

MANUAL TECNICO

Valery Galvez - 202200141
Organizacion de lenguajes y compiladores 2

INTRODUCCION

Este manual técnico describe el diseño, desarrollo e implementación del proyecto JavaLang Interpreter, un intérprete de un lenguaje inspirado en Java, construido con Flex, Bison y C. Aplicando así los fundamentos de análisis léxico, sintáctico, semántico y de ejecución de código.

En las siguientes secciones se detalla la gramática formal utilizada, la arquitectura general del sistema, la implementación de cada uno de los módulos principales, los retos técnicos enfrentados y las soluciones aplicadas, así como los resultados de pruebas realizadas y las métricas de cobertura obtenidas.

REQUISITOS

- Sistema operativo: GNU/Linux Ubuntu 20.04 o superior (se recomienda Linux por compatibilidad con Flex/Bison)
- Compilador: C, GCC (GNU Compiler Collection)
- Herramientas de análisis:
 - Flex (para análisis léxico)
 - Bison (para análisis sintáctico)
 - Librerías gráficas: GTK+ 3
 - Utilidades:
 - Make (automatización de compilación)
- Git (control de versiones, opcional)

GRAMÁTICA FORMAL Y ARQUITECTURA GENERAL

- Gramática Formal

La gramática del lenguaje JavaLang se define en notación BNF extendida, dejo algunas producciones:

```
<program> ::= <function_list> <statement_list_opt>
           | <statement_list>

<function_list> ::= ε
                  | <function_list> <main_function>

<main_function> ::= "public" "static" "void" <id> "(" ")" "{" <function_body> "}"
                   | "public" "static" "void" <id> "(" "String[]" <id> ")" "{" <function_body> "}"

<statement> ::= <declaration>
               | <assignment>
               | <if_statement>
               | <while_statement>
               | <for_statement>
               ...
               | "continue" ";"
               | "System" "." "out" "." "println" "(" <expression> ")" ";"
               | "System" "." "out" "." "println" "(" ")" ";"
               | "{" <statement_list> "}"

<declaration> ::= <type> <id> ";"
                 | <type> <id> "=" <expression> ";"
                 | <type> <declaration_list> ";"

<type> ::= "int"
          | "float"
          | "double"
          | "boolean"
          | "char"
          ...
          | "double[]"
          | "String[]"

<assignment> ::= <id> "=" <expression> ";"
                | <array_access> "=" <expression> ";"
                | <id> "+=" <expression> ";"
                | <id> "-=" <expression> ";"
                | <id> "*=" <expression> ";"
                | <id> "/=" <expression> ";"

<expression> ::= <literal>
               | <array_initialization>
               | <id>
               | <id> "=" <expression>
               | <array_access> "=" <expression>
               | <id> "+=" <expression>
               ...
               | "(" <expression> ")"

<literal> ::= <int_literal>
            | <float_literal>
            | <double_literal>
            | <string_literal>
            | <char_literal>
            | "true"
            | "false"
            | "null"
```

- Arquitectura General
 - La arquitectura del sistema sigue el flujo clásico de un compilador:
 - Lexer (Flex): identifica tokens a partir del código fuente.
 - Parser (Bison): construye el árbol de sintaxis siguiendo la gramática.
 - AST (Árbol de Sintaxis Abstracta): representa la estructura lógica del programa.
 - Análisis Semántico: verifica tipos de datos, declaraciones y uso correcto de identificadores.
 - Intérprete: ejecuta las instrucciones del AST.
 - Interfaz Gráfica (GTK3): permite al usuario escribir, abrir y ejecutar código .usl, así como visualizar reportes.

ESPECIFICACIÓN DEL LENGUAJE

Identificadores

- Inician con letra o guion bajo, seguidos de letras, números o guiones bajos.

Comentarios

- Línea: // comentario
- Multilínea: /* comentario */

Tipos de Datos

- int, double, char, string, boolean.

Estructuras de Control

- Condicionales: if, if-else, switch-case.
- Ciclos: while, for, do-while.

Funciones

- Declaración con tipo de retorno o void.
- Parámetros tipados.

Arreglos

- Soporte para arreglos 1D y 2D.

