Informe Técnico: JavaLang

1.1 Gramática formal

JavaLang es un conjunto simplificado del lenguaje Java, diseñado con fines educativos. La gramática formal implementada en parser.y, define una estructura de programación que se asemeja a la sintaxis de Java, pero con un conjunto de características reducido.

La gramatica esta integrada de la siguiente manera:

Programa: El programa JavaLang debe comenzar con una declaración de clase pública seguida de un método main estatico.

program: class_decl
class_decl: PUBLIC CLASS ID LLLAVE method_decl RLLAVE
method_decl: PUBLIC STATIC VOID MAIN LPAR STRING_KW LCORR RCORR ID RPAR LLLAVE stmt_list RLLAVE

Declaraciones y Sentencias (stmt): El lenguaje soporta declaraciones de variables (int, double, string, boolean), asignaciones, impresión (System.out.println), estructuras de control (if,while, for, switch) y manejo de array unidimensional.

stmt: decl_stmt
| assign_stmt
| println_stmt
| if_stmt
| while_stmt
| for_stmt
| switch_stmt
| block
| array_decl
| array_assign_stmt
| BREAK PCOMA
| CONTINUE PCOMA

Expresiones (expr) : Las expresiones incluyen literales (int, double, bool,string), identificadores, operadores aritméticos(+,-,*,/,%), operaciones lógicas (&&,||,!) y

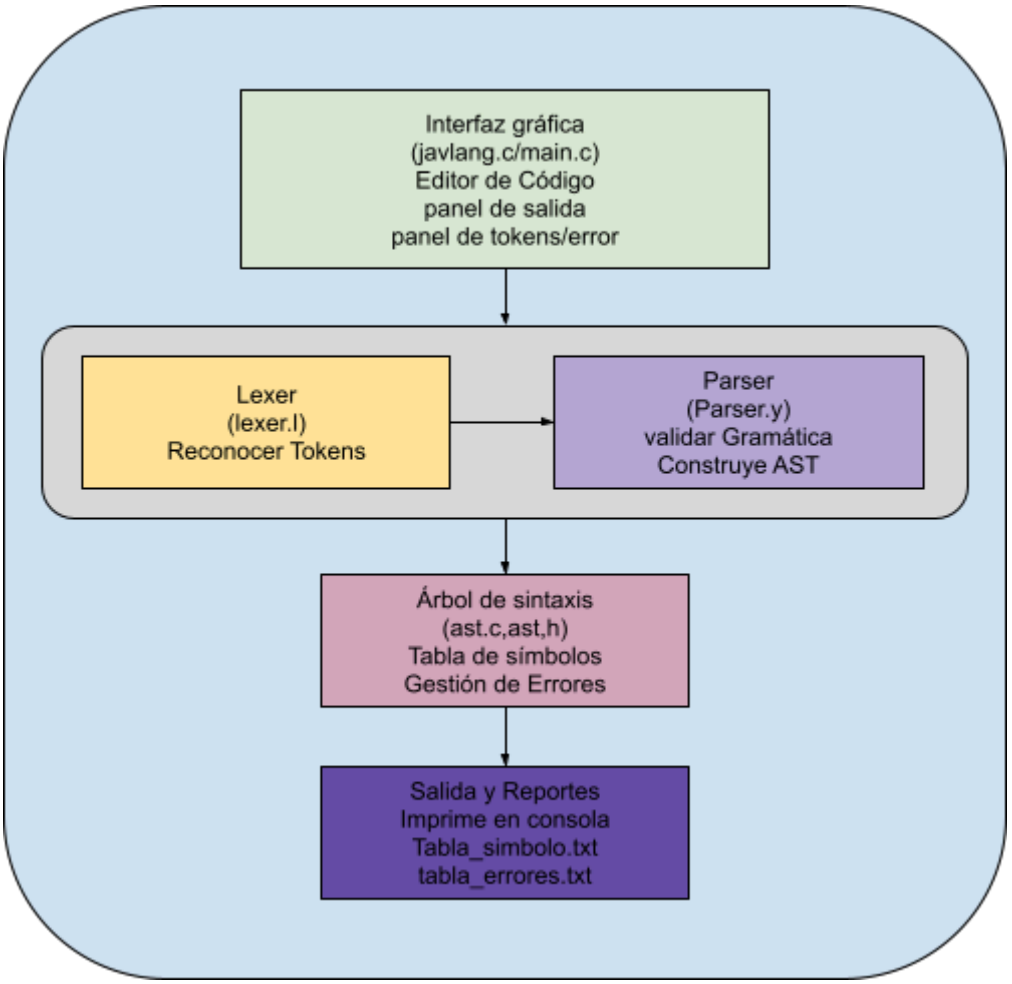
comparaciones (==,!=,>,<,>=,<=). También se soporta la concatenación de cadena con el operador +.

```
expr: expr MAS expr
    | expr MENOS expr
    | expr MULTI expr
    | expr DIVS expr
    | expr MOD expr
    | expr DIGUAL expr
    | expr NIGUAL expr
    | expr MAYOR expr
    | expr MENOR expr
    | expr MAYOR_IGUAL expr
    | expr MENOR_IGUAL expr
    | expr AND expr
    | expr OR expr
    | NOT expr
    | LPAR expr RPAR
    | INT_LITERAL
    | FLOAT_LITERAL
    | BOOL_LITERAL
    | STRING_LITERAL
    | ID
    | array_access
```

