

Diseño de Compiladores 1

Trabajos Prácticos 3 y 4



Autores:

Roman Luna

romanluna1234@gmail.com

Savo Gabriel Koprivica

skoprivica@alumnos.exa.unicen.edu.ar

Grupo:

20

Cátedra:

Diseño de compiladores 1

Profesores:

Jose A. Fernandez Leon

Introducción	2
Generación de Código Intermedio	2
Árbol Sintáctico	2
Notación posicional de Yacc	8
Consideración y decisiones tomadas con respecto a los errores	8
Comprobaciones Semánticas	9
Salida Generacion deCodigo Intermedio	9
La salida para el código intermedio es El árbol sintáctico y una lista de errores semánticos.	9
Generación de la Salida	12
Generacion de Codigo Assembler	12
Operaciones Aritméticas	12
Generación de las etiquetas destino	12
Modificaciones a las etapas anteriores	12
Implementación de los temas particulares	12
Conclusiones	12

Introducción

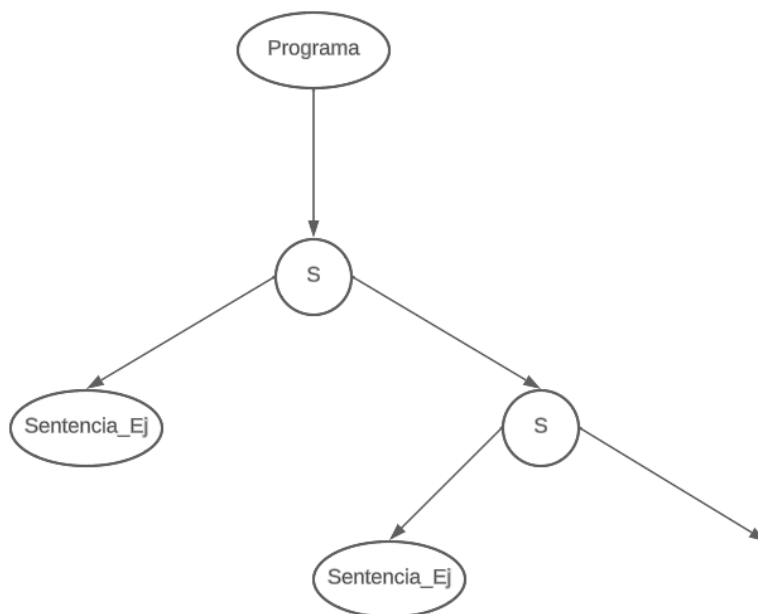
El presente informe exhibe la continuación del desarrollo de la creación de un compilador basado en el lenguaje Java. En este informe se verá la creación de la Generación de código intermedio (Árbol Sintactico y Comprobaciones sintácticas) y Generación de salida (Assembler y Salida de Assebre). Las decisiones tomadas y cómo resolvimos diversos problemas o cuestiones que nos fueron surgiendo.

Generación de Código Intermedio

Árbol Sintáctico

El Árbol Sintactico es una estructura de datos que representa la estructura jerárquica y la semántica de un programa después de haber sido analizado sintácticamente. Es una representación intermedia que se utiliza comúnmente en la fase de análisis semántico de un compilador. Se compone por: nodos y hojas, donde los nodos del árbol representan operaciones o construcciones, mientras que las hojas representan operaciones o valores. Cada nodo puede tener cero o más nodos hijos, dependiendo de la construcción gramatical que representa.

Estructura general del Árbol



La forma en la que se genera el árbol es a partir de un nodo Control llamado “programa”, se le crea un hijo llamado sentencia. La manera en la que se crea el árbol es:

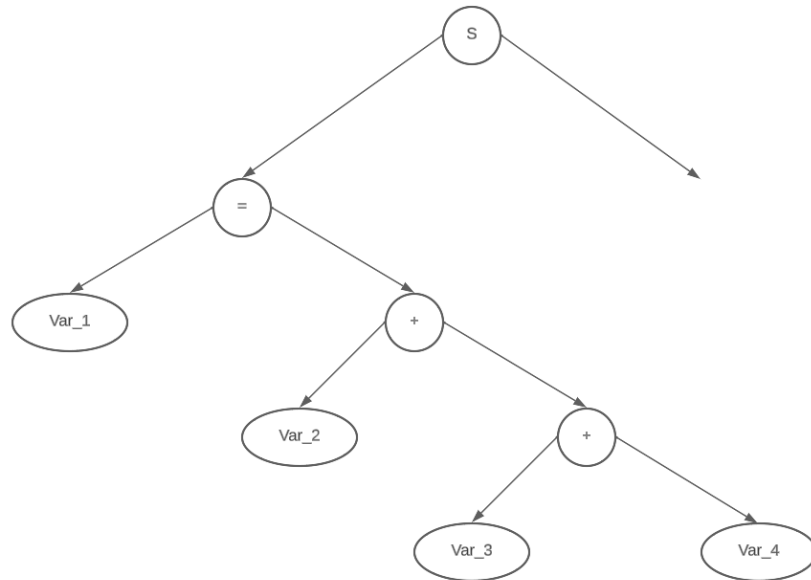
1º) Género un nodo Sentencia

2º) En su hijo izquierdo agregó una sentencia ejecutable, ya sea una asignación, llamado Función, etc.

