



Proyecto integrador

Proyecto

Proyecto Integrador

Equipo: **Nolasco Ramírez Heber_ 2203330153**

Gonzalez Cavazos Erick Alan

Hernandez Gonzalez Luis Angel

Cruz Bonifacio Luis Fernando

Profesor: **Muñoz Quintero Dante Adolfo**

Asignatura: **PROGRAMACION DE SISTEMAS DE BASE II**

9no. Semestre – Grupo “G”

2024-3

Contenido

Introduccion	4
Link Github:	4
🎯 Objetivos Principales	4
🏗 Estructura del Proyecto	4
📝 Funcionamiento General.....	5
🤖 API de Hugging Face: Análisis Inteligente	5
Desarrollo del proyecto.....	8
- `lexer.flex` — Definición del analizador léxico (JFlex)	8
Codigo:.....	8
- `Main.java` — Clase principal del proyecto.....	11
Codigo:.....	11
- `AnalizadorSemantico.java` — Analizador semántico	14
Codigo:.....	14
- `Token.java` - Organiza y estructura cada token reconocido.....	19
Codigo:.....	19
- `CódigoIntermedio.java` — Clase para instrucciones de código intermedio.....	20
Codigo:.....	20
- `GeneradorIntermedio.java` — Generador de código intermedio.....	22
Codigo:.....	22
- `callapi.java` — Llamada a la API de Hugging Face desde Java.....	27
Codigo:.....	27
- `entrada.txt` — Archivo de entrada de ejemplo	30
Codigo:.....	30
- `salida.txt` — Archivo de tokens generado por el leer	31
Codigo:.....	31
Codigos Space (Hugging Face):.....	34
Codigo:.....	34
- `requirements.txt` — Todas las bibliotecas que se tienen que importar.	36
Codigo:.....	36
Pruebas y resultados	37
Space Hugging Face	37
Link: Proyecto - a Hugging Face Space by HBAB	37

Interfaz.....	37
Ejemplo 1. Error (Ingresando código no compatible)	37
Ejemplo 2. Sin errores.....	38
Ejecucion desde java (Usando la api, con analizador léxico, semántico y generador de código intermedio).....	39
Ejemplo 1.	39
Ejemplo 1. Sin errores.....	39
Ejemplo 1. Con errores.....	41
Ejemplo 1. Errores	42
Conclusión.....	43

Introducción

Este proyecto simula las primeras etapas de un compilador utilizando Java. Mejora y optimiza el analizador léxico creado en la materia Sistemas Base I, incorporando análisis semántico, generación de código intermedio, análisis inteligente asistido por IA mediante Hugging Face y la capacidad de leer scripts de Python.

Link Github:

[H3B3RR](#)

🎯 Objetivos Principales

- Analizar código fuente y descomponerlo en **tokens** utilizando **JFlex**.
- Validar estructuras semánticas como **declaraciones, expresiones, estructuras de control y llamadas a funciones**.
- Generar **código intermedio** (three-address code) representado como instrucciones.
- Integrar una **API de Hugging Face** para realizar análisis asistido por modelos de lenguaje, útil para detectar errores o sugerencias.

🧾 Estructura del Proyecto

Archivo/Clase	Descripción
lexer.flex	Define las reglas del analizador léxico mediante expresiones regulares para JFlex.
Main.java	Clase principal. Ejecuta el lexer y escribe los tokens en salida.txt.
Token.java	Clase que representa cada token: tipo, valor, y posición. Es fundamental para el análisis semántico.

Archivo/Clase	Descripción
AnalizadorSemantico.java	Usa la lista de tokens para validar estructuras semánticas .
GeneradorIntermedio.java	Genera instrucciones de código intermedio basadas en los análisis previos.
CodigoIntermedio.java	Representa cada instrucción en código intermedio (por ejemplo, en formato de tres direcciones).
callapi.java	Llama a la API de Hugging Face, enviando código y recibiendo análisis inteligente.
entrada.txt	Archivo de entrada con código fuente a analizar (Scripts de Python)
salida.txt	Archivo de salida generado automáticamente con los tokens reconocidos.

Funcionamiento General

1. El archivo entrada.txt contiene el código fuente en lenguaje definido.
2. lexer.flex analiza el archivo y genera los **tokens**.
3. Token.java organiza y estructura cada token reconocido.
4. AnalizadorSemantico.java valida la semántica usando esos tokens.
5. GeneradorIntermedio.java produce código intermedio (tipo three-address code).
6. callapi.java (opcional) envía el código a una API de Hugging Face para recibir un análisis semántico asistido por inteligencia artificial.

API de Hugging Face: Análisis Inteligente

Puede visitar el link.

API: [Proyecto - a Hugging Face Space by HBAB](#)

El proyecto incluye una **API propia basada en Hugging Face y Gradio** que permite el análisis **sintáctico y semántico** de código Python.

El modelo usado inicialmente es Salesforce/codet5-base, diseñado específicamente para tareas como resumen, generación y traducción de código fuente. Este modelo fue preentrenado principalmente con código Python, Java, JavaScript, y otros lenguajes populares.

Sin embargo, en esta implementación, el comportamiento está **orientado específicamente al análisis de código en Python**, por cómo se define el *prompt* enviado al modelo:

Dependencias necesarias

Instálalas en tu entorno Python con:

```
pip install gradio transformers torch pyflakes
```

Funcionamiento de la API

1. Análisis de Sintaxis

Utiliza pyflakes para detectar errores sintácticos en el código.

2. Análisis Semántico

Emplea el modelo **Salesforce/codet5-base** de Hugging Face para identificar errores lógicos o semánticos y dar recomendaciones.

3. Interfaz Gradio

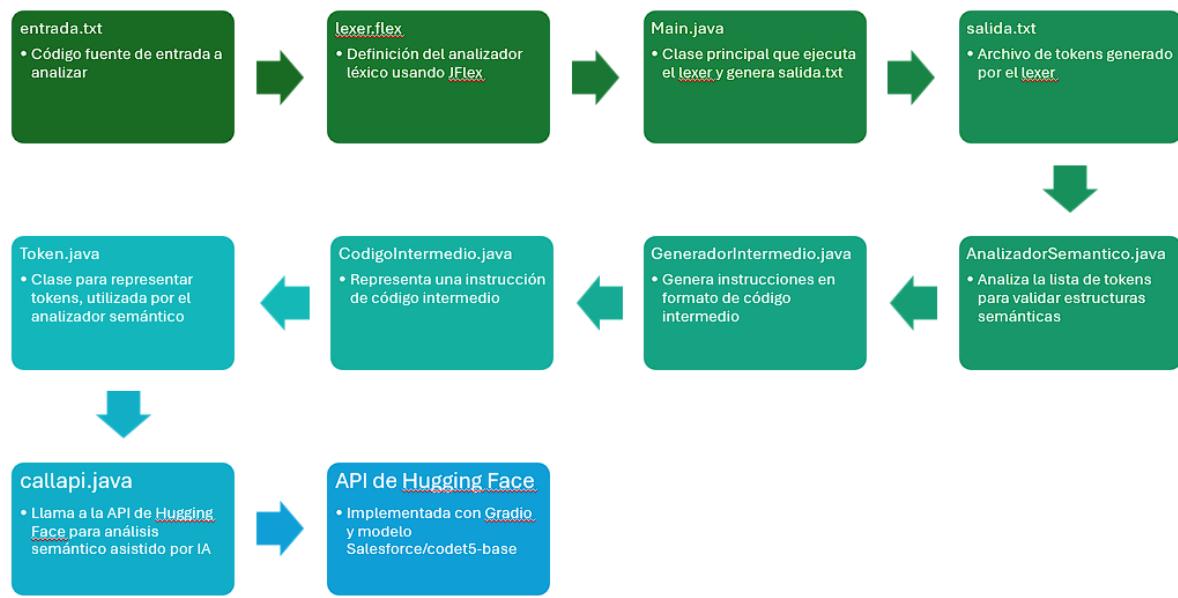
Permite al usuario interactuar gráficamente pegando su código Python y recibiendo análisis en tiempo real.

