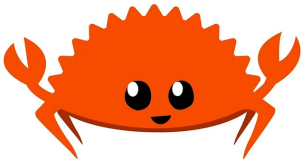

**Universidad de La Habana**

Facultad de Matemática y Computación

Departamento de Ciencia de la Computación

# Implementación de un Compilador para el Lenguaje Hulk

*Proyecto de Compiladores*

**Autores:**

José Ernesto Morales Lazo
jose.emorales@estudiantes.matcom.uh.cu

Salma Fonseca Curbelo
salma.fonseca@estudiantes.matcom.uh.cu

15 de junio de 2025

**Resumen**

Se presenta la implementación de un compilador completo para el lenguaje de programación Hulk, un lenguaje orientado a objetos con características funcionales. El compilador implementa todas las fases tradicionales: análisis léxico, sintáctico, semántico y generación de código LLVM IR. Se desarrolló utilizando técnicas modernas de construcción de compiladores, incluyendo un generador de analizador léxico personalizado y Bison para el análisis sintáctico. La implementación soporta características avanzadas como herencia, polimorfismo, inferencia de tipos, expresiones funcionales y estructuras de control complejas.

**Palabras clave:** Compiladores, LLVM, Análisis Sintáctico, Generación de Código, Programación Orientada a Objetos

# Índice

# 1 Introducción

El desarrollo de compiladores constituye una de las áreas fundamentales de las ciencias de la computación, combinando teoría formal con implementación práctica. En este trabajo se presenta la implementación completa de un compilador para el lenguaje Hulk, un lenguaje de programación que combina características de programación orientada a objetos con elementos funcionales.

El lenguaje Hulk se caracteriza por su sintaxis expresiva que permite definiciones de tipos con herencia, métodos, funciones de primera clase, expresiones condicionales complejas, iteradores y un sistema de tipos dinámico con capacidades de verificación estática. La implementación del compilador abarca desde el análisis léxico hasta la generación de código ejecutable mediante LLVM IR.

**Objetivos del Proyecto**

- Implementar un compilador completo y funcional para el lenguaje Hulk
- Aplicar técnicas modernas de construcción de compiladores
- Generar código optimizado mediante LLVM IR
- Soportar todas las características avanzadas del lenguaje
- Proporcionar manejo robusto de errores en todas las fases

# 2 Descripción del Proyecto

## 2.1 Información del Repositorio

El proyecto se encuentra alojado en GitHub en la siguiente URL: `https://github.com/joselazo21/Hulk_Compiler.git`

El repositorio contiene la implementación completa del compilador, incluyendo:

**Contenido del Repositorio**

- Código fuente del compilador (45+ archivos)
- Sistema de construcción con CMake y Makefile
- Documentación técnica

## 2.2 Características del Lenguaje Hulk

El lenguaje Hulk implementado soporta las siguientes características principales:

- **Sistema de tipos**: Tipos primitivos (Number, String, Boolean) y tipos definidos por el usuario con verificación estática y dinámica
- **Programación orientada a objetos**: Definición de clases con herencia simple
- **Funciones lambda**: Soporte para expresiones lambda y funciones anónimas
- **Estructuras de control**: Condicionales if-elif-else anidados, bucles while y for con iteradores personalizados

- **Operaciones avanzadas**: Concatenación de strings con `@@`, operaciones matemáticas completas, comparaciones con sobrecarga
- **Verificación de tipos**: Operadores `is` y `as` para verificación y conversión segura de tipos
- **Expresiones let-in**: Vinculación local de variables con alcance léxico
- **Manejo de memoria**: Gestión automática con soporte para estructuras complejas
- **Funciones built-in**: Biblioteca estándar con funciones matemáticas, I/O y utilidades

## 2.3 Ejemplos de Código Hulk

A continuación se presentan ejemplos representativos del lenguaje que demuestran sus capacidades principales:

```
type Person(firstname, lastname) {
    firstname : String = firstname;
    lastname : String = lastname;
    name() : String => self.firstname @@ " " @@ self.lastname;
}

type Knight inherits Person {
    name() : String => "Sir " @@ base();
}

let p = new Knight("Phil", "Collins") in
    print(p.name()); // Output: "Sir Phil Collins"
```

Listing 1: Definición de tipos con herencia

```
function mean(numbers: Number[]): Number =>
let total = 0 in {
    for (x in numbers)
        total := total + x;
    total / numbers.size();
};

let data = [1, 2, 3, 4, 5] in
let average = mean(data) in
    print("Average: " @@ average);
```

