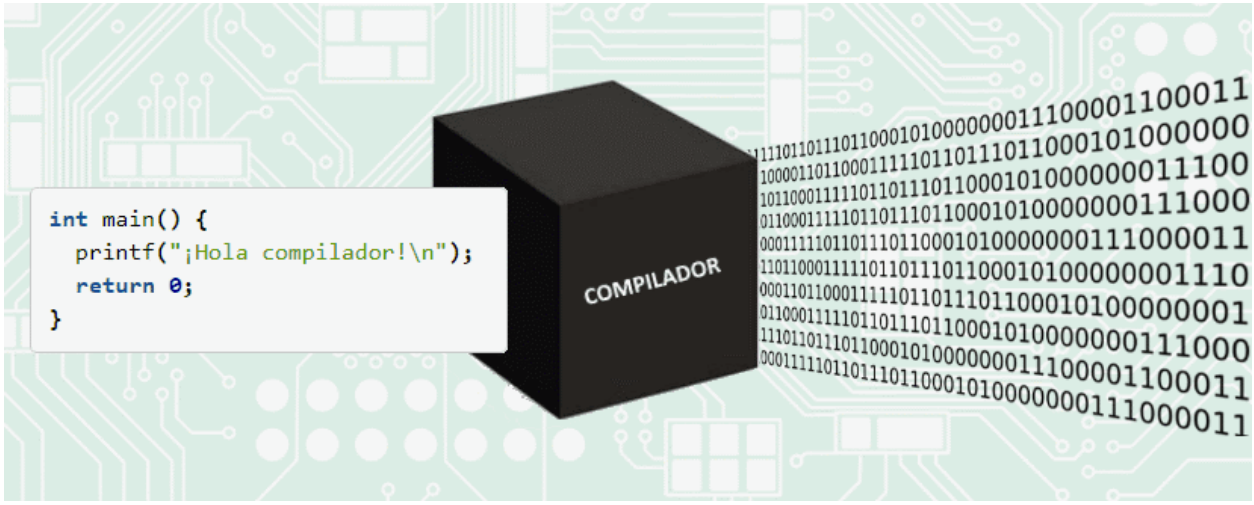
DISEÑO DE COMPILADORES

INFORME TRABAJOS PRÁCTICOS 3 Y 4

GENERACION DE CODIGO INTERMEDIO Y SALIDA



Grupo 16 - Integrantes

Lautaro Lopez - 249714

Franco Pocatino - 249282

Ayudante designada: Antonela Tommasel

Fecha de entrega: 22/11/2022

Introducción

En esta parte del trabajo se van a tratar la generación de código intermedio y chequeos semánticos mediante el tema asignado, el cual es Árbol Sintáctico, y la generación de la salida del compilador, es decir, el código en formato Assembler, para el cual se generará un archivo de tipo ".asm" que representara lo formado a partir del el código intermedio.

Temas Asignados TPs 3 y 4 - *10 15 17 19 21*

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 (<condición>) **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 15: for con continue

for (i =: n; <condición_for>; +/- j) < bloque_de_sentencias_ejecutables > ;

- El bloque se ejecutará hasta que se cumpla la condición.
- En cada ciclo, el incremento (+) o decremento (-) de la variable de control será igual a j.
- Se deberán efectuar los siguientes chequeos de tipo:
  - i debe ser una variable de tipo entero (1-2-3-4-5-6).
  - n y j deben ser constantes de tipo entero (1-2-3-4-5-6).
  - <condición_for> debe ser una comparación de i con m. Por ejemplo: i < m
    Donde m puede ser una variable, constante o expresión aritmética de tipo entero (1-2-3-4-5-6).

  Nota:
  - Para Tercetos y Árbol Sintáctico, los chequeos de tipos se debe efectuar durante la generación de código intermedio
  - Para Polaca Inversa, los chequeos de tipos se deben efectuar en la última etapa (TP 4)
- Si se detecta una sentencia **break**, se debe salir de la iteración en la que se encuentra.
- Si se detecta una sentencia **continue**, se debe forzar el salto al próximo ciclo de la iteración, efectuando la actualización de la variable de control previamente a dicho ciclo.

Tema 17: Break con etiquetado

Al detectar una etiqueta precediendo a la sentencia de control asignada (temas 11 al 16), y un break con una etiqueta, se deberá proceder de la siguiente forma:

- Se debe salir de la iteración etiquetada con la etiqueta indicada en el break.
- Se deberá chequear que exista tal etiqueta, y que la sentencia break 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) {
        while (j < 5) {
            break :outer;  //El break provocará salir de la iteración etiquetada con outer
        };
    };
```

Tema 19: Diferimiento

Al detectarse la presencia de la palabra reservada **defer** al comienzo de una sentencia o bloque de selección o iteración, la ejecución de dicha sentencia o bloque deberá diferirse al final del ámbito donde se encuentra la sentencia diferida.

Tema 21: Conversiones Explícitas

- Todos los grupos que tienen asignado el tema 21, deben reconocer una conversión explícita, indicada mediante la palabra reservada para tal fin.
- Por ejemplo, un grupo que tiene asignada la conversión **tof64**, debe considerar que cuando se indique la conversión **tof64**(expresión), el compilador debe efectuar una conversión del tipo del argumento al tipo indicado por la palabra reservada. (en este caso **f64**). Dado que el otro tipo asignado al grupo es entero (1-2-3-4-5-6), el argumento de una conversión, debe ser de dicho tipo. Entonces, sólo se podrán efectuar operaciones entre dos operandos de distinto tipo, si se convierte el operando de tipo entero (1-2-3-4-5-6) al tipo de punto flotante asignado, mediante la conversión explícita que corresponda. En otro caso, se debe informar error.
- En el caso de asignaciones, si el lado izquierdo es del tipo entero (1-2-3-4-5-6), y la expresión del lado derecho es de tipo diferente, se deberá informar error por incompatibilidad de tipos.

Controles en Tiempo de Ejecución

c) Overflow en sumas de datos de punto flotante

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.

f) Resultados negativos en restas de enteros sin signo:

El código Assembler deberá controlar el resultado de la operación indicada. Este control se aplicará a operaciones entre enteros sin signo. En caso que una resta entre datos de este tipo arroje un resultado negativo, 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.

Cambios etapas anteriores

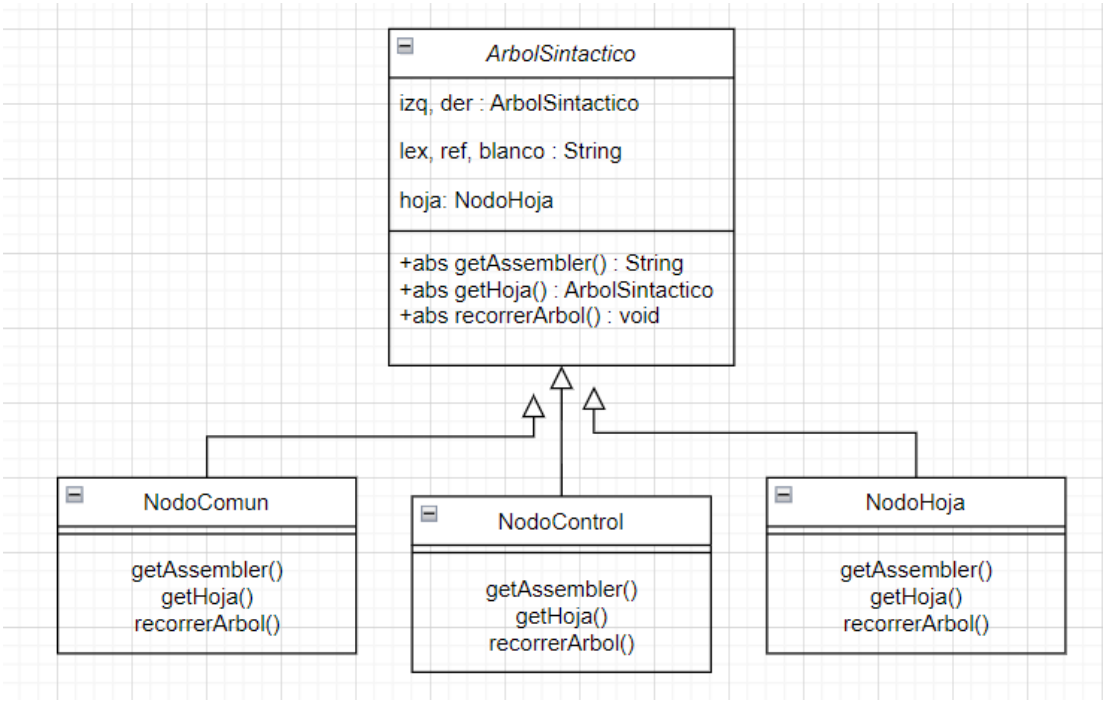
Los principales modificaciones que se introdujeron a el trabajo elaborado en las etapas anteriores, fueron “segmentaciones” dentro del reconocimiento de la gramática, esto surgió tanto como para poder reconocer más errores, como para tomar ciertas acciones semánticas que eran necesarias de implementar para así poder realizar revisiones semánticas correspondientes a la etapa 3 del trabajo práctico, un ejemplo de esto fue lo realizado en la declaración de función. En este caso se declara por separado cabecera, parámetros y retorno. En este caso se realiza antes de que se declare el bloque de función, se realiza la carga en la tabla de símbolos donde se asocia los parámetros de la función con la referencia a esta en la tabla (el alta del token es realizado en “***cabeceraFun***” , por eso solo se solicita la referencia). Pero es necesario hacerlo en esta instancia porque como se ve al final del bloque se le asigna el tipo de retorno asociado a la función. Cosa necesaria de contar al momento de analizar semánticamente el bloque, porque debemos comparar el tipo de retorno declarado con el que debería devolver la función entonces contamos con la tabla de símbolos actualizada para poder hacer la comparación.

```
d_fun :     cabeceraFun parametros_fun retornoFun{
            Token ID = (Token) $1.obj;
            Parametro[] a = (Parametro[]) $2.obj;
            if(a.length>1){
                ts.setParametro1(ID.getRef(), a[0]);
                ts.setParametro2(ID.getRef(), a[1]);
            }
            else if (a.length == 1)
                ts.setParametro1(ID.getRef(), a[0]);
```

```
            if (ts.getUso(((Token) $1.obj).getRef()).equals("Funcion")) // Se
asigna tipo de función

                ts.setTipo(((Token) $1.obj).getRef(), $3.sval);

            } bloq_general_fun {

                        agregarResultado("Declaracion FUNCIÓN. Linea " +
                             lineaUltimoToken);

                         Token ID = (Token) $1.obj;

                      if(!funciones.get(pila.getAmbitoActual()).isRetorno())

                           AgregarErrorSemantico("Función sin sentencia de
                retorno asociada. Linea " + lineaUltimoToken);

                     $$ = new ParserVal(new NodoHoja("Declaración función" +
                             ID.getLexema(), ""));


      funciones.get(pila.getAmbitoActual()).setCuerpoFuncion((ArbolSintactico)
                                                                    $5.obj);

                     funciones.get(pila.getAmbitoActual()).setId(ID.getRef());