Estructura del Space en Hugging Face

- **Archivo app.py:**

- Archivo requirements.txt:

Diagrama de uso



Desarrollo del proyecto.

- `lexer.flex` — Definición del analizador léxico (JFlex)

El archivo [lexer.flex](#) define el analizador léxico del proyecto utilizando JFlex. Su función principal es leer el código fuente de entrada y convertirlo en una secuencia de tokens, que serán utilizados por las siguientes etapas del compilador, como el análisis semántico y la generación de código intermedio.

El lexer reconoce palabras clave como if, else, while, for, function, return, class, interface, import y package, y las etiqueta como PALABRA_CLAVE. También reconoce identificadores (nombres de variables y funciones) y las etiqueta como IDENTIFICADOR. Los números se etiquetan como NÚMERO, las cadenas de texto como LITERAL, los operadores como OPERADOR y los delimitadores como DELIMITADOR. Los espacios en blanco y saltos de línea se ignoran. Cualquier otro símbolo no reconocido se etiqueta como TEXTO NO RECONOCIDO.

Por cada token reconocido, el lexer escribe una línea en el archivo de salida con el formato: TIPO_TOKEN: lexema. Por ejemplo: IDENTIFICADOR: num1, OPERADOR: =, NÚMERO: 5, DELIMITADOR: ;. El archivo de salida generado por este lexer es la entrada para el análisis semántico y la generación de código intermedio. Esto permite separar claramente las etapas del compilador y facilita la detección de errores léxicos.

Código:

```
import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.Reader;

%%

%public
%class Lexer
%unicode
%standalone
```

```

%{

private BufferedWriter writer;

public Lexer(Reader in, BufferedWriter writer){
    this.zzReader = in;
    this.writer = writer;
}

public boolean isEOF() {
    return zzAtEOF;
}

private void escribeToken(String lexema, String token) {
    try {
        writer.write(token + ": " + lexema + "\n");
        writer.flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
%}

DIGITO    =[0-9]
LETRA     =[a-zA-Z_]
IDENTIFICADOR = {LETRA}({LETRA}|{DIGITO})*
LITERAL   = "\\".*?\\\"|[^\"]*\""
OPERADOR   = ">>>|>>|<<|++|-+|=|-*=|=/=|/=|==|!=|<=|>=|&&||||+|-|*|/|%""
DELIMITADOR = ";"|","|"( )|{|}|[ ]|.|":"

```

```
ESPACIO    = [ \t\n\r]+  
  
%%  
  
"if"|"else"|"while"|"for"|"function"|"return"|"class"|"interface"|"import"|"package" {  
    escribeToken(yytext(), "PALABRA_CLAVE"); }  
  
{IDENTIFICADOR}           { escribeToken(yytext(), "IDENTIFICADOR"); }  
  
{DIGITO}+                 { escribeToken(yytext(), "NÚMERO"); }  
  
{LITERAL}                  { escribeToken(yytext(), "LITERAL"); }  
  
{OPERADOR}                 { escribeToken(yytext(), "OPERADOR"); }  
  
{DELIMITADOR}              { escribeToken(yytext(), "DELIMITADOR"); }  
  
{ESPACIO}                  { /* Ignorar espacios */ }  
.                         { escribeToken(yytext(), "TEXTO NO RECONOCIDO"); }
```

- `Main.java` — Clase principal del proyecto

El archivo [Main.java](#) es la clase principal del proyecto y coordina todas las etapas del análisis de código fuente. Su funcionamiento es el siguiente:

1. Recibe dos argumentos al ejecutarse: el archivo de entrada (código fuente a analizar) y el archivo de salida (donde se guardarán los tokens generados por el lexer).
2. Realiza el análisis léxico utilizando la clase Lexer, que lee el archivo de entrada y escribe los tokens reconocidos en el archivo de salida. Al finalizar, muestra un mensaje indicando que el análisis léxico se completó.
3. Ejecuta el análisis semántico usando la clase AnalizadorSemantico, que lee el archivo de tokens generado y verifica errores semánticos, como variables no declaradas o tipos incorrectos. Muestra un mensaje al completar esta etapa.
4. Genera el código intermedio llamando a la clase GeneradorIntermedio, que procesa el archivo de tokens y produce instrucciones en formato de tres direcciones, mostrando el resultado en consola.
5. Llama a la API de Hugging Face utilizando la clase callapi. Para esto, lee nuevamente el archivo de entrada, envía el código fuente a la API y muestra en consola el análisis de sintaxis y el análisis lógico devueltos por la API.
6. Si ocurre algún error de entrada/salida durante cualquiera de las etapas, el programa lo muestra en consola.

Código:

```
import java.io.*;

public class Main {
    public static void main(String[] args) throws Exception {
        if (args.length < 2) {
            System.out.println("Uso: java Main <archivo_entrada>
<archivo_salida>"); // Mensaje de uso
            return;
        }

        String archivoEntrada = args[0];
        String archivoSalida = args[1];

        try (
            BufferedReader reader = new BufferedReader(new
FileReader(archivoEntrada));
            BufferedWriter writer = new BufferedWriter(new
FileWriter(archivoSalida))
        ) {
            Lexer lexer = new Lexer(reader, writer);
        }
    }
}
```

```

        while (true) {
            lexer.yylex();
            if (lexer.isEOF()) break;
        }
        System.out.println("Análisis léxico completado. Tokens guardados
en " + archivoSalida); // Léxico
        System.out.println(); // Espacio

        // Analizador semántico
        AnalizadorSemantico analizadorSemantico = new
AnalizadorSemantico();
        analizadorSemantico.analizar(archivoSalida);
        System.out.println("Análisis semántico completado."); //
Semántico
        System.out.println(); // Espacio

        // Generación de código intermedio
        System.out.println("Generación de código intermedio:");
        GeneradorIntermedio.generarDesdeArchivo(archivoSalida);
        System.out.println(); // Espacio

        // Llamada a la API de Hugging Face
        System.out.println("Hugging Face API llamada. \n Dice: \n"); //
Semántico

        StringBuilder codigoFuente = new StringBuilder();
        try (BufferedReader br = new BufferedReader(new
FileReader(archivoEntrada))) {
            String linea;
            while ((linea = br.readLine()) != null) {
                codigoFuente.append(linea).append("\n");
            }
        }
        try {
            String[] respuesta =
callapi.llamarApiHuggingFace(codigoFuente.toString());
            System.out.println("❖ Sintaxis:\n" + respuesta[0]); //
Sintaxis
            System.out.println(); // Espacio
            System.out.println("❖ Análisis lógico:\n" + respuesta[1]);
// Hugging Face - análisis lógico
            System.out.println(); // Espacio
        } catch (IOException e) {
            System.out.println("Error al llamar a la API de Hugging
Face:"); // Error en Hugging Face

```

```
        e.printStackTrace();
    }

} catch (IOException e) {
    e.printStackTrace();
}
}
```

- `AnalizadorSemantico.java` — Analizador semántico

El archivo [AnalizadorSemantico.java](#) implementa el analizador semántico del proyecto. Su función principal es leer el archivo de tokens generado por el analizador léxico y verificar errores semánticos en el código fuente.

