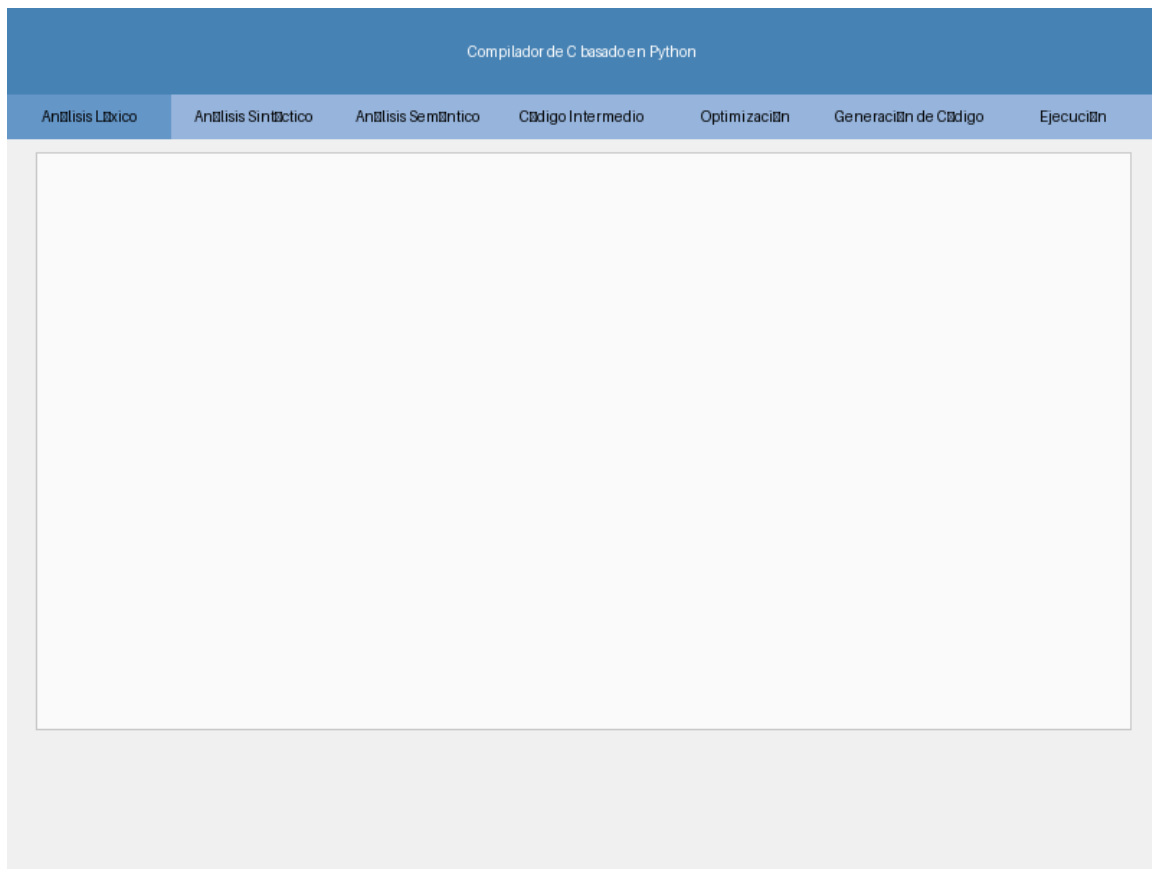


Manual con Capturas - Compilador de C basado en Python

Vista General de la Interfaz

La interfaz principal del compilador está organizada en pestañas, cada una correspondiente a una fase del proceso de compilación:



1. Análisis Léxico

El analizador léxico identifica tokens en el código fuente C. Los tokens incluyen palabras clave, identificadores, operadores, constantes, etc.

Compilador de C - Análisis Léxico

Código Fuente:

```
int main() {  
    int x = 10;  
    printf("Valor: %d\n", x);  
    return 0;  
}
```

Resultados del Análisis Léxico:

```
[{"type": "INT", "value": "10", "line": 1},  
{"type": "ID", "value": "main", "line": 1},  
...]
```

Tokens identificados en el código:

El análisis léxico identifica cada componente del código como un token con tipo, valor y posición. Esta información es crucial para detectar errores léxicos como identificadores inválidos o caracteres no reconocidos.

2. Análisis Sintáctico

El analizador sintáctico verifica la estructura del código según las reglas gramaticales del lenguaje C. Construye un árbol de sintaxis abstracta (AST).

Compilador de C - Análisis Sintáctico

Código Fuente:

```
int main() {  
    int x = 10;  
    printf("Valor: %d\n", x);  
    return 0;  
}
```

Resultados del Análisis Sintáctico:

```
{  
  "type": "program",  
  "body": [  
    {  
      "type": "function_declaration",  
      "name": "main",  
      ...  
    }  
  ]  
}
```

Estructura del AST generado:

El Árbol de Sintaxis Abstracta (AST) representa la estructura jerárquica del código. El analizador sintáctico puede detectar problemas como paréntesis desbalanceados o puntos y coma faltantes.

3. Análisis Semántico

El analizador semántico verifica la coherencia del significado del código. Confirma que los tipos de datos sean compatibles y que las variables estén declaradas.

Compilador de C - Análisis Semántico

Código Fuente:

```
int main() {  
    int x = 10;  
    printf("Valor: %d\n", x);  
    return 0;  
}
```

Resultados del Análisis Semántico:

Variables:

x: {type: int, initialized: true, scope: main}

Funciones:

main: {return_type: int, params: []}

Verificación de tipos y ámbitos:

El análisis semántico verifica que las variables sean usadas de manera consistente con sus tipos declarados y que estén en el ámbito correcto. También verifica la compatibilidad de tipos en expresiones y asignaciones.

4. Código Intermedio

El generador de código intermedio produce una representación independiente de la máquina del código fuente, usando un formato de tres direcciones.

Compilador de C - Código Intermedio

Código Fuente:	Resultados del Código Intermedio:
<pre>int main() { int x = 10; printf("Valor: %d\n", x); return 0; }</pre>	<pre>FUNCTION_BEGIN main t1 = 10 x = t1 PARAM "Valor: %d\n" PARAM x t2 = CALL printf, 2 t3 = 0 RETURN t3 FUNCTION_END</pre>

5. Optimización

El optimizador de código mejora el código intermedio para hacerlo más eficiente, aplicando técnicas como propagación de constantes y eliminación de código muerto.

Compilador de C - Optimización

Código Fuente:	Resultados de la Optimización:
<pre>int main() { int x = 10; printf("Valor: %d\n", x); return 0; }</pre>	<pre>FUNCTION_BEGIN main x = 10 PARAM "Valor: %d\n" PARAM x CALL printf 2 RETURN 0 FUNCTION_END</pre>

6. Generación de Código

El generador de código convierte el código intermedio optimizado en código ensamblador específico para la arquitectura objetivo.

Compilador de C - Generación de Código

Código Fuente:	Resultados del Generación de Código:
<pre>int main() { int x = 10; printf("Valor: %d\n", x); return 0; }</pre>	<pre>.globl main main: pushq %rbp movq %rsp, %rbp subq \$16, %rsp movl \$10, -4(%rbp) ...</pre>

7. Ejecución

La fase de ejecución compila y ejecuta el código generado para mostrar la salida del programa.

Compilador de C - Ejecución

Código Fuente:

```
int main() {  
    int x = 10;  
    printf("Valor: %d\n", x);  
    return 0;  
}
```

Resultados de la Ejecución:

```
(Simulación) Valor: 10
```