Informe de Funcionamiento del Compilador HULK

Desarrollado en C, sin uso de librerías externas

Melani Forsythe Matos
Orlando de la Torre Leal
Olivia Ibañez Mustelier

1 Introducción

El presente informe describe el funcionamiento del compilador desarrollado para el lenguaje HULK.

El compilador ha sido implementado **en C**, sin el uso de librerías externas, empleando únicamente estructuras, algoritmos y rutinas desarrolladas por los propios autores.

Su diseño modular abarca los siguientes componentes:

- Un lexer basado en autómatas DFA.
- Un generador de parser LALR(1).
- Un chequeador semántico.
- Un generador de código LLVM IR optimizado.

2 Lexer: Análisis Léxico

2.1 Estructura General

- `generar_lexer.c`: Generación automática de lexer en C.
- `regex_parser.c`: Parser de expresiones regulares a NFA.

- `nfa_to_dfa.c`: Conversión de NFA a DFA.
- `regex_to_dfa.c`: Funciones simplificadas para crear DFA.
- `utils.c`: Funciones utilitarias y gestión de estados.
- `lexer.c`: Lexer generado que procesa texto de entrada.

2.2 Proceso de Generación

1. **Lectura de tokens**: `leer_tokens()` lee las definiciones en `tokens.def`, que describe los tokens como expresiones regulares.
2. **Conversión a NFA y DFA**: Cada expresión regular se convierte primero en un NFA (autómata no determinista) y luego en un DFA (autómata determinista).
3. **Generación de C**: `generar_lexer.c` escribe el código C que contiene la implementación de los DFA para cada token, generando funciones como `next_token()`.

2.3 Funcionamiento del Lexer

El lexer generado:

- Procesa la entrada línea por línea.
- Ignora espacios en blanco, tabs y saltos de línea.
- Escanea la entrada con cada DFA en orden de prioridad, seleccionando el token con la coincidencia más larga (Maximal Munch).
- Soporta el reconocimiento de literales numéricos, cadenas (con escapes), palabras clave, identificadores, operadores, delimitadores y comentarios.
- Devuelve un struct `Token` con los campos: tipo, lexema, longitud, línea, columna.
- En caso de error léxico (símbolo no reconocido), devuelve un token especial `TOKEN_ERROR`.

2.4 Tokens Soportados

- Palabras clave: `if`, `else`, `while`, `for`, `return`, `let`, `type`, `protocol`, etc.
- Operadores aritméticos: `+`, `-`, `*`, `/`, `%`, `^`.
- Operadores de comparación: `==`, `!=`, `<`, `<=`, `>`, `>=`.
- Operadores lógicos: `&&`, `||`, `!`.
- Operadores de asignación: `=`, `:=`.
- Delimitadores: `(`, `)`, {, }, `[`, `]`, `,`, `;`.
- Literales: números enteros y flotantes, booleanos (`true`/`false`), cadenas con escapes.
- Comentarios: comentarios de línea y comentarios multilínea.

3 Generador de Parser LALR(1)

Informe Profesional: Generador de Parser LR(1)

Un generador de parser es una herramienta especializada que, a partir de la descripción formal de una gramática (conjunto de reglas de producción, terminales y no terminales), produce automáticamente el código necesario para analizar cadenas de símbolos y determinar si pertenecen al lenguaje definido. En esencia, convierte la gramática en un autómata de parsing que dirige un proceso de "shift-reduce", construyendo un Árbol de Síntaxis Abstracta (AST) que refleja la estructura jerárquica de la entrada.

Flujo de Datos

- **Carga de Gramática**: Lectura de un archivo de gramática con secciones de no terminales, terminales, símbolo inicial y producciones; conversión a estructuras `Grammar`, `Symbol` y `Production`.
- **Cálculo de FIRST y FOLLOW**: Obtención de los conjuntos FIRST (símbolos iniciales posibles) y FOLLOW (símbolos que pueden seguir a cada no terminal), esenciales para lookaheads LR(1).
- **Construcción del Autómata de Items**: Generación de items LR(1) (producción con punto y lookaheads) y aplicación de clausura y transición para formar los estados del autómata.

