JavaLang Interpreter

Informe Técnico

Universidad San Carlos de Guatemala
Facultad de Ingeniería — Ingeniería en Ciencias y Sistemas
Organización de Lenguajes y Compiladores 2

Autor: Juan Esteban Chacón Trampe — 202300431
Título del Proyecto: JavaLang Interpreter
Fecha: 2025-09-17

Introducción

Este documento describe la construcción de un intérprete académico denominado JavaLang Interpreter. A continuación Se detallara la gramática del lenguaje, las decisiones de diseño, la arquitectura por capas (lexer, parser, AST, semántico y GUI), así como los retos afrontados y los resultados de las pruebas. El objetivo es documentar de forma clara y el porque de cada componente contribuye a una ejecución correcta y satisfactoria del lenguaje, manteniendo coherencia entre línea de comandos y la interfaz gráfica.

Resumen

Se presenta el diseño e implementación de un intérprete para el lenguaje JavaLang, construido con Flex/Bison y un runtime en C. Este Incluye una interfaz gráfica en GTK para edición y ejecución de archivos .usl, generación de AST y reportes de símbolos/errores. El documento resume la gramática formal, arquitectura, módulos y pruebas.

Objetivos y alcance

- Implementar un intérprete académico con sintaxis estilo Java, suficiente para ejercicios de OLC2.
- Proveer una GUI para editar/ejecutar y visualizar reportes (AST, símbolos, errores).
- Asegurar coherencia semántica entre CLI y GUI, con manejo correcto de codificaciones.
- Mantener el código modular: lexer/parser/AST/semántico GUI desacoplados.

1. Gramática formal y lenguaje soportado

- Expresiones aritméticas, lógicas y relacionales; precedencia y asociatividad según Java.
- Literales: enteros, flotantes, booleanos, char ('a', \uXXXX), string, null.
- Casts de tipo en posición postfix (alta precedencia) y conversión numérica.
- Control: if/else, while, for, for-each en arreglos, switch/case/default con etiquetas apiladas.
- Funciones builtin: String.valueOf(x), longitud de arrays, entre otras utilidades mínimas.

La gramática se define en `entriesTools/parser.y`. Fragmentos ilustrativos (resumen):

- Expresiones y casts:

    - `postfix: primary | postfix '(' arglist? ')' | '(' tipo ')' postfix` (cast con mayor precedencia)
    - `expr: expr '+' expr | expr '-' expr | expr '*' expr | expr '/' expr | ...`
- Control de flujo:
    - `if_stmt: IF '(' expr ')' stmt (ELSE stmt)?`
    - `switch: SWITCH '(' expr ')' '{' case_list default_opt '}'`
    - `case_list: case_list CASE const ':' | case_list DEFAULT ':' | /* vacio */`
- Declaraciones y arrays:
    - `decl: tipo ID ('=' expr)? ('[' expr ']')* ';'`
    - `foreach: FOR '(' tipo ID ':' expr ')' stmt`

Tipos de datos y conversiones

- Primitivos: BYTE, SHORT, INT, FLOAT, DOUBLE, CHAR, BOOL, STRING, y NULO.
- Arrays: de dimensión n, con utilidades de indexación y longitud.
- Conversión implícita: jerarquía numérica ancha (p. ej., INT → DOUBLE) y explícita via cast.
- Comparación con `null`: solo igualdad/desigualdad; otras operaciones con `null` son error.

Precedencia y asociatividad (resumen de mayor a menor)

1. Postfix: `()`, indexación `[]`, acceso; Cast `(T)expr` (elevado sobre postfix para evaluar temprano el tipo)
2. Unarios: `+`, `-`, `!`, bitwise unario si aplica
3. Multiplicativos: `*`, `/`, `%`
4. Aditivos: `+`, `-`
5. Shift: `<<`, `>>`, `>>>` (si aplica)
6. Relacionales: `<`, `<=`, `>`, `>=`
7. Igualdad: `==`, `!=`
8. AND bit a bit, XOR, OR (si aplica)
9. Lógicos: `&&`, `||`
10. Ternario (si aplica), Asignación

2. Arquitectura general

- Lexer (Flex): `entriesTools/lexer.l`
- Parser (Bison): `entriesTools/parser.y`
- AST y nodos: `src/ast/**`
- Contexto/semántico (símbolos, tipos, resultados): `src/context/**`
- Ejecutable CLI: `build/calc`
- GUI (GTK): `gui/editor.c`, `gui/run.c`

Flujo: Código USL → Lexer → Parser (AST) → Evaluación sobre contexto (tabla de símbolos, tipos) → E/S y reportes → GUI opcional.

Justificación de diseño

- Separación clara de etapas permite pruebas y mantenibilidad por módulo.
- AST fuertemente tipado simplifica evaluaciones y generación de reportes.
- Contexto/semántico centraliza reglas de tipos, símbolos y errores.
- GUI opera como cliente del ejecutable CLI para mantener paridad funcional.

3. Módulos principales

3.1 Lexer (lexer.l)

- Tokens para identificadores, literales, operadores y palabras clave.
- Manejo de \uXXXX en literales char/string; normalización de saltos de línea.
- Control de comentarios y espacios; reporte de líneas/columnas para diagnósticos. Justificación:
- Resolver \uXXXX en el lexer simplifica el parser y mantiene consistencia de codificación.
- La trazabilidad (línea, columna) es clave para `ERR|` y UX en GUI.

