# Havana University Language for Kompilers (HULK)

## Informe Técnico de Implementación

Gabrile Andrés Pla Lasa[1] and Carlos Daniel Largacha Leal[2]

[1] Universidad de La Habana gabriel.aplasa@estudiantes.matcom.uh.cu
[2] Universidad de La Habana carlos.dlargacha@estudiantes.matcom.uh.cu

**Resumen** Este informe presenta el desarrollo de un compilador para el lenguaje de programación Hulk, un lenguaje educativo de tipado estático con orientación a objetos y herencia. La implementación cubre todas las etapas del proceso de compilación: análisis léxico y sintáctico mediante Flex y Bison, análisis semántico completo, inferencia y chequeo de tipos, y generación de código intermedio (LLVM IR). El backend se apoya en la infraestructura de LLVM para producir código eficiente y portable, incluyendo mecanismos de despacho dinámico a través de vtables. El sistema ha sido diseñado con una arquitectura modular basada en visitantes, facilitando su mantenibilidad y extensión. Se incluye un flujo de compilación automatizado y una etapa de vinculación que permite ejecutar programas Hulk como binarios nativos.

**Keywords:** Compiladores, LLVM, Lenguajes educativos, Generación de código, Análisis semántico, Hulk, Flex, Bison, Tipado estático, Herencia, Vtables, Visitors

## 1. Introducción

**Hulk** es un lenguaje de programación imperativo, fuertemente tipado y con tipado estático, diseñado como una herramienta educativa para la enseñanza de conceptos fundamentales en compilación y semántica de lenguajes. Inspirado en lenguajes como Python y C++, Hulk ofrece una sintaxis sencilla y expresiva, soporte para funciones de alto nivel, clases con herencia, y una semántica clara orientada a la seguridad de tipos. Su diseño modular lo convierte en una plataforma ideal para la implementación de compiladores modernos.

Este proyecto implementa un compilador completo para Hulk, desarrollado en C++ y basado en las herramientas Flex, Bison y LLVM. El proceso de compilación inicia con el análisis léxico y sintáctico, a partir del cual se construye un árbol de sintaxis abstracta (AST). A continuación, se realiza un análisis semántico en múltiples etapas: definición de scopes, verificación del uso correcto de variables, inferencia de tipos y chequeo de consistencia entre tipos. Una vez validado el programa, se procede a la generación de código intermedio (IR) en LLVM, que se emite a un archivo `.ll`. Este código es posteriormente traducido a ensamblador

mediante `llc` y vinculado junto a un pequeño runtime en C, produciendo un ejecutable nativo.

La arquitectura del compilador se estructura en visitantes especializados para cada fase, permitiendo una separación clara de responsabilidades y facilitando su extensión. Gracias a la integración con LLVM, el backend es capaz de generar código eficiente y portable, al tiempo que ofrece mecanismos avanzados como vtables, despacho dinámico y representación estructurada de tipos. Además, el sistema de construcción del proyecto, basado en un `Makefile`, automatiza por completo el flujo de compilación y ejecución, permitiendo compilar archivos fuente de Hulk con una sola instrucción y facilitando pruebas rápidas durante el desarrollo.

## 2. Análisis léxico

El *lexer* o analizador léxico constituye la primera fase del proceso de compilación y tiene como objetivo principal transformar una secuencia de caracteres en una secuencia de unidades léxicas o *tokens*, los cuales representan los elementos básicos del lenguaje de programación. Para esta tarea se ha utilizado **Flex**, una herramienta generadora de analizadores léxicos que, a partir de una especificación basada en expresiones regulares, produce automáticamente el código C necesario para reconocer los patrones definidos. Su integración con **Bison**, el generador sintáctico utilizado en este proyecto, permite una comunicación fluida entre ambas fases del compilador.

### 2.1. Constructor del lexer con Flex

En la implementación actual del compilador de Hulk, el lexer reconoce los diferentes tipos de tokens definidos en la gramática del lenguaje, agrupándolos en categorías como identificadores, literales numéricos y de cadena, operadores aritméticos y lógicos, símbolos de puntuación, y palabras clave reservadas. Las expresiones regulares definidas permiten capturar con precisión la estructura léxica de cada uno de estos grupos. Además, se contemplan constructos particulares del lenguaje Hulk, como operadores específicos (`@`, `@@`) o asignaciones destructivas.

