Informe de Laboratorio 12

Analizador Léxico-Sintáctico para Mini-0

Nota

Estudiantes	Escuela	Asignatura
Piero Omar De La Cruz Mancilla Enyelbert Anderson Panta Huaracha	Carrera Profesional de Ingeniería de Software	Compiladores

Correos	Semestre	Código
pdelacruzm@ulasalle.edu.pe epantah@ulasalle.edu.pe	V	3.5.6.21

Laboratorio	Tema	Duración
12	Analizador Léxico-Sintáctico para Mini-0	2 semanas

1. Introducción

Este proyecto implementa un **compilador completo para el lenguaje Mini-0** que consta de dos fases fundamentales: análisis léxico y análisis sintáctico. El lenguaje Mini-0 es un lenguaje imperativo educativo diseñado para demostrar los principios fundamentales de la construcción de compiladores.

La fase de **análisis léxico** utiliza **Flex** (Fast Lexical Analyzer) para convertir el código fuente en una secuencia de tokens significativos, reconociendo 43 tipos diferentes de tokens incluyendo palabras reservadas, operadores, identificadores y literales.

La fase de **análisis sintáctico** implementa un **parser recursivo descendente** escrito en C que valida la estructura sintáctica del programa mediante análisis **LL(1)**. La gramática original fue transformada eliminando recursión izquierda y ambigüedades para hacerla compatible con este tipo de análisis.

El proyecto incluye un sistema robusto de **detección y reporte de errores**, tanto léxicos como sintácticos, con información precisa sobre la ubicación y naturaleza de cada error. Además, se proporciona documentación formal completa que incluye la **tabla de análisis LL(1)** con conjuntos FIRST y FOLLOW, así como una suite exhaustiva de casos de prueba.

2. Objetivos

2.1. Objetivo General

Implementar un analizador léxico-sintáctico completo para el lenguaje Mini-0 utilizando técnicas de análisis recursivo descendente LL(1).

2.2. Objetivos Específicos

- Desarrollar un analizador léxico robusto usando Flex que reconozca todos los tokens del lenguaje Mini-0
- Transformar la gramática original para eliminar recursión izquierda y ambigüedades
- Implementar un parser recursivo descendente en C que valide la sintaxis del lenguaje
- Documentar formalmente la gramática LL(1) con conjuntos FIRST, FOLLOW y tabla de parsing
- Crear un sistema de manejo de errores que reporte información precisa y útil
- Desarrollar una suite completa de casos de prueba que cubra todas las reglas gramaticales

3. Arquitectura del Sistema

3.1. Estructura del Proyecto

Listing 1: Estructura de directorios del compilador

```
mini0-compiler/
|----Makefile                  # Sistema de construccin
|----mini0_lex.l               # Especificacin  lxica  Flex
|----token.h                   # Definiciones de tokens
|----token.c                   # Implementacin de tokens
|----parser.h                  # Interfaz del parser
|----parser.c                  # Parser recursivo descendente
|----readme.md                 # Documentacin de usuario
|----report.md                 # Informe tcnico  bsico
|----ll1_analysis.md           #  Anlisis  LL(1) formal completo
|----test_basic.mini0          # Prueba bsica
|----test_hex.mini0            # Prueba literales hex
|----test_string.mini0         # Prueba strings
|----test_errors.mini0         # Prueba errores lxicos
|----test_completo.mini0       # Prueba exhaustiva
+----tests/                    # Suite de pruebas
    |----parser_valid.mini0
    |----parser_error_missing_end.mini0
    +----parser_error_bad_syntax.mini0
```

3.2. Componentes Principales

- **Analizador Léxico**: Generado por Flex a partir de `mini0_lex.l`
- **Sistema de Tokens**: Módulo `token.c/h` con estructuras y utilidades
- **Parser Sintáctico**: Analizador recursivo descendente en `parser.c`

