DISEÑO DE COMPILADORES I
TRABAJO PRÁCTICO NRO. 3 Y 4

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

BARREIRO SANDOVAL, CAMILA
camilarbs29@gmail.com

GUERRA, MARTÍN
martinalejandroguerra12@gmail.com

LACOSTE, YAGO
yagolacos@gmail.com

Grupo 8
Temas asignados:
4 6 10 14 19 20
Árbol Sintáctico
Controles en tiempo de ejecución - a d

Ayudante: **GONZÁLEZ, MAILÉN**

RESUMEN
Habiendo realizado las modificaciones pertinentes de la primera entrega, correspondiente al Análisis Léxico y Análisis Sintáctico, en los apartados del presente trabajo se recorre el proceso de desarrollo de las fases de Análisis Semántico, Generación de Código Intermedio y Generación de Código Assembler. Se incluyen consideraciones de implementación, pruebas y resultados obtenidos.

1. INTRODUCCIÓN

El presente informe corresponde a la Segunda Entrega del Trabajo Práctico de Cursada, de la materia Diseño de Compiladores, de la Facultad de Ciencias Exactas, Universidad del Centro de la Provincia de Buenos Aires (UNICEN).
En estas etapas de desarrollo, nos centramos en obtener una salida en código assembler, a partir de la generación de código intermedio. Así, en conjunto con la primera entrega, se completan las fases de compilación faltantes, como se muestra en la siguiente imagen:

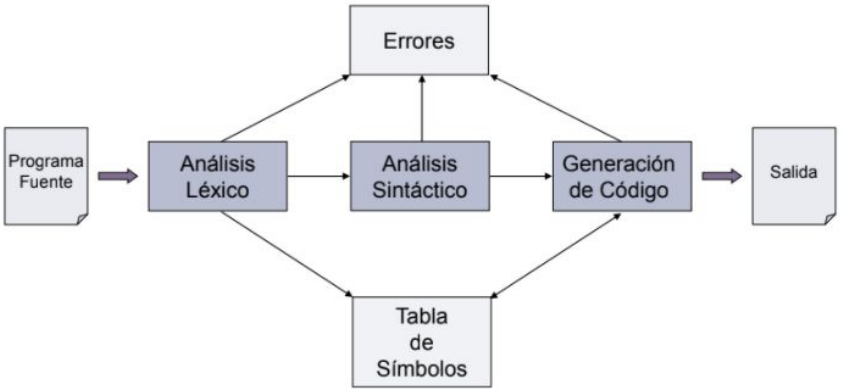


Figura 1 - Fases de la compilación

La estructura utilizada para la generación de código intermedio fue el Árbol Sintáctico, la cual resulta menos restrictiva a comparación de los métodos Polaca Inversa y Tercetos. Por otra parte, a la hora de generar código assembler, incorporamos el chequeo de errores en tiempo de ejecución, tanto para cuando se produce una división por cero, como también si se generan resultados negativos en las restas.
El informe se divide en dos partes, primero se evalúa la Generación de Código Intermedio y Análisis Semántico, luego la Generación de Código Assembler.
Cabe aclarar que continuamos utilizando el lenguaje de programación Java, para realizar el desarrollo de las etapas que comprende esta entrega.

2. ACLARACIONES PERTINENTES

A partir de las correcciones realizadas sobre la entrega de las primeras dos fases de compilación, se modificaron los aspectos que se detallan a continuación:

- Tabla de símbolos: desde un principio fue construida por medio de una estructura *HashTable*, pudiendo acceder a ella con un String, la key, que se corresponde a los lexemas identificados. Actualmente, el value fue modificado para contener, en vez de un id asociado al lexema, una lista de *Attribute*. Este listado contiene las distintas apariciones del lexema en el código de entrada, siempre que estas se encuentren en distintos ámbitos, o bien los usos difieran. Para ejemplificar, si consideramos la siguiente porción de código:

```
DOUBLE a;
DOUBLE b;
PROC a (DOUBLE a) NA = 2_ul{
        a = 3.0;
};
a(b);
```

para el lexema *a*, el listado tendrá una longitud igual a cuatro: la variable definida en el ámbito del main, el nombre del procedimiento, el parámetro del procedimiento y el llamado al procedimiento. Aunque el lexema *a* se encuentra cinco veces en el código, la asignación dentro del procedimiento hace referencia al parámetro, por eso no se almacena nuevamente en la tabla de símbolos.