3°) Al generar una nueva rama que corresponde a una sentencia ejecutable, si el hijo izquierdo del nodo “sentencia” es null, lo agrego en ese lugar. Sino Repito desde el paso uno.

El hijo derecho siempre va a corresponder a un nodo “sentencia”, y el hijo izquierdo a una subrama generada por una sentencia ejecutable.

Generación de una rama asignación:



Una asignación se genera de las hojas a la raíz, que en este caso es “=”.

1°) se crean los nodos hojas correspondientes a var3 y var4

2°) se crea un nodo padre “+” y se le agregan los dos nodos hoja creados

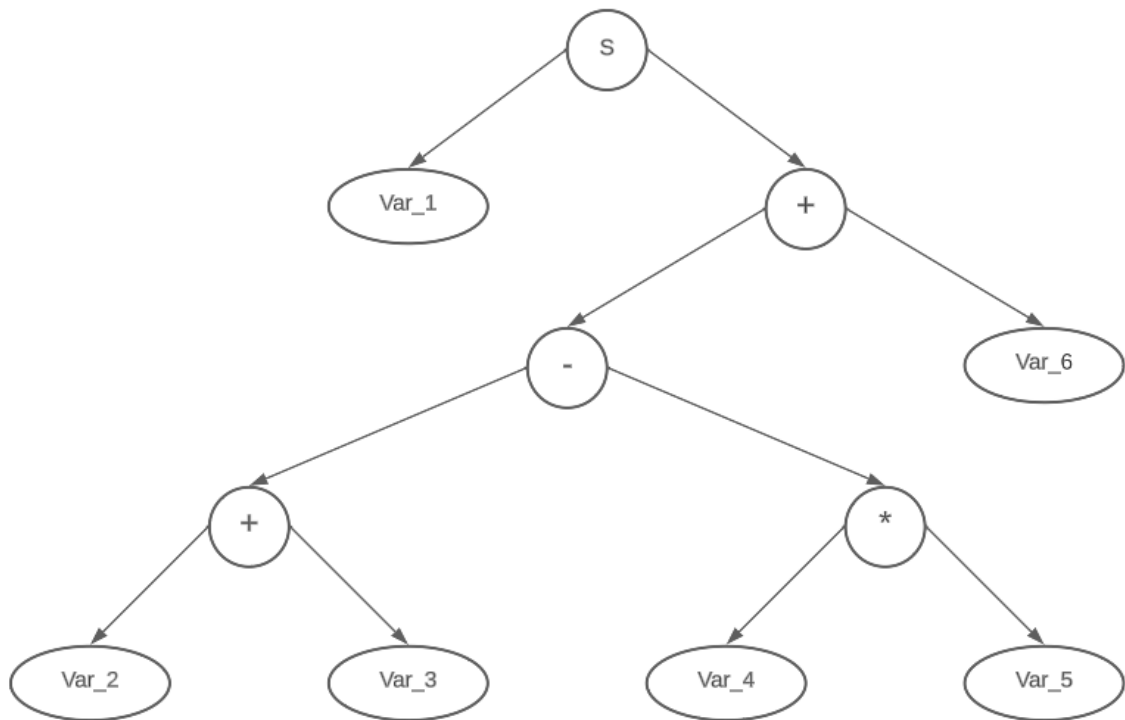
3°) se crea el nodo hoja de var2. luego se crea otro nodo “+” y se le agregan como hijos var2 y el subárbol generado anteriormente.

4°) Se repite el mismo proceso hasta generar todo el lado derecho de la asignación

