Generador de Lexer

Abel Ponce González Josue Rolando Naranjo Seiro Gabriel Alonso Coro

23 de junio de 2025

Índice

1. Explicación del Flujo del Generador de Lexer

Este generador de lexer sigue un enfoque clásico de teoría de compiladores, transformando especificaciones basadas en expresiones regulares en un analizador léxico eficiente. A continuación se explica el flujo detallado del sistema dividido en partes lógicas.

1.1. Arquitectura General

El sistema está organizado en varios componentes interconectados:

- **Expresiones Regulares**: Representadas por **regex_parser.y** y procesadas por Bison.
- **Árbol de Sintaxis Abstracta (AST)**: Clases en el directorio `ast`.
- **Autómatas Finitos**: Implementación de NFAs y DFAs en `automata`.
- **Construcción de NFAs**: Clase `NfaBuild` en `build_nfa.cpp`.
- **Lexer**: Implementación final en `lexer_dfa.cpp` y `lexer_dfa.hpp`.

El flujo general es:

$$\text{Regex} \rightarrow \text{AST} \rightarrow \text{NFA} \rightarrow \text{DFA} \rightarrow \text{Lexer Funcional}$$

2. Procesamiento de Expresiones Regulares

En esta etapa transformamos la cadena de la expresión regular en un AST que luego servirá para construir los autómatas.

2.1. Análisis Léxico de la Expresión Regular

El análisis léxico de la expresión regular se realiza carácter a carácter en `simple_lexer.cpp`. Cada carácter se convierte en un token Bison mediante `yylex`:

Listing 1: Fragmento de `simple_lexer.cpp`

```
yy::parser::symbol_type yylex(RegexParserDriver& driver) {
    char c = driver.nextChar();
    bool in_char_class = driver.inCharClass();
    switch (c) {
        case '(':
            return yy::parser::make_LPAREN(driver.location);
        case ')':
            return yy::parser::make_RPAREN(driver.location);
        case '*':
            if (in_char_class)
                return yy::parser::make_CHAR('*', driver.location);
            else
                return yy::parser::make_STAR(driver.location);
        // ... otros casos para '+', '?', '|', etc.
```

```
        default:
            if (std::isalnum(c))
                return yy::parser::make_CHAR(c, driver.location);
            // ...
    }
}
```

2.2. Análisis Sintáctico de la Expresión Regular

El parser generado por Bison (a partir de `regex_parser.y`) utiliza la gramática siguiente para construir el AST. Aquí un fragmento de la regla de unión:

Listing 2: Regla Bison para unión en $\texttt{regex}_{p}arser.y$

```
union_expr:
      concat_expr
    | union_expr "|" concat_expr {
        auto binOp = std::make_shared<BinOp>($1, $3, BinaryOperator::Union);
        $$ = std::make_shared<Expression>(binOp);
    }
;
```

2.3. Interfaz de Análisis

La clase `RegexParserDriver` coordina el análisis léxico y sintáctico. Su método `parse` recibe la cadena de entrada y devuelve el AST resultante:

Listing 3: Método `RegexParserDriver::parse`

```
std::shared_ptr<Expression> RegexParserDriver::parse(const std::string& input) {
    // Inicializar el lexer con la entrada
    lexer_init(input);

    // Crear e invocar al parser de Bison
    yy::parser parser(*this);
    int result = parser.parse();

    // El AST queda en this->result
    return result == 0 ? this->result : nullptr;
}
```

3. Construcción del Autómata Finito No Determinista (NFA)

En esta fase transformamos el AST generado a un NFA que representa la expresión regular de forma automática.

3.1. Proceso de Construcción

La clase `NfaBuild` recorre el AST de manera recursiva y delega en métodos específicos según el tipo de nodo:

Listing 4: Método principal de `NfaBuild`

```
std::shared_ptr<NFA>
NfaBuild::buildFromExpression(const std::shared_ptr<Expression>& expr) {
    switch (expr->getType()) {
        case Expression::Type::Atom:
            return buildFromAtom(*expr->getAtom());
        case Expression::Type::BinOp:
            return buildFromBinaryOp(*expr->getBinOp());
        case Expression::Type::UnOp:
            return buildFromUnaryOp(*expr->getUnOp());
    }
    return nullptr; // nunca debera llegar aqu
}
```

3.2. Construcción desde Operadores Binarios

Para operadores binarios (concatenación y unión), se construyen los NFAs de los operandos y luego se combinan:

Listing 5: Construcción de NFA para operadores binarios

```
std::shared_ptr<NFA>
NfaBuild::buildFromBinaryOp(const BinOp& binOp) {
    auto leftNFA = buildFromExpression(binOp.left);
    auto rightNFA = buildFromExpression(binOp.right);
    switch (binOp.op) {
        case BinaryOperator::Concat:
            return NFA::concatenate(leftNFA, rightNFA);
        case BinaryOperator::Union:
            return NFA::alternate(leftNFA, rightNFA);
    }
    return nullptr;
}
```

3.3. Construcción desde Operadores Unarios

Para operadores unarios como Kleene star (`*`), plus (`+`) y question mark (`?`), se aplica la operación correspondiente al NFA del operando:

Listing 6: Construcción de NFA para operadores unarios

```
std::shared_ptr<NFA>
NfaBuild::buildFromUnaryOp(const UnOp& unOp) {
    auto operandNFA = buildFromExpression(unOp.operand);
    switch (unOp.op) {
        case UnaryOperator::KleeneStar:
            return NFA::kleeneStar(operandNFA);
```

```
        case UnaryOperator::Plus:
            return NFA::plus(operandNFA);
        case UnaryOperator::QuestionMark:
            return NFA::optional(operandNFA);
    }
    return nullptr;
}
```

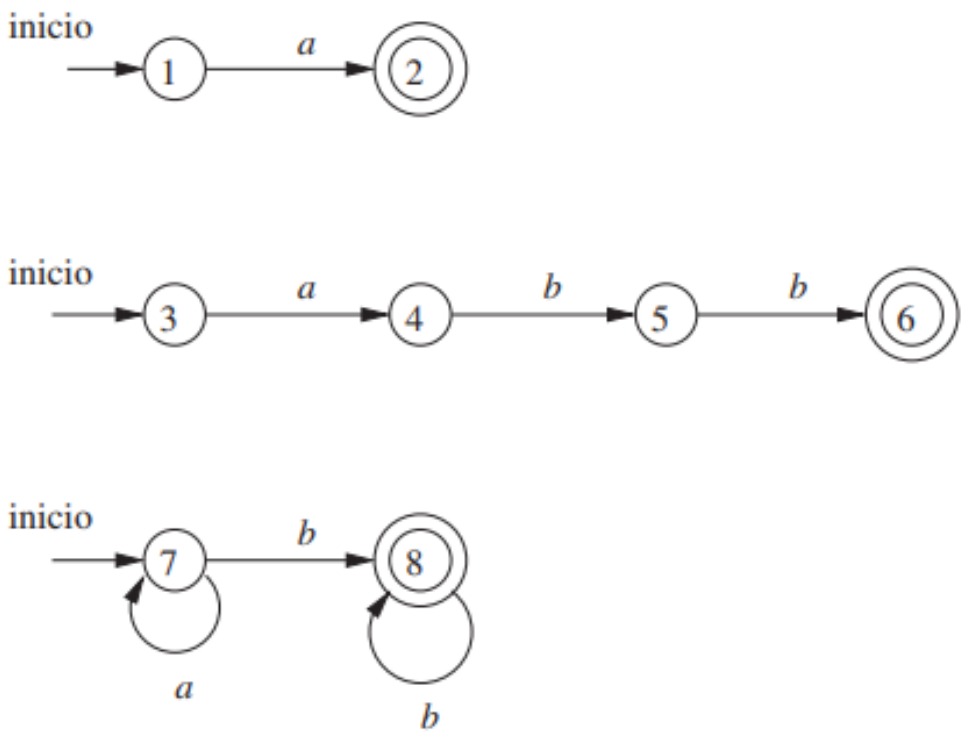

Figura 3.51: AFNs para **a**, **abb** y **a***b+

4. Construcción del Autómata Finito Determinista (DFA)

En esta sección describimos cómo combinamos los NFAs de los distintos patrones y aplicamos el algoritmo de subconjuntos para obtener un DFA optimizado.