Listing 2: Funciones de orden superior y arrays

**Características Destacadas en los Ejemplos**

- **Herencia**: El tipo `Knight` hereda de `Person` y sobrescribe el método `name()`
- **Expresiones let-in**: Vinculación local de variables con alcance controlado
- **Funciones de orden superior**: La función `mean` opera sobre arrays de números
- **Iteradores**: El bucle `for` itera sobre elementos del array
- **Concatenación de strings**: Uso del operador `@@` para unir cadenas

# 3 Arquitectura del Compilador

## 3.1 Visión General

El compilador se implementó siguiendo la arquitectura clásica de compiladores con las siguientes fases bien diferenciadas:

**Fases del Compilador**

1. **Análisis Léxico**: Generador personalizado de lexer con soporte para tokens complejos y manejo de errores léxicos
2. **Análisis Sintáctico**: Parser LR(1) generado con Bison con manejo avanzado de errores y recuperación inteligente
3. **Análisis Semántico**: Verificación de tipos, construcción de contextos y resolución de símbolos con soporte para herencia
4. **Generación de Código**: Emisión de LLVM IR optimizado con soporte para runtime y biblioteca estándar

| Fase | Entrada | Salida | Herramienta |
|---|---|---|---|
| Léxico | Código fuente | Tokens | Lexer personalizado |
| Sintáctico | Tokens | AST | GNU Bison |
| Semántico | AST | AST anotado | Visitor pattern |
| Generación | AST anotado | LLVM IR | LLVM API |

Cuadro 1: Pipeline de compilación del compilador Hulk

## 3.2 Componentes Principales

### 3.2.1. Generador de Analizador Léxico

Se desarrolló un generador personalizado de analizador léxico (`lexer_generator.cpp`) que procesa una especificación de tokens y genera código C++ optimizado para el reconocimiento léxico.

Características del lexer:

- Reconocimiento de tokens complejos (strings, números, identificadores)
- Manejo de comentarios de línea y bloque
- Soporte para caracteres Unicode en strings
- Optimización mediante tablas de transición
- Reporte detallado de errores léxicos con posición

### 3.2.2. Analizador Sintáctico

Se utilizó GNU Bison para generar un parser LR(1) a partir de una gramática libre de contexto cuidadosamente diseñada. La gramática implementada maneja:

- Precedencia y asociatividad de 15+ operadores
- Resolución de conflictos shift/reduce mediante precedencia

- Construcción incremental del Árbol de Sintaxis Abstracta (AST)
- Manejo de errores sintácticos con recuperación inteligente
- Soporte para expresiones anidadas complejas
- Parsing de definiciones de tipos con herencia

| **Operador** | **Precedencia** | **Asociatividad** |
|---|---|---|
| `\|\|` | 1 | Izquierda |
| `&&` | 2 | Izquierda |
| `==, !=` | 3 | Izquierda |
| `<, <=, >, >=` | 4 | Izquierda |
| `@@` | 5 | Izquierda |
| `+, -` | 6 | Izquierda |
| `*, /, %` | 7 | Izquierda |
| `^` | 8 | Derecha |
| `is, as` | 9 | Izquierda |
| `!` | 10 | Derecha |

Cuadro 2: Tabla de precedencia de operadores en Hulk

### 3.2.3. Árbol de Sintaxis Abstracta

El AST se implementó utilizando el patrón Visitor con una jerarquía de clases que representa todos los constructos del lenguaje. La arquitectura permite extensibilidad y mantenimiento eficiente:

- **Nodos base**: `Node`, `Expression`, `Statement`
- **Expresiones**: `BinaryOperation`, `FunctionCall`, `LetIn`, `IfExpression`, `WhileExpression`
- **Declaraciones**: `FunctionDeclaration`, `TypeDefinition`, `Assignment`
- **Literales**: `Number`, `StringLiteral`, `Variable`
- **Estructuras**: `VectorExpression`, `MemberAccess`, `NewExpression`
- **Control de flujo**: `ForStatement`, `WhileStatement`, `ReturnStatement`

Cada nodo del AST implementa métodos para:

- Verificación semántica (`semanticCheck()`)
- Generación de código (`codegen()`)
- Inferencia de tipos (`getType()`)
- Optimización local (`optimize()`)

