

UNAM, SEMESTER 2026-1

Compiler with Assembly Code Generation

Account Numbers

320013447
318276588
320195019
320095531
320042645
320280087

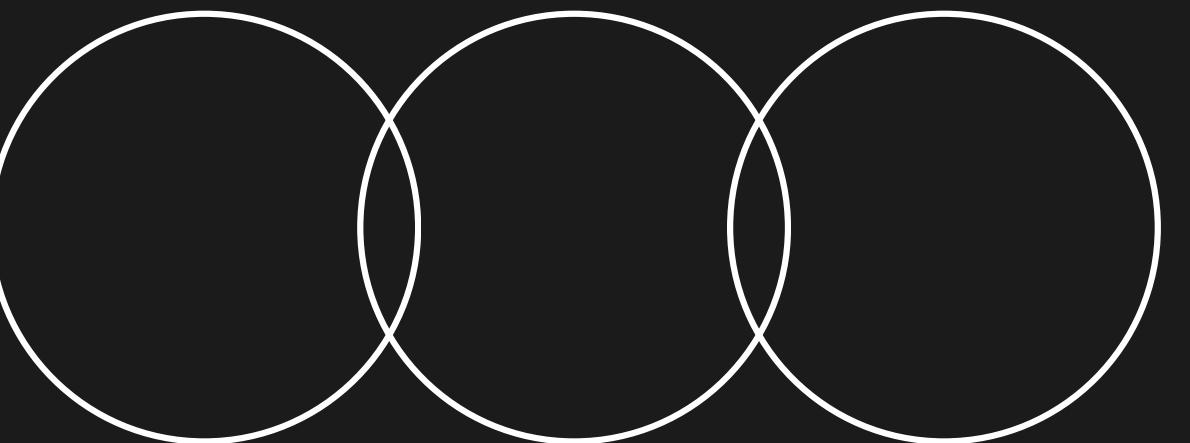
Professor: Prof. René Adrián Dávila Pérez
Delivery date: Monday, December 1st, 2025



Motivación del Compilador

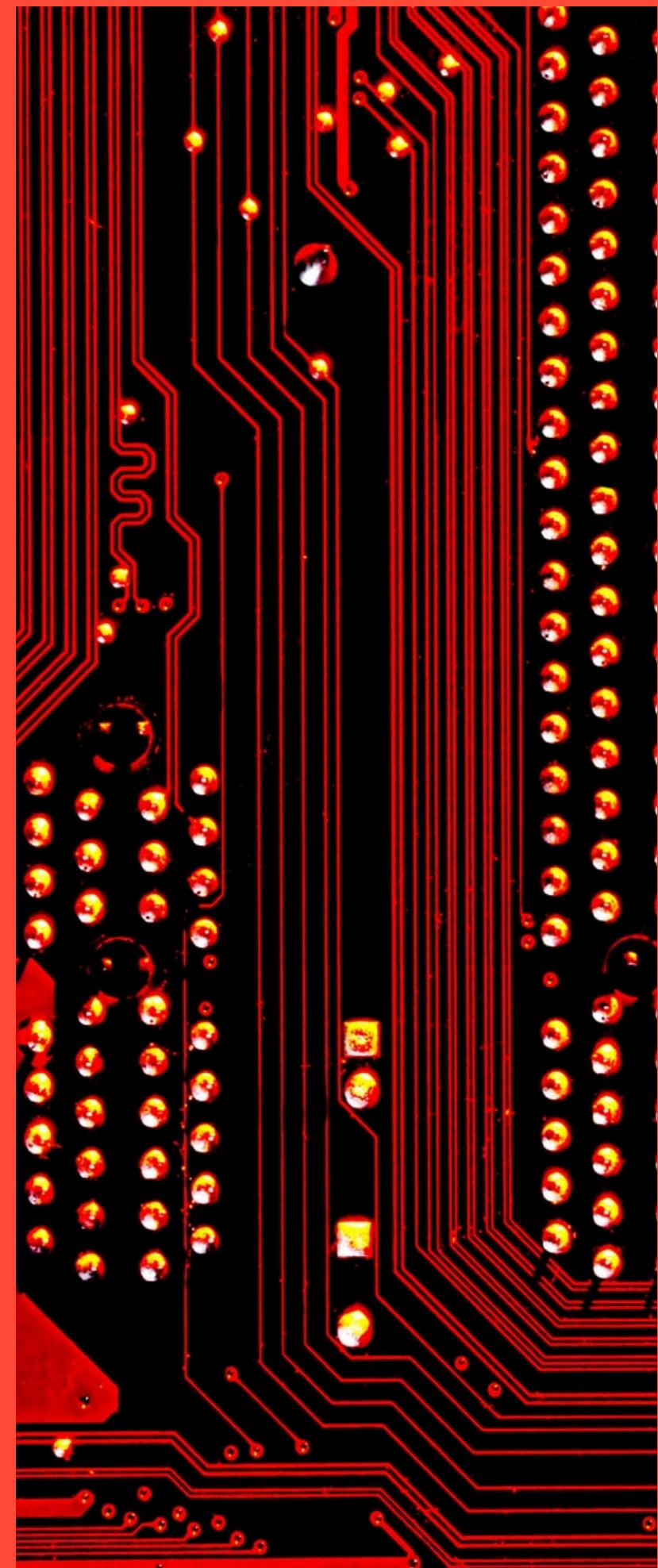
Importancia del Proyecto

El Compilador gráfico en Python (Lark) traduce código a LLVM IR, permitiendo visualizar árboles de sintaxis y detectar errores mediante análisis léxico, sintáctico y semántico.



