
# H.U.L.K

Diego Viera, Pablo Gómez, Luis Alejandro Arteaga

June 2025

# Índice

# 1. Introdución

Este documento presenta el diseño e implementación de un compilador desarrollado en **Rust**, utilizando **LLVM** para la generación de código intermedio y finalmente compilar usando **clang**. El proyecto abarca las fases esenciales de la construcción de compiladores: desde el análisis léxico-sintáctico hasta la generación de código, aplicando técnicas avanzadas estudiadas en el curso.

El proceso de compilación para HULK sigue una arquitectura por capas que inicia con el archivo principal `main.rs`. Este componente realiza secuencialmente las siguientes operaciones:

1. Lectura del código fuente desde `script.hulk` como cadena de texto
2. Ejecución del `parser` para construir el Árbol de Sintaxis Abstracta (`AST`) (proceso de lexer en runtime del parser).

Tras superar las fases de análisis sin errores léxicos o sintácticos, el flujo continúa con:

- Validación semántica mediante recorrido del `AST` desde su nodo raíz
- Generación de código intermedio(`LLVM-IR`) tras verificación exitosa
- Construcción del compilado final usando `clang`.

El ciclo de compilación finaliza con la generación del archivo `hulk/output.ll`. La gestión del proceso se simplifica mediante un Makefile con comandos específicos:

```
# Compila el codigo fuente y genera output.ll
make compile

# Ejecuta el compilador LLVM sobre el codigo generado
make execute

# Elimina artefactos de compilacion
make clean
```

Este diseño modular permite una transición fluida entre análisis, validación y generación de código, manteniendo una estructura de proyecto clara y mantenible.

# 2. Generador de Lexer Basado en Autómatas

Inspirado en los principios establecidos en *Compilers: Principles, Techniques, and Tools* (el libro del dragón), el generador de lexer de este proyecto se construyó mediante una secuencia de etapas estructuradas que reflejan el flujo clásico del análisis léxico en compiladores. A continuación, se describen los componentes principales y los algoritmos asociados en cada fase del proceso.

## 2.1. Construcción del AST de Expresiones Regulares

El proceso comienza con la definición de un conjunto de nodos del Árbol de Sintaxis Abstracta (AST) para representar expresiones regulares. Estos nodos modelan operadores básicos como:

- **Unión**
- **Clausura**
- **Clausura positiva**
- **Estrella de Kleene**
- **Opcional**
- **Wildcard**
- **Conjunto**
- **Símbolos atómicos** (caracteres literales, rangos, clases de caracteres)

Un *parser* utilizamos **lalrpop** para generar un parser especializado y construir nuestro AST

## 2.2. Generación de Autómatas Finitos No Deterministas (NFA)

Una vez obtenido el AST, se aplica el algoritmo para convertir cada expresión regular en un autómata finito no determinista (NFA). Este método asegura que cada subexpresión se traduzca en una máquina que acepte exactamente el lenguaje definido por ella. Se realizaron test para su chequeo.

## 2.3. Composición de un NFA Global

Luego de construir un NFA por cada expresión regular/token, se unifican en un NFA global que permite identificar múltiples tipos de tokens en una sola pasada. Esto se logra añadiendo un nuevo estado inicial con transiciones $\epsilon$ hacia los estados iniciales de cada NFA individual y se establece una precedencia entre los estados de aceptación, con el fin de resolver conflictos en casos de prefijos comunes, como es el caso de (a) y (a*b).

## 2.4. Conversión de NFA a DFA

El NFA global se convierte en un autómata determinista (DFA) mediante el algoritmo de *construcción del subconjunto*, también conocido como *powerset construction*. Este algoritmo genera un conjunto de estados DFA, donde cada estado representa un conjunto de estados del NFA original.

Cada transición del DFA corresponde a leer un símbolo y moverse al conjunto de estados accesibles desde los actuales (incluyendo la cerradura $\epsilon$).

### 2.5. Tokenización con el DFA

Una vez construido el DFA, se utiliza para recorrer el texto de entrada carácter por carácter. En cada paso:

1. Se avanza en el DFA según el símbolo leído.
2. Se registra el último estado de aceptación alcanzado (si lo hay).
3. Cuando no hay transición válida:
    - Se genera un token si hubo un estado final válido.
    - Se retrocede hasta ese punto.
    - Se reinicia el DFA desde el estado inicial.

Esto permite realizar la segmentación léxica del texto de manera eficiente, en tiempo lineal.

