Informe Técnico – JavaLang Interpreter

Bienvenido a la versión técnica del proyecto. Aquí te contamos, de forma directa y práctica, cómo está armado el intérprete/compilador, qué se agregó en esta fase (ARM64 y arreglos irregulares) y cómo validamos que todo funcione. La idea es que lo puedas leer como una guía técnica, no como un paper: con ejemplos, notas y decisiones claras.

Índice

Resumen rápido

- Backend ahora genera código para **AArch64 (ARM64)** listo para ensamblar y ejecutar (vía QEMU o máquina ARM nativa).
- Soporte para **arreglos irregulares (jagged)**: arreglos cuyas filas pueden tener longitudes distintas (p. ej., `int[][]`).
- Se reforzó el manejo de **strings** (uso de `strdup` en puntos clave) y se separaron buffers para evitar choques.
- Se corrigió un caso difícil de **segfault** relacionado con inicializadores de arreglos y preservación de

registros en ARM64.

- Se agregaron pequeñas **utilidades de diagnóstico** (formatos y prints opt-in) para depurar sin molestar al usuario final.

> Idea principal: emitir ensamblador claro y seguro que respete la ABI de ARM64, con un modelo de arreglos simple y consistente, y utilidades mínimas de runtime.

Qué cambió en esta fase (ARM64 + arreglos irregulares)

En esta entrega el foco estuvo en el backend y el runtime:

- Generación de código para **AArch64**, respetando convención de llamada y alineación de pila.
- **Jagged arrays**: el arreglo exterior es de punteros de 8 bytes; cada subarreglo lleva su propia cabecera y datos.
- **Arrays.add / Arrays.indexOf**: rutas separadas para elementos de 4B y 8B, incluyendo punteros y `double`.
- **foreach** sobre `int[][]`: carga con stride de 8B y guardia `null` para filas no inicializadas.
- **Strings**: duplicación controlada (cuando se almacenan o retornan) para evitar aliasing con buffers.