- **Sistema de Construcción**: Makefile que genera dos ejecutables independientes
- **Documentación Formal**: Análisis LL(1) completo con tablas de parsing

3.3. Diagrama de Flujo del Compilador

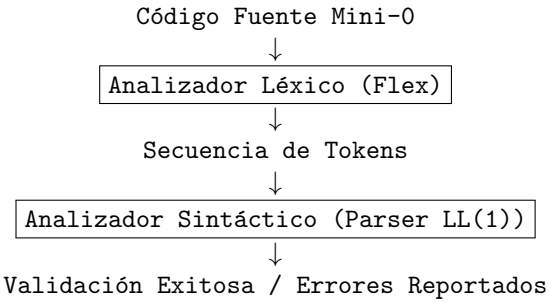

Figura 1: Pipeline del compilador Mini-0

4. Fase 1: Análisis Léxico

4.1. Especificación del Lenguaje Mini-0

El lenguaje Mini-0 incluye las siguientes características:

- **Tipos de datos**: `int`, `bool`, `string`, `char`
- **Arreglos**: Sintaxis `[]type`, con soporte multidimensional
- **Funciones**: Con parámetros y valores de retorno tipados
- **Estructuras de control**: `if/else/end`, `while/loop`
- **Operadores**: Aritméticos (+, -, *, /), relacionales (¿, ¡, ¿=, ¡=), lógicos (and, or, not)
- **Literales**: Números decimales, hexadecimales (0x...), strings con escapes

4.2. Conjunto de Tokens

Cuadro 1: Tokens del lenguaje Mini-0

Categoría	Tokens	Cantidad
Palabras reservadas	if, else, end, while, loop, fun, return, etc.	17
Operadores	+, -, *, /, ¿, ¡, ¿=, ¡=, =, ¡¿, and, or, not	13
Delimitadores	(,), [,], ,, :	6
Identificadores	ID (letra/guión bajo + alfanumérico)	1
Literales	LITNUMERAL, LITSTRING	2
Especiales	NL (nueva línea), ERROR	2
Total		**43**

4.3. Patrones Léxicos

Listing 2: Definiciones de patrones en Flex

```
LETRA          [a-zA-Z]
DIGITO         [0-9]
HEXDIGITO      [0-9a-fA-F]
ID             ({LETRA}|_)({LETRA}|{DIGITO}|_)*
DECIMAL        {DIGITO}+
HEXADECIMAL    0x{HEXDIGITO}+
STRING         \"([^\\\"\n]|\\.)*\"
ESPACIO        [ \t\r]
SALTO_LINEA    \n
```

4.4. Procesamiento de Literales Especiales

4.4.1. Números Hexadecimales

Listing 3: Conversión de hexadecimal a decimal

```c
int hex_to_dec(const char *hex) {
    int valor;
    sscanf(hex, "%x", &valor);
    return valor;
}

// Ejemplos de uso:
// 0xFF -> 255
// 0x0A -> 10
// 0xDEADBEEF -> 3735928559
```

4.4.2. Strings con Caracteres de Escape

Cuadro 2: Caracteres de escape soportados

Escape	Significado
\n	Nueva línea (newline)
\t	Tabulación horizontal
\\	Backslash literal
\”	Comilla doble literal

Listing 4: Función de procesamiento de escapes

```c
char* procesar_string(const char *str) {
    int len = strlen(str);
    char *resultado = (char*)malloc(len + 1);
    int j = 0;

    // Saltar comillas inicial y final
    for (int i = 1; i < len - 1; i++) {
        if (str[i] == '\\' && i + 1 < len - 1) {
            switch(str[i + 1]) {
                case 'n': resultado[j++] = '\n'; i++; break;
                case 't': resultado[j++] = '\t'; i++; break;
                case '\\': resultado[j++] = '\\'; i++; break;
```

```
                case '"': resultado[j++] = '"'; i++; break;
                default: resultado[j++] = str[i]; break;
            }
        } else {
            resultado[j++] = str[i];
        }
    }
    resultado[j] = '\0';
    return resultado;
}
```