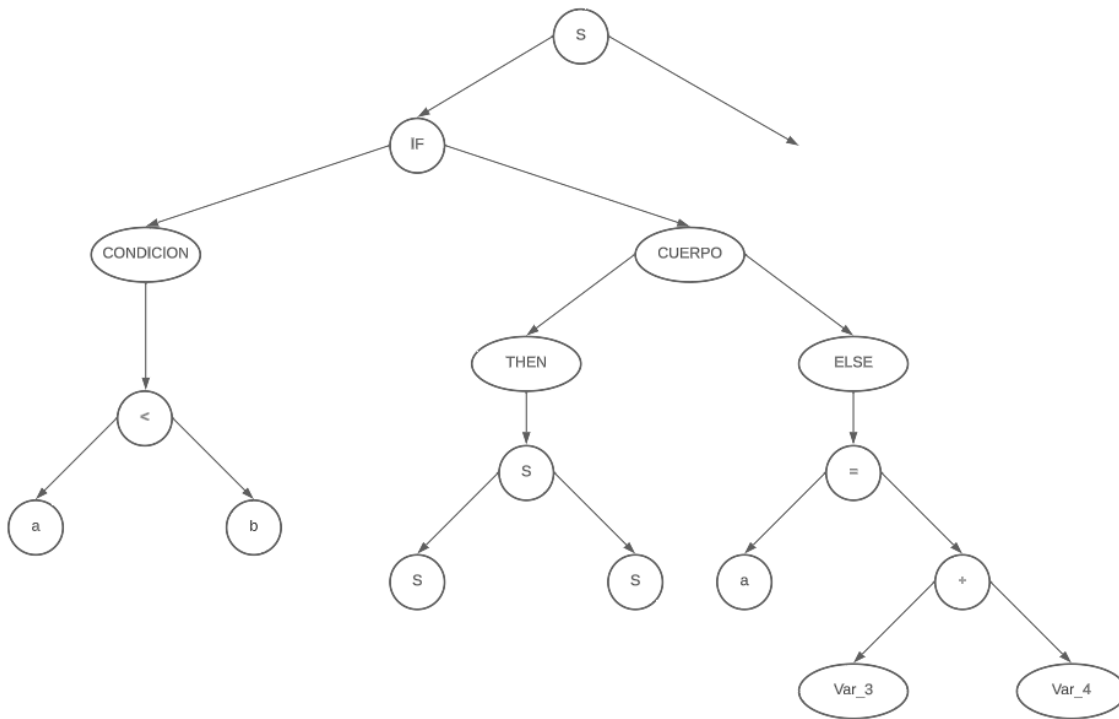
5°) se crea el nodo var1. se crea el nodo “=” y se le asigna el subárbol del lado derecho de la asignación y el lado izquierdo.

Para realizar esta asignación se realizan los chequeos semánticos necesarios para verificar que no hay errores de tipo por ejemplo.

El proceso de generación de los subÁrboles asignación respeta la procedencia de las operaciones, como se ve en el siguiente ejemplo:



Generación de subárbol de sentencia IF:

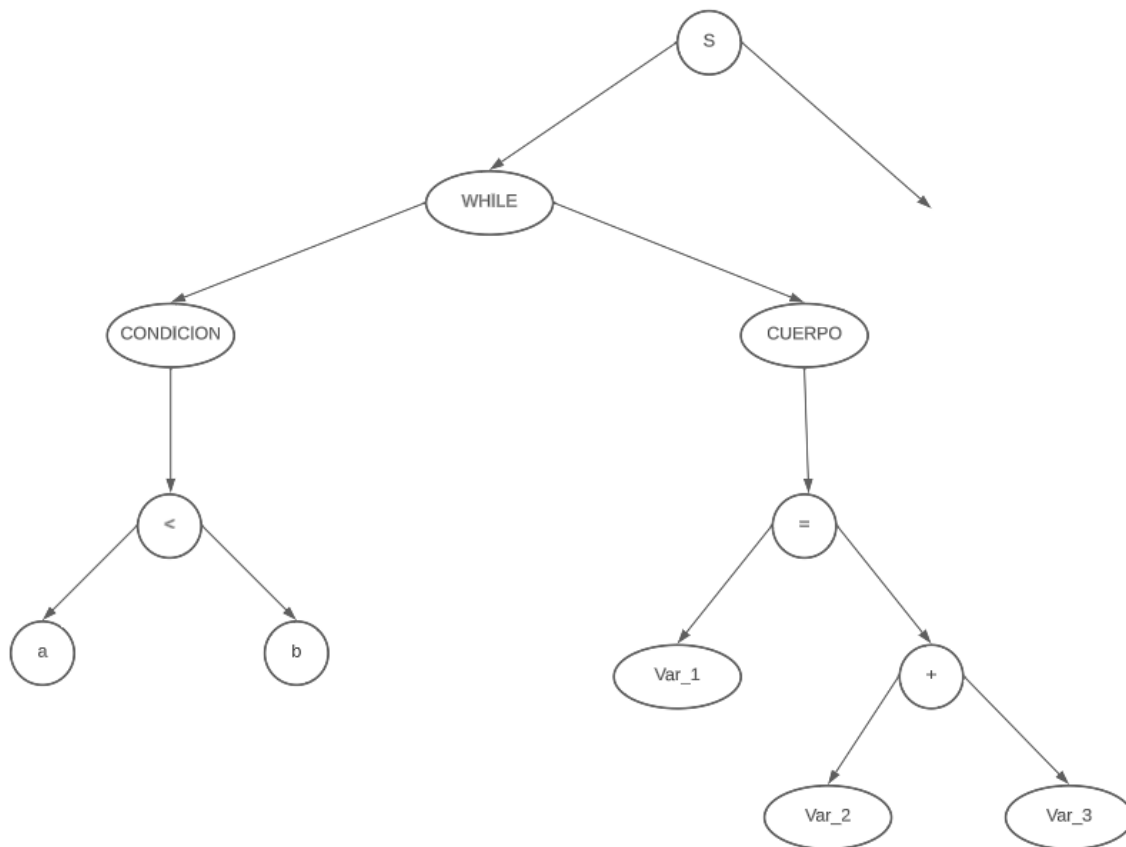


Para la generación de un subÁrbol IF es necesario utilizar nodos de control para las bifurcaciones, tanto para la condición y el cuerpo, como para el bloque then y el se del mismo.

