

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	3
Generación de Código Intermedio	3
Árbol Sintáctico	3
Manejo de creación de funciones anidadas	8
Manejo de declaraciones de clases anidadas y métodos con funciones	9
Notación posicional de Yacc	10
Consideración y decisiones tomadas con respecto a los errores	10
Comprobaciones Semánticas	11
Salida Generacion deCodigo Intermedio	11
Generación de la Salida	14
Generacion deCodigo Assembler	14
Operaciones Aritméticas	14
Generación de las etiquetas destino y variables auxiliares	14
Modificaciones a las etapas anteriores	15
Implementación de los temas particulares	15
Controles en Tiempo de Ejecución	16
Salida Assembler	16
Secciones con Problemas	18
Ejecución del archivo Jar	20
Conclusiones	20

Introducción

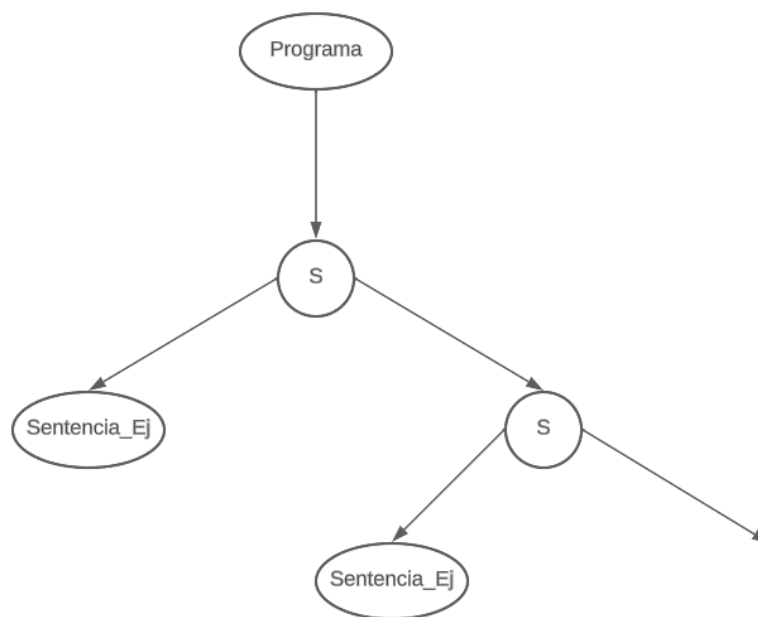
El presente informe exhibe la continuación del desarrollo de la creación de un compilador basado en el lenguaje Java. En este informe se verá la creación de la Generación de código intermedio (Árbol Sintáctico y comprobaciones sintácticas) y Generación de salida (Assembler y salida de Assebre). Las decisiones 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



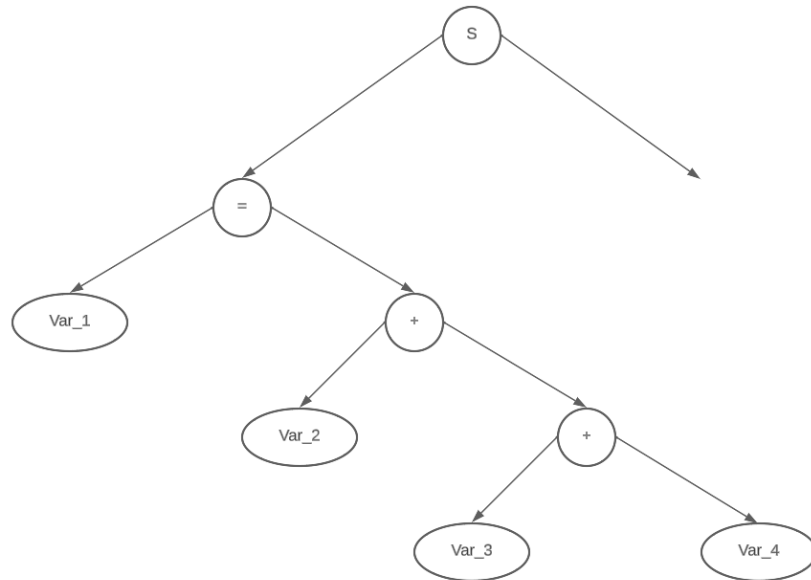
La forma en la que se genera el árbol es a partir de un nodo Control llamado “programa”, se le crea un hijo llamado sentencia. La manera en la que se crea el árbol es:

- 1° Género un nodo Sentencia (S)
- 2° Se genera una sentencia ejecutable y se agrega en su hijo izquierdo, ya sea, una sentencia ejecutable de tipo asignación, llamado Función, etc.

