Compilador de HULK

Integrantes:

Lia S. Lopeéz Rosales C312
Ariadna Velázquez Rey C311

Asignatura: Compilación
Universidad de La Habana

June 23, 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 modular y estructurada de etapas de compilación, coordinadas desde la función principal en `main.cpp`. Cada etapa utiliza componentes propios implementados en C++ puro, sin herramientas externas. El procedimiento real es el siguiente:

1. **Inicialización**:
   - Procesamiento de argumentos de línea de comandos para determinar el archivo fuente (por defecto, `script.hulk`)
   - Lectura y validación del archivo fuente mediante la función `readFile()`
2. **Análisis Léxico**:
   - Instanciación de la clase `Lexer` con el código fuente
   - Generación secuencial de tokens usando `nextToken()`, almacenando cada token en un vector
   - Reporte inmediato de errores léxicos mediante `getErrors()`; si existen, el programa termina
3. **Carga y Construcción de la Gramática**:
   - Instanciación de `GrammarAugment` y carga de la gramática BNF desde archivo
   - Cálculo de conjuntos `FIRST` y `FOLLOW`
   - Construcción de conjuntos de items LR(1) con `LR1ItemSetBuilder`
   - Generación de tablas de parsing (`ACTION` y `GOTO`) con `LR1ParsingTableGenerator`, configurando precedencias
4. **Parsing y Construcción del AST**:
   - Instanciación de `SemanticActionDispatcher` para asociar acciones semánticas
   - Instanciación de `ParserDriver` y ejecución del ciclo de parsing LR(1) sobre la lista de tokens
   - Construcción del AST y acumulación de errores sintácticos
   - Validación del AST generado (no vacío, sin nodos nulos)
   - Reporte de errores sintácticos; si existen, el programa termina
5. **Impresión del AST**:

- Impresión de la estructura del AST usando `ASTPrinter` para diagnóstico

6. **Análisis Semántico**:
   - Instanciación de `SemanticAnalyzer` y análisis del AST
   - Gestión de ámbitos, variables, funciones y tipos mediante `SymbolTable`
   - Reporte de errores semánticos si existen
7. **Generación de Código LLVM**:
   - Instanciación de `CodeGenContext`, que encapsula el contexto de generación y el sistema de tipos
   - Generación de código LLVM IR recorriendo el AST
   - Volcado del IR generado en el archivo `hulk-low-code.ll`
8. **Finalización**:
   - El programa termina exitosamente si no hubo errores fatales en ninguna etapa

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

- Detección temprana y reporte inmediato de problemas léxicos, sintácticos y semánticos
- Mensajes de error descriptivos con información de ubicación precisa
- 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 completamente en C++ mediante una clase propia `Lexer`, sin utilizar Flex ni generadores automáticos. El diseño se basa en un autómata manual que recorre el texto fuente carácter por carácter, identificando tokens y gestionando errores de manera precisa y eficiente.

2.1 Estructura e Interfaz

El lexer se encapsula en la clase `Lexer`, que recibe el código fuente como una cadena y expone el método principal `nextToken()`, encargado de devolver el siguiente token del flujo. Cada token se representa mediante la estructura `Token`, que almacena el tipo (`TokenType`), el lexema y la ubicación exacta (línea y columna) en el código fuente. Los errores léxicos se encapsulan en la estructura `LexerError`, que incluye mensaje, ubicación y el lexema problemático.

2.2 Lógica de Tokenización y Autómata

El proceso de análisis léxico se realiza mediante un autómata implementado manualmente, con funciones especializadas para cada tipo de patrón:

- **Identificadores y Palabras Clave**: Reconocidos por el método `identifierOrKeyword()`, que detecta patrones del tipo `[a-zA-Z_][a-zA-Z0-9_]*` y los compara contra un mapa de palabras clave para distinguirlos de identificadores.

- **Números**: El método `number()` reconoce literales numéricos enteros y decimales, permitiendo la presencia de un punto decimal.
- **Cadenas**: El método `stringLiteral()` reconoce literales de cadena delimitadas por comillas dobles, preservando secuencias de escape y reportando errores en caso de cadenas sin cerrar.
- **Operadores y Puntuación**: El método `matchOperator()` detecta operadores de uno o dos caracteres (como `==`, `!=`, `@@`, etc.) y signos de puntuación mediante análisis por adelantado y un switch eficiente.
- **Espacios en Blanco**: El método `skipWhitespace()` omite espacios, tabulaciones y saltos de línea antes de analizar el siguiente token.

