

# 3. Herramientas para el análisis lexicográfico

- Definición y conceptos
- Expresiones regulares
- Autómatas finitos
- Patrones
- Java Regex

# Definición y conceptos

- La palabra léxico en el sentido tradicional significa “referente a las palabras”. En términos de lenguajes de programación, las palabras son objetos como nombres de variables, números, palabras reservadas, etc. A estas palabras se les llama ***tokens***.
- Un analizador léxico o lexer, toma una cadena de letras y la divide en tokens. Además, el lexer filtra lo que separa los tokens (espacios, cambio de línea, etc.).
- El propósito principal de un lexer es facilitar la vida a la siguiente fase de la compilación, el análisis sintáctico.
- En teoría, el trabajo que realiza el lexer, podría ser hecho durante el análisis sintáctico, sin embargo, las fases se mantienen separadas por tradición, eficiencia y modularidad.

# Expresiones regulares

- Las expresiones regulares son una notación algebraica que nos permite describir conjuntos de cadenas sobre un alfabeto (a lo que llamamos un lenguaje).

# Expresiones regulares

| Expresión regular            | Lenguaje   | Descripción   |
|------------------------------|--|---|
| <b>a</b>                     | $\{“a”\}$  | El conjunto consistente en la letra “a”.  |
| <b><math>\epsilon</math></b> | $\{“”\}$   | El conjunto que contiene la cadena vacía.   |
| <b>s   t</b>                 | $L(s) \cup L(t)$                                   | Cadenas de ambos lenguajes.   |
| <b>st</b>                    | $\{vw \mid v \in L(s), w \in L(t)\}$               | Cadenas construidas al concatenar una cadena del primer lenguaje y una cadena del segundo lenguaje. |
| <b>s*</b>                    | $\{“”\} \cup \{vw \mid v \in L(s), w \in L(s^*)\}$ | Cada cadena en el lenguaje es una concatenación de cualquier número de cadenas                      |

# Expresiones regulares - Notación corta

- Si quisiéramos describir números enteros no negativos, podríamos utilizar la siguiente expresión regular: **(011213141516171819) (011213141516171819)\***
- La cantidad de dígitos involucrados en la expresión pueden hacerla confusa y se pone peor cuando debemos utilizar todo el alfabeto para describir los nombres de las variables.
- Debido a esto, se introduce una notación corta, los corchetes cuadrados [ ].
- Por ejemplo:
  - **(011213141516171819)** se puede describir como **[0-9]**.
  - **[a-zA-Z]** describe cualquier letra del alfabeto en minúsculas y mayúsculas.
  - **[0-9][0-9]\*** describe cualquier número entero no negativo.
- Cuando queremos indicar uno o ninguna ocurrencia utilizamos **?**.
- Cuando queremos indicar una o más ocurrencias utilizamos **+**.

# Expresiones regulares - Ejemplos

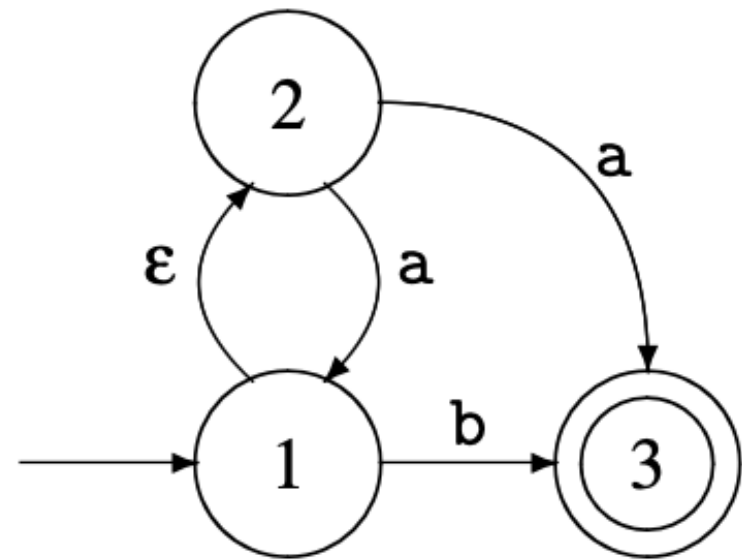
- Palabras reservadas (keywords): Una palabra reservada como **if** se describe con una expresión regular exactamente igual.
- Nombres de variables: **[a-zA-Z][a-zA-Z\_0-9]\***
- Enteros: **[+-]?[0-9]+**
- Flotantes: **[+-](((([0-9]+.[0-9]\*|.[0-9]+)([eE][+-]?[0-9]+)?)|[[0-9]+[eE][+-]?[0-9]+))**
- Cadenas de caracteres: **"([a-zA-Z0-9]|[a-zA-Z])"**