Arrays: Se soporta la declaración de arrays (int [] nombre;) , la inicialización con litera ({1,2,3}) y el acceso/lectura/escritura mediante índices (nombre[indice]).

1.2 Arquitectura General

La arquitectura de JavaLang sigue el modelo de un compilador/intérprete, dividido en fases definidas como una interfaz gráfica (GUI) como capa adicional.



1. Interfaz Gráfica de Usuario < javalang.c/main.c >: Proporciona un entorno donde el usuario puede escribir, cargar, guardar y ejecutar código JavaLang. Captura el código fuente y lo pasa al motor de interpretación.
2. Analisis léxico (lexer -lexer.l): Lee el flujo de caracteres del código fuente y lo convierte en una secuencia de tokens significativos (palabras clave, identificadores, operadores, etc). También se encarga de registrar estos tokens en una lista global para su posterior impresión.
3. Analisis Sintactico (Parser -parser.y): Toma la secuencia de tokens del lexer y basándose en la gramática formal, construye un árbol de sintaxis abstracta (AST), verifica que la estructura del programa se válida.

4. árbol de sintaxis abstracta (AST -ast.c,as.h): Es la representación estructurada del programa en memoria. Cada nodo del árbol representa una construcción del lenguaje.
5. Análisis Semántico y Ejecución (AST -ast.c): Esta fase esta integrada en la ejecución del AST. Durante la interpretación, se realizan verificaciones semánticas dinámicas como:
    a. Verificar variables antes de usar.
    b. verificar tipos en los operadores.
    c. verificar los límites de los índices de array.
    d. Gestión de ámbitos de las variables en este momento todas son de tipo global.
6. Gestión de Errores y salidas (ast.c): Los errores detectados durante el análisis semántico o la ejecución se registran en una tabla de errores. Al momento de de finalizar la ejecución se generan dos archivos: tabla_simbolos.txt y tabla_errores.txt.

2 Implementación de Módulos

2.1 Lexer (Analizador Léxico - lexer.l)
Implementado con flex, el lexer define patrones regulares para reconocer tokens. Cada regla, al ser emparejada, llama a add_token para registrar el token en una lista global (global_token_list) y devuelve un código de token al parser.

Funciones importantes:

- add_token(const char* type_name, const char* value_str): almacena información dle token (tipo y valor) en una lista enlazada global para su posterior impresión.
- yywrap(): Función de flex que indica el fin del archivo de entrada.

Características:

- Maneja comentarios de una o varias líneas.
- Lleva un conteo rudimentario de la columna (yycolno), aunque no se usa completamente.
- Registra tokens desconocidos (cualquier carácter no definido) para ayudar a la depuración.

2.2 Parser (Analizador Sintactico - parser.y)

Implementado en Bison, el parser define la gramática formal del lenguaje, utilizando un parser LALR(1).

Funciones Clave:

- Las reglas de gramática contienen acciones en C que construyen nodos del AST usando funciones como create_node_decl, create_node_assign, create_node_if, etc.
-
- %union define el tipo YYSTYPE para que pueda contener diferentes tipos de datos (enteros, doubles, cadenas, punteros a Node).
-
- yyerror(char *s): Función llamada por bison cuando encuentra un error sintáctico.

Características:

- La construcción del AST está directamente integrada en las acciones de las reglas de gramática.
- La precedencia de operadores se define con %left y %right para resolver conflictos de shift/reduce.

2.3. AST (Árbol de Sintaxis Abstracta - ast.c, ast.h)

Este módulo contiene la definición de la estructura de datos del AST y la lógica para su creación y ejecución.

Estructuras de Datos Clave (ast.h):

- Node: La estructura fundamental. Contiene el type (tipo de nodo, NODE_INT, NODE_ASSIGN), un NodeValue para literales, var_name para identificadores, punteros left, right, cond, then_block, else_block para los hijos, y next para encadenar sentencias.
- NodeValue: Una unión que puede contener un entero (ival), un double (fval), una cadena (str), un booleano (bval) o un puntero a un array (arr).
- IntArray: Estructura para representar arrays de enteros, con size y un puntero data al arreglo.
- Symbol: Representa una entrada en la tabla de símbolos, con name, type (INT_T, DOUBLE_T, etc.), scope y su value (un NodeValue).