OBJETIVO GENERAL

Desarrollar un compilador con interfaz gráfica, análisis integral y generación de código LLVM IR.



Objetivos Específicos

Metas y Diagrama General

Diseño de gramática

Crear una gramática libre de contexto para declaraciones, expresiones y condicionales

Análisis léxico y sintáctico

Generar tokens y árboles de sintaxis mediante Lark (LALR)

Análisis semántico:

Verificar uso correcto de variables mediante una tabla de símbolos.

Generación de código LLVM

Generar código intermedio y estructuras de control usando llvmlite

1. **Interfaz gráfica**
2. **Validación exhaustiva**
3. **Documentación visual**

Marco Teórico

- **Compilador/Ensamblador:** Traducción completa a bajo nivel.
- **Intérprete:** Ejecución línea por línea.
- **Proyecto:** Funciona como compilador generando LLVM IR.

El proyecto opera como un sistema híbrido que compila el código fuente generando LLVM IR



Gramáticas Libres de Contexto (CFG)

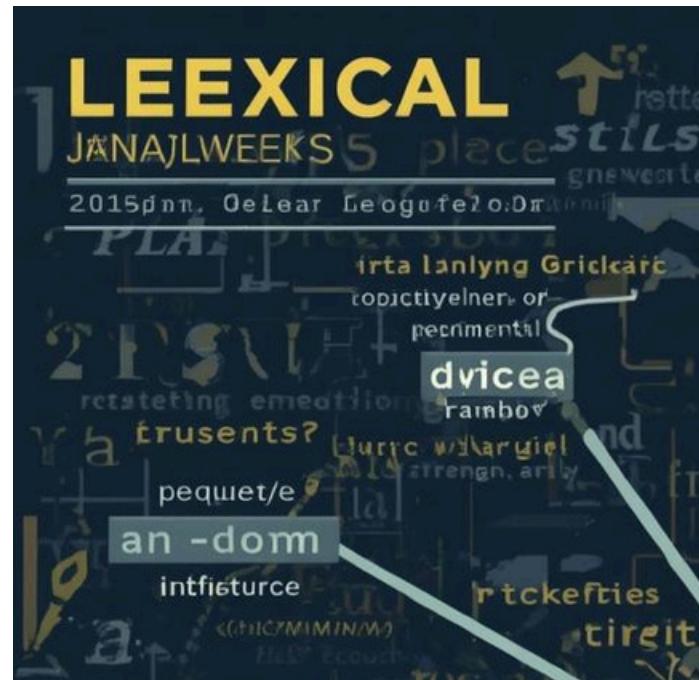
Árboles de Sintaxis Abstracta (AST)

LLVM y su Representación Intermedia (LLVMIR)

Autómatas Finitos y Análisis Léxico

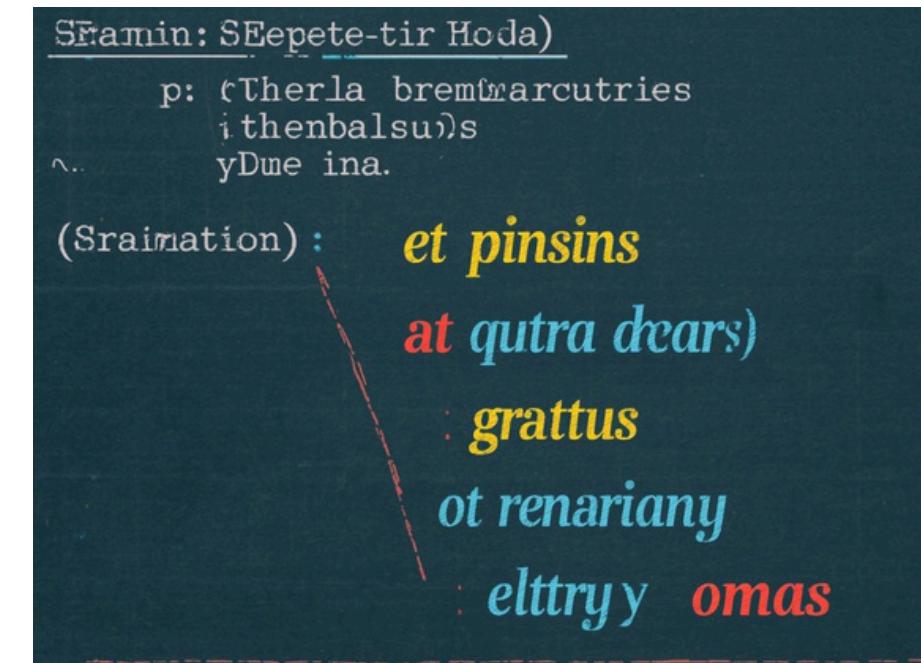
Tablas de Símbolos

Fases del Compilador



01

Análisis léxico
convierte texto en
tokens
identificables y
significativos.