3.2 Parser (parser.y)

- Precedencia ajustada: cast por encima de postfix para castear correcto.
- Switch completo: casos apilados, default único, control de break/continue/return.
- Inclusión de literal `null` y comparaciones con `==`/`!=`.
- For-each sobre arreglos (apoyado por utilidades de arrays).
- Uso de posiciones (línea, columna) para enriquecer reportes de errores y símbolos. Justificación:
- Elevar cast evita ambigüedades comunes en expresiones complejas (p. ej., `(int)a[0]`).
- Switch con apilamiento refleja el comportamiento esperado de etiquetas contiguas.
- For-each aporta ergonomía con arrays, alineado a ejercicios de laboratorio.

3.3 AST (`src/ast/**`)

- `AbstractExpresion.*` y builders de nodos.
- Aritméticas, lógicas, relacionales, terminales (identificador, primitivo), cast, listas.
- Builtins: `String.valueOf` con formato de dígitos significativos, manejo de -0.0.
- Tablas de operaciones por combinación de tipos (suma, división); coerción controlada. Nodos relevantes (ejemplos):
- `nodo_suma`, `nodo_division`, `nodo_relacional`, `nodo_logico`, `nodo_cast`.
- `nodo_identificador`, `nodo_literal_*`, `nodo_lista_expresiones`.
- Instrucciones: `nodo_if`, `nodo_bloque`, `nodo_while`, `nodo_for`, `nodo_switch`.

Contrato de evaluación (resumen):

- Entrada: contexto (símbolos, tipos), nodo AST.
- Salida: `Result` con tipo, valor y marca de control (normal, break, continue, return).
- Errores: se reportan con `ERR|desc|ambito|line|col|Tipo` sin abortar el proceso cuando es posible.

3.4 Semántico y contexto (`src/context/**`)

- Tabla de símbolos con ámbitos, tipos, valores por defecto, utilidades de conversión.
- Result/TipoDato: soporta NULO/ARRAY y control de flujos.
- Utilidades de arrays: indexación multidimensional, longitud total/superficial.

- Manejadores de errores con prefijo `ERR|` para consumo por la GUI. Símbolos y ámbitos:
- Tabla jerárquica (bloque anidado) con resolución por sombras.
- Inserción de declaraciones sin inicializador con valores por defecto (según tipo).
- Tipado: utilidades `tipo_utils.*` y conversiones numéricas en `conversion_utils.*`.

Arrays:

- Soporte multidimensional con utilidades de dimensiones y aplanamiento cuando conviene.
- Validación de límites e índices enteros; reporte de errores precisos.

3.5 GUI (GTK)

- Editor con TextView, consola, atajos (F5 ejecutar, F6 AST, F9 errores, F10 símbolos).
- Reportes: genera DOT/PNG del AST; tablas de errores y símbolos.
- Normalización de salida a UTF-8 al mostrar en consola/diálogos para evitar GTK-CRITICAL.
- Captura y combinación de stdout/stderr del intérprete; parseo de líneas especiales (ERR|, SYM|). Detalles de implementación:
- `gui/run.c` invoca `./build/calc` con el buffer actual (guardado a archivo temporal).
- Consolida stdout+stderr y los muestra en `GtkTextView` inferior.
- Normaliza al insertar: valida UTF-8 y, si llegan bytes 0x80–0xFF (Latin-1), los mapea a U+0080..U+00FF y emite UTF-8, manteniendo controles ASCII como `\uXXXX`.
- Diálogos de Errores/Símbolos usan `GtkTreeView` con columnas parseadas de `ERR|` y `SYM|`.

4. Retos técnicos y soluciones

- Precedencia de cast vs. postfix → Se elevó cast en la gramática.
- Switch con múltiples etiquetas y default → Pila de etiquetas y control completo.
- Formato de números (String.valueOf) → Dígitos significativos estilo Java; suprimir -0.0.
- Suma numérica/concatenación → Tablas por tipo y correcciones en combinaciones int/double.
- Null y comparaciones → Literal `null` y lógica de igualdad segura.
- UTF-8 en GUI → Se agregó capa de normalización/escape en GUI; no se modificó semántica del intérprete.
- Índices de arreglos multidimensionales → Validación de límites y utilidades de flatenning.
- Warnings de compilación (C) → Limpieza de indentación engañosa y mapas duplicados. Justificación y aprendizajes:
- La elevación de cast reduce conflictos del parser y errores semánticos aguas abajo.
- La tabla de suma/división por tipo evita cascadas de `if` y facilita mantenimiento.
- La normalización UTF-8 en GUI evita dependencias de locale y mantiene paridad con CLI.

5. Construcción y estructura del proyecto

- `Makefile` compila lexer/parser, AST, contexto y GUI; genera `build/calc` y `build/editor`.
- Estructura:
  - `entriesTools/`: `lexer.l`, `parser.y`
  - `src/ast/**`: nodos, builders, `ast_to_dot.*`
  - `src/context/**`: símbolos, tipos, utils
  - `gui/**`: editor, run, estilos
  - `reportes/`: salidas AST

6. Limitaciones conocidas

- No se implementan objetos/clases completos ni genéricos.
- Operadores bit a bit avanzados y ternario pueden estar limitados según alcance del curso.
- Dependencia de Graphviz para PNG del AST (opcional).

7. Referencias rápidas a archivos

- Gramática: `entriesTools/parser.y`
- Lexer: `entriesTools/lexer.l`
- Builtins y formato numérico: `src/ast/nodos/expresiones/builtins.c`
- Suma/división: `src/ast/nodos/expresiones/aritmeticas/suma.c`, `division.c`
- Contexto/errores: `src/context/error_reporting.*`, `context.*`
- GUI: `gui/editor.c`, `gui/run.c`

8. Conclusiones

El intérprete cumple con la ejecución de programas JavaLang de alcance académico, con una GUI práctica y reportes útiles. Se priorizó la corrección semántica y la robustez de la salida en entornos UTF-8.