                           pila.eliminarUltimoAmbito();}

;
```

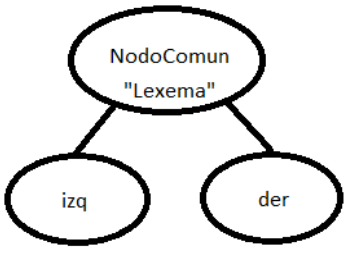
TP3 - Generación de Código Intermedio - Chequeos Semánticos

Para comenzar a plantear la generación de código intermedio debemos basarnos en la estructura de implementación asignada, que en este caso es un árbol binario denominado "**Árbol Sintáctico**", para el cual definimos nodos que pueden ser de diferentes tipos utilizando abstracción.

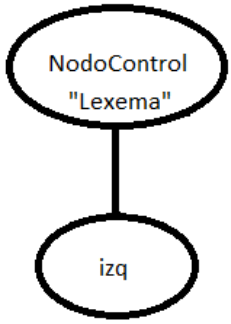


Estos tipos de nodo poseen distinto comportamiento y características,

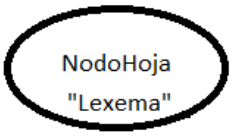
- **Nodo Común:** Este nodo posee dos hijos árbol, izquierda y derecha, respectivamente, y es el nodo más habitual. A la hora de generar el assembler debemos generar el assembler para ambos hijos.



- **Nodo Control:** Este nodo sólo posee un hijo, el izquierdo, y es usado en casos especiales y en términos de distinción y orden, por ejemplo el nodo “Then” y el nodo “Else” son nodos de control, cuyos únicos hijos son el cuerpo asociado a ellos.



- **Nodo Hoja:** Por último está el nodo hoja, el cual no genera assembler y solo lleva el lexema asociado, la referencia a la tabla de símbolos y el tipo.



Todos estos nodos cuentan con una particularidad, y es que poseen un un atributo “**hoja**” de tipo Nodo Hoja, el cual es asignado en tiempo de ejecución, para el cual fueron designados constructores especiales según sea la hoja un nuevo **auxiliar**, una **variable**, o una **constante**. Este atributo es obtenido mediante el método “**getHoja()**”. De todas formas esto es utilizado como recurso para el trabajo práctico 4 y lo veremos más adelante.

Como salida de este apartado del compilador se nos solicitó una forma de visualizar el código intermedio de forma clara, para ello siendo que nuestra estructura es un árbol binario, utilizamos el método abstracto “**recorrerArbol(b)**”, implementado en cada nodo, donde “**b**” es el blanco acumulado que es utilizado para realizar las tabulaciones a la hora de realizar la impresión para separar nodos padre de nodos hijo.

Ejemplo de salida:

```
Sentencia. Tipo : error
  Hijo Izquierdo
  Sentencia out
    Hijo:
    id_a.main. Tipo : f64
  Hijo Derecho
  Sentencia. Tipo : error
    Hijo Izquierdo
    =:. Tipo : f64
      Hijo Izquierdo
      a. Tipo : f64
      Hijo Derecho
      /. Tipo : f64
        Hijo Izquierdo
        12.. Tipo : f64
        Hijo Derecho
        4.. Tipo : f64
    Hijo Derecho
    Sentencia. Tipo : error
      Hijo Izquierdo
      Sentencia out
        Hijo:
        id_a.main. Tipo : f64
      Hijo Derecho
      Sentencia. Tipo : error
        Hijo Izquierdo
        =:. Tipo : f64
          Hijo Izquierdo
          a. Tipo : f64
          Hijo Derecho
          *. Tipo : f64
            Hijo Izquierdo
            12.. Tipo : f64
            Hijo Derecho
            4.. Tipo : f64
        Hijo Derecho
```

Impresión de recorrerArbol(b)

Resultado del árbol.

También es solicitado como parte de la entrega la instancia de la tabla de símbolos y las referencias correspondientes, junto a los errores encontrados, tanto léxicos, sintácticos y semánticos. Cabe destacar que en caso de haber algún tipo de error no se generará código assembler.

```
Tabla de simbolos
Simbolo: 3:      Tipo: ui8 ---- Uso: Constante
Simbolo: 2:      Tipo: ui8 ---- Uso: Constante
Simbolo: id_a.main:      Tipo: ui8 ---- Uso: Variable
```

Tabla de símbolos correspondiente al ejemplo anterior.

Ejemplo de error:

```
main {
    ui8 a;
    a=:2+3.2;
}
```

```
Errores semanticos encontrados ---------------
Tipos no compatibles en asignación . Linea 4
```

Por ejemplo, en este caso estamos intentando realizar una suma de un ui8 con un f64, y guardarlo en una variable de tipo ui8.

Toda esta generación de nodos es realizada desde la **gramatica** (gramatica_final.y) y ejecutada desde el parser, en donde para armar el árbol debemos ir recorriendo los componentes de la gramática, y a medida que son detectados iremos utilizando la notación posicional del Yacc para generar los **ParserVal**, "**$$**" para referirnos a la salida de ese componente, y "**$n**" para referirnos a la posición puntual de ese componente.

Por *ejemplo*, si tenemos el siguiente componente:

```
if :  IF cond THEN bloque_then ELSE bloque_else END_IF
```

$$ = new ParserVal() → if (salida)

$2 → ***cond*** *, $4 →* ***bloque_then*** *, $6 →* ***bloque_else***

IF, THEN, ELSE y END_IF son palabras reservadas.

Cada vez que la notación posicional es utilizada, nos metemos dentro de ese componente para conseguir su valor.

