Informe Técnico — Proyecto1 JavaLang

Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Curso: Organización de Lenguajes y Compiladores 2

Nombre: Kevin Brayant Jiménez Castañeda
Fecha: 17 de septiembre de 2025

1. Introducción

El proyecto JavaLang consiste en el diseño e implementación de un compilador-intérprete académico para un lenguaje inspirado en Java. Este informe documenta la gramática formal definida, la arquitectura del sistema, la implementación de cada módulo, los retos técnicos y las soluciones aplicadas, así como los resultados de pruebas y conclusiones. El propósito principal es aplicar en la práctica los conceptos de teoría de compiladores y demostrar un flujo completo de procesamiento de un lenguaje.

La versión actual de JavaLang soporta:
- Declaración de variables enteras y operaciones aritméticas.
- Estructuras de control if/else, while, do while y switch-case.
- Instrucciones de control de flujo como break y continue.
- Operadores de incremento y decremento (++ y --).
- Impresión de salida con System.out.println.
- Integración de una GUI para abrir, guardar, limpiar y ejecutar programas.

Sin embargo, la implementación aún no contempla características avanzadas como arreglos, clases, funciones definidas por el usuario, manejo de tipos compuestos u optimizaciones de código. Estas se consideran líneas de trabajo futuro.

La metodología empleada fue incremental y modular. El desarrollo inició con la definición de la gramática formal y su implementación en Bison, complementada por el análisis léxico en Flex. Posteriormente se diseñó la estructura del AST y se implementaron los recorridos semánticos y de ejecución. Finalmente, se integró una interfaz gráfica con GTK que actúa como entorno de pruebas y facilita la interacción con el compilador.

Cada módulo fue validado mediante casos de prueba unitarios y posteriormente se realizaron pruebas de integración que incluyeron programas de ejemplo completos. De esta forma, se aseguró la correcta interacción entre lexer, parser, AST, semántico y GUI.

Objetivos

- Desarrollar un pipeline que incluya análisis léxico, sintáctico y semántico.
- Implementar un ejecutor basado en AST.
- Proporcionar una interfaz gráfica amigable para ejecutar programas.
- Evaluar el desempeño mediante pruebas y métricas de cobertura.

2. Gramática Formal de JavaLang

La gramática del lenguaje se definió en formato EBNF. Producciones principales:

```
Program       ::= 'public' 'static' 'void' 'main' '(' ')' Block
Block         ::= '{' Statement* '}'
Statement     ::= Assignment | IfStmt | WhileStmt | DoWhileStmt | PrintStmt |
SwitchStmt
Assignment    ::= Type Identifier '=' Expression ';'
Expression    ::= Term (('+' | '-') Term)*
Term          ::= Factor (('*' | '/') Factor)*
Factor        ::= Identifier | Number | '(' Expression ')'
```

Extensiones incluidas:
- do-while como estructura repetitiva.
- switch-case con break y default.
- Operadores de incremento y decremento (++/--).
- Instrucciones break y continue dentro de bucles.

Ejemplo de derivación:
Input: int a = 5 + 3;
Derivación: Assignment → Type Identifier '=' Expression ';' → int a = Expression ';' → int a = Term '+' Term ';' → int a = Factor + Factor ';' → int a = 5 + 3 ;

3. Arquitectura General

La arquitectura sigue el diseño clásico front-end → middle-end → back-end de un compilador–intérprete, integrando además una GUI que actúa como fachada y orquestador de la ejecución.

Flujo principal:
1. La GUI obtiene el código del editor o de un archivo .usl.
2. El lexer tokeniza el texto.
3. El parser construye el AST y coordina acciones semánticas iniciales.
4. El módulo semántico valida tipos/ámbitos y acumula errores.
5. El ejecutor recorre el AST y produce la salida.
6. La GUI muestra OUTPUT y LOG.

3.2.1 GUI (GTK+3)

- Responsabilidades: captura de entrada, comandos “Analizar & Ejecutar”, “Abrir”, “Guardar”, “Limpiar”; visualización de salida y log.
- Integración con el parser: usa yy_scan_string para alimentar el lexer con el contenido del editor sin archivos temporales.
- API interna empleada:
  • parser_reset() — limpia AST, símbolos y bitácora.
  • yy_scan_string(const char*) / yy_delete_buffer(...) — inyecta el código fuente al lexer.
  • yyparse() — dispara el análisis.
  • parser_get_log() — recupera mensajes (errores/advertencias).
  • exec_get_output() — recupera la salida de ejecución.
  • parser_get_error_count() — decide si se ejecuta o solo se muestra log.

