

UNIVERSIDAD TECNICA DE ORURO

FACULTAD NACIONAL DE INGENIERIA

INGENIERIA INFORMATICA



TRABAJO FINAL INF – 3631 A “COMPILADORES”

**DESARROLLO DE UN COMPILADOR PARA ARDUINO EN ESPAÑOL**



**Docente:** M.Sc. Ing. Saul Mamani Mamani

**Estudiante:** Jenry Sullcani Nina, Dylan Escobar Siles

**Fecha de presentacion y defensa:** 30 de diciembre de 2024

Oruro – Bolivia

2024

Contenido

[1](#_gjdgxs) INTRODUCCIÓN 1

[1.1](#_30j0zll) Presentación 1

[1.2](#_1fob9te) Objetivo 1

[1.3](#_3znysh7) Lenguaje propuesto 1

[1.4](#_2et92p0) Alcances 1

[1.5](#_tyjcwt) Ingeniería del proyecto 1

[2](#_3dy6vkm) ANALISIS LÉXICO 2

[2.1](#_1t3h5sf) Definición del alfabeto 2

[2.2](#_4d34og8) Definición de expresiones regulares 2

[2.3](#_2s8eyo1) Implementación del analizador léxico 2

[2.4](#_17dp8vu) Tabla de símbolos 2

[2.5](#_3rdcrjn) Autómatas finitos 2

[3](#_26in1rg) ANÁLISIS SINTÁCTICO 2

[3.1](#_lnxbz9) Diseño de la gramática 2

[3.2](#_35nkun2) Árbol de análisis sintáctico 2

[3.3](#_1ksv4uv) Diagrama de sintaxis 2

[3.4](#_44sinio) Implementación del analizador sintáctico 2

[4](#_2jxsxqh) ANÁLISIS SEMÁNTICO 2

[4.1](#_z337ya) Árbol de análisis semántico 2

[4.2](#_3j2qqm3) Tabla de símbolos extendida 2

[4.3](#_1y810tw) Implementación del analizador semántico 2

[5](#_4i7ojhp) GENERACIÓN DE CÓDIGO 2

[5.1](#_2xcytpi) Implementación 2

[6](#_1ci93xb) CONCLUSIONES Y RECOMENDACIONES 2

[BIBLIOGRAFÍA 3](#_3whwml4)

[ANEXOS 3](#_2bn6wsx)

[BIBLIOGRAFIA 4](#_qsh70q)

[ANEXOS 5](#_3as4poj)

# INTRODUCCIÓN

## Presentación

El presente proyecto consiste en el desarrollo de un compilador que tiene como objetivo transformar clases escritas en Java en sentencias SQL para la creación de bases de datos relacionales. Este compilador está diseñado para facilitar el proceso de diseño y generación de estructuras de bases de datos a partir de un código fuente Java bien estructurado, permitiendo una integración eficiente entre el diseño orientado a objetos y el modelo relacional.

El proyecto se enfoca en interpretar y analizar clases Java que contengan atributos definidos, con el fin de generar tablas SQL correspondientes. Cada atributo de la clase se convierte en una columna de la tabla, mientras que los nombres de las clases se transforman en nombres de tablas. Este proceso automatiza una tarea clave en el desarrollo de aplicaciones que combinan programación orientada a objetos con bases de datos.

## Objetivo

El objetivo principal de este proyecto es desarrollar un compilador que transforme clases Java en código SQL para la creación de bases de datos relacionales, automatizando la generación de esquemas y facilitando la integración entre la programación orientada a objetos y el diseño de bases de datos.

## Alcances

1. Reconocer clases Java con atributos definidos y convertirlas en tablas SQL.
2. Mapear tipos de datos de Java (como int, String, float) a tipos de datos SQL equivalentes (como INT, VARCHAR, FLOAT).
3. Generar sentencias CREATE TABLE con nombres de tablas basados en los nombres de las clases y columnas basadas en los atributos de las clases.
4. Manejar atributos con modificadores de acceso (“private”, “public”) y diferentes tipos de datos primitivos y de texto.
5. Diseñar una arquitectura extensible para soportar futuras mejoras, como la generación de claves primarias, relaciones entre tablas (claves foráneas) y restricciones adicionales (como NOT NULL o DEFAULT).