- *Attribute*: clase que contiene los Strings *lexeme*, *scope* e *id*; los atributos *type* (corresponde al tipo del identificador) y *use* (uso que se le da al identificador en el programa); un booleano *declared* (indica si el identificador se encuentra declarado); y un listado de parámetros (para asociar con el nombre de los procedimientos definidos, sus parámetros formales).
- scope: ámbito del identificador. Se utilizó el literal @ para separar los distintos ámbitos a los que pertenece un identificador. Existe un ámbito principal que es el main, por lo tanto, todos los identificadores poseen como scope, al comienzo, *ID@Main*. Luego serán los procedimientos los que definan nuevos ámbitos, que se concatenarán a los scope de los identificadores, según corresponda.
- *Type*: clase que define los tipos aceptados por el compilador DOUBLE y ULONGINT, como también ERROR, el cual resulta de utilidad a la hora de chequear la compatibilidad de tipos, que se explicará en las próximas secciones.
- *Use*: enumerado que contiene los usos reconocibles por el compilador.
- DOUBLE: Como se aprecia en la Figura 2, se incorporaron nuevos estados al autómata, para reconocer de forma correcta los DOUBLE. Dicha modificación impactó de forma directa en la clase *StateMatrix*, inicializando los nuevos estados con sus respectivas transiciones y acciones semánticas.

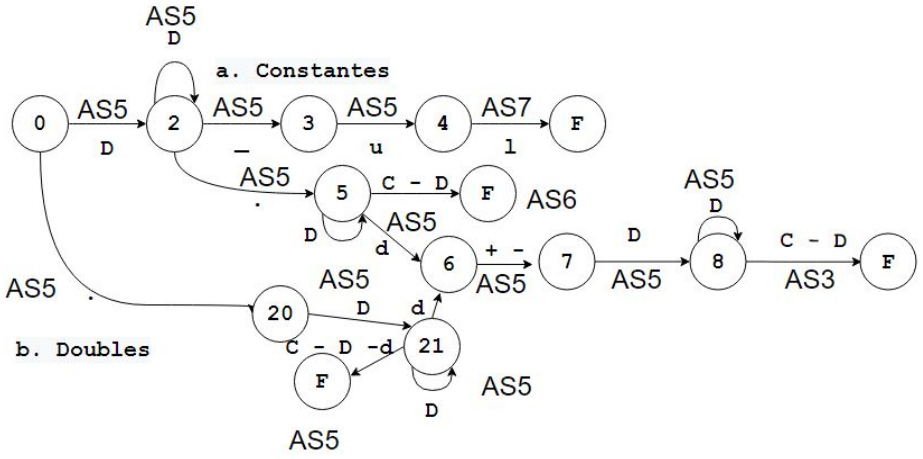


Figura 2 - Autómata Finito - Corrección reconocimiento DOUBLES

- Conversiones explícitas: en la gramática, la regla *factor* ahora define las conversiones explícitas, así pueden incorporarse en todo lugar donde se permiten expresiones.
- Sentencias de control FOR:
    - Cuando el cuerpo del FOR se encuentra formado por una sola sentencia, la estructura no contiene llaves, el punto y coma final es el de la sentencia declarada. Mientras que, si se define un bloque de sentencias, se incorporan las llaves, colocando un punto y coma en la llave de cierre
    - Los ejemplos proporcionados contemplan que siempre se compare del lado izquierdo la misma variable que se inicializó al comienzo de la sentencia. Caso contrario, el compilador no contempla ese error. Será sobre esta variable que se realice también el incremento o decremento.

```
ULONGINT i, a, b;
FOR(i = 2_ul; i < b; DOWN 1_ul)
        a = b;

FOR(i = 2_ul; i < b; DOWN 1_ul){
        a = b;
        a = b;
};
```

- Sentencias IF: se producen errores relativos a la condición, que no habían sido considerados en la primera entrega.

Por otra parte, resulta importante aclarar que para el desarrollo de la presente entrega, se asumieron los siguientes aspectos:

- Orden de sentencias declarativas y ejecutables: la primera regla de la gramática, *programa*, se define como *lista_sentencias_declarativas lista_sentencias_ejecutables*. Esta regla impone una limitación sobre el código fuente, ya que primero se deberán escribir todas las sentencias declarativas, y luego las ejecutables. Se realizó esta modificación para poder armar correctamente la representación intermedia, la cual debe contener sólo las sentencias ejecutables. Los casos de prueba que se adjuntan respetan este orden.
- Invocaciones y definiciones de procedimientos: cuando se llama a un procedimiento, los parámetros no pueden tener el mismo nombre que el invocador. Entendemos que se debería permitir, ya que procedimientos y variables pueden tener el mismo nombre, pero es una limitación propia del desarrollo, que no supimos cómo solucionar.
- En caso de existir errores léxicos, los errores sintácticos y semánticos no se visualizan en la salida, porque consideramos que los errores de las siguientes etapas se encuentran atados a los de la primera, por lo tanto serán incorrectos.
- Existen errores que el Analizador Léxico no reconoce, por ejemplo, que una palabra reservada se escriba en minúscula, en ese caso, se asume que se trata de un identificador. Cuando esto ocurre, la gramática accede por la regla que más se parece a lo que está definido en el código, dando errores que no se condicen con las sentencias de entrada.