**Algoritmo clave:** simulación de DFA con estrategia de *maximal munch* (registro del estado final más reciente).

## 3. Análisis Léxico

**Nota:** Aunque en el repositorio del compilador está implementado un generador de lexer, este no está integrado con el compilador y cumple solo el propósito de demostrar dicho conocimiento. En la sección actual se explica el proceso de lexing usado para el flujo principal del compilador funcional.

A pesar de que el compilador realizado no utiliza un lexer externo como `Logos`, `LALRPOP` puede generar **internamente** un lexer a partir de las expresiones regulares definidas en `parser.lalrpop`, no se ha tomado la decisión de implementar el lexer por separado sino que esto ocurra durante el mismo parsing. Este proceso funciona así:

- **Definición de terminales:** Al comienzo del archivo, se definen terminales como `Identifier`, `Num`, `Str`, etc., utilizando regexes (por ejemplo, `r"[A-Za-z][A-Za-z_0-9]*"` para identificadores) y literales (por ejemplo, `"function"` para palabras clave).
- **Generación automática del lexer:** LALRPOP toma todas esas definiciones y genera un lexer que, en tiempo de ejecución, escanea el input caracter por caracter, buscando siempre la coincidencia más larga entre patrones; si un literal y un regex empatan en longitud, el literal tiene prioridad. .
- **Proceso de tokenización:** Se omiten espacios y comentarios. El lexer produce tokens como `(String, Span)` o `(OperatorToken, Span)` según

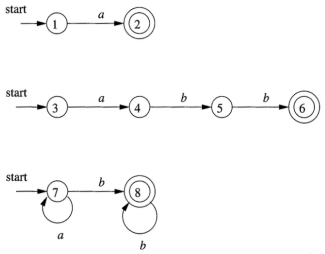


Figure 3.51: NFA's for **a**, **abb**, and **a**$^*$**b**$^+$

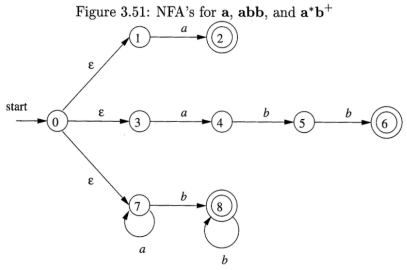


Figure 3.52: Combined NFA

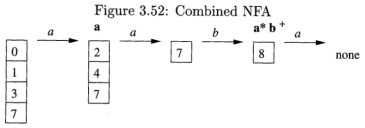


Figura 1: NFA

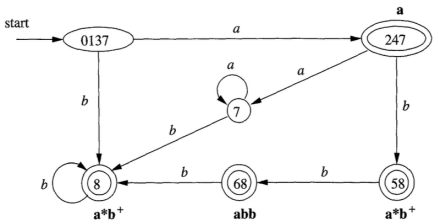


Figura 2: DFA

las reglas. Los literales, palabras reservadas, símbolos de operadores, regexes, actúan como terminales, es decir operan sin conocimiento del resto de la gramática: sólo agrupan texto en tokens .

# 4. Tokens y posicionamiento con Span

El archivo `tokens.rs` define los distintos tipos de tokens que se utilizan durante el análisis léxico y sintáctico. Estos tokens se agrupan en tres enumeraciones principales:

- `KeywordToken`: representa palabras clave del lenguaje como `if, while, function`, entre otras.
- `OperatorToken`: incluye todos los operadores del lenguaje, como `+, -, ==, :=,` etc.
- `DelimiterToken`: agrupa los símbolos de puntuación como paréntesis, comas, puntos y coma, y flechas (`=>`).

## 4.1. Span

Cada token es acompañado por una estructura llamada `Span`, la cual contiene las posiciones de inicio y fin del token dentro del código fuente **(medidas en offsets de byte)**. Esta información es esencial para mantener un seguimiento preciso de la ubicación de cada elemento del programa.

El `Span` es generado automáticamente por `LALRPOP` en el archivo `parser.lalrpop`, utilizando las anotaciones `@L` y `@R` que corresponden a los `offsets` de los `tokens`. Gracias a esto, el `AST` generado por el compilador conserva metadatos de ubicación en todos sus nodos.

Por último, esta información es fundamental para el archivo `parser_w_errors.rs`, que aprovecha los `Span` para generar mensajes de error sintáctico detallados y amigables, indicando la línea, columna y contexto exacto en el que ocurre un problema de análisis.

# 5. Análisis Sintáctico

## 5.1. Tipo de Parser

- Por defecto, `LALRPOP` genera un parser de tipo `LR(1)` (específicamente con una variante llamada `lane table`, más compacta).
- Esto significa un análisis ascendente, con un símbolo de `lookahead`, garantizando un parse completo y eficiente `O(n)` sin backtracking.