```
NodoComun cuerpo = new NodoComun("CuerpoIF",(ArbolSintactico) $4.obj,
(ArbolSintactico) $6.obj);

$$ = new ParserVal(new NodoComun("IF" ,(ArbolSintactico)
$2.obj,cuerpo));
```

Como vemos en este ejemplo, utilizamos $$ para referirnos a que estamos creando un valor para el "***if***", el cual va a ser un nodo común formado por la condición (***cond***), ubicada en "***$2.obj***", donde ".obj" hace referencia al objeto, y otro nodo del mismo tipo, denominado cuerpo, en donde se situarán como hijos tanto el "***bloque_then***" ($4.obj) y el "***bloque_else***" ($6.obj). Es importante destacar que realizamos un cast a ArbolSintactico en todos los casos.

Y a su vez los no terminales a los que hacemos referencia (cond, bloque_then y bloque_else) cuentan con una definicion para su valor como no terminal en formato de nodos de arbol a recorrer, cosa que nos facilitará el recorrido para generar el assembler y los datos que necesitamos para poder hacerlo efectivamente, por ejemplo, generar la condición antes del IF, y del modo que lo recorremos para generar el assembler en el orden deseado, esta definido dentro de las clases nodo común, nodo control y nodo hoja.

Dentro de este apartado también fueron creados diversos métodos para poder discernir entre la creación de los nodos del árbol, como también estructuras para funciones específicas, entre ellas se destaca la pila de ámbitos (clase "***PilaAmbitos***") para poder reconocer el ámbito en el que nos encontramos en todo momento, así mantener certezas sobre las reglas de alcance para los identificadores definida en el enunciado del trabajo práctico .

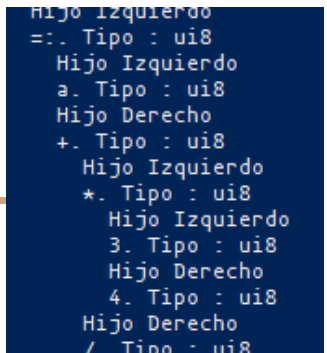
También fue creada la clase "***ElemFuncion***", para diferir entre el contenido de una función, siendo si esta tiene o no sentencias diferidas (explicado mas a adelante), si se refiere al retorno o al cuerpo común y corriente. Es utilizada en el mapa de funciones, como valor correspondiente a la clave string ámbito.

`funciones = HashMap<String, ElemFuncion>().` Este string es el ámbito obtenido en la "PilaAmbitos", y el cuerpo con sus correspondientes sentencias diferidas son cargadas cuando termina este ámbito. Se crearon también estructuras para facilitar información como puede ser el Parámetro, o las listas auxiliares que se usa tanto para cuando se utiliza una declaración de variables, del estilo ***"ui8 a,b,c,d,e;"*** donde primero se dan de alta individualmente los identificadores como variable y luego se les setea el tipo (que en la gramática se obtiene despues), o cuando invocamos una función, la lista de parámetros que estamos utilizando a modo de comparación con los parámetros de la función invocada.

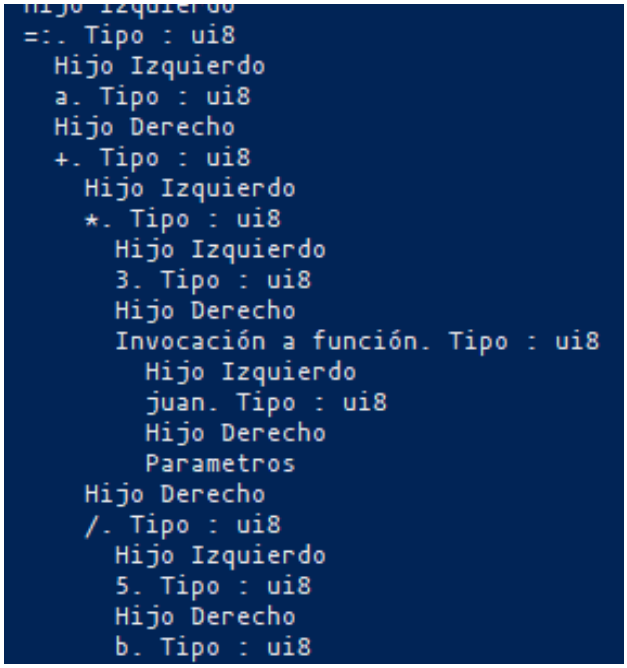
Nuevos errores considerados

En esta etapa los errores que fueron considerados fueron todos los relacionados con la existencia de variables, funciones, constantes y tags. Estos fueron resueltos mediante la pila de ámbitos ya explicada anteriormente donde podremos tanto ver si existe una variable dentro del ámbito actual y los anidados y además al momento de crear un identificador nuevo ver si este existe en el ámbito actual.

Se implementaron también todas las revisiones de tipo. La forma en la que se logró esto es en el momento que se van creando los nodo



hoja estos llevan consigo información sobre su tipo entonces al unir dos nodos en un solo nodo común (Los utilizados para las operaciones) se va comparando el tipo de sus hojas al momento de creación entonces si son el mismo, se le asigna ese tipo al nodo común, si son distintos, se marca el nodo como con error. Entonces de esta manera si a los nodos que hacen las operaciones más abarcativas como una asignación o un casteo, si nos llega nodo con tipo "error" sabemos que dentro de la expresión se encuentran tipos distintos. Un ejemplo de código con los tipos adjuntos a los nodos se muestra en la figura. Y esto sucede independientemente de si son constantes o son variables o invocaciones a función . Como se muestra en la figura, estos datos son obtenidos de la tabla de símbolos cuando corresponde.

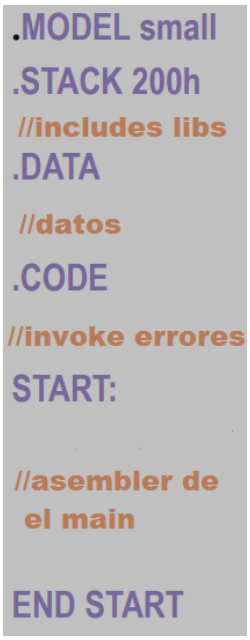


Por otro lado se reconocieron errores que involucren tipos erróneos al momento de declarar por ejemplo una sentencia for, una condición dentro de los when, en los tipos de retorno de acuerdo a lo que devuelve la función, etc. En las funciones también se reconocen los tipos y cantidad de parámetros necesarios para hacer las correspondientes invocaciones a ellas.

TP4 - Generación de la salida (Código Assembler)

Una vez generado el código intermedio para el compilador, debemos tomar este último como entrada para la siguiente etapa, la generación de código Assembler. Esto implica que vamos a ir recorriendo el árbol formado en la etapa anterior, dentro de lo que son las sentencias en general en post-orden hacia la derecha generando código de salida dependiendo del nodo en el que nos encontremos y del lexema asociado al mismo. De todas maneras este recorrido de post-orden puede variar según las necesidades de la estructura que estamos generando, como sucede por ejemplo en el arbol generado para las expresiones aritmeticas donde es necesario un nivel de precedencia entre los operandos de las operaciones, (no es lo mismo 7/4 que 4/7) pero de todas maneras para la estructura general se trata de respetar eso. Pero antes debemos *estructurar* el formato de

salida, y es por ello que como primer paso de este trabajo creamos la clase "***GeneradorAssembler***" que va a ser la encargada de llevarlo a cabo, añadiendo las **librerías** utilizadas para windows brindadas por el "*masm32*", generando los **datos** a partir de la tabla de símbolos y por último el **código** a partir de los nodos. La estructura mencionada es la siguiente:



Esta clase a partir del parser que recibe como parámetro y las librerías descargadas, le dará valor a las variables de tipo String:

- "***librerías***" → Compuesta por el set de instrucciones, el modelo de memoria utilizado en windows, el tamaño de la pila y las librerías del masm32.
- "***data***" → Es donde se declaran las variables y datos provenientes de la tabla de símbolos, junto a la declaración de los mensajes de error.
- "***code***" → Zona de código, compuesta por las etiquetas a errores y comienzo del main (start).

Estos componentes son alterados por los métodos **genData()** y **genCode()**. Para después generar la salida en el método **getSalida()**.

Una vez ya definida la estructura de la implementación debemos comenzar a generar el código Assembler, este mismo es obtenido mediante el método abstracto "***getAssembler()***" ubicado en Árbol Sintáctico, y es implementado según el tipo de nodo a tratar. Un dato no menos importante es que este método, como la estructura en sí, hace uso del **polimorfismo**, es decir, en nodos de control como en nodos comunes podemos tener el mismo lexema con distinto comportamiento, esto se verá reflejado en ciertos casos como el de la asignación de etiquetas que se verá más adelante.

Como primera instancia es llamado con el nodo principal, el nodo del programa o main, y ya desde ahí irá recorriendo el árbol y generando salida para todo el mismo.

Este método consta de tomar como entrada el lexema del nodo en el que estamos parados y realizar un *switch-case* con el mismo para saber qué hacer en cada caso posible, y concatenarle a la salida la generación de sus hijos. Por *ejemplo*, este es el caso del cast a f64 que es uno de los temas que tenemos asignados:

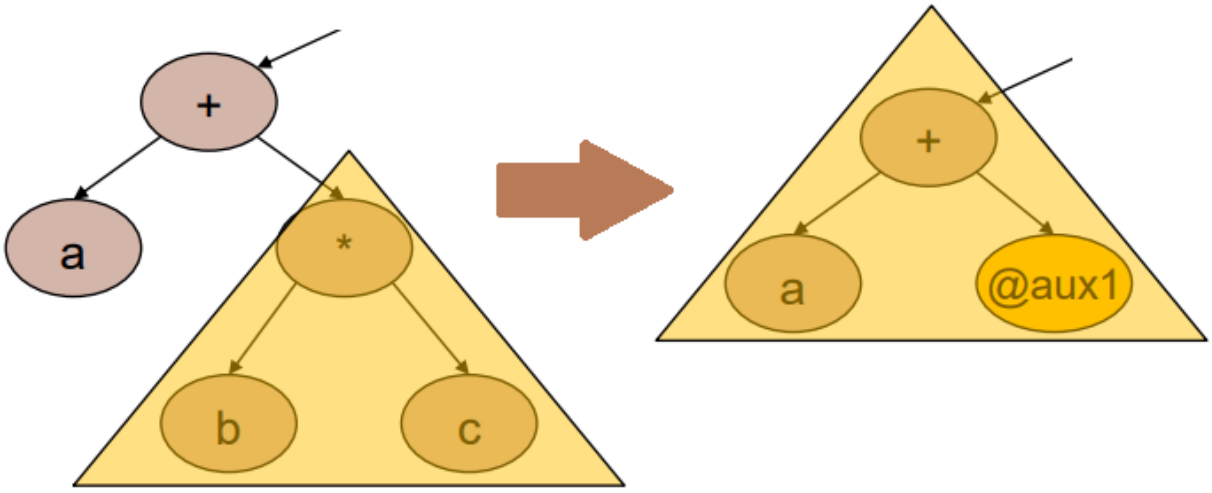
```
case "Casteo TOF64": {