3. GENERACIÓN DE CÓDIGO INTERMEDIO

En el proceso de traducir un programa fuente a código destino, un compilador puede construir una o más representaciones intermedias, las cuales pueden tener variedad de formas. Una de ellas es el árbol sintáctico, la cual nos fue asignada por la cátedra para el desarrollo del presente trabajo. Se trata de una representación comprimida del árbol de parsing, flexible, ya que en sus nodos se puede almacenar cualquier tipo de información que resulte relevante para la etapa de Generación de Código Assembler, por ejemplo, la entrada de un identificador o constante en la Tabla de Símbolos.

3.1. Decisiones de Diseño e Implementación

Para la implementación del Árbol Sintáctico, creamos una clase abstracta llamada *SyntacticTree*, que contiene un nodo izquierdo, un nodo derecho y un atributo. Los nodos son también de tipo *SyntacticTree*, mientras que el atributo es del tipo *Attribute*, que se explicó en el apartado 2. Existe una clase por cada componente de las sentencias ejecutables, las cuales extienden a *SyntacticTree* e implementan los métodos *generateAssemblerCodeRegister* y *generateAssemblerCodeVariable* que permite generar el código assembler asociado, dependiendo el tipo de los operandos. Existen nodos intermedios utilizados principalmente para concatenar sentencias y generar bifurcaciones. Estos resultan de utilidad para no perder información durante el reconocimiento de reglas, y especificar saltos en el código assembler, respectivamente.

Para ejemplificar lo explicado anteriormente, tomaremos el caso de la sentencia IF. A medida que el parser recorre la gramática, ejecuta la porción de código relativo a cada regla. Entre estas líneas, se encuentra la creación de los nodos del árbol, sean hojas cuando se trata de símbolos terminales, o bien intermedios para los no terminales. Para el caso particular que estamos analizando, es preciso reconocer claramente los bloques relativos a la comparación, el bloque de sentencias si la misma se cumple (THEN), y el salto si esta no se cumple (ELSE). Es por esto que las clases *SyntacticTreeIF*, *SyntacticTreeIFBODY*, *SyntacticTreeCMP*, *SyntacticTreeIFELSE*, *SyntacticTreeIFTHEN*, nos permiten reconocer qué lugar de la sentencia se está analizando. Luego, para poder generar el Assembler correspondiente, dicho árbol se recorrerá, y sobre cada nodo padre de dos hijos hoja, se ejecutará el método mencionado *generateAssemblerCode*.

3.2 Notación posicional

La herramienta BYACC/J, utilizada para generar el analizador gramatical LALR, cuando compila correctamente, crea dos clases *Parser* y otra llamada *ParserVal*. La primera contiene, por un lado, todo lo escrito por nosotros en la gramática (reglas, código asociado a ellas, etc), y por otro, lo necesario para que el parser funcione de la forma deseada. *ParserVal*, nos permite almacenar el valor semántico para el parser. Particularmente, a dicha clase añadimos cuatro atributos que son accedidos en la gramática por medio de la notación posicional, para cumplir con el fin que se describe a continuación:

- tree: su tipo es *SyntacticTree*. Nos permite crear la representación intermedia, el árbol sintáctico, a medida que se recorren las reglas de la gramática. Por lo general, es accedido con la notación $$.tree, y se le asigna un new de las clases que extienden a *SyntacticTree*, o bien se pasa la referencia del árbol que se encuentra creado hasta ese momento, por ejemplo $$.tree = $1.tree.
- attributes: lista de *Attribute*. Utilizada para acceder a los atributos que se corresponden a una entrada de la tabla de símbolos. La notación utilizada para su acceso depende de la posición, pero un ejemplo podría ser $1.attributes.get(0).getLexeme(), por medio del cual obtenemos el lexema del primer elemento de la lista, para el identificador que se encuentra en la primera posición de la regla.
- attributesSetteable: lista de *String*. Almacena los lexemas de los identificadores que se reconocen en las sentencias declarativas, para luego poder setear su uso y tipo. También se utiliza con el mismo fin para los parámetros formales. El acceso posicional es muy similar a lo explicado en el item anterior, difiere el uso que le damos a lo que devuelve dicho acceso.
- type: su tipo es Type. Tanto para las conversiones explícitas, los parámetros formales, como para las sentencias declarativas, almacena el tipo siendo así accesible desde otras partes de la

gramática, y no solo en la regla donde se lo define. Se setea en la regla *tipo*, por medio del acceso $$.type, y luego se utiliza posicionalmente en los casos antes mencionados.

3.3 Bifurcaciones en sentencias de control

Las bifurcaciones en sentencias de control son importantes, para poder determinar qué acción es necesaria tomar cuando se cumple o no la condición que hace que dicha sentencia de control cicle, en el caso del FOR, o bien realice ciertas acciones u otras, en el caso del IF.
El árbol sintáctico como representación intermedia, y gracias a la forma en que implementamos las distintas clases explicadas en la sección 3.1, no necesita algoritmos especiales para identificar dichas bifurcaciones. Las reglas gramaticales se encuentran seccionadas de tal manera que, cuando se encuentran los bloques de sentencias que precisan ser bifurcados, se crea un nodo particular que representa la bifurcación, conteniendo para la izquierda un grupo de sentencias, y para la derecha, el otro grupo. Así, cuando se recorra el árbol para la generación de assembler, estos nodos especiales nos permitirán escribir el código asociado a las bifurcaciones, ya sean saltos o etiquetas, según resulte necesario.
No consideramos necesario volcar un pseudocódigo, ya que no existe un algoritmo especial para realizar las bifurcaciones.