Un aspecto relevante del analizador léxico desarrollado es que detecta y reporta errores léxicos de manera inmediata, deteniendo el proceso de análisis ante la primera ocurrencia inválida. Esta estrategia permite un control más riguroso sobre la validez del código fuente desde sus primeras etapas. Asimismo, se aprovecha la capacidad del motor de expresiones regulares de Flex para identificar patrones que, aunque sintácticamente correctos desde el punto de vista formal, no son válidos en Hulk según su definición oficial, como identificadores mal formados o números con sufijos ilegales. En tales casos, se generan mensajes de error explícitos indicando la línea y el lexema conflictivo.

```
[Line 1] Lexer error: identificador inválido '_hola'
```

El analizador también incluye una función auxiliar para procesar caracteres de escape dentro de cadenas literales, permitiendo una interpretación coherente de secuencias como `\n`, `\t` o `\¨`.

### 2.2. Lexer personalizado con motor de expresiones regulares

Adicionalmente, se ha desarrollado un generador de lexer propio, implementado en C++ y basado en un motor de expresiones regulares parametrizable. Esta alternativa experimental permite una mayor flexibilidad y portabilidad en comparación con la solución tradicional basada en herramientas externas como Flex.

Este lexer está compuesto por tres elementos principales:

- Una estructura `Token` genérica, capaz de representar cualquier tipo de token mediante el uso de `std::variant`, incluyendo su tipo, valor, lexema, línea y columna de aparición.
- Una clase `RegexEngine` encargada de almacenar las reglas léxicas, aplicar patrones de error y realizar el proceso de tokenización sobre una entrada fuente utilizando expresiones regulares de la biblioteca estándar de C++.
- Una clase contenedora `Lexer`, que permite gestionar la fuente, invocar la tokenización y recorrer los tokens generados de manera secuencial o en bloque.

Durante el análisis, el motor recorre el texto fuente e intenta emparejar, en orden, cada fragmento de entrada contra los patrones registrados. El proceso contempla dos fases: primero se evalúan los patrones que definen errores léxicos explícitos, y luego las reglas válidas. Si ningún patrón coincide, se lanza una excepción que identifica el carácter inesperado junto a su posición en el archivo.

Esta implementación permite definir reglas de manera declarativa, con convertidores de valor opcionales para adaptar lexemas a tipos como flotantes, cadenas o booleanos. También se permite marcar reglas como ignorables, útil para espacios en blanco y saltos de línea.

A continuación se ilustra un ejemplo mínimo del uso del motor de expresiones regulares para definir un lexer personalizado. En este caso, se construyen reglas para un lenguaje aritmético elemental, definiendo tipos de tokens como números, identificadores y operadores básicos:

```
1          enum class MyTokenType {
2              NUMBER,       // Ej: 42
3              IDENTIFIER,   // Ej: x
4              PLUS,         // +
5              MINUS,        // -
6              MULTIPLY,     // *
7              DIVIDE,       // /
8              LPAREN,       // (
9              RPAREN,       // )
10             END           // Fin de entrada
```

```
11          };
12
13          using MyLexer = lexer::Lexer<MyTokenType>;
14
15          int main() {
16              MyLexer lexer;
17
18              lexer.engine().add_rule(R"(\d+\.?\d*)",
                    MyTokenType::NUMBER);
19              lexer.engine().add_rule(R"([a-zA-Z_]\w*)",
                    MyTokenType::IDENTIFIER);
20              lexer.engine().add_rule(R"(\+)", MyTokenType::
                    PLUS);
21              lexer.engine().add_rule(R"(\-)", MyTokenType::
                    MINUS);
22              lexer.engine().add_rule(R"(\*)", MyTokenType::
                    MULTIPLY);
23              lexer.engine().add_rule(R"(\/)", MyTokenType::
                    DIVIDE);
24              lexer.engine().add_rule(R"(\()", MyTokenType::
                    LPAREN);
25              lexer.engine().add_rule(R"(\))", MyTokenType::
                    RPAREN);
26              lexer.engine().add_rule(R"(\s+)", MyTokenType::
                    END, nullptr, true);
27          }
```

**Listing 1.1.** Ejemplo de configuración de un lexer personalizado mediante expresiones regulares

Cabe destacar que, a pesar de estar plenamente funcional, este lexer aún no se encuentra integrado en el flujo principal del compilador, el cual continúa utilizando Flex como herramienta predeterminada para el análisis léxico.

