INFORME DE COMPILADOR

Tomás Achával Vinuesa

28 DE NOVIEMBRE DE 2025

UNVIME
Ingeniería en sistemas de información

Base teórica que justifica el desarrollo de un compilador

Antes de poder ejecutar un programa, primero, debe traducirse a un formato en el que una computadora pueda ejecutarlo y el sistema de software que se encarga de esta traducción se llama compilador.

un compilador es un programa que puede leer un programa en un lenguaje fuente y traducirlo en un programa equivalente en otro lenguaje destino

Estos compiladores se diferencian de los intérpretes, ya que estos últimos son otro tipo de procesador de lenguaje, pero, que en vez de producir un programa destino como una traducción, el intérprete nos da la apariencia de ejecutar directamente las operaciones especificadas en el programa de origen con las entradas proporcionadas

El programa destino en lenguaje máquina que produce un compilador es, por lo general, más rápido que un intérprete al momento de asignar las entradas a las salidas. No obstante, por lo regular, el intérprete puede ofrecer mejores diagnósticos de error que un compilador, ya que ejecuta el programa fuente instrucción por instrucción.

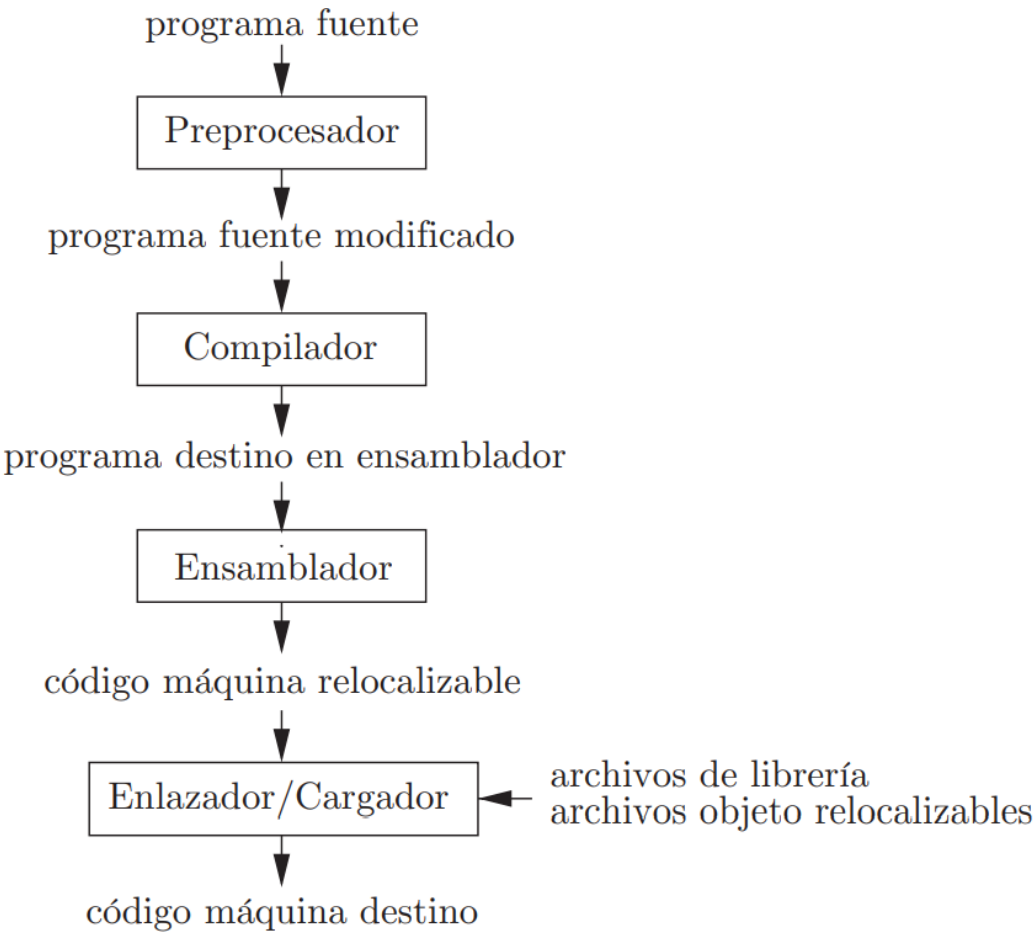
Por lo tanto, la razón por la que desarrollamos un compilador es para disponer de un puente que nos permita traducir un programa en lenguaje de alto nivel, que es un lenguaje más fácil de entender para nosotros, a un lenguaje máquina que la computadora pueda ejecutar, obteniendo así programas eficientes sin sacrificar la claridad ni la portabilidad del código fuente

Ciclo de vida de la traducción de un programa fuente a destino.

