Compilador de HULK

Integrantes:

Lia S. Lopeéz Rosales C312
Ariadna Velázquez Rey C311

Asignatura: Compilación
Universidad de La Habana

June 19, 2025

1 Introducción

Este informe describe el proceso de creación de un compilador para el lenguaje HULK (Havana University Language for Kompilers) utilizando como lenguaje base C++.

El compilador está diseñado siguiendo una arquitectura modular que divide el proceso de compilación en etapas bien definidas, cada una con responsabilidades específicas y mecanismos de manejo de errores robustos.

1.1 Flujo del Programa

El flujo del programa sigue una secuencia clara y estructurada de etapas de compilación. El punto de entrada es la función `main()` en `main.cpp`, que coordina todo el proceso:

1. **Inicialización**:
   - Procesamiento de argumentos de línea de comandos para determinar el archivo fuente
   - Apertura y validación del archivo `.hulk` especificado
   - Configuración del entorno de compilación
2. **Análisis Léxico y Sintáctico**:
   - Ejecución del análisis léxico mediante `yylex()`
   - Verificación de errores léxicos mediante `lex_error`
   - Ejecución del análisis sintáctico con `yyparse()`
   - Construcción del AST durante el proceso
3. **Validación del AST**:
   - Verificación de la validez del árbol mediante `is_valid_ast()`
   - Comprobación de nodos nulos y estructura correcta
   - Impresión de información de diagnóstico sobre los nodos raíz
4. **Análisis Semántico**:
   - Creación del analizador semántico (`SemanticAnalyzer`)
   - Ejecución del análisis sobre el AST
   - Verificación de tipos y reglas semánticas
5. **Generación de Código**:
   - Inicialización del contexto de generación (`CodeGenContext`)
   - Generación de código LLVM IR
   - Escritura del código generado en `hulk-low-code.ll`
6. **Finalización**:
   - Liberación de recursos mediante `delete_ast()`

- Cierre de archivos y limpieza de memoria
- Retorno del estado de la compilación

En cada etapa, el compilador implementa un manejo de errores robusto que permite:

- Detección temprana de problemas
- Mensajes de error descriptivos con información de ubicación
- Limpieza apropiada de recursos en caso de fallo
- Códigos de retorno específicos para diferentes tipos de error

2 Análisis Léxico: Lexer

El analizador léxico del compilador **HULK** está implementado utilizando **Flex**, con un diseño que prioriza la precisión en el seguimiento de posición y el manejo de errores. La implementación incluye características avanzadas para el seguimiento de ubicación y la detección temprana de errores.

2.1 Configuración y Características

El lexer está configurado con las siguientes características principales:

- **Opciones de Flex**:
  - `noyywrap`: Desactiva el procesamiento de múltiples archivos
  - `yylineno`: Habilita el seguimiento automático de líneas
  - `nounput`: Optimiza el lexer eliminando funciones no utilizadas
- **Seguimiento de Posición**:
  - Variable `yycolumn` para seguimiento preciso de columnas
  - Macro `YY_USER_ACTION` que actualiza automáticamente:
    - Línea inicial y final (`first_line`, `last_line`)
    - Columna inicial y final (`first_column`, `last_column`)

2.2 Tokens y Patrones

El lexer reconoce los siguientes tipos de tokens:

- **Literales**:
  - **Números**: Enteros y decimales (`[0-9]+(\.[0-9]+)?`)
  - **Cadenas**: Con soporte para caracteres de escape (`"(?:\\.|[]^")*"`)
  - **Booleanos**: `True` y `False`
  - **Null**: Valor nulo
- **Operadores**:

  - **Aritméticos**: `+`, `-`, `*`, `/`, `%`, `^`
  - **Comparación**: `<`, `>`, `<=`, `>=`, `==`, `!=`
  - **Lógicos**: `&`, `|`, `!`
  - **Concatenación**: `@`, `@@`