El analizador mantiene una tabla de símbolos para registrar las variables declaradas y sus tipos, así como un conjunto de funciones y una lista de identificadores que deben ser ignorados (por ejemplo, "print", "range", "input", "len").

Durante el análisis, el programa recorre la lista de tokens y realiza las siguientes tareas principales:

- Detecta declaraciones de variables (por ejemplo, `a = 5`) y las agrega a la tabla de símbolos, verificando el tipo de dato asignado.
- Detecta declaraciones de funciones, registra sus nombres y parámetros, y analiza el cuerpo de la función para verificar que todas las variables utilizadas estén declaradas o sean parámetros.
- Verifica el uso de variables fuera de funciones, mostrando un error si se utiliza una variable no declarada.
- Muestra mensajes de error semántico en consola cuando detecta variables no declaradas o tipos incorrectos.

El archivo también incluye una clase interna `Token` para representar cada token con su tipo y valor, y un método `main` que permite ejecutar el analizador semántico desde la línea de comandos, recibiendo como argumento el archivo de tokens a analizar.

En resumen, [AnalizadorSemantico.java](#) ayuda a detectar errores de significado en el código fuente, como el uso de variables no declaradas o asignaciones de tipos incorrectos, antes de la generación de código intermedio.

Código:

```
import java.io.*;
import java.util.*;

public class AnalizadorSemantico {
    private Map<String, String> tablaSimbolos = new HashMap<>();
    private Set<String> funciones = new HashSet<>();
    private Set<String> ignorar = new HashSet<>(Arrays.asList("print",
"range", "input", "len"));

    public AnalizadorSemantico() {
        // Constructor vacío
    }
}
```

```
public void analizar(String archivoTokens) throws IOException {
    List<Token> tokens = new ArrayList<>();
    BufferedReader br = new BufferedReader(new
FileReader(archivoTokens));
    String linea;

    while ((linea = br.readLine()) != null) {
        String[] partes = linea.split(": ");
        if (partes.length == 2) {
            tokens.add(new Token(partes[0], partes[1]));
        }
    }
    br.close();

    for (int i = 0; i < tokens.size(); i++) {
        Token token = tokens.get(i);

        // Declaración de variable (asignación)
        if (token.getTipo().equals("IDENTIFICADOR")) {
            if (i + 1 < tokens.size() && tokens.get(i +
1).getValor().equals("=")) {
                Token nombreToken = token;
                Token valorToken = tokens.get(i + 2);
                String tipo = "int";
                if (valorToken.getTipo().equals("CADENA")) {
                    tipo = "string";
                } else if (valorToken.getTipo().equals("NUMERO")) {
                    tipo = "int";
                }
                tablaSimbolos.put(nombreToken.getValor(), tipo);
            }
        }

        // Declaración de función
        if (token.getTipo().equals("PALABRA_CLAVE") &&
token.getValor().equals("def")) {
            if (i + 1 < tokens.size()) {
                Token nombreFuncion = tokens.get(i + 1);
                funciones.add(nombreFuncion.getValor());
                // Agregar parámetros a tabla de símbolos temporal
                Set<String> parametros = new HashSet<>();
                int j = i + 3; // Salta "def", nombre, "("
                while (j < tokens.size() &&
!tokens.get(j).getValor().equals(")")) {

```

```

                if (tokens.get(j).getTipo().equals("IDENTIFICADOR"))
{
    parametros.add(tokens.get(j).getValor());
}
j++;
}
// Analizar cuerpo de la función (por indentación)
int cuerpoInicio = j + 2; // Salta ")", ":" 
int cuerpoFin = cuerpoInicio;
int indentNivel = 1;
// Tabla de símbolos local para la función
Set<String> simbolosLocales = new HashSet<>(parametros);
while (cuerpoFin < tokens.size() && indentNivel > 0) {
    Token t = tokens.get(cuerpoFin);
    if (t.getTipo().equals("INDENT")) indentNivel++;
    if (t.getTipo().equals("DEDENT")) indentNivel--;
    cuerpoFin++;
}
// Analizar cuerpo y registrar variables locales
for (int k = cuerpoInicio; k < cuerpoFin - 1; k++) {
    Token t = tokens.get(k);
    // Registrar variables locales en asignación
    if (t.getTipo().equals("IDENTIFICADOR")) {
        if (k + 1 < tokens.size() && tokens.get(k +
1).getValor().equals("=")) {
            simbolosLocales.add(t.getValor());
        }
    }
}
// Analizar cuerpo y reportar solo si no es local,
global, función o ignorada
for (int k = cuerpoInicio; k < cuerpoFin - 1; k++) {
    Token t = tokens.get(k);
    if (t.getTipo().equals("IDENTIFICADOR")) {
        String nombre = t.getValor();
        if (!simbolosLocales.contains(nombre) &&
!tablaSimbolos.containsKey(nombre) && !funciones.contains(nombre) &&
!ignorar.contains(nombre)) {
            System.out.println("Error semántico:
variable '" + nombre + "' no declarada.");
        }
    }
}
i = cuerpoFin - 1; // Saltar cuerpo de la función
}

```

```
    }

        // Uso de variable fuera de función
        if (token.getTipo().equals("IDENTIFICADOR")) {
            String nombre = token.getValor();
            if (!tablaSimbolos.containsKey(nombre) &&
!funciones.containsKey(nombre) && !ignorar.containsKey(nombre)) {
                // Evitar marcar parámetros de función como error fuera
de función
                boolean esParametro = false;
                if (i > 0 && tokens.get(i - 1).getValor().equals("("))
esParametro = true;
                if (!esParametro)
                    System.out.println("Error semántico: variable '" +
nombre + "' no declarada.");
            }
        }
    }

// Clase interna Token
public static class Token {
    private String tipo;
    private String valor;

    public Token(String tipo, String valor) {
        this.tipo = tipo;
        this.valor = valor;
    }

    public String getTipo() {
        return tipo;
    }

    public void setTipo(String tipo) {
        this.tipo = tipo;
    }

    public String getValor() {
        return valor;
    }

    public void setValor(String valor) {
        this.valor = valor;
    }
}
```

```
}

// Método main para ejecutar el analizador
public static void main(String[] args) {
    if (args.length != 1) {
        System.out.println("Uso: java AnalizadorSemantico
<archivo_tokens>");
        return;
    }
    AnalizadorSemantico analizador = new AnalizadorSemantico();
    try {
        analizador.analizar(args[0]);
    } catch (IOException e) {
        System.out.println("Error al leer el archivo: " +
e.getMessage());
    }
}
```

- `Token.java` - Organiza y estructura cada token reconocido.

La clase [Token.java](#) representa un token individual generado durante el análisis léxico del código fuente.

Cada objeto Token almacena dos atributos principales:

- tipo: indica el tipo de token (por ejemplo, "IDENTIFICADOR", "NÚMERO", "OPERADOR", etc.).
- valor: contiene el texto o lexema específico reconocido en el código fuente.

La clase incluye un constructor para inicializar ambos atributos y métodos getTipo() y getValor() para acceder a ellos.

Esta clase es utilizada por el analizador semántico y otras etapas del compilador para manipular y analizar la secuencia de tokens producida por el lexer.