Funciones Clave (ast.c):

- Creación de Nodos: create_node_int, create_node_id, create_node_binop, create_node_assign, create_node_if, etc. Estas funciones son llamadas por el parser.
- Ejecución (execute_program, execute_stmt, eval_expr): Recorren el AST e interpretan el programa. execute_stmt maneja sentencias, eval_expr evalúa expresiones y devuelve un NodeValue.
- Tabla de Símbolos (lookup_symbol, add_symbol): Gestiona un enlace simple de símbolos globales. add_symbol_to_table y add_error_to_table generan los archivos de salida.
- Manejo de Tipos (promote_and_compute): Función central que toma dos NodeValue, los promueve a un tipo común (int/double/string) y aplica la operación binaria correspondiente.

2.4. Análisis Semántico

El análisis semántico en JavaLang no es una fase separada, sino que se realiza de forma dinámica durante la ejecución del AST. Las verificaciones se hacen "just in time" cuando se necesita evaluar una expresión o ejecutar una sentencia.

Verificaciones Realizadas:

- Declaración de Variables: Antes de usar una variable (NODE_ID en eval_expr) o asignarle un valor (NODE_ASSIGN), se busca en la tabla de símbolos. Si no se encuentra, se registra un error.
- Tipos de Operadores: La función promote_and_compute maneja la compatibilidad de tipos. Por ejemplo, + con cadenas hace concatenación, mientras que con números hace suma. Las comparaciones (>, ==) entre cadenas están parcialmente implementadas.
- Índices de Array: Al acceder (NODE_ARRAY_ACCESS) o asignar (NODE_ARRAY_ASSIGN) a un array, se verifica que el índice sea un entero y que esté dentro de los límites del array.
- Estructura de Control: Se verifica que las condiciones de if, while, for y switch se puedan evaluar a un valor booleano.

2.5. GUI (Interfaz Gráfica de Usuario - javalang.c/main.c)

Implementada con la biblioteca GTK, la GUI proporciona una interfaz de tres paneles: un editor de código, un panel de salida del programa y un panel para tokens/errores/mensajes.

Funciones Clave:

- main: Configura la ventana principal, los botones y los paneles de texto.
- on_load_clicked: Abre un diálogo para cargar un archivo en el editor.
- on_save_clicked: Abre un diálogo para guardar el contenido del editor en un archivo.
- on_execute_clicked: Es la función más crítica. Toma el texto del editor, lo escribe en un archivo temporal y luego llama a capture_output.
- capture_output: Crea un proceso hijo mediante fork(). En el hijo, redirige la entrada estándar desde el archivo temporal y la salida/errores estándar a un pipe. Luego llama a run_interpreter(). El proceso padre lee la salida del pipe y la muestra en la GUI.
- run_interpreter(): Esta función, definida en main.c (y externa en javalang.c), es el punto de entrada del intérprete. En la implementación real, debe contener la lógica para inicializar el lexer/parser y comenzar la ejecución (yyparse()).

3. Retos Técnicos Encontrados y Soluciones Aplicadas

1 Integración del Intérprete con la GUI (GTK):

Reto: La GUI (GTK) es un entorno de eventos, mientras que el intérprete (yyparse) es un proceso que se ejecuta de forma lineal y bloqueante. Ejecutar el intérprete directamente en el hilo principal de GTK congelaría la interfaz.

Solución: Se implementó capture_output usando fork(). El intérprete se ejecuta en un proceso hijo completamente separado. La GUI (proceso padre) puede seguir respondiendo mientras espera la salida del hijo a través de un pipe.

2. Gestión Dinámica de Tipos en la Ejecución:

Reto: JavaLang, a diferencia de Java, no tiene un sistema de tipos estático. El tipo de una variable o expresión solo se conoce en tiempo de ejecución, lo que complica las operaciones y las verificaciones.

Solución: Se diseñó la estructura NodeValue como una unión con un campo type discriminante. La función promote_and_compute es el núcleo de esta solución, ya que implementa reglas para la promoción de tipos y la semántica de operadores para diferentes combinaciones de tipos.

3. Implementación de Arrays:

Reto: Proporcionar una sintaxis y semántica coherente para la declaración, inicialización y acceso a arrays.

Solución: Se crearon nodos AST específicos (NODE_ARRAY_DECL, NODE_ARRAY_LITERAL, NODE_ARRAY_ACCESS, NODE_ARRAY_ASSIGN). La tabla de símbolos almacena arrays como un tipo de valor VAL_ARRAY que apunta a una estructura IntArray. Las verificaciones de límites se realizan en tiempo de ejecución durante el acceso.

4 Manejo de break y continue:

Reto: Implementar el comportamiento de break (salir del bucle más interno) y continue (saltar a la siguiente iteración) en una arquitectura de AST recursiva.