## 3. Análisis sintáctico y construcción del AST

En la construcción de compiladores, una **gramática** es un conjunto de reglas formales que describen la estructura sintáctica válida de un lenguaje. Estas reglas definen cómo pueden combinarse los *tokens* producidos por el analizador léxico para formar expresiones, sentencias y programas completos.

Un **parser** (o analizador sintáctico) es el componente del compilador encargado de aplicar estas reglas gramaticales para construir una representación estructurada del código fuente, generalmente en forma de un árbol. Este árbol puede ser:

- un *árbol de derivación* (parse tree), que refleja la aplicación explícita de cada regla gramatical;

- o un *árbol de sintaxis abstracta* (AST), una versión simplificada que omite detalles redundantes y captura directamente la semántica estructural del programa.

El **AST** es una estructura de datos fundamental para las siguientes etapas del compilador, como el análisis semántico y la generación de código intermedio, ya que permite representar de forma jerárquica y compacta las construcciones del lenguaje.

### 3.1. Construcción del parser con Bison

En este proyecto, el parser fue implementado con **Bison**, una herramienta para generar analizadores sintácticos LR. Bison permite definir una gramática mediante reglas de producción, cada una de las cuales puede estar asociada a una acción semántica en `C++`. Estas acciones permiten construir nodos del AST a medida que se reconoce la entrada.

La gramática implementada describe expresiones, declaraciones, bloques, funciones, estructuras condicionales, bucles, y un sistema de tipos con herencia. Cada construcción sintáctica relevante genera un nodo específico del AST.

En términos generales, el parser actúa como un constructor jerárquico del AST: cada vez que se reconoce una estructura válida, se instancia un nodo correspondiente que agrupa sus componentes sintácticos. Esto permite que al finalizar el análisis sintáctico, se haya construido un árbol completo que representa la estructura lógica del programa fuente.

### 3.2. Representación del AST

Cada nodo del AST hereda de la clase base abstracta `ASTNode`, que define las funciones virtuales `print()` y `accept()` (esta última para el patrón *Visitor*). Se han definido múltiples tipos de nodos, como:

- Literales: `FloatNode`, `BoolNode`, `StringNode`.
- Operadores: `UnaryOpNode`, `BinOpNode`.
- Control de flujo: `IfNode`, `WhileNode`.
- Funciones y bloques: `FunctionNode`, `BlockNode`.
- Tipos definidos por el usuario: `TypeNode`, `AttributeNode`, `MethodNode`, `InheritsNode`.
- Llamadas y acceso a miembros: `CallFuncNode`, `MethodCallNode`, `MemberAccessNode`.

Esta jerarquía permite recorrer el árbol con facilidad, así como aplicar operaciones como verificación semántica, generación de código intermedio o impresión estructurada del árbol.

### 3.3. Limitaciones del análisis sintáctico

Dado que Bison genera analizadores LR (con *lookahead* limitado), el parser detecta y reporta **un único error por vez**. Ante un error sintáctico, se interrumpe el análisis y se muestra un mensaje indicando la línea y el símbolo problemático:

```
[Line 12] Parser error: cerca de 'if'
```

Esto implica que, a diferencia de analizadores sintácticos más tolerantes (como los utilizados en algunos entornos de desarrollo), este parser requiere corrección inmediata del primer error antes de continuar con el análisis.

## 4. Chequeo semántico

En esta fase el compilador verifica que el programa sea semánticamente correcto según las reglas definidas del lenguaje Hulk luego de haber superado la fase léxica y sintáctica El objetivo principal es garantizar que todas las construcciones tengan sentido en el contexto del programa tales como la existencia de símbolos tipos adecuados y uso coherente de expresiones. Además el chequeo semántico añade anotaciones significativas al AST tales como información de tipo y enlace a símbolos para su posterior procesamiento.