# Autómatas finitos

- Un autómata finito es, en el sentido abstracto, una máquina que tiene un número finito de estados y un número finito de transiciones entre estos.
- Una transición entre estados se etiqueta con un carácter del alfabeto de entrada, pero también se puede utilizar el símbolo  $\epsilon$  para indicar transiciones vacías.
- Un autómata finito se puede utilizar para decidir si una cadena de entrada es miembro de un conjunto de cadenas particulares.
- Seleccionamos uno de los estados como estado inicial y en cada paso hacemos uno de los siguientes pasos:
  - Seguimos una transición vacía a otro estado
  - Leemos un carácter de la entrada y seguimos la transición etiquetada por ese carácter.
- Cuando todos los caracteres de la entrada han sido leídos, vemos si el estado actual está marcado como de aceptación. Si es así, la cadena leída de la entrada está en el lenguaje definido por el autómata.

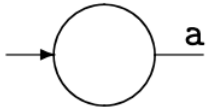
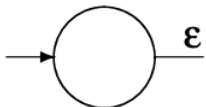
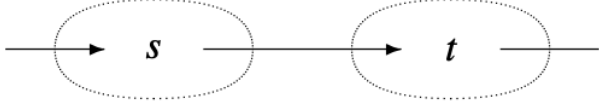
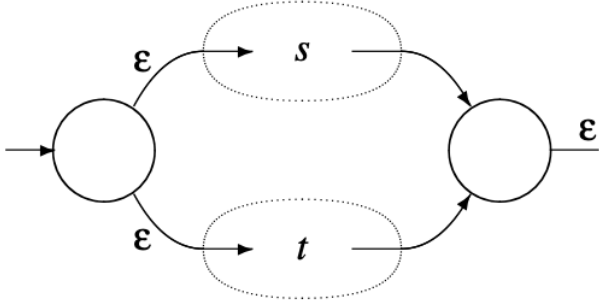
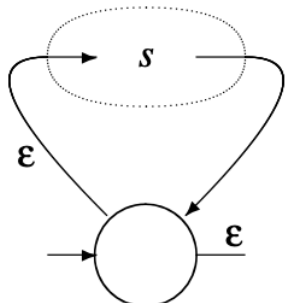
# Autómatas finitos - Ejemplo

| from | to | by         |
|------|----|------------|
| 1    | 2  | $\epsilon$ |
| 2    | 1  | a          |
| 1    | 2  | $\epsilon$ |
| 2    | 1  | a          |
| 1    | 3  | b          |