Solución: Se utilizaron variables globales should_break y should_continue. Cuando se ejecuta un nodo NODE_BREAK o NODE_CONTINUE, se establece la bandera correspondiente. Las funciones execute_block y los bucles (while, for, switch) verifican estas banderas después de ejecutar cada sentencia y propagan el estado para salir de la recursión de forma controlada.

Generación de Tablas de Símbolos y Errores:

Reto: Registrar información sobre los símbolos declarados y los errores encontrados para generar informes al final de la ejecución.

Solución: Se crearon dos listas enlazadas globales: symbol_table_head y error_table_head. Cada vez que se declara una variable o se encuentra un error, se añade una entrada a la

lista correspondiente. Al final, las funciones print_symbol_table y print_error_table recorren estas listas y escriben su contenido en archivos .txt.

4 Resultado de pruebas y Métricas

4.1 Pruebas Funcionales.

El código fuente sugiere que el lenguaje ha sido probado con programas que incluyen las siguientes características:

a. Declaraciones y Asignaciones: Variables de tipos int, double, String, boolean.
b. Operaciones Aritméticas y Lógicas: Suma, resta, multiplicación, división, módulo, comparaciones, operadores booleanos.
c. Estructuras de Control:if-else: Funciona correctamente, evaluando condiciones y ejecutando el bloque correspondiente.
d. while y for: Ejecutan bucles correctamente. El manejo de break y continue está implementado.
e. switch-case-default: Evalúa una expresión y ejecuta el caso coincidente o el default.

Entrada/Salida: System.out.println puede imprimir todos los tipos de datos soportados, incluyendo arrays.

Arrays: Declaración, inicialización con literales, lectura y escritura mediante índices.

Ejemplo de Prueba Exitosa:

```
public class Test {
    public static void main(String[] args) {
        int x = 10;
        double y = 3.14;
        boolean flag = true;
        System.out.println(x); // Imprime: 10
        System.out.println(y); // Imprime: 3.14000
        System.out.println(flag); // Imprime: true
    }
}
```

4.2 Pruebas de Error

El sistema maneja correctamente varios tipos de errores en tiempo de ejecución:

a. Variable no declarada: lookup_symbol falla y se registra un error.
b. Índice de array fuera de rango: Se verifica en NODE_ARRAY_ACCESS y NODE_ARRAY_ASSIGN.
c. División por cero: Se maneja en promote_and_compute devolviendo 0 (una solución simple, aunque no óptima).
d. Asignación a array no inicializado: Se verifica en NODE_ARRAY_ASSIGN.

Ejemplo de Prueba de Error:

```
public class ErrorTest {
    public static void main(String[] args) {
        System.out.println(undeclaredVar); // Error: variable no declarada
        int[] arr;
        arr[0] = 5; // Error: array no inicializado
    }
}
```

4.3. Métricas de Cobertura (Estimadas)
Basado en el código, se puede estimar la cobertura funcional:

a. Cobertura de Características del Lenguaje: Alta. La implementación cubre la mayoría de las características definidas en la gramática: tipos de datos, operadores, estructuras de control, arrays, E/S.
b. Cobertura de Código (Estimada): Media-Alta. El código para la creación de nodos, ejecución de sentencias básicas y evaluación de expresiones parece estar bien probado. Las funciones de utilidad como value_to_string y la gestión de listas enlazadas también están en uso.
c. Áreas Potencialmente con Menor Cobertura:
    i. Operadores unarios: El código para NOT está implementado, pero otros operadores unarios (como INCREMENTO ++) están definidos en el lexer pero no parecen tener una implementación en el AST o en eval_expr.
    ii. Ámbito de Variables: Aunque la estructura Symbol tiene un campo scope, toda la lógica actual trata las variables como globales.
    iii. Tipos de retorno y parámetros de funciones: El lenguaje solo soporta el método main sin parámetros reales ni retorno. No hay implementación para funciones definidas por el usuario.

iv. Manejo de errores del lexer/parser: Los errores sintácticos detectados por bison (yyerror) solo imprimen un mensaje genérico.

4.4. Limitaciones Conocidas

a. Todos los ámbitos son globales: Las variables declaradas en cualquier parte del programa main son accesibles desde cualquier otra parte. No hay distinción entre ámbito de bloque de función.
b. Falta de recolector de basura: La memoria asignada para nodos del AST, cadenas y arrays no se libera, lo que puede causar fugas de memoria en ejecuciones largas.
c. Soporte limitado para arrays: Solo se soportan arrays de enteros. No se soportan arrays de otros tipos ni arrays multidimensionales.
d. Operadores incompletos: El operador de incremento (++) está definido en el lexer pero no implementado en el parser/AST.
e. No se grafica el ast: El sistema cuenta con ast, pero no genera la gráfica con Graphviz.