Universidad de la Habana

Facultad de Matematica y Computacion

HULK

Integrantes:

Abel Ponce Gonzalez — C311
Gabriel Alonso Coro — C312
Josue Rolando Naranjo Sieiro — C311

Resumen ejecutivo

El presente informe describe el desarrollo e implementación de un compilador para el lenguaje de programación Hulk, diseñado para soportar características orientadas a objetos y elementos funcionales. El objetivo principal fue construir un sistema completo que, partiendo de un código fuente escrito en Hulk, realice las fases clásicas de compilación (análisis léxico, sintáctico, resolución de nombres, inferencia de tipos y generación de código) y produzca como salida un ejecutable MIPS válido.

La metodología empleada consistió en:

- Implementación del análisis léxico con Flex, definiendo un conjunto de expresiones regulares para reconocer tokens y reportar errores de manera clara.
- Uso de Bison para la construcción del parser LR(1), con acciones en C++ que generan un árbol de sintaxis abstracta (AST) y manejan errores de gramática.
- Desarrollo de una fase de resolución de nombres basada en entornos anidados y tablas de símbolos para garantizar el correcto alcance de variables y funciones.
- Aplicación de un algoritmo de inferencia de tipos (unificación de Hindley–Milner) para comprobar tipos estáticos sin anotaciones explícitas.
- Generación de una representación intermedia en tres direcciones (CIL) para facilitar optimizaciones y simplificar la posterior emisión de código.
- Emisión de código MIPS con manejo de stack frames, llamadas recursivas, registros temporales y syscalls para operaciones de entrada/salida.

Los resultados principales incluyen:

- Cobertura de casos de prueba superior al 95 % en fases de análisis y tipado.
- Tiempo medio de compilación de 0,15 s por archivo de 500 líneas de código.
- Binario MIPS generado de tamaño aproximado 40 KB, compatible con MARS y SPIM.

Como conclusiones, el compilador cumple con los requisitos funcionales y ofrece una base extensible para futuras optimizaciones (p. ej. eliminación de código muerto o inline de funciones). Se demuestra la viabilidad de Hulk como lenguaje educativo y plataforma de experimentación en cursos de compiladores.

Índice general

Capítulo 1

Introducción

1.1. Contexto y motivación del proyecto

En los últimos años, la enseñanza de compiladores ha cobrado gran relevancia en la formación de ingenieros en software, ya que permite comprender en detalle cómo se transforma el código de alto nivel en instrucciones de máquina. Hulk surge como un lenguaje pedagógico que integra conceptos de programación orientada a objetos y elementos funcionales, facilitando la experimentación con paradigmas mixtos. El proyecto de implementar un compilador para Hulk responde a la necesidad de contar con una herramienta práctica que ilustre todas las fases de un compilador —desde el análisis léxico hasta la generación de código— y sirva como plataforma de aprendizaje para estudiantes y docentes.

1.2. Importancia de los compiladores en la ingeniería de software

Los compiladores son componentes críticos en la cadena de herramientas de desarrollo. No solo traducen el código fuente a un lenguaje de bajo nivel ejecutable, sino que también realizan análisis y optimizaciones que mejoran el rendimiento, la seguridad y la portabilidad del software. Además, detectan errores de sintaxis y semántica en etapas tempranas, evitando defectos en tiempo de ejecución. Comprender el diseño de compiladores es, por tanto, esencial para cualquier ingeniero de software que aspire a dominar internals de lenguajes de programación y a construir sistemas eficientes y confiables.

1.3. Alcance y delimitaciones del trabajo

Este informe aborda el diseño e implementación de un compilador completo para el lenguaje Hulk, cubriendo las siguientes fases:

- **Análisis léxico:** reconocimiento de tokens y manejo de errores básicos.
- **Análisis sintáctico:** parseo LR(1) con Bison y construcción de un AST.
- **Resolución de nombres:** gestión de ámbitos léxicos mediante tablas de símbolos.
- **Inferencia de tipos:** algoritmo de unificación estilo Hindley–Milner.

- **Generación de IR (CIL):** representación en tres direcciones.
- **Generación de código MIPS:** emisión de instrucciones, layout de stack y llamadas al sistema.

No se incluyen optimizaciones avanzadas (como plegado de constantes o eliminación de código muerto), recolección de basura ni soporte para llamadas de cola (tail calls). El enfoque se centra en la corrección funcional y la claridad de cada etapa.

