

Manual de Usuario - Compilador de C basado en Python

Introducción

Este manual describe el funcionamiento de un compilador de C basado en Python con una interfaz web interactiva. El compilador permite visualizar cada fase del proceso de compilación, incluyendo análisis léxico, sintáctico, semántico, generación de código intermedio, optimización y ejecución. Esta herramienta está diseñada principalmente con fines educativos.

Vista General de la Aplicación

La aplicación está organizada en pestañas, cada una correspondiente a una fase del proceso de compilación. La interfaz incluye: - Un editor de código C donde el usuario puede escribir o pegar código - Secciones de visualización de resultados para cada fase - Mensajes de error detallados con sugerencias - Visualizaciones gráficas para ayudar a entender el 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. Esta fase detecta errores léxicos como caracteres no reconocidos o literales mal formados.

Ejemplo de código:

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

Tokens generados:

```
[ {'type': 'INT', 'value': 'int', 'line': 1, 'position': 1}, {'type': 'ID', 'value': 'main', 'line': 1, 'position': 5}, {'type': 'LPAREN', 'value': '(', 'line': 1, 'position': 9}, {'type': 'RPAREN', 'value': ')', 'line': 1, 'position': 10}, {'type': 'LBRACE', 'value': '{', 'line': 1, 'position': 12}, ... ]
```

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) a partir de los tokens. Esta fase detecta errores como paréntesis desbalanceados, falta de puntos y coma, o estructuras de control mal formadas.

Ejemplo de AST generado:

```
{ "type": "program", "body": [ { "type": "function_declaration", "name": "main", "return_type": "int", "params": [], "body": { "type": "block_statement", "body": [ { "type": "variable_declaration", "name": "x",
```

```
"data_type": "int", "initial_value": { "type": "literal", "value": 10 } },  
... ] } } ] }
```

3. Análisis Semántico

El analizador semántico verifica la coherencia del significado del código. Confirma que los tipos de datos sean compatibles en las operaciones y que las variables estén declaradas antes de su uso. Construye una tabla de símbolos para rastrear variables, funciones y sus atributos.

Ejemplo de tabla de símbolos:

```
{ "variables": { "x": { "type": "int", "line_declared": 2, "scope": "main",  
"initialized": true } }, "functions": { "main": { "return_type": "int",  
"parameters": [], "line_declared": 1 }, "printf": { "return_type": "int",  
"parameters": ["const char*", "..."], "line_declared": 0, "is_extern": true }  
} }
```

4. Generación de Código Intermedio

El generador de código intermedio produce una representación independiente de la máquina del código fuente. Este código de tres direcciones es más fácil de optimizar y traducir a código de máquina.

Ejemplo de código intermedio:

```
FUNCTION_BEGIN main t1 = 10 x = t1 PARAM "Valor: %d " PARAM x t2 = CALL
printf, 2 t3 = 0 RETURN t3 FUNCTION_END
```

5. Optimización de Código

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

Ejemplo de código optimizado:

```
FUNCTION_BEGIN main x = 10 PARAM "Valor: %d " PARAM x CALL printf, 2 RETURN 0
FUNCTION_END
```

6. Generación de Código de Máquina

El generador de código convierte el código intermedio optimizado en código ensamblador específico para la arquitectura objetivo (como x86).

Ejemplo de código ensamblador:

```
; Código Ensamblador Simulado .text .globl main main: pushq %rbp movq %rsp,
%rbp subq $16, %rsp ; Inicializar x = 10 movl $10, -4(%rbp) ; Llamada a
printf("Valor: %d ") leaq .LC0(%rip), %rdi movl -4(%rbp), %esi call
printf@PLT ; Retornar 0 movl $0, %eax leave ret .section .rodata .LC0:
.string "Valor: %d "
```

7. Ejecución

La fase de ejecución compila y ejecuta el código generado para mostrar la salida del programa. En entornos donde no está disponible el compilador real (como Replit), se proporciona una simulación educativa.

Ejemplo de salida:

```
(Simulación) Valor: 10
```

Detección de Errores

El compilador proporciona mensajes de error detallados en cada fase del proceso. Los mensajes incluyen: - Número de línea y posición exacta del error - Descripción del problema en lenguaje claro - Sugerencias para la corrección - Indicadores visuales que señalan el punto exacto del error

Ejemplos de errores comunes:

Tipo de Error	Ejemplo	Mensaje
Error léxico	int 2x = 10;	Error léxico: Identificador inválido '2x' en línea 1, posición 5. Los identi
Error sintáctico	if (x > 0) { print(x) }	Error sintáctico: Falta punto y coma (;) después de 'print(x)' en línea 2.
Error semántico	int x = "hello";	Error semántico: No se puede asignar string a variable de tipo int en lí

Características Adicionales

El compilador incluye varias características adicionales que lo hacen ideal para entornos educativos: - Interfaz completamente en español - Visualización del flujo de control del programa - Simulación de ejecución cuando el compilador real no está disponible - Sugerencias contextuales para corregir errores comunes - Ejemplos de código predefinidos para cada fase del compilador