### 4.1. Scopes contextos y tipos

Los *scopes* o entornos de definición constituyen la base para la resolución de identificadores en diferentes niveles del programa. Cada vez que se inicia el cuerpo de una función, se evalúa una expresión `let` o se declara un tipo, se crea un nuevo `Scope` que puede acceder a símbolos definidos localmente así como también a los contenidos de su scope padre.

La clase `Scope` guarda internamente un mapa de símbolos cada uno con información de tipo estático y dinámico el nodo AST asociado y el valor LLVM correspondiente. Las búsquedas se realizan mediante `lookup` o `localLookup` para distinguir entre resolución global y local. La definición de un símbolo mediante `define` permite sobrescribir definiciones locales sin afectar scopes anidados exteriores

El `Context` representa el entorno global del compilador. En su creación se inicializan los tipos primitivos (Number, String, Boolean y Object) y además se definen en el scope global constantes matemáticas como PI y E. El contexto mantiene también registros de funciones definidas por el usuario y funciones built-in junto con su información de tipo permitiendo distinguir entre funciones normales y aquellas con comportamiento especial

El sistema de tipos está representado por la clase `Type` que puede ser primitivo u objeto definido por el usuario con herencia simple. La información de herencia atributos métodos y parámetros de constructor se almacena en estructuras asociadas dentro del tipo. Y el `TypeRegistry`, accesible desde el contexto, sirve como repositorio único para consultar y crear tipos facilitando la consistencia en la verificación de conformidad y compatibilidad de tipos

### 4.2. Visitors de definición y uso

El chequeo semántico en Hulk se realiza en múltiples pasadas mediante visitantes especializados que recorren el árbol de sintaxis abstracta. Las dos primeras

pasadas, a cargo de los visitantes `DefinitionVisitor` y `UsageCheckerVisitor`, tienen como objetivo garantizar que todos los símbolos estén correctamente definidos y que su uso posterior respete el contexto en el que aparecen. Ambas fases están diseñadas para detectar múltiples errores en una sola ejecución, permitiendo al usuario obtener retroalimentación más completa durante el proceso de compilación.

El `DefinitionVisitor` se encarga de visitar las definiciones del programa, asegurando que todas las funciones, tipos, métodos, atributos y variables estén registradas en los entornos adecuados. Durante este recorrido se construye la jerarquía de scopes del programa, asociando un nuevo `Scope` con cada bloque, función, declaración `let` o definición de tipo. Las funciones se agregan a la tabla global mediante `defineFunction`, los atributos y métodos se registran en el `TypeRegistry`, y los parámetros de tipo se almacenan como símbolos especiales dentro del scope del tipo correspondiente. Además, en esta fase también se resuelve la herencia entre tipos, detectando conflictos básicos como herencia de tipos no definidos o ambigüedades en la definición de parámetros. Aunque no se realiza aquí un análisis de tipos completo, esta primera pasada establece la estructura base sobre la cual las siguientes fases se apoyan.

A continuación, el `UsageCheckerVisitor` recorre nuevamente el AST validando que el uso de los identificadores declarados sea correcto. Este visitante comprueba que las variables hayan sido previamente definidas en el entorno de ejecución actual o en sus ancestros, que las funciones invocadas hayan sido registradas globalmente, y que todos los tipos referenciados por nombre existan en el `TypeRegistry`. También se valida que las construcciones particulares del lenguaje, como el uso de `self` o `base`, se empleen dentro del contexto de una declaración de tipo y que no se accedan atributos de manera indebida desde fuera de su ámbito. En el caso de instanciaciones, se asegura que no se creen objetos a partir de tipos primitivos, y en la herencia, se verifica la ausencia de ciclos y la validez del tipo padre.