02

Análisis sintáctico
construye un árbol de
sintaxis abstracta para
validación.

03

Análisis semántico
garantiza la coherencia y
correcta interpretación
del código.

```
return self.builder.sd
elif tag == 'compare':
    left_val = self.eval_expr(
        right_val = self.eval_expr(
            op = content[1].value
            # Comparison with ii result
            res_ii = self.builder.icmp(
                op, left_val, right_val
            )
        )
    )

```

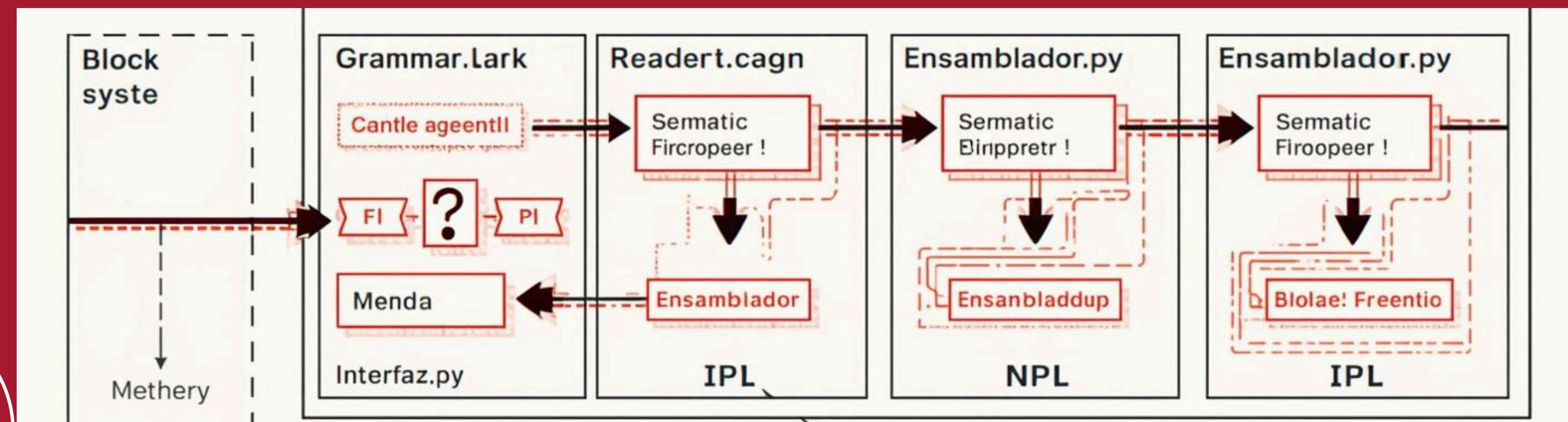
04

Generación de código
produce instrucciones
LLVM IR para ejecución
eficiente.

Arquitectura del Sistema

El compilador implementado sigue una arquitectura modular clásica. Cada fase opera de manera independiente pero coordinada para transformar el código fuente en LLVM IR.

Código Fuente → Análisis Léxico → Parsing → AST → Análisis Semántico → LLVM IR → Salida



Arquitectura del Sistema

01 grammar.lark

Módulo de Gramática

Define la sintaxis del lenguaje mediante EBNF.

02 Lark con parser LALR

Analizador Léxico y Sintáctico

Genera el lexer y parser

03 ast_builder.py

Transformador AST

Convierte el árbol sintáctico en un Árbol de Sintaxis Abstracta semánticamente estructurado.

04 semantic_analyzer.py

Recorre el AST, valida:

- Declaración antes de uso
- No redeclaración
- Compatibilidad de tipos
- Alcances válidos

05 Generador de Código assembler.py

Traduce el AST a LLVM IR usando llvmlite.