4.5. Manejo de Comentarios

El lexer soporta dos tipos de comentarios que son ignorados durante el análisis:

- **Línea simple**: `// comentario hasta fin de línea`
- **Multilínea**: `/* comentario en múltiples líneas */`

Listing 5: Reglas Flex para comentarios

```
"//".*                { /* Ignorar comentario de lnea */ }
"/*"                  {
    int c;
    while ((c = input()) != 0) {
        if (c == '\n') linea_actual++;
        if (c == '*') {
            if ((c = input()) == '/') break;
            unput(c);
        }
    }
}
```

5. Fase 2: Análisis Sintáctico

5.1. Parser Recursivo Descendente

El parser implementa un analizador sintáctico top-down que construye el árbol de derivación desde la raíz hacia las hojas. Cada función del parser corresponde a un no-terminal de la gramática.

5.2. Transformaciones de la Gramática

5.2.1. Eliminación de Recursión Izquierda

Problema Original:

$$E \to E + T \mid E - T \mid T$$

Transformación Aplicada:

$$E \to T\ E'$$
$$E' \to +\ T\ E' \mid -\ T\ E' \mid \varepsilon$$

5.2.2. Jerarquía de Precedencia

Cuadro 3: Niveles de precedencia de operadores (de menor a mayor)

Nivel	Operadores	No-terminal
1	or	or-expr
2	and	and-expr
3	=, ¡¿	equality-expr
4	¿, ¡, ¿=, ¡=	relational-expr
5	+, -	additive-expr
6	*, /	multiplicative-expr
7	-, not (unarios)	unary-expr
8	literales, ( )	primary-expr

5.2.3. Resolución de Ambigüedades

Ambigüedad 1: Declaración vs Asignación

Listing 6: Código ambiguo

```
x: int     // Declaracion
x = 10     // Asignacion
```

Solución: Lookahead de 1 token

Listing 7: Resolución en el parser

```
if (cur_token()->tipo == TK_ID &&
    peek_token(1)->tipo == TK_COLON) {
    // Es declaracion: ID : type
    parse_declaration();
} else if (cur_token()->tipo == TK_ID &&
          (peek_token(1)->tipo == TK_EQ ||
           peek_token(1)->tipo == TK_LBRACKET)) {
    // Es asignacion: ID = expr o ID[expr] = expr
    parse_assignment();
}
```

Ambigüedad 2: Operador = Dual
El símbolo = tiene dos significados:

- En contexto de sentencia: operador de asignación
- En contexto de expresión: operador de comparación

Solución: Separación sintáctica por contexto en las producciones de la gramática.

5.3. Gramática LL(1) Resultante

La gramática transformada consta de **70 producciones** organizadas en secciones:

- Producciones 1-14: Programa y declaraciones
- Producciones 15-27: Sentencias
- Producciones 28-70: Expresiones con precedencia

5.3.1. Ejemplo de Producciones

Listing 8: Estructura de programa

```
⟨program⟩ → ⟨decl-or-fun-list⟩

⟨decl-or-fun⟩ → fun ID ( ⟨param-list⟩ ) : ⟨type⟩ NL* ⟨stmt-list⟩ end NL*
              | ID : ⟨type⟩ NL*

⟨type⟩ → ⟨array-prefix⟩ ⟨base-type⟩
⟨array-prefix⟩ → [ ] ⟨array-prefix⟩ | ε
⟨base-type⟩ → int | bool | string | char
```