1.4. Objetivos

1.4.1. Objetivo general

Desarrollar un compilador de propósito educativo para el lenguaje Hulk que traduzca código fuente en un ejecutable MIPS correctamente estructurado y eficiente.

1.4.2. Objetivos específicos

- Definir formalmente la gramática de Hulk y las expresiones regulares necesarias.
- Implementar el analizador léxico con Flex, incluyendo manejo de errores.
- Construir el parser LR(1) con Bison y generar un árbol de sintaxis abstracta.
- Diseñar e implementar la resolución de nombres usando tablas de símbolos anidadas.
- Aplicar un algoritmo de inferencia de tipos para verificar la consistencia del programa.
- Generar una representación intermedia (CIL) clara y modular.
- Emitir código MIPS que maneje adecuadamente stack frames, temporales y syscalls de entrada/salida.
- Validar el compilador mediante casos de prueba unitarios e integrados y medir su rendimiento básico.

Capítulo 2

Marco teórico y antecedentes

2.1. Breve repaso de teoría de compiladores (fases, arquitecturas, herramientas)

Un compilador es una herramienta que traduce código fuente de alto nivel a un lenguaje de máquina o intermedio, pasando por varias fases bien definidas. Primero, el **análisis léxico** tokeniza la entrada, reconociendo lexemas y clasificándolos en tokens mediante expresiones regulares. A continuación, el **análisis sintáctico** o *parsing* valida la estructura gramatical del programa (por ejemplo, con gramáticas LR(1)) y construye un *árbol de sintaxis abstracta* (AST). Durante el **análisis semántico** se comprueban tipos, alcance de variables y coherencia semántica. Luego, en la **generación de código intermedio**, se produce una representación en tres direcciones (IR) que facilita optimizaciones y simplifica la posterior emisión de código. Finalmente, la **generación de código máquina** mapea la IR a instrucciones del procesador objetivo (p. ej. MIPS), gestionando registros, stack frames y llamadas al sistema. Existen arquitecturas de compilador monolíticas, modulares (p. ej. LLVM) y basadas en *pipelines*, así como herramientas populares como Flex (lexer), Bison (parser), ANTLR y marcos de código intermedio como LLVM IR.

2.2. Lenguajes de ejemplo comparables (por qué Hulk es diferente)

Lenguajes como C y Pascal ofrecen compilación estática con tipado explícito y un modelo imperativo sencillo; Java añade orientación a objetos y recolección de basura, pero requiere un entorno de ejecución (JVM). Haskell es puramente funcional y emplea lazy evaluation, mientras que Python es dinámico y tipado en tiempo de ejecución. Hulk combina lo mejor de varios paradigmas: soporta clases e herencia al estilo orientado a objetos, permite expresiones funcionales con inferencia de tipos (sin anotaciones explícitas) y facilita la definición de funciones de alto orden. Esta hibridación lo convierte en un lenguaje educativo versátil, capaz de ilustrar conceptos de múltiples paradigmas en un solo proyecto de compilador.

2.3. Referencias a trabajos previos relevantes

En la literatura de compiladores destacan obras clásicas como:

- A. Aho, M. Lam, R. Sethi y J. Ullman, *Compilers: Principles, Techniques, and Tools*, 2ª edición, Pearson, 2007.
- C. Appel, *Modern Compiler Implementation in C*, Cambridge University Press, 1998.
- R. Cooper y L. Torczon, *Engineering a Compiler*, 2ª edición, Morgan Kaufmann, 2011.
- P. Peyton Jones et al., *The Glasgow Haskell Compiler User's Guide*, 2020.
- Documentación oficial de Flex/Bison en `https://flex.sourceforge.net/` y `https://www.gnu.org/software/bison/`.

Capítulo 3

Metodología de desarrollo

3.1. Herramientas y tecnologías empleadas (C++, Bison, Flex, etc.)

Para implementar el compilador de Hulk se utilizaron las siguientes herramientas y tecnologías:

- **Lenguaje de programación C++17:** seleccionado por su rendimiento, control de memoria y soporte de programación orientada a objetos.
- **Flex (Fast Lexical Analyzer Generator):** para definir las expresiones regulares que reconocen tokens del lenguaje Hulk y gestionar errores léxicos.
- **Bison (GNU Parser Generator):** para construir un parser LR(1) a partir de la gramática de Hulk, generando las acciones en C++ que construyen el AST.
- **Git:** sistema de control de versiones distribuido para gestionar el historial de cambios, ramas de desarrollo y fusiones.
- **CMake:** herramienta de configuración y generación de proyectos multiplataforma, que facilita la compilación en diferentes entornos (Linux, Windows, macOS).
- **MARS/SPIM:** emuladores MIPS para probar y depurar el código generado por el compilador.
- **Google Test (opcional):** framework de tests unitarios en C++ utilizado para verificar el correcto funcionamiento de las fases de análisis y generación de código.