06 Interfaz Gráfica

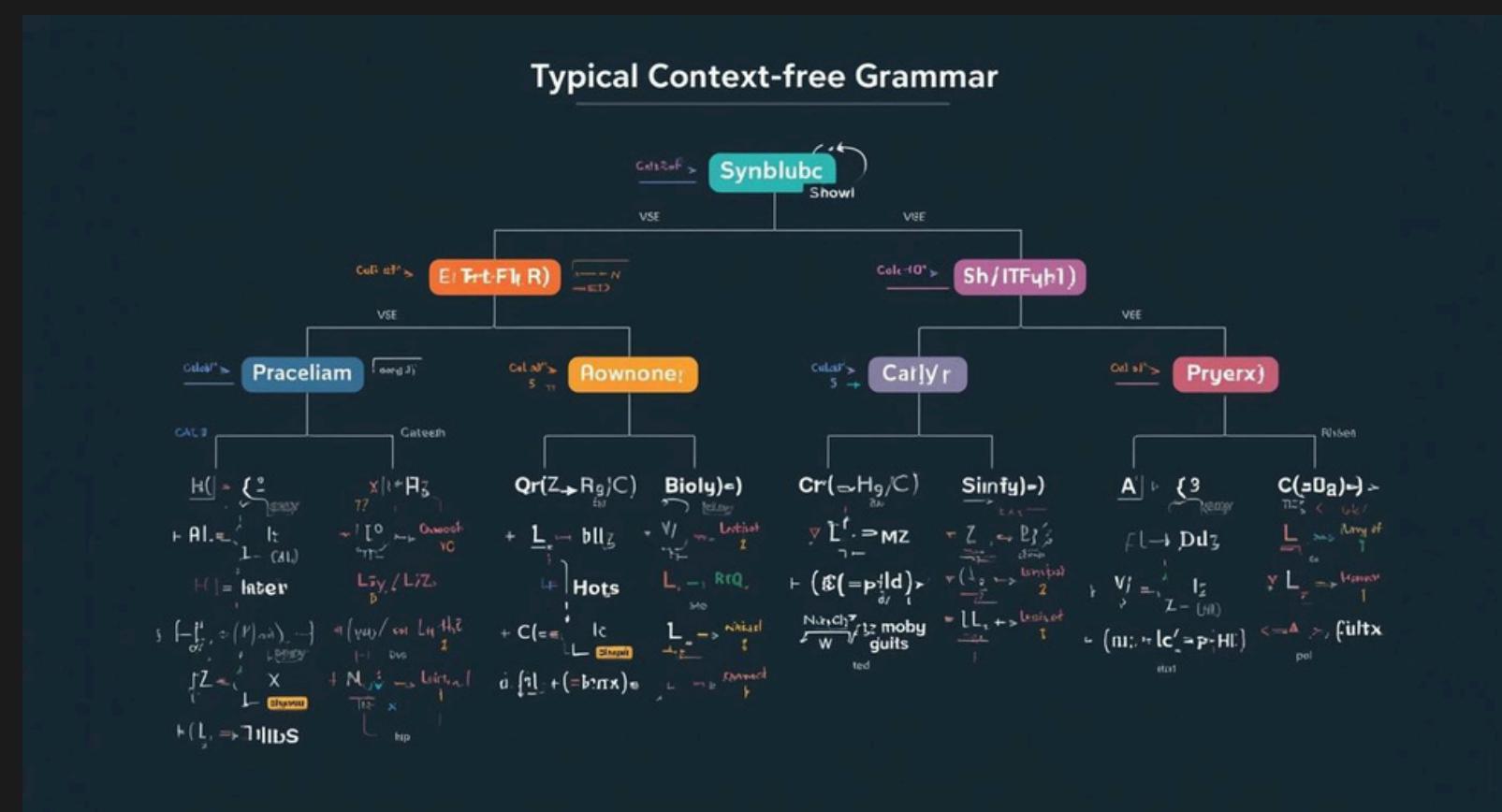
- Editor de código integrado
- Visualización de AST y errores
- Generación y visualización de LLVM IR

Gramática y Análisis Sintáctico

Base Sintáctica: Define la estructura esencial para el parser.