Un programa fuente puede dividirse en módulos guardados en archivos separados. La tarea de recolectar el programa de origen se confía algunas veces a un programa separado, llamado preprocesador. El preprocesador también puede expandir algunos fragmentos de código abreviados de uso frecuente, llamados macros, en instrucciones del lenguaje fuente.

Después, el programa fuente modificado se alimenta a un compilador. El compilador puede producir un programa destino en ensamblador como su salida, ya que es más fácil producir el lenguaje ensamblador como salida y es más fácil su depuración. A continuación, el lenguaje ensamblador se procesa mediante un programa llamado ensamblador, el cual produce código máquina relocalizable como su salida.

A menudo, los programas extensos se compilan en partes, por lo que quizás haya que enlazar (vincular) el código máquina relocalizable con otros archivos objeto relocalizables y archivos de biblioteca para producir el código que se ejecute en realidad en la máquina. El enlazador resuelve las direcciones de memoria externas, en donde el código en un archivo puede hacer referencia a una ubicación en otro archivo. Entonces, el cargador reúne todos los archivos objeto ejecutables en la memoria para su ejecución.



Al explicar esto también me parece interesante comentar cómo trabajan los compiladores de java, donde, un programa fuente en Java puede compilarse primero en un formato intermedio, llamado bytecodes y después, una máquina virtual los interpreta. Lo que nos da el beneficio de que los bytecodes que se compilan en una máquina pueden interpretarse en otra.

Funcionamiento de un compilador

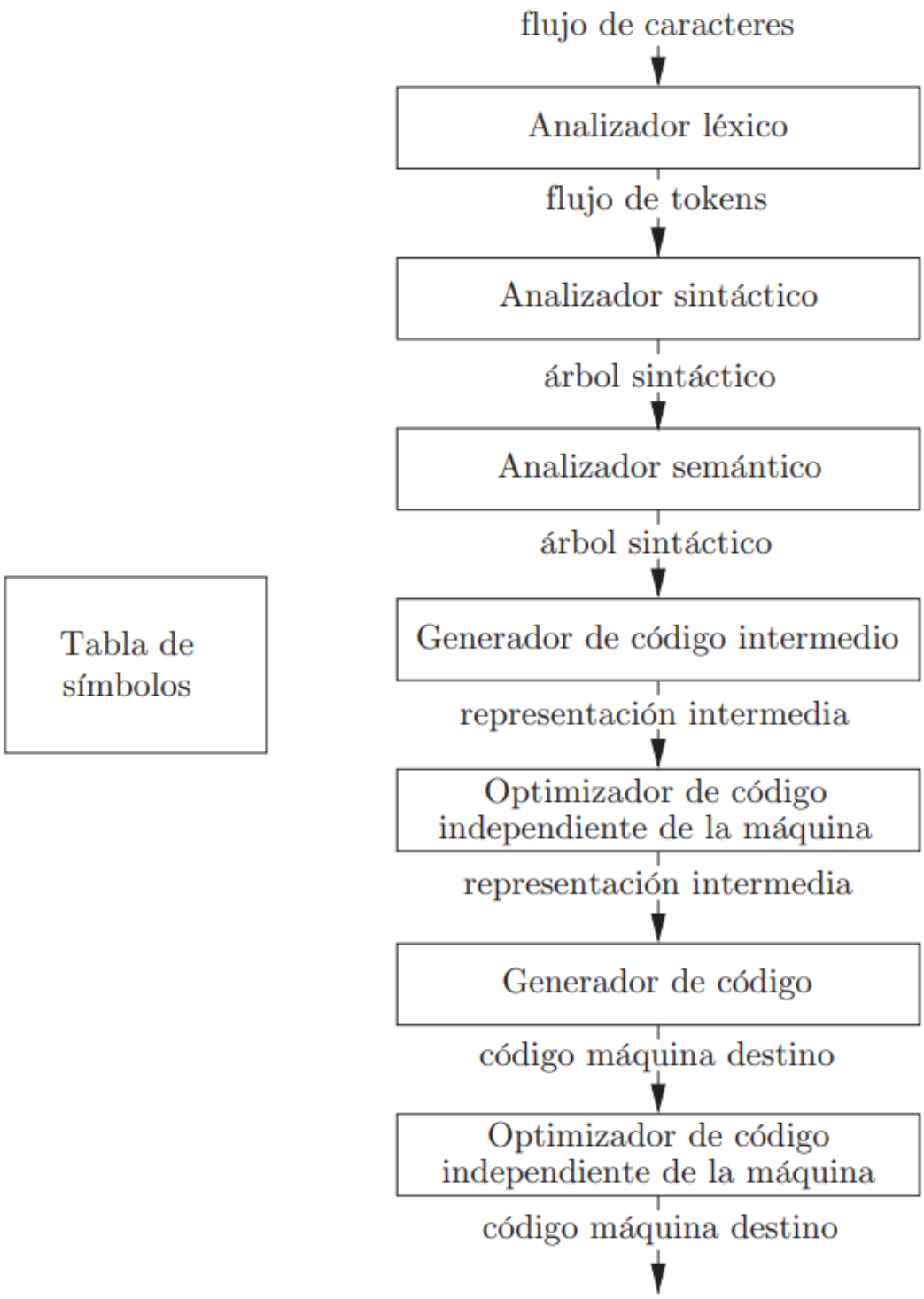
El compilador mapea un programa fuente a un programa destino con equivalencia semántica, hay dos procesos en esta asignación: análisis y síntesis.