2.3 Tipos de Tokens y Estructura

El enum `TokenType` define todos los tipos de tokens reconocidos, incluyendo:

- Literales: números, cadenas, booleanos, nulo, identificadores
- Operadores aritméticos, lógicos, de comparación y concatenación
- Palabras clave del lenguaje (control de flujo, declaración, POO, funciones matemáticas, constantes)
- Signos de puntuación y operadores especiales
- Token especial para fin de archivo y para símbolos desconocidos

Cada token almacena su lexema y la ubicación exacta (línea y columna) donde fue encontrado, facilitando el diagnóstico y reporte de errores.

2.4 Manejo de Errores

El lexer implementa un sistema robusto de manejo de errores léxicos:

- **Reporte Inmediato**: Ante símbolos no reconocidos, cadenas sin cerrar o tokens malformados, se genera un `LexerError` con mensaje descriptivo, ubicación y el lexema problemático.
- **Acumulación de Errores**: Todos los errores detectados se almacenan y pueden ser consultados tras el análisis mediante `getErrors()`.
- **Precisión**: Cada error incluye la línea y columna exacta, permitiendo mensajes claros y útiles para el usuario.

2.5 Características Especiales y Extensibilidad

- **Seguimiento de Ubicación**: El lexer actualiza línea y columna en cada avance, asegurando precisión en la localización de tokens y errores.
- **Gestión de Strings**: Las cadenas preservan los caracteres de escape y eliminan automáticamente las comillas delimitadoras.

- **Modularidad**: La implementación en C++ puro facilita la extensión para nuevos tipos de tokens o reglas léxicas.
- **Integración**: El lexer está diseñado para integrarse fácilmente con el parser y el resto del compilador, proporcionando tokens y errores con toda la información contextual necesaria.

3 Análisis Sintáctico: Parser

El analizador sintáctico del compilador **HULK** está implementado como un generador LR(1) propio en C++, que abarca desde la normalización de la gramática hasta la ejecución del autómata y la construcción del AST. El sistema es completamente modular y extensible, y no depende de herramientas externas como Bison.

3.1 Flujo de Generación del Parser

El proceso de generación y ejecución del parser se compone de varias etapas bien diferenciadas:

1. **Normalización de Gramática (EBNF a BNF):**
   - El archivo de gramática fuente se escribe en EBNF para mayor expresividad.
   - El módulo `GrammarNormalizer` convierte automáticamente la EBNF a BNF, generando reglas auxiliares para secuencias, alternativas, repeticiones y opcionales.
   - El resultado es una lista de reglas BNF listas para la construcción del autómata.
2. **Construcción y Augmentación de la Gramática:**
   - El objeto `GrammarAugment` lee la gramática BNF, identifica terminales y no terminales, y realiza la augmentación necesaria para LR(1).
   - Calcula los conjuntos `FIRST` y `FOLLOW` para todos los símbolos, fundamentales para la predicción y la construcción de items LR(1).
3. **Construcción del Autómata LR(1):**
   - El `LR1ItemSetBuilder` genera el conjunto de estados del autómata LR(1) a partir de la gramática augmentada.
   - Cada estado es un conjunto de items LR(1), definidos por la posición del punto y el lookahead.
   - Se implementa el algoritmo de **closure** para expandir los items y el de **goto** para construir las transiciones entre estados.
   - Se utiliza una caché para optimizar la computación de closures.
4. **Generación de Tablas de Parsing:**
   - El `LR1ParsingTableGenerator` recorre los estados del autómata y genera las tablas `ACTION` y `GOTO`.
   - Se resuelven automáticamente los conflictos shift/reduce y reduce/reduce usando precedencia y asociatividad, configuradas explícitamente en el código.

- Se almacena, para cada estado, el conjunto de tokens esperados, facilitando la recuperación y reporte de errores.

5. **Ejecución del Parser:**
   - El `ParserDriver` implementa el ciclo de parsing LR(1) clásico: mantiene una pila de estados y una pila de valores semánticos.
   - Para cada token, consulta la tabla `ACTION` para decidir entre shift, reduce, accept o error.
   - Las reducciones invocan acciones semánticas a través del `SemanticActionDispatcher`, que construye el AST de manera incremental.
   - El parser soporta recuperación de errores y sincronización mediante tokens especiales (como punto y coma).