5.4. Análisis LL(1): Conjuntos FIRST y FOLLOW

5.4.1. Conjuntos FIRST (ejemplos)

$$\begin{aligned}\text{FIRST}(\langle\text{program}\rangle) &= \{\texttt{fun}, \texttt{ID}, \varepsilon\}\\ \text{FIRST}(\langle\text{type}\rangle) &= \{[\,], \texttt{int}, \texttt{bool}, \texttt{string}, \texttt{char}\}\\ \text{FIRST}(\langle\text{expr}\rangle) &= \{(\,), \texttt{ID}, \texttt{LITNUMERAL}, \texttt{LITSTRING},\\ &\qquad \texttt{-}, \texttt{not}, \texttt{true}, \texttt{false}\}\end{aligned}$$

5.4.2. Conjuntos FOLLOW (ejemplos)

$$\begin{aligned}\text{FOLLOW}(\langle\text{program}\rangle) &= \{\$\}\\ \text{FOLLOW}(\langle\text{type}\rangle) &= \{\texttt{NL}, \texttt{,}, \texttt{)}\}\\ \text{FOLLOW}(\langle\text{expr}\rangle) &= \{\texttt{NL}, \texttt{,}, \texttt{)}, \texttt{]}, \texttt{and}, \texttt{or}, \ldots\}\end{aligned}$$

5.4.3. Verificación de Propiedad LL(1)

Condiciones LL(1): Para cada no-terminal A con producciones $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$:

1. $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \emptyset$ para $i \neq j$
2. Si $\varepsilon \in \text{FIRST}(\alpha_i)$, entonces $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \emptyset$ para todo $j \neq i$

Verificación: Todas las producciones cumplen ambas condiciones.

5.5. Arquitectura del Parser

Listing 9: Funciones principales del parser

```
// Estructura del programa
static void parse_program();
static void parse_declaration_or_function();

// Tipos
static void parse_type();

// Sentencias
```

```
static void parse_statement();
static void parse_statements_until(TipoToken t1, TipoToken t2,
                                   bool allow_else);

// Expresiones (por nivel de precedencia)
static void parse_expression();
static void parse_or_expr();
static void parse_and_expr();
static void parse_equality();
static void parse_relational();
static void parse_additive();
static void parse_multiplicative();
static void parse_unary();
static void parse_primary();
```

5.6. Sistema de Lookahead

Listing 10: Mecanismo de inspección de tokens

```
// Token actual
static Token *cur_token() {
    if (pos >= num_tokens) return NULL;
    return tokens[pos];
}

// Token a distancia 'offset' del actual
static Token *peek_token(int offset) {
    int idx = pos + offset;
    if (idx >= num_tokens) return NULL;
    return tokens[idx];
}

// Avanzar al siguiente token
static void advance() {
    if (pos < num_tokens) pos++;
}
```

6. Manejo de Errores

6.1. Estrategia de Detección

El compilador implementa detección de errores en dos niveles:

1. **Errores Léxicos**: Detectados por Flex durante la tokenización
2. **Errores Sintácticos**: Detectados por el parser durante la validación

6.2. Errores Léxicos

Cuadro 4: Tipos de errores léxicos detectados

Error	Descripción	Ejemplo
Carácter inválido	Símbolo no pertenece al alfabeto	`$`, `@`, `#`
String sin cerrar	Cadena sin comilla final	`"hola mundo`
Hex inválido	0x sin dígitos hex válidos	`0xGHI`

6.2.1. Ejemplo de Error Léxico

Listing 11: test_errors.mini0

```
fun main(): int
    x = 10$20     // Error: '$' es invalido
    y = @variable // Error: '@' es invalido
    z = test#error // Error: '#' es invalido
    return 0
end
```

Salida del compilador:

```
Error lexico en linea 2: caracter no reconocido '$'
Error lexico en linea 3: caracter no reconocido '@'
Error lexico en linea 4: caracter no reconocido '#'
```

6.3. Errores Sintácticos

Cuadro 5: Tipos de errores sintácticos detectados

Error	Mensaje
Token esperado	Se esperaba ')' después de lista de parámetros
Tipo inválido	Tipo desconocido (se esperaba int, bool, string o char)
Bloque incompleto	Bloque no terminado (se esperaba 'end' o 'loop')
Expresión inválida	Expresión no válida
Sentencia no reconocida	Sentencia no reconocida