### 5.2. Proceso en `parser.lalrpop`

- Definición de la gramática `CFG` en secciones como `Program, Statement, Expr,` etc., donde se mezclan reglas no terminales y acciones semánticas (constructores de `AST`).
- `LALRPOP` construye internamente un autómata `LR(0)`, calcula conjuntos `LR(1)`, y genera tablas de parsing (`shift/reduce`).
- Se genera código Rust con funciones que emplean un stack de estados:
  - Operaciones `shift` (empujar token al stack).
  - Operaciones `reduce` (aplicar una regla, armar un `AST`).
  - Estado `accept` cuando termina.
- Este parser se invoca dentro del wrapper `parser_w_errors.rs`.

### 5.3. Manejo de Errores Sintácticos (`parser_w_errors.rs`)

En lugar de usar el parser directamente, se envuelve en un wrapper para generar mensajes de error más humanos:

- Captura errores como `InvalidToken, UnrecognizedToken, UnrecognizedEof, ExtraToken,` etc.
- Utiliza información de posición `(Span, offset)` y funciones como `get_line_context, build_caret_point,` etc.
- Genera un mensaje con la línea, columna, y una flecha (`^...`) para mostrar el lugar del error con un snippet.
- Convierte internamente tokens inesperados en mensajes.

## 6. Análisis Semántico

El análisis semántico en nuestro compilador está gestionado por la estructura `SemanticAnalyzer`, definida en `semantic_analyzer.rs`. Este analizador implementa el patrón `Visitor` para recorrer el `AST` e ir aplicando validaciones semánticas nodo por nodo. Para llevar a cabo este proceso, el SemanticAnalyzer mantiene:

- Un contexto semántico (`SemanticContext`) que guarda información sobre los símbolos visibles (variables, funciones declaradas, tipos definidos y el tipo o función actual en análisis).
- Una pila de contextos para simular los distintos ámbitos (`scopes`).
- Una lista de errores semánticos (`Vec<SemanticError>`) recolectados durante el análisis.
- Un árbol de tipos (`TypeTree`) para verificar compatibilidad entre tipos definidos por el usuario o nativos.

## 6.1. Tipos de Errores Semánticos

Los errores semánticos se representan como variantes del enum `SemanticError`, definido en `semantic_errors.rs`, y cada uno incluye un mensaje personalizado y un `Span` con la posición del error en el código fuente para su posterior reporte. Los principales errores que se detectan son:

- `DivisionByZero`: Se detecta una división por cero en tiempo de compilación.
- `UndefinedIdentifier`: Se usa un identificador que no ha sido declarado en el ámbito actual.
- `RedefinitionOfVariable / RedefinitionOfFunction / RedefinitionOfType`: Se intenta declarar una variable, función o tipo con un nombre ya existente en el mismo contexto.
- `UndeclaredFunction`: Se llama a una función que no ha sido definida previamente.
- `InvalidArgumentsCount`: El número de argumentos en una llamada a función no coincide con el esperado.
- `InvalidTypeArgument / InvalidTypeArgumentCount`: El tipo de uno o más argumentos no coincide con la firma esperada de la función o tipo, o se pasan demasiados/pocos argumentos genéricos.
- `InvalidFunctionReturn`: El tipo retornado por una función no coincide con su declaración.
- `InvalidBinaryOperation / InvalidUnaryOperation`: Operaciones entre tipos incompatibles o inválidas.
- `InvalidConditionType`: La condición de una estructura `if, while` o similar no tiene un tipo `booleano`.
- `UndefinedType`: Se hace referencia a un tipo no declarado.
- `InvalidTypeFunctionAccess / InvalidTypeProperty / InvalidTypePropertyAccess`: Se accede a métodos o propiedades inexistentes o privadas en un tipo definido por el usuario.
- `CycleDetected`: Se detecta una dependencia cíclica entre definiciones de tipos (herencia).
- `InvalidPrint / InvalidIterable`: Intentos de imprimir o iterar sobre elementos no válidos.

Todos estos errores se reportan mediante el método `report`, que imprime un mensaje detallado con el contexto del código y una flecha que apunta al lugar exacto del problema, facilitando la depuración por parte del usuario.