3.4 Procedimientos

Los procedimientos, al igual que el main, son módulos distintos dentro del código fuente. Particularmente para los procedimientos, necesitamos resolver qué hacer cuando encontramos sus declaraciones; las mismas se consideran código muerto hasta que el procedimiento es invocado, ya que si esto no sucede, nunca se ejecutan sus sentencias. Como este trabajo tiene como fin aplicar conceptos teóricos, y no ofrece mejoras, ni presta atención a la eficiencia de la implementación, optamos por la solución que se describe a continuación. La declaración de cada procedimiento define su propio árbol, y cada uno de estos árboles será volcado al código assembler, sea o no invocado. Para almacenar cada árbol, decidimos llevar una lista de punteros que referencie a cada sub árbol creado por cada uno de estos procedimiento, es decir que cada vez que se declara un procedimiento, en la lista se almacena la raíz de su respectivo árbol (que contiene el mismo atributo que el ID del procedimiento), a la cual se le asignará el subárbol creado por las sentencias ejecutables definidas dentro. Esto fue posible gracias a la utilización del puntero *tree*, de tipo *SyntacticTree* (explicado en la sección 3.2), donde concatenamos todos los nodos correspondientes a cada sentencia ejecutable. Además, cuando en la gramática se encuentra un procedimiento nuevo, se incrementa un contador interno, para no perder referencia de la posición de la lista en la que tenemos que concatenar el árbol, y en caso que las sentencias ejecutables se "interrumpan" por la definición de un procedimiento anidado, supiéramos donde continuarlo. En el caso de los procedimientos hermanos, utilizamos una lista auxiliar que cargamos de la misma forma que se explicó anteriormente, y luego se añadía a la lista principal (para no pisar los procedimientos ya cargados).

3.5 Errores considerados

Para esta etapa, los errores considerados se desprenden de la consigna del trabajo práctico otorgado por la cátedra. A continuación se los detalla y explica mediante un ejemplo:

- Variables / procedimientos no declarados: Como el concepto de declaración y alcance van de la mano, por medio del método *isReachable*, implementado en la clase *SemanticAnalyzer*, identificamos si una variable o procedimiento se encuentra o no al alcance, donde se quiere usar. Para esto, comparamos los scopes de los atributos almacenados en la tabla de símbolos para el lexema en cuestión, con el scope de la variable o procedimiento que queremos saber si está declarado y al alcance. Ya que cada ámbito alcanzable se encuentra separado por el literal @ dentro del scope, el método itera sobre este último mencionado, desconcatenando los ámbitos y realizando la búsqueda en la tabla de símbolos. En caso que no se encuentre, se emite un error que informa que no existe variable o procedimiento declarado al alcance.

```
PROC z (DOUBLE g) NA = 5_ul {
        PROC c (DOUBLE m) NA = 0_ul {
                DOUBLE a;
                a = 2.0; PERMITIDO, EXISTE VARIABLE a DECLARADA EN MISMO ÁMBITO
        };
        PROC d () NA = 0_ul {
                a = 2.0; NO SE ENCUENTRA DECLARACIÓN DE VARIABLE AL ALCANCE
        };
};

DOUBLE a;
PROC c (DOUBLE b) NA = 0_ul {
        a = 2.0; PERMITIDO, EXISTE VARIABLE a DECLARADA AL ALCANCE
};

z(a); PERMITIDO, EXISTE PROCEDIMIENTO z DECLARADO AL ALCANCE
d(); NO SE ENCUENTRA DECLARACIÓN DE PROCEDIMIENTO AL ALCANCE
```