   salida += getIzq().getAssembler();

   salida += "MOV BL, " + getIzq().getHoja().getRefString() + "\n"

             + "MOV BH, 0" + "\n"

             + "MOV @auxConversion, BX"

             + "FILD " + "@auxConversion" + "\n";

             + "FST " + nuevoAux + "\n";

    break;

           }
```

Este es un nodo de control, por ejemplo, por lo que solo tiene un hijo y es el izquierdo.

Entonces cuando detecta que el lexema de entrada es "*Casteo TOF64*", que fue cargado en el árbol con ese valor, sigue avanzando por el hijo izquierdo y genera el assembler asociado al mismo y luego genera el asociado al casteo.

Aritmética

Para el cálculo de **operaciones aritméticas** funciona de la misma manera, solo que estas están dadas por un nodo común, con 2 hijos, y debemos de asignarle el resultado a una nueva variable **auxiliar** que estará situada como un nuevo nodo **hoja**.



Para ello vamos a utilizar de ejemplo a la suma,

```
case "+": {
   salida += this.getIzq().getAssembler() + this.getDer().getAssembler();
   cantAux++;
   nuevoAux = "@aux" + cantAux;
   sub1 = getIzq().getHoja();
   sub2 = getDer().getHoja();
   hoja = new NodoHoja(nuevoAux, sub1.getTipo(), nuevoAux,true);
```