- **Palabras Clave**:
  - **Control de Flujo**: `if`, `else`, `elif`, `while`, `for`
  - **Declaraciones**: `function`, `let`, `in`, `type`
  - **POO**: `new`, `self`, `inherits`, `base`
  - **Funciones Matemáticas**: `sin`, `cos`, `max`, `min`, `sqrt`, `exp`, `log`, `rand`
- **Identificadores**: Patrón `[a-zA-Z_][a-zA-Z0-9_]*` con validación específica

2.3 Características Especiales

La implementación incluye funcionalidades adicionales:

- **Gestión de Strings**:
  - Eliminación automática de comillas delimitadoras
  - Preservación de caracteres de escape
- **Depuración**:
  - Mensajes informativos para cada token reconocido
  - Información de posición para facilitar el desarrollo
- **Integración con el Parser**:
  - Comunicación mediante `yylval` para pasar valores
  - Sincronización de posición mediante `yylloc`

3 Análisis Sintáctico: Parser

El analizador sintáctico del compilador **HULK** está implementado utilizando **Bison**, con un diseño que prioriza la construcción precisa del AST y el manejo de expresiones complejas. La implementación sigue una estructura clara y modular que facilita la extensión del lenguaje.

3.1 Estructura del Parser

El parser está organizado en secciones bien definidas:

- **Declaraciones Iniciales**:
  - Estructura `YYLTYPE` para seguimiento de ubicación
  - Vector `root` para almacenar el AST
  - Funciones auxiliares como `vectorize` para manejo de argumentos

- **Tipos Semánticos**:
  - Tipos básicos: `num`, `str`, `boolean`
  - Tipos compuestos: `node`, `list`, `param`
  - Tipos específicos: `if_branch`, `let_decl`, `type_body`

3.2 Tokens y Precedencia

La gramática define una jerarquía clara de operadores:

- **Operadores de Mayor Precedencia**:
  - Multiplicación, División, Módulo (`MUL`, `DIV`, `MOD`)
  - Suma y Resta (`ADD`, `SUB`)
- **Operadores de Comparación**:
  - Relacionales: `<`, `>`, `<=`, `>=`
  - Igualdad: `==`, `!=`
- **Operadores Lógicos**:
  - AND (`&`), OR (`|`), NOT (`!`)
- **Operadores Especiales**:
  - Concatenación: `@`, `@@`
  - Funciones matemáticas: `sin`, `cos`, `sqrt`, etc.

3.3 Reglas Gramaticales

La gramática está estructurada en niveles jerárquicos:

- **Nivel Superior**:
  - `program`: Lista de statements
  - `statement`: Expresiones, declaraciones y bloques
- **Expresiones**:
  - Literales: números, strings, booleanos, null
  - Operaciones binarias y unarias
  - Llamadas a funciones y métodos
  - Expresiones de control de flujo
- **Estructuras de Control**:
  - `if_expr`: Soporte para `if`, `elif`, `else`
  - `while_expr`: Bucles while con condición

  - `for_expr`: Bucles for con rangos

- **Programación Orientada a Objetos**:
  - Declaración de tipos con `type_decl`
  - Herencia mediante `inherits`
  - Métodos y atributos con `method_decl` y `attribute_decl`
  - Llamadas a métodos y constructores

3.4 Construcción del AST

Durante el análisis sintáctico, se construye el AST de manera incremental:

- **Nodos Básicos**:
  - Literales y identificadores
  - Operadores binarios y unarios
  - Llamadas a funciones
- **Nodos Compuestos**:
  - Bloques de código
  - Estructuras de control
  - Declaraciones de tipos y funciones
- **Información de Ubicación**:
  - Cada nodo almacena su línea de origen (`yylloc.first_line`)
  - Facilita el reporte preciso de errores

4 Chequeo Semántico

El chequeo semántico en el compilador **HULK** se implementa mediante un sistema sofisticado que utiliza el **Patrón Visitor** para recorrer y analizar el AST. La implementación se divide en varios componentes principales que trabajan en conjunto para garantizar la corrección semántica del programa.

4.1 Componentes Principales

- **SemanticAnalyzer**: Implementa el patrón Visitor y coordina todo el análisis semántico.
- **SymbolTable**: Gestiona los símbolos y ámbitos del programa.
- **FunctionCollector**: Recolecta y analiza las declaraciones de funciones.

