar, los códigos generados fueron seguidos mediante la herramienta MASM32 junto con el debugger OllyDbg (nuestro caso la versión v1.10). De esta forma se podía ver visualmente como los registros seguían las órdenes impuestas por la entrada luego de traducir la salida en assembler.

Aclaraciones y Excepciones

Si bien se cumplen las funcionalidades requeridas, cabe aclarar que el compilador contiene diversas carencias que afectan su uso y no alcanzan ni por proximidad a un compilador real del día a día. Es por eso que a continuación se va a describir diversas aclaraciones:

- Si existe un error no cubierto por la gramática, se lanza un error general como lo puede ser “Se esperaba }”, “Se esperaba ;”, “Se esperaba)”,o “Falta nombre del Programa” junto con la línea donde existe el error real.
- El compilador detectará de a un solo error a la vez por sentencia. Si se posee una sentencia que contiene más de un error, se deberá ir solventando de a uno.
- A la hora de intentar declarar una función con el nombre de una variable, además de arrojar el error correspondiente, arroja otro error diciendo: “el retorno no es el mismo que el tipo de la función” además se tendrá otro error diciendo: “La función no cuenta con ningún retorno”.
- Si a una cadena le falta el caracter de cerrado “ ‘ “, todos los errores consecutivos que se tengan en la entrada se informarán una línea anterior a la del propio error.
- Si se tiene una sentencia ejecutable y dentro de esta se intenta poner una sentencia declarativa (por ejemplo la declaración de una variable dentro de un “WHILE”), se arrojarán errores inesperados como los mencionados en el primer punto.
- Si bien no se lo pide, cabe aclarar que cuando en una función determinada contiene un retorno (ya que si o si debe contener uno) pero éste último se encuentra por ejemplo anidado dentro de una condición “IF”, la gramática NO se encarga de arrojar error informando que la función puede llegar a no tomar un camino para retornar algo.
- Si bien se explicó en el punto anterior que la gramática no se encarga de eso, la generación de código sí. Haciendo que si por algún error del programador se llega al final de una función sin retornar nada, se salte a una etiqueta de error y se finalice con una excepción.
- Para el caso de la recursión mutua lo que se hace es la garantización de que si desde la función “a” se llama a la función “b”, en esta última no se podrá llamar a la función “a” mutuamente. Cabe aclarar que si desde “b” se llama a otra función “c” y desde ésta última se llama a la función “a”, esto no será controlado por el compilador.

Modo de uso y Casos de prueba

La manera en la que se debe utilizar el compilador es la siguiente:

- Se debe abrir una consola
- Dirigirse a la carpeta del trabajo TPE_Compiladores
- Una vez en la dirección se debe ejecutar el siguiente comando “java -jar TPE_Compiladores.jar”
- Luego de ejecutar este comando aparecerá un mensaje el cual indicará que se ingrese el nombre del archivo que quiere ser leído.

- Se debe ingresar uno de los posibles casos de prueba o confeccionar una entrada.

Los posibles archivos que se suministran para la ejecución son los siguientes:

- ☐ **alcanceVariablesConError.txt** muestra errores detectados relacionados con el alcance de cada variable
- ☐ **alcanceVariablesSinError.txt** muestra el uso correcto del alcance de las variables
- ☐ **ambitosEtiquetasConErrores.txt** muestra errores relacionados a las etiquetas
- ☐ **ambitosEtiquetasSinErrores.txt** muestra como se utilizan correctamente las etiquetas
- ☐ **analisisLexicoConErrores.txt** muestra las entradas pedidas para la primera etapa con errores detectados
- ☐ **analisisLexicoSinErrores.txt** muestra las entradas pedidas para la primera etapa sin los errores
- ☐ **cadenaConError.txt** muestra una cadena escrita de forma incorrecta
- ☐ **CasosPruebaSinError.txt** muestra las distintas sentencias escritas reconocidas sin errores
- ☐ **chequeoTiposConError.txt** muestra que errores se detectan en el chequeo de tipos
- ☐ **chequeoTiposSinError.txt** muestra el chequeo de tipos sin errores
- ☐ **entrada_sent_decl.txt** muestra sentencias declarativas implementadas
- ☐ **entrada_sent_eje.txt** muestra sentencias ejecutables implementadas
- ☐ **ErroresEtapa4.txt** muestra los errores que se captan en la etapa 4
- ☐ **ErroresVar.txt muestra** los errores captados para las variables
- ☐ **expresionConError.txt** muestra las distintas expresiones con los errores detectados
- ☐ **expresionSinError.txt** muestra las distintas expresiones sin errores
- ☐ **if.txt** muestra las distintas sentencias if sin errores
- ☐ **operacionesVarConError.txt** muestra operaciones de las variables con errores detectados
- ☐ **operacionesVarSinError.txt** muestra operaciones de las variables sin errores
- ☐ **parametroConError.txt** muestra los errores detectados para los parámetros de las funciones
- ☐ **parametroSinError.txt** muestra funciones con sus parámetros usados correctamente
- ☐ **redeclaracionFuncionesConError.txt** muestra los errore detectados cuando se redeclaran funciones
- ☐ **redeclaracionFuncionesSinError.txt** muestra funciones redeclaradas correctamente
- ☐ **whenConErrores.txt** muestra errores que pueden surgir al no cumplir la condición del when
- ☐ **whenSinError.txt** muestra sentencias when sin errores
- ☐ **while.tx**t muestra las distintas sentencias de while sin errores

Para finalizar, el Árbol Sintáctico es mostrado por consola luego de ejecutar el ".jar" y el código assembler es generado en un archivo ".asm" con el nombre del archivo de la entrada.

Conclusión

Con la confección de esta entrega se pudo comprender como funciona internamente el compilador y lo útiles que son estos en el área de los lenguajes de programación, pudiendo afianzar a fondo la conexión que tiene éste entre las distintas etapas presentadas como lo son el análisis léxico, análisis sintáctico, la generación de código intermedio y finalmente la generación de código assembler, mencionando además la constante utilización de la tabla de símbolos junto con el módulo de errores.
La evolución del trabajo fue haciendo que principalmente la gramática sufra cambios muy grandes, ya que o se evoluciona para agrandar el funcionamiento o simplemente se detectaban acarreo de errores de etapas pasadas junto con pensamientos enfocados erróneamente.
Desde una perspectiva grupal, se aprendió a trabajar con las herramientas brindadas por la cátedra como lo es el YACC o también el MASM32 junto con el debugger antes mencionado OllyDbg, como también a administrar en el proceso un trabajo con el cual se presentaban dudas constantes.
Como autocrítica, se podría haber manejado mucho mejor los tiempos en la primera entrega, ya que por ignorancia se creyó que era correcta, sin embargo le faltaba mucho trabajo para llegar a una entrega en condiciones.