3° Cuando genera una nueva rama que corresponde a una sentencia ejecutable y la quiero agregar, voy a verificar si el hijo izquierdo del nodo S es null, en caso de serlo, agregó como hijo izquierdo de S a nueva rama generada. 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

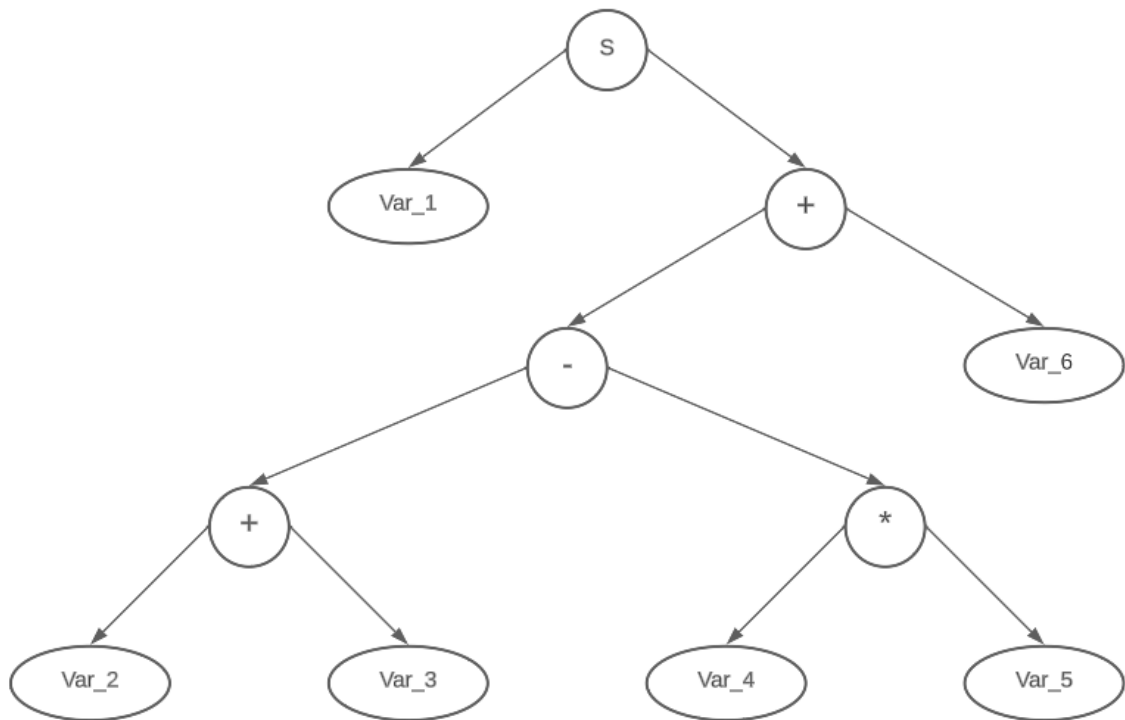
3° Se crea el nodo hoja de var2 luego se crea otro nodo "+" y se le agregan como hijos var2 y el subárbol generado anteriormente.