Gramática formal de Javalang

```
%start programa;

programa: lista_declaraciones;

lista_declaraciones: lista_declaraciones declaracion
                   | declaracion
                   ;

// Reglas para tipos primitivos
declaracion:    tipoPrimitivo corchetes_lista TOKEN_IDENTIFIER '(' lista_parametros_opt ')'
                | declaracion_funcion
                | declaracion_main
                | tipoPrimitivo corchetes_lista TOKEN_IDENTIFIER var_tail ';'
                | TOKEN_FINAL tipoPrimitivo corchetes_lista TOKEN_IDENTIFIER var_tail ';'
                ;

declaracion_funcion:    TOKEN_VOID TOKEN_IDENTIFIER '(' lista_parametros_opt ')' bloque
                        ;

lista_parametros_opt:   lista_parametros
                        | %empty
                        ;

lista_parametros:   lista_parametros ',' parametro
                    | parametro
                    ;
```

```
parametro:  tipoPrimitivo corchetes_lista TOKEN_IDENTIFIER corchetes_lista
            ;

declaracion_main:   TOKEN_PUBLIC TOKEN_STATIC TOKEN_VOID TOKEN_MAIN
                    '(' TOKEN_DSTRING '[' ']' TOKEN_IDENTIFIER ')' bloque
                    ;

lista_argumentos_opt:   lista_Expr
                        | %empty
                        ;

lSentencia: lSentencia sentencia
            | sentencia
            ;

sentencia:  bloque
            | declaracion_var ';'
            | if_stmt
            | while_stmt
            | for_stmt
            | switch_stmt
            | TOKEN_BREAK ';'
            | TOKEN_CONTINUE ';'
            | TOKEN_RETURN expr ';'
            | TOKEN_RETURN ';'
            | expr ';'
            | error ';'
            ;

lista_Expr: lista_Expr ',' lista_item
            | lista_item

lista_item: expr
            | inicializador_arreglo
            ;

bloque: '{' lSentencia '}'
        | '{' '}'
        ;

declaracion_var:    tipoPrimitivo declarador_completo
                    | TOKEN_FINAL tipoPrimitivo declarador_completo
                    ;


declarador_completo: corchetes_lista TOKEN_IDENTIFIER corchetes_lista '=' inicializador_arr
                    | corchetes_lista TOKEN_IDENTIFIER corchetes_lista '=' expr
                    | corchetes_lista TOKEN_IDENTIFIER corchetes_lista
                    ;

corchetes_lista:    corchetes_lista '[' ']'
                    | %empty
                    ;

inicializador_arreglo : '{' lista_Expr '}'
                        | '{' '}'
                        ;
```

```
// Cola de declarador para nivel superior (dimensiones derechas y opcional inicializador)
var_tail: corchetes_lista '=' inicializador_arreglo
        | corchetes_lista '=' expr
        | corchetes_lista
        ;


asignacion_expr: TOKEN_IDENTIFIER '=' expr
               | primary_expr '[' expr ']' '=' expr
               | TOKEN_IDENTIFIER TOKEN_PLUS_ASSIGN expr
               | TOKEN_IDENTIFIER TOKEN_MINUS_ASSIGN expr
               | TOKEN_IDENTIFIER TOKEN_MULT_ASSIGN expr
               | TOKEN_IDENTIFIER TOKEN_DIV_ASSIGN expr
               | TOKEN_IDENTIFIER TOKEN_MOD_ASSIGN expr
               | TOKEN_IDENTIFIER TOKEN_AND_ASSIGN expr
               | TOKEN_IDENTIFIER TOKEN_OR_ASSIGN expr
               | TOKEN_IDENTIFIER TOKEN_XOR_ASSIGN expr
               | TOKEN_IDENTIFIER TOKEN_LSHIFT_ASSIGN expr
               | TOKEN_IDENTIFIER TOKEN_RSHIFT_ASSIGN expr
               | TOKEN_IDENTIFIER TOKEN_URSHIFT_ASSIGN expr
               ;


expr: asignacion_expr
    | expr TOKEN_OR expr
    | expr TOKEN_AND expr
    | expr '|' expr
    | expr '^' expr
    | expr '&' expr
    | expr TOKEN_IGUAL_IGUAL expr
    | expr TOKEN_DIFERENTE expr
    | expr '<' expr
    | expr '>' expr
    | expr TOKEN_MENOR_IGUAL expr
    | expr TOKEN_MAYOR_IGUAL expr
    | expr TOKEN_LSHIFT expr
    | expr TOKEN_RSHIFT expr
    | expr TOKEN_URSHIFT expr
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | expr '%' expr
    | primary_expr
    ;


primary_expr: '(' expr ')'
            | '(' tipoPrimitivo ')' primary_expr
            | TOKEN_IDENTIFIER
            | primary_expr '[' expr ']'
            | expresion_creacion_arreglo
            | TOKEN_PARSE_INT '(' expr ')'
            | TOKEN_PARSE_FLOAT '(' expr ')'
            | TOKEN_PARSE_DOUBLE '(' expr ')'
            | TOKEN_STRING_VALUEOF '(' expr ')'
```

```
            | TOKEN_STRING_JOIN '(' expr ',' lista_Expr ')'
            | TOKEN_ARRAYS_INDEXOF '(' expr ',' expr ')'
            | primary_expr TOKEN_DOT TOKEN_LENGTH
            /* soporte de método de instancia: primary_expr . IDENT '(' expr ')' para add/e
            | TOKEN_IDENTIFIER '(' lista_argumentos_opt ')'
            | TOKEN_PRINT '(' lista_Expr ')'
            | primitivo
            | '-' primary_expr %prec NEG
            | '!' primary_expr
            | '~' primary_expr
            | primary_expr TOKEN_INCREMENTO
            | primary_expr TOKEN_DECREMENTO
            | primary_expr TOKEN_DOT TOKEN_IDENTIFIER '(' expr ')'
            ;

primitivo: TOKEN_INTEGER
    | TOKEN_HEX_INTEGER
    | TOKEN_FLOAT_LIT
    | TOKEN_DOUBLE_LIT
    | TOKEN_STRING_LITERAL
    | TOKEN_CHAR_LITERAL
    | TOKEN_TRUE
    | TOKEN_FALSE
    ;


tipoPrimitivo: TOKEN_DINT
    | TOKEN_DFLOAT
    | TOKEN_DDOUBLE
    | TOKEN_DSTRING
    | TOKEN_DCHAR
    | TOKEN_DBOOLEAN
    ;


if_stmt: TOKEN_IF '(' expr ')' bloque %prec IFX
        | TOKEN_IF '(' expr ')' bloque TOKEN_ELSE bloque
        | TOKEN_IF '(' expr ')' bloque TOKEN_ELSE if_stmt {
        ;


while_stmt: TOKEN_WHILE '(' expr ')' bloque;


for_init: declaracion_var
    | expr            ;

for_expr_opt: expr
            | %empty
            ;


for_stmt:
    TOKEN_FOR '(' for_head ')' bloque
    ;


for_head:
```

```
    tipoPrimitivo corchetes_lista TOKEN_IDENTIFIER corchetes_lista ':' expr
    | tipoPrimitivo corchetes_lista TOKEN_IDENTIFIER corchetes_lista '=' inicializador_arre
    | tipoPrimitivo corchetes_lista TOKEN_IDENTIFIER corchetes_lista '=' expr ';' for_expr_
    | tipoPrimitivo corchetes_lista TOKEN_IDENTIFIER corchetes_lista ';' for_expr_opt ';' f
    | for_init_opt ';' for_expr_opt ';' for_expr_opt
    ;


for_init_opt: for_init
    | %empty
    ;


switch_stmt: TOKEN_SWITCH '(' expr ')' '{' case_list '}' ;



case_list: case_list case_stmt
        | case_stmt
        ;

case_stmt:  TOKEN_CASE expr ':' lSentencia_opt
            | TOKEN_DEFAULT ':' lSentencia_opt
            ;


lSentencia_opt: lSentencia
                | %empty
                ;


expresion_creacion_arreglo: TOKEN_NEW tipoPrimitivo lista_dims_expr dims_vacias
                          | TOKEN_NEW tipoPrimitivo lista_dims_expr
                          ;

lista_dims_expr: lista_dims_expr '[' expr ']'
                | '[' expr ']'
                ;

dims_vacias
    : dims_vacias '[' ']'
    | '[' ']'
    ;
```