El análisis (backend) divide el programa fuente en componentes e impone una estructura gramatical sobre ellas. Después utiliza esta estructura para crear una representación intermedia del programa fuente. Si la parte del análisis detecta que el programa fuente está mal formado en cuanto a la sintaxis, o que no tiene una semántica consistente, entonces debe proporcionar mensajes informativos para que el usuario pueda corregirlo. La parte del análisis también recolecta información sobre el programa fuente y la almacena en una estructura de datos llamada tabla de símbolos, la cual se pasa junto con la representación intermedia a la parte de la síntesis.

La síntesis(frontend) construye el programa destino deseado a partir de la representación intermedia y de la información en la tabla de símbolos.

Fases de un compilador

El proceso de compilación opera como una secuencia de fases, cada una de las cuales transforma una representación del programa fuente en otro. En nuestro caso, el compilador se realizará usando Java y las herramientas pertinentes a cada una de las fases como veremos a continuación



Algunos compiladores tienen una fase de optimización de código independiente de la máquina, para realizar transformaciones sobre la representación intermedia, para que el back-end pueda producir un mejor programa destino de lo que hubiera producido con una representación intermedia sin optimizar

Desarrollo del compilador

El compilador desarrollado consta de 2 partes ejecutables, la primera es la clase Principal.java que se encarga de automatizar la generación de los códigos fuentes de los analizadores léxico y sintáctico (Evitando el uso de consola para este fin), la cual trabaja con rutas relativas para facilitar el uso del compilador en otros equipos.