```
if (sub1.getTipo().equals("ui8")) {
    salida += "MOV AL, " + sub1.getRefString() + "\n"
            + "ADD AL, " + sub2.getRefString() + "\n"
            + "MOV " + nuevoAux + ", AL" + "\n";
                } else {
           salida += "FLD " + sub1.getRefString() + "\n"
            + "FADD " + sub2.getRefString() + "\n"
            + "JO errorOverflow" + "\n"
            + "FST " + nuevoAux + "\n";
                }
                break; }
```

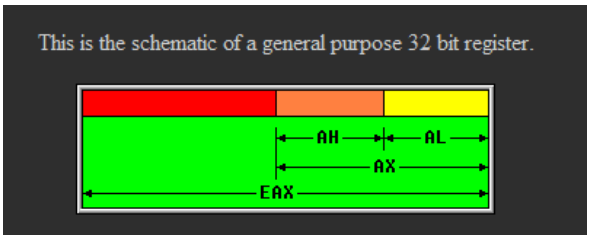
Tenemos una variable estática "***cantAux***" ubicada en el padre, la cual es un entero inicializado en cero que utilizaremos cada vez que definimos un auxiliar, de esta manera siempre usamos un auxiliar distinto.

Obtenemos ambas hojas de los hijos de la suma utilizando el método getHoja(), y generamos el nuevo auxiliar, con el tipo de la suma (es importante destacar que se utiliza el del hijo izquierdo porque va a ser el mismo que el del derecho, y esto es **chequeado previamente** a generar Assembler). Ahora dependiendo del tipo relacionado a la operación generamos assembler para una suma de enteros cortos sin signo de 8 bits (**ui8**) o flotantes signados de 64 bits (**f64**), en caso de los últimos utilizamos el el coprocesador matemático 80X87 para operaciones de punto flotante, que aumenta el juego de instrucciones del 80X86 mejorando su capacidad de tratamiento de números.

Si es de tipo flotante debemos tener en cuenta uno de los controles en tiempo de ejecución que se nos fue asignado, el cual es si el resultado excede del rango (overflow), para en este caso emitir un mensaje de error y terminar.

Aritmética en ui8

Para el resto de operaciones el procedimiento es similar, exceptuando la ***multiplicación*** *y la* ***división***. En el caso de la multiplicación, el resultado que vamos a obtener será de 16 bits y quedará guardado en el registro AX (de 16 bits) para mantener la coherencia de los registros de 8 bits, se toma la parte menos significativa que se encontrará guardada en el registro AL. Respetando el formato del gráfico adjunto. Algo similar ocurre en la división, donde se debe respetar que el dividendo se debe guardar en AX previo a efectuar la operación y en el div se indicará el registro que actuará como divisor en la cuenta, y el resultado que nos interesa, el cociente, se guardará dentro de AL.



Aritmética en f64

En el caso de las operaciones de punto flotante, como se mencionó anteriormente, utiliza el **coprocesador 80X87**, en el cual no utilizamos registros convencionales si no que usamos instrucciones de transferencia de datos para trabajar con los 8 registros de este procesador que están organizados en una pila. Por ejemplo, una suma en f64:

FLD src → Introduce una copia de mem en el tope de la pila (*ST(0)*).

FADD src → Realiza la suma entre el tope de pila y memoria y lo guarda en el tope.

FST dest → Copia el tope de la pila a memoria sin afectar el puntero de pila.

Etiquetas

La única manera de respetar el anidamiento de condiciones e iteraciones en assembler es mediante el uso de pilas, en el caso de medir a qué dirección saltar, es decir, a qué etiqueta, funciona de la misma manera. Generamos dos pilas para cumplir con todos los temas que se nos asignaron para poder mantener el orden de iteración dentro de la generación de la salida, "***etiquetas***" y "***sentenciasFor***". La pila de **etiquetas** es modificada tanto por el ***if*** como por el ***for,*** ya que ambos tienen una condición por la que preguntar, entonces en base al resultado del método "***comparación(salto)***", el cual es llamado cada vez que el lexema es un comparador (<,>,>=,<=,=,*!*=), y se debe hacer un salto a la etiqueta ubicada en el tope de la pila, generada en el cuerpo. Por otro lado la pila de **sentenciasFor** guarda otro tipo de información necesaria para poder generar las etiquetas y las correspondientes instrucciones de continue , y break con tag,

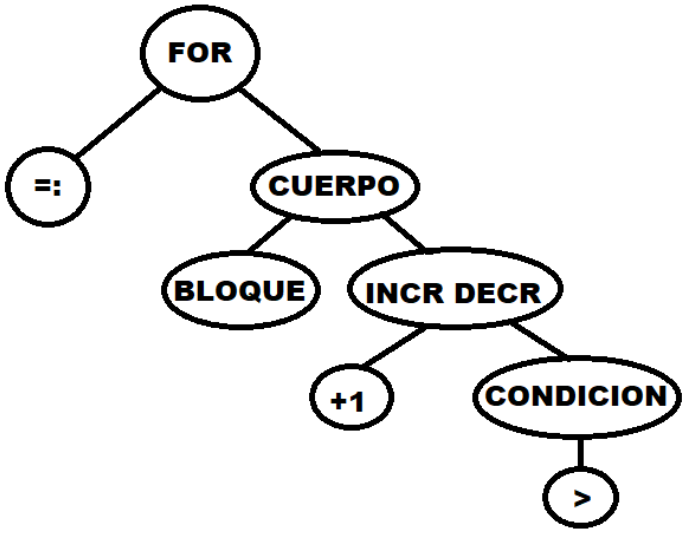
Etiquetas if

En el caso de las sentencias *if*, tenemos la certeza de que posee un bloque de sentencias acordes al *then* como también existe la posibilidad de que esté acompañado o no de un bloque de sentencias asociados al *else,* para ello sabemos que contamos con una estructura que hace uso del polimorfismo como mencionamos anteriormente, por lo que un nodo *if* puede ser tanto un nodo de control como un nodo común. En el caso de que sea un nodo de control solo recorremos el nodo hijo izquierdo y quitamos la última etiqueta agregada al final y la concatenamos a la salida junto al ":", mientras que si es un nodo común recorremos ambos hijos, primero el izquierdo (*then*) y quitamos la última agregada (etiqueta del *else*) y generamos la etiqueta correspondiente a la salida, es decir, el fin del bloque y preparamos la etiqueta del else con su correspondiente ":" para luego debajo generar el assembler del bloque else.

Etiquetas for

Para entender correctamente el planteo de las etiquetas del for, es necesario remarcar cómo cambiamos la **estructura** tradicional del for por una que podemos usar a nuestra conveniencia para la generación de código assembler (ver figura), en un orden que nos permita ir dejando las etiquetas a nuestra elección y simplificando los saltos, además de tener la posibilidad de resolver los breaks y continues. La estructura ***sentenciaFor*** cumple

la función de para la *etiquetaN* generar los labels necesarios para poder utilizar las sentencias **break continue** y **break** con **tag**. Esto es debido a que se les pasa un entero, que hicimos que coincida con el número de etiqueta utilizado para comparar condiciones. Entonces si la condición del for utiliza la etiqueta 5 entonces los apartados serán ***increment5 condition5*** y ***next5***. Se utilizó el formato de pila para poder tener en cuenta los break y continue individuales entonces cuando se detecta uno sabemos que se refiere al for que se encuentra en el tope de la pila ***SentenciaFor***. También lleva información adicional como la referencia del registro que usa la variable del for, esto es debido a que el campo incremento y decremento es del estilo "***+1"***, por lo que no encontramos ningún identificador asociado para poder sumarle o restarle a la variable i que utilice ese for. Y por donde se genera en la figura el segmento ***incr_decr*** no tenemos forma de guardarlo y esta estructura nos facilita esa información. Por otro lado, el label de condición es necesario para hacer el salto una vez realizada la asignación porque puede que no ingrese al for por la condición. Y finalmente esta pila también nos brinda información para cuando tenemos que saltar a determinado tag, si esa estructura tiene tag y podemos consultar su label de next para el correcto funcionamiento del break con tag.



Pseudocódigo Assembler

código **asignación**

salto incondicional a ***condition***

etiqueta1:

  código **cuerpo**

increment:

  código **incremento/decremento**

condition:

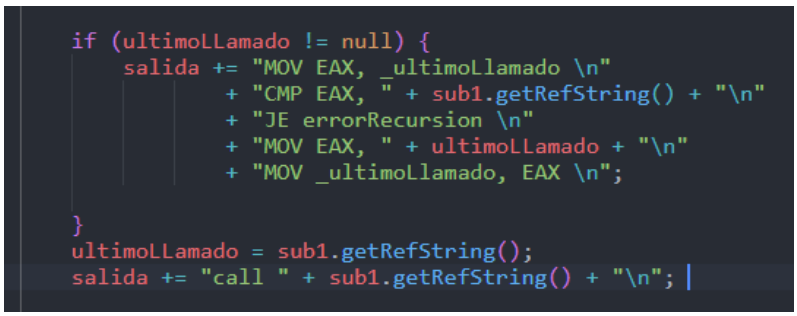
  salto a ***etiqueta1*** si la condición se cumple

next:

Errores detectados en ejecución

Recursión mutua

En este caso la forma de detectar estos casos lo que se hizo fue como en el primer llamado sabemos que se hace desde el main donde no se puede hacer otro call, lo que se hace es en el primer call es guardar en el string ***ultimoLlamado*** a dónde vamos a saltar, luego cuando hay otra invocación se comparan las direcciones del registro últimoLlamado con la de call entonces mientras estas no sean iguales se van "pisando" entre sí en ese registro y en el momento que el call resulte igual con la dirección a la que se quiere saltar se hará el Jump equals.