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

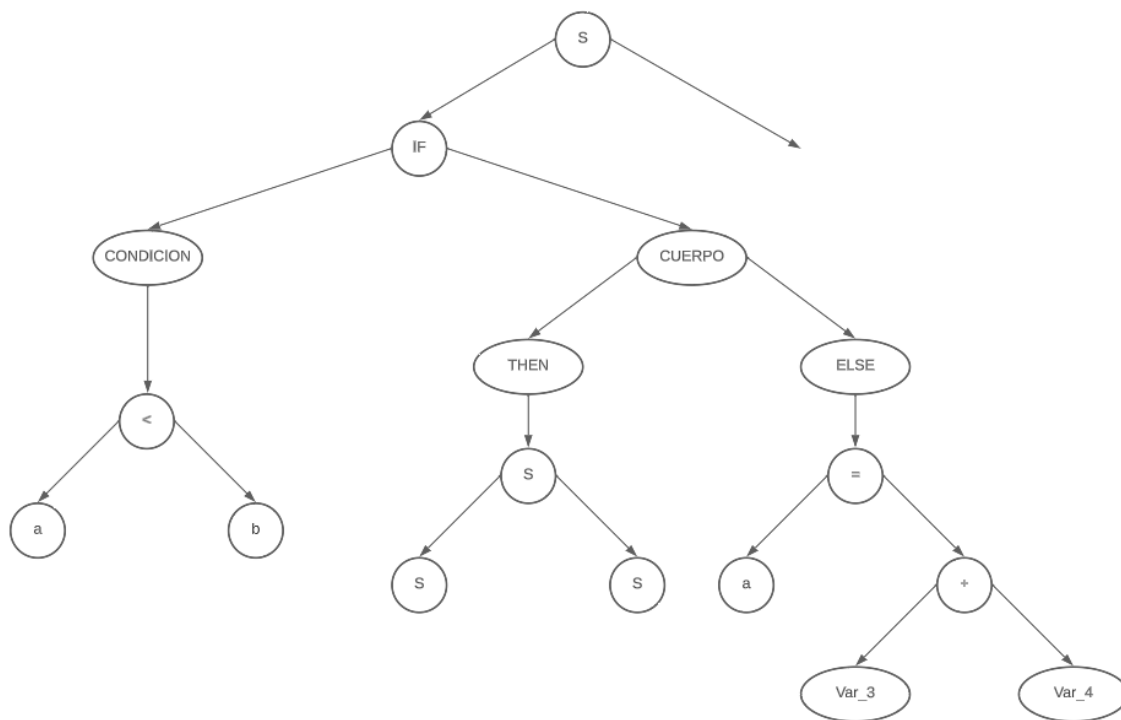
5° Se crea el nodo var1 se crea el nodo "=" y se le asigna el subárbol del lado derecho de la asignación y el lado izquierdo.

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

El proceso de generación de los subÁrboles asignación respeta la precedencia de las operaciones, como se ve en el siguiente diagrama de 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 else.

Los pasos son:

1° Género el subÁrbol de la condición

2° Conecto ese subÁrbol a un nodo de control "condición"

3° Creó el cuerpo del IF, genero el subárbol para la rama del then y el subÁrbol para la rama del else.

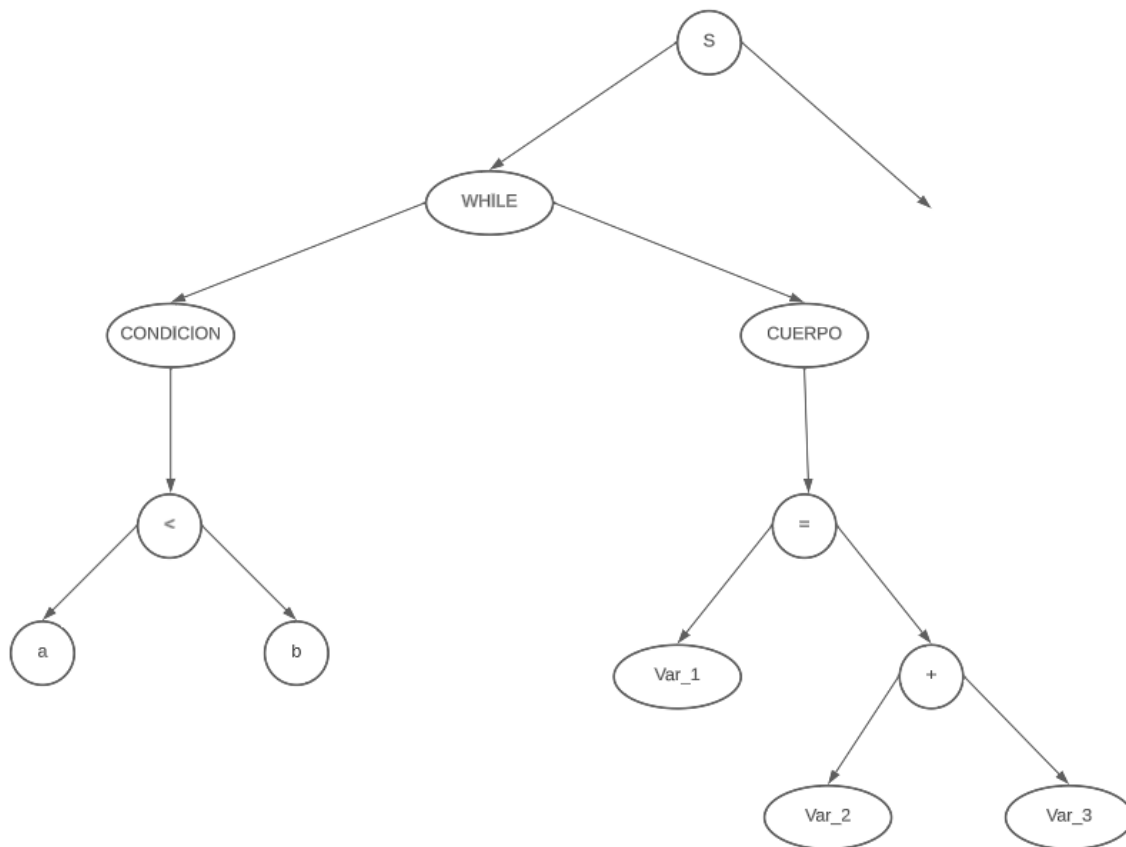
4° Se une cada subÁrbol a su respectivo nodo de control, then o else, los cuales serán utilizados luego para la generación de código assembler

