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

El análisis semántico del compilador HULK se encarga de verificar la coherencia del programa a nivel de tipos, ámbitos, herencia y accesibilidad de miembros. Para esto se recorre el AST generado por el analizador sintáctico llevando un contexto del programa a cada nodo visitado, tendiendo en cuenta 4 enfoques principales:

- **Context Builder**: Construye un contexto con información sobre el ámbito global, incluyendo tipos, funciones y protocolos.
- **Scoped Visit**: Verifica la validez de las declaraciones y expresiones dentro de su ámbito, asegurando que los identificadores estén definidos y sean accesibles.
- **Type Inference**: Realiza la inferencia de tipos para las expresiones, determinando el tipo de cada expresión según las reglas del lenguaje y las declaraciones previas. Anota variables y argumentos con los tipos inferidos en caso de no estar anotados explicitamente.
- **Type Checker**: Verifica la coherencia de los tipos en las expresiones y declaraciones, asegurando que las operaciones sean válidas según las reglas del lenguaje.

1. **Context:**
   - Representa el contexto del programa, incluyendo tipos, funciones y protocolos.
   - Permite la creación de ámbitos anidados para tipos, funciones y expresiones.
   - Facilita la resolución de identificadores y la verificación de accesibilidad.
   - Los protocolos se manejan como tipos a nivel sémantico.
2. **Context Builder:**
   - Construye el contexto global con los tipos y funciones predefinidos.
   - Recorre el AST para identificar y registrar las declaraciones de tipos, funciones y protocolos.
   - Valida que los tipos anotados en las firmas de las declaraciones.
   - Valida la herencia de tipos.
3. **Scoped Visit:**
   - Asegura que las declaraciones de variables y funciones estén en el ámbito correcto.
   - Asegura que los identificadores estén definidos y sean accesibles.
   - Verifica tipos anotados en las expresiones.
4. **Type Inference:**
   - Recorre el AST para inferir los tipos de las expresiones. Este proceso se repite mientras se anoten tipos nuevos. Como la cantidad de tipos a anotar es finita, el proceso termina eventualmente.
   - Anota variables y argumentos con los tipos inferidos en caso de no estar anotados explicitamente. Si no se pueden anotar todos, se reporta un error para que sean anotado explicitamente.
   - Determina el tipo de cada expresión según las reglas del lenguaje y las declaraciones previas.
5. **Type Checker:**
   - Verifica la coherencia de los tipos en las expresiones y declaraciones.
   - Asegura que las operaciones sean válidas según las reglas del lenguaje.
   - Verifica que las llamadas a funciones y métodos sean consistentes con sus firmas.
   - Verifica que las asignaciones sean compatibles con los tipos de las variables.

- Asegura que las expresiones condicionales y de control de flujo tengan tipos válidos.
- Valida que las clases y protocolos se usen correctamente
- Valida la coherencia cuando se sobreescriben métodos, misma firma para tipos, y covariacia y contravariancia para protocolos.
- Asegura que las asignaciones sean compatibles con los tipos de las variables.

2.4. Generación de código

La generación de código del compilador HULK se basa en la generación de código intermedio a **LLVM IR**. Esta fase toma el AST validado por el análisis semántico y lo traduce a el lenguaje intermedio LLVM, usando la API que LLVM proporciona para hacer generación de código desde C++, sin necesidad de escribir el código directamente, el cual luego es compilado a código máquina usando el compilador clang++-19.

1. **Tipos:**
   - **Number**: Representa enteros y flotantes, mapeados a `double` en LLVM.
   - **Boolean**: Representa valores booleanos, mapeados a `i1` en LLVM.
   - **String**: Representa cadenas de texto, mapeadas a `[N x i8]` en LLVM.
   - **Types**: Representa estructuras de datos complejas, mapeadas a `struct` en LLVM.
2. **Variables y asignaciones:** Las variables se representan como registros en LLVM, y las asignaciones se traducen a instrucciones de movimiento de datos. Se usa la instrucción **alloca** para reservar espacio de memoria en la pila para las variables, `store` para asignarles valores y `load` para leer sus valores.
3. **Control de flujo:** Las estructuras de control (condicionales, ciclos) se traducen a instrucciones de salto y bifurcación en LLVM.
4. **Funciones:** Las funciones se representan como bloques de código en LLVM, con sus parámetros y tipos de retorno definidos. Se usa la instrucción `call` para invocar funciones y `ret` para retornar valores.
5. **Tipos y objetos:**
   - Los tipos de datos complejos (clases, estructuras) se representan como `structs` en LLVM. Las instancias de objetos se crean usando la instrucción `alloca` para reservar espacio en la pila y se inicializan con los valores correspondientes.
   - Los campos de los objetos se acceden mediante la instrucción `getelementptr`, que permite calcular la dirección de un campo específico dentro de un objeto.
   - La herencia se maneja creando la estructura base y las estructuras derivadas, donde los campos de la clase base se incluyen al inicio de la estructura derivada. Esto permite acceder a los campos de la clase base desde la clase derivada.
   - Los métodos se representan como funciones en LLVM, y se accede a ellos mediante la vtable de métodos. Cada clase tiene su propia vtable, que es un array de punteros a funciones. Cuando se invoca un método, se busca en la vtable correspondiente a la instancia del objeto.
   - La vtable se genera también colocando las direcciones de los métodos de la clase base primero, y luego los de la clase derivada, sobreescribiendo métodos de la clase base en caso de ser necesario. Esto permite que al invocar un método desde una instancia de la clase derivada, se acceda al método correcto según el tipo de la instancia.