Cómo leer esta gramática

- Es una especificación Bison/Yacc. Si ves `%empty`, significa "producción vacía".
- Las reglas de `for_stmt`, `if_stmt`, `switch_stmt` y `expresion_creacion_arreglo` reflejan el comportamiento del lenguaje de entrada (similar a Java).
- Para arreglos, presta atención a `inicializador_arreglo`, `lista_dims_expr` y `dims_vacias`: de aquí salen los casos regulares y jagged.

> Tip: si quieres ubicar una construcción del lenguaje en el código, empieza por la regla de la gramática

> y busca su visitante/constructor en `src/` (AST) y luego su emisión en `codegen`.

Arquitectura y módulos del proyecto

La estructura completa está detallada en el Proyecto 1, pero aquí va un resumen práctico de esta fase:

- `entriesTools/lexer.l` y `entriesTools/parser.y`: análisis léxico y sintáctico (Flex/Bison).
- `src/ast/*`: construcción del AST (nodos y recorrido).
- `src/context/*`: contexto de ejecución/compilación, tipos, resultados y valores de arreglos.
- `src/utils/*`: utilidades (formato de números estilo Java, conversión de char a UTF-8, etc.).
- `src/*_reporter.c`: generación de reportes (errores, símbolos, AST).
- `src/ast_grapher.*`: exportación de AST en DOT/SVG/PDF.
- `src/output_buffer.*`: manejo de salida (bufferizado) para impresión.
- `src` (backend ARM64): emisores por categoría (declaraciones, reasignaciones, expresiones) y helpers de runtime para arreglos.

> Si buscas "dónde se crea un arreglo en memoria", revisa los helpers en `src/context/` y las llamadas desde el codegen ARM64.

Generación de código AArch64 (cómo lo hacemos)

La generación sigue el flujo clásico: del AST emitimos **ensamblador ARM64** que luego se ensambla, enlaza y ejecuta. A continuación, los puntos clave.