Lo primero en generarse en Principal.java es un analizador léxico usando JFlex y basándose en el contenido del archivo Lexer.flex, donde se definen expresiones regulares (patrones) que describen cómo se ve cada uno de los tokens que enumeramos en otra clase llamada Tokens.java

Una de las expresiones regulares que tenemos en el Lexer.flex es el siguiente:

```
L = [a-zA-Z_]
D = [0-9]
ID = {L}({L}|{D})*
```

Que describe como es el patrón de un ID, luego nos dice que ese mismo ID corresponde al Token “Identificador”

```
{ID} {lexeme=yytext(); return Identificador;}
```

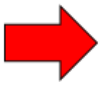
Estos Tokens están enumerados en Tokens.java del siguiente modo:

```
public enum Tokens {
    Int,
    Identificador,
```

Con esta información, JFlex es capaz de construir un autómata de estados finitos (La clase Lexer.java), el cual es nuestro analizador léxico que lee los caracteres y los va agrupando en unidades con significado que llamaremos lexemas y luego se clasifican estos lexemas asignándoles una etiqueta (Token) que hayamos definido anteriormente en la clase Tokens.java

Ejemplo de un análisis léxico de este tipo:

```
int main(){
int x = 5;
return x;
}
```



LINEA 1	Simbolo
	int -> Token: Int main -> Token: Main (-> Token: ParentesisAbre) -> Token: ParentesisCierra { -> Token: LlaveAbre
LINEA 2	
	int -> Token: Int x -> Token: Identificador = -> Token: Igual 5 -> Token: Numero ; -> Token: PuntoYComa
LINEA 3	
	return -> Token: Return x -> Token: Identificador ; -> Token: PuntoYComa
LINEA 4	
.	} -> Token: LlaveCierra

Luego, nuestra clase Principal.java, instancia otro analizador léxico (LexerCup.java) pero que a diferencia del primero, la salida de este es menos visual y más enfocado en su uso por parte del Parser (analizador sintáctico), donde el LexerCup lee los caracteres, los agrupa en lexemas válidos y los empaqueta en un tipo de objeto especial llamado Symbol con los que luego trabajará el Parser

Si el primer analizador léxico trabajaba con Lexer.flex y Tokens.java, Ahora el segundo tendrá al archivo LexerCup.flex como la fuente de especificaciones para JFlex Y tendrá algunas configuraciones específicas que están orientadas a poder trabajar con la librería CUP (Librería que genera el analizador sintáctico) y en lugar de trabajar con tokens enumerados, ahora se genera automáticamente una clase llamada sym.java que funciona como “diccionario” de códigos, donde transforma estos nombres en números

Ejemplo de como se ve la declaración de un lexema en LexerCup.flex:

```
{ID} {return new Symbol(sym.Identificador, yychar, yyline, yytext());}
```

Ejemplo de como se ve el symbol Identificador en la clase sym.java:

```
/** CUP generated class containing symbol constants. */
public class sym {
  /* terminals */
  public static final int Identificador = 19;
```

Ahora sigue la generación del analizador sintáctico (Sintax.java) y del ya mencionado archivo sym.java (estos se generan juntos). El analizador sintáctico se encargará de verificar que la secuencia de tokens que le entrega el lexer, tenga sentido gramatical según las reglas de nuestro lenguaje, y esas reglas se definen en otro archivo Sintax.cup, en el cual se describe:

- Un mecanismo para capturar los errores cuando ocurran en la sintaxis (Es la parte de parser code y es código que se inyecta dentro del analizador sintáctico generado)

```
parser code {:
    private Symbol s;

    public void syntax_error(Symbol s){
        this.s = s;
    }

    public Symbol getS(){
        return this.s;
    }
:};
```

- Luego especifica los símbolos terminales (aquellos que son hojas del árbol de análisis y que por lo tanto no se pueden dividir más) y no terminales (Estos son símbolos compuestos que representan estructuras gramaticales formadas por combinaciones de símbolos terminales y no terminales, vendrían a ser los nodos intermedios y la raíz del árbol)

```
terminal Int, Bool, Void, Main, Return, True, False, Igual, Suma, Resta, Multiplicacion, Division,
        ParentesisAbre, ParentesisCierra, PuntoYComa, LlaveAbre, LlaveCierra, Identificador, Numero,
        If, Then, Else, While, And, Or, Not, Mayor, Menor, IgualIgual, Modulo, Var, UMENOS;

non terminal NodoPrograma INICIO;
non terminal List<NodoAST> LISTA_SENTENCIAS;
non terminal NodoAST SENTENCIA, DECLARACION, ASIGNACION, RETORNO, EXPRESION, IF_SENTENCIA, WHILE_SENTENCIA, BLOQUE;
```