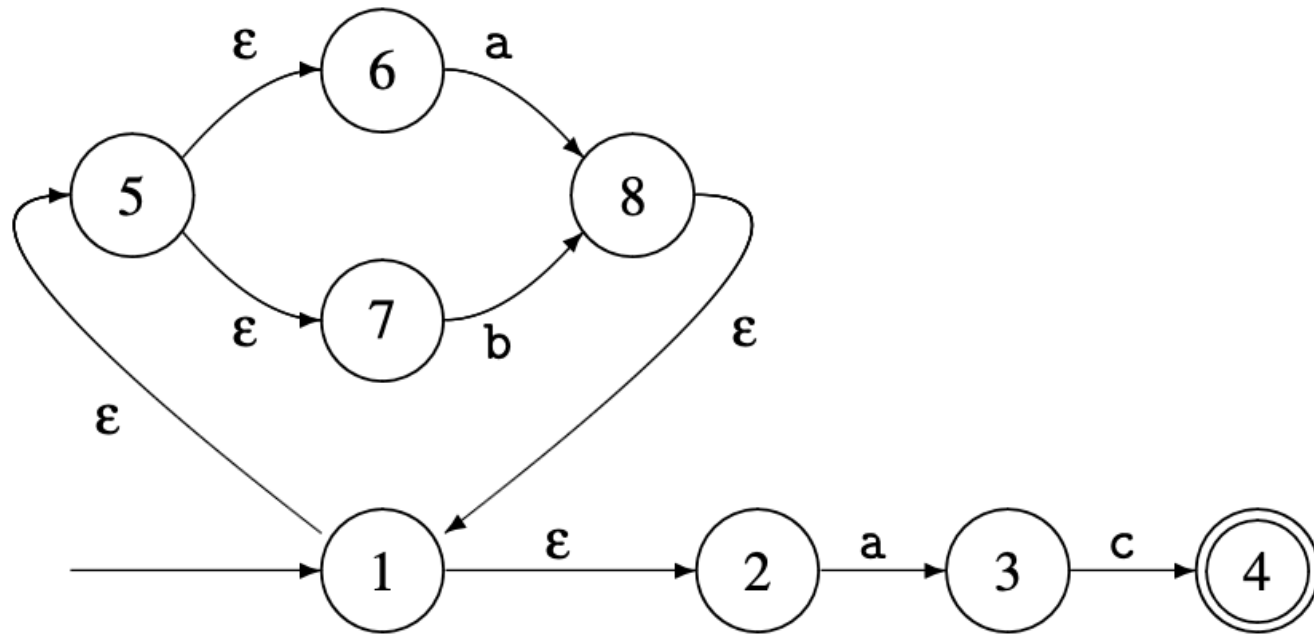




# Autómatas finitos - Conversión de expresión regular en autómata finito

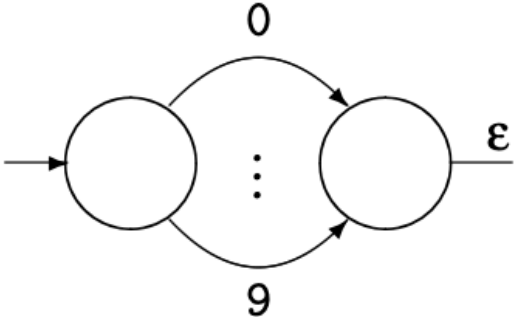
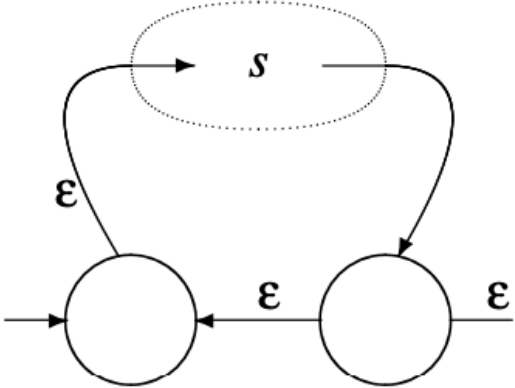
| Regular expression | NFA fragment  |
|--------------------|---|
| $a$                |    |
| $\epsilon$         |    |
| $st$               |    |
| $s t$              |   |
| $s^*$              |  |