IMPLEMENTACIÓN DE MÓDULOS

1 Lexer

El analizador léxico se implementó con Flex. Se definieron reglas para:

```
JavaLang > src > lexer.l
38  %% 
39
40 "int"      { printf("TOKEN INT\n"); return INT; }
41 "float"    { printf("TOKEN FLOAT\n"); return FLOAT; }
42 "double"   { printf("TOKEN DOUBLE\n"); return DOUBLE; }
43 "boolean"  { printf("TOKEN BOOLEAN\n"); return BOOLEAN; }
44 "char"     { printf("TOKEN CHAR\n"); return CHAR; }
45 "String"   { printf("TOKEN STRING_TYPE\n"); return STRING_TYPE; }
46 "class"    { printf("TOKEN CLASS\n"); return CLASS; }
47 "public"   { printf("TOKEN PUBLIC\n"); return PUBLIC; }
48 "private"  { printf("TOKEN PRIVATE\n"); return PRIVATE; }
49 "protected" { printf("TOKEN PROTECTED\n"); return PROTECTED; }
50 "static"   { printf("TOKEN STATIC\n"); return STATIC; }
51 "final"    { printf("TOKEN FINAL\n"); return FINAL; }
52 "void"     { printf("TOKEN VOID\n"); return VOID; }
53 "System"   { printf("TOKEN SYSTEM\n"); return SYSTEM; }
54 "out"      { printf("TOKEN OUT\n"); return OUT; }
55 "println"  { printf("TOKEN PRINTLN\n"); return PRINTLN; }
56 "equals"   { printf("TOKEN EQUALS_METHOD\n"); return EQUALS_METHOD; }
57 "if"       { printf("TOKEN IF\n"); return IF; }
58 "else"     { printf("TOKEN ELSE\n"); return ELSE; }
59 "while"    { printf("TOKEN WHILE\n"); return WHILE; }
60 "for"      { printf("TOKEN FOR\n"); return FOR; }
61 "do"       { printf("TOKEN DO\n"); return DO; }
62 "switch"   { printf("TOKEN SWITCH\n"); return SWITCH; }
63 "case"     { printf("TOKEN CASE\n"); return CASE; }
64 "default"  { printf("TOKEN DEFAULT\n"); return DEFAULT; }
65 "break"    { printf("TOKEN BREAK\n"); return BREAK; }
66 "continue" { printf("TOKEN CONTINUE\n"); return CONTINUE; }
67 "return"   { printf("TOKEN RETURN\n"); return RETURN; }
68 "new"      { printf("TOKEN NEW\n"); return NEW; }
69 "this"     { printf("TOKEN THIS\n"); return THIS; }
70 "true"    { printf("TOKEN TRUE\n"); return TRUE_TOKEN; }
71 "false"   { printf("TOKEN FALSE\n"); return FALSE_TOKEN; }
72 "null"    { printf("TOKEN NULL\n"); return NULL_TOKEN; }
73 "int[]"   { printf("TOKEN INT_ARRAY\n"); return INT_ARRAY; }
74 "float[]" { printf("TOKEN FLOAT_ARRAY\n"); return FLOAT_ARRAY; }
75 "String[]" { printf("TOKEN STRING_ARRAY\n"); return STRING_ARRAY; }
76
77 "++"      { printf("TOKEN INCREMENT\n"); return INCREMENT; }
78 "--"      { printf("TOKEN DECREMENT\n"); return DECREMENT; }
79 "+="      { printf("TOKEN PLUS_ASSIGN\n"); return PLUS_ASSIGN; }
80 "-="      { printf("TOKEN MINUS_ASSIGN\n"); return MINUS_ASSIGN; }
81 "*="      { printf("TOKEN MULT_ASSIGN\n"); return MULT_ASSIGN; }
82 "/="      { printf("TOKEN DIV_ASSIGN\n"); return DIV_ASSIGN; }
83 "=="      { printf("TOKEN EQUALS\n"); return EQUALS; }
84 "!="      { printf("TOKEN NOT_EQUALS\n"); return NOT_EQUALS; }
85 "<="      { printf("TOKEN LESS_EQUALS\n"); return LESS_EQUALS; }
86 ">="      { printf("TOKEN GREATER_EQUALS\n"); return GREATER_EQUALS; }
87 "&&"    { printf("TOKEN AND\n"); return AND; }
88 "||"      { printf("TOKEN OR\n"); return OR; }
89
90 "+"      { printf("TOKEN PLUS\n"); return PLUS; }
91 "-"      { printf("TOKEN MINUS\n"); return MINUS; }
92 "*"      { printf("TOKEN MULT\n"); return MULT; }
93 "/"      { printf("TOKEN DIV\n"); return DIV; }
94 "%"      { printf("TOKEN MOD\n"); return MOD; }
95 "="      { printf("TOKEN ASSIGN\n"); return ASSIGN; }
96 "<"      { printf("TOKEN LESS\n"); return LESS; }
97 ">"      { printf("TOKEN GREATER\n"); return GREATER; }
98 "!"      { printf("TOKEN NOT\n"); return NOT; }
99
100 "("     { printf("TOKEN LPAREN\n"); return LPAREN; }
101 ")"     { printf("TOKEN RPAREN\n"); return RPAREN; }
102 "{"     { printf("TOKEN LBRACE\n"); return LBRACE; }
103 "}"     { printf("TOKEN RBRACE\n"); return RBRACE; }
104 "["     { printf("TOKEN LBRACKET\n"); return LBRACKET; }
105 "]"     { printf("TOKEN RBRACKET\n"); return RBRACKET; }
106 ";"     { printf("TOKEN SEMICOLON\n"); return SEMICOLON; }
107 ","     { printf("TOKEN COMMA\n"); return COMMA; }
108 ":"     { printf("TOKEN DOT\n"); return DOT; }
109 ":"     { printf("TOKEN COLON\n"); return COLON; }
110
111 [0-9]+.[0-9]+[dD] {
112     yytext[yylen-1] = '\0';
113     yyval.float_val = atof(yytext);
114     printf("TOKEN DOUBLE_LITERAL: %f\n", yyval.float_val);
115     return DOUBLE_LITERAL;
116 }
117
118 [0-9]+.[0-9]+ {
119     printf("TOKEN FLOAT_LITERAL: %s\n", yytext);
120     yyval.float_val = atof(yytext);
121     return FLOAT_LITERAL;
122 }
123
124 [0-9]+ {
125     printf("TOKEN INT_LITERAL: %s\n", yytext);
```

2 Parser

El parser se implementó en Bison, utilizando la gramática definida.