## 6.2. Explicación de `returned_types.rs`

Este archivo define dos estructuras clave:

- **FunctionInfo**: contiene el nombre de una función, su lista de argumentos con sus tipos y su tipo de retorno. Se utiliza para validar llamadas a funciones.
- **SemanticContext**: estructura mutable que se actualiza durante el recorrido del AST, donde se almacenan:
  - `symbols`: tabla de variables (nombre → tipo).
  - `declared_functions`: funciones ya definidas.
  - `declared_types`: tipos definidos por el usuario.
- `current_type` y `current_function`: ayudan a contextualizar errores dentro de clases o funciones específicas.

## 6.3. Estructura y uso del módulo `types_tree` en el análisis semántico

El módulo `types_tree` es uno de los pilares del sistema de tipos del compilador. Su objetivo es representar y gestionar la jerarquía de tipos definidos en el lenguaje, tanto los tipos básicos incorporados como aquellos definidos por el usuario. Este módulo permite validar herencias, acceder a métodos y atributos, y detectar errores semánticos relacionados con el sistema de tipos durante el análisis del `AST`.

#### Representación de los Tipos: `TypeNode`

Cada tipo del lenguaje es representado por un nodo (`TypeNode`) que contiene la siguiente información:

- `type_name`: nombre del tipo.
- `params`: parámetros definidos para la construcción del tipo.
- `depth`: profundidad del tipo en la jerarquía (útil para determinar el ancestro común más cercano).
- `parent` y `children`: relaciones de herencia con otros tipos.
- `variables`: mapa de atributos (nombre → tipo).
- `methods`: mapa de métodos asociados al tipo.

Este nodo también ofrece métodos auxiliares para agregar atributos, métodos, definir la herencia y acceder a métodos mediante `get_method`.

**Estructura General del Árbol:** `TypeTree`

El árbol de tipos (`TypeTree`) es la estructura central que mantiene todos los `TypeNode`. Este árbol tiene un nodo raíz predefinido llamado `."bject"`, y al inicializarse se agregan también los tipos primitivos del lenguaje: `String`, `Number`, `Boolean` y un tipo especial `Unknown`.

Los métodos más relevantes del árbol son:

- `add_type`: agrega un nuevo tipo al árbol, asociándolo a su padre (si existe).
- `get_type`: permite obtener información de un tipo por su nombre.
- `find_method`: busca recursivamente un método en un tipo y sus ancestros.
- `is_ancestor`: comprueba si un tipo es ancestro de otro.
- `find_lca`: encuentra el ancestro común más cercano entre dos tipos.
- `check_cicle`: detecta ciclos de herencia ilegales (herencia circular).

**Uso durante el Análisis Semántico**

Durante el análisis semántico, el `TypeTree` es utilizado de las siguientes formas clave:

1. **Verificación de herencia válida y sin ciclos.** Cuando se declara un nuevo tipo, se inserta en el `TypeTree`. Posteriormente se invoca `check_cicle` para detectar si existe un ciclo de herencia. En caso de encontrar uno, se genera el error semántico `CycleDetected`.
2. **Propagación y herencia de métodos.** Cuando se accede a un método en una expresión de tipo (`type.method()`), se llama a `find_method`. Si el método no se encuentra en el tipo directamente, se busca en sus ancestros. Si no se encuentra en ningún lugar, se lanza el error `InvalidTypeFunctionAccess`.
3. **Acceso a propiedades y atributos.** Cuando se accede a una propiedad (`type.prop`), se consulta el `variables` del `TypeNode` correspondiente. Si no existe la propiedad, se lanza el error `InvalidTypeProperty`. Si existe pero no es accesible (por ejemplo, si es privada), se lanza `InvalidTypePropertyAccess`.
4. **Determinación del tipo resultante entre operaciones.** El árbol permite usar la función `find_lca` para determinar el ancestro común entre dos tipos, lo cual es útil cuando se necesitan unificar tipos (por ejemplo, en operaciones ternarias o conversiones implícitas).
5. **Resolución de tipos en expresiones `new`.** Cuando se instancia un nuevo objeto con `new`, se verifica que el tipo exista (`get_type`), y que se le pasen los argumentos correctos. En caso de errores, se lanza `UndefinedType` o `InvalidTypeArgumentCount`.

# 7. Generación de código con LLVM

La etapa final del proceso de compilación corresponde a la generación de código intermedio en formato LLVM.

El enfoque de esta sección no se centra en los detalles sintácticos específicos del lenguaje intermedio, sino en los principios fundamentales y abstracciones clave que permiten traducir construcciones de alto nivel propias de HULK a una representación intermedia ejecutable. Se explican los mecanismos de transformación que puentean la brecha entre un lenguaje orientado a expresiones complejas y su equivalente en LLVM-IR.