Ambos visitantes utilizan una estructura común para el manejo de errores, acumulando los mensajes de error en una lista interna que luego puede ser consultada por la interfaz del compilador. Esto permite continuar el análisis a pesar de encontrar errores tempranos, ofreciendo al usuario una visión más completa de los problemas en su código en lugar de detenerse ante el primer fallo encontrado. La combinación de estas dos pasadas iniciales garantiza que todas las entidades del programa estén correctamente declaradas y utilizadas antes de proceder al análisis de tipos.

```
1    type Point{};
2    type Point{};
3
4    let pt = new Poinll() in
5        print("x: " @ pt.getX() @ "; y: " @ pt.getY());
6
7    print(x*2);
```

**Listing 1.2.** Ejemplo de codigo con errores

```
1    [Line 2] Error semantico: El tipo Point ya fue declarado.
2    [Line 4] Error semantico: El tipo Poinll no esta definido
3    [Line 7] Error semantico: variable x no definida en este
        scope.
```

**Listing 1.3.** Múltiples errores capturados

### 4.3. Visitor de inferencia de tipos

El `TypeInferenceVisitor` constituye la tercera pasada semántica en el compilador de Hulk. Su propósito es recorrer el árbol de sintaxis abstracta e inferir los tipos dinámicos de expresiones, variables, funciones y atributos cuando estos no han sido declarados de forma explícita. A diferencia de las fases previas, que se enfocan en la validación estructural del programa, esta etapa está orientada a deducir información adicional que complemente la representación semántica del código.

Este visitor actualiza progresivamente un puntero `lastType`, que almacena el tipo inferido más reciente, y asocia dicha información a los símbolos presentes en la tabla de símbolos. Para manejar casos ambiguos, se incorpora lógica adicional que compara tipos y busca ancestros comunes en la jerarquía de tipos, a través del método `findLowestCommonAncestor` del `TypeRegistry`. Además, mediante la función auxiliar `putTypeOnVariables`, se garantiza que la información de tipos deducida a partir de expresiones se propague a las variables involucradas, incluso si no tienen un tipo explícito declarado.

#### Inferencia en nodos primitivos y operadores

Los nodos que representan literales, como `FloatNode`, `BoolNode` y `StringNode`, asignan directamente el tipo correspondiente a `lastType`. En el caso de operadores unarios y binarios, se deduce el tipo resultante en función del operador. Por ejemplo, expresiones aritméticas como `a + b` requieren que ambos operandos sean numéricos, y producen un resultado del mismo tipo. Operaciones booleanas (`&`, `|`, `!`) exigen operandos booleanos, mientras que comparaciones (`==`, `<`, etc.) infieren siempre tipo `Bool`. En todos los casos, se valida la compatibilidad entre operandos y se registra el tipo de cada uno en su símbolo correspondiente, si aplica.

#### Variables y declaraciones locales

Cuando se encuentra una `VariableNode`, se extrae su tipo dinámico desde la tabla de símbolos. Si el visitor se encuentra en modo de verificación (`checkVariableType`), se lanza un error en caso de que la variable aún no tenga un tipo inferido. En expresiones `let-in`, se procesa primero el valor de cada binding y luego se asocia el tipo inferido al identificador introducido. En caso de existir un tipo declarado, este sobreescribe el inferido. Esto garantiza coherencia entre lo declarado y lo inferido, pero también permite operar de forma totalmente dinámica si no se especifica un tipo.

### Funciones y llamadas

En las funciones definidas por el usuario, el visitor recorre el bloque y deduce el tipo de retorno en base al tipo inferido de la última expresión. Si la función no posee un tipo declarado, se le asigna el tipo inferido automáticamente. Posteriormente se vuelve a visitar cada argumento para confirmar que su tipo esté definido, y se almacena el tipo de retorno dinámico en la estructura `FunctionInfo`. Al invocar funciones, si se trata de una función incorporada, se utilizan los tipos esperados de sus argumentos para refinar la inferencia de cada expresión argumental. Para funciones del usuario, se toma directamente su tipo de retorno dinámico inferido previamente.

### Control de flujo

Los nodos de control como `if` y `while` implican inferencia de tipo condicional. En el caso de `if`, se determina el tipo común más específico entre las ramas `if`/`elif`/`else`, utilizando nuevamente la operación de ancestro común. Si alguna rama carece de un tipo inferido, se lanza un error. Para los bucles `while`, el tipo inferido corresponde al tipo del cuerpo, lo que permite cadenas de expresiones incluso dentro de estructuras repetitivas.