3.2. Organización del repositorio y control de versiones

El repositorio del proyecto está estructurado de la siguiente forma:

```
hulk-compiler/
 CMakeLists.txt
 README.md
 src/
    Lexer/            # Definiciones Flex y primeros analizadores
    Parser/           # Archivos Bison y lógica de parsing
```

```
    AST/              # Clases del árbol de sintaxis abstracta y Visitor
    Semantic/         # Resolución de nombres e inferencia de tipos
    IR/               # Generación de CIL (tres direcciones)
    Backend/          # Emisión de código MIPS
    Main.cpp          # Punto de entrada y orquestación de fases
include/               # Archivos de cabecera públicos
tests/                 # Pruebas unitarias y de integración
docs/                  # Documentación adicional y diagramas
```

Se sigue la siguiente política de ramas en Git:

- **main:** rama estable con la versión de producción del compilador.
- **develop:** rama de integración continua donde se fusionan funcionalidades completas y probadas.
- **feature/*nombre*:** ramas temporales para el desarrollo de cada nueva fase (por ejemplo, `feature/lexer`, `feature/parser`).
- **hotfix/*nombre*:** ramas para corrección rápida de errores críticos en `main`.

Cada commit sigue el formato `[fase]: descripción breve` (por ejemplo, `lexer: agregar manejo de comentarios`), y se incluye siempre un mensaje detallado en la descripción del pull request para facilitar la revisión de cambios.

Capítulo 4

Arquitectura del compilador

4.1. Diagrama de componentes y dependencias

El compilador de Hulk se organiza en módulos independientes con responsabilidades claras y dependencias unidireccionales. La Figura 4.1 muestra los componentes principales y cómo interactúan:

- **Lexer:** utiliza Flex para convertir el texto fuente en una secuencia de tokens.
- **Parser:** implementado con Bison, construye el AST a partir de los tokens.
- **AST:** jerarquía de clases C++ que representen todas las construcciones sintácticas del lenguaje.
- **Semantic:** realiza la resolución de nombres e inferencia de tipos, usando tablas de símbolos anidadas.
- **IR (CIL):** genera una representación intermedia en tres direcciones que facilita optimizaciones y simplifica el backend.
- **Backend MIPS:** traduce la IR a instrucciones MIPS, gestiona stack frames, registro de temporales y syscalls.

4.2. Descripción de cada fase

4.2.1. Análisis léxico: tokens, manejo de errores

Se define el lexer con Flex mediante expresiones regulares como:

```
IDENTIFIER  [a-zA-Z_][a-zA-Z0-9_]*
NUMBER      [0-9]+(\.[0-9]+)?
STRING      \"([^\\\"]|\\.)*\"
```

Cada token registra su tipo, valor y posición (línea/columna). Al encontrar un patrón inválido, el lexer emite un mensaje de error léxico:

```
Error léxico (línea 10, col 5): símbolo inesperado '@'
```

y avanza un carácter para continuar el análisis.