- **Generación de Tablas ACTION y GOTO**: Extracción de las tablas que dictan, en cada estado y ante cada símbolo, si se realiza shift, reduce, accept o transición de estado.
- **Parsing Shift-Reduce**: Ejecución del algoritmo sobre la pila de estados y símbolos, aplicando las tablas para desplazar tokens o reducir producciones, invocando builders de nodos AST.
- **Salida de AST**: Resultado final en memoria: un AST que servirá para chequeo semántico o generación de código.

Componentes de la Carpeta `grammar/`

- `grammar.c`: Gestión de la estructura `Grammar`, incluyendo mecanismos de creación, búsqueda y liberación de símbolos y producciones.
- `load_grammar.c`: Funciones para parsear la representación textual de la gramática y poblar las estructuras de C.
- `production.c`: Representación y administración de cada producción (lado izquierdo y secuencia de símbolos derecho).
- `symbol.c`: Definición de `Symbol` (terminal, no terminal, ϵ, `EOF`), manejo de unicidad y utilidad para comparación e impresión.

Componentes de la Carpeta `parser/`

- `first_follow.c`: Algoritmo iterativo para calcular conjuntos FIRST y FOLLOW, base para lookaheads y resolución de conflictos.
- `item.c`: Modela el concepto de item LR(1): posición del punto en la producción y conjunto de lookaheads.
- `containerset.c`: Estructura de datos para representar conjuntos de símbolos, con operaciones de inserción y combinación.
- `state.c`: Construcción de estados del autómata como conjuntos de items y gestión de transiciones.
- `automaton.c`: Implementa clausura y transición de items para generar el autómata completo.
- `lr1_table.c`: Genera las tablas ACTION y GOTO a partir del grafo de estados.

- `parser.c`: Ejecuta el parsing shift-reduce usando las tablas, gestiona la pila y construye el AST.

El parser inicialmente diseñado utiliza la técnica LR(1), que ofrece un análisis sintáctico muy potente al considerar lookaheads explícitos para cada producción, reduciendo al máximo los conflictos. Sin embargo, los parsers LR(1) tienden a generar tablas de gran tamaño y, en nuestra implementación, provocaban desbordamientos de pila al manejar gramáticas complejas. Por ello, se adoptó un parser LALR (Look-Ahead LR) que fusiona estados compatibles del autómata LR(1), reduciendo significativamente la cantidad de estados y el tamaño de las tablas. Esta adaptación mantiene la capacidad de detección de errores y resolución de conflictos de LR(1) mientras mejora la eficiencia en memoria y velocidad de parsing, siendo adecuada para implementaciones de gramáticas de complejidad moderada, como la nuestra.

4 Chequeo Semántico

El análisis semántico verifica que el código fuente cumpla con las reglas de significado del lenguaje HULK, asegurando la corrección de tipos, alcance de variables y estructura del programa.

Componentes Principales
Type Collector: Función: `collect_types()` Recorre el AST para recolectar declaraciones de tipos y protocolos Verifica que no haya redefiniciones de tipos built-in Crea estructuras básicas para tipos y protocolos en el contexto Inicializa tipos y funciones built-in de HULK (`Number`, `String`, `Boolean`, `print`, `sqrt`, etc.) Crea el protocolo `Iterable`

Type Builder Función: `build_types()`
Responsabilidad: Completa la información de los tipos recolectados Verifica y establece: - Parámetros de tipos - Atributos - Métodos - Herencia (incluyendo validación de herencia circular)
Procesa declaraciones de funciones globales Maneja protocolos y sus métodos

Type Checker Función: `type_check_program()`
Responsabilidad: Verifica el correcto uso de tipos en expresiones Gestiona el alcance de variables (`scope`) Comprueba: - Compatibilidad de tipos en operaciones - Correcto uso de métodos y atributos - Conformidad con protocolos - Correcto número y tipo de argumentos en llamadas - Maneja casos especiales como `self` y `base`