Código:

```
public class Token {  
    private String tipo;  
    private String valor;  
  
    public Token(String tipo, String valor) {  
        this.tipo = tipo;  
        this.valor = valor;  
    }  
  
    public String getTipo() {  
        return tipo;  
    }  
  
    public String getValor() {  
        return valor;  
    }  
}
```

- `CodigoIntermedio.java` — Clase para instrucciones de código intermedio

La clase [CodigoIntermedio.java](#) representa una instrucción de código intermedio en formato de tres direcciones.

Esta clase se utiliza para almacenar y mostrar operaciones intermedias generadas durante el análisis del código fuente, facilitando la optimización y posterior traducción a código máquina o ensamblador.

Cada objeto de esta clase contiene:

- operador: el operador de la instrucción (por ejemplo, "+", "-", "=", "CALL", etc.).
- arg1: el primer argumento o variable involucrada en la operación.
- arg2: el segundo argumento (puede ser null o vacío si la operación es unaria o de asignación simple).
- resultado: la variable o temporal donde se almacena el resultado de la operación.

El método `toString()` devuelve una representación en texto de la instrucción, siguiendo el formato típico de código de tres direcciones, por ejemplo: `t1 = a + b` o, para operaciones unarias o asignaciones simples: `a = 5`

Código:

```
public class CodigoIntermedio {  
    public String operador;  
    public String arg1;  
    public String arg2;  
    public String resultado;  
  
    public CodigoIntermedio(String operador, String arg1, String arg2,  
String resultado) {  
        this.operador = operador;  
        this.arg1 = arg1;  
        this.arg2 = arg2;  
        this.resultado = resultado;  
    }  
  
    @Override  
    public String toString() {  
        if (arg2 == null || arg2.isEmpty()) {  
            return resultado + " = " + operador + " " + arg1;  
        } else {  
            return resultado + " = " + operador + " " + arg1 + " " + arg2;  
        }  
    }  
}
```

```
        }
        return resultado + " = " + arg1 + " " + operador + " " + arg2;
    }
}
```

- `GeneradorIntermedio.java` — Generador de código intermedio

El archivo [GeneradorIntermedio.java](#) implementa el generador de código intermedio para el proyecto. Su función principal es leer el archivo de tokens generado por el analizador léxico y transformar la secuencia de tokens en instrucciones de código intermedio en formato de tres direcciones.

El proceso funciona de la siguiente manera:

- Lee el archivo de tokens línea por línea y almacena cada token como un arreglo de tipo y valor.
- Recorre la lista de tokens y, dependiendo de los patrones detectados, genera instrucciones de código intermedio utilizando la clase `CodigoIntermedio`.
- Detecta asignaciones simples y operaciones aritméticas, generando instrucciones para cada caso.
- Reconoce llamadas a funciones, instrucciones de retorno, condicionales (if), bucles while y definiciones de funciones, generando las instrucciones correspondientes y etiquetas para el control de flujo.
- Utiliza variables temporales (`t1`, `t2`, etc.) y etiquetas (`L1`, `L2`, etc.) para representar resultados intermedios y saltos en el flujo del programa.
- Al finalizar el análisis, imprime en consola todas las instrucciones de código intermedio generadas.

Este archivo permite transformar el código fuente en una representación intermedia más sencilla de analizar y optimizar, facilitando la posterior traducción a código máquina o a otro lenguaje.

Código:

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

public class GeneradorIntermedio {
    private static int tempCount = 1;
    private static List<CodigoIntermedio> instrucciones = new ArrayList<>();

    private static String nuevoTemporal() {
        return "t" + (tempCount++);
    }
}
```

```

    public static void generarDesdeArchivo(String archivoTokens) throws
IOException {
    tempCount = 1;
    instrucciones = new ArrayList<>();

    // Lee los tokens generados por el lexer
    List<String[]> tokens = new ArrayList<>();
    try (BufferedReader br = new BufferedReader(new
FileReader(archivoTokens))) {
        String linea;
        while ((linea = br.readLine()) != null) {
            String[] partes = linea.split(": ");
            if (partes.length == 2) {
                tokens.add(partes);
            }
        }
    }

    for (int i = 0; i < tokens.size(); i++) {
        String tipo = tokens.get(i)[0];
        String valor = tokens.get(i)[1];

        // Asignación: IDENTIFICADOR = EXPRESION
        if (tipo.equals("IDENTIFICADOR") && i + 2 < tokens.size()
            && tokens.get(i + 1)[0].equals("OPERADOR") && tokens.get(i +
1)[1].equals("=")) {
            String var = valor;
            // Asignación directa
            if (tokens.get(i + 2)[0].equals("NÚMERO") || tokens.get(i +
2)[0].equals("LITERAL") || tokens.get(i + 2)[0].equals("IDENTIFICADOR")) {
                instrucciones.add(new CódigoIntermedio("=", tokens.get(i +
2)[1], null, var));
                i += 2;
            }
            // Operación aritmética simple: a = b OP c
            else if ((tokens.get(i + 2)[0].equals("IDENTIFICADOR") ||
tokens.get(i + 2)[0].equals("NÚMERO"))
                && i + 4 < tokens.size()
                && tokens.get(i + 3)[0].equals("OPERADOR")
                && (tokens.get(i + 3)[1].matches("\\"+-|\\*|\\*|//|/|%"-
|=|\\*=|/=|//=|%="|\\^|\\||&&|&|~|<<|>>|\\."))
                    && (tokens.get(i + 4)[0].equals("IDENTIFICADOR") ||
tokens.get(i + 4)[0].equals("NÚMERO")))) {
                String temp = nuevoTemporal();

```

```

        instrucciones.add(new CódigoIntermedio(tokens.get(i + 3)[1],
tokens.get(i + 2)[1], tokens.get(i + 4)[1], temp));
        instrucciones.add(new CódigoIntermedio("=", temp, null,
var));
        i += 4;
    }
}
// Llamada a función: IDENTIFICADOR ( PARAMETROS )
else if (tipo.equals("IDENTIFICADOR") && i + 1 < tokens.size()
&& tokens.get(i + 1)[0].equals("DELIMITADOR") &&
tokens.get(i + 1)[1].equals("(")) {
    String funcion = valor;
    List<String> parametros = new ArrayList<>();
    i += 2; // Saltar al '('
    while (i < tokens.size() &&
!(tokens.get(i)[0].equals("DELIMITADOR") && tokens.get(i)[1].equals(")"))) {
        if (tokens.get(i)[0].equals("IDENTIFICADOR") ||
tokens.get(i)[0].equals("NÚMERO") || tokens.get(i)[0].equals("LITERAL")) {
            parametros.add(tokens.get(i)[1]);
        }
        i++;
    }
    instrucciones.add(new CódigoIntermedio("CALL", funcion,
String.join(", ", parametros), null));
}
// Return: return EXPRESION
else if (tipo.equals("PALABRA_CLAVE") && valor.equals("return"))
{
    if (i + 1 < tokens.size() && (tokens.get(i +
1)[0].equals("IDENTIFICADOR") || tokens.get(i + 1)[0].equals("NÚMERO") || tokens.get(i + 1)[0].equals("LITERAL")))
    {
        instrucciones.add(new CódigoIntermedio("RETURN",
tokens.get(i + 1)[1], null, null));
        i += 1;
    }
}
// If: if ( CONDICION ):
else if (tipo.equals("PALABRA_CLAVE") && valor.equals("if") && i + 1 < tokens.size()
&& tokens.get(i + 1)[0].equals("DELIMITADOR") &&
tokens.get(i + 1)[1].equals("(")) {
    i += 2; // Saltar al '('
    StringBuilder condicion = new StringBuilder();
    while (i < tokens.size() &&
!(tokens.get(i)[0].equals("DELIMITADOR") && tokens.get(i)[1].equals(")"))) {

```