### Tipos, métodos y atributos

En los nodos relacionados con la definición de tipos, se propaga el visitor sobre los miembros internos. Para los métodos, se aplica el mismo esquema de inferencia de funciones, pero además se actualiza la estructura `FunctionType` correspondiente al método dentro del `TypeRegistry`. Si no se declara tipo de retorno para el método, se asume el tipo inferido en su cuerpo. Los atributos, por su parte, son tratados como variables internas del tipo y su tipo puede derivarse del inicializador o bien declararse de forma explícita. En caso de ambigüedad o conflicto, se produce un error semántico.

### Acceso a miembros y llamadas a métodos

Para expresiones como `obj.attr`, el visitor primero infiere el tipo del objeto y luego consulta el atributo en la definición del tipo correspondiente. Si el atributo no existe o su tipo no ha sido previamente inferido, se produce un error. De forma similar, para llamadas a métodos, se busca el método en la jerarquía del tipo del objeto. Si se encuentra, se devuelve su tipo de retorno. El visitor también impone restricciones: no se permite invocar métodos sobre tipos primitivos, ni acceder a atributos si no hay un tipo asociado.

### Propagación y robustez

Este visitor está diseñado para ser robusto ante código parcialmente declarado. Gracias a su estructura en múltiples pasadas, permite deducir información a partir del uso concreto de las expresiones, refinando los tipos incluso en ausencia de declaraciones explícitas.

### 4.4. Visitor de chequeo de Tipos

El `TypeCheckerVisitor` es responsable de verificar que todas las expresiones del programa respeten el sistema de tipos en tiempo de compilación. A diferencia del `TypeInferenceVisitor`, cuyo propósito era inferir el tipo dinámico de cada subexpresión evaluando su significado, el chequeador de tipos se encarga de validar y asignar el tipo estático de cada componente, en base a lo declarado explícitamente o inferido previamente.

Ambos visitantes recorren el mismo árbol sintáctico abstracto, pero su objetivo difiere fundamentalmente. Mientras que la inferencia se apoya en la semántica del lenguaje para deducir tipos posibles a partir de las construcciones utilizadas, el chequeo busca garantizar que los tipos inferidos coincidan —o sean compatibles vía subtipado— con los tipos estáticos definidos por el programador o por el entorno del lenguaje. Es decir, el `TypeCheckerVisitor` actúa como una capa de verificación que asegura que el tipo inferido dinámicamente se encuentra alineado con el tipo estático previsto.

En este proceso, el tipo estático se asigna formalmente en la estructura del entorno si la verificación es satisfactoria, mientras que cualquier inconsistencia —ya sea por declaraciones incorrectas, incompatibilidad entre operandos o argumentos de funciones— se reporta como un error semántico. Además de validar los tipos, este visitor complementa la información de cada nodo con su tipo estático, permitiendo que fases posteriores de la compilación se beneficien de una representación bien tipada del programa.

Por tanto, el trabajo realizado por el `TypeCheckerVisitor` no sólo garantiza la solidez semántica del programa, sino que también establece las bases necesarias para la posterior traducción a código intermedio o generación de código nativo, al proporcionar un sistema de tipos consistente, explícito y validado.

## 5. Generación de Código

La generación de código en Hulk se basa en un visitante `LLVMCodeGenVisitor` que recorre el AST y emite instrucciones en LLVM IR. El objetivo de esta etapa es transformar las construcciones del lenguaje en representaciones de bajo nivel que puedan ser posteriormente optimizadas y ejecutadas. Esta traducción considera tanto primitivas como flotantes, booleanos y cadenas, así como estructuras más complejas como funciones, condicionales y bucles. La implementación se apoya en el módulo `llvm::IRBuilder`, que permite construir instrucciones LLVM de manera programática y segura.