- Se definen las reglas de precedencia que especifican el orden de prioridad para resolver las operaciones

```
precedence left Or;
precedence left And;
precedence left IgualIgual;
precedence left Mayor, Menor;
precedence left Suma, Resta;
precedence left Multiplicacion, Division, Modulo;
precedence right Not, UMENOS;
precedence nonassoc Else; //Esto para evitar que el else se asocie a más de un if o al if equivocado
```

- Se declara el símbolo inicial de la gramática (o sea, por donde el parser debe comenzar a analizar el código fuente)

```
start with INICIO;
```

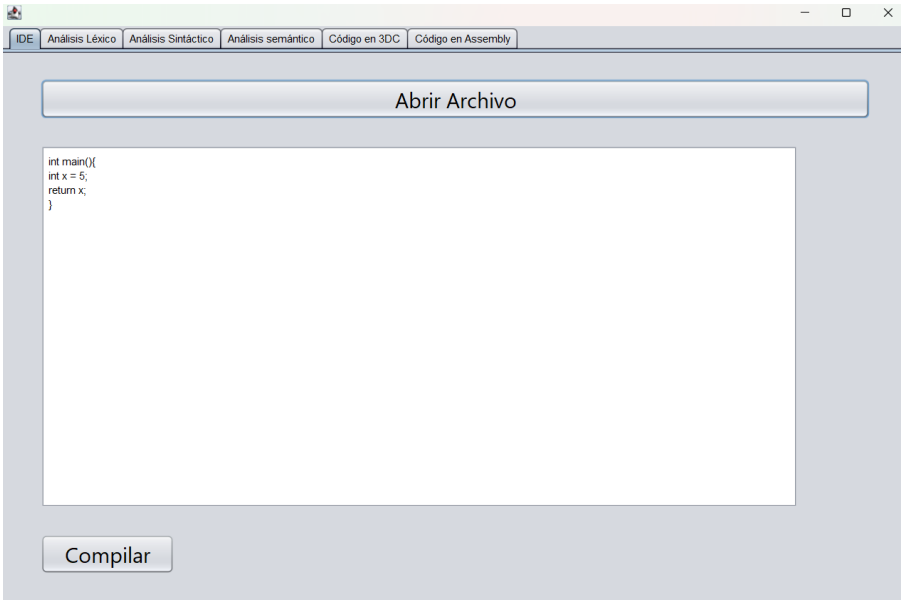
- Luego están las reglas de producción que definen las distintas estructuras válidas que pueden tener los símbolos no terminales (Estas indican en cada caso que deben ejecutar en caso de encontrar esa estructura en el código)

  Por ejemplo tenemos la estructura de inicio que es la regla raiz, ya que define que todo programa válido en mi lenguaje debe ser una función main (de cualquiera de los tipos válidos) y que el cuerpo de la función (las distintas sentencias) deben estar dentro de llaves

```
INICIO ::= Int:tipoRetorno Main ParentesisAbre ParentesisCierra LlaveAbre LISTA_SENTENCIAS:sentencias LlaveCierra
        {:
            List<NodoAST> lista = (sentencias == null) ? new ArrayList<>() : (List<NodoAST>)sentencias;
            RESULT = new NodoPrograma(tipoRetorno.toString(), lista);
        :}
    | Void:tipoRetorno Main ParentesisAbre ParentesisCierra LlaveAbre LISTA_SENTENCIAS:sentencias LlaveCierra
        {:
            List<NodoAST> lista = (sentencias == null) ? new ArrayList<>() : (List<NodoAST>)sentencias;
            RESULT = new NodoPrograma(tipoRetorno.toString(), lista);
        :}
    | Bool:tipoRetorno Main ParentesisAbre ParentesisCierra LlaveAbre LISTA_SENTENCIAS:sentencias LlaveCierra
        {:
            List<NodoAST> lista = (sentencias == null) ? new ArrayList<>() : (List<NodoAST>)sentencias;
            RESULT = new NodoPrograma(tipoRetorno.toString(), lista);
        :}
;
```