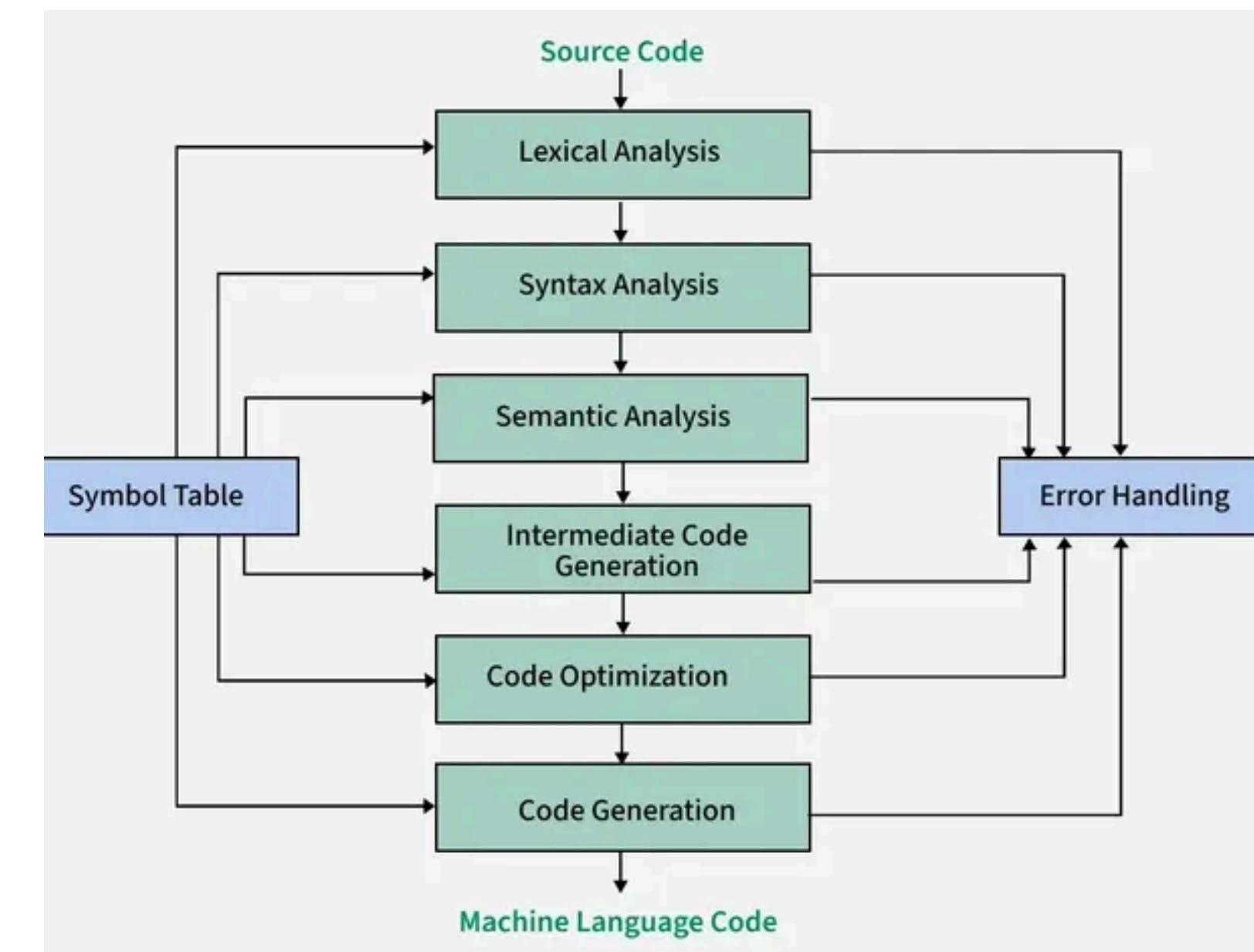
- **Operaciones:** Soporta declaraciones, asignaciones y expresiones (aritméticas/lógicas).
 - **Control de Flujo:** Maneja estructuras condicionales If-Else y bloques {}.
 - **Tipos de Datos:** Incluye literales numéricos, cadenas y caracteres.

Conexión teórica y práctica



Análisis Semántico

- **Gestión de Identificadores:** Verifica la existencia previa y la unicidad (evita redeclaraciones) mediante una Tabla de Símbolos.
- **Chequeo de Tipos:** Asegura la consistencia de datos en asignaciones y operaciones aritméticas/lógicas.
- **Control de Alcance (Scopes):** Valida la visibilidad y vida de las variables dentro de bloques anidados y estructuras de control.