```

        condicion.append(tokens.get(i)[1]).append(" ");
        i++;
    }
    String etiquetaTrue = "L" + tempCount++;
    String etiquetaFalse = "L" + tempCount++;
    instrucciones.add(new CódigoIntermedio("IF",
condicion.toString().trim(), null, etiquetaTrue));
    instrucciones.add(new CódigoIntermedio("GOTO", null, null,
etiquetaFalse));
    instrucciones.add(new CódigoIntermedio("LABEL", null, null,
etiquetaTrue));
    // Aquí podrías procesar instrucciones internas del if si lo
deseas
    instrucciones.add(new CódigoIntermedio("LABEL", null, null,
etiquetaFalse));
}
// While: while ( CONDICION ):
else if (tipo.equals("PALABRA_CLAVE") && valor.equals("while"))
&& i + 1 < tokens.size()
    && tokens.get(i + 1)[0].equals("DELIMITADOR") &&
tokens.get(i + 1)[1].equals("(") {
    String etiquetaInicio = "L" + tempCount++;
    String etiquetaFin = "L" + tempCount++;
    instrucciones.add(new CódigoIntermedio("LABEL", null, null,
etiquetaInicio));
    i += 2;
    StringBuilder condicion = new StringBuilder();
    while (i < tokens.size() &&
!(tokens.get(i)[0].equals("DELIMITADOR") && tokens.get(i)[1].equals("")))
) {
        condicion.append(tokens.get(i)[1]).append(" ");
        i++;
    }
    instrucciones.add(new CódigoIntermedio("IF",
condicion.toString().trim(), null, etiquetaInicio + "_BODY"));
    instrucciones.add(new CódigoIntermedio("GOTO", null, null,
etiquetaFin));
    instrucciones.add(new CódigoIntermedio("LABEL", null, null,
etiquetaInicio + "_BODY"));
    // Procesar cuerpo del while si se desea
    instrucciones.add(new CódigoIntermedio("GOTO", null, null,
etiquetaInicio));
    instrucciones.add(new CódigoIntermedio("LABEL", null, null,
etiquetaFin));
}
// Definición de función: def IDENTIFICADOR ( PARAMS ):
```

```
        else if (tipo.equals("PALABRA_CLAVE") && valor.equals("def")
                  && i + 1 < tokens.size() && tokens.get(i +
1)[0].equals("IDENTIFICADOR")) {
            String nombreFuncion = tokens.get(i + 1)[1];
            instrucciones.add(new CodigoIntermedio("FUNC_BEGIN",
nombreFuncion, null, null));
            // Aquí podrías procesar los parámetros si lo deseas
            // Saltar hasta el final del bloque de la función (si tienes
tokens de indentación)
            // Si no, simplemente continúa
            }
        }

        // Imprime el código intermedio generado
System.out.println("Código intermedio generado:");
for (CodigoIntermedio instr : instrucciones) {
    System.out.println(instr);
}
}
```

- `callapi.java` — Llamada a la API de Hugging Face desde Java

El archivo [callapi.java](#) implementa la conexión y el consumo de una API externa de Hugging Face desde Java. Su objetivo es enviar código fuente como texto a un modelo alojado en Hugging Face y recibir un análisis sintáctico y lógico del mismo.

El funcionamiento es el siguiente:

- El método llamarApiHuggingFace recibe el código fuente como una cadena de texto.
- Realiza una petición POST a la API de Hugging Face, enviando el código en formato JSON. La respuesta de esta petición contiene un EVENT_ID que identifica la solicitud.
- Utiliza el EVENT_ID para realizar una petición GET a la misma API y así obtener el resultado del análisis.
- El resultado se recibe en formato de eventos (Server-Sent Events), por lo que el método extrae el arreglo de datos relevante del texto de respuesta.
- Devuelve un arreglo de cadenas con dos elementos: el primero corresponde al análisis de sintaxis y el segundo al análisis lógico realizado por el modelo de Hugging Face.
- Si ocurre algún error en la comunicación o en el formato de la respuesta, el método lanza una excepción con el mensaje correspondiente.

El archivo también incluye un método main de ejemplo que muestra cómo usar la función llamarApiHuggingFace y cómo imprimir los resultados en consola.

Código:

```
import java.io.*;
import java.net.*;
import java.nio.charset.StandardCharsets;

public class callapi {
    public static void main(String[] args) {
        String codigo = "let num1 = \"10\";\nlet num2 = 5;\nfunction suma(a,\nb) {\n    if (a > b) {\n        return a - b;\n    } else {\n        return\n        b + a;\n    }\n}\nlet resultado = suma(num1, num2);\nconsole.log(resultado);";
        try {
```

```
        String[] respuesta = llamarApiHuggingFace(codigo);
        System.out.println("📝 Sintaxis:\n" + respuesta[0]);
        System.out.println("🧩 Análisis lógico:\n" + respuesta[1]);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static String[] llamarApiHuggingFace(String codigo) throws
IOException {
    // Paso 1: POST para obtener EVENT_ID
    String apiUrl = "https://hbab-
proyecto.hf.space/gradio_api/call/predict";
    String jsonInput = "{\"data\": [" + codigo.replace("\n",
"\n").replace("\"", "\\\"") + "]}";

    HttpURLConnection postConn = (HttpURLConnection) new
URL(apiUrl).openConnection();
    postConn.setRequestMethod("POST");
    postConn.setRequestProperty("Content-Type", "application/json");
    postConn.setDoOutput(true);

    try (OutputStream os = postConn.getOutputStream()) {
        byte[] input = jsonInput.getBytes(StandardCharsets.UTF_8);
        os.write(input, 0, input.length);
    }

    StringBuilder postResponse = new StringBuilder();
    try (BufferedReader br = new BufferedReader(
            new InputStreamReader(postConn.getInputStream(),
StandardCharsets.UTF_8))) {
        String responseLine;
        while ((responseLine = br.readLine()) != null) {
            postResponse.append(responseLine.trim());
        }
    }

    // Extraer EVENT_ID del JSON de respuesta
    String eventId = null;
    String resp = postResponse.toString();
    int idx = resp.indexOf("\"event_id\":\"");
    if (idx != -1) {
        int start = idx + 12;
        int end = resp.indexOf("\"", start);
        eventId = resp.substring(start, end);
    }
}
```

```
        } else {
            throw new IOException("No se pudo obtener EVENT_ID de la
respuesta: " + resp);
        }

        // Paso 2: GET para obtener el resultado
        String getUrl = "https://hbab-
proyecto.hf.space/gradio_api/call/predict/" + eventId;
        HttpURLConnection getConn = (HttpURLConnection) new
URL(getUrl).openConnection();
        getConn.setRequestMethod("GET");

        StringBuilder getResponse = new StringBuilder();
        try (BufferedReader br = new BufferedReader(
                new InputStreamReader(getConn.getInputStream(),
StandardCharsets.UTF_8))) {
            String responseLine;
            while ((responseLine = br.readLine()) != null) {
                getResponse.append(responseLine.trim());
            }
        }