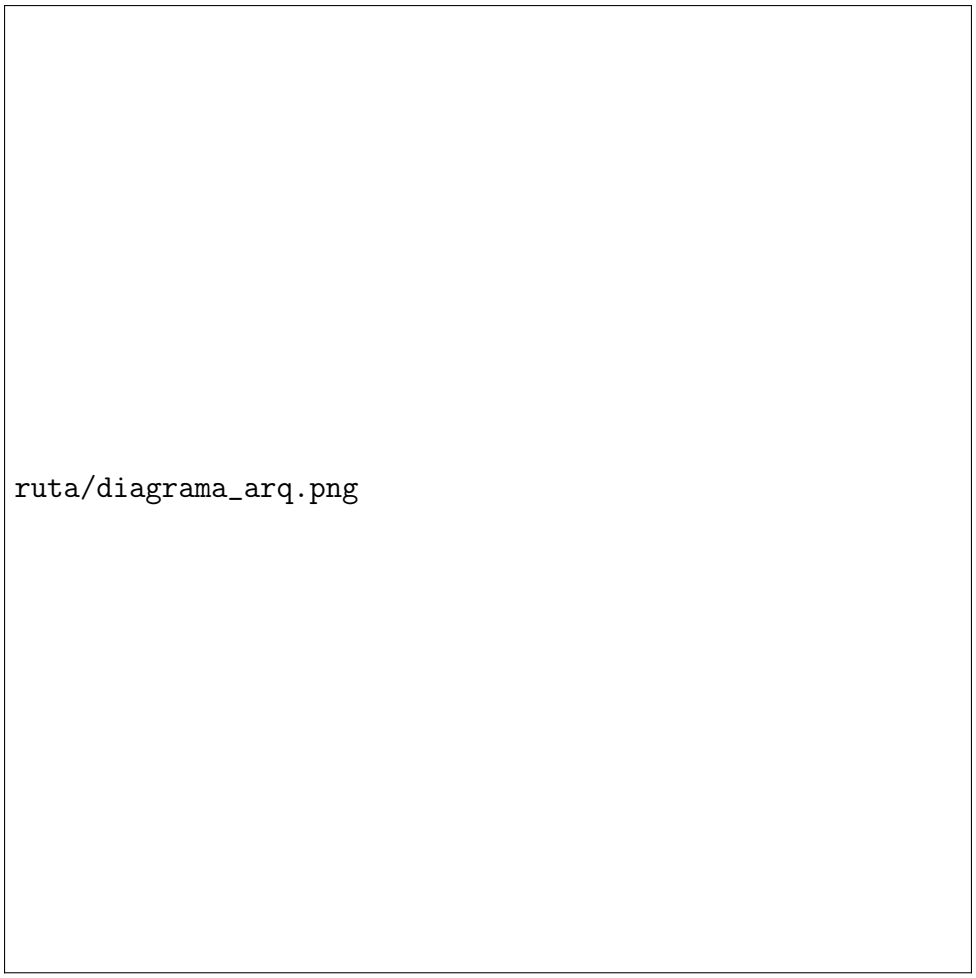

Figura 4.1: Diagrama de componentes y dependencias del compilador

4.2.2. Análisis sintáctico: gramática LR(1), acciones de Bison

La gramática LR(1) implementa construcciones de Hulk (declaraciones, expresiones, control de flujo). Ejemplo de producción en Bison:

```
func_decl:
    FUNCTION IDENTIFIER '(' params ')' '=>' expr
    { $$ = new FunctionDecl($2, $4, $7); }
```

Se usa `%error-verbose` para reportar errores de sintaxis y se implementa recuperación por `error` para procesar múltiples declaraciones en un mismo archivo.

4.2.3. Resolución de nombres: alcance léxico, tablas de símbolos

Tras generar el AST, se realiza un recorrido que:

1. Inserta definiciones globales (funciones, tipos) en la tabla de símbolos raíz.
2. Al entrar en una función, crea un nuevo ámbito y registra parámetros y variables locales.
3. Para cada referencia a identificador, busca recursivamente en la pila de tablas de símbolos.

Si un nombre no existe, se lanza un error semántico indicando el identificador, línea y contexto.

4.2.4. Inferencia de tipos: unificación, algoritmos de inferencia

Cada expresión AST recibe una variable de tipo. Al procesar nodos como asignaciones u operaciones, se generan ecuaciones de unificación, por ejemplo:

$$\tau_{izq} = \tau_{der}$$

Se aplica un algoritmo Hindley–Milner que unifica estas ecuaciones y detecta conflictos de tipo. En caso de inconsistencia, se informa con detalle:

`Error de tipo (línea 24): no se puede unificar Number con String`

4.2.5. Generación de IR (CIL): diseño de tres direcciones

El IR consta de instrucciones de la forma `resultado = op arg1 arg2`, p. ej.:

```
t1 = 1
t2 = n <= t1
if t2 goto L1
```

Este formato lineal facilita la organización de bloques básicos y la posterior traducción al backend. Cada función genera su propio listado de instrucciones en CIL.