5° Una vez generado ambas ramas del cuerpo, se conectan con un nodo control "cuerpo"

6° Por último, la subRama "condición" y la subRama "cuerpo" se conectan al nodo de control "IF".

7° Conectar el nodo "IF" al hijo izquierdo de un nodo sentencias del árbol general.

Generación de SubÁrbol de sentencia While:

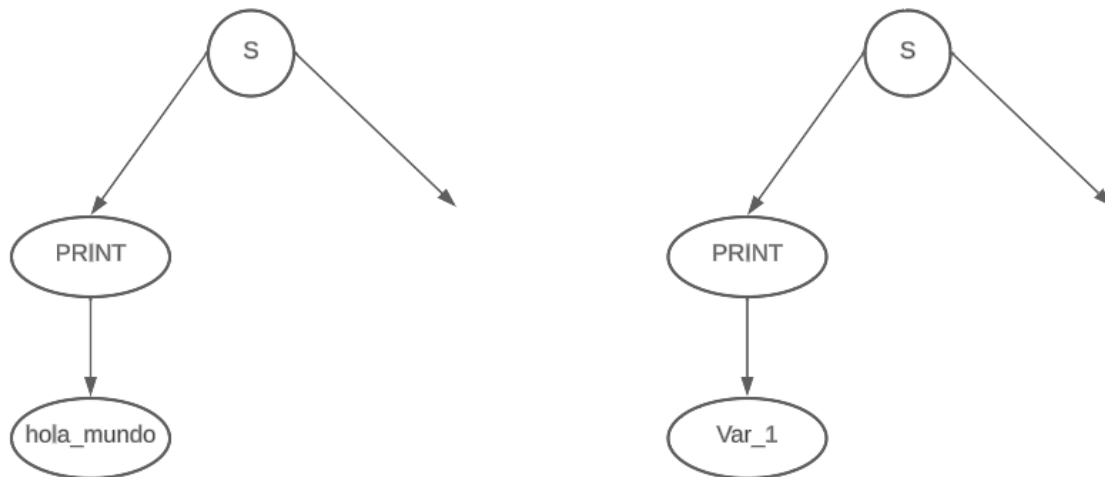


Para la generación de un subÁrbol WHILE es necesario utilizar nodos de control para las bifurcaciones, tanto para la condición como para el cuerpo.

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

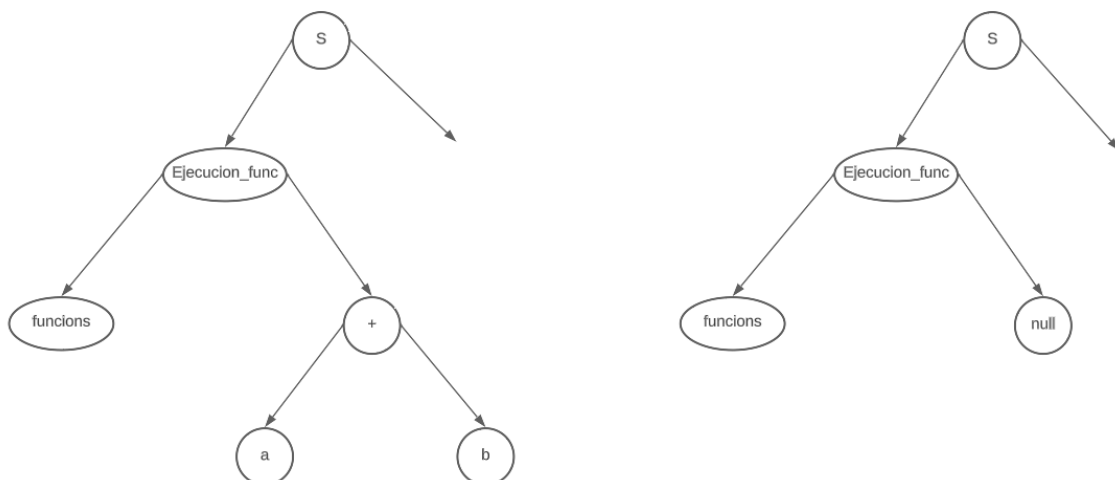
Generación de subÁrbol de sentencia PRINT:



Los pasos para su generación son:

- 1° Género el nodo con la cadena
- 2° Lo conecto a un nodo de control "PRINT", el cual será utilizado luego para generar el código assembler
- 3° 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, está conformado por 2 nodos hijos, uno es el lexema de la función y el otro el parámetro de esta:

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

Manejo de creación de funciones anidadas

Cuando se quiere generar una función, como vimos anteriormente, se genera un Árbol Sintáctico. En el momento que se quiere generar una función dentro de otra función,

se debe tener cuidado para poder separar estos 2 árboles y no se pisen entre ellos. Ejemplo de un caso donde se crear una función dentro de otra:

```
VOID fun1 (LONG parametro) {
    LONG _numero3,
    _numero3 = parametro,

    VOID fun2 () {
        _numero1 = _numero2 + _numero3,

        IF (_numero1 == 25_1) {
            PRINT %fun2 _numero1 es igual a 25%,
        } ELSE {
            PRINT %fun2 _numero1 es distinto de 25%,
        } END_IF,
        RETURN,
    },

    fun2(),

    RETURN,
},
```

Al compilar el código, cuando se encuentra la definición de una función, se comienza a generar su árbol. Lo cual lleva un puntero a la raíz de este y un puntero que lo recorre el árbol para sumar los subÁrboles que se suman este por las distintas sentencias dentro de la función. Si dentro de las sentencias de la función se encuentra con una declaración de función, lo que se hace es, guardar los 2 punteros en una pila y hacer que estos punteros apunten a un nuevo árbol. De esta manera sigue, como antes con esta nueva función. Cuando termina la declaración de función, verifica si la pila está vacía, en caso de no estarlo desapila y sigue con la función anterior. Esto se repite hasta que la pila esté vacía.

Manejo de declaraciones de clases anidadas y métodos con funciones

Cuando se crea una clase, a medida que se crean los atributos de esta se va guardando los lexemas en una lista. Que luego son asociados a la clase, esta lista se utiliza para generar las variables de los objetos que sean del tipo de clase, para generar las variables en otras clases que hereden de esta o para asociarla a un método y este puede saber cuando modifica una variable de la clase y modifique la variables del objeto que llamó a este método.

La generación de métodos es similar a la de funciones, pero además, se lleva un contador para saber cuantas funciones se generan dentro de otras y un booleano que sirve para saber si está dentro de un método y así no guardar en la lista de la clase, las variables que se declaren. Cuando se define una función se aumenta en 1 el contador y cuando se

finaliza se reduce. En el caso que llegue a 0, se entiende que se terminó la creación de un método, entonces, se le asocia la lista de variables creadas dentro de la clase, para poder identificarlas, y se pone en false el booleano que identifica que está dentro de un método.

En el caso, que se declare una clase estando dentro de una, la información de la lista de variables, el contador, el lexema de la clase y el booleano se guarda su contenido en una pila. Luego, se les asigna los valores correspondientes para la nueva clase a crear, cuando finaliza de crear la clase, al igual que funciones, verifica si la pila está vacía, en caso de no estarlo extrae la información de la anterior clase y continua. Este proceso se repite hasta vaciar la pila.

Notación posicional de Yacc

Yacc (Yet Another Compiler Compiler) es una herramienta que genera analizadores sintácticos (parsers) para procesar lenguajes de programación. Utiliza una gramática formal definida en un archivo.y para producir un parser en Java u otro lenguaje, siendo comúnmente usado junto con herramientas como Lex para construir compiladores e intérpretes.

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 \$\$ 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.

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.