La acción que provoca que se encuentre esa gramática, es la generación del árbol AST, el cual comienza desde los nodos más simples (nodos hojas) y va creando otros nodos más grandes que van conteniendo a los más pequeños (Construyendo el AST desde abajo hacia arriba) hasta que llega a nuestro nodo raíz llamado NodoPrograma

El otro Archivo ejecutable que tenemos es la parte gráfica del compilador (la clase FrmPrincipal.java), que funciona como un IDE sencillo, donde tiene un botón que permite abrir archivos de texto (Códigos fuentes) desde nuestra computadora o podemos escribir en la misma pantalla



Además de esto tiene un botón con el texto Compilar que llama al método compilar() el cual se encarga de ir llamando en orden a otros métodos para ir haciendo el proceso de compilación del código fuente (el mismo método se detiene en el momento en el que encuentra un error en alguna de las partes del código fuente).

El orden de ejecución de este método es:

1. Se limpian las pantallas (excepto la del IDE)
2. Análisis Léxico
3. Análisis Sintáctico y generación del AST
4. Impresión del árbol AST

5. Generación de la tabla de símbolos
6. Análisis semántico
7. Impresión de la tabla de símbolos
8. Generación del código intermedio (código de 3 direcciones 3DC)
9. Traducción del código 3DC a Assembly x86-64

Como ya se explicó hasta la forma en que se genera el árbol AST, ahora vamos a ver cómo es el proceso a través del cual podemos mostrar el árbol:

Para empezar, todos los nodos de nuestro árbol AST siguen una plantilla que es determinada por la clase abstracta NodoAST:

```
public abstract class NodoAST {
    public abstract String analizar(TablaSimbolos ts) throws ExcepcionSemantica;
    public abstract Object generarCodigo();
    public abstract String imprimir(String indent);
}
```

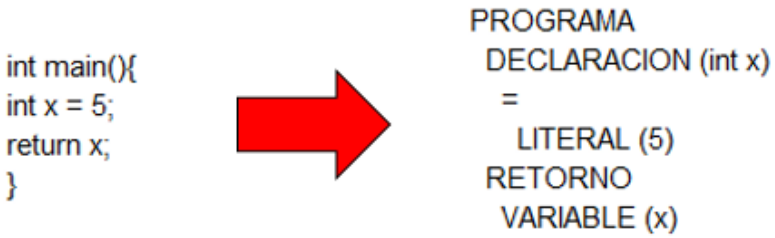
Esta define los métodos que van a tener que sobreescribir cada una de las clases que extiendan de esta y es a través del polimorfismo que cada uno de los nodos se puede ir mostrando, a través del método imprimir, de la manera más conveniente. Además de esto, los propios nodos tienen en su método imprimir llamadas a los métodos imprimir de sus clases hijas, donde lo único que se pasan es el nivel de indentación para que, gráficamente, se pueda entender los niveles del árbol

Por ejemplo para mostrar todo un bloque IF se hace lo siguiente:

```
@Override
public String imprimir(String indent) {
    String res = indent + "IF\n" + condicion.imprimir(indent + "  ") + indent + "THEN\n" + bloqueThen.imprimir(indent + "  ");
    if (bloqueElse != null) { //Con el bloque else hago lo mismo que con el bloque Then si es que hay una parte else
        res += indent + "ELSE\n" + bloqueElse.imprimir(indent + "  ");
    }
    return res; //Retorna el arbol de todo el bloque if-else
}
```

Donde se llama al método imprimir de sus nodos hijos “condición” y “bloqueElse” y estos a su vez llaman al método imprimir de sus propios nodos hijos y así hasta construir todo el árbol AST de manera gráfica

Ejemplo de cómo se vería el AST de un código fuente específico:



Los distintos tipos de Nodos que se pueden generar son:

- NodoAsignacion
- NodoBloque
- NodoDeclaración
- NodoIdentificador
- NodoIf
- NodoLiteral

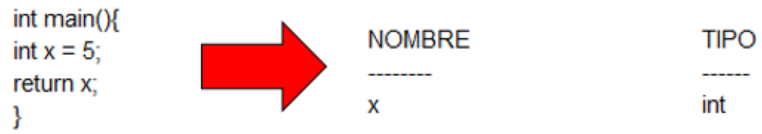
- NodoOperacionBinaria
- NodoOperacionUnaria
- NodoPrograma
- NodoRetorno
- NodoWhile

Cada uno de estos se encarga de distintas partes del código fuente (En el código original del compilador se detalla casi a nivel de instrucción por instrucción que es lo que va haciendo cada nodo)

Ahora lo siguiente que vamos a ver es cómo se construye la tabla de símbolos y se realiza el análisis semántico.