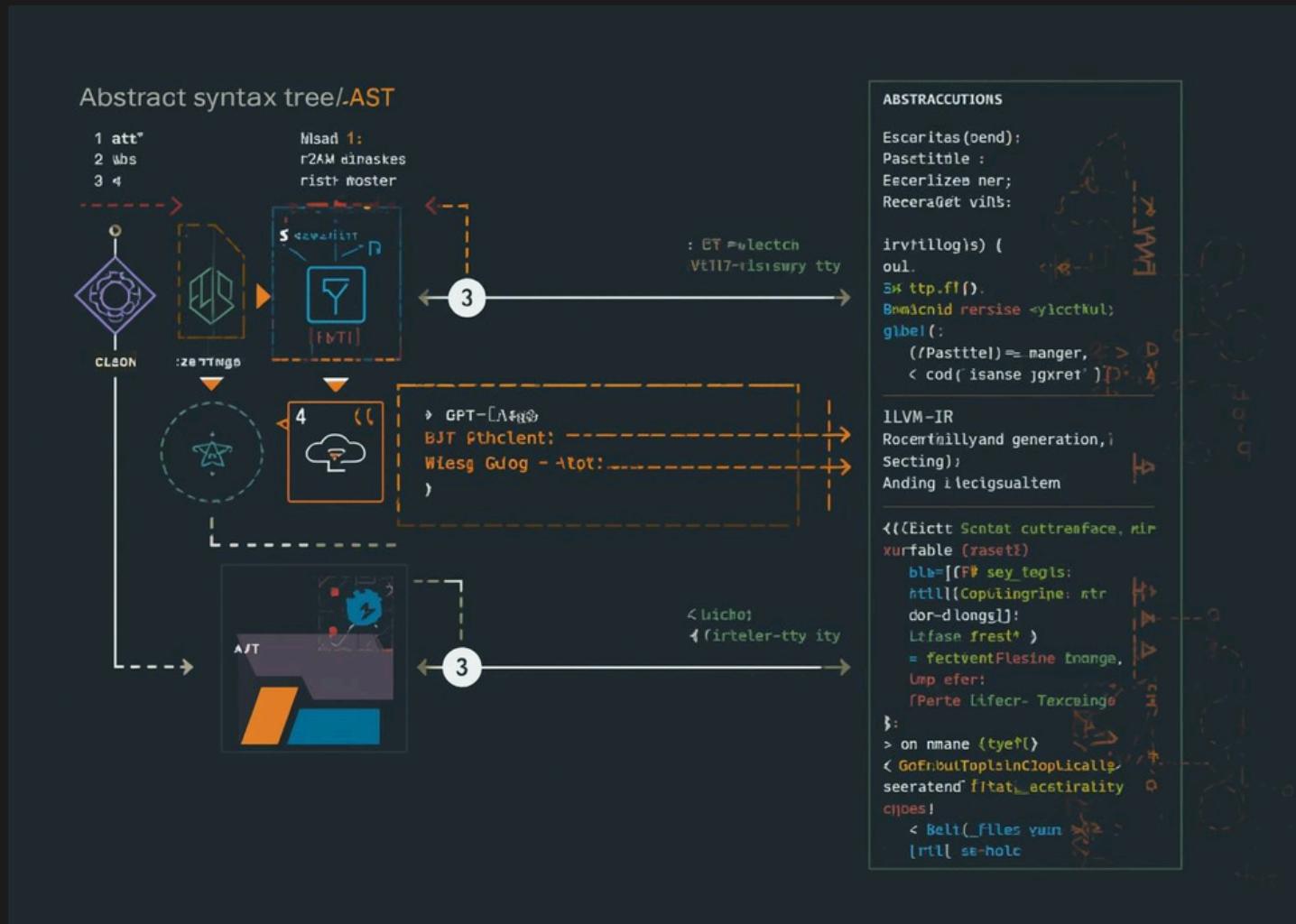


Generación de Código LLVM IR

Estructura y funcionalidad

Crucial para la **traducción** de lenguajes de alto nivel a instrucciones que la máquina puede entender y ejecutar.

- El compilador produce LLVM IR en estilo SSA:
- Instrucciones RISC (add, mul, icmp...)
- Flujo con bloques básicos y br condicional
- Manejo de variables con alloca, store, load
- Estructura de funciones con parámetros y retorno



Resultados y Validación

Los resultados muestran un funcionamiento estable y consistente con los objetivos planteados.

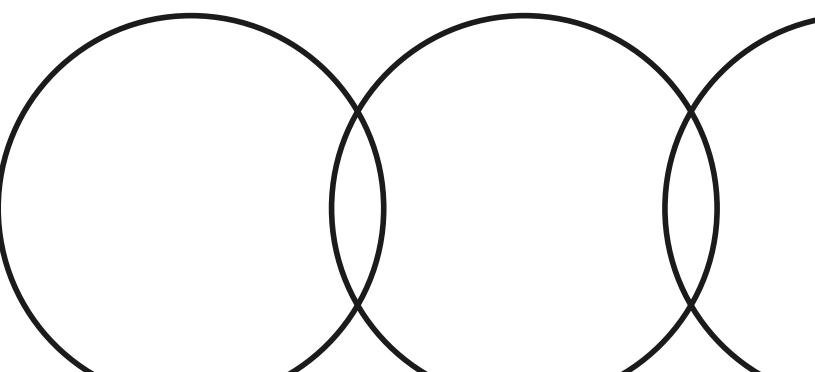
Se verificó que el compilador procesara correctamente:

- Declaraciones de variables
- Asignaciones simples y compuestas
- Operaciones aritméticas y lógicas
- Expresiones anidadas
- Estructuras de control if–else
- Bloques de código

Logros Técnicos

¿Por qué elegir nuestro producto?

Hemos desarrollado un **compilador funcional** que integra generación de código LLVM IR y una interfaz gráfica amigable, facilitando la interacción y aprendizaje del usuario durante el proceso de compilación.