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. Los errores que se detectan, léxico, sintáctico y semántico, pueden afectar los que se detectan en las etapas posteriores, por ejemplo, si en la primera etapa que se detectan errores léxicos, detecta uno probablemente los errores sintácticos y semánticos detectados sean erróneos.

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 se realizan 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 deCodigo 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:
- Lexema Nodo Hoja: var1#global
- Hijo Derecho:
- Lexema Nodo Hoja: 10_l
- Hijo Derecho:
- Lexama Nodo: Sentencia
- Hijo Izquierdo:
- Lexama Nodo: =
- Hijo Izquierdo:
- Lexema Nodo Hoja: var2#global
- Hijo Derecho:
- Lexema Nodo Hoja: 25_l
- Hijo Derecho:
- Lexama Nodo: Sentencia
- Hijo Izquierdo:
- Lexama Nodo: =
- Hijo Izquierdo:
- Lexema Nodo Hoja: _var3#global
- Hijo Derecho:
- Lexema Nodo Hoja: 5_ui
- Hijo Derecho:
- Lexama Nodo: Sentencia
- Hijo Izquierdo:
- Lexama Nodo: =
- Hijo Izquierdo:
- Lexema Nodo Hoja: _var4#global
- Hijo Derecho:
- Lexema Nodo Hoja: 10.52
- Hijo Derecho:
- Lexama Nodo: Sentencia
- Hijo Izquierdo:
- Lexama Nodo: =
- Hijo Izquierdo:
- Lexema Nodo Hoja: var5#global
- Hijo Derecho:
- Lexema Nodo Hoja: 2.0e+5
- Hijo Derecho:
- Lexama Nodo: Sentencia
- Hijo Izquierdo:

```

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

Las operaciones aritméticas son acciones matemáticas fundamentales que se aplican a los números para realizar cálculos. Incluyen la suma (+), que combina números; la resta (-), que encuentra la diferencia entre ellos; la multiplicación (*), que halla el producto de dos o más números; y la división (/), que determina el cociente.

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
```

La secuencia de instrucciones anteriores primero mueve el valor de la variable al registro EAX de 32 bits, luego le suma el contenido de variable2 al registro EAX. Y por último le asigna a @aux el valor resultante en EAX.

Los cambios realizados a los identificadores de las variables para un correcto funcionamiento del assembler fueron los 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 el 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 un registro LIFO (Last In First Out) comúnmente llamado 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" y el entero "indiceAux" que lleva el índice del último auxiliar asignado dando por resultado "@aux+indiceAux".

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

- 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 (Uso de nombre), lo que se realiza es, pedir a la clase los distintos lexemas de sus atributos y estos se utilizan para generar las nuevas variables, con el nombre de esa clase concatenada al principio (en caso de ser herencia en una clase) o del objeto (en caso de ser la creación de un objeto de clase). En el caso de los métodos, se realiza de la misma manera pero estos además tienen un atributo donde tienen el lexema del método de la clase original (por ejemplo, el método c1.c2.met1() tiene guardado el lexema de met1()) para facilitar su invocación.
- Para la sobreescritura de métodos decidimos no realizar nada más de lo hecho, ya que estaría implementado por medio de la herencia por composición - Uso de nombre. Ya que en el caso de querer crear otro método con el mismo nombre de la clase que hereda, esto se permitiría. Ejemplo:

```
CLASS c1 {  
    VOID met() {  
        *{codigo}*  
    }  
}
```

```

}
CLASS c2 {
    VOID met() {
        *{codigo}*
    }
    c1,
}
c2 objt,
objt.met(),
objt.c1.met(),

```

Como se puede ver en este caso, se puede acceder al método creado en c2 que tienen el mismo nombre que c1.

Si se quisiera optar por la opción de modificar directamente el método en c1 y dejar el método de c2. Lo que pensamos como una posible implementación sería, utilizar una palabra reservada similar a “Override” que nos indique que el siguiente método será la reescritura de un método heredado. Cuando se quisiera comenzar a generar el método, se buscaría dentro de una lista de “métodos heredados” y se verificará que exista un método heredado con el mismo lexema del que se quiere crear. Por ejemplo, usando la situación del ejemplo de antes, dentro de c2 si utilizamos “Override” antes de declarar met, se buscaría entre los métodos heredados de c1 y se cambiará la dirección que utiliza para invocarse por la de el met creado en c2. Pero si c1 hereda de otra clase que también tienen el mismo nombre no tengo forma de diferenciarlas y decir cual sobrescribir.