## Ingeniería del proyecto

1. **Metodología**

Las fases principales de un compilador son:

* + **Análisis léxico (scanner)**:  
    Esta fase convierte el código fuente en una secuencia de tokens, que son las unidades léxicas del lenguaje (palabras clave, identificadores, operadores, etc.). El análisis léxico elimina espacios en blanco, comentarios, y verifica la validez léxica de los elementos.
  + **Análisis sintáctico (parser)**:  
    Aquí se analiza la estructura gramatical de los tokens generados en la fase léxica, verificando que sigan las reglas del lenguaje. Se genera un árbol sintáctico (AST, Abstract Syntax Tree) que representa jerárquicamente el código.
  + **Análisis semántico**:  
    Esta fase verifica el significado del código. Comprueba tipos de datos, declaraciones previas de variables, y coerción de tipos. También puede realizar análisis de alcance y detectar errores semánticos.
  + **Generación de código intermedio**:  
    Convierte el árbol sintáctico en una representación intermedia que sea más cercana al código máquina pero independiente de la arquitectura específica. Este código es más fácil de optimizar.

1. **Notaciones**
   * Autómatas Finitos .- Un **Autómata Finito** es un modelo matemático usado para procesar lenguajes regulares, representando cadenas válidas mediante un conjunto de estados, transiciones, un estado inicial y estados finales. Se utiliza principalmente en el análisis léxico de compiladores para identificar tokens, como palabras clave o números. Puede ser determinista (DFA), con transiciones únicas, o no determinista (NFA), con múltiples opciones o transiciones vacías, y opera aceptando o rechazando cadenas según las reglas definidas.
   * Diagrama Sintactico.- Un **diagrama sintáctico** es una representación gráfica de la gramática de un lenguaje, utilizada en la fase de análisis sintáctico de un compilador. Muestra cómo se estructuran las construcciones válidas del lenguaje mediante nodos y conexiones que representan reglas gramaticales. Cada nodo corresponde a un elemento como palabras clave, operadores o expresiones, mientras que las flechas indican el flujo entre ellos. Se usa para visualizar y validar la estructura de código fuente de manera clara y precisa.
   * Árbol Sintactico.- Un **árbol sintáctico** es una representación jerárquica de la estructura gramatical de una cadena de entrada, utilizada en la fase de análisis sintáctico de un compilador. Cada nodo del árbol representa una regla de la gramática aplicada, y las hojas representan los símbolos terminales (como palabras clave, operadores o literales). El árbol muestra cómo se descompone una expresión o sentencia en sus componentes sintácticos, reflejando la jerarquía y las relaciones entre ellos, como en un operador con mayor precedencia sobre otros.
   * BNF.- La **gramática formal en BNF (Backus-Naur Form)** es una notación para describir la sintaxis de un lenguaje de manera formal. Utiliza un conjunto de reglas de producción para especificar cómo se pueden formar cadenas válidas en un lenguaje a partir de símbolos terminales y no terminales.
   * **EBNF.-** Es una extensión de la notación **BNF** que simplifica y mejora la representación de lenguajes formales mediante la adición de operadores y convenciones adicionales. Estas mejoras permiten describir gramáticas de manera más compacta, expresiva y fácil de leer. Es particularmente útil para definir lenguajes de programación y especificaciones de sintaxis de manera clara y precisa.
2. **Herramientas**
   * Lenguaje de programación: Python
   * IDE: Visual Studio Code

# ANÁLISIS LÉXICO

## Definición del alfabeto

**Palabras Reservadas (relevantes para clases)**: class, static, public, private, protected.

**Símbolos especiales**: {, }, ;.

**Tipos de Datos**: int, float, double, char, boolean, String.

**Identificadores**: Nombres de clases y atributos.

## Definición de expresiones regulares

**Modificador de clase:** static

**clase:** class,

**Modificador de acceso:** (private|public|protected)

**Símbolo { :** {

**Símbolo }:** }

**Fin de sentencia:** ;

**TipoDato:** (int|float|String|double|char|boolean)

**Identificador:** [a-zA-Z\_][a-zA-Z0-9\_]\*

## Implementación del analizador léxico

import re

def analizador\_lex(source):