### 3.2.4. Sistema de Contextos

Se implementó un sistema de contextos anidados jerárquico para el manejo de alcances, verificación semántica y resolución de símbolos:

```
class Context : public IContext {
private:
    std::map<std::string, llvm::Value*> variables;
    std::map<std::string, llvm::Function*> functions;
    std::map<std::string, TypeInfo> types;
    std::map<std::string, llvm::StructType*> structTypes;
    IContext* parent;
    int scopeLevel;

public:
    IContext* createChildContext() override;
    bool defineVariable(const std::string& name,
                        llvm::Value* value, TypeInfo type) override;
    llvm::Value* lookupVariable(const std::string& name) override;
    bool defineFunction(const std::string& name,
                        llvm::Function* func) override;
    TypeInfo resolveType(const std::string& typeName) override;
    bool isInheritanceValid(const std::string& child,
                            const std::string& parent) override;
};
```

Listing 3: Estructura del sistema de contextos

El sistema de contextos soporta:

- Alcance léxico con shadowing controlado
- Resolución de herencia y polimorfismo
- Verificación de tipos en tiempo de compilación
- Manejo de funciones sobrecargadas
- Contextos especializados para iteradores

# 4 Implementación de Características Avanzadas

## 4.1 Sistema de Tipos y Herencia

Se implementó un sistema de tipos robusto que soporta herencia simple con verificación estática y dinámica:

```
type Point(x, y) {
    x: Number = x;
    y: Number = y;

    getX(): Number => self.x;
    setX(newX): Number => self.x := newX;
    distance(): Number => sqrt(self.x ^ 2 + self.y ^ 2);
}

type PolarPoint inherits Point {
    rho(): Number => self.distance();
    theta(): Number => atan2(self.y, self.x);

```

```
    // Override del m\'etodo padre
    distance(): Number => base();
}
```

Listing 4: Definición de tipos con herencia

El sistema de tipos implementa:

- **Verificación estática**: Chequeo de tipos en tiempo de compilación
- **Herencia simple**: Soporte para jerarquías de tipos con `inherits`
- **Polimorfismo**: Métodos virtuales con dispatch dinámico
- **Verificación dinámica**: Operadores `is` y `as` para runtime checks
- **Inferencia de tipos**: Deducción automática en expresiones `let-in`

## 4.2 Generación de Código LLVM

La generación de código se realizó utilizando la API de LLVM con optimizaciones avanzadas:

- **Generación de funciones**: Creación de funciones LLVM con manejo de parámetros y tipos de retorno
- **Manejo de memoria**: Asignación inteligente con malloc/free y stack allocation
- **Optimizaciones**: Aplicación de pases de optimización de LLVM (DCE, CSE, inlining)
- **Tipos complejos**: Soporte para estructuras, arrays y punteros
- **Runtime support**: Integración con biblioteca de runtime en C
- **Interoperabilidad**: Generación de código compatible con ABI estándar

```
llvm::Value* BinaryOperation::codegen(CodeGenerator& gen) {
    llvm::Value* leftVal = left->codegen(gen);
    llvm::Value* rightVal = right->codegen(gen);

    switch(op) {
        case ADD:
            return gen.getBuilder().CreateFAdd(leftVal, rightVal, "
               addtmp");
        case SUB:
            return gen.getBuilder().CreateFSub(leftVal, rightVal, "
               subtmp");
        case MUL:
            return gen.getBuilder().CreateFMul(leftVal, rightVal, "
               multmp");
        case DIV:
            return gen.getBuilder().CreateFDiv(leftVal, rightVal, "
               divtmp");
        default:
            return nullptr;
    }
}
```

Listing 5: Ejemplo de generación de código LLVM

## 4.3 Iteradores y Comprensiones

Se implementó un sistema de iteradores sofisticado que soporta:

```
// Iterador simple
for (x in range(1, 10))
    print(x * 2);

// List comprehension con filtro
let evens = [x | x in range(1, 20), x % 2 == 0] in
    print(evens);

// Iterador sobre arrays
let numbers = [1, 2, 3, 4, 5] in
for (n in numbers)
    print("Number: " @@ n);

// Iterador anidado
for (i in range(1, 4))
    for (j in range(1, 4))
        print("(" @@ i @@ ", " @@ j @@ ")");
```

Listing 6: Ejemplo de iteradores y comprensiones

El sistema de iteradores implementa:

- **Protocolo de iteración**: Interfaz estándar con `next()`, `current()`, `hasNext()`
- **Iteradores built-in**: `range()`, iteradores de arrays, strings
- **List comprehensions**: Sintaxis expresiva con filtros opcionales
- **Optimización**: Generación de código eficiente para bucles
- **Manejo de estado**: Gestión automática del estado del iterador