6.3.1. Ejemplos de Errores Sintácticos

Error 1: Paréntesis sin cerrar

Listing 12: Código con error

```
fun main(): int
    x: int
    x = (5 + 3 // Falta ')'
    return 0
end
```

Listing 13: Salida del parser

```
Error sintactico en linea 3: Se esperaba ')' despues de la expresion
(token: LITNUMERAL '3')
```

Error 2: Bloque sin terminar

Listing 14: Código con error

```
fun main(): int
    x: int
    x = 10
    if x > 5
        x = x + 1
    // Falta 'end' aqui
```

Listing 15: Salida del parser

```
Error sintactico en EOF: Bloque no terminado correctamente
(se esperaba 'end' o 'loop')
```

6.4. Función de Reporte

Listing 16: Sistema de reporte de errores

```
static void error_at_token(Token *tk, const char *msg) {
    if (!tk) {
        fprintf(stderr, "Error sintactico en EOF: %s\n", msg);
    } else {
        fprintf(stderr,
                "Error sintactico en linea %d: %s (token: %s '%s')\n",
                tk->linea, msg,
                tipo_token_str(tk->tipo),
                tk->lexema);
    }
    had_error = true;
}
```

6.5. Códigos de Salida

Cuadro 6: Códigos de retorno del programa

Código	Significado
0	Análisis exitoso sin errores
1	Error al abrir archivo de entrada
2	Errores léxicos o sintácticos detectados

7. Sistema de Construcción

7.1. Makefile

Listing 17: Makefile del proyecto

```
CC = gcc
CFLAGS = -Wall -g
LEX = flex

all: mini0_lex mini0_parser

mini0_lex: lex.yy.o token.o
  $(CC) $(CFLAGS) -o mini0_lex lex.yy.o token.o -lfl

mini0_parser: lex.yy_nomain.o token.o parser.o
  $(CC) $(CFLAGS) -DBUILD_WITH_PARSER_MAIN \
        -o mini0_parser lex.yy_nomain.o token.o parser.o -lfl

lex.yy.o: lex.yy.c
  $(CC) $(CFLAGS) -c lex.yy.c

lex.yy_nomain.o: lex.yy.c
  $(CC) $(CFLAGS) -c -DBUILD_WITH_PARSER_MAIN \
        -o lex.yy_nomain.o lex.yy.c

lex.yy.c: mini0_lex.l
  $(LEX) mini0_lex.l

parser.o: parser.c parser.h token.h
  $(CC) $(CFLAGS) -c parser.c

token.o: token.c token.h
  $(CC) $(CFLAGS) -c token.c

clean:
  rm -f mini0_lex mini0_parser lex.yy.c *.o

.PHONY: all clean
```

7.2. Proceso de Compilación

1. Flex genera `lex.yy.c` desde `mini0_lex.l`
2. GCC compila cada módulo a archivos objeto (.o)
3. Se generan dos ejecutables:
   - `mini0_lex`: Analizador léxico standalone
   - `mini0_parser`: Parser completo (léxico + sintáctico)

8. Casos de Prueba

8.1. Programa Básico

Listing 18: test_basic.mini0

```
fun main(): int
    x: int
```

```
    y: bool
    x = 42
    y = true

    if x > 10
        return 1
    end

    return 0
end
```

Propósito: Verificar declaraciones, asignaciones, if y return.