Los pasos son:

- 1º) Género el subÁrbol de la condición
- 2º) Conecto ese subÁrbol a un nodo de control "condición"
- 3º) Creó el cuerpo del IF, genero el subárbol para la rama del then y el subÁrbol para la rama del else.
- 4º) Uno cada subÁrbol a su respectivo nodo de control, then o else, los cuales serán utilizados luego para la generación de código assembler
- 5º) Una vez generado ambas ramas del cuerpo, se conectan con un nodo control "cuerpo"
- 6º) Por último, la subRama "condición" y la subRama "cuerpo" se conectan al nodo de control "IF".
- 7º) Conectar el nodo "IF" al hijo izquierdo de un nodo sentencias del árbol general.

Generación de SubÁrbol de sentencia While:

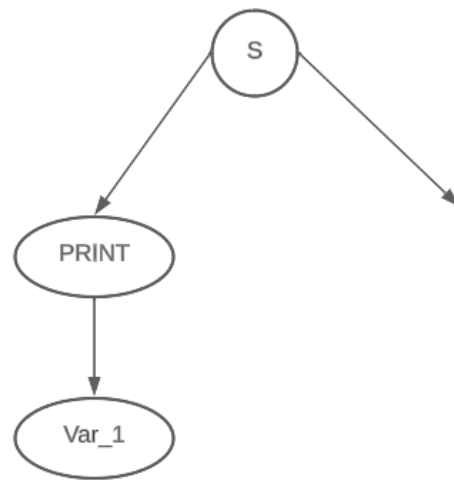
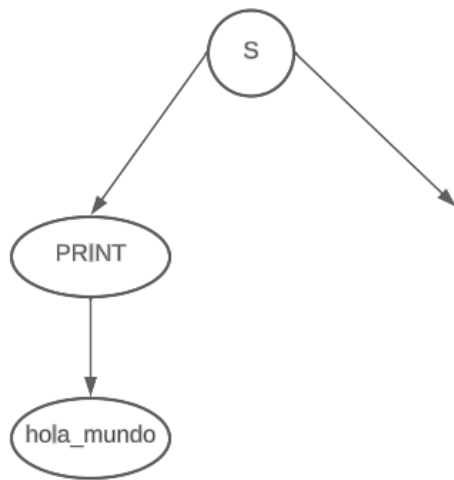


Para la generación de un subÁrbol IF es necesario utilizar nodos de control para las bifurcaciones, tanto para la condición como para el cuerpo.

Los pasos para su generación son:

- 1º) Género el subÁrbol de la condición
- 2º) Conecto ese subÁrbol a un nodo de control "condición"
- 3º) Género el subÁrbol del cuerpo
- 4º) Conectó ese subÁrbol a un nodo de control "cuerpo"
- 5º) Generar un nodo de control "WHILE" y conectar los subÁrboles generados al mismo.
- 6º) Conectar el nodo "WHILE" a un nodo sentencia del programa

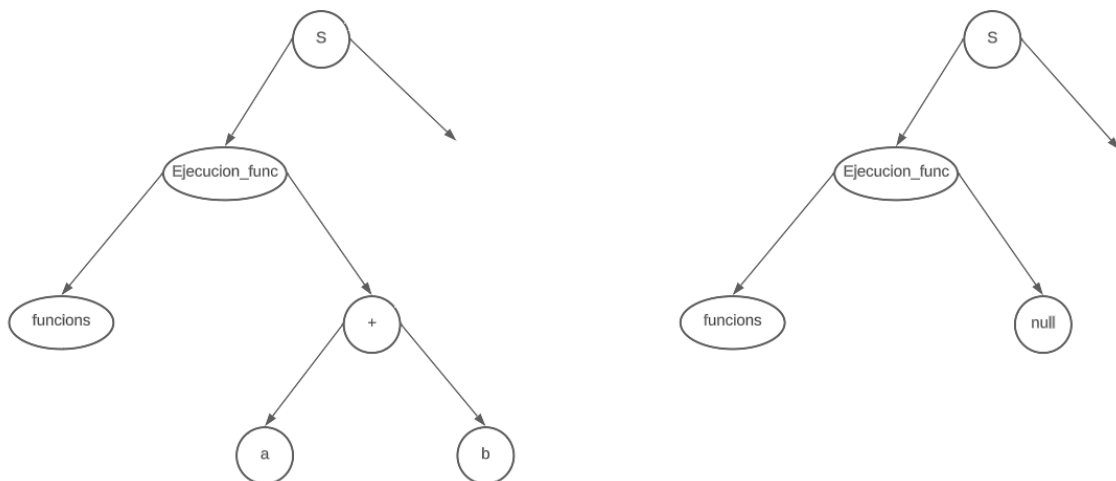
Generación de subÁrbol de sentencia PRINT:



Los pasos para su generación son:

- 1º) Género el nodo con la cadena
- 2º) Lo conecto a un nodo de control "PRINT", el cual será utilizado luego para generar el código assembler
- 3º) Conectar el nodo PRINT creado a un nodo "sentencia" del programa

Generación de subÁrbol de sentencia ejecución Funcion:



Para la ejecución de las funciones decidimos crearlo de la siguiente manera:

- 1°) Generamos el subÁrbol del parámetro en caso de tenerlo y el subÁrbol del nombre de la función
- 2°) Se conectan a un nodo de control “ejecución func”
- 3°) Se conectan a un nodo “sentencia” del programa

Notación posicional de Yacc

La notación posicional Yacc, la utilizamos para:

- Generar los nodos del árbol
- Verificar que las sentencias detectadas no contienen ningún error sintáctico o semántico
- Para asignarles el identificador a los nodos hoja o para realizar modificaciones en la tabla de símbolos.

Con \$n nos posicionamos en el token que nos interesa de la sentencia para obtener su valor, por ejemplo, para la regla:

```
-sentencias_ejecucion_funcion: ID '(' expr_aritmetico ')'
```

Si quiero obtener el valor de ID, utilizó la regla \$1, que hace referencia a su primer valor.

En nuestro caso \$n podía tomar el valor de un String, de un nodo de ÁrbolSintáctico o de un Símbolo, dependiendo de los valores retornados por las sentencias anteriores.

Con \$\$ la regla retorna el valor que le asignemos, ya sea el valor de un token haciendo \$\$=\$1 por ejemplo, \$\$ = simbolo1 o \$\$ = nodoArbol. Retornamos esos valores para luego ser utilizados en reglas posteriores.

Consideración y decisiones tomadas con respecto a los errores

En esta etapa del compilador detectamos errores semánticos, tales como tipos de variables incorrectas, variables no declaradas, etc. Dichos errores son detectados en el procesamiento de las reglas. En caso de encontrar un error semántico o sintáctico,

decidimos utilizar un “nodoError” constante. El mismo es retornado en caso de ocurrido un error y agregado al árbol. Esta acción no genera que se creen nodos innecesarios en el árbol ya que el mismo es constante y siempre que se detecta un error, se hace referencia al mismo nodo. El compilador sigue ejecutando aunque encuentre un error y se almacena al igual que los errores sintácticos en una lista de Strings. Si se detectó un error de una etapa anterior, por ejemplo, sintáctico o léxico, es probable que no se detecten de manera correcta los errores semánticos detectados en esta etapa.

Los errores semántico son mostrados al final de la ejecución del compilador. También a los símbolos se les agregó el atributo de “Usada”, el cual es utilizado una vez finalizada la compilación, para mostrar qué variables fueron usadas en el lado derecho de una asignación o en una comparación.

Comprobaciones Semánticas

Las comprobaciones semánticas las fuimos realizando a la misma vez que las comprobaciones sintácticas. En el procesamiento de cada regla, se verifican todos los errores semánticos propuestos por la cátedra y además incluimos el error semántico de las variables no inicializadas.

Salida Generacion de Codigo Intermedio

La salida para el código intermedio es El árbol sintáctico y una lista de errores semánticos. para el siguiente código:

```
{
    LONG num1,
    LONG var1;var2,
    UINT _var3,
    FLOAT _var4;var5,

    var1 = 10_l,
    var2 = 25_l,
    _var3 = 5_ui,
    _var4 = 10.52,
    var5 = 2.0e+5,

    VOID f1(){
        var1 = 5_l+var2,
        IF(var1 !! 100_l){
            var1 = var1 - 10_l,
        }ELSE{
            var1 = var1 * 2_l,
        }END_IF,
        RETURN,
    },

    PRINT %e1 valor de _var3 es%,

    *{esto es un comentario}*
```

```
WHILE(_var3 <= 1000_ui)DO{
    _var3 = _var3 - 1_ui,
}
```

La salida generada es:

ARBOL:

-Lexema: Programa

-Lexama Nodo: Sentencia

-Hijo Izquierdo:

-Lexama Nodo: =

-Hijo Izquierdo:

-Lexama Nodo Hoja: var1#global

-Hijo Derecho:

-Lexama Nodo Hoja: 10_l

-Hijo Derecho:

-Lexama Nodo: Sentencia

-Hijo Izquierdo:

-Lexama Nodo: =

-Hijo Izquierdo:

-Lexama Nodo Hoja: var2#global

-Hijo Derecho:

-Lexama Nodo Hoja: 25_l

-Hijo Derecho:

-Lexama Nodo: Sentencia

-Hijo Izquierdo:

-Lexama Nodo: =

-Hijo Izquierdo:

-Lexama Nodo Hoja: _var3#global

-Hijo Derecho:

-Lexama Nodo Hoja: 5_ui

-Hijo Derecho:

-Lexama Nodo: Sentencia

-Hijo Izquierdo:

-Lexama Nodo: =

-Hijo Izquierdo:

-Lexama Nodo Hoja: _var4#global

-Hijo Derecho:

-Lexama Nodo Hoja: 10.52

-Hijo Derecho:

-Lexama Nodo: Sentencia

-Hijo Izquierdo:

-Lexama Nodo: =

-Hijo Izquierdo:

-Lexama Nodo Hoja: var5#global

-Hijo Derecho:

-Lexama Nodo Hoja: 2.0e+5

-Hijo Derecho:

-Lexama Nodo: Sentencia

-Hijo Izquierdo:

-Lexama: PRINT

```
Lexema Nodo Hoja: el valor de _var3 es
Hijo Derecho:
Lexama Nodo: Sentencia
Hijo Izquierdo:
Lexama Nodo: WHILE
Hijo Izquierdo:
Lexema: condicion_while
Lexama Nodo: <=
Hijo Izquierdo:
Lexema Nodo Hoja: _var3#global
Hijo Derecho:
Lexema Nodo Hoja: 1000_ui
Hijo Derecho:
Lexema: cuerpo_while
Lexama Nodo: =
Hijo Izquierdo:
Lexema Nodo Hoja: _var3#global
Hijo Derecho:
Lexama Nodo: -
Hijo Izquierdo:
Lexema Nodo Hoja: _var3#global
Hijo Derecho:
Lexema Nodo Hoja: 1_ui
```

WARNINGS:

Variable num1#global no usada
Variable var5#global no usada
Variable _var4#global no usada

Errores Semanticos:

No hubo ningún error Semántico

Generación de la Salida

Generacion de Codigo Assembler

La generación de código assembler es la última etapa del compilador. Esta se encarga de convertir la información obtenida en la tabla de símbolos, así como el árbol que generamos para el assembler.

El código assembler lo generamos a partir del árbol y la tabla de símbolos una vez finalizada su creación. Si no ocurre ningún error léxico, sintáctico y semántico, generamos el código assembler. Para generarlo, creamos una función "getAssembler" en los nodos del árbol, la cual se encarga a partir del lexema, tipo y uso que contiene el nodo, elegir qué comandos assembler agregar a la salida. A su vez, creamos una clase "GeneradorAssembler" la cual se encarga de armar la estructura de la salida del compilador, convirtiendo los datos de la tabla de símbolos en formato assembler, agregando el encabezado y la parte final.

Operaciones Aritméticas

Tanto las operaciones aritméticas como las comparaciones las fuimos generando a partir de if anidados, los cuales dependiendo de las características del nodo y de sus hijos agregaba unos comandos u otros a la salida Assembler. Por ejemplo, si el nodo detectado es una suma en la cual ambos hijos son identificadores o constantes, el código assembler generado será:

```
MOV EAX , "variable"  
ADD EAX , "variable2"  
MOV @aux , EAX
```

Los cambios realizados a los identificadores de las variables para un correcto funcionamiento del assembler fueron las siguientes:

-Identificadores, constantes y cadenas: Se le agregó al principio de cada identificador el signo "\$", para diferenciarlo de las variables auxiliares. Reemplazamos "#" por "\$", ya que el lenguaje ensamblador no reconoce "#" como un carácter válido. También reemplazamos "." por "_" ya que punto es utilizado para otras funciones dentro del assembler. Por último si el identificador contiene un "+" o un "-" se reemplaza por "\$", para resolver conflictos.

Generación de las etiquetas destino y variables auxiliares

Para generar variables auxiliares, utilizamos una pila, la cual lleva los auxiliares utilizados. En el tope de la pila se encuentra la última variable asignada y la cual vamos a usar probablemente en la próxima instrucción. Las variables las armamos concatenando el string "@aux+el indiceAux" que lleva el índice del último auxiliar asignado.

Para el manejo de las etiquetas, también utilizamos una pila de labels, la cual se maneja de la misma manera que la pila de auxiliares. Las etiquetas las usamos para saltos de comparaciones y llamados a funciones.