# Autómatas finitos - Ejemplo

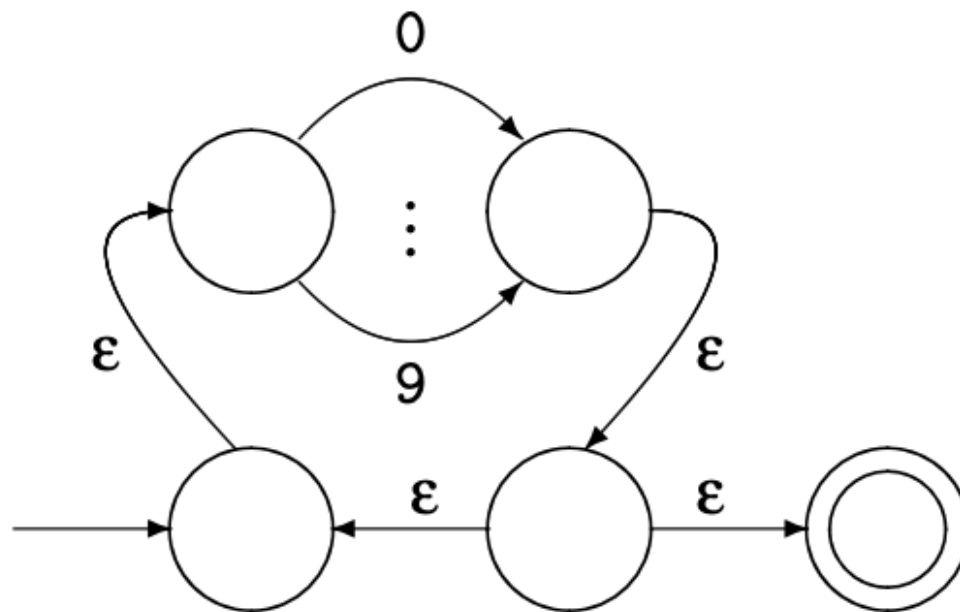


$(a|b)^*ac$

# Autómatas finitos - Conversión de expresión regular en autómata finito

| Regular expression | NFA fragment  |
|--------------------|---|
| $\epsilon$         | —   |
| $[0-9]$            |  <p>The diagram shows a Non-deterministic Finite Automaton (NFA) fragment for the regular expression <math>[0-9]</math>. It consists of two states: a start state on the left and an end state on the right. A directed edge connects the start state to the end state, labeled with the characters '0', a vertical ellipsis '...', and '9', representing the range of digits. The end state has a self-loop labeled with the empty string <math>\epsilon</math>.</p>  |
| $s^+$              |  <p>The diagram shows an NFA fragment for the regular expression <math>s^+</math>. It features three states: a start state on the left, a middle state labeled 's' enclosed in a dashed oval, and an end state on the right. Transitions are as follows: an arrow from the start state to the 's' state labeled with <math>\epsilon</math>; an arrow from the 's' state to the end state labeled with <math>s</math>; and an arrow from the end state back to the start state labeled with <math>\epsilon</math>. The end state also has a self-loop labeled with <math>\epsilon</math>.</p> |