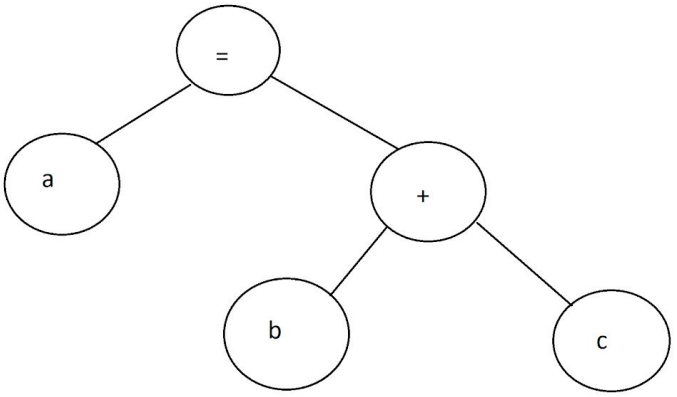
- Variables / procedimientos redeclarados: Por medio del método *isRedeclared*, declarado en *SemanticAnalyzer*, identificamos si una variable o procedimiento poseían el mismo nombre, estando en el mismo ámbito, lo cual genera un error de redeclaración de procedimiento o variable, dependiendo el caso de prueba. Para esto, listamos las variables que contenían el mismo nombre que la que se estaba declarando, y por medio de un String *globalScope*, chequeamos el ámbito de la variable a definir. Así pudimos determinar si en esta lista se encontraba alguna variable o procedimiento con el mismo nombre y scope. Si no se encontraba una se marcaba como declarada y se dejaba en la lista, sino se elimina la última agregada (ya que el léxico a medida que reconoce tokens los incorpora a la tabla de símbolos, sin importar estos errores semánticos), y mostrabamos el error.

```
DOUBLE a; PERMITIDO, NO EXISTE VARIABLE a EN MISMO ÁMBITO
PROC z (DOUBLE g) NA = 5_ul {
        PROC c (DOUBLE a) NA = 2_ul { PERMITIDO, NO EXISTE VARIABLE a EN MISMO ÁMBITO
                DOUBLE a; REDEFINICIÓN DE VARIABLE (PARÁMETRO DEFINE MISMA VARIABLE)
        };
        PROC c (DOUBLE b) NA = 3_ul { REDEFINICIÓN DE PROCEDIMIENTO
                DOUBLE c;
        };
        PROC d (DOUBLE h) NA = 0_ul {
                DOUBLE a; PERMITIDO, NO EXISTE VARIABLE a EN MISMO ÁMBITO
                DOUBLE a; REDEFINICIÓN DE VARIABLE
        };
};

PROC c (DOUBLE b) NA = 0_ul { PERMITIDO, NO EXISTE PROCEDIMIENTO c EN MISMO ÁMBITO
        DOUBLE a; PERMITIDO, NO EXISTE VARIABLE a EN MISMO ÁMBITO
};
```

- Operaciones entre operandos del mismo tipo: Para este caso lo que implementamos fue un método que a medida que reconocemos las reglas factor, término o expresión, y en estas no se realice una conversión explícita, tomamos el nodo creado del árbol en esa regla y chequeamos que sus dos

hijos sean del mismo tipo. Si estos son del mismo tipo, al nodo unión se le setea el tipo del lado derecho para que se pueda seguir chequeando en los nodos de más arriba. Por ejemplo, si para la asignación a=b+c, se crea un árbol donde en el nodo suma se verifica que tanto b como c tengan el mismo tipo. Si no son compatibles, en el nodo suma se setea el tipo error, y así todos sus padres también tendrán este tipo. Este chequeo de tipos se resuelve en el código intermedio, es decir, en el árbol.

```
ULONGINT a;
ULONGINT b;
ULONGINT c;

a = b + c;
```



- Control de niveles de anidamiento permitidos: los procedimientos definen un número de niveles de anidamiento permitidos, es decir que existe una cantidad específica de procedimientos que pueden estar anidados uno dentro de otro, y la misma no puede superarse. Para este chequeo implementamos distintos métodos que serán explicados por medio de un ejemplo:

```
PROC a () NA = 3_ul {
    PROC b () NA = 2_ul {
        PROC c () NA = 5_ul {
                DOUBLE a;
            };
        };
    };
```

  - La clase *SemanticAnalyzer* posee una lista de Integer, llamada *NA*. Cada entero almacenado corresponde a los niveles de anidamiento permitidos, y cada posición de la misma se relaciona de forma directa con el procedimiento en análisis. Para el ejemplo, NA = {3, 2, 5}
  - Cada vez que la gramática reconoce la regla *asignacion_NA*, es decir que se lee *NA=NRO_ULONGINT*, se ejecutan los siguientes métodos implementados dentro de la clase *SemanticAnalyzer*:
    - *checkNA*: recibe por parámetro el scope construido hasta el momento. Realiza un split del scope, para saber la cantidad de ámbitos recorridos (sin tener en cuenta el ámbito actual). Itera sobre la lista *NA*, y consulta si el contenido de la lista en la posición es menor que la longitud del scope. En caso afirmativo, retorna la posición, sino, continua el recorrido. Retornar la posición significa que ese procedimiento está superando el nivel de anidamiento permitido. Si la lista se recorre completa, y nunca se cumplió la condición, devuelve -1, es decir, que ningún procedimiento excede los niveles de anidamiento permitidos.
    - *addNA*: agrega a la lista *NA* el nivel de anidamiento que posee el procedimiento que se está analizando en ese momento

      Para el ejemplo dado, en un principio la lista *NA* está vacía, por lo que *checkNA* devolverá -1, mientras que *addNA* hará que NA = {3}. Cuando se analiza PROC b, el scope tiene una longitud de 2, pero como no se tiene en cuenta el ámbito actual, la

comparación será 3<1; como es falso retorna -1 y se añade NA = {3, 2}. Por último, para PROC c, el scope tiene una longitud de 2 (restando el ámbito actual), la comparación será 3<2, para la primera iteración, y 2<2, para la segunda iteración. Como ambos casos son falsos, el método retorna -1. Se añade NA = {3, 2, 5}. Como ningún procedimiento supera el número de anidamiento permitido, no se generan errores.