### 5.1. Generación de Código de expresiones

Al igual que en el análisis semántico, el visitante accede al contexto para obtener información de símbolos, tipos y funciones. No obstante, aquí el foco está en cómo esos elementos se traducen a operaciones concretas. Para los literales flotantes, booleanos y cadenas, se generan constantes adecuadas. En el

caso de las cadenas, se define una variable global con atributos específicos que garantizan su correcto manejo en memoria. Los operadores unarios como negación y lógica utilizan instrucciones especializadas según el tipo del operando. Para los operadores binarios, se emiten instrucciones aritméticas o lógicas dependiendo del operador, incluyendo soporte especial para la potencia mediante la intrínseca `llvm.pow` y para el operador módulo, que se emula a partir de operaciones elementales debido a la ausencia de una instrucción directa en LLVM.

La semántica de asignación distingue entre variables y atributos. En el primer caso, se realiza un `store` directo. En el segundo, se invoca el acceso correspondiente al atributo y se almacena allí el valor. El bloque `let-in` genera espacios de almacenamiento para las variables locales y asocia los valores iniciales mediante `alloca` y `store`. Las variables primitivas y los objetos definidos por el usuario se diferencian en el tratamiento del almacenamiento: mientras los primeros almacenan directamente su valor, los segundos manejan punteros a estructuras.

Las funciones se traducen a través de una serie de pasos que consideran tanto el tipo estático como la convención de representación para tipos definidos por el usuario. Los argumentos se mapean a registros internos mediante asignaciones con `alloca` y `store`, asegurando que puedan ser accedidos dentro del cuerpo de la función. El bloque de instrucciones se traduce recursivamente y el resultado final se retorna explícitamente. Para funciones built-in como `print`, `sin` o `log`, se reutilizan funciones externas declaradas en `runtime.c`, y se invocan mediante llamadas (`call`) con las conversiones necesarias.

El programa principal se genera de manera opcional si existen instrucciones fuera de cualquier función. En ese caso, se construye una función `main` y se traduce cada una de las instrucciones del cuerpo principal. Las llamadas a funciones se manejan de forma distinta dependiendo de si la función es built-in o definida por el usuario. En el segundo caso, se realiza un `bitcast` cuando el argumento corresponde a un subtipo, garantizando la coherencia con el tipo esperado en la firma.

La construcción de condicionales emplea múltiples bloques `BasicBlock`, uno por cada rama y un bloque final para la fusión. Los resultados de cada rama se combinan mediante un nodo `phi`, lo que permite que la estructura condicional produzca un valor. Un enfoque similar se sigue para los bucles `while`, donde se reserva espacio para almacenar el resultado de cada iteración, que luego es recuperado en la salida del bucle.

Finalmente, los operadores `is` y `as` se traducen como comparaciones de subtipado estructural en tiempo de compilación y conversiones de tipo mediante `bitcast`. Dado que el sistema de tipos ya ha validado la corrección de estas operaciones, la traducción en LLVM se limita a reflejar esta verificación mediante instrucciones simples, manteniendo así la eficiencia del código generado.

### 5.2. Generación de Código de tipos

La traducción de tipos definidos por el usuario en Hulk requiere una estrategia estructurada para representar atributos, métodos y relaciones de herencia en LLVM IR. Cada tipo se modela como una estructura (`StructType`) con campos

correspondientes a una vtable y a los atributos definidos localmente y heredados. La vtable, declarada como una estructura separada, contiene punteros a las implementaciones de los métodos disponibles, y se instancia globalmente como una constante por tipo.

El visitante `LLVMCodeGenVisitor` inicia el proceso a través del método `visit(TypeNode&)`, que construye una declaración adelantada de la estructura para permitir referencias recursivas. A continuación, se recopila la jerarquía de métodos heredados mediante un recorrido desde la raíz de la herencia. Se garantiza así que los métodos redefinidos sobrescriban los de los ancestros y que el orden se preserve de forma determinista en la vtable. Cada método se declara con nombre calificado (`Tipo.Método`) y se define una firma explícita en LLVM basada en los tipos estáticos. Posteriormente, se construye el tipo de la vtable mediante punteros genéricos (`i8*`), que luego se convierten a sus tipos reales mediante `bitcast` al momento de instanciar la vtable concreta del tipo.