Resultados y Validación

Validación correcta

RESULTS

```
Compilación exitosa!

--- Preview Ensamblador ---
; ModuleID = "module"
target triple = "unknown-unknown-unknown"
target datalayout = ""

define void @"main"()
... (Ver completo en botón Assembler)
```

Enter your code:

```
var a = 10;
var b = 5;
var suma = a + b;
var compleja = a * b - 2;
```

RESULTS

```
program
    var_declaration
        a
        =
        number 10
        ;
    var_declaration
        b
        =
        number 5
        ;
    var_declaration
        suma
        =
```

RESULTS

```
target datalayout = ""

define void @"main"()
{
entry:
    %"a" = alloca i32
    store i32 10, i32* %"a"
    %"b" = alloca i32
    store i32 5, i32* %"b"
    %"suma" = alloca i32
    %"a.1" = load i32, i32* %"a"
    %"b.1" = load i32, i32* %"b"
    %"add" = add i32 %"a.1", %"b.1"
    store i32 %"add", i32* %"suma"
```

Resultados y Validación

Enter your code:

```
var x = 100;
var y = 50;
var resultado = 0;

if (x != y) {
    var temp = x - y;
    if (temp > 20) {
        resultado = 1;
    }
}
```

RESULTS

```
program
  var_declaration
    x
    =
    number 100
    ;
  var_declaration
    y
    =
    number 50
    ;
  var_declaration
    resultado
    =
```

caso estructura if

RESULTS

```
define void @"main"()
{
entry:
%"x" = alloca i32
store i32 100, i32* %"x"
%"y" = alloca i32
store i32 50, i32* %"y"
%"resultado" = alloca i32
store i32 0, i32* %"resultado"
%"x.1" = load i32, i32* %"x"
%"y.1" = load i32, i32* %"y"
%"cmp" = icmp ne i32 %"x.1", %"y.1"
%"bool_to_int" = zext i1 %"cmp" to i32
```

RESULTS

```
br i1 %"ifcond", label %"then", label %"merge
then:
%"temp" = alloca i32
%"x.2" = load i32, i32* %"x"
%"y.2" = load i32, i32* %"y"
%"sub" = sub i32 %"x.2", %"y.2"
store i32 %"sub", i32* %"temp"
%"temp.1" = load i32, i32* %"temp"
%"cmp.1" = icmp sgt i32 %"temp.1", 20
%"bool_to_int.1" = zext i1 %"cmp.1" to i32
%"ifcond.1" = icmp ne i32 %"bool_to_int.1", 0
br i1 %"ifcond.1", label %"then.1", label %"merge.1"
merge:
ret void
```

Resultados y Validación

caso error cadena

Enter your code:

```
var string = "Cadena";
```

RESULTS

Compilación exitosa!

Error general!:

Operación no soportada: string

RESULTS

```
program
    var_declaracion
        string
        =
        string "Cadena"
    ;
```

RESULTS

Error generando ensamblador: Operación no soportada: string

Resultados y validación

Enter your code:

```
var entero = 100
```

RESULTS

Error sintáctico:
Unexpected token Token('\$END', '') at line 1, column 14.
Expected one of:
* SEMI
* LOGICAL_OP
* COMPARISON_OP
* ARITH_OP

Enter your code:

```
var entero = 100;  
suma = entero + 1;
```

RESULTS

Errores semánticos:
- Error: variable 'suma' no declarada antes de asignar.

Enter your code:

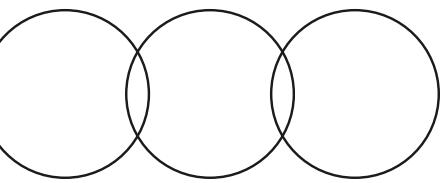
```
var a = 10;  
var b = 5;  
  
var suma = a + b;  
var compleja = a * b - 2;  
  
a = compleja / 2;  
var a = 10 * 5;
```

RESULTS

Errores semánticos:
- Error: variable 'a' ya declarada.