# Autómatas finitos - Ejemplo

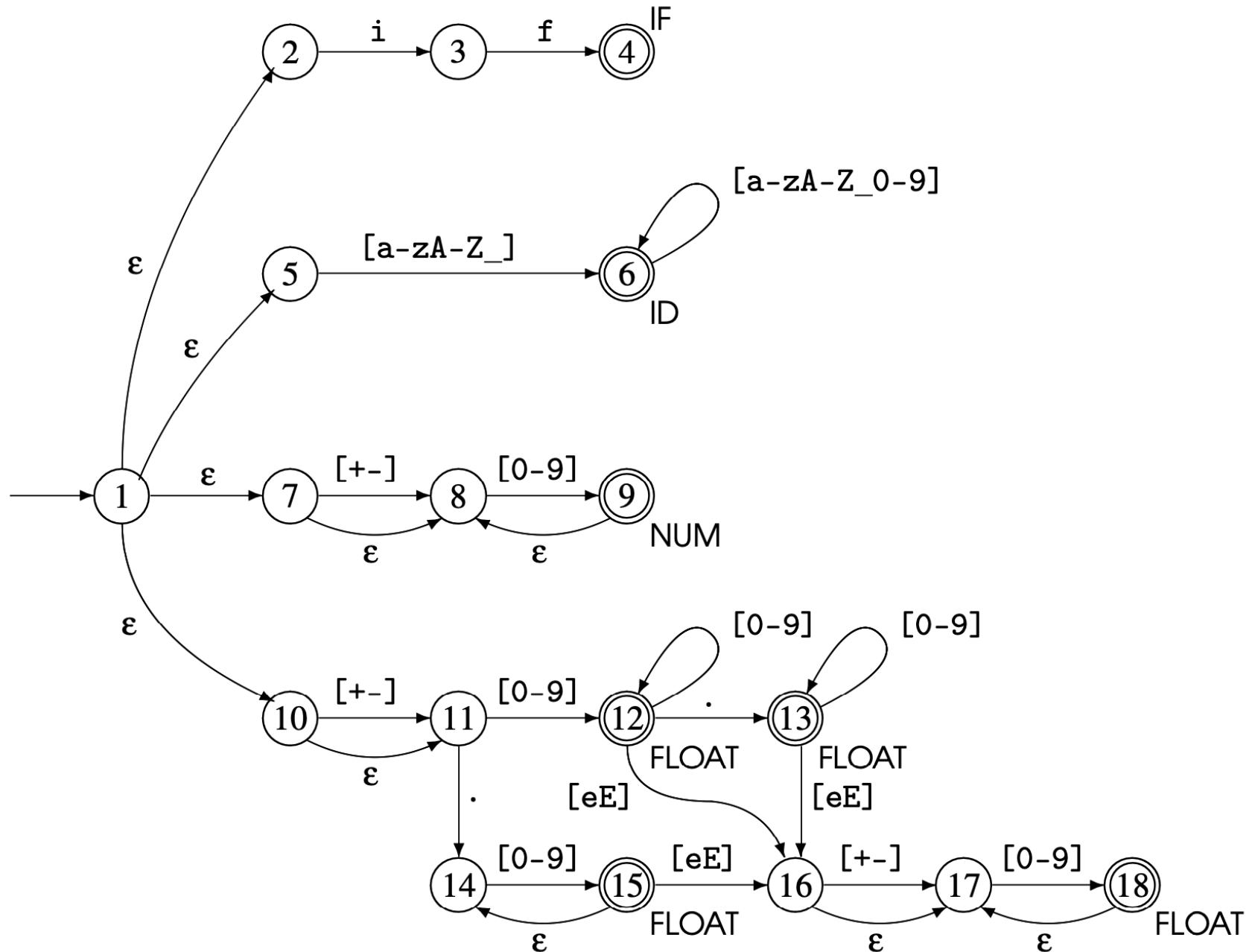


$[0-9]^+$

# Autómatas finitos - Construcción

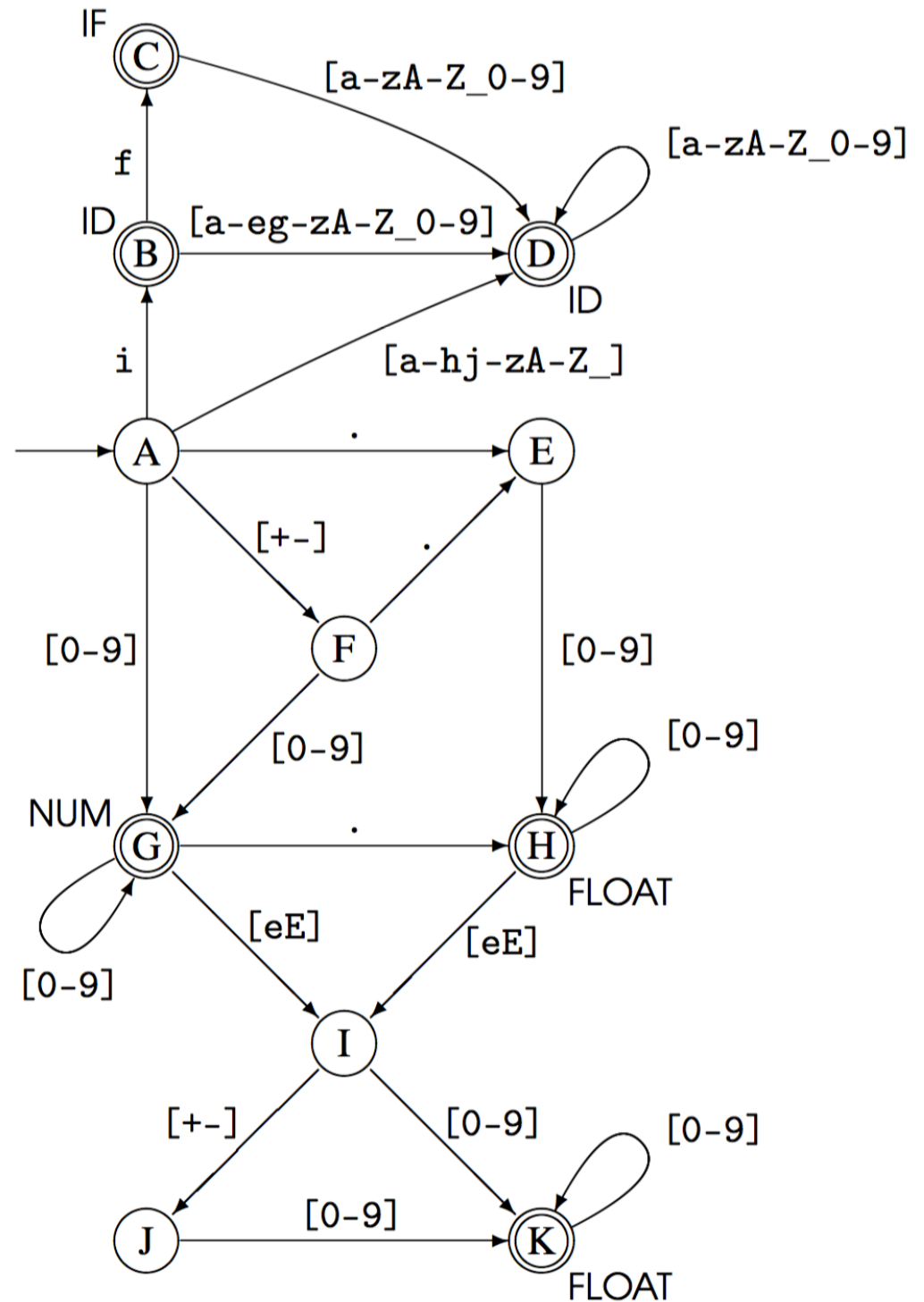
- Ya sabemos como convertir una expresión regular en un autómata finito. Lo que queremos ahora es algo más: un programa que haga análisis léxico.
- Para hacer esto podemos considerar los siguientes pasos:
  1. Construir un autómata finito para cada expresión regular.
  2. Marcar los estados de aceptación con el nombre del token que acepta.
  3. Combinar los autómatas finitos en uno nuevo.
  4. Convertir el nuevo autómata finito en uno que sea determinístico.