8.2. Literales Hexadecimales

Listing 19: test_hex.mini0

```
fun main(): int
    x: int
    x = 0xFF   // 255 en decimal
    x = 0x0F   // 15 en decimal
    return x
end
```

Salida esperada:

```
LITNUMERAL [0xFF] -> valor: 255
LITNUMERAL [0x0F] -> valor: 15
```

8.3. Strings con Escapes

Listing 20: test_string.mini0

```
fun main(): int
    s: string
    s = "Hola\nMundo"
    s = "Tab:\taqui"
    s = "Comillas: \"hola\""
    return 0
end
```

8.4. Programa Completo

Listing 21: test_completo.mini0 - Ejercita múltiples características

```
contador: int

fun factorial(n: int): int
    resultado: int
    resultado = 1

    while n > 1
        resultado = resultado * n
        n = n - 1
```

```
    loop

    return resultado
end

fun main(): int
    numeros: []int
    x: int
    flag: bool

    numeros = new [10] int
    numeros[0] = 0xA5

    x = factorial(5)

    if x >= 100 and x <= 200
        flag = true
    else
        flag = false
    end

    return x
end
```

9. Despliegue y Ejecución

9.1. Requisitos del Sistema

- Sistema operativo: Linux (WSL en Windows)
- Herramientas: Flex 2.6+, GCC 11+, Make 4.3+
- Entorno: Alpine Linux o Ubuntu

9.2. Instalación de Dependencias

Listing 22: Instalación en Alpine Linux

```
# Abrir WSL
wsl

# Instalar dependencias
apk add flex gcc make musl-dev flex-dev
```

9.3. Compilación del Proyecto

Listing 23: Comandos de compilación

```
# Limpiar archivos previos
make clean

# Compilar todo el proyecto
make

```

```
# Salida esperada:
# flex mini0_lex.l
# gcc -Wall -g -c lex.yy.c
# gcc -Wall -g -c token.c
# gcc -Wall -g -o mini0_lex lex.yy.o token.o -lfl
# gcc -Wall -g -c parser.c
# gcc -Wall -g -o mini0_parser ...
```

9.4. Ejecución de Pruebas

9.4.1. Prueba del Lexer

Listing 24: Ejecutar analizador léxico

```
# Sintaxis
./mini0_lex <archivo.mini0>

# Ejemplo
./mini0_lex test_basic.mini0

# Salida
=== ANALISIS LEXICO COMPLETADO ===
Total de tokens encontrados: 42

Linea   1: FUN         [fun]
Linea   1: ID          [main]
Linea   1: LPAREN      [(]
...
```

9.4.2. Prueba del Parser

Listing 25: Ejecutar analizador sintáctico

```
# Caso exitoso
./mini0_parser tests/parser_valid.mini0
Analisis sintactico completado
$ echo $?
0

# Caso con error
./mini0_parser tests/parser_error_missing_end.mini0
Error sintactico en EOF: Bloque no terminado...
$ echo $?
2
```

10. Documentación Formal

10.1. Archivo ll1_analysis.md

Este documento contiene el análisis formal completo del parser:

- **Gramática formal**: 70 producciones numeradas

- **Conjuntos FIRST**: Para todos los no-terminales
- **Conjuntos FOLLOW**: Para todos los no-terminales
- **Tabla LL(1)**: Matriz completa de análisis sintáctico
- **Verificación LL(1)**: Prueba de condiciones
- **Conflictos resueltos**: Documentación de ambigüedades
- **Ejemplo de derivación**: Paso a paso

10.2. Enlace al Repositorio

Todo el código fuente, documentación y casos de prueba están disponibles en:

`https://github.com/EnyelbertAnderson/Compiladores_Lab_10`

11. Resultados y Análisis

11.1. Métricas del Proyecto

Cuadro 7: Estadísticas del código fuente

Métrica	Valor
Líneas de código C	∼800
Líneas de especificación Flex	∼200
Tipos de tokens	43
Producciones gramaticales	70
Funciones del parser	15
Casos de prueba	9
Palabras reservadas	17
Operadores soportados	18

11.2. Complejidad Algorítmica

- **Análisis léxico**: $O(n)$ donde n es el tamaño del archivo
- **Análisis sintáctico**: $O(n)$ donde n es el número de tokens
- **Memoria**: $O(n)$ para almacenar tokens
- **Lookahead**: Constante ($k = 1$ token)