Ejemplo de producción para condicionales:

```
JavaLang > src >  parser.y
664 literal:
700 ;
701
702 if_statement:
703   IF LPAREN expression RPAREN statement {
704     printf("If statement sin else\n");
705     $$ = new_ast_if($3, $5, NULL);
706     if ($$) $$->line = yylineno;
707   }
708   | IF LPAREN expression RPAREN statement ELSE statement {
709     printf("If statement con else\n");
710     $$ = new_ast_if($3, $5, $7);
711     if ($$) $$->line = yylineno;
712   }
713 ;
714
715 while_statement:
716   WHILE LPAREN expression RPAREN statement {
717     printf("While statement parser\n");
718     $$ = new_ast_while($3, $5);
719   }
720   | DO statement WHILE LPAREN expression RPAREN SEMICOLON {
721     printf("Do-While statement parser\n");
722     $$ = new_ast_do_while($5, $2);
723   }
724 ;
725
726 for_statement:
727   FOR LPAREN for_init SEMICOLON expression SEMICOLON expression RPAREN statement {
728     printf("For statement parser\n");
729     $$ = new_ast_for($3, $5, $7, $9);
730   }
731 ;
732
733 return_statement:
734   RETURN {
735     printf("Return statement parser\n");
736     $$ = new_ast_return(NULL);
737   }
738   | RETURN expression {
739     printf("Return statement parser\n");
740     $$ = new_ast_return($2);
741   }
742 ;
743
744 switch_statement:
745   SWITCH LPAREN expression RPAREN LBRACE switch_sections RBRACE {
746     printf("Switch statement parser\n");
747     /* Usamos un compound stmt temporal para contener las secciones */
748     $$ = new_ast_compound_stmt($6, NULL);
749     if ($$) $$->line = yylineno;
750   }
751 ;
752
753 switch_sections:
754   /* empty */ {
755     $$ = NULL;
756   }
757   | switch_sections switch_section {
758     if (!$1) {
```

3 AST

El árbol de sintaxis abstracta se representó con estructuras en C

```
avatang / include / ast.h
1 #ifndef AST_H
2
3 typedef enum {
4     AST_CONTINUE,
5     AST_MAIN_FUNCTION,
6     AST_PRINT_STMT=21,
7
8     AST_CONTINUE,
9
10    // Estructuras de agrupación
11    AST_COMPOUND_STMT,
12    AST_ARRAY_DECLARATION,
13    AST_ARRAY_ACCESS,
14    AST_ARRAY_INIT,
15    AST_ARRAY_ASSIGNMENT,
16    AST_STRING_EQUALS,
17    AST_METHOD_CALL,
18    AST_NEW_ARRAY,           /* new <type>[size] */
19    AST_NEW_ARRAY_INIT,
20    AST_NEW_OBJECT,
21
22    // Otros
23    AST_PROGRAM
24 } ast_node_type;
25
26 // Tipos de operadores (coinciden con los tokens)
27 typedef enum {
28     OP_PLUS, OP_MINUS, OP_MULT, OP_DIV, OP_MOD,
29     OP_INCREMENT, OP_DECREMENT, OP_POST_INCREMENT,
30     OP_ASSIGN, OP_PLUS_ASSIGN, OP_MINUS_ASSIGN, OP_MULT_ASSIGN, OP_DIV_ASSIGN,
31     OP_EQUALS, OP_NOT_EQUALS, OP_LESS, OP_GREATER, OP_LESS_EQUALS, OP_GREATER_EQUALS,
32     OP_AND, OP_OR, OP_NOT, OP_UNMINUS, OP_POST_DECREMENT
33 } operator_type;
34
35 //***** ESTRUCTURAS DE NODOS *****
36 // Estructura base de todos los nodos del AST
37 typedef struct ast_node {
38     ast_node_type type;
39     int line_number; // Para reporte de errores (opcional)
40     struct ast_node *next; // Para listas de statements
41
42     union {
43         // Literales
44         int int_value;
45         float float_value;
46         double double_value;
47         char *string_value;
48         char char_value;
49         int bool_value;
50
51         // Variable (solo nombre)
52         char *var_name;
53
54         // Tipo de dato
55         char *type_name;
56
57         // Operadores binarios y unarios
58         struct {
59             operator_type op;
60             struct ast_node *left;
61             struct ast_node *right;
62         } op;
63
64         // Declaración: tipo, nombre, valor (opcional)
65         struct {
66             struct ast_node *var_type;
67             char *var_name;
68             struct ast_node *value;
69         } declaration;
70
71         // Asignación: nombre, valor
72         struct {
73             char *var_name;
74             struct ast_node *value;
75         } assignment;
76
77         // If: condición, then, else
78         struct {
79             struct ast_node *condition;
80             struct ast_node *then_branch;
81             struct ast_node *else_branch;
82         } if_stmt;
83
84         // While: condición, cuerpo
85         struct {
86             struct ast_node *condition;
87         }
88     };
89
90 }
```

```
avatang / ast.c
1 #include "../include/ast.h"
2 #include "../include/parser.h" // Para los tokens
3 #include <stdlib.h>
4 #include <string.h>
5
6 //***** FUNCIONES DE CREACIÓN *****
7
8 // Literales y valores básicos
9 ast_node* new_ast_int_literal(int value) {
10     ast_node *node = malloc(sizeof(ast_node));
11     node->type = AST_INT_LITERAL;
12     node->data.int_value = value;
13     node->next = NULL;
14     return node;
15 }
16
17 ast_node* new_ast_float_literal(float value) {
18     ast_node *node = malloc(sizeof(ast_node));
19     node->type = AST_FLOAT_LITERAL;
20     node->data.float_value = value;
21     node->next = NULL;
22     return node;
23 }
24
25 ast_node* new_ast_double_literal(double value) {
26     ast_node *node = malloc(sizeof(ast_node));
27     node->type = AST_DOUBLE_LITERAL;
28     node->data.double_value = value;
29     node->next = NULL;
30     return node;
31 }
32
33 ast_node* new_ast_string_literal(char *value) {
34     ast_node *node = malloc(sizeof(ast_node));
35     node->type = AST_STRING_LITERAL;
36     node->data.string_value = strdup(value); // Copia la string
37     node->next = NULL;
38     return node;
39 }
40
41 ast_node* new_ast_char_literal(char value) {
42     ast_node *node = malloc(sizeof(ast_node));
43     node->type = AST_CHAR_LITERAL;
44     node->data.char_value = value;
45     node->next = NULL;
46     return node;
47 }
48
49 ast_node* new_ast_bool_literal(int value) {
50     ast_node *node = malloc(sizeof(ast_node));
51     node->type = AST_BOOL_LITERAL;
52     node->data.bool_value = value;
53     node->next = NULL;
54     return node;
55 }
56
57 ast_node* new_ast_null_literal(void) {
58     ast_node *node = malloc(sizeof(ast_node));
59     node->type = AST_NULL_LITERAL;
60     node->next = NULL;
61     return node;
62 }
63
64 ast_node* new_ast_variable(char *name) {
65     ast_node *node = malloc(sizeof(ast_node));
66     node->type = AST_VARIABLE;
67     node->data.var_name = strdup(name); // Copia el nombre
68     node->next = NULL;
69     return node;
70 }
71
72 ast_node* new_ast_type(char *type_name) {
73     ast_node *node = malloc(sizeof(ast_node));
74     node->type = AST_TYPE;
75     node->data.type_name = strdup(type_name); // Copia el nombre del tipo
76     node->next = NULL;
77     return node;
78 }
79
80 // Expresiones
81 ast_node* new_ast_binary_op(operator_type op, ast_node *left, ast_node *right) {
82     ast_node *node = malloc(sizeof(ast_node));
83     node->type = AST_BINARY_OP;
84     node->data.op.op = op;
85     node->data.op.left = left;
86     node->data.op.right = right;
87 }
```