4.1. Construcción de `LexerNFA`

La clase `LexerNFA` agrupa múltiples NFAs (uno por cada patrón de token) para construir un autómata único:

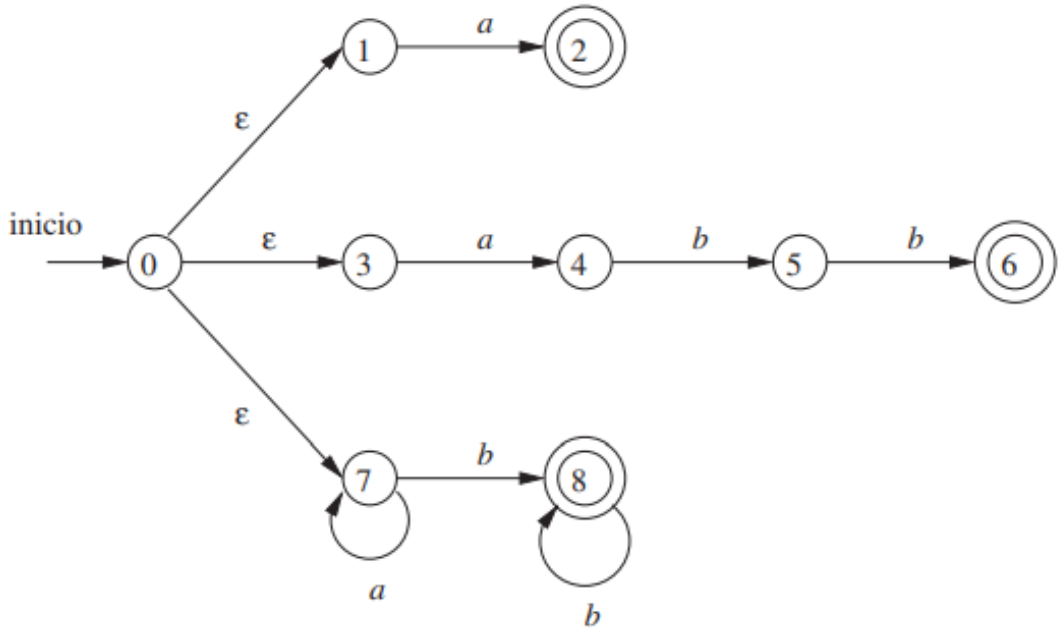

Figura 3.52: AFN combinado

Listing 7: Añadir patrones en `LexerNFA`

```
void LexerNFA::addPattern(const std::string& tokenName,
                          std::shared_ptr<NFA> nfa,
                          bool isIgnorable) {
    TokenPattern pattern;
    pattern.name = tokenName;
    pattern.nfa = nfa;
    pattern.isIgnorable= isIgnorable;
    patterns.push_back(pattern);
}

void LexerNFA::build() {
    // Combina todos los NFAs en un NFA global:
    //  Se crean transiciones epsilon desde un nuevo estado inicial
    // hacia cada NFA de patrn.
    //  Se conservan los estados finales etiquetados con tokenName.
}
```

4.2. Conversión de NFA a DFA

El método `LexerNFA::toDFA()` invoca el algoritmo de subconjuntos para convertir el NFA combinado en un DFA:

Listing 8: Conversión de NFA a DFA

```
std::shared_ptr<LexerDFA> LexerNFA::toDFA() {
```

```
    auto dfa = std::make_shared<LexerDFA>();
    dfa->build(*this);
    return dfa;
}

void LexerDFA::build(const LexerNFA& nfa) {
    // 1. Calcular la epsilon-clausura del conjunto inicial:
    auto initialSet = nfa.getCombinedNfa()->startStates();
    auto startClosure = nfa.getCombinedNfa()->epsilonClosure(initialSet);
    pendingStateSets.push(startClosure);
    visited.insert(startClosure);

    // 2. Mientras haya conjuntos pendientes:
    while (!pendingStateSets.empty()) {
        auto currentSet = pendingStateSets.front();
        pendingStateSets.pop();
        int dfaStateId = dfa.addState(currentSet,
                                      nfa.getCombinedNfa()->hasFinal(currentSet));

        // Para cada smbolo de entrada:
        for (char symbol : nfa.alphabet()) {
            auto nextSet = nfa.getCombinedNfa()
                              ->move(currentSet, symbol);
            auto closure = nfa.getCombinedNfa()
                               ->epsilonClosure(nextSet);
            if (!visited.count(closure)) {
                pendingStateSets.push(closure);
                visited.insert(closure);
            }
            dfa.addTransition(dfaStateId, symbol,
                              dfa.idOfState(closure));
        }
    }

    // 3. Minimizar el DFA resultante:
    minimizeDFA();
}
```

4.3. Algoritmo de Subconjuntos (Construcción de DFA)

La construcción de subconjuntos parte de la idea de que cada estado del autómata finito determinista (DFA) se corresponde con un conjunto de estados del autómata finito no determinista (NFA) original. Tras procesar una cadena de entrada $a_1 a_2 \cdots a_n$, el DFA se encuentra en el "estado–conjunto" que agrupa todas las configuraciones posibles en que podría hallarse el NFA, partiendo de su estado inicial y siguiendo en paralelo todas las transiciones etiquetadas con $a_1 a_2 \cdots a_n$.