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 compara el resultado con una constante que contiene el máximo valor permitido. Si el resultado supera ese valor, salta a la etiqueta error y termina la ejecución.
- Overflow en producto de entero: Luego de realizar la multiplicación, verificó con la instrucción JO(saltar si hubo desbordamiento) que no hubo overflow, en caso de haberlo salto a la etiqueta del error y culmina la ejecución.
- Para realizar la conversión explícita, al detectar un nodo TOF (Conversión Explícita), 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_l dd 20
$var1$global dd ?
$21_l 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

```

Secciones con Problemas

En esta sección se enuncian y explican los errores sin solución durante la generación del compilador:

- Error al crear una clase anidada dentro de otra y crear un método y querer usar las variables de la clase padre: Esto genera errores porque no encontramos la forma de generar las variables correctamente, por lo que al ejecutar el método los resultados pueden ser inesperados. Se puede compilar y ejecutar pero los resultados pueden ser incorrectos.
- Error al crear un objeto dentro de una clase que a su vez hereda de otra: Esto genera que se creen variables de más que no corresponden y algunas variables no se crean. Esto ocurre porque no llegamos a generar de manera correcta los lexemas. Se puede usar el código pero en algunos casos puede generar errores inesperados.
- Error al querer hacer una conversión TOF con un menos_menos: Por alguna razón el assembler no resuelve de manera correcta el código cuando encuentro una variable como: TOF(var1--).
- Error al mandar como parámetro de una función una variable con TOF o con menos_menos: No se genera bien el assembler, por lo que no es posible utilizarlo. Esto se soluciona agregando ifs y consultando si el lexema tiene menos menos o Tof, y escribir la instrucción de una u otra forma dependiendo la combinación que tenga.
- Error en la comparación de mayor o igual, menor o igual en números Flotantes: Por alguna razón el assembler generado para realizar estas comparaciones no funciona de manera correcta. Esto creemos que puede ser porque los flags al realizar la comparación no se cargan de manera correcta, por lo que no funciona correctamente.

Estos errores no los llegamos a resolver porque nos quedamos sin tiempo.

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"

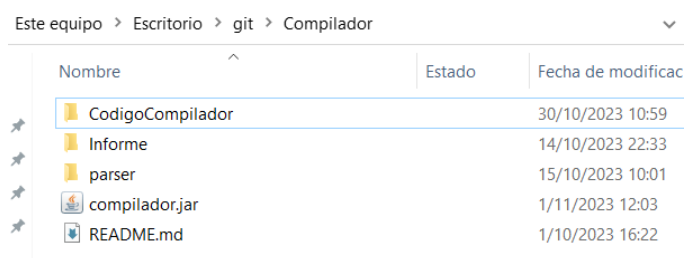
VERSIÓN DEL ECLIPSE:

Version: 2023-09 (4.29.0)

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

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

2° Ubicarse 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)



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 ubicado 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 la ejecución de este trabajo práctico, hemos comprendido cómo hacer un análisis léxico, sintáctico y semántico de un código. Se ha revelado cómo un compilador

examina meticulosamente un código, generando tokens e identificando sentencias que desempeñarán un papel fundamental en las fases subsecuentes. Asimismo, hemos profundizado en la detección de errores léxicos y sintácticos, obteniendo una visión general del funcionamiento del compilador hasta este punto.

En las etapas 3 y 4, hemos explorado e implementado el proceso mediante el cual un compilador, basándose en la gramática y la generación de tokens previamente codificada, estructura de manera precisa los elementos necesarios para la subsiguiente generación del código en lenguaje ensamblador. Este código resultante es crucial, ya que será ejecutado en el procesador con el propósito de obtener los resultados esperados, de acuerdo con las instrucciones proporcionadas en el código fuente.

Además, hemos abordado la identificación y gestión de errores semánticos y de tiempo de ejecución, adquiriendo la capacidad de detectarlos y corregirlos de manera efectiva. A lo largo de este proceso, nos hemos enfrentado a desafíos que, mediante un enfoque reflexivo y creativo, hemos logrado superar con éxito. Este análisis detallado revela cómo, a partir de un código fuente, el compilador se encarga de examinar minuciosamente cada palabra, verificando la corrección de la sintaxis y, finalmente, ejecutando las operaciones especificadas en el código de prueba.