Compilador HULK

Ernesto Abreu Peraza
Eduardo Brito Labrada
Darío López Falcón

1. Lenguaje HULK

HULK (*Havana University Language for Kompilers*) es un lenguaje de programación didáctico, seguro en tipos, orientado a objetos e incremental, diseñado para el curso Introducción a los Compiladores de la carrera de Ciencias de la Computación en la Universidad de La Habana.

En una visión general, HULK es un lenguaje de programación orientado a objetos con herencia simple, polimorfismo y encapsulación a nivel de clase. Además, en HULK es posible definir funciones globales fuera del alcance de todas las clases. También se puede definir una única expresión global que constituye el punto de entrada al programa.

La mayoría de las construcciones sintácticas en HULK son expresiones, incluyendo instrucciones condicionales y ciclos. HULK es un lenguaje estáticamente tipado con inferencia de tipos opcional, lo que significa que algunas (o todas) las partes de un programa pueden ser anotadas con tipos, y el compilador verificará la consistencia de todas las operaciones.

2. Detalles de implementación

La implementación del compilador de **HULK** se estructura en varias etapas del proceso de compilación, cada una con responsabilidades bien definidas y desarrolladas en el lenguaje de programación `C++`.

El proceso de compilación en HULK se divide en cuatro fases principales:

1. **Análisis léxico**: transforma la secuencia de caracteres del programa fuente en una secuencia de *tokens* significativos para el lenguaje.
2. **Análisis sintáctico**: construye un árbol de sintaxis abstracta (**AST**) validando la estructura gramatical del programa.
3. **Análisis semántico**: verifica la coherencia del programa a nivel de tipos, ámbitos, herencia y accesibilidad de miembros.
4. **Generación de código**: traduce el **AST** validado a código intermedio **LLVM IR** para su posterior compilación a código máquina optimizado.

2.1. Análisis léxico

El análisis léxico del compilador de HULK se implementó mediante un generador de analizadores léxicos desarrollado desde cero en `C++`. Este generador permite construir analizadores léxicos a partir de expresiones regulares definidas para cada tipo de token.

2.1.1. Funcionamiento del motor de expresiones regulares

- **Estructuras de datos:**
  - **State**: representa un estado con 'out' (lista de transiciones).
  - **Transition**: tipo (CHAR, ANY, EPSILON), carácter asociado (para tipo CHAR) y estado destino.
  - **NFA**: par de estados (inicio y aceptación) que definen el autómata (representado como un grafo dirigido).
- **Construcción de Thompson:** compone los NFA mediante operaciones específicas:
  - `char_nfa`: literales
  - `any_nfa`: comodín '.'
  - `concat_nfa`: concatenación con epsilon
  - `alt_nfa`: alternación con nuevo inicio y fin
  - `star_nfa`: bucle para '*'
  - `plus_nfa`: uno o más (implementado usando `star_nfa`)
  - `char_class_nfa`: clases y rangos de caracteres
- **Parser de expresiones:** realiza un análisis sintáctico descendente recursivo para construir el AST de las operaciones regex:
  - `parse()`: invoca `parse_expr()` y verifica el fin del patrón.
  - `parse_expr()`: maneja alternación ('|') y llama a `parse_term`.
  - `parse_term()`: agrupa factores hasta encontrar ')' o '|' y llama a `parse_factor`.
  - `parse_factor()`: aplica operadores '*', '+' o '?' y llama a `parse_primary` cuando corresponde.
  - `parse_primary()`: procesa '()', '.', '[]' y literales, llamando a `parse_expr` cuando es necesario.
- **Simulación del NFA:**
  - `add_state`: agrega un estado y sigue transiciones épsilon (prioritarias por no consumir entrada).
  - `match`: procesa caracteres, transita por CHAR y ANY, y verifica estados de aceptación al final.

2.2. Análisis sintáctico

El parser del compilador HULK implementa un **analizador descendente recursivo** que transforma la secuencia de tokens en un AST. Su diseño sigue la estructura gramatical del lenguaje HULK y se organiza en dos componentes principales:

1. **Funcionamiento General:**
   - **Entrada**: Recibe tokens del analizador léxico (identificadores, palabras clave, operadores, etc.)
   - **Proceso**: Implementa métodos especializados para cada construcción del lenguaje:
     - Declaraciones (funciones, clases, protocolos)
     - Expresiones (operaciones, llamadas, estructuras de control)
     - Sentencias (asignaciones, bloques)
   - **Salida**: Genera un AST estructurado para el análisis semántico
2. **Características principales:**
   - **Manejo de errores**: Detecta y reporta inconsistencias sintácticas con mensajes descriptivos
   - **Precedencia de operadores**: Resuelve correctamente expresiones complejas mediante funciones anidadas, teniendo en cuenta la gramática escrita para el lenguaje en `grammar_LL1.txt`
   - **Tipado**: Soporta anotaciones de tipo opcionales en declaraciones

2.3. Análisis semántico

2.4. Generación de código