El proceso de generación de código LLVM sigue un flujo estructurado que aprovecha el **Árbol de Sintaxis Abstracta (AST)** ya validado en fases anteriores. La estrategia central se basa en el **patrón Visitor**, que permite recorrer el AST de manera sistemática para emitir el código equivalente en LLVM-IR.

## Flujo Principal

1. **Preparación del entorno**: Inicialización de módulos, funciones y variables globales en LLVM
2. **Procesamiento de definiciones**:
   - Generación de estructuras para tipos personalizados
   - Creación de prototipos de funciones
3. **Generación del cuerpo principal**:
   - Traducción de expresiones en la función `main` de hulk
   - Manejo de estructuras de control y llamadas a funciones

## Mecanismo de Traducción

La implementación sigue estos principios fundamentales:

- Cada nodo del AST implementa el método `accept(Visitor)`
- El Visitor contiene la lógica específica para:
  - *Nodos de declaración*: Generan estructuras globales
  - *Nodos de expresión*: Producen código ejecutable
- Las anotaciones de tipos añadidas durante el análisis semántico:
  - Determinan las firmas de funciones en LLVM
  - Especifican conversiones de tipo implícitas
  - Guían la selección de operaciones primitivas

## 7.1. Manejo de Variables

### 7.1.1. Declaración y Asignación:

La reserva de memoria para variables locales se realiza en tiempo de ejecución mediante la instrucción `alloca`, que asigna espacio en el stack. Las operaciones de escritura y lectura de valores se efectúan utilizando las instrucciones `store` y `load` respectivamente.

Un registro de símbolos gestiona el contexto de ejecución, estableciendo una correspondencia entre cada identificador de variable y su ubicación en memoria dentro del ámbito correspondiente.

### 7.1.2. Ámbito y Visibilidad:

El sistema de símbolos implementa múltiples niveles de ámbito (scopes), permitiendo:

- Declaraciones locales (dentro de funciones, bloques de código y estructuras de control)
- Variables globales (accesibles desde cualquier punto del programa)
- Reutilización de identificadores en ámbitos internos (shadowing)
- Resolución de nombres mediante búsqueda jerárquica en ámbitos anidados

La gestión de la jerarquía de ámbitos se implementa mediante una pila dinámica que se actualiza automáticamente al entrar y salir de diferentes contextos léxicos.

## 7.2. Manejo de Funciones

### 7.2.1. Declaración de Funciones:

El nodo `FunctionDefNode` representa la definición de una función. Su conversión a LLVM IR comprende las siguientes etapas:

1. Especificación del tipo de retorno y los tipos de los parámetros
2. Instanciación de un nuevo objeto `llvm`
3. Creación del bloque básico inicial
4. Asignación de identificadores a parámetros y registro en la tabla de símbolos
5. Generación del contenido funcional con gestión del flujo de control y entorno local

#### 7.2.2. Llamadas a Funciones:

Las invocaciones funcionales se gestionan mediante el nodo `FunctionCallNode`, ejecutando estas operaciones:

- Evaluación secuencial de los argumentos proporcionados
- Comprobación de conformidad de tipos entre argumentos y parámetros
- Generación de la instrucción `call`
- Recuperación del valor de retorno (cuando aplica) y asignación en el contexto actual

#### 7.2.3. Ciclos (`WhileNode,ForNode`)

El sistema implementa estructuras de repetición con gestión explícita de condición e iteración. La traducción general comprende:

- Bloque para evaluación de la condición
- Bloque para ejecución del cuerpo iterativo

### 7.3. Sistema de Tipos

#### 7.3.1. Tipos Básicos

Se incluye soporte para los siguientes tipos primitivos:

- Number (`i32`, `i64`, `float`, `double`)
- Boolean (`i1`)
- String (representados como punteros a `ptr`)

#### 7.3.2. Tipos Personalizados (`TypeDefNode`)

El mecanismo habilita la definición de tipos definidos por el usuario, con funcionalidades para:

- Atributos estáticos y dinámicos
- Métodos asociados (implementados mediante funciones con primer parámetro tipo instancia)
- Relaciones de herencia y composición

#### 7.3.3. Operaciones sobre Tipos

**Acceso a Atributos (`TypePropAccessNode`)** :
Se emplean instrucciones `getelementptr` (GEP) para determinar el desplazamiento del atributo dentro de la estructura. Posteriormente, se utiliza una instrucción `load` para recuperar su valor almacenado.

**Modificación de Atributos** :
El procedimiento comprende:

- Cómputo de la dirección mediante GEP
- Comprobación de tipos del valor asignado
- Emisión de la instrucción `store` para actualizar el atributo