Flujo de Ejecución

Recolección de Tipos: Crea el contexto inicial con tipos básicos Registra todas las declaraciones de tipos/protocolos

Construcción de Tipos: Completa la información de cada tipo Establece relaciones de herencia Valida estructura de tipos y protocolos

Verificación Semántica: Recorre el AST verificando reglas semánticas Anota tipos en nodos del AST Reporta errores de tipo, alcance y uso

Manejo de Errores

Cada fase genera sus propios errores semánticos. Los errores incluyen: - Tipos no definidos - Incompatibilidad de tipos - Redefiniciones - Uso incorrecto de herencia - Violaciones de alcance - Argumentos incorrectos - Etc.

Los errores se reportan con ubicación (línea, columna).

Estructuras Clave

`Context`: Mantiene el estado de tipos, protocolos y funciones
`Scope`: Maneja el alcance de variables durante el type checking
`HulkErrorList`: Acumula y gestiona los errores encontrados

5 Generación de Código LLVM IR

5.1 Estructura de los Módulos

- `generacion_codigo.c`: Generador principal de LLVM IR.
- `aux_generacion.c`: Funciones auxiliares para generación.
- `optimizador.c`: Optimizaciones de constantes.
- `entorno.c`: Gestión de variables, etiquetas, temporales.
- `runtime.c`: Funciones auxiliares de runtime (print, conversión de tipos, concatenación).
- `soporte_llvm.c`: Manejo de strings globales y funciones externas.

5.2 Proceso de Generación

1. **Optimización**: Aplicación de `optimizar_constantes()` al AST (constant folding, propagación de constantes).
2. **Recorrido del AST**: Generación recursiva de instrucciones LLVM IR para cada nodo.

3. **Gestión de Variables y Temporales**: Asignación de `alloca`, registros temporales `%tN` y etiquetas `L_N`.
4. **Control de flujo**: Traducción de estructuras condicionales (`if-else`), bucles (`while`, `for`).
5. **Funciones**: Declaración y llamada a funciones, soporte para funciones con parámetros y retorno.
6. **Cadenas**: Manejo de strings como constantes globales con soporte de `strdup`, `strcat2`.
7. **Conversión de tipos**: `int_to_string`, `float_to_string`, `bool_to_string`.

5.3 Funciones y Construcciones Soportadas

- Literales: enteros, flotantes, booleanos, cadenas.
- Variables: declaración, asignación, acceso.
- Operadores aritméticos: suma, resta, multiplicación, división, módulo, potencia.
- Operadores lógicos: `AND`, `OR`, `NOT`.
- Comparaciones: igualdad, desigualdad, mayor, menor, etc.
- Concatenación de cadenas.
- Bloques de código ({ }).
- Control de flujo: `if-else`, `while`, `for`.
- Funciones: declaración, parámetros, retorno, llamadas.
- Métodos y atributos de tipos definidos por el usuario.
- Declaracion de vectores y indexación.
- Manejo de estructuras `type`.
- Runtime: impresión de enteros y cadenas (`print_int`, `print_str`).
- Conversión implícita de tipos cuando es posible.

5.4 Optimización

- **Constant Folding**: operaciones binarias con operandos constantes son evaluadas en tiempo de compilación.
- **Propagación de constantes**: valores constantes propagados en bloques `let-in`.
- **Simplificaciones algebraicas**: eliminación de operaciones redundantes (multiplicación por 1, suma de 0).
- **Advertencias**: división o módulo por cero.

5.5 Ejemplo de Código Generado (LLVM IR)

```
define i32 @main() {
entry:
  %0 = add i32 0, 5
  %1 = add i32 0, 10
  %2 = add i32 %0, %1
  call void @print_int(i32 %2)
  ret i32 0
}
```

6 Resumen

El compilador HULK permite transformar código fuente en LLVM IR optimizado, partiendo de un análisis léxico con DFA, un parser LALR(1), un verificador semántico robusto y un generador de código eficiente.