Este diseño permite recorrer el árbol de forma recursiva y ejecutar las instrucciones.

4 Análisis Semántico.

Se implementaron validaciones para:

- Declaración previa de variables y funciones.
- Tipos de datos en expresiones aritméticas y lógicas.
- Dimensiones correctas en arreglos.



The image shows two code files side-by-side in a code editor. The left file is `semantic.h` and the right file is `semantic.c`. Both files are part of a C project named `JavaLang`.

semantic.h (Header File):

```
JavaLang > include > < semantic.h > ...
1 #ifndef SEMANTIC_H
2 #define SEMANTIC_H
3
4 #include "ast.h"
5
6 // ===== TIPOS DE DATOS =====
7
8 typedef enum {
9     TYPE_UNKNOWN,
10    TYPE_INT,
11    TYPE_FLOAT,
12    TYPE_DOUBLE,
13    TYPE_STRING,
14    TYPE_CHAR,
15    TYPE_BOOLEAN,
16    TYPE_VOID,
17    TYPE_FUNCTION,
18    TYPE_ARRAY_INT,
19    TYPE_ARRAY_FLOAT,
20    TYPE_ARRAY_STRING,
21    TYPE_ARRAY_CHAR,
22    TYPE_ARRAY_BOOLEAN
23 } data_type_t;
24
25 // ===== ESTRUCTURA DE SÍMBOLOS =====
26
27 // CORREGIR: Una sola definición de symbol
28 typedef struct symbol {
29     char *name;           // Nombre del símbolo
30     data_type_t type;    // Tipo de dato (CORREGIDO: era symbol_type)
31     char *scope_name;    // Nombre del scope donde se declaró
32     int line;            // Línea donde se declaró
33     int is_initialized;  // Si ha sido inicializada
34     int scope_level;    // Nivel del scope
35     struct symbol *next; // Lista enlazada
36 } symbol_t;
37
38 // ===== TABLA DE SÍMBOLOS =====
39
40 typedef struct symbol_table {
41     symbol_t *symbols;   // Lista de símbolos
42     int scope_level;    // Nivel del scope
43     struct symbol_table *parent; // Scope padre
44 } symbol_table_t;
45
46 // ===== VARIABLES GLOBALES =====
47
48 // AGREGAR: Declaración externa
49 extern symbol_table_t *global_symbol_table;
50 extern symbol_table_t *current_scope;
51 extern int semantic_errors;
52 extern int shadowing_policy;
53
54 // ===== FUNCIONES DE GESTIÓN DE SCOPES =====
55
56 symbol_table_t* create_scope(symbol_table_t *parent);
57 void destroy_scope(symbol_table_t *scope);
58 void enter_scope(void);
59 void exit_scope(void);
60 extern char current_scope_name[256];
61
62 // ===== FUNCIONES DE GESTIÓN DE SÍMBOLOS =====
63
64 symbol_t* declare_symbol(const char *name, data_type_t type);
65 symbol_t* lookup_symbol(const char *name);
66 symbol_t* lookup_symbol_current_scope(const char *name);
67
68 // ===== ANÁLISIS SEMÁNTICO =====
69
70 int check_semantics(ast_node *root);
71 data_type_t analyze_node(ast_node *node);
72 data_type_t analyze_expression(ast_node *expr);
73
74 // ===== TABLA DE SÍMBOLOS GLOBAL =====
75
76 symbol_table_t* create_symbol_table(void);
77 void add_symbol(symbol_table_t *table, const char *name, data_type_t type, const char *scope, int line);
78 symbol_t* find_symbol_in_scope(symbol_table_t *table, const char *name, const char *scope);
79
80 // CORREGIR: Una sola declaración de print_symbol_table
81 void print_symbol_table(void);
82 void semantic_analysis(ast_node *root);
83 void analyze_statement(ast_node *stmt);
84
85 // ===== FUNCIONES AUXILIARES =====
86
87 data_type_t string_to_type(const char *type_str);
88 const char* type_to_string(data_type_t type);
89
```