# Autómatas finitos - Construcción



# Autómatas finitos

## - Construcción



# Patrones - Un lexer recursivo descendente

## Propósito.

- Un lexer obtiene un flujo de tokens (unidades léxicas) a partir de un flujo de caracteres, reconociendo patrones léxicos.
- Los **lexers** también son llamados **scanners**, **analizadores léxicos** y **tokenizers**.
- Cada token tiene dos atributos principales: **tipo** y **texto**.



# Patrones - Un lexer recursivo descendente

## Discusión.

- El objetivo de un **lexer** es emitir una secuencia de tokens. Cada token tiene dos atributos principales: un **tipo** de token (categoría de símbolo) y el **texto** asociado con él. Se dice que todas las palabras dentro de una categoría en particular tienen el mismo tipo de token, aunque su texto asociado es diferente.
- Para construir un lexer a mano, escribimos un método para cada definición de token (regla léxica). Es decir, si tenemos una definición de token **T**, debe existir un método **T()**. Estos métodos reconocen el patrón expresado en la regla léxica asociada.
- Para hacer que el lexer parezca una enumeración de tokens, es útil definir un método llamado **nextToken()**. Este método utiliza el carácter apuntado por el cursor para enlutar el flujo de control al método de reconocimiento apropiado.

# Patrones - Un lexer recursivo descendente

- Se presenta la base de un método **nextToken()** típico que salta espacios y comentarios.

```
public Token nextToken() {
    while ( «lookahead-char»!=EOF ) { // EOF==-1 per java.io
        if ( «comment-start-sequence» ) { COMMENT(); continue; }
        ... // other skip tokens
        switch ( «lookahead-char» ) { // which token approaches?
            case «whitespace» : { consume(); continue; } // skip
            case «chars-predicting-T1» : return T1(); // match T1
            case «chars-predicting-T2» : return T2();
            ...
            case «chars-predicting-Tn» : return Tn();
            default : «error»
        }
    }
    return «EOF-token»; // return token with EOF_TYPE token type
}
```