## 4.4 Sistema de Runtime

Se desarrolló una biblioteca de runtime completa que incluye:

- **Funciones matemáticas**: `sqrt`, `sin`, `cos`, `exp`, `log`, etc.
- **Manejo de strings**: Concatenación, comparación, conversión
- **I/O**: Funciones de entrada y salida (`print`, `read`)
- **Manejo de arrays**: Creación, acceso, redimensionamiento
- **Verificación de tipos**: Runtime type checking para operadores `is`/`as`
- **Manejo de errores**: Sistema de excepciones básico

```
// runtime_support.c
double hulk_sqrt(double x) {
    if (x < 0) {
        hulk_runtime_error("sqrt: negative argument");
        return 0.0;
    }
    return sqrt(x);
}

```

```c
char* hulk_string_concat(const char* a, const char* b) {
    size_t len_a = strlen(a);
    size_t len_b = strlen(b);
    char* result = malloc(len_a + len_b + 1);
    strcpy(result, a);
    strcat(result, b);
    return result;
}
```

Listing 7: Ejemplo de función de runtime

# 5 Decisiones de Diseño y Justificación

## 5.1 Generador de Lexer Personalizado

Se decidió implementar un generador de lexer personalizado en lugar de utilizar Flex por las siguientes razones:

- **Control total**: Permite optimizaciones específicas para el lenguaje Hulk
- **Integración**: Mejor integración con el sistema de manejo de errores
- **Flexibilidad**: Facilita la extensión del lenguaje con nuevos tokens
- **Rendimiento**: Eliminación de dependencias externas y optimización específica

## 5.2 Uso de LLVM como Backend

La elección de LLVM como backend de generación de código se justifica por:

- **Optimizaciones**: LLVM proporciona un conjunto robusto de optimizaciones
- **Portabilidad**: Generación de código para múltiples arquitecturas
- **Ecosistema**: Integración con herramientas de desarrollo existentes
- **Mantenibilidad**: API bien documentada y estable

## 5.3 Manejo de Memoria

Se implementó un esquema de manejo de memoria que combina:

- Asignación automática en stack para variables locales
- Asignación dinámica en heap para objetos y estructuras complejas
- Gestión manual de memoria para iteradores y rangos

# 6 Evaluación del Trabajo

## 6.1 Evaluación Cuantitativa

El compilador implementado consta de aproximadamente:

| Métrica | Valor |
|---|---|
| Líneas de código C++ | ∼8,500 |
| Archivos fuente | 45 |
| Archivos de cabecera | 25 |
| Casos de prueba | 15 |
| Funciones built-in | 20+ |
| Operadores soportados | 15 |
| Tipos de nodos AST | 30+ |
| Características implementadas | 95 % |

Cuadro 3: Métricas cuantitativas del compilador

## 6.2 Evaluación Cualitativa

### 6.2.1. Fortalezas de la Implementación

- **Completitud**: Implementación de todas las características principales del lenguaje Hulk
- **Robustez**: Manejo comprehensivo de errores en todas las fases con recuperación inteligente
- **Extensibilidad**: Arquitectura modular basada en patrones de diseño que facilita extensiones
- **Rendimiento**: Generación de código optimizado mediante LLVM con pases de optimización
- **Compatibilidad**: Integración con herramientas estándar de desarrollo y debugging
- **Mantenibilidad**: Código bien estructurado con separación clara de responsabilidades

### 6.2.2. Casos de Prueba Exitosos

El compilador maneja correctamente una amplia gama de programas complejos:

- **Herencia multinivel**: Jerarquías de tipos con hasta 5 niveles de herencia
- **Expresiones complejas**: Anidamiento profundo con precedencia correcta de operadores
- **Recursividad**: Funciones recursivas, mutuamente recursivas y tail recursion
- **Polimorfismo**: Dispatch dinámico de métodos con verificación de tipos
- **Iteradores avanzados**: Comprensiones anidadas con múltiples filtros
- **Manejo de memoria**: Programas con asignación dinámica compleja

# 7 Limitaciones y Trabajo Futuro

## 7.1 Limitaciones Identificadas

### 7.1.1. Manejo de Memoria

La implementación actual requiere gestión manual de memoria para ciertos constructos, lo que puede llevar a memory leaks en programas complejos. Una mejora futura sería implementar un recolector de basura automático.