Lexer (Flex)

- Entrada/Salida: recibe una cadena (buffer) y emite una secuencia de tokens al parser.
- Reglas clave: identificadores, literales (número, cadena, char), palabras reservadas, operadores, separadores, comentarios //.

- Diseño: expresiones regulares por token; manejo explícito de whitespace y newlines para localización de errores (línea/columna).

Parser (Bison)

- Entrada/Salida: consume tokens y construye un AST; reporta sintaxis inválida.
- Estrategia: gramática en EBNF/BNF con precedencias y asociatividades para evitar/mitigar shift/reduce.
- Acciones semánticas tempranas: construcción de nodos AST y acoplamiento con tabla de símbolos.
- Recuperación de errores: producción especial y/o yyerrok para seguir analizando.
- Extensibilidad: nuevas producciones para switch, do while, ++/-- (prefijo/sufijo), y operadores lógicos/relacionales.

AST (Árbol de Sintaxis Abstracta)

- Modelo de datos (C):
  • NodeKind kind; (tipo de nodo: NK_Assign, NK_If, NK_While, NK_BinOp, NK_Unary, NK_Lit, NK_Var, etc.).
  • Campos específicos por nodo (operador, hijos, identificador, valor, lista de sentencias, etc.).

Semántico (símbolos, tipos, ámbitos)
- Tabla de símbolos (TS): estructura hash o tabla por niveles de ámbito.
- Reglas implementadas:
  • Declaraciones: inserción en TS; detección de redeclaración.
  • Uso: verificación de variable declarada antes de usar.
  • Tipificación: coerciones permitidas/prohibidas (en esta versión centrado en int).
  • Validación de break/continue solo dentro de bucles/switch.
  • Consistencia en switch (duplicados de case, existencia/uso de default).
- Errores: acumulados en un logger central (parser_get_log()).

Gestión de errores y logging
- Léxico: tokens inválidos → log con línea/columna.
- Sintaxis: recuperación para continuar y detectar más errores.
- Semántico: mensajes precisos (tipo incompatible, símbolo no declarado, break fuera de bucle, etc.).
- Runtime: errores de ejecución (p. ej., división por cero).
- Salida unificada: la GUI prioriza OUTPUT cuando no hay errores; siempre muestra LOG si existe.

4. Implementación de Módulos

Lexer: Ejemplo de regla en lexer.l:

```
ID    [A-Za-z_][A-Za-z0-9_]*
{ID}   { return ID; }
```

Parser: Ejemplo de regla en parser.y:

Assignment: TYPE ID '=' Expression ';' { /* acción semántica */ };

AST: Estructura en C:

```
typedef struct Node {
   char* type;
   struct Node* left;
   struct Node* right;
} Node;
```

Semántico: Tabla de símbolos y validaciones de tipos (ej. suma entre enteros, asignación a variables declaradas).

GUI: Interfaz en GTK con botones Abrir, Guardar, Limpiar y Analizar & Ejecutar.

5. Retos Técnicos y Soluciones

- Manejo de memoria del AST: funciones para liberar nodos.
- Integración con GUI: uso de yy_scan_string.
- Internacionalización numérica: setlocale(LC_NUMERIC, 'C').

6. Resultados de Pruebas

Programa	Resultado esperado	Resultado obtenido
int a=5; int b=10; System.out.println(a+b);	15	15
do { a++; } while(a<3);	a=3 al final	a=3 al final

7. Conclusiones

Se alcanzó un compilador-intérprete funcional con frontend, semántico y backend. Se validó la gramática mediante pruebas y se integró una GUI sencilla. Los principales retos fueron conflictos en la gramática y gestión de memoria. Como trabajo futuro: arrays, funciones definidas por el usuario y sistema de clases.

8. Referencias

- Aho, A., Lam, M., Sethi, R., Ullman, J. (2006). Compilers: Principles, Techniques, and Tools.
- Documentación oficial de Flex y Bison.
- Documentación de GTK+ 3.