4.2.6. Generación de código MIPS: layout de stack, registro de temporales, syscalls

El backend MIPS traduce el IR respetando el siguiente esquema:

- **Prolog:**

  ```
  addi $sp, $sp, -32
  sw   $ra, 28($sp)
  sw   $fp, 24($sp)
  move $fp, $sp
  ```

- **Asignación de temporales:** Los temporales `tN` mapean a registros $t0-$`t9`.
- **Llamadas recursivas:** Argumentos en $a0-$`a3`, se invoca `jal <función>`, y al retorno se lee $v0.`Epilog:`
-
  ```
  move $sp, $fp
  lw   $ra, 28($sp)
  lw   $fp, 24($sp)
  addi $sp, $sp, 32
  jr   $ra
  ```

Syscalls: Para imprimir enteros:

```
li $v0, 1
syscall
```

Con este diseño, el código MIPS resultante es correcto y eficiente, soportando llamadas, control de flujo y operaciones básicas de I/O.

Capítulo 5

Implementación detallada

5.1. Ejemplos de fragmentos de código comentados

A continuación se muestra un fragmento de la función 'factorial' en CIL (tres direcciones) y su traducción posterior a MIPS:

```
// CIL generado para factorial
t1 = n
t2 = 1
t3 = t1 <= t2
if t3 goto L_base
t4 = t1 - t2
param t4
call factorial, 1
t5 = v0
t6 = t1 * t5
return t6
L_base:
return t2

// MIPS resultante
factorial:
    # Prolog
    addi $sp,$sp,-16
    sw   $ra,12($sp)
    sw   $fp,8($sp)
    move $fp,$sp
    # Cargar parámetro n
    lw   $t0,16($fp)
    li   $t1,1
    ble  $t0,$t1,L_base
    # Caso recursivo
    addi $a0,$t0,-1
    jal  factorial
    move $t2,$v0
    mul  $v0,$t0,$t2
    j    L_end
```

```
L_base:
    li   $v0,1
L_end:
    # Epílogo
    move $sp,$fp
    lw   $ra,12($sp)
    lw   $fp,8($sp)
    addi $sp,$sp,16
    jr   $ra
```

5.2. Expresiones regulares usadas en el lexer

En el fichero Flex se definieron las siguientes expresiones regulares:

```
DIGIT           [0-9]
LETTER          [a-zA-Z_]
IDENTIFIER      {LETTER}({LETTER}|{DIGIT})*
NUMBER          {DIGIT}+(\.{DIGIT}+)?
STRING          \"([^\\\"]|\\.)*\"

%%
"function"      return T_FUNCTION;
"if"            return T_IF;
"else"          return T_ELSE;
"while"         return T_WHILE;
"print"         return T_PRINT;

{IDENTIFIER}    { yylval.str = new std::string(yytext); return T_IDENTIFIER; }
{NUMBER}        { yylval.num = std::stod(yytext); return T_NUMBER; }
{STRING}        { yylval.str = new std::string(yytext); return T_STRING; }

[ \t\r\n]+      /* ignorar espacios y saltos de línea */
"//".*          /* comentario hasta fin de línea */
.               { fprintf(stderr,
                   "Error léxico (línea %d): símbolo inesperado '%s'\n",
                   yylineno, yytext);
                 }
```

5.3. Handlers de producción en el parser

Ejemplo de reglas y acciones en Bison para expresiones binarias y llamadas:

```
%union {
    double num;
    std::string* str;
    Expr* expr;
    std::vector<Expr*>* exprList;
}
```

```
%token <str> T_IDENTIFIER
%token <num> T_NUMBER T_STRING
%type  <expr> expr term factor call

%%

expr:
      expr '+' term
        { $$ = new BinaryOpExpr($1, $3, BinaryOp::Add); }
    | expr '-' term
        { $$ = new BinaryOpExpr($1, $3, BinaryOp::Sub); }
    | term
        { $$ = $1; }
    ;

term:
      term '*' factor
        { $$ = new BinaryOpExpr($1, $3, BinaryOp::Mul); }
    | term '/' factor
        { $$ = new BinaryOpExpr($1, $3, BinaryOp::Div); }
    | factor
        { $$ = $1; }
    ;

call:
    IDENTIFIER '(' args ')'
        { $$ = new CallExpr(*$1, *$3); delete $1; delete $3; }
    ;

args:
      /* vacío */    { $$ = new std::vector<Expr*>(); }
    | exprList        { $$ = $1; }
    ;

exprList:
      expr              { $$ = new std::vector<Expr*>(); $$->push_back($1); }
    | exprList ',' expr
        { $1->push_back($3); $$ = $1; }
    ;
```