        // Extraer los dos resultados del JSON (muy simple, para ejemplo)
        // Extraer los dos resultados del JSON (adaptado para Server-Sent
Events)
        String result = getResponse.toString();
        int dataIdx = result.indexOf("data: [");
        if (dataIdx != -1) {
            int start = result.indexOf("[", dataIdx);
            int end = result.indexOf("]", start);
            String data = result.substring(start + 1, end);
            String[] parts = data.split("\\\",\\\"");
            // Limpiar comillas y caracteres de escape
            for (int i = 0; i < parts.length; i++) {
                parts[i] = parts[i].replaceAll("^\"|\"$", "").replace("\n",
"\n").replace("\u274c", "X").replace("\u00ed", "í");
            }
            return parts;
        } else {
            throw new IOException("No se pudo extraer el resultado de la
respuesta: " + result);
        }
    }
}
```

- `entrada.txt` — Archivo de entrada de ejemplo

El archivo [entrada.txt](#) contiene el código fuente de ejemplo que será analizado por el proyecto. Este archivo es la entrada principal para el analizador léxico, el analizador semántico y el generador de código intermedio.

En este ejemplo, el archivo define dos variables (num1 y num2), una función llamada suma que recibe dos parámetros y utiliza una estructura condicional if-else para retornar la resta o la suma de los parámetros según la condición. Después, se declara una variable resultado que almacena el resultado de llamar a la función suma con los valores de num1 y num2, y finalmente se imprime el valor de result en consola.

Este archivo puede ser modificado por el usuario para probar diferentes fragmentos de código y observar cómo el sistema analiza, detecta errores y genera el código intermedio correspondiente.

Código:

```
let num1 = "10";
let num2 = 5;

function suma(a, b){
    if (a > b) {
        return a - b;
    } else {
        return b + a;
    }
}

let resultado = suma(num1, num2);
console.log(result);
```

- `salida.txt` — Archivo de tokens generado por el leer

El archivo [salida.txt](#) contiene la secuencia de tokens generada por el analizador léxico a partir del código fuente de entrada. Cada línea representa un token reconocido, indicando su tipo (por ejemplo, IDENTIFICADOR, OPERADOR, NÚMERO, DELIMITADOR, PALABRA_CLAVE, etc.) y el lexema correspondiente extraído del código.

Este archivo es utilizado como entrada para el analizador semántico y el generador de código intermedio. Permite separar claramente la etapa de análisis léxico del resto del proceso de compilación, facilitando la detección de errores léxicos y el análisis posterior del código.

En resumen, [salida.txt](#) es el resultado del análisis léxico y sirve como base para las siguientes etapas del compilador, proporcionando una representación estructurada y simplificada del código fuente original.

Código:

```
IDENTIFICADOR: let
IDENTIFICADOR: num1
OPERADOR: =
TEXTO NO RECONOCIDO: "
NÚMERO: 10
TEXTO NO RECONOCIDO: "
DELIMITADOR: ;
IDENTIFICADOR: let
IDENTIFICADOR: num2
OPERADOR: =
NÚMERO: 5
DELIMITADOR: ;
PALABRA_CLAVE: function
IDENTIFICADOR: suma
DELIMITADOR: (
```

IDENTIFICADOR: a
DELIMITADOR: ,
IDENTIFICADOR: b
DELIMITADOR:)
DELIMITADOR: {
PALABRA_CLAVE: if
DELIMITADOR: (
IDENTIFICADOR: a
OPERADOR: >
IDENTIFICADOR: b
DELIMITADOR:)
DELIMITADOR: {
PALABRA_CLAVE: return
IDENTIFICADOR: a
OPERADOR: -
IDENTIFICADOR: b
DELIMITADOR: ;
DELIMITADOR: }
PALABRA_CLAVE: else
DELIMITADOR: {
PALABRA_CLAVE: return
IDENTIFICADOR: b
OPERADOR: +
IDENTIFICADOR: a
DELIMITADOR: ;
DELIMITADOR: }
DELIMITADOR: }
IDENTIFICADOR: let
IDENTIFICADOR: resultado

```
OPERADOR: =
IDENTIFICADOR: suma
DELIMITADOR: (
IDENTIFICADOR: num1
DELIMITADOR: ,
IDENTIFICADOR: num2
DELIMITADOR: )
DELIMITADOR: ;
IDENTIFICADOR: console
DELIMITADOR: .
IDENTIFICADOR: log
DELIMITADOR: (
IDENTIFICADOR: result
DELIMITADOR: )
DELIMITADOR: ;
```

Codigos Space (Hugging Face):

- `app.py` — App creada en hugging que integra los modelos.

El archivo [app.py](#) implementa una aplicación web sencilla utilizando Gradio y modelos de Hugging Face para analizar código fuente.

Su funcionamiento es el siguiente:

- Importa las librerías necesarias: gradio para la interfaz web, transformers para el modelo de análisis semántico, pyflakes para el análisis de sintaxis, y módulos estándar para manejo de entrada/salida.
- Carga un modelo de Hugging Face (Salesforce/codet5-base) para generación y análisis de código.
- Define la función `analizar_codigo`, que recibe código fuente como texto:
 - Primero, realiza un análisis de sintaxis usando pyflakes y capture cualquier error de sintaxis.
 - Si hay errores de sintaxis, los devuelve como resultado.
 - Si no hay errores, utiliza el modelo de Hugging Face para realizar un análisis semántico del código y devuelve el resultado generado.
- Crea una interfaz web con Gradio, donde el usuario puede ingresar código, y muestra el resultado del análisis en pantalla.
- Finalmente, lanza la aplicación web para que pueda ser utilizada desde el navegador.

Codigo:

```
import gradio as gr
from transformers import pipeline
import pyflakes.api
from io import StringIO
import sys

# Análisis semántico con modelo Hugging Face
model = pipeline("text2text-generation", model="Salesforce/codet5-base")

def analizar_codigo(code):
```

```
# Análisis de sintaxis
buffer = StringIO()
sys.stderr = buffer
pyflakes.api.check(code, "análisis")
errores = buffer.getvalue()
sys.stderr = sys.__stderr__

if errores:
    return f"Errores de sintaxis:\n{errores}"

# Análisis semántico
resultado = model(code, max_length=256, do_sample=False)
return f"Análisis semántico:\n{resultado[0]['generated_text']}"