semantic.c (Source File):

```
JavaLang > src > < semantic.c > @ analyze_expression(ast_node *)
1 #include "../include/semantic.h"
2 #include "../include/errores.h"
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6
7 // Variables globales
8 symbol_table_t *current_scope = NULL;
9 symbol_table_t *global_symbol_table = NULL;
10 int semantic_errors = 0;
11 int shadowing_policy = 0; // Por defecto prohibir shadowing
12 char current_scope_name[256] = "global"; // Nombre del scope actual
13 static int semantic_loop_depth = 0; // Profundidad de loops anidados
14 void clear_global_symbol_table(void); // Prototipo para evitar implicit declaration
15
16 // ===== GESTIÓN DE SCOPES =====
17
18 symbol_table_t* create_scope(symbol_table_t *parent) {
19     symbol_table_t *scope = malloc(sizeof(symbol_table_t));
20     scope->symbols = NULL;
21     scope->scope_level = parent ? parent->scope_level + 1 : 0;
22     scope->parent = parent;
23     return scope;
24 }
25
26 void destroy_scope(symbol_table_t *scope) {
27     if (!scope) return;
28
29     symbol_t *current = scope->symbols;
30     while (current) {
31         symbol_t *next = current->next;
32         free(current->name);
33         free(current);
34         current = next;
35     }
36     free(scope);
37 }
38
39 void enter_scope(void) {
40     current_scope = create_scope(current_scope);
41     printf("Entrando a scope nivel %d\n", current_scope->scope_level);
42 }
43
44 void exit_scope(void) {
45     if (!current_scope) return;
46
47     printf("Saliendo de scope nivel %d\n", current_scope->scope_level);
48     symbol_table_t *old_scope = current_scope;
49     current_scope = current_scope->parent;
50     destroy_scope(old_scope);
51 }
52
53 // ===== GESTIÓN DE SÍMBOLOS =====
54 symbol_t* declare_symbol(const char *name, data_type_t type) {
55     if (!current_scope) {
56         enter_scope(); // Crear scope global si no existe
57     }
58
59     // Verificar shadowing
60     symbol_t *existing = lookup_symbol_current_scope(name);
61     if (existing) {
62         if (shadowing_policy == 0) {
63             report_semantic_error_local(0, 0, "Variable ya declarada en este scope", current_scope_name);
64             return NULL;
65         }
66     }
67
68     // Crear nuevo símbolo
69     symbol_t *symbol = malloc(sizeof(symbol_t));
70     symbol->name = strdup(name);
71     symbol->type = type;
72     symbol->is_initialized = 0;
73     symbol->scope_level = current_scope->scope_level;
74     symbol->scope_name = strdup("local"); // AGREGAR ESTA LÍNEA
75     symbol->line = 0; // AGREGAR ESTA LÍNEA
76     symbol->next = current_scope->symbols;
77     current_scope->symbols = symbol;
78
79     printf("Declarando variable '%s' de tipo %s en scope %d\n",
80           name, type_to_string(type), current_scope->scope_level);
81
82     // 🔥 AGREGAR: También registrar en tabla global para reporte
83     add_symbol_to_table(name, type, "global", 0);
84
85     return symbol;
86 }
87
88 symbol_t* lookup_symbol(const char *name) {
89
```

5 Interfaz Gráfica (GUI)

Se utilizó GTK3 para desarrollar un editor básico con:

- Apertura y guardado de archivos .usl.
- Ejecución del código con salida en consola integrada.
- Botones para generar reportes: AST, tabla de símbolos y tabla de errores.

```
JavaLang > include > C gui.h > analyze_code_for_semantic_errors(const char *)
1  #ifndef GUI_H
2  #define GUI_H
3
4  #include <gtk/gtk.h>
5  #include "ast.h"
6
7  // Estructura para manejar los widgets de la aplicación
8  typedef struct {
9      GtkWidget *window;
10     GtkWidget *notebook;
11     GtkWidget *source_view;
12     GtkTextBuffer *source_buffer;
13     GtkWidget *console_view;
14     GtkTextBuffer *console_buffer;
15     GtkWidget *ast_view;
16     GtkWidget *symbols_view;
17     GtkWidget *errors_view;
18     GtkWidget *execute_button;
19     GtkWidget *image_view;
20     GtkWidget *image_scroll;
21     GtkTextBuffer *errors_buffer;
22 } AppWidgets;
23
24 // Funciones principales de la GUI
25 int main_gui(int argc, char *argv[]);
26 void build_gui(void);
27
28 // Funciones de manejo de eventos
29 void execute_code(GtkButton *button, gpointer user_data);
30 void on_execute_clicked(GtkButton *button, gpointer user_data);
31
32 // AGREGAR: Funciones faltantes que usas en gui.c
33 void on_run_clicked(GtkButton *button, gpointer user_data);
34 void on_symbols_clicked(GtkButton *button, gpointer user_data);
35 void on_ast_clicked(GtkButton *button, gpointer user_data);
36
37 // Funciones para enviar información a la GUI
38 void send_token_to_gui(const char* token_type, const char* token_value);
39 void update_ast_treeview(ast_node *ast_root);
40 void append_to_console(const char *text);
41 void clear_console(void);
42
43 // Funciones para reportes
44 void show_symbol_table_report(GtkButton *button, gpointer user_data);
45 void show_ast_graphviz(GtkButton *button, gpointer user_data);
46 int generate_ast_dot(ast_node *root, const char *filename);
47 void generate_ast_dot_node(FILE *file, ast_node *node, int *node_id, int parent_id);
48 const char* get_op_string(int op);
49 void show_error_report_gui(GtkButton *button, gpointer user_data);
50 void analyze_lexical_errors(const char *code);
51 int analyze_code_for_semantic_errors(const char *code);
52 void show_image_in_app(const char *image_path);
53 void update_errors_view(void);
54
55
56 // Variables globales de buffers
57 extern GtkTextBuffer *console_buffer;
58 extern GtkTextBuffer *source_buffer;
59
60 #endif
```

```
JavaLang > src > C gui.c > ...
222
223 // ===== FUNCION DE EJECUCION =====
224 // Función principal para ejecutar código
225
226 // helper que re-habilita UI desde el hilo principal
227 static gboolean finish_ui_update(gpointer user_data) {
228     if (widgets && widgets->execute_button) {
229         gtk_widget_set_sensitive(GTK_WIDGET(widgets->execute_button), TRUE);
230     }
231     execution_in_progress = 0;
232     return G_SOURCE_REMOVE;
233 }
234
235 static gpointer execute_worker(gpointer data) {
236     // 'data' es gchar* sanitizado (heap), se liberará aquí
237     gchar *sanitized = (gchar*)data;
238
239     append_to_console_threadsafe("✖ Iniciando ejecución...\n");
240     init_error_system();
241
242     // FASE 1: léxico
243     analyze_lexical_errors(sanitized);
244
245     // FASE 2: parse
246     setup_lexer_for_string(sanitized);
247     yrestart(NULL);
248     int parse_result = yyparse();
249
250     if (parse_result != 0) {
251         append_to_console_threadsafe("✖ Errores sintácticos encontrados\n");
252         int sem_count = analyze_code_for_semantic_errors(sanitized);
253         if (sem_count > 0) append_to_console_threadsafe("✖ Errores semánticos detectados\n");
254         else append_to_console_threadsafe("✓ Análisis semántico: sin errores aparentes\n");
255     } else {
256         append_to_console_threadsafe("✓ Análisis sintáctico: EXITOSO\n");
257         ast_node *ast_root = get_ast_root();
258         if (ast_root) {
259             semantic_analysis(ast_root);
260             append_to_console_threadsafe("✖ Ejecutando programa...\n");
261             execute_program(ast_root);
262         } else {
263             append_to_console_threadsafe("⚠ No hay AST disponible para ejecución\n");
264         }
265     }
266
267     append_to_console_threadsafe("✓ Análisis completado\n");
268
269     // liberar y reactivar UI en hilo principal
270     g_idle_add(finish_ui_update, NULL);
271     g_free(sanitized);
272     return NULL;
273 }
274
275 void execute_code(GtkButton *button, gpointer user_data) {
276     if (execution_in_progress) {
277         append_to_console("⚠ Ejecución ya en progreso\n");
278         return;
279     }
280     execution_in_progress = 1;
281
282     if (widgets && widgets->execute_button) {
283         gtk_widget_set_sensitive(GTK_WIDGET(widgets->execute_button), FALSE);
284     }
285
286     // Obtener el código del área de texto (hacer esto en hilo principal)
287     GtkTextBuffer *buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(widgets->source_view));
288     GtkTextIter start, end;
289     gtk_text_buffer_get_start_iter(buffer, &start);
290     gtk_text_buffer_get_end_iter(buffer, &end);
291     gchar *code = gtk_text_buffer_get_text(buffer, &start, &end, FALSE);
292     gchar *sanitized = g_utf8_make_valid(code, -1);
293     g_free(code);
294
295     // Lanzar hilo que hará el trabajo pesado. el worker liberará 'sanitized'
296     GThread *t = g_thread_new("javalang-worker", execute_worker, sanitized);
297     g_thread_unref(t);
298 }
299
300 // ===== FUNCIONES DE REPORTE =====
301 char* generate_symbols_html(void) {
302     const size_t CAP = 65536;
303     char *html_content = malloc(CAP);
304     if (!html_content) return NULL;
305     size_t off = 0;
306     html_content[0] = '\0';
307
308     off += snprintf(html_content + off, CAP - off,
309                     "<!DOCTYPE html>\n"
310                     "<html>\n"
311                     "<head>\n"
312                     "</head>\n"
313                     "<body>\n"
314                     "</body>\n"
315                     "</html>\n");
316 }
```

