**Debe tener al menos: reconocedor para números enteros, números reales e identificadores, para los cuales se deberán presentar:**

1. **Expresión regular**

**Expresiones Regulares**

* **Números enteros**: ^[+-]?\d+$
* **Números reales**: ^[+-]?\d\*\.\d+([eE][+-]?\d+)?$
* **Identificadores**: ^[a-zA-Z\_][a-zA-Z0-9\_]\*$

#### Números Enteros

* **AFN-Ɛ**:

Estados: { q0, q1, q2 }

Alfabeto: { +, -, 0-9 }

Transiciones:

q0 -> q1 [+, -]

q1 -> q2 [0-9]

q2 -> q2 [0-9]

Estado inicial: q0

Estado(s) final(es): { q2 }

* **AFD**:

Estados: { q0, q1, q2 }

Alfabeto: { +, -, 0-9 }

Transiciones:

q0 -> q1 [+, -]

q0 -> q2 [0-9]

q1 -> q2 [0-9]

q2 -> q2 [0-9]

Estado inicial: q0

Estado(s) final(es): { q2 }

* **AFD Mínimo**:

Estados: { q0, q1 }

Alfabeto: { +, -, 0-9 }

Transiciones:

q0 -> q1 [+, -]

q0 -> q1 [0-9]

q1 -> q1 [0-9]

Estado inicial: q0

Estado(s) final(es): { q1 }

#### Números Reales

* **AFN-Ɛ**:

Estados: { q0, q1, q2, q3, q4, q5 }

Alfabeto: { +, -, ., 0-9, e, E }

Transiciones:

q0 -> q1 [+, -]

q0 -> q2 [0-9]

q2 -> q2 [0-9]

q2 -> q3 [.]

q3 -> q4 [0-9]

q4 -> q4 [0-9]

q4 -> q5 [e, E]

q5 -> q1 [+, -]

q5 -> q4 [0-9]

Estado inicial: q0

Estado(s) final(es): { q4 }

* **AFD**:

Estados: { q0, q1, q2, q3, q4, q5 }

Alfabeto: { +, -, ., 0-9, e, E }

Transiciones:

q0 -> q1 [+, -]

q0 -> q2 [0-9]

q2 -> q2 [0-9]

q2 -> q3 [.]

q3 -> q4 [0-9]

q4 -> q4 [0-9]

q4 -> q5 [e, E]

q5 -> q1 [+, -]

q5 -> q4 [0-9]

Estado inicial: q0

Estado(s) final(es): { q4 }

* **AFD Mínimo**:

Estados: { q0, q1, q2, q3, q4, q5 }

Alfabeto: { +, -, ., 0-9, e, E }

Transiciones:

q0 -> q1 [+, -]

q0 -> q2 [0-9]

q2 -> q2 [0-9]

q2 -> q3 [.]

q3 -> q4 [0-9]

q4 -> q4 [0-9]

q4 -> q5 [e, E]

q5 -> q1 [+, -]

q5 -> q4 [0-9]

Estado inicial: q0

Estado(s) final(es): { q4 }

#### Identificadores

* **AFN-Ɛ**:

Estados: { q0, q1, q2 }

Alfabeto: { a-z, A-Z, \_, 0-9 }

Transiciones:

q0 -> q1 [a-z, A-Z, \_]

q1 -> q1 [a-z, A-Z, \_, 0-9]

Estado inicial: q0

Estado(s) final(es): { q1 }

* **AFD**:

Estados: { q0, q1 }

Alfabeto: { a-z, A-Z, \_, 0-9 }

Transiciones:

q0 -> q1 [a-z, A-Z, \_]

q1 -> q1 [a-z, A-Z, \_, 0-9]

Estado inicial: q0

Estado(s) final(es): { q1 }

* **AFD Mínimo**:

Estados: { q0, q1 }

Alfabeto: { a-z, A-Z, \_, 0-9 }

Transiciones:

q0 -> q1 [a-z, A-Z, \_]

q1 -> q1 [a-z, A-Z, \_, 0-9]

Estado inicial: q0

Estado(s) final(es): { q1 }

**5. Función reconocedora implementado en un lenguaje de programación a elección a partir de la tabla de transición del AFD mínimo**

**Resumen del Analizador Léxico**

**1. Objetivo del Proyecto**

El objetivo del proyecto es crear un analizador léxico para el lenguaje de programación Pascal. El analizador debe ser capaz de identificar varios componentes léxicos tales como operadores aritméticos, operadores relacionales, palabras reservadas, signos de puntuación, tipos de datos, identificadores, números reales y asignaciones.