gr.Interface(fn=analizar_codigo, inputs="text", outputs="text").launch()
```

- `requirements.txt` — Todas las bibliotecas que se tienen que importar.

El archivo [requeriments.txt](#) contiene la lista de dependencias necesarias para ejecutar la aplicación web de análisis de código en Python.

Incluye los siguientes paquetes:

- gradio: para crear la interfaz web interactiva.
- transformers: para utilizar modelos de inteligencia artificial de Hugging Face.
- torch: backend requerido por transformers para el procesamiento de modelos.
- pyflakes: para realizar el análisis de sintaxis del código fuente.

Este archivo permite instalar fácilmente todas las dependencias ejecutando el comando
pip install -r [requeriments.txt](#)

en la terminal, asegurando que el entorno de Python tenga todo lo necesario para ejecutar la aplicación.

Código:

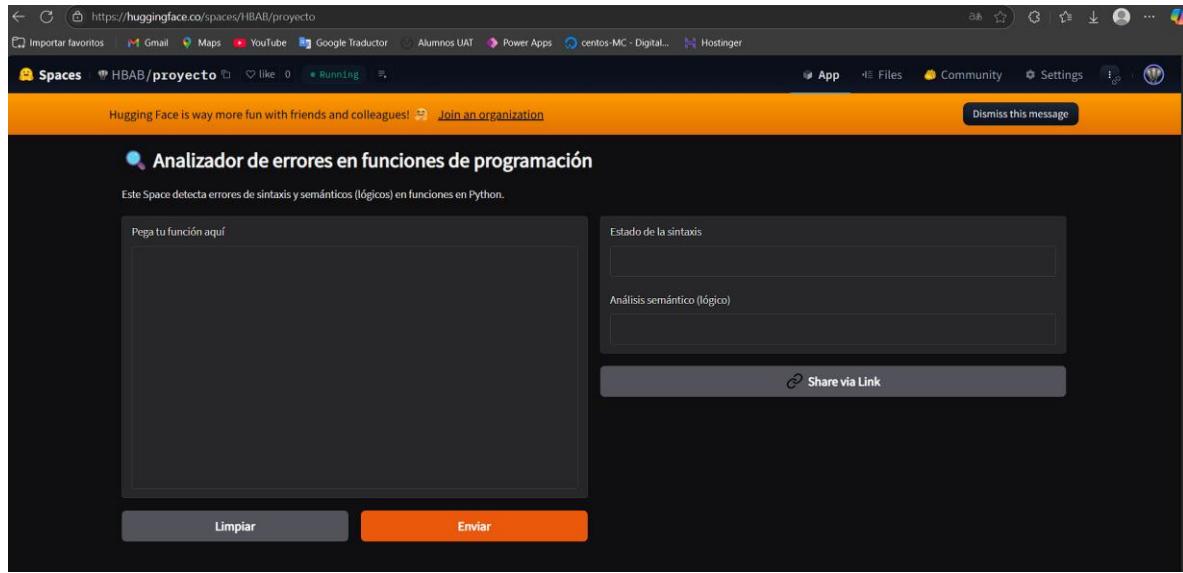
```
gradio
transformers
torch
pyflakes
```

Pruebas y resultados

Space Hugging Face

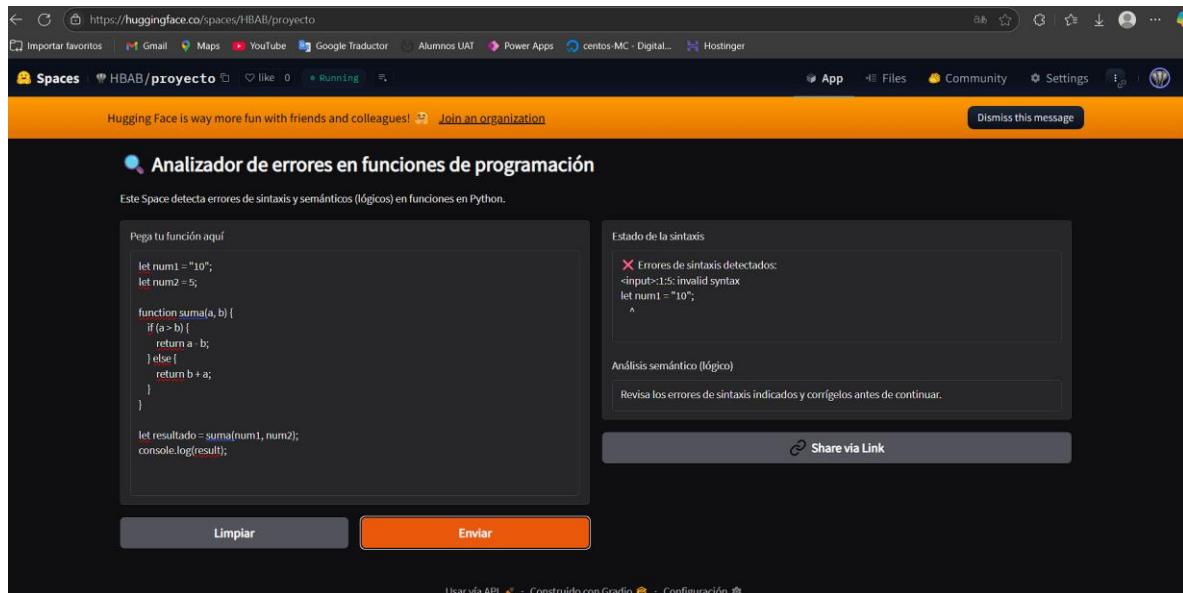
Link: [Proyecto - a Hugging Face Space by HBAB](https://huggingface.co/spaces/HBAB/proyecto)

Interfaz



The screenshot shows the 'Análizador de errores en funciones de programación' (Programming Function Error Analyzer) space. The interface has a dark theme with orange highlights. On the left, there's a text input field labeled 'Pega tu función aquí' (Paste your function here) containing Python code. On the right, there are two output sections: 'Estado de la sintaxis' (Syntax State) and 'Análisis semántico (lógico)' (Semantic Analysis (Logical)). The syntax state section shows an error message: '<input>:15: invalid syntax'. The semantic analysis section says 'Revise los errores de sintaxis indicados y corrígelos antes de continuar.' (Review the indicated syntax errors and fix them before continuing.) At the bottom, there are 'Limpiar' (Clear) and 'Enviar' (Send) buttons.

Ejemplo 1. Error (Ingresando código no compatible)



This screenshot shows the same space after pasting the following Python code into the text input field:

```
let num1 = "10";
let num2 = 5;

function suma(a, b) {
    if (a > b) {
        return a - b;
    } else {
        return b + a;
    }

let resultado = suma(num1, num2);
console.log(resultado);
```

The 'Estado de la sintaxis' section displays an error message: '<input>:15: invalid syntax'. The 'Análisis semántico (lógico)' section contains the instruction 'Revise los errores de sintaxis indicados y corrígelos antes de continuar.' (Review the indicated syntax errors and fix them before continuing.)

Ejemplo 2. Sin errores

The screenshot shows the Hugging Face Spaces interface. In the top navigation bar, there are links for 'Spaces', 'HBAB / proyecto', 'like 0', 'Running 1', and 'Logs'. On the right side of the header are 'App', 'Files', 'Community', 'Settings', and a user icon. A message 'Hugging Face is way more fun with friends and colleagues!' with a 'Join an organization' button is displayed. Below the header, there are two main sections: 'code' and 'output'. The 'code' section contains the following Python code:

```
code
num1 = "10"
num2 = 5

def suma(a,b):
    if a > b:
        return a - b
    else:
        return b + a

resultado = suma(num1,num2)
print(resultado)
```

The 'output' section displays the semantic analysis results:

```
output
Análisis semántico:
resultado=resultado(resultado)resultado(resultado)
```

Below these sections are 'Limpiar' (Clean) and 'Enviar' (Send) buttons. At the bottom of the page, there are links for 'Usar vía API', 'Configuración', and 'Alumnos UAT'.

This screenshot shows the same Hugging Face Spaces interface as the previous one, but with incomplete code. The 'code' section now contains:

```
code
num1 = "10"
num2 = 5
```

The 'output' section shows the beginning of the semantic analysis:

```
output
Análisis semántico:
n unction( num1 , num2){
```

The 'Enviar' (Send) button is visible at the bottom of the code area. The browser's address bar shows the URL <https://huggingface.co/spaces/HBAB/proyecto>.

Ejecucion desde java (Usando la api, con analizador léxico, semántico y generador de código intermedio)

Ejemplo 1.

```
(base2) PS C:\Users\Heber\OneDrive\Universidad\Nolasco Ramirez Heber_10\Programacion Sistemas Base II\Programacion\Unidad_4\ProyectoIntegrado_fin> java Main entrada.txt salida.txt
Análisis léxico completado. Tokens guardados en salida.txt

Error semántico: variable 'num1' no declarada.
Error semántico: variable 'num2' no declarada.
Error semántico: variable 'suma' no declarada.
Error semántico: variable 'a' no declarada.
Error semántico: variable 'b' no declarada.
Error semántico: variable 'a' no declarada.
Error semántico: variable 'b' no declarada.
Error semántico: variable 'a' no declarada.
Error semántico: variable 'b' no declarada.
Error semántico: variable 'a' no declarada.
Error semántico: variable 'b' no declarada.
Error semántico: variable 'a' no declarada.
Error semántico: variable 'b' no declarada.
Error semántico: variable 'resultado' no declarada.
Error semántico: variable 'suma' no declarada.
Error semántico: variable 'num1' no declarada.
Error semántico: variable 'num2' no declarada.
Error semántico: variable 'print' no declarada.
Error semántico: variable 'resultado' no declarada.
Análisis semántico completado.