Para definir este procedimiento, introducimos dos operaciones fundamentales sobre conjuntos de estados de $N = (Q, \Sigma, \delta, q_0, F)$:

- ϵ–cerradura(T): el conjunto de todos los estados alcanzables desde cualquier $s \in T$ mediante únicamente transiciones ϵ.

- mover(T, a): el conjunto de estados alcanzables desde algún $s \in T$ por medio de una transición etiquetada con el símbolo a.

El autómata determinista resultante $D = (Q_D, \Sigma, \delta_D, S_0, F_D)$ se construye de la siguiente manera:

1. El estado inicial de D es
$$S_0 \;=\; \epsilon\text{-cerradura}\big(\{q_0\}\big).$$
2. Iterativamente, mientras existan estados "sin procesar" en Q_D:
   - Para cada símbolo $a \in \Sigma$, calcular
   $$U \;=\; \epsilon\text{-cerradura}\big(\text{mover}(T, a)\big),$$
   donde T es el siguiente estado–conjunto sin procesar.
   - Si U no pertenece aún a Q_D, añadirlo como nuevo estado sin procesar.
   - Definir la transición $\delta_D(T, a) = U$.
3. El conjunto de estados de aceptación de D es
$$F_D \;=\; \big\{\, T \in Q_D \mid T \cap F \neq \varnothing \big\},$$
es decir, todos los conjuntos que contengan al menos un estado de aceptación de N.

Aunque teóricamente el número de subconjuntos puede crecer exponencialmente con respecto al número de estados de N, en la práctica se observa que para expresiones regulares empleadas en análisis léxico el tamaño del DFA resultante es manejable y comparable al del NFA original. Esto convierte al algoritmo de subconjuntos en una técnica eficaz y ampliamente utilizada en la generación de analizadores léxicos.‴

5. Minimización del DFA

Al construir un autómata finito determinista (DFA) para un analizador léxico, es habitual obtener máquinas que aceptan el mismo lenguaje pero difieren en el número o en los nombres de sus estados. Sin embargo, existe un único DFA (hasta renombrado de estados) que tiene el número mínimo de estados posible para un lenguaje dado. El objetivo de la minimización es transformar cualquier DFA en su versión «mínima», agrupando los estados que son indistinguibles desde el punto de vista del lenguaje que reconocen.

5.1. Equivalencia de estados

Dos estados s y t de un DFA se dicen *equivalentes* si, para toda cadena de entrada x, la lectura de x desde s conduce a un estado de aceptación si y sólo si la lectura de x desde t también lo hace. De manera intuitiva, no existe ninguna cadena capaz de «distinguir» a s de t, es decir, ninguna que lleve a uno de ellos a un estado de aceptación y al otro a un estado de no aceptación.

5.2. Partición inicial

El algoritmo de minimización comienza separando los estados en dos clases:

- *Estados de aceptación.*
- *Estados de no aceptación.*

Esta partición inicial refleja la distinción más básica: leer la cadena vacía ya diferencia ambos tipos.

5.3. Refinamiento iterativo

A partir de la partición inicial, se aplica un proceso de *refinamiento* que, para cada bloque de la partición y para cada símbolo de entrada a, divide ese bloque en subgrupos según el destino de a:

1. Sea G un bloque cualquiera de la partición actual.
2. Para cada estado $s \in G$, consideramos a qué bloque de la partición actual conduce s al leer el símbolo a.
3. Agrupamos en un subbloque todos los estados de G que tengan el mismo bloque destino bajo a.
4. Repetimos para cada símbolo $a \in \Sigma$ y para cada bloque G.

De este modo, dos estados permanecen en el mismo bloque si y sólo si, para *todos* los símbolos de entrada, transitan al mismo bloque de la partición previa. Si algún bloque se subdivide, se reemplaza por sus subbloques; y se repite el refinamiento hasta que *ningún* bloque pueda dividirse más.