- Cuando se termina de reconocer un procedimiento, se utiliza el método *deleteNA* que elimina de la lista el último Integer que corresponde al nivel de anidamiento del procedimiento que se terminó de leer.

A continuación se presenta un ejemplo con errores, y se procede a realizar el mismo seguimiento:

```
PROC a () NA = 1_ul { SUPERA NIVEL DE ANIDAMIENTO
    PROC b () NA = 2_ul {
        PROC c () NA = 5_ul {
            DOUBLE a;
        };
    };
};
```

Para el primer caso, aplica el mismo seguimiento que el ejemplo sin errores, mientras que addNA hará NA = {1}. Cuando se analiza PROC b, el scope tiene una longitud de 2, pero como no se tiene en cuenta el ámbito actual, la comparación será 1<1; como es falso retorna -1 y se añade NA = {1, 2}. Por último, para PROC c, el scope tiene una longitud de 2 (restando el ámbito actual), la comparación será 1<2, para la primera iteración, y al no cumplirse la condición, el método retorna 0 (posición que genera el error). Se añade el error correspondiente y NA = {1, 2, 5}. Como no existen más procedimientos anidados, se comienzan a eliminar los elementos de la lista. Hay que tener presente que el error es siempre por el procedimiento padre, osea el que resulta más restrictivo en número de anidamientos permitidos, mirando hacia arriba en el anidamiento.

- Ámbito prefijado al usar una variable (ID::ID): Considerando que se pueden utilizar variables de otros ámbitos, chequeamos que el procedimiento (ID izquierdo) se encuentre declarado en algún ámbito alcanzable, buscando en la tabla de símbolo el nombre del procedimiento, y recorriendo los scopes de la lista de atributos asociada. En caso de no encontrarlo, se imprime un error por procedimiento no declarado. Si se encuentra el procedimiento al alcance, se procede a la búsqueda de la variable (ID derecho) en el ámbito del procedimiento en cuestión. Para esto creamos el método *checkIDdospuntosID* que busca la variable y devuelve el tipo para ser utilizado para el chequeo de tipos. Si no encuentra dicha variable se emite un mensaje de error y se retorna el tipo error. Para el siguiente ejemplo:

```
PROC b (DOUBLE hola) NA = 2_ul {
    DOUBLE hola;
    PROC a () NA = 3_ul {
        DOUBLE hola;
        b::hola = 2.3;
    };
};
```

buscamos que *b* este declarada como procedimiento al alcance. Como lo está, buscamos que la variable *hola* esté declarada en el ámbito del procedimiento *b*, lo cual resulta verdadero, entonces puede asignarse 2.3 a *hola* porque también coinciden los tipos. Acá se ve claramente, que ID::ID

fuerza la utilización de la variable *hola* del procedimiento *b* y no la variable declarada en el procedimiento *a*.

3.6. Casos de prueba

Se adjunta en el comprimido del trabajo, un documento *.txt* que contiene los casos de prueba considerados, que sirve como entrada del compilador.

4. GENERACIÓN DE CÓDIGO ASSEMBLER

Una vez obtenida la representación intermedia buscada, el árbol sintáctico, arribamos a la última fase de la compilación, la Generación de Código Assembler, por medio de la cual se obtiene una salida en lenguaje ensamblador. Valiéndonos de las ventajas de dicha representación, y por medio de la implementación de distintos métodos que serán descritos en las próximas secciones, el conjunto de instrucciones obtenidas fueron volcadas en un archivo *.asm*. Este último podrá utilizarse como entrada en un programa ensamblador (como MASM32, ofrecido como referencia por la cátedra), que producirá como salida un ejecutable que se corresponde al código fuente original (entrada del Análisis Léxico, primera fase del Compilador).

4.1 Proceso de Generación de Código Assembler

El árbol sintáctico resulta una estructura flexible a la hora de ser utilizado como representación intermedia. Como ya se explicó en la sección 3.1, se implementaron distintas clases que extienden de *SyntacticTree*, para que cada una fuese responsable de crear su propio código assembler. Así como en el Análisis Léxico cada acción semántica era ejecutada según la combinación de fila/columna, de la matriz de transición de estados, para la generación de assembler, en el recorrido del árbol cada nodo conoce cómo crear su propio assembler, ejecutando el método correspondiente.