Generación de código intermedio:
Código intermedio generado:

Hugging Face API llamada.
Dice:

? Sintaxis:
An\u00f3lisis sem\u00f3ntico:
resultado(resultado)resultado(resultado)resultado

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 1 out of bounds for length 1
at Main.main(Main.java:50)
```

Ejemplo 1. Sin errores



```
Entrada.txt X salida.txt app.py 2 requirements.txt lexerflex J Main.java J AnalizadorSemanticoo.java J Token.java J GeneradorIntermedio.java J CódigoIntermedio.java ... Entrada.txt
Unidad_4 > ProyectoIntegrado_fin > Entrada.txt
1 a = 5
2 b = 10
3 suma = a + b
4 print(suma)
```

```
(base2) PS C:\Users\Heber\OneDrive\Universidad\Nolasco Ramirez Heber_10\Programacion Sistemas Base II\Programacion\Unidad_4\ProyectoIntegrado_fin> java Main entrada.txt salida.txt
Análisis léxico completado. Tokens guardados en salida.txt

Análisis semántico completado.

Generación de código intermedio:
Código intermedio generado:

Hugging Face API llamada.
Dice:

? Sintaxis:
An\u00f3lisis sem\u00f3ntico:
= a + b
print(suma)
```

Salida.txt

The screenshot shows a terminal window with three tabs at the top: "entrada.txt", "salida.txt", and "app.py". The "salida.txt" tab is active, displaying the following text:

```
Unidad_4 > ProyectoIntegrado_fin > salida.txt
1 IDENTIFICADOR: a
2 TEXTO NO RECONOCIDO: =
3 NÚMERO: 5
4 IDENTIFICADOR: b
5 TEXTO NO RECONOCIDO: =
6 NÚMERO: 10
7 IDENTIFICADOR: suma
8 TEXTO NO RECONOCIDO: =
9 IDENTIFICADOR: a
10 TEXTO NO RECONOCIDO: +
11 IDENTIFICADOR: b
12 IDENTIFICADOR: print
13 TEXTO NO RECONOCIDO: [
14 IDENTIFICADOR: suma
15 TEXTO NO RECONOCIDO: ]
16
```

Ejemplo 1. Con errores

```
Unidad_4 > ProyectoIntegrado_fin > entrada.txt
1 a 5
2 b = 10
3 suma = a + b
4 print(suma)

(base2) PS C:\Users\Heber\OneDrive\Universidad\Wolasco Ramirez Heber_10\Programacion Sistemas Base II\Programacion\Unidad_4\ProyectoIntegrado_fin> java Main entrada.txt salida.txt
Analisis léxico completado. Tokens guardados en salida.txt
Error semántico: variable 'a' no declarada.
Error semántico: variable 'a' no declarada.
Analisis semántico completado.

Generación de código intermedio:
Código intermedio generado:

Hugging Face API llamada,
Dice:

? Sintaxis:
Errores de sintaxis:
an@00ellipsis:1:3: invalid syntax
a 5
^

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 1 out of bounds for length 1
at Main.main(Main.java:50)
```

Salida.txt

```
Unidad_4 > ProyectoIntegrado_fin > salida.txt
1 IDENTIFICADOR: a
2 NÚMERO: 5
3 IDENTIFICADOR: b
4 TEXTO NO RECONOCIDO: =
5 NÚMERO: 10
6 IDENTIFICADOR: suma
7 TEXTO NO RECONOCIDO: =
8 IDENTIFICADOR: a
9 TEXTO NO RECONOCIDO: +
10 IDENTIFICADOR: b
11 IDENTIFICADOR: print
12 TEXTO NO RECONOCIDO: (
13 IDENTIFICADOR: suma
14 TEXTO NO RECONOCIDO: )
15 |
```

Ejemplo 1. Errores

```
entra.txt  salida.txt  app.py  2
Unidad_4 > ProyectoIntegrado_fin > entra.txt
1 def mayor(x, y):
2     if x > y:
3         return x
4     else:
5         return y
6
7 resultado = mayor(8, 15)
8 print(resultado)

at Main.main(Main.java:50)
(base2) PS C:\Users\Heber\OneDrive\Universidad\Nolasco Ramirez Heber_10\Programacion Sistemas Base II\Programacion\Unidad_4\ProyectoIntegrado_fin> java Main entra.txt salida.txt
Análisis léxico completado. Tokens guardados en salida.txt

Análisis semántico completado.

Generación de código intermedio:
Código intermedio generado:
null = FUNC_BEGIN mayor
null = RETURN x
null = RETURN y

Hugging Face API llamada.
Dice:

? Sintaxis:
Análisis semántico:
== (0,= (0,
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 1 out of bounds for length 1
at Main.main(Main.java:50)
```

Conclusión

Conclusión

Este proyecto representa un esfuerzo integral por emular las fases fundamentales de un compilador tradicional, comenzando desde el análisis léxico hasta alcanzar una representación de código intermedio. Mediante el uso de herramientas como JFlex para la generación del analizador léxico y el desarrollo personalizado de módulos en Java para el análisis semántico y la generación de instrucciones intermedias, se logró construir un sistema modular, estructurado y extensible que refleja el flujo clásico de compilación.

Sin embargo, el aporte más innovador del proyecto radica en la integración de técnicas de inteligencia artificial a través de modelos de lenguaje de la plataforma Hugging Face. Este enfoque complementa la validación estructural del compilador con una validación lógica más abstracta, que permite detectar errores de tipo semántico o lógico que normalmente no serían capturados por el compilador, como condiciones erróneas, errores de lógica o estructuras mal planteadas.

La herramienta construida permite, por un lado, visualizar el procesamiento de código fuente desde su análisis léxico hasta su transformación intermedia, y por otro lado, proporciona al usuario un sistema de retroalimentación automatizado basado en IA que analiza funciones escritas en lenguaje Python. Si bien esta característica está limitada actualmente a dicho lenguaje (dado que el modelo de IA utilizado fue entrenado principalmente en Python y el sistema de verificación sintáctica usa herramientas específicas como Pyflakes), este componente sienta las bases para la expansión futura hacia lenguajes como JavaScript, C++ o Java, a través de prompts adaptativos y herramientas externas de análisis sintáctico.

Además, se destaca la aplicabilidad práctica del proyecto como recurso educativo, ya que puede servir como plataforma de aprendizaje para estudiantes que deseen entender cómo funcionan los compiladores y, al mismo tiempo, experimentar con tecnologías actuales de análisis de código asistido por inteligencia artificial.

En resumen, este proyecto no solo replica técnicamente las partes más importantes de un compilador, sino que también abre la puerta a una nueva forma de asistencia en la programación, combinando algoritmos deterministas con modelos probabilísticos avanzados. La combinación de estos enfoques tradicionales y modernos demuestra el potencial que tienen los sistemas híbridos para mejorar la experiencia del desarrollo de software, reducir errores y acelerar el proceso de validación de código en entornos de aprendizaje y desarrollo profesional.