Punto de entrada y contrato

Para compilar desde la GUI/CLI, llamamos al generador con la raíz del AST y la ruta de salida.

Fragmento (GUI), `src/main.c`:

```
// Asegurar carpeta de salida
mkdir("arm", 0777);
// Generar ensamblador ARM64 en arm/salida.s
int rc = arm64_generate_program(ast_root, "arm/salida.s");
```

Contrato del generador, `src/codegen/arm64_codegen.h`:

```
// Genera un archivo en ensamblador AArch64
int arm64_generate_program(AbstractExpresion *root, const char *out_path);
```

Convención de llamada y registros

- Parámetros escalares en **w0–w7/x0–x7**; flotantes en **d0–d7**. Retornos en **w0/x0** o **d0**.

- Respetamos registros caller/callee-saved en nuestros helpers (p. ej., `new_array_flat_ptr`).
- Pila alineada a 16 bytes; uso de **x29** como frame pointer.

Marco de pila y locales

- Cada función reserva un frame fijo de **1024 bytes** al entrar; locales por offsets relativos a `x29`.
- No hay subajustes dinámicos de `sp` dentro del cuerpo: menos riesgo de desalineación y corrupción.

Literales y utilidades de impresión

- Recolectamos literales (`core_add_string_literal` / `core_add_double_literal`) en `.data`.
- Formatos listos para `printf`: `fmt_int`, `fmt_double`, `fmt_string`, `fmt_char` y `fmt_ptr` (útil para diagnosticar punteros).
- Soporte para `char` → UTF-8 (`char_to_utf8`) y formato de `double` estilo Java (`java_format_double`).

Modelo de arreglos y helpers

- Memoria de arreglo:
  - Cabecera: `dims` (int32, offset 0) + `sizes[dims]` (int32 c/u) + padding hasta 8 bytes.
  - Datos alineados a 8B justo después de la cabecera.
- Tamaños:
  - Elementos tipo int-like: **4 bytes** (`int`, `char`, `boolean`).
  - Punteros y `double`: **8 bytes**.
- Constructores:
  - `new_array_flat` (4B) y `new_array_flat_ptr` (8B, limpia a cero los datos).

  Extracto del helper de 8 bytes:

```
new_array_flat_ptr:
        stp x29, x30, [sp, -16]!
        mov x29, sp
        // ... calcula tamaño cabecera + datos (8B por elemento) ...
        mov x0, x22        // tamaño total
        bl malloc          // reservar
        mov x23, x0        // base del bloque
        str w19, [x23]     // dims
        // ... copiar sizes[] y limpiar zona de datos a cero ...
        ret
```

- Direccionamiento:
  - `array_element_addr` (4B) y `array_element_addr_ptr` (8B). Soportan acceso lineal o pointer-of-pointer.

Extracto de direccionamiento 8B plano:

```
// x0 = arr_ptr, x1 = indices, w2 = num_indices
array_element_addr_ptr:
        // ... calcula offset lineal con strides ...
        add x0, x9, x17
        add x0, x0, x19, lsl #3   // 8 bytes por elemento
        ret
```

- Jagged arrays:
  - El arreglo exterior es **1D de punteros (8B)**; cada posición apunta a un subarreglo con su cabecera.
  - Los inicializadores anidados preservan el puntero a cabecera viva y calculan la base de datos a partir de esa cabecera.

Expresiones y sentencias relevantes

- **Arrays.add (a.add(elem))**
  - Reserva nuevo arreglo de tamaño `n+1` con el helper correcto (4B u 8B).
  - Copia con strides acordes y escribe el último elemento; si es string, se **duplica** (`strdup`).