5.4. Construcción del DFA mínimo

Cuando el refinamiento alcanza estabilidad (la partición no cambia), cada bloque de la partición final agrupa estados mutuamente equivalentes. Para construir el DFA mínimo:

- Se elige un representante por cada bloque, que será un estado en el nuevo DFA.
- El estado inicial es el representante del bloque que contenía el estado inicial original.
- Un representante es de aceptación si y sólo si su bloque contiene algún estado de aceptación.
- Para cada bloque G y símbolo a, si desde cualquier $s \in G$ el DFA original transita a un estado en el bloque H, entonces en el DFA mínimo se añade una transición desde el representante de G al representante de H con el símbolo a.

5.5. Justificación de corrección

- *Invariancia de la partición*: tras cada refinamiento, dos estados en el mismo bloque no pueden diferenciarse por ninguna cadena de longitud hasta la iteración actual.
- *Terminación*: dado que la partición original es finita y cada refinamiento es estricto, el proceso acaba en un número finito de pasos.
- *Minimalidad*: el DFA construido no puede tener menos estados, pues cada bloque final agrupa exactamente esos estados que son indistinguibles, y cualquier agrupación mayor violaría la definición de equivalencia.

De esta forma se garantiza que el DFA resultante reconozca el mismo lenguaje que el original y tenga el menor número de estados posible.‘‘‘

Listing 9: Minimización de DFA con Hopcroft

```
void LexerDFA::minimizeDFA() {
    // Particin inicial: estados de aceptacin vs. no aceptacin
    auto P = initialPartition();
    auto W = P; // conjuntos pendientes de refinamiento

    while (!W.empty()) {
        auto A = W.back(); W.pop_back();
        for (char c : alphabet) {
            // X = predecesores de A por c
            auto X = predecessorSet(A, c);
            for (auto& Y : P) {
                auto inter = intersection(X, Y);
                auto diff = difference(Y, X);
                if (!inter.empty() && !diff.empty()) {
                    // Refinar Y en inter y diff
                    replacePartition(P, Y, inter, diff);
                    // Actualizar conjuntos pendientes
                    updateWorkList(W, Y, inter, diff);
                }
            }
        }
    }
    // Reconstruir el DFA usando la particin P refinada
    rebuildFromPartition(P);
}
```

6. Utilización del Lexer

En esta sección vemos cómo se configura y utiliza el lexer generado para tokenizar una entrada de prueba.

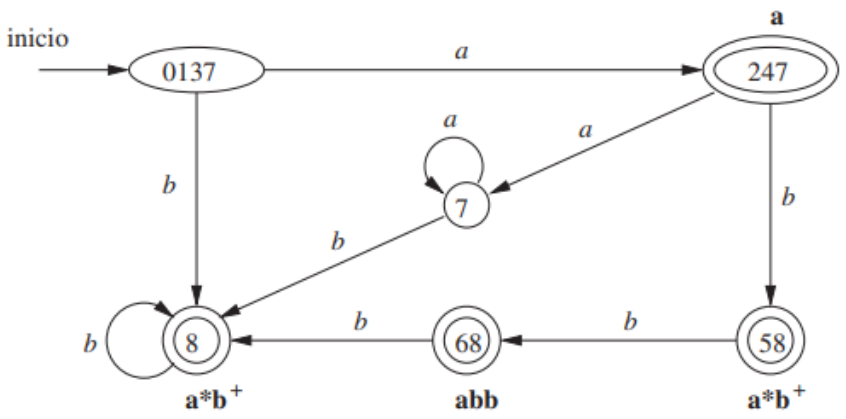

Figura 3.54: Grafo de transición para un AFD que maneja los patrones **a**, **abb** y **a*b+**

6.1. Configuración del Lexer

En `complete_lexer_test.cpp` se define una función auxiliar que, dado un conjunto de reglas (patrones y nombres de token), construye el lexer completo:

Listing 10: Función para crear el lexer en `complete`$_l$`exer`$_t$`est.cpp`

```
std::shared_ptr<LexerDFA> crearLexer(const std::vector<TokenRule>& rules) {
    RegexParserDriver driver;
    NfaBuild builder;
    LexerNFA lexerNfa;

    // Para cada regla, parsear la expresin y aadir el NFA al lexer
    for (const auto& rule : rules) {
        auto expr = driver.parse(rule.pattern);
        auto nfa = builder.buildFromExpression(expr);
        lexerNfa.addPattern(rule.name, nfa, rule.ignorable);
    }

    // Construir y devolver el DFA final
    return lexerNfa.toDFA();
}
```

6.2. Análisis Léxico

El método `scan` de `LexerDFA` recorre la cadena de entrada, aplicando en cada posición el algoritmo de la coincidencia más larga:

Listing 11: Implementación de `LexerDFA::scan`

```
std::vector<Lexeme> LexerDFA::scan(const std::string& input) {
    std::vector<Lexeme> tokens;
```

```
    size_t position = 0;

    while (position < input.length()) {
        // Buscar la coincidencia ms larga desde position
        Match match = findLongestMatch(input, position);

        if (match.length == 0) {
            // Error lxico: avanzar un carcter
            position++;
            continue;
        }

        // Avanzar la posicin
        position += match.length;

        // Si no es un token ignorable, agregarlo a la lista
        if (!match.isIgnorable) {
            tokens.push_back(Lexeme{
                match.tokenName,
                input.substr(position - match.length, match.length),
                position - match.length
            });
        }
    }
    return tokens;
}
```

6.3. Algoritmo de "Maximal Munch"

El principio de *maximal munch* (a veces llamado "longest match" o "mayor coincidencia") garantiza que, al procesar la entrada, el analizador léxico siempre elige el token más largo posible en cada paso. Teóricamente, el algoritmo se describe así:

1. **Inicio** Se sitúa el cursor en la posición actual de la cadena de entrada y se arma un puntero auxiliar para explorar hacia adelante.

2. **Exploración en el DFA** A partir del estado inicial del DFA, se consume carácter a carácter, transitando siempre que exista una transición válida para el símbolo leído.

3. **Registro de coincidencias finales** Cada vez que el DFA entra en un estado de aceptación (final), se guarda:

   - La posición alcanzada como *longitud de la coincidencia más larga* hasta el momento.
   - El tipo de token correspondiente a ese estado.
   - Si dicho token debe incluirse o ignorarse (por ejemplo, espacios o comentarios).

4. **Terminación de la exploración** La exploración continúa hasta que:

   - No existe transición definida para el siguiente carácter, o

- Se alcanza el final de la entrada.

En ese momento, el algoritmo retrocede (si es necesario) hasta la posición registrada de la última aceptación válida y retorna el token identificado allí.

5. **Reinicio para el siguiente token** Se avanza el cursor de la entrada a la posición donde terminó la *mayor coincidencia*, y se repite el proceso desde el estado inicial para extraer el siguiente token.

‘‘‘

Listing 12: Implementación de `LexerDFA::findLongestMatch`

```
Match LexerDFA::findLongestMatch(const std::string& input, size_t startPos) {
    int currentState = startState;
    size_t currentPos = startPos;
    size_t longestMatchLength = 0;
    std::string longestMatchToken;
    bool longestMatchIsIgnorable = false;

    while (currentPos < input.length()) {
        char c = input[currentPos];
        if (!dfa.hasTransition(currentState, c)) break;
        currentState = dfa.nextState(currentState, c);
        currentPos++;
        if (dfa.isAccepting(currentState)) {
            longestMatchLength = currentPos - startPos;
            longestMatchToken = dfa.tokenName(currentState);
            longestMatchIsIgnorable = dfa.isIgnorable(currentState);
        }
    }

    return Match{
        longestMatchToken,
        longestMatchLength,
        longestMatchIsIgnorable
    };
}
```

7. Ejemplo de Flujo Completo

A continuación se muestra un ejemplo de uso que ilustra todo el flujo: desde la definición de las reglas hasta la tokenización de una cadena de entrada.

Listing 13: Ejemplo en `complete_lexer_test.cpp`

```
#include "lexer_dfa.hpp"
#include "regex_parser_driver.hpp"
#include "build_nfa.hpp"
#include "token_rule.hpp"

int main() {
```

```
    // Definicin de reglas: nombre, patrn y si es ignorable
    std::vector<TokenRule> rules = {
        TokenRule("NUMBER", "[0-9]+"),
        TokenRule("IDENTIFIER", "(_|[a-zA-Z])(_|[a-z0-9A-Z])*"),
        TokenRule("WHITESPACE", "[ \\t\\n\\r]+", true)
    };

    // Construir el lexer
    auto lexer = crearLexer(rules);

    // Cadena de entrada a tokenizar
    std::string input = "x123 456";

    // Escaneo lxico
    auto tokens = lexer->scan(input);

    // Imprimir resultados
    for (const auto& tok : tokens) {
        std::cout << tok.name
                  << " ('" << tok.lexeme << "')"
                  << " en posicin " << tok.position
                  << std::endl;
    }

    return 0;
}
```