expresiones = {

"Modificador de clase": r"static",

"clase": r"class",

"Modificador de acceso": r"(private|public|protected)",

"Simbolo {": r"{",

"Simbolo }": r"}",

"Simbolo ,": r",",

"Fin de sentencia": r";",

"TipoDato": r"(int|float|String|double|char|boolean|string)",

"Identificador": r"[a-zA-Z\_][a-zA-Z0-9\_]\*",

}

tokens = []

for linea in source.splitlines():

linea = linea.strip()

if not linea: # Ignore empty lines

continue

# Divide la línea en palabras y analiza cada palabra individualmente

palabras = re.findall(r"[a-zA-Z\_][a-zA-Z0-9\_]\*|[{},;]", linea)

for palabra in palabras:

coincide = False

for tipo, expresion in expresiones.items():

if re.fullmatch(expresion, palabra):

coincide = True

tokens.append((tipo, palabra))

break

if not coincide:

print(f"No hay coincidencias para: {palabra}")

return tokens

## Tabla de símbolos

| Lexema | Token | Expresión regular |
| --- | --- | --- |
| Static | Modificador de clase | static |
| class | Clase | class |
| private,public,protected | Modificador de acceso | (private|public|protected) |
| { | Símbolo { | { |
| } | Símbolo } | } |
| ; | Fin de sentencia | ; |
| int,float,String,double,boolean,char | Tipo de dato | (int|float|String|double|char|boolean|String) |
| a,b,c,d,1a,2d,c3,A,B,..... | Identificador | [a-zA-Z\_][a-zA-Z0-9\_]\*c |

## Autómatas finitos

// modelar al menos 3 autómatas finitos



# ANÁLISIS SINTÁCTICO

## Diseño de la gramática

### Reglas BNF

ClassDeclaration ::= Modificador\_de\_acceso Clase Identificador Simbolo\_{ ClassBody Simbolo\_} |Clase Identificador Simbolo\_{ ClassBody Simbolo\_}

ClassBody ::= VariableDeclaration| VariableDeclaration ClassBody

VariableDeclaration ::= Modificador\_de\_acceso TipoDato Identificador Simbolo\_; | TipoDato Identificador Simbolo\_;

Identificadores ::= Identificador|Identificador,Identificadores

### Reglas EBNF

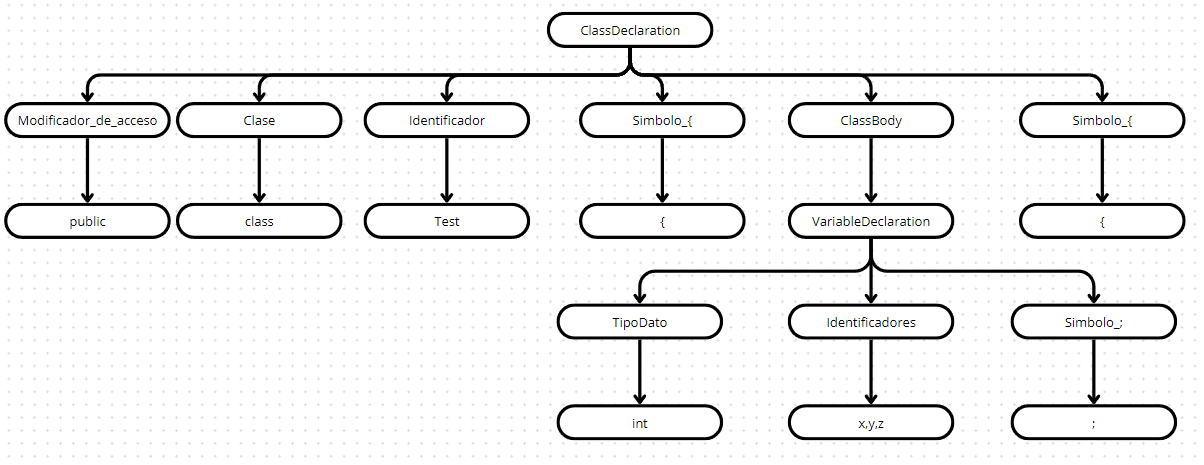
ClassDeclaration ::= [“Modificador\_de\_acceso”] “Clase”   
“Identificador” “Simbolo\_{“ ClassBody “Simbolo\_}”