6 Interprete

En este se realizaron las ejecuciones para que corriera las entradas:

```
JavaLang > src > C runtime.c > ...
11 #ifndef TYPE_INT
12 #define TYPE_INT 0
13 #define TYPE_FLOAT 1
14 #define TYPE_STRING 2
15 #define TYPE_BOOLEAN 3
16 #define TYPE_DOUBLE 4
17 #endif
18
19 static ConsolePrintCallback console_callback = NULL;
20
21 // Nombre que espera la GUI; guarda el callback
22 void set_console_print_safe(ConsolePrintCallback callback) {
23     console_callback = callback;
24 }
25
26 // Helper que invoca el callback o hace fallback a stdout
27 static void console_invoke(const char *msg) {
28     if (!msg) return;
29     if (console_callback) console_callback(msg);
30     else printf("%s", msg);
31 }
32
33 // ----- Separación OUTPUT vs DEBUG -----
34 static int runtime_debug_enabled = 0; // por defecto apagado
35 static int runtime_break_flag = 0;
36 static int runtime_continue_flag = 0;
37
38 void set_runtime_debug(int enabled) {
39     runtime_debug_enabled = enabled;
40 }
41
42 // Salida del programa (System.out.println) -> se muestra siempre en la consola GUI
43 void console_print_program(const char *msg) {
44     if (!msg) return;
45     if (console_callback) console_callback(msg);
46     else printf("%s", msg);
47 }
48
49 // Mensajes internos / depuración -> solo si runtime_debug_enabled == 1
50 void runtime_log(const char *msg) {
51     if (!msg) return;
52     if (!runtime_debug_enabled) return;
53     if (console_callback) console_callback(msg);
54     else printf("%s", msg);
55 }
56
57 // Tabla de simbolos para variables
58 typedef struct runtime_symbol {
59     char *name;
60     int type; // 0=int, 1=float, 2=string, 3=boolean, 4=double
61     union {
62         int int_val;
63         float float_val;
64         double double_val;
65         char *string_val;
66         int bool_val;
67         void *array_val;
68     } value;
69     struct runtime_symbol *next;
70 } runtime_symbol_t;
71
72 static runtime_symbol_t *symbol_table = NULL;
73
74 // Tabla de arrays en tiempo de ejecución
75 typedef struct {
76     int elem_type; // usar códigos compatibles con sym.type o semantic types
77     int length;
78     int *ints; // por ahora solo soporte int[]
79 } runtime_array_t;
80
81 #define TYPE_ARRAY_INT 100
82
83 typedef struct array_map {
84     ast_node *expr;
85     runtime_array_t *arr;
86     struct array_map *next;
87 } array_map_t;
88 static array_map_t *array_map_head = NULL;
89
90 // Declaraciones forward
91 void show_symbol_table(void);
92 void cleanup_runtime(void);
93 int is_string_expression(ast_node *node);
94 char* get_string_value(ast_node *node);
95 void convert_symbol_to_string(runtime_symbol_t *sym, char *str);
96 void convert_to_string(ast_node *node, char *str);
97 void build_concatenated_string(ast_node *expr, char *result);
98 //auxiliares
99 int builtin_invoke_int(const char *fullname, ast_node *arg, int *out_int);
100
101 OpenWebsite GenerateCommitMessage
```

```
JavaLang > src > C runtime.c > ...
51 // Función para evaluar expresiones
52 int evaluate_expression(ast_node *expr) {
53     if (!expr) return 0;
54
55     switch (expr->type) {
56         case AST_INT_LITERAL:
57             return expr->data.int_value;
58
59         case AST_FLOAT_LITERAL:
60             return (int)expr->data.float_value; // Convertir a int por simplicidad
61
62         case AST_DOUBLE_LITERAL:
63             return (int)expr->data.double_value; // Convertir a int por simplicidad
64
65         case AST_BOOL_LITERAL:
66             return expr->data.bool_value;
67
68         case AST_VARIABLE:
69             runtime_symbol_t *sym = find_or_create_symbol(expr->data.var_name);
70
71             // RETORNAR SEGUN EL TIPO DE LA VARIABLE
72             switch (sym->type) {
73                 case 0: return sym->value.int_val; // int
74                 case 1: return (int)sym->value.float_val; // float -> int
75                 case 2: return strlen(sym->value.string_val) ? sym->value.string_val : ""; // string -> longitud
76                 case 3: return sym->value.bool_val; // boolean
77                 case 4: return (int)sym->value.double_val; // double -> int
78                 default: return 0;
79             }
80
81         case AST_ARRAY_ACCESS:
82             // expr->data.array_access.array_name and index expression
83             const char *name = expr->data.array_access.array_name;
84             ast_node *index_expr = expr->data.array_access.index;
85             if (!name || !index_expr) return 0;
86             runtime_symbol_t *sym = find_or_create_symbol(name);
87             if (!sym || sym->type != TYPE_ARRAY_INT) return 0;
88             runtime_array_t *arr = (runtime_array_t*)sym->value.array_val;
89             if (!arr) return 0;
90             int idx = evaluate_expression(index_expr);
91             if (idx < 0 || idx >= arr->length) {
92                 char msg[256];
93                 snprintf(msg, sizeof(msg), "⚠ Array index out of bounds: %s[%d] (len=%d)\n", name, idx, arr->length);
94                 runtime_log(msg);
95                 return 0;
96             }
97             return arr->ints[idx];
98
99         case AST_METHOD_CALL:
100             char recv_name[256] = "";
101             ast_node *recv = expr->data.method_call.receiver;
102             if (recv) {
103                 if (recv->type == AST_VARIABLE && recv->data.var_name) {
104                     strcpy(recv_name, recv->data.var_name, sizeof(recv_name)-1);
105                     recv_name[sizeof(recv_name)-1] = '\0';
106                 } else {
107                     char tmp[256];
108                     strncpy(tmp, get_string_value(recv), sizeof(tmp)-1);
109                     tmp[sizeof(tmp)-1] = '\0';
110                     strcpy(recv_name, tmp, sizeof(recv_name)-1);
111                     recv_name[sizeof(recv_name)-1] = '\0';
112                 }
113             }
114             const char *mname = expr->data.method_call.method_name ? expr->data.method_call.method_name : "";
115
116             // StringBuilder.append(receiver.append(arg))
117             if (recv_name[0] != '\0' && strcmp(mname, "append") == 0) {
118                 runtime_symbol_t *recv_sym = find_or_create_symbol(recv_name);
119                 char *argstr = runtime_eval_as_string(expr->data.method_call.arg);
120                 if (recv_sym->type != TYPE_STRING) {
121                     recv_sym->type = TYPE_STRING;
122                     recv_sym->value.string_val = strdup(argstr ? argstr : "");
123                 } else {
124                     size_t cur = strlen(recv_sym->value.string_val) ? recv_sym->value.string_val : "";
125                     size_t add = strlen(argstr ? argstr : "");
126                     char *n = malloc(cur + add + 1);
127                     if (!n) return 0;
128                     n[0] = '\0';
129                     if (recv_sym->value.string_val) strcpy(n, recv_sym->value.string_val);
130                     strcat(n, argstr ? argstr : "");
131                     free(recv_sym->value.string_val);
132                     recv_sym->value.string_val = n;
133                 }
134             }
135             // devuelve la longitud como valor entero (evaluate_expression requiere int)
136             return (int)strlen(recv_sym->value.string_val) ? recv_sym->value.string_val : "";
137
138         default:
139             break;
140     }
141
142     return 0;
143 }
144
145 void set_console_callback(ConsolePrintCallback callback);
146 int builtin_invoke_int(const char *fullname, ast_node *arg, int *out_int);
147 int builtin_invoke_double(const char *fullname, ast_node *arg, double *out_double);
148 char* builtin_invoke_string(const char *fullname, ast_node *arg);
149 int builtin_dispatch(const char *fullname, struct ast_node *arg, int *out_int, double *out_double, char **out_str);
150 char *runtime_eval_as_string(struct ast_node *arg);
151 void set_console_print_safe(ConsolePrintCallback callback);
152 void console_print_program(const char *msg);
153 void runtime_log(const char *msg);
154
155 char* get_string_value(struct ast_node *node);
156 void build_concatenated_string(struct ast_node *expr, char *result);
157 void cleanup_runtime(void);
158
159 void set_runtime_debug(int enabled);
160
161 #endif
```

```
JavaLang > include > C runtime.h > builtin_dispatch(const char *, ast_node *, int *, double *, char *)
1 #ifndef RUNTIME_H
2 #define RUNTIME_H
3
4 #include "ast.h"
5
6 typedef void (*ConsolePrintCallback)(const char* message);
7 // Funciones del intérprete
8 void execute_program(ast_node *root);
9 ast_node* find_main_method(ast_node *program_node);
10 int evaluate_expression(ast_node *expr);
11 void execute_statement(ast_node *stmt);
12 void show_symbol_table(void);
13 void cleanup_runtime(void);
14
15 void set_console_callback(ConsolePrintCallback callback);
16 int builtin_invoke_int(const char *fullname, ast_node *arg, int *out_int);
17 int builtin_invoke_double(const char *fullname, ast_node *arg, double *out_double);
18 char* builtin_invoke_string(const char *fullname, ast_node *arg);
19 int builtin_dispatch(const char *fullname, struct ast_node *arg, int *out_int, double *out_double, char **out_str);
20 char *runtime_eval_as_string(struct ast_node *arg);
21 void set_console_print_safe(ConsolePrintCallback callback);
22 void console_print_program(const char *msg);
23 void runtime_log(const char *msg);
24
25 char* get_string_value(struct ast_node *node);
26 void build_concatenated_string(struct ast_node *expr, char *result);
27 void cleanup_runtime(void);
28
29 void set_runtime_debug(int enabled);
30
31 #endif
```