Resulta importante resaltar que se utilizaron dos técnicas distintas para construir las sentencias assembler, lo cual derivó en dos métodos, *generateAssemblerCodeRegister* y *generateAssemblerCodeVariable*, que son implementados por cada nodo del árbol. Por un lado, para las operaciones entre datos de tipo ULONGINT, se trabajó con los registros del procesador, mediante la técnica de seguimiento de registros. Por otro, para las operaciones entre datos de tipo DOUBLE, se utilizó el co-procesador 80X87, y el mecanismo para generar código fue el de variables auxiliares.

Los métodos *generateAssemblerCodeRegister* y *generateAssemblerCodeVariable*, devuelven un String que permite concatenar las instrucciones generadas en cada nodo, para luego poder almacenarlas en el archivo de salida. No creemos necesario ahondar demasiado en el funcionamiento de los métodos, ya que el fin de cada uno es el mismo, solo varía el conjunto de instrucciones que concatena cada uno. Lo que nos parece interesante aclarar, es que existen nodos intermedios que no generan código assembler, ya que el objetivo de su uso es encadenar bloques de sentencias, por ende, los métodos antes mencionados no poseen utilidad alguna, aunque se los ejecute durante el recorrido. Un ejemplo de esto es la clase *SyntacticTreeBODY*, utilizada en el cuerpo del procedimiento. Por otra parte, existen otros nodos que realizan las mismas acciones, sin importar la técnica utilizada, por lo que para esas clases sólo se implementa y ejecuta el método *generateAssemblerCodeRegister*, siendo *SyntacticTreeFOR* un claro ejemplo de ello.

El recorrido del árbol se realizó con el método *getMostLeftTree*, mediante el cual, a partir de la raíz del árbol, buscamos el subárbol más izquierdo con hijos hoja, generamos el assembler correspondiente(con los métodos antes explicados), y reemplazamos dicho subárbol por una variable auxiliar o registro, dependiendo de la técnica.

4.2 Operaciones aritméticas

Las operaciones aritméticas consideradas fueron la suma, resta, multiplicación y división, teniendo las dos primeras mayor precedencia sobre las últimas.

	ULONGINT	DOUBLE
Suma	ADD	FADD
Resta	SUB	FSUB
Multiplicación	MUL	FMUL
División	DIV	FDIV

Para el caso de la suma y la multiplicación, por la propiedad de conmutatividad no hubo que preocuparse por el problema del registro del lado izquierdo o derecho respecto del otro dato a operar. En el caso que fuesen dos variables, una variable y una constante, una variable y un registro o una constante y un registro, se colocó el resultado de la operación sobre el registro encontrado; en caso que no haya ningún registro en utilización, se creó uno en el cual se guardó el valor del operando del lado izquierdo y se almacena el resultado sobre este mismo.
Para el caso de la división y la resta, se debió analizar de qué lado se encontraba el registro, ya que estas operaciones carecen de la propiedad "Conmutatividad". Por esta razón, en el momento que se detectó que del lado izquierdo se encontraba una variable o constante en lugar de un registro, se tuvo que colocar su valor en un registro y realizar la operación entre ambos registros.

4.3 Etiquetas y sentencias de control

Dado que se explicó en el práctico anterior, en los procedimientos lo que realizamos fue que al recorrer en la gramática cargamos una lista de *SyntacticTree* donde cada puntero contiene anidado las sentencias ejecutables de cada procedimiento. Al momento de realizar el código assembler se decidió crear etiquetas en donde se ponía como nombre el ámbito del procedimiento para que se diferencie de otros (se decidió realizarlo de esta manera ya que el nombre más el ámbito serán únicos), seguido del código assembler de sus sentencias ejecutables.
Para las sentencias de control decidimos realizarlas de la siguiente forma:
En el caso de la sentencia de control IF, como se explicó en la sección anterior, al crear un árbol para este en la clase *SyntacticTreeIF* declaramos una pila llamada JLABEL de string estática para que todas las clases anidadas puedan acceder a ella e ir apilando las etiquetas correspondientes para que se realicen los saltos necesarios. En el caso de que se llegue al nodo donde se realiza la comparación, lo que se hizo fue apilar la etiqueta de posible salto ya que se desconoce la dirección a la que se debería saltar, y se almacena en la pila JLABEL del *SyntacticTreeIF*. Al recorrer el nodo que se encuentra a la izquierda del nodo cuerpo se ejecutan las sentencias ejecutables que en ella se encuentran. Al final de esto, se agrega la condición de salto de que se terminó ese bloque con sus sentencias; esta etiqueta se apiló pero antes se desapiló la agregada anteriormente. Cuando se recorre el nodo que contiene el "else" se ejecutan las sentencias que este posee y al final se agrega la etiqueta que se almaceno en el recorrido anterior para saber a donde tiene que saltar el primer bloque. Como último paso, se agrega la etiqueta de donde debería saltar en caso que termine el nodo "else".
En el caso que sea una sentencia FOR lo que hicimos fue definir también un árbol donde tenemos la cabecera del FOR que engancha la asignación y del otro lado un nodo FOR. Para este nodo asignación lo que hicimos fue asignar a la variable el valor inicial para que se ejecute el FOR. Luego de generar su assembler de asignación a una variable colocamos la primera etiqueta, esto lo que va hacer es que cuando termine de realizar toda la comparación y ejecución de sentencias pueda volver al inicio. Esta etiqueta la almacenamos también en una pila de labels para ir apilando y desapilando las próximas. A su vez, también nos quedamos con el atributo de la variable que le asignamos el valor inicial para poderlo incrementar o decrementar luego. Una vez pasada la etapa inicial ingresamos al nodo de comparación para que pueda devolver el código assembler y agregarle una condición de salto como se realizar en la sentencia "if" almacenando la etiqueta de salto con una nueva etiqueta para que salte en caso de que la condición no se