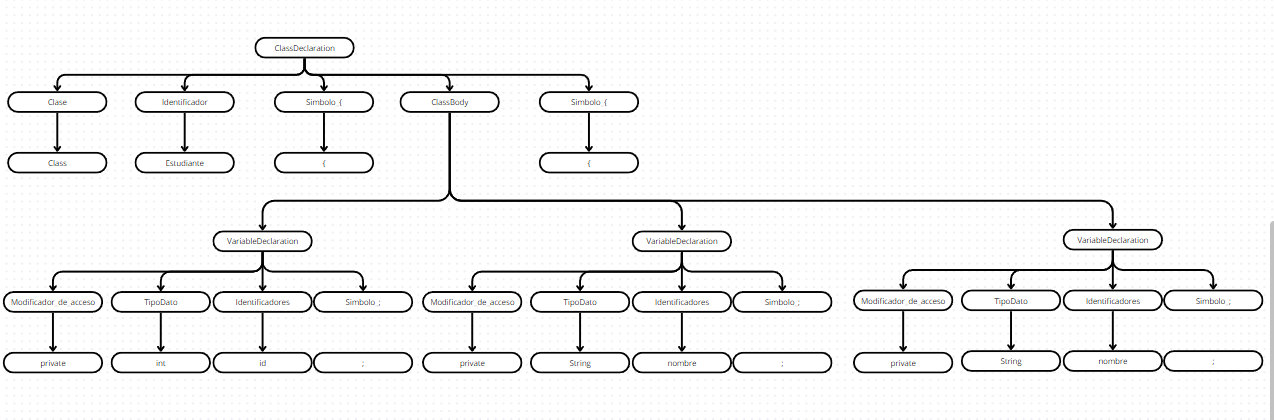
ClassBody ::= {VariableDeclaration}

VariableDeclaration ::= [“Modificador\_de\_acceso”] “TipoDato” Identificadores "Simbolo\_;"

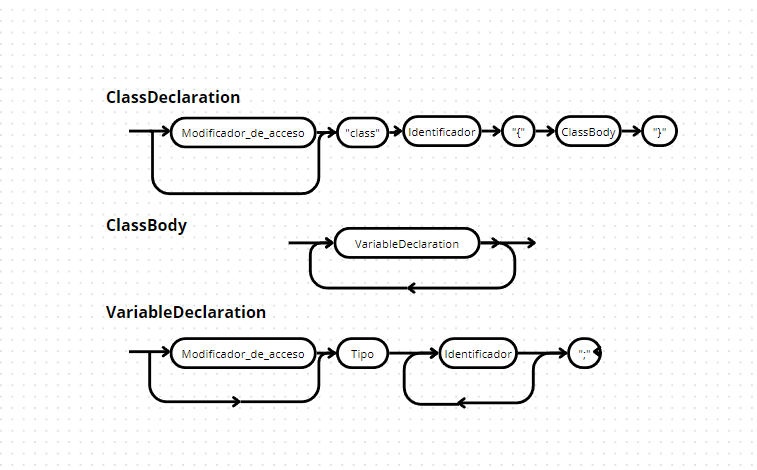
Identificadores ::= {“Identificador”}

## Árbol de análisis sintáctico





## Diagrama de sintaxis



## Implementación del analizador sintáctico

class ASTNode:

def \_\_init\_\_(self, type, value=None, children=None):

self.type = type # Node type (e.g., 'ClassDeclaration', 'VariableDeclaration')

self.value = value # Node value (e.g., class name, variable name)

self.children = children or [] # Child nodes

class Parser:

def \_\_init\_\_(self, tokens):

self.tokens = tokens

self.current\_token\_index = 0

def parse(self):

"""Parses the tokens and returns the root of the AST."""

return self.parse\_ClassDeclaration()

def parse\_ClassDeclaration(self):

"""Parses a class declaration."""

if self.current\_token()[0] == "Modificador de acceso":

self.consume\_token()

if self.current\_token()[0] == "clase":

self.consume\_token()

if self.current\_token()[0] == "Identificador":

class\_name = self.current\_token()[1]

self.consume\_token()

if self.current\_token()[0] == "Simbolo {":

self.consume\_token()

class\_body = self.parse\_ClassBody()

if self.current\_token()[0] == "Simbolo }":

self.consume\_token()

return ASTNode("ClassDeclaration", class\_name, [class\_body])

raise SyntaxError(f"Error Sintactico: Unexpected token {self.current\_token()}")

def parse\_ClassBody(self):