La tabla de símbolos es una estructura de datos que se encarga de almacenar la información sobre los identificadores, en mi caso almacena el nombre y el tipo de cada variable, ya que esta tabla será utilizada en el análisis semántico para verificar cosas como: Que la variable que se quiere usar esté ya declarada, que no esté declarada una variable que estamos queriendo declarar, que los tipos de datos que queremos operar/retornar sean los adecuados.

Ejemplo de la tabla de símbolos generada:



La tabla de símbolos se va llenando en el método analizar de la clase NodoDeclaración.java, donde primero se crea el símbolo y luego se intenta insertar

```
Simbolo nuevoSimbolo = new Simbolo(nombre: this.identificador, tipo: this.tipo); //Crea el simbolo con su nombre y su tipo
if (!ts.insertar(s: nuevoSimbolo)) {
    throw new ExcepcionSemantica("Error Semántico: La variable '" + this.identificador + "' ya ha sido declarada.");
}//Inserta el simbolo en la tabla de simbolos y si ese método nos devuelve true (no hubo problemas) entonces no lanza
```

Esta tabla de símbolos será fundamental durante el proceso del análisis semántico, este análisis consiste en verificar la coherencia del código fuente, considera aspectos como:

- Que el tipo de retorno de la función main sea el esperado
- Que se operen tipos de datos compatibles con las operaciones
- Que una expresión retorne el tipo de dato correcto
- Que se asignen tipos de datos correctos a las variables declaradas
- Que una variable esté declarada antes de usarla
- Que una variable no esté ya declarada antes de declararla
- Etc.

El análisis semántico se realizará, al igual que la forma de mostrar el árbol AST, en forma de cascada, donde la clase padre llama al método Analizar de su clase hija y cada uno de estos revisa si su nodo cumple lo esperado y avisa a la clase padre sobre el tipo de dato que va a devolver. Si ocurre un error, se produce una excepción semántica que se muestra en pantalla y se detiene el proceso de compilación

Ejemplo de análisis de la semántica en el nodo If:

```
@Override
public String analizar(TablaSimbolos ts) throws ExcepcionSemantica {
    String tipoCond = condicion.analizar(ts);
    if (!tipoCond.equals(anObject: "bool")) {
        throw new ExcepcionSemantica("La condición del IF debe ser de tipo bool. Se encontró: " + tipoCond);
    } //Revisamos que la condición sea de tipo booleana
    bloqueThen.analizar(ts); //analiza el bloque Then
    if (bloqueElse != null) { //Si definimos una parte else entonces también llama a su método analizar
        bloqueElse.analizar(ts);
    }
    return "OK"; //si no encuentra errores semanticos en ninguna de las 3 partes nos indica que todo está bien
}
```

Acá vemos como este método llama al método analizar del nodo hijo (condición) y este le retorna el tipo de dato que va a devolver y en caso de que no devuelva un booleano, lanza una excepción

Para mostrar la tabla de símbolos el proceso es muy simple, ya que la misma clase TablaSimbolos.java tiene una función pública que retorna un string con la tabla

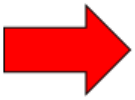
```
public String imprimir() { //Esto es simplem
    StringBuilder sb = new StringBuilder();
    sb.append(str: "NOMBRE\t\tTIPO\n");
    sb.append(str: "--------\t\t------\n");

    for (Simbolo s : tabla.values()) { //Cic
        sb.append(str: s.getNombre())
          .append(str: "\t\t")
          .append(str: s.getTipo())
          .append(str: "\n");
    }
    return sb.toString(); //lo devuelvo como
}
```