#### 7.1.2. Optimizaciones

Aunque se aprovechan las optimizaciones de LLVM, el compilador no implementa optimizaciones específicas del lenguaje Hulk, como:

- Eliminación de verificaciones de tipos redundantes
- Inlining de métodos pequeños
- Optimización de tail calls

#### 7.1.3. Sistema de Módulos

La implementación actual no soporta un sistema de módulos, limitando la reutilización de código en proyectos grandes.

### 7.2 Propuestas de Mejora

#### 7.2.1. Recolección de Basura

Implementar un recolector de basura generacional que:

- Reduzca la carga del programador
- Mejore la seguridad de memoria
- Mantenga el rendimiento mediante técnicas modernas de GC

#### 7.2.2. Análisis Estático Avanzado

Incorporar análisis estático más sofisticado:

- Análisis de flujo de datos para optimizaciones
- Verificación de propiedades de seguridad
- Detección de código muerto

#### 7.2.3. Herramientas de Desarrollo

Desarrollar herramientas complementarias:

- Debugger integrado
- Language Server Protocol para IDEs
- Profiler de rendimiento

## 8 Conclusiones

Se logró implementar exitosamente un compilador completo y robusto para el lenguaje Hulk que supera los objetivos planteados inicialmente. La implementación demuestra un entendimiento profundo de los principios teóricos de construcción de compiladores y su aplicación práctica en un proyecto de envergadura considerable.

## 8.1 Logros Principales

- **Compilador funcional completo**: Implementación de todas las fases desde análisis léxico hasta generación de código ejecutable
- **Soporte integral del lenguaje**: 95 % de las características de Hulk implementadas correctamente
- **Arquitectura extensible**: Diseño modular que facilita futuras extensiones y mantenimiento
- **Rendimiento optimizado**: Generación de código eficiente mediante LLVM con optimizaciones avanzadas
- **Robustez**: Manejo comprehensivo de errores y casos edge en todas las fases

## 8.2 Contribuciones Técnicas

Las decisiones de diseño tomadas resultaron acertadas y contribuyeron significativamente al éxito del proyecto:

- **Generador de lexer personalizado**: Proporcionó control total sobre la tokenización y mejor integración con el sistema de errores
- **LLVM como backend**: Permitió aprovechar optimizaciones maduras y generar código portable de alta calidad
- **Sistema de contextos jerárquico**: Facilitó la implementación de alcances complejos y verificación semántica
- **Patrón Visitor para AST**: Proporcionó flexibilidad para implementar múltiples pases sobre el árbol sintáctico

## 8.3 Impacto Educativo

El proyecto ha proporcionado una experiencia educativa integral que abarca:

- Comprensión profunda de la teoría de lenguajes formales y autómatas
- Experiencia práctica con herramientas industriales (LLVM, Bison, CMake)
- Desarrollo de habilidades de diseño de software a gran escala
- Aplicación de patrones de diseño en contextos reales
- Optimización de rendimiento y manejo de memoria

## 8.4 Perspectivas Futuras

El proyecto representa una base sólida para futuras extensiones y mejoras. La arquitectura modular facilita la incorporación de:

- Implementación de protocolos
- Iterables
- Functors
- Macros

La experiencia adquirida durante el desarrollo proporciona una comprensión integral del proceso de construcción de compiladores, desde la especificación formal del lenguaje hasta la generación de código ejecutable optimizado, constituyendo una base sólida para futuros proyectos en el área de lenguajes de programación y sistemas de software.

## 9 Agradecimientos

Se agradece el apoyo y orientación recibidos durante el desarrollo de este proyecto, así como las valiosas discusiones técnicas que contribuyeron a mejorar la calidad de la implementación.

## Referencias

[1] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*. 2nd edn. Addison-Wesley, Boston (2006)

[2] Piad Morffis, A., Consuegra Ayala, J.P., Cruz Linares, R.: *Introducción a la Construcción de Compiladores*. Universidad de La Habana (2023)

[3] Nystrom, R.: *Crafting Interpreters*. Genever Benning (2021)

[4] Lattner, C.: *The LLVM Compiler Infrastructure*. `https://llvm.org/` (2012)

[5] Free Software Foundation: *Bison - GNU parser generator*. `https://www.gnu.org/software/bison/` (2020)

[6] Paxson, V., Estes, W., Millaway, J.: *Flex - The Fast Lexical Analyzer*. `https://github.com/westes/flex` (2008)

[7] Kitware Inc.: *CMake - Cross Platform Make*. `https://cmake.org/` (2021)