4.2 Tabla de Símbolos

La tabla de símbolos (`SymbolTable`) es una estructura sofisticada que maneja:

- **Ámbitos Anidados**:
  - Vector de mapas para manejar ámbitos (`scopes`)
  - Métodos `enterScope()` y `exitScope()` para gestión de ámbitos
  - Búsqueda de símbolos en ámbitos anidados
- **Tipos de Símbolos**:
  - Variables con tipo y estado de constante
  - Funciones con tipo de retorno y parámetros
  - Tipos definidos por el usuario con atributos y métodos
- **Tipos Predefinidos**:
  - Object, Number, String, Boolean, Null
  - Jerarquía de tipos y relaciones de subtipado

4.3 Análisis de Tipos

El sistema implementa un sofisticado análisis de tipos que incluye:

- **Inferencia de Tipos**:
  - Análisis de uso de parámetros en el cuerpo de funciones
  - Inferencia basada en operaciones y contexto
  - Resolución de tipos más específicos comunes
- **Verificación de Operaciones**:
  - Operaciones aritméticas (`+`, `-`, `*`, `/`, `%`)
  - Operaciones de comparación (`>`, `<`, `>=`, `<=`)
  - Operaciones lógicas (`&`, `|`, `!`)
  - Concatenación de strings (`@`, `@@`)
- **Jerarquía de Tipos**:
  - Verificación de conformidad de tipos (`conformsTo`)
  - Búsqueda del ancestro común más bajo (`lowestCommonAncestor`)
  - Manejo de herencia

4.4 Validación Semántica

El analizador semántico realiza múltiples validaciones:

- **Declaraciones**:
  - Unicidad de nombres en el ámbito actual
  - Tipos correctos en declaraciones
- **Funciones**:
  - Compatibilidad de tipos en argumentos
  - Inferencia de tipos de retorno
  - Validación de funciones matemáticas built-in
- **Programación Orientada a Objetos**:
  - Declaración y uso de tipos
  - Validación de herencia
  - Llamadas a métodos y constructores
  - Acceso a atributos
- **Control de Flujo**:
  - Tipos correctos en condiciones
  - Validación de bucles
  - Comprobación de expresiones let-in

5 Generación de Código LLVM

La generación de código intermedio en el compilador **Hulk** se realiza utilizando **LLVM IR** (Intermediate Representation). El sistema está diseñado de manera modular, con una clara separación de responsabilidades entre la generación de código, el manejo de tipos y el contexto de generación.

5.1 Componentes Principales

La implementación se divide en tres componentes principales:

- **LLVMGenerator**: Implementa el patrón Visitor para recorrer el AST y generar código LLVM IR.
- **CodeGenContext**: Encapsula todo el estado y contexto necesario para la generación de código.
- **TypeSystem**: Maneja el sistema de tipos, incluyendo definiciones de tipos y sus instancias.

5.2 Estructura del Generador

El generador de código (`LLVMGenerator`) implementa el patrón Visitor y proporciona métodos específicos para cada tipo de nodo del AST:

- Manejo de literales (números, booleanos, strings)
- Operaciones binarias y unarias
- Funciones built-in y definidas por el usuario
- Estructuras de control (if, while, for)
- Soporte para POO (declaración de tipos, instanciación, llamadas a métodos)

5.3 Contexto de Generación

El `CodeGenContext` mantiene el estado durante la generación de código:

- **Estado LLVM**: Contexto global, builder y módulo
- **Gestión de Ámbitos**: Pilas para variables locales y funciones
- **Sistema de Tipos**: Integración con el sistema de tipos para POO
- **Pila de Valores**: Mecanismo para pasar valores entre nodos durante el recorrido del AST

5.4 Sistema de Tipos

El `TypeSystem` proporciona soporte completo para POO:

- Definición de tipos con atributos y métodos
- Soporte para herencia
- Gestión de instancias y sus variables
- Manejo de constructores y llamadas base

5.5 Flujo de Generación

El proceso de generación sigue estos pasos:

1. **Inicialización**:
   - Configuración del contexto LLVM y builder
   - Declaración de funciones externas (printf, malloc, operaciones matemáticas)
   - Registro de tipos y funciones del usuario
2. **Generación de Código**:
   - Recorrido del AST usando el patrón Visitor

- Generación de instrucciones LLVM para cada tipo de nodo
- Manejo de la pila de valores para comunicación entre nodos

3. **Gestión de Memoria**:
   - Uso de `alloca` para variables locales
   - Manejo de memoria para strings y objetos
   - Gestión de ámbitos anidados
4. **Verificación**:
   - Validación del módulo LLVM generado
   - Generación del archivo IR final

5.6 Ejemplo de Generación

Consideremos la generación de código para una operación binaria:

1. El visitor procesa recursivamente los operandos izquierdo y derecho
2. Los valores resultantes se obtienen de la pila de valores
3. Se genera la instrucción LLVM correspondiente según el operador
4. El resultado se coloca en la pila para uso posterior

Por ejemplo, para la expresión `a + b` donde ambos son números:

- Se generan las cargas de las variables `a` y `b`
- Se crea una instrucción `fadd` usando el builder
- El resultado se almacena en la pila de valores

6 Manejo de Errores

El manejo de errores en el compilador **Hulk** está implementado de manera modular y robusta, con mecanismos específicos en cada fase de la compilación. El sistema está diseñado para detectar y reportar errores de manera precisa, proporcionando información detallada sobre la ubicación y naturaleza de cada error.

6.1 Errores Léxicos

El analizador léxico, implementado con **Flex**, incluye un sistema de seguimiento de posición preciso:

- **Seguimiento de Posición**: Se mantiene un registro exacto de línea y columna para cada token mediante las variables `yylineno` y `yycolumn`.
- **Detección de Errores**:

  - Identificadores inválidos (comenzando con guiones bajos o números)
  - Caracteres no reconocidos en el lenguaje
  - Tokens malformados

- **Reporte de Errores**: Los errores se reportan inmediatamente con:
  - Descripción del error
  - Línea y columna exacta del error
  - Token problemático

6.2 Errores Sintácticos

El parser, construido con **Bison**, implementa un sistema de recuperación de errores:

- **Sincronización**: Uso de tokens de sincronización (como el punto y coma) para recuperarse de errores.
- **Localización**: Aprovecha la información de ubicación (`YYLTYPE`) para reportar errores con precisión.
- **Validación de AST**: Verificación de la validez del árbol sintáctico generado:
  - Comprobación de nodos nulos
  - Validación de estructura del árbol
  - Verificación de completitud

6.3 Errores Semánticos

El analizador semántico implementa un sistema sofisticado de detección y reporte de errores:

- **Estructura de Error**:
  - Clase `SemanticError` para encapsular errores
  - Información de línea y mensaje descriptivo
  - Acumulación de errores para reporte completo
- **Tipos de Errores**:
  - Errores de tipo en operaciones
  - Variables no declaradas o mal utilizadas
  - Errores en llamadas a funciones
  - Problemas de herencia y tipos
- **Recuperación**:
  - Continuación del análisis tras errores
  - Inferencia de tipos en casos ambiguos
  - Manejo de tipos desconocidos

6.4 Errores en Generación de Código

La fase de generación de código LLVM incluye:

- **Manejo de Excepciones**: Captura y manejo de errores durante la generación.
- **Validación de IR**: Verificación del código LLVM generado.
- **Reporte de Errores**: Mensajes detallados sobre problemas en la generación.

6.5 Integración en el Flujo Principal

El `main.cpp` coordina el manejo de errores entre las diferentes etapas:

- **Verificación Secuencial**:
  - Validación de apertura de archivo
  - Comprobación de errores léxicos
  - Verificación de errores sintácticos
  - Validación del AST generado
  - Control de errores semánticos
  - Manejo de errores en generación de código
- **Limpieza de Recursos**: Liberación apropiada de memoria y recursos en caso de error.
- **Códigos de Retorno**: Uso de códigos de salida para indicar el tipo de error encontrado.