Una vez que ya se revisó que se cumplan todas las reglas léxicas, sintácticas y semánticas se realiza la generación de código en 3 direcciones (o código 3DC), este proceso sigue exactamente el mismo principio que la impresión del AST y el análisis semántico, que consiste en la generación de código en cascada donde cada una de las clases padre va llamando al método para generar código de sus clases hijas y estas van a añadiendo a una clase “generador” sus distintas líneas de código hasta que se genera todo el código 3DC

Ejemplo del código 3DC generado a partir del código fuente:

```
int main(){
int x = 5;
if(x==5) then{
x = x+ 1;
}
else{
x= x-1;
}
return x;
}
```



```
x = 5
t1 = x == 5
if t1 == 0 goto L1
t2 = x + 1
x = t2
goto L2
L1:
t3 = x - 1
x = t3
L2:
return x
```

Ejemplo del método GenerarCódigo() de la clase NodoIf.java:

```
@Override
public Object generarCodigo() {
    String cond = (String) condicion.generarCodigo(); //llama al metodo generarCodigo del n
    String etiquetaElse = Generador.getInstancia().nuevaEtiqueta(); //Genera las nuevas eti
    String etiquetaFin = Generador.getInstancia().nuevaEtiqueta(); //Como para cuando termi
    Generador.getInstancia().escribir("if " + cond + " == 0 goto " + etiquetaElse); //Acá r
    bloqueThen.generarCodigo(); //Si el código no saltó en el paso anterior (caso condición
    Generador.getInstancia().escribir("goto " + etiquetaFin); //Esto es básicamente que si
    Generador.getInstancia().escribir(etiquetaElse + ":"); //esto indicará donde ejecutar s
    if (bloqueElse != null) { //en caso de que exista una parte Else entonces generamos el
        bloqueElse.generarCodigo();
    }
    Generador.getInstancia().escribir(etiquetaFin + ":"); //Escribimos la etiqueta de fin j

    return null; //Sentencias no necesitan retornar nada
}
```

Acá lo más interesante está realmente en la clase generador.java la cual es la encargada de llevar la cuenta de las etiquetas y las variables temporales ya utilizadas y poder ser accesible desde cualquier parte del código para que se pueda escribir 3DC en él. En esta parte es donde aparecen varios problemas como garantizar que no se asignen nombres de variables o etiquetas a más de 1 cosa, entre otros problemas descritos como comentarios en la clase, y que son resueltos gracias a la implementación del patrón singleton que asegura que solo exista 1 instancia del generador y que esta sea accesible en todo el programa

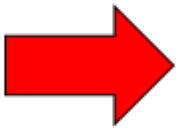
```
public static Generador getInstancia() { //
    if (instancia == null) {
        instancia = new Generador();
    }
    return instancia;
} //Fundamental para implementar singleton
```

Esto asegura que solo pueda existir 1 instancia del generador

Finalmente este código que se muestra en pantalla es traducido por otra clase llamada GeneradorAssembly.java que básicamente funciona como un switch donde analiza la estructura de cada línea de código del 3DC generado y produce un nuevo código equivalente a este pero en Assembly y este se muestra en pantalla:

Ejemplo del código Assembly generado a partir del código 3DC:

```
x = 5
t1 = x == 5
if t1 == 0 goto L1
t2 = x + 1
x = t2
goto L2
L1:
t3 = x - 1
x = t3
L2:
return x
```



```
    .section .data
    x: .quad 0
    t1: .quad 0
    t2: .quad 0
    t3: .quad 0

    .section .text
    .global main
main:
    push rbp
    mov rbp, rsp

    mov qword [x], 5
    mov rax, [x]
    mov [t1], rax
    mov rax, [t1]
    cmp rax, 0
    je L1
    mov rax, [x]
    mov rbx, 1
    add rax, rbx
    mov [t2], rax
    mov rax, [t2]
    mov [x], rax
    jmp L2
L1:
    mov rax, [x]
    mov rbx, 1
    sub rax, rbx
    mov [t3], rax
    mov rax, [t3]
    mov [x], rax
L2:
    mov rax, [x]
    leave
    ret

    pop rbp
    ret
```