```
if (ultimoLLamado != null) {
    salida += "MOV EAX, _ultimoLlamado \n"
            + "CMP EAX, " + sub1.getRefString() + "\n"
            + "JE errorRecursion \n"
            + "MOV EAX, " + ultimoLLamado + "\n"
            + "MOV _ultimoLlamado, EAX \n";

}
ultimoLLamado = sub1.getRefString();
salida += "call " + sub1.getRefString() + "\n";
```

Overflow en sumas float

La forma de detectar una suma fuera de rango fue mediante la instrucción "***JO errorOverflow***" la cual realiza un salto hacia la etiqueta errorOverflow que define el mensaje de error y termina la ejecución si hubo desbordamiento.

Resultado negativo en resta entera

Similar al anterior, este chequeo se realiza con la instrucción "***JS errorResta***", que salta a la etiqueta de error si el signo del resultado es negativo, invoca el mensaje de error y finaliza la ejecución.

Temas especiales

Tema 10 - Constantes y Sentencia When

En este caso era necesario resolver un nuevo tipo de sentencia y declaración, que son la sentencia **when** y las **constantes** respectivamente. Un requisito indispensable para poder hacer este tipo de funcionalidad es que la generación del código dentro del bloque "***When***" debía ser resuelto en tiempo de compilación, es decir, previo a generar el código assembler ya se debía tener la condición resuelta y no generar las instrucciones ni cargar los elementos en la tabla de símbolos en caso de que esta condición no se cumpliese. Primero por el lado de las constantes con nombre se utilizó el mismo procedimiento que para todo identificador, es decir en el ámbito que se encuentra no puede haber un identificador que use su mismo nombre y luego semánticamente se le guardó un valor asociado para facilidades también a la hora de generar código assembler y resolver el when. Lo más complejo sin duda fue resolver la condición del When antes de la generación de código. Entonces para empezar se restringió el tipo que se podía manejar constantes enteras, esto implica tanto con y sin nombre. Entonces si *"a"* es una constante se pueden procesar comparaciones del tipo *"(a < 3)"* .Por lo que lo único que se realiza desde la gramática es obtener las referencias a esos tokens a y 3.