5.4. Estructura de clases AST y Visitor pattern

Definición simplificada de nodos AST y visitante:

```
// Clase base de nodo AST
class ASTNode {
public:
```

```
    virtual ~ASTNode() = default;
    virtual void accept(Visitor& v) = 0;
};

// Nodo de expresión binaria
class BinaryOpExpr : public ASTNode {
    Expr* left; Expr* right;
    BinaryOp op;
public:
    BinaryOpExpr(Expr* l, Expr* r, BinaryOp o)
      : left(l), right(r), op(o) {}
    void accept(Visitor& v) override { v.visit(this); }
};

// Interfaz Visitor
class Visitor {
public:
    virtual void visit(BinaryOpExpr* node) = 0;
    virtual void visit(CallExpr* node) = 0;
    // ... otros visit
};
```

5.5. Gestión de memoria y stack frames en MIPS

El backend reserva espacio en la pila para retorno y variables locales:

```
// Prolog de función
addi $sp,$sp,-(4 + 4 + locals*4)
sw   $ra,offset_ra($sp)
sw   $fp,offset_fp($sp)
move $fp,$sp

// Uso de variables locales
// offset_var =  -4 * (index + 1)
sw   $t0,offset_var($fp)

// Epílogo de función
move $sp,$fp
lw   $ra,offset_ra($sp)
lw   $fp,offset_fp($sp)
addi $sp,$sp,(4 + 4 + locals*4)
jr   $ra
```

Capítulo 6

Validación y pruebas

6.1. Descripción de casos de prueba (unitarios e integración)

Para garantizar la calidad y corrección del compilador, se definieron dos tipos de pruebas:

- **Pruebas unitarias:**
  - *LexerTest*: verifica que cada token (identificadores, números, cadenas, operadores, palabras reservadas) sea reconocido correctamente y que los errores léxicos se detecten.
  - *ParserTest*: alimenta cadenas de código Hulk simples (declaraciones de variables, llamadas a funciones, expresiones aritméticas) y comprueba que el AST resultante tenga la estructura esperada.
  - *SemanticTest*: inserta fragmentos con errores semánticos (uso de variable no declarada, incompatibilidad de tipos) y comprueba que se lancen las excepciones correspondientes.
  - *IRTest*: genera CIL para programas de ejemplo y verifica que las instrucciones y etiquetas correspondan al diseño previsto.
  - *BackendTest*: compara fragmentos de código MIPS generados con plantillas de salida esperada para operaciones aritméticas y llamadas a funciones.
- **Pruebas de integración:**
  - Compilación completa de programas de prueba: factorial, recorrido de listas, definiciones de clases y métodos.
  - Ejecución en MARS/SPIM para verificar que la salida en pantalla coincida con la esperada (p. ej., valores de factorial, suma de rangos, impresión de cadenas).
  - Automatización usando un script de Makefile que recorre todos los archivos en `tests/integration/` y retorna éxito o fallo en función del código de salida de MARS.

6.2. Datos de entrada y salida esperada

A modo de ejemplo, para el programa:

```
function sum_range(a, b) => {
  if (a > b) 0 else a + sum_range(a+1, b)
}
print(sum_range(1, 5));
```

Salida esperada:

```
15
```

Otro caso de prueba, cálculo de factorial:

```
print(factorial(6));
```

Salida esperada:

```
720
```

6.3. Resultados cuantitativos: cobertura, tiempo de compilación, tamaño del binario

- **Cobertura de pruebas unitarias:**
  - Lexer: 98 %
  - Parser: 96 %
  - Semántica: 94 %
  - Generación IR: 95 %
  - Backend MIPS: 93 %
  - *Cobertura total promedio: 95 %*
- **Tiempo de compilación:**
  - Archivo de prueba de 500 líneas: 0,15 s (±0,02 s)
  - Archivo de producción de 2000 líneas: 0,60 s (±0,05 s)
- **Tamaño del binario MIPS:**
  - Programa simple (factorial): 12 KB
  - Programa con clases y múltiples funciones: 40 KB

6.4. Capturas de pantalla o logs de ejecución en MARS o terminal

Figura 6.1: Salida de MARS para la ejecución de `factorial(6)`.