Modificaciones a las etapas anteriores

Los cambios realizados en las etapas anteriores son:

- La Matriz de transición de estados y la matriz de acciones semánticas se modificaron para cumplir con los requerimientos de la cátedra. Para ello realizamos cambios en los estados por los que pasa un token hasta formarse y modificamos el funcionamiento de ciertas acciones semánticas.
- Realizamos cambios en la gramática, agregando y modificando reglas para resolver la generación de código intermedio de manera correcta.

Implementación de los temas particulares

- Sobreescritura de métodos: Para permitir la sobrescritura de métodos, verificamos que el ámbito de los métodos sea distinto si tienen el mismo nombre.
- Para realizar el chequeo de variable Usada, agregamos al objeto símbolo el atributo usado, el cual al encontrarse en el lado derecho de una asignación o comparación ya considero como usada la variable. Luego al finalizar la ejecución del parser recorro la lista imprimiendo las variables cuyo atributo se encuentre en false
- Forward declaration: Para resolverlo, agregamos una regla que permita esto y luego al querer utilizar un método o variable de dicha clase, verificamos si la misma ya fue declarada o su declaración se realizará con posterioridad.
- Para permitir la herencia por composición, agregamos a las sentencias declarativas de clases, la posibilidad de declarar una clase de la forma "nombre clase," lo cual permite que la clase herede todos los métodos y variables de la clase que hereda. Esta herencia puede tener hasta tres niveles, en caso de superarse se notifica el error.

Controles en Tiempo de Ejecución

Los controles son los siguientes:

- División por 0 en enteros y flotantes: para realizar este punto la lógica utilizada fue, para los enteros Largos y enteros Sin Signo, comparamos su valor con cero, en caso de ser igual saltó a la etiqueta error y terminó la ejecución del programa. Para el caso de los flotantes, primero realizó la resta del número consigo mismo y si el resultado es cero, significa que el divisor es cero y saltó a la etiqueta de error.
- Overflow en suma de datos flotantes: Primero realizó la suma, luego muevo los flags al registro AH. Verificar si hubo overflow y saltar a la etiqueta de error en caso de que se cumpla el error.
- Overflow en producto de entero: Luego de realizar la multiplicación, verificó con la instrucción JO que no hubo overflow, en caso de haberlo salto a la etiqueta del error y culminar la ejecución.
- Para realizar la conversión explícita, al detectar un nodo TOF, utiliza la instrucción FILD para pasar el número de 16 a 32 bits y así poder realizar la operación requerida.

Salida Assembler

La salida que proporciona el compilador al terminar su ejecución es la siguiente:

Para el código:

```
{
    LONG var1,

    FLOAT var2;var3,

    var1 = 20_l,

    var2 = 10.,

    var3 = TOF(var1) + var2,

    IF(var3 == 30.0){
        PRINT %var3 es igual a 30.0%,
    }ELSE{
        PRINT %var3 es distinto a 30.0%,
    }END_IF,

    var1 = var1 + 1_l,

    IF(var1 == 21_l){
        PRINT %var1 es igual a 21%,
    }ELSE{
        PRINT %var1 es distinto de 21%,
    }END_IF,
}
```

La salida generada es:

```
.386
.model flat, stdcall
option casemap :none
include \masm32\include\windows.inc
include \masm32\include\kernel32.inc
include \masm32\include\user32.inc
includelib \masm32\lib\kernel32.lib
includelib \masm32\lib\user32.lib
.data
$errorDivisionPorCero db " Error Assembler:No se puede realizar la division por
cero"
$errorOverflowMultEntero db " Error Assembler: overflow en producto de enteros "
$errorOverflowSumaFlotantes db " Error Assembler: overflow en la suma de flotantes
"
$30_0 dd 30.0
$var3$global dd ?
$var1$es$distinto$de$21 db "var1 es distinto de 21" , 0
```

```

$var1$es$igual$a$21 db "var1 es igual a 21" , 0
$var2$global dd ?
$var3$es$igual$a$30_0 db "var3 es igual a 30.0" , 0
@aux2 dd ?
@aux1 dd ?
$20_1 dd 20
$var1$global dd ?
$21_1 dd 21
$1_1 dd 1
$10_ dd 10.
$var3$es$distinto$a$30_0 db "var3 es distinto a 30.0" , 0
.code
main:
MOV EAX , $20_1
MOV $var1$global, EAX
FLD $10_
FST $var2$global
FILD $var1$global
FLD @aux1
FST $var3$global
FLD $var3$global
FCOM $30_0
FSTSW AX
SAHF
JNE label1
invoke MessageBox, NULL, addr $var3$es$igual$a$30_0, addr $var3$es$igual$a$30_0,
MB_OK
JMP label2
label1:
invoke      MessageBox,      NULL,      addr      $var3$es$distinto$a$30_0,      addr
$var3$es$distinto$a$30_0, MB_OK
label2:
MOV EAX , $var1$global
ADD EAX , $1_1
MOV @aux2 , EAX
MOV EAX , @aux2
MOV $var1$global , EAX
MOV EAX , $var1$global
CMP EAX , $21_1
JNE label3
invoke MessageBox, NULL, addr $var1$es$igual$a$21, addr $var1$es$igual$a$21, MB_OK
JMP label4
label3:
invoke      MessageBox,      NULL,      addr      $var1$es$distinto$de$21,      addr
$var1$es$distinto$de$21, MB_OK
label4:
invoke ExitProcess, 0
errorDivisionPorCero:
invoke MessageBox, NULL, addr $errorDivisionPorCero , addr $errorDivisionPorCero ,
MB_OK
invoke ExitProcess, 0
errorOverflowMultEntero:
invoke      MessageBox,      NULL,      addr      $errorOverflowMultEntero      ,      addr
$errorOverflowMultEntero , MB_OK
invoke ExitProcess, 0
errorOverflowSumaFlotantes:
invoke      MessageBox,      NULL,      addr      $errorOverflowSumaFlotantes      ,      addr
$errorOverflowSumaFlotantes , MB_OK
invoke ExitProcess, 0