cumpla.Después de generar el cuerpo de la sentencia for con sus respectivas sentencias ejecutables solo queda enganchado el nodo de incremento o decremento donde devuelve su assembler asignado un incremento o decremento al ID almacenado como estático para que lo modifique. Por último se agrega la etiqueta de salto para que vuelva al inicio si se cumple seguida abajo de la etiqueta en caso que no se cumpla desde el inicio. Esta representación quedó muy similar a la propuesta por la cátedra cuando hay una sentencia de while.

4.4 Errores considerados

En la construcción del código assembler, fue necesario tener en cuenta errores que deben ser chequeados en tiempo de ejecución. Como consigna general, el código generado por el compilador debe detectar los errores correspondientes, emitir un mensaje de error y finalizar la ejecución. A continuación se detallan los errores considerados:

- División por cero: El código Assembler chequea que el divisor sea diferente de cero antes de efectuar una división, tanto para datos de tipo DOUBLE como para ULONGINT. En *.data* del código assembler se generan siempre dos espacios en memoria para almacenar el valor cero de ambos tipos, es decir

  _ceroDOUBLE DQ 0.0 _ceroULONGINT DD 0

  Luego, en el método *generateAssemblerCodeRegister* se realiza una comparación entre el hijo derecho del subárbol más izquierdo, es decir el divisor, y la variable _ceroULONGINT, por medio de la instrucción CMP. Después escribimos JE Error_Division_Cero, que nos va a permitir saltar en caso que la comparación sea verdadera, es decir, que se esté queriendo dividir por cero. Cuando se ejecuta el método *generateAssemblerCodeVariable*, se pone en el tope de la pila el divisor, con la instrucción FLD, y la comparación se realiza con la instrucción FCOMP entre _ceroDOUBLE y el tope de la pila. Igual que el caso anterior, por último se escribe JE Error_Division_Cero. El salto por esa etiqueta invoca MessageBox para mostrar el mensaje de error, y ExitProcess para finalizar la ejecución.
- Resta negativa: Para enteros sin signo, el código Assembler chequea que la resta entre datos de este tipo no arroje un resultado negativo, ya que, de ser así deberá emitir un mensaje de error y terminar. El procedimiento es muy similar al explicado anteriormente, la diferencia radica en que para el método *generateAssemblerCodeRegister*, la comparación se realiza con la instrucción CMP, entre el operando izquierdo y el derecho, y la instrucción de salto utilizada es JB Error_Resta_Negativa. Dicho salto invoca MessageBox para mostrar el mensaje de error, y ExitProcess para finalizar la ejecución.

4.5. Casos de prueba

Se adjunta en el comprimido del trabajo, un documento *.txt* que contiene los casos de prueba considerados, que sirve como entrada del compilador.

5. CONCLUSIONES

Con la culminación del desarrollo del presente trabajo, logramos tomar conciencia de la cantidad de detalles que se tienen en cuenta a la hora de compilar y ejecutar un código fuente. Los IDEs de programación logran invisibilizar y realizar estas tareas de forma tan eficiente, que muchas veces automatizamos apretar un botón para correr nuestros programas, olvidando lo que eso implica.

Consideramos que la implementación de las últimas fases de la compilación, y la posterior ejecución del código assembler, nos permitió terminar de entender algunos conceptos claves de la programación, tales como el ámbito o el alcance de las variables, que no nos habían quedado claros en el transcurso de la carrera.

En cuanto a aspectos técnicos del desarrollo, nos encontramos con la limitación de no lograr armar un diseño del proyecto con visión a futuro, lo que se tradujo en la reimplementación reiterada de estructuras como el árbol sintáctico, hasta que derivamos en la última idea que es la que se presenta. A su vez, consideramos que la asignación de temas puntuales nos facilitó algunos aspectos del desarrollo, ya que, como se mencionó en otras secciones, la flexibilidad del árbol, por ejemplo, nos dio libertades a la hora de su implementación. Somos conscientes que existen repeticiones de código, sobre todo en la gramática, que quizás eran evitables, pero no supimos abstraer.

Considerando que el objetivo de la materia no se centra en desarrollos que optimicen la eficiencia, sino que busca plasmar los conceptos teóricos por medio de una práctica concreta, creemos que los objetivos propuestos fueron alcanzados.