# Patrones - Un lexer recursivo descendente

- Para utilizar este patrón, se crea una instancia de un lexer a partir de una cadena de entrada o lector de flujo. El parser utiliza este objeto lexer para obtener los tokens a través del método **nextToken()**.

```
MyLexer lexer = new MyLexer("«input-sentence»"); // create lexer
MyParser parser = new MyParser(lexer); // create parser
parser.«start_rule»(); // begin parsing, looking for a list sentence
```

# Patrones - Un lexer recursivo descendente

## Implementación.

- Como ejemplo de implementación, se construirá un lexer para un lenguaje de listas anidadas. Nuestro objetivo es un lexer que podamos tratar como una enumeración.

```
ListLexer lexer = new ListLexer(args[0]);  
Token t = lexer.nextToken();  
while ( t.type != Lexer.EOF_TYPE ) {  
    System.out.println(t);  
    t = lexer.nextToken();  
}  
System.out.println(t); // EOF
```

```
$ java Test '[a, b ]'  
<'[,LBRACK>  
<'a',NAME>  
<',' ,COMMA>  
<'b',NAME>  
<']',RBRACK>  
<'<EOF>',<EOF>>  
$
```

# Patrones - Un lexer recursivo descendente

- Para la implementación, necesitaremos objetos Token, un lexer abstracto y un lexer concreto.

```
public class Token {  
    public int type;  
    public String text;  
    public Token(int type, String text) {this.type=type; this.text=text;}  
    public String toString() {  
        String tname = ListLexer.tokenNames[type];  
        return "<'"+text+"', "+tname+">";  
    }  
}
```

# Patrones - Un lexer recursivo descendente

- El lexer concreto, necesita definir tipos de tokens.

```
public class ListLexer extends Lexer {  
    public static int NAME = 2;  
    public static int COMMA = 3;  
    public static int LBRACK = 4;  
    public static int RBRACK = 5;  
    public static String[] tokenNames =  
        { "n/a", "<EOF>", "NAME", "COMMA", "LBRACK", "RBRACK" };  
    public String getTokenName(int x) { return tokenNames[x]; }  
  
    public ListLexer(String input) { super(input); }  
    boolean isLETTER() { return c>='a'&&c<='z' || c>='A'&&c<='Z'; }
```

# Patrones - Un lexer recursivo descendente

- El método nextToken().

```
public Token nextToken() {  
    while ( c!=EOF ) {  
        switch ( c ) {  
            case ' ': case '\t': case '\n': case '\r': WS(); continue;  
            case ',': consume(); return new Token(COMMA, ",");  
            case '[': consume(); return new Token(LBRACK, "[" );  
            case ']': consume(); return new Token(RBRACK, "]" );  
            default:  
                if ( isLETTER() ) return NAME();  
                throw new Error("invalid character: "+c);  
        }  
    }  
    return new Token(EOF_TYPE, "<EOF>");  
}
```

# Patrones - Un lexer recursivo descendente

- Los métodos NAME() y WS().

```
/** NAME : ('a'..'z' | 'A'..'Z')+; // NAME is sequence of >=1 letter */
Token NAME() {
    StringBuilder buf = new StringBuilder();
    do { buf.append(c); consume(); } while ( isLETTER() );
    return new Token(NAME, buf.toString());
}
```

```
/** WS : (' ' | '\t' | '\n' | '\r')* ; // ignore any whitespace */
void WS() {
    while ( c==' ' || c=='\t' || c=='\n' || c=='\r' ) consume();
}
```



```

public abstract class Lexer {
    public static final char EOF = (char)-1; // represent end of file char
    public static final int EOF_TYPE = 1;    // represent EOF token type
    String input; // input string
    int p = 0;    // index into input of current character
    char c;       // current character

    public Lexer(String input) {
        this.input = input;
        c = input.charAt(p); // prime lookahead
    }

    /** Move one character; detect "end of file" */
    public void consume() {
        p++;
        if ( p >= input.length() ) c = EOF;
        else c = input.charAt(p);
    }

    /** Ensure x is next character on the input stream */
    public void match(char x) {
        if ( c == x ) consume();
        else throw new Error("expecting "+x+"; found "+c);
    }

    public abstract Token nextToken();
    public abstract String getTokenName(int tokenType);
}

```

- El lexer abstracto.