```



```
end main
```

Ejecución del archivo Jar

El archivo .jar fue creado para 2 versiones distintas de jdk, para la 17.0.8 y para la 11. Ambos archivos están subidos al drive para que sea utilizado el correspondiente.

VERSIÓN DEL JDK:

```
JAVA_RUNTIME_VERSION="17.0.8+9-LTS-211"
```

```
JAVA_VERSION="17.0.8"
```

```
JAVA_VERSION_DATE="2023-07-18"
```

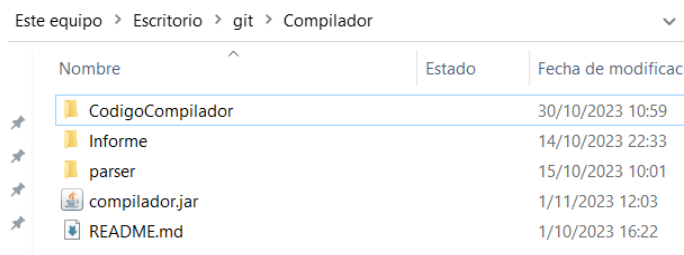
VERSIÓN DEL ECLIPSE:

```
Version: 2023-09 (4.29.0)
```

Para poder ejecutar el jar es necesario seguir los siguientes pasos:

1°) abrir una consola de comandos de windows(cmd)

2°) pararte en la carpeta en la que se encuentra el archivo compilador.jar utilizando el comando cd "direccion de la carpeta" (en este caso es en la carpeta ../compilador)



Este equipo > Escritorio > git > Compilador			
Nombre	Estado	Fecha de modificac	
CodigoCompilador		30/10/2023 10:59	
Informe		14/10/2023 22:33	
parser		15/10/2023 10:01	
compilador.jar		1/11/2023 12:03	
README.md		1/10/2023 16:22	

3°) Una vez parado en la carpeta ejecutamos el siguiente comando:

```
"java -jar compilador.jar"
```

4°) En caso de que no ocurra ningún error tendría que aparecer un mensaje en la consola de la siguiente forma:

```
La carpeta a partir de la cual el compilador toma el valor ingresado es \CodigoCompilador\src\Testeos\  
Ejemplo de dato a ingresar: [CPP_general.txt] o Sintactico\CPP_general.txt
```

```
Ingrese la direccion del archivo a compilar:
```

La manera de ingresar la dirección de un código para ser compilado es, por ejemplo, si el archivo txt se encuentra en "C:\Users\rolus\OneDrive\Escritorio\git\Compilador\CodigoCompilador\src\Testeos\lexico.LX_pruebaConErrores.txt", El dato que debo ingresar al compilador es: lexico\LX_pruebaConErrores.txt.

Para probar cada código es necesario volver a ejecutar los pasos anteriores.

Conclusiones

Durante el desarrollo del trabajo práctico pudimos comprender el funcionamiento del léxico y el sintáctico de un compilador. Como un compilador analiza un código y a partir de este genera tokens e identifica sentencias que luego van a ser usadas en etapas posteriores. Como son detectados los errores léxicos y sintácticos y una idea general de cómo funciona un compilador hasta este punto. En las etapas 3 y 4 pudimos ver e implementar como un compilador a partir de la gramática y los tokens generados por las etapas anteriores genera las estructuras necesarias para luego poder generar el código assembler que se ejecutará en el procesador para obtener los resultados deseados con el código que le proporcionamos. Pudimos ver los errores semánticos y en tiempo de ejecución y cómo los detectamos. También nos encontramos con dificultades las cuales pudimos resolver pensando las cosas de diferente manera. Cómo a partir de un código el compilador se encarga de analizar cada palabra y verificar que todo sea correcto, llegando a ejecutar y realizar las operaciones proporcionadas en el código de prueba.