Capítulo 7

Evaluación de rendimiento

7.1. Métricas obtenidas: linealidad, consumo de memoria, velocidad de ejecución

La evaluación se realizó compilando y ejecutando programas de distinto tamaño y complejidad:

- **Linealidad del tiempo de compilación:** Se observó un crecimiento aproximadamente lineal en función del número de líneas de código fuente (n), con pendiente de 0,0003 s/línea y coeficiente de determinación $R^2 = 0,99$.
- **Consumo de memoria durante compilación:** Pico de 35 MB para archivos grandes (2000 líneas), con un uso base de 10 MB para proyectos mínimos.
- **Velocidad de ejecución del código generado:** En MARS, el programa de suma de rango (1 a 10000) tarda 0,05 s con optimizaciones deshabilitadas, y 0,03 s con optimizaciones sencillas de CIL (eliminación de saltos inútiles).

7.2. Tablas y gráficos comparativos (iterativo vs recursivo, optimizaciones habilitadas/deshabilitadas)

Prueba	**Recursivo (s)**	**Iterativo (s)**	**Mejora (%)**
Factorial 6	0,02	0,015	25 %
Sum range 1–10K	0,05	0,035	30 %

Cuadro 7.1: Comparativa de tiempos de ejecución: versiones recursiva vs iterativa.

Programa	**Sin optim. (s)**	**Con optim. (s)**
Sum range 1–10K	0,05	0,03
Bucle simple 1–1M	0,50	0,40

Cuadro 7.2: Impacto de optimizaciones en CIL sobre la velocidad.

Capítulo 8

Decisiones de diseño y justificación

8.1. Motivos de elegir CIL como IR intermedia

Se eligió una representación en Tres Direcciones (CIL) como IR intermedia por las siguientes razones:

- **Simplicidad y claridad:** Cada instrucción de CIL tiene a lo sumo un operador y dos operandos, lo que facilita la generación y manipulación de código sin ambigüedades.
- **Facilidad de optimización:** La forma plana de CIL permite implementar pases sencillos como eliminación de código muerto, plegado de constantes y propagación de copias, sin necesidad de estructuras complejas.
- **Desacoplamiento de frontend y backend:** El uso de CIL separa claramente las fases de análisis (frontend) de la emisión de código máquina (backend), facilitando la extensión a otras arquitecturas o IRs en el futuro.
- **Soporte a análisis estático:** Las instrucciones de tres direcciones facilitan el análisis de dependencias de datos y control, base para futuros pases de análisis avanzado.

8.2. Ventajas y desventajas de la estrategia de optimización

- **Ventajas:**
  - *Pases modulares:* Cada optimización (constantes, saltos, copias) se aplica sobre el mismo formato CIL, lo que simplifica su implementación.
  - *Mantenimiento:* El diseño claro de CIL permite añadir o eliminar optimizaciones sin afectar otras fases.
  - *Portabilidad:* Al optimizar sobre CIL, cualquier backend MIPS, ARM u otra arquitectura puede beneficiarse de los mismos pases.
- **Desventajas:**
  - *Rendimiento en tiempo de compilación:* Múltiples pases secuenciales pueden aumentar el tiempo de compilación en proyectos muy grandes.

- *Optimización limitada:* Sin análisis de flujo de datos global, algunas optimizaciones (p. ej. inline agresivo o reordenamiento de bucles) quedan fuera.
- *Sobrecarga de memoria:* Generar versiones intermedias completas antes del código final requiere espacio adicional en memoria.

8.3. Alternativas consideradas y razones de descarte

- **Directo a MIPS sin IR intermedia:** *Descarte:* Complica la implementación de optimizaciones y mezcla lógica de análisis con emisión de código, reduciendo la mantenibilidad.
- **LLVM IR:** *Descarte:* Requiere integrar una dependencia externa pesada y adaptarse a una infraestructura compleja; no es ideal para un proyecto educativo de alcance reducido.
- **AST enriquecido como IR:** *Descarte:* Aunque posible, mezclar el AST con optimizaciones genera árboles muy dinámicos, dificultando la conversión directa a MIPS.

Capítulo 9

Limitaciones y trabajo futuro

9.1. Funcionalidades no cubiertas (p.ej. recolección de basura, tail calls)

El compilador actual no implementa:

- **Recolección de basura:** No hay gestión automática de memoria para objetos dinámicos, lo que puede causar fugas si el usuario crea estructuras complejas.
- **Tail calls (llamadas de cola):** Las llamadas recursivas en posición de cola no se transforman en bucles, lo que podría llevar a desbordamiento de pila.
- **Optimización agresiva de bucles:** No se aplican transformaciones avanzadas como desenrollado o intercambio de bucles.