caso error, sintactico y semantic

Futuras Mejoras

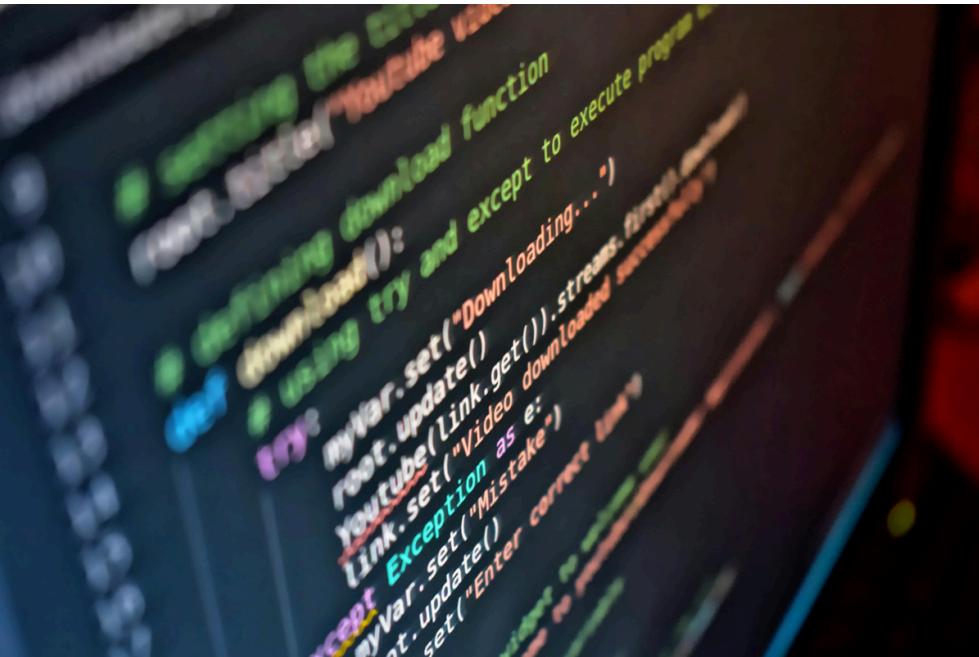


Proyecciones para el proyecto



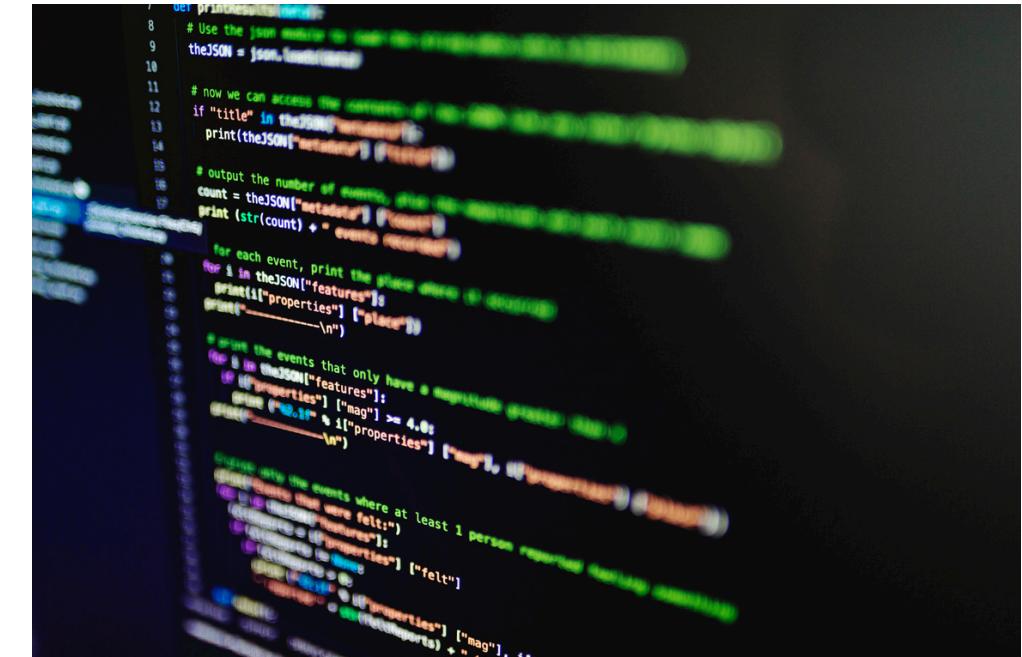
Soporte de funciones

Se planea **implementar soporte** para funciones, permitiendo la creación de programas más complejos y haciendo el compilador más útil para el desarrollo de software real.



Manejo de errores

Se busca mejorar el **manejo de errores** para proporcionar mensajes más informativos al usuario, facilitando la identificación y corrección de problemas en el código fuente.



Optimización del código

La optimización del **código generado** es esencial, con el objetivo de mejorar la eficiencia y el rendimiento de los programas compilados, haciendo el compilador más competitivo y eficaz.

BIBLIOGRAFÍA

1. AHO, A. V., LAM, M. S., SETHI, R., & ULLMAN, J. D. (2006). **COMPILERS: PRINCIPLES, TECHNIQUES, AND TOOLS** (2ND ED.). PEARSON EDUCATION.
2. COOPER, K. D., & TORCZON, L. (2012). **ENGINEERING A COMPILER** (2ND ED.). MORGAN KAUFMANN.
3. LATTNER, C., & ADVE, V. (2004). LLVM: A COMPILATION FRAMEWORK FOR LIFELONG PROGRAM ANALYSIS & TRANSFORMATION. PROCEEDINGS OF THE INTERNATIONAL SYMPOSIUM ON CODE GENERATION AND OPTIMIZATION (CGO), 75–86. [HTTPS://DOI.ORG/10.1109/CGO.2004.1281665](https://doi.org/10.1109/CGO.2004.1281665)
4. LARK-PARSER. (2023). LARK – A MODERN PARSING LIBRARY FOR PYTHON. [HTTPS://GITHUB.COM/LARK-PARSER/LARK](https://github.com/lark-parser/lark)
5. LLVM PROJECT. (2023). LLVM LANGUAGE REFERENCE MANUAL. [OLA](#)