11.3. Logros Principales

1. Gramática LL(1) completa sin ambigüedades
2. Parser recursivo descendente robusto
3. Sistema de detección de errores preciso
4. Documentación formal exhaustiva
5. Suite de pruebas completa
6. Código modular y extensible

12. Conclusiones

12.1. Conclusiones Técnicas

1. Se implementó exitosamente un compilador de dos fases para Mini-0 utilizando **Flex** para análisis léxico y un **parser recursivo descendente** para análisis sintáctico.
2. La **transformación de gramática** eliminando recursión izquierda y factorizando alternativas permitió construir un parser LL(1) eficiente con lookahead de un solo token.
3. El **sistema de manejo de errores** proporciona mensajes claros y precisos que facilitan la depuración de programas Mini-0.
4. La **documentación formal** con conjuntos FIRST, FOLLOW y tabla LL(1) demuestra el rigor académico del proyecto y sirve como referencia para futuras extensiones.
5. La arquitectura modular del código facilita la extensión del compilador con nuevas fases como análisis semántico y generación de código.

12.2. Aprendizajes Clave

- Técnicas formales de transformación de gramáticas
- Implementación de parsers recursivos descendentes
- Análisis LL(1) y construcción de tablas de parsing
- Diseño de sistemas de detección y reporte de errores
- Uso de herramientas profesionales (Flex, Make, GCC)
- Documentación técnica y académica de proyectos de software

12.3. Trabajo Futuro

1. **Fase 3 - Análisis Semántico**: Implementar tabla de símbolos, verificación de tipos y resolución de identificadores
2. **Fase 4 - Código Intermedio**: Generar representación en tres direcciones
3. **Fase 5 - Optimización**: Aplicar técnicas de optimización de código
4. **Fase 6 - Generación de Código**: Producir código ensamblador o bytecode
5. **Mejoras al Parser**: Recuperación de errores más sofisticada, mensajes con sugerencias

13. Referencias

1. Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley.
2. Levine, J. (2009). *flex & bison: Text Processing Tools*. O'Reilly Media.
3. Grune, D., van Reeuwijk, K., Bal, H. E., Jacobs, C. J. H., & Langendoen, K. (2012). *Modern Compiler Design* (2nd ed.). Springer.
4. Cooper, K. D., & Torczon, L. (2011). *Engineering a Compiler* (2nd ed.). Morgan Kaufmann.

5. Free Software Foundation. (2023). *GCC, the GNU Compiler Collection*. `https://gcc.gnu.org/`
6. The Flex Project. (2023). *Flex: The Fast Lexical Analyzer*. `https://github.com/westes/flex`
7. Especificación del lenguaje Mini-0. Documento del curso de Compiladores, Universidad La Salle, 2025.
8. Repositorio del proyecto: `https://github.com/EnyelbertAnderson/Compiladores_Lab_10`

14. Rúbrica de Evaluación

Cuadro 8: Rúbrica para contenido del Informe y evidencias

Contenido y demostración		Puntos	Checklist	Estudiante	Profesor
1. GitHub	Repositorio clonado con estructura adecuada y entregables revisables	3	X	3	
2. Commits	Código asociado a commits planificados con explicaciones detalladas	3	X	3	
3. Ejecución	Comandos de ejecución y pruebas explicados gradualmente	3	X	3	
4. Gramática	Transformaciones de gramática y tabla LL(1) documentadas	3	X	3	
5. Errores	Sistema de manejo de errores completo y documentado	2	X	2	
6. Pruebas	Suite de casos de prueba exhaustiva	2	X	2	
7. Ortografía	Documento sin errores ortográficos	2	X	2	
8. Madurez	Informe muestra evolución del código con explicaciones precisas, diagramas y acabado profesional	2	X	2	
Total		20		20	