RETOS TÉCNICOS Y SOLUCIONES

- Recursión por la izquierda en la gramática: se reestructuraron producciones para evitar conflictos en Bison.
- Ambigüedad en expresiones lógicas: se estableció precedencia y asociatividad en las reglas del parser.
- Manejo de errores sintácticos: se definieron producciones de error en Bison para permitir la recuperación y continuar el análisis.
- Integración con GTK3: se ajustó el uso de hilos para no bloquear la interfaz durante la ejecución del código.
- Manejo de Ámbitos y Tabla de Símbolos: Diseñar bien las estructuras de datos ahorra tiempo.
- Difícil visualizar la estructura del AST durante el desarrollo: Implementar correctamente la creación de las hojas y padres, además de tener contenido correcto en la clase header.
- Errores sin conocimiento de fallo: Usar macros de debug, para visualizar los errores más comunes y colocarlo para ser visible o no.

CONCLUSIONES

- Se logró implementar un intérprete académico funcional para el lenguaje JavaLang.
- La integración de Flex y Bison permitió reforzar conocimientos en análisis de lenguajes formales.
- Los retos técnicos principales fueron la ambigüedad de la gramática, la integración con GTK3, problemas con el interprete y con la semantica.
- El sistema es extensible, permitiendo agregar nuevas características como clases u operadores avanzados en futuras versiones.