```
}

 private int convertirValor(String ref) { // Se trata
                                          // una refere
                                          // que se pid
  int resultado = -1;
  if(erroresSemanticos.isEmpty()){ // Si hay error sem
  try {
    resultado = Integer.parseInt(ref);
  } catch (Exception e) {
    resultado = Integer.parseInt(ts.getValor(ref));

  }
  }
  return resultado;
}
```

Luego tendremos dos casos donde su valor sea directamente el lexema, el 3 en la tabla de símbolos se representa con 3, y el otro caso donde tengamos que solicitar el valor asociado a “a”. Sintetizado en el código de la figura.

Y luego eso dependiendo del lexema reconocido en la comparación (“<”,” > “, “<=”. etc) se hace la comparación en sí de ambos números, luego de esto lo que sucede es que esta comparación devuelve un valor booleano, y dependiendo de este al no terminal cond_when se le asigna un árbol sintáctico con la comparación que fue hecha (simplemente a modo ilustrativo ya que la condición fue resuelta). Entonces en el when se pregunta si él no termina ***cond_when*** tiene cuerpo o no. Y así ya sabemos si la condición fue correcta o no. En caso de que sea correcta se crea el nodo del arbol adjuntando el cuerpo.

Un detalle surgido es que las modificaciones en la tabla de símbolos, altas de variables funciones, tags, etc, no tenían forma de saber si están en un contexto de when correcto o erróneo y si deberían quedarse en la tabla o no. Por lo que para resolver esto se implementó una pila denominada ***“identificadoresWhen”*** la cual guarda una lista de strings, que representan a la referencias a los identificadores. Entonces lo que sucede es que cuando comienza un bloque de when se habilita un boolean que dice si está dentro de un when o no, entonces cada vez que se modifique la tabla de símbolos y el contexto es dentro de un when se agrega el identificador al tope de la pila. Entonces cada uno de los identificadores que se agregan en el when están guardados en el tope y en caso de que la condición resulte falsa solo hay que iterar en toda esa lista de agregados durante el when del tope disminuyendo su referencia en la tabla de símbolos. La estructura es una pila debido a que si se anidan varios when distintos estos tendrán distintos identificadores entonces si en el when de más adentro se da true, solo hay que agregar la lista de referencias a la lista del when de afuera debido a que si la condición de afuera da false se deben de eliminar de cualquier manera. Y la flag dentroWhen solo se baja cuando la pila de identificadores se vacía.

Tema 17 - Break con tag

El break con etiquetado fue resuelto por dos partes, primero en la parte semántica sólo se efectúan dos revisiones que son que el tag exista en todo el ámbito, que es resuelto por una búsqueda desapilado cada uno de los ámbitos buscando un identificador que tenga

ese nombre, y el segundo es que el tag que estamos efectuando el break se encuentra anidado en las sentencias de iteración que estamos generando. El tag se guarda de manera simple y se comparan lexemas porque si existe (revisado previamente) no es necesario buscarlo por identificador. Entonces cuando hay un for con tag solo se agrega al pila de ***etiquetasAnidadas***.

Luego al momento de generar el assembler para esto se hace uso de la estructura ***sentenciaFor*** previamente explicada, que también guarda el tag del for que está generando, entonces como en assembler solo tenemos que hacer JMP a la etiqueta de next asociada al for que tiene el tag, por lo que iteramos a través de toda la pila de ***sentenciasFor*** utilizada al momento de generar assembler buscando cual es el label de next de la sentencia que tiene el tag utilizado por el break (previamente revisado que existe ese tag y se encuentra anidado por él).

Tema 19 - Sentencias Diferidas

Este requerimiento fue resuelto de manera simple haciendo uso del mapa de ***<String,ElemFuncion>*** donde el string es el ámbito donde se encuentra la sentencia diferida y el ElemFuncion como comentamos antes tiene un nodo. Entonces la manera de resolver esto fue en el árbol original de ejecución cuando se encuentra una sentencia diferida lo que se coloca en el árbol de la función es un NodoHoja que solo indica que hay una sentencia diferida. Y esta sentencia en sí es agregada al ArbolSintactico ***sentenciasDiferidas*** del elemento función del ámbito actual, el cual es obtenido gracias a la pila de ámbitos. La manera de que estas sentencias en sí queden al final es que al cerrar cada ámbito, tanto el programa como las funciones, se ejecuta el método ***setCuerpoFunción()*** que representa la raíz, entonces cuando se dispara este método significa que se cerró el ámbito y en ***cuerpoFuncion*** tenemos todo el árbol original de código intermedio, por lo que lo único que se hace en el ElemFuncion es preguntar si hay sentenciasDiferidas, si las hay se insertan a la izquierda, por lo que debido al orden elegido para recorrer los nodos y generar su assembler las haría quedar en el último lugar de dicho ámbito ya que se recorre en post orden a derecha. Por lo que en assembler no fue necesario agregar ningún tipo de complejidad solo organizándolo en el código intermedio es suficiente.

Conclusión

Como cierre a este trabajo podemos decir que logramos comprender con mayor profundidad todas las etapas por las que pasa un compilador desde que le ingresa una entrada en formato de código basado en un lenguaje específico, hasta que se genera la salida en formato Assembler. Todos los pasos desde el comienzo cuando identifica los tokens, como a medida que avanza de fase y va actualizando la tabla de símbolos y las constantes evoluciones de la misma, el uso de la herramienta Yacc, la generación de código intermedio a partir de una gramática y una estructura específica y la creación de la salida en código máquina, fueron profundizados en base a los temas que nos tocaron y eso implica que un compilador es incluso mucho más grande de lo que creemos y nuestro diseño es solo una leve representación de uno real.