3.2 Automatización y Manipulación de Autómatas

- **Items LR(1):** Cada item se representa como una tupla `(lhs, rhs, dot, lookahead)`. El closure expande los items considerando los lookaheads, siguiendo el algoritmo estándar de LR(1).
- **Closure y Goto:** El closure se implementa de forma iterativa y eficiente, utilizando caché para evitar recomputaciones. El goto genera nuevos kernels y expande el autómata.
- **Transiciones:** Las transiciones entre estados se almacenan en una tabla de mapeo, permitiendo la navegación eficiente durante el parsing.

3.3 Tablas y Resolución de Conflictos

- **Tabla ACTION:** Para cada estado y símbolo terminal, se almacena la acción a realizar (shift, reduce, accept o error).
- **Tabla GOTO:** Para cada estado y no terminal, se almacena el siguiente estado tras una reducción.
- **Precedencia y Asociatividad:** Los conflictos se resuelven usando precedencia y asociatividad configurables, permitiendo un control fino sobre el comportamiento del parser.
- **Tokens Esperados:** Se mantiene, para cada estado, el conjunto de tokens válidos, lo que permite mensajes de error precisos y sugerencias de recuperación.

3.4 Construcción del AST y Acciones Semánticas

- **Dispatcher de Acciones:** El `SemanticActionDispatcher` asocia cada producción con una acción semántica, que puede construir nodos del AST, listas, o realizar validaciones.
- **Pila de Valores:** El parser utiliza una pila de valores polimórficos (`ParserValue`) para manejar tokens, nodos AST y listas de nodos.
- **Integración con el AST:** Cada reducción puede crear o combinar nodos del AST, permitiendo la construcción incremental y modular del árbol sintáctico.

3.5 Ventajas y Características Especiales

- **Modularidad Total:** Cada componente (normalización, augmentación, autómata, tablas, parser) es independiente y extensible.
- **Diagnóstico y Depuración:** El sistema permite imprimir los conjuntos de items, las tablas de parsing y los tokens esperados, facilitando la depuración y el análisis de la gramática.
- **Recuperación de Errores:** El parser implementa mecanismos de recuperación y sincronización, permitiendo continuar el análisis tras errores sintácticos.
- **Extensibilidad:** Es sencillo añadir nuevas reglas, operadores o modificar la gramática, gracias a la separación clara de responsabilidades y la automatización de la normalización y generación de tablas.

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 manualmente en C++ (sin Flex), incluye un sistema preciso de seguimiento de posición y manejo de errores:

- **Seguimiento de Posición**: El lexer mantiene un registro exacto de línea y columna para cada token y error, actualizando estos valores a medida que avanza por el texto fuente.
- **Detección de Errores**:
  - Símbolos no reconocidos (caracteres fuera del alfabeto del lenguaje)
  - Literales de cadena sin cerrar
  - Tokens malformados
- **Reporte y Acumulación de Errores**: Cada vez que se detecta un error, se crea un objeto `LexerError` que almacena un mensaje descriptivo, la ubicación exacta (línea y columna) y el lexema problemático. Todos los errores se acumulan en un vector interno y pueden ser consultados tras el análisis mediante el método `getErrors()`.
- **Precisión**: La información de ubicación se gestiona manualmente, sin depender de herramientas externas, lo que permite mensajes de error claros y útiles para el usuario.

6.2 Errores Sintácticos

El parser, implementado como un autómata LR(1) propio en C++, gestiona los errores sintácticos de manera robusta y extensible:

- **Detección de Errores**: Cuando se encuentra un token inesperado (no existe una acción válida para el estado actual y el token), el parser reporta el error, indicando el token problemático, su ubicación y el conjunto de tokens esperados.
- **Sincronización y Recuperación**: El parser implementa mecanismos de recuperación mediante la búsqueda de tokens de sincronización (como el punto y coma), permitiendo continuar el análisis tras un error y acumular múltiples errores en una sola pasada.

- **Acumulación de Errores**: Todos los errores sintácticos se almacenan en un vector interno y se reportan al finalizar el análisis.
- **Validación de AST**: Tras el análisis, se verifica la validez del árbol sintáctico generado, comprobando la ausencia de nodos nulos o estructuras incompletas, y reportando errores si es necesario.
- **Precisión**: La información de ubicación de los errores se obtiene directamente de los tokens, permitiendo mensajes detallados y precisos.

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.