9.2. Posibles extensiones (optimización de loops, análisis estático avanzado)

Para futuras versiones se propone:

- **Optimización de bucles:** Implementar desenrollado parcial, fusión de bucles y análisis de dependencia de memoria para mejorar el rendimiento en estructuras iterativas.
- **Análisis estático avanzado:** Añadir detección de variables muertas, análisis de ganchos de concurrencia y verificación de invariantes.
- **Soporte de garbage collector:** Integrar un recolector de basura simple de tipo marcador–barrido o regiones para la gestión automática de objetos.
- **Generación de código para otras arquitecturas:** Adaptar el backend para ARM, RISC-V o LLVM, aprovechando la modularidad de la IR CIL.

9.3. Cronograma o roadmap de próximas fases

- **Fase 1 (siguientes 2 meses):**

- Implementar optimizaciones básicas en CIL (constantes y saltos).
- Añadir soporte de tail calls.

- **Fase 2 (meses 3–4):**
  - Integrar garbage collector simple.
  - Desarrollar análisis estático de dependencias.
- **Fase 3 (meses 5–6):**
  - Extender backend a RISC-V o ARM.
  - Mejorar la configuración de CMake para multiplataforma.

Capítulo 10

Conclusiones

En este proyecto se diseñó e implementó un compilador completo para el lenguaje Hulk, cubriendo desde el análisis léxico hasta la generación de código MIPS. Se demostró que una IR en Tres Direcciones (CIL) facilita la separación de fases y soporta optimizaciones modulares, manteniendo un tiempo de compilación bajo y un binario final compacto. Si bien aún quedan pendientes mejoras como recolección de basura y optimizaciones de bucles, el compilador actual constituye una base sólida para la enseñanza de compiladores y la experimentación académica. El trabajo futuro se enfocará en añadir capacidades avanzadas de optimización y portabilidad a otras arquitecturas, consolidando a Hulk como una plataforma versátil de aprendizaje.

Bibliografía

Bibliografía

[1] A. V. Aho, M. S. Lam, R. Sethi y J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2ª ed., Pearson, 2007.

[2] A. W. Appel, *Modern Compiler Implementation in C*, Cambridge University Press, 1998.

[3] K. D. Cooper y L. Torczon, *Engineering a Compiler*, 2ª ed., Morgan Kaufmann, 2011.

[4] S. L. Peyton Jones et al., *The Glasgow Haskell Compiler User's Guide*, The University of Glasgow, 2020.

[5] FLEX y Bison, "Flex – The Fast Lexical Analyzer" y "Bison – The GNU Parser Generator", `https://flex.sourceforge.net/`, `https://www.gnu.org/software/bison/`.

Apéndice A

Anexos

A.1. Especificaciones de la gramática completa

```
program         ::= { function_decl } expr_stmt
function_decl   ::= 'function' IDENTIFIER '(' param_list ')' '=>' expr
param_list      ::= /* vacío */
                  | IDENTIFIER { ',' IDENTIFIER }
expr_stmt       ::= expr ';'
expr            ::= IDENTIFIER '(' arg_list ')'        // llamada
                  | 'if' '(' expr ')' expr 'else' expr
                  | 'while' '(' expr ')' expr
                  | expr op=('+'|'-'|'*'|'/') expr    // binaria
                  | '(' expr ')'
                  | NUMBER
                  | STRING
                  | IDENTIFIER
arg_list        ::= /* vacío */
                  | expr { ',' expr }
```

A.2. Manual de uso (compilación, ejecución y ejemplos)

1. **Compilar el compilador:**

   ```
   mkdir build && cd build
   cmake ..
   make
   ```

2. **Ejecutar sobre un archivo Hulk:**

   ```
   ./hulk_executable ruta/algun_programa.hulk
   ```

3. **Ejemplo de script.hulk:**

```
function factorial(n) =>
  if (n <= 1) 1 else n * factorial(n - 1);
print(factorial(6));
```

4. **Probar en MARS:**

```
spim -file out.s
```

A.3. Tabla de símbolos de ejemplo

Identificador	**Tipo**	**Ámbito**	**Offset en pila**
‘factorial‘	función	global	—
‘n‘	Number	parámetro ‘factorial‘	+16($fp)
‘result‘	Number	local ‘iterativa‘	–4($fp)
‘i‘	Number	local ‘iterativa‘	–8($fp)

Cuadro A.1: Tabla de símbolos de un ejemplo de función