Decisiones de diseño (por qué así)

- **Frame fijo (1024B)**: hace el direccionamiento trivial con `[x29 - N]` y reduce bugs por desalineación.
- **Cabeceras a 8B**: acceso natural para ints y punteros/doubles sin trampas.
- **Doble familia de helpers (4B / 8B)**: el código generado queda simple y explícito.
- **Regla de "elemento puntero"**: strings, `double` y subarreglos se tratan como 8B; esto simplifica jagged arrays.
- **Cadenas "estables"**: `strdup` al almacenar/retornar; así no dependemos de buffers temporales.
- **Buffers separados** (`tmpbuf` y `joinbuf`): evitan colisiones en concatenaciones y joins.
- **Guardias null en foreach**: resiliencia en arreglos parcialmente inicializados.
- **Instrumentación opcional**: puntos de log en caminos críticos que se pueden apagar sin tocar el usuario final.

Desafíos y cómo se resolvieron

- Integración **AArch64 ABI**:
  - Validamos que llamadas a libc (`printf` , `malloc` , `strcat` , `strdup` , `strtol`) no corrompan registros vitales.
  - En helpers propios, preservamos callee-saved y documentamos el contrato de entrada/salida.
- **Arreglos irregulares**:

- Cálculo correcto de la base de datos tras `sizes[]` con padding a 8B.
- Direccionamiento pointer-of-pointer para accesos y `foreach`.

- Caso difícil: **segfault** en prueba integral
  - Hallazgo: en inicializadores 1D, **x0** (puntero a cabecera) se clobberaba tras llamadas anidadas (`strtol`, etc.). Resultado: se guardaba un puntero inválido.
  - Fix: preservar cabecera en **x20**, calcular base desde **x20** y restaurar **x0** antes de devolver. Con esto, el puntero guardado es válido y estable.
  - Además, reforzamos `Arrays.add` de 8B (copias con `ldr/str xN`) y añadimos guardias en `foreach`.

- **Strings y buffers temporales**:
  - Decidimos duplicar retornos/almacenes y separar buffers para eliminar aliasing y corrupción.

- **Alineación y frame**:
  - Mantener `sp` alineado en todas las rutas y evitar subajustes durante el cuerpo de las funciones.

Pruebas y cómo correrlas

- Entorno: ensamblamos **ARM64** y ejecutamos bajo **QEMU aarch64** con glibc del sistema.
- Para instalar todas las herramientas necesarias (toolchain AArch64 y QEMU), consulta el README en la raíz del proyecto; ahí están los paquetes sugeridos y cómo verificarlos.
- Pruebas destacadas:
  - `test/proyecto1/examen_final.usl`: mezcla arreglos 2D/jagged, `foreach`, recursión, `switch/while`, `String.join/valueOf`, `parseInt`, `Arrays.add/indexOf`.
  - `test/proyecto1/prueba_arreglos2.usl`, `prueba_foreach*.usl`, `prueba_funciones.usl`, `prueba_doubles.usl`: cubren rutas específicas de tamaño de elemento y control de flujo.

- Hallazgos:
  - Se eliminaron fallas por stride/tamaño en `Arrays.add` y se estabilizó el manejo de strings.
  - El clobber en inicializadores 1D quedó corregido preservando cabecera y restaurando registros al final.

- Cobertura (cualitativa):
  - Cubierto: declaraciones, reasignaciones, creación 1D/2D/jagged, accesos e iteración, joins, casteos, recursión.
  - En seguimiento: `add` sobre arreglos de punteros en escenarios complejos (mezcla de inicializadores y parseos anidados).

> Próximo paso: agregar un test mínimo de regresión para "crear `int[]` con parseos + `append` en `int[][]`" y evitar regresiones futuras.

Glosario breve

- **ABI**: reglas de llamada entre funciones (qué va en qué registro, quién guarda qué, etc.).
- **Frame (marco de pila)**: bloque reservado en la pila para locales y temporales durante una función.
- **Jagged array**: arreglo de arreglos donde cada "fila" puede tener tamaño distinto.
- **Stride**: cantidad de bytes que se avanza para ir de un elemento al siguiente.
- **Helper de runtime**: función de apoyo (en C/ASM) que realiza tareas comunes como reservar y preparar un arreglo.

¿Dudas o quieres profundizar en alguna parte? Revisa los fuentes en `src/` (están organizados por responsabilidad) y, si estás depurando, habilita la instrumentación opcional para ver lo que pasa sin romper el flujo del usuario.