"""Parses the body of a class."""

body\_nodes = []

while self.current\_token()[0] in ("TipoDato", "Modificador de acceso"):

body\_nodes.append(self.parse\_VariableDeclaration())

return ASTNode("ClassBody", children=body\_nodes)

def parse\_VariableDeclaration(self):

"""Parses a variable declaration with support for multiple variables."""

modifier = None

if self.current\_token()[0] == "Modificador de acceso":

modifier = self.current\_token()[1]

self.consume\_token()

# Parse data type

if self.current\_token()[0] == "TipoDato":

data\_type = self.current\_token()[1]

self.consume\_token()

else:

raise SyntaxError(f"Error Sintactico: Unexpected token {self.current\_token()}, expected data type")

# Parse variable names (comma-separated)

variables = []

while self.current\_token()[0] == "Identificador":

var\_name = self.current\_token()[1]

variables.append(var\_name)

self.consume\_token()

if self.current\_token()[0] == "Simbolo ,":

self.consume\_token() # Consume comma and continue parsing

else:

break

# Ensure the declaration ends with a semicolon

if self.current\_token()[0] == "Fin de sentencia":

self.consume\_token()

else:

raise SyntaxError(f"Error Sintactico: Unexpected token {self.current\_token()}, expected ';'")

# Create an AST node for each variable

return ASTNode(

"VariableDeclaration",

value={"type": data\_type, "variables": variables, "modifier": modifier}

)

def current\_token(self):

"""Returns the current token."""

if self.current\_token\_index < len(self.tokens):

return self.tokens[self.current\_token\_index]

return (None, None) # End of tokens

def consume\_token(self):

"""Moves to the next token."""

self.current\_token\_index += 1

# ANÁLISIS SEMÁNTICO

## Árbol de análisis semántico

// 1 arbol semantico

## Tabla de símbolos extendida

//

## Implementación del analizador semántico

# GENERACIÓN DE CÓDIGO

## Implementación (Si aplica)

# CONCLUSIONES Y RECOMENDACIONES

Este proyecto se centra en el análisis sintáctico como etapa clave en el desarrollo de un compilador. Hasta este punto, se han implementado herramientas para interpretar y estructurar clases de Java, lo que permite generar un árbol de sintaxis abstracta (AST) que representa su estructura. Aunque no se aborda el análisis semántico ni la generación completa de código, el proyecto establece una base sólida para transformar el código fuente en un modelo útil para futuras etapas. La conclusión es que el análisis sintáctico proporciona el puente esencial entre la entrada en lenguaje de programación y la salida esperada, en este caso, en forma de SQL.

# BIBLIOGRAFÍA

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley. ISBN: 978-0321486813

Appel, A. W. (2002). *Modern Compiler Implementation in Java*. Cambridge University Press. ISBN: 978-0521820608

Cooper, K., & Torczon, L. (2012). *Engineering a Compiler* (2nd ed.). Morgan Kaufmann. ISBN: 978-0120884780

Grune, D., van Reeuwijk, K., Bal, H. E., Jacobs, C. J. H., & Langendoen, K. (2012). *Modern Compiler Design* (2nd ed.). Springer. ISBN: 978-1461446989

Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation*. Morgan Kaufmann. ISBN: 978-1558603202

Fischer, C. N., & LeBlanc, R. J. (2009). *Crafting a Compiler* (3rd ed.). Pearson. ISBN: 978-0136067054

# ANEXOS

**Anexo 1. URL del código en Gitlab o Github**

// Código fuente del compilador

// Ejemplos de programas en el nuevo lenguaje

// Diagramas adicionales

# BIBLIOGRAFIA

//Colocar la lista de libros y las páginas web que se han consultado, en formato APA

# ANEXOS

**Fotos de su Tablero Kamban (tablero físico)**

**Capturas de pantalla de algunas historias de usuario en Trello y el tablero en Trello (adjuntar URL de Trello)**

**Captura de pantalla de los commits y las ramas de GIT (adjuntar la URL de GITHUB)**