La estructura del tipo incluye como primer campo un puntero a la vtable y luego los campos correspondientes a los atributos, en orden de herencia. Si un atributo tiene un tipo compuesto, su estructura se define recursivamente. Durante la visita al nodo de herencia (`InheritsNode`) se registra la lista de argumentos del constructor del padre, lo cual permite su inicialización posterior. Por otro lado, los atributos definidos explícitamente son almacenados junto con sus inicializadores en un mapa indexado por tipo y nombre.

Los métodos definidos dentro de un tipo se traducen mediante el método `visit(MethodNode&)`. El cuerpo del método se convierte en una función de LLVM con un primer argumento implícito que representa a `self`. Cada argumento se asigna a una variable local mediante instrucciones `alloca` y `store`. El valor de `self` se registra en el ámbito local y se utiliza para acceder a campos y métodos del objeto. Al finalizar el cuerpo del método, se inserta una instrucción `ret` que retorna el resultado computado.

La creación de objetos se implementa en `visit(NewNode&)` y sigue un esquema fijo. Se reserva memoria en la pila para la instancia, se inicializa la vtable en la primera posición y luego se evalúan los argumentos del constructor. Estos valores se almacenan en variables locales del constructor para luego ser usados en la inicialización de atributos. Cada atributo se inicializa ya sea con su valor por defecto o con el proporcionado por el constructor. Además, se realiza recursivamente la inicialización del constructor del tipo padre, si existe, mediante los argumentos registrados previamente. Esto garantiza que el objeto se construya respetando la jerarquía completa.

El acceso a atributos se realiza desde el puntero `self` mediante desplazamientos estructurados. El índice de cada atributo en la estructura del objeto se determina en tiempo de compilación y se utiliza para obtener punteros a campos específicos, a través de instrucciones `getelementptr`. Las operaciones de lectura usan `load` y las de escritura usan `store`, dependiendo de si el nodo es un objetivo de asignación.

Las llamadas a métodos sobre tipos definidos utilizan un esquema de despacho dinámico basado en vtables. En `visit(MethodCallNode&)`, se obtiene el puntero

a la vtable desde el objeto receptor y se accede al campo correspondiente al método mediante el índice calculado previamente. El puntero recuperado se convierte mediante `bitcast` al tipo real de la función que será invocada. Esto incluye adaptar la firma para reflejar el tipo dinámico del receptor. A continuación, se construye la lista de argumentos, encabezada por el receptor y seguida por los argumentos explícitos de la llamada, y se genera la instrucción `call`. Esta técnica permite mantener el polimorfismo en tiempo de ejecución sin comprometer la eficiencia, al evitar la introspección de tipos durante la ejecución.

Finalmente, el acceso a métodos heredados mediante la palabra clave `base` se implementa mediante un `bitcast` explícito del receptor al tipo base donde se encuentra el método original. Esto permite invocar directamente la función del ancestro sin pasar por el mecanismo de vtables, preservando la semántica de llamada estática que caracteriza a este tipo de invocación.

En conjunto, esta arquitectura de generación de código para tipos en Hulk permite soportar de forma eficiente y segura características como herencia, atributos compuestos, constructores parametrizados, despacho dinámico y llamadas a métodos base, todo dentro del marco de tipos fuertes y verificación estática.

## 6. Conclusiones

La implementación de un compilador completo para el lenguaje Hulk ha permitido explorar de manera integral el ciclo de vida de un programa fuente, desde su análisis léxico hasta su ejecución como binario nativo. El uso de LLVM como backend resultó fundamental para abstraer la generación de código y dotar al compilador de capacidades avanzadas, como el uso de tablas de métodos (vtables) y soporte para herencia y despacho dinámico.

La modularidad del diseño, basada en el patrón Visitor, facilitó la separación de preocupaciones entre las distintas fases de compilación, y el sistema de construcción con `Makefile` automatizó de forma efectiva el workflow del proyecto. Además, la experiencia adquirida con herramientas clásicas como Flex y Bison refuerza el valor formativo del desarrollo de compiladores desde cero.

Como trabajo futuro se plantea la incorporación de optimizaciones a nivel de IR, soporte para excepciones y la extensión del lenguaje con características adicionales como vectores, iteradores y protocolos.