**2. Estructura del Proyecto**

El proyecto está organizado de la siguiente manera:

* **main.js**: El archivo principal que controla la lectura del archivo de entrada y coordina el análisis léxico.
* **analizador.js**: Contiene las funciones que verifican si un token es un operador, palabra reservada, signo de puntuación o tipo de dato.
* **automatas/**: Contiene los autómatas que reconocen las asignaciones, identificadores y números reales.
  + **automata\_asignacion.js**
  + **automata\_identificadores.js**
  + **automata\_reales.js**
* **pascal.txt**: El archivo de entrada que contiene el código Pascal a ser analizado.

**3. Componentes del Proyecto**

**3.1. main.js**

Este archivo se encarga de:

* Leer el contenido del archivo pascal.txt.
* Dividir el contenido en tokens.
* Pasar cada token a la función analizar para determinar su tipo léxico.

###### main.js

import fs from 'fs';

import { esOperadorAritmetico, esOperadorRelacional, esPalabraReservada, esPuntuacion, esTipoDato } from './analizador.js';

import { automata\_asignacion } from './automatas/automata\_asignacion.js';

import { automata\_identificadores } from './automatas/automata\_identificadores.js';

import { automata\_reales } from './automatas/automata\_reales.js';

// Leer el archivo pascal.txt

fs.readFile("pascal.txt", "utf8", (err, data) => {

if (err) {

console.error(err);

return;

}

const text = data;

const text\_separado = text.split(/[\s\r\n]+/);

const array = text\_separado.filter(a => a !== '');

analizar(array);

});

function analizar(contenido) {

for (let i = 0; i < contenido.length; i++) {

const token = contenido[i];

if (esPalabraReservada(token)) {

console.log(token, " --> Es una palabra reservada");

} else if (esTipoDato(token)) {

console.log(token, " --> Es un tipo de dato");

} else if (esOperadorAritmetico(token)) {

console.log(token, " --> Es un Operador Aritmético");

} else if (esOperadorRelacional(token)) {

console.log(token, " --> Es un Operador Relacional");

} else if (esPuntuacion(token)) {

console.log(token, " --> Es un signo de puntuación");

} else if (token === ")" || token === "(") {

console.log(token, " --> Es un paréntesis");

} else if (automata\_asignacion(token)) {

console.log(token, " --> Es una asignación");

} else if (automata\_identificadores(token)) {

console.log(token, " --> Es un identificador");

} else if (automata\_reales(token)) {

console.log(token, " --> Es un número real");

} else {

console.log("ERROR LÉXICO ----->", token);

}

}

}

##### 3.2. ****analizador.js****

Contiene funciones que verifican si un token pertenece a un conjunto específico, como operadores, palabras reservadas, etc.

###### analizador.js

function esTokenValido(token, conjunto, convertirMayuscula = false) {

if (convertirMayuscula) {

token = token.toUpperCase();

}

return conjunto.has(token);

}

const operadoresAritmeticos = new Set(['+', '-', '/', '\*']);

const operadoresRelacionales = new Set(['>=', '<=', '<>', '>', '<', '=']);

const palabrasReservadas = new Set([

'FUNCTION', 'PROCEDURE', 'PROGRAM', 'WHILE', 'IF', 'ELSE', 'THEN', 'TO', 'DO', 'TRUE', 'FALSE',

'AND', 'OR', 'NOT', 'AS', 'ARRAY', 'BEGIN', 'END', 'BREAK', 'CASE', 'CONTINUE', 'DEFAULT', 'DIV',

'EXIT', 'FILE', 'FINALLY', 'FOR', 'GOTO', 'MOD', 'OF', 'READ', 'WRITE', 'RECORD', 'REPEAT', 'SELF',

'SET', 'STRING', 'TRY', 'TYPE', 'UNIT', 'USES', 'VAR', 'WITH', 'WRITELN'

]);

const simbolosPuntuacion = new Set(['.', ':', ';', ',']);

const tiposDatos = new Set(['BOOLEAN', 'CHAR', 'STRING', 'WORD', 'INTEGER', 'BYTE', 'REAL']);

export function esOperadorAritmetico(token) {

return esTokenValido(token, operadoresAritmeticos);

}

export function esOperadorRelacional(token) {

return esTokenValido(token, operadoresRelacionales);

}

export function esPalabraReservada(token) {

return esTokenValido(token, palabrasReservadas, true);

}

export function esPuntuacion(token) {

return esTokenValido(token, simbolosPuntuacion);

}

export function esTipoDato(token) {

return esTokenValido(token, tiposDatos, true);

}

##### 3.3. ****Autómatas****

Estos archivos contienen funciones que utilizan expresiones regulares para verificar si un token es una asignación, identificador o número real.

###### automata\_asignacion.js

export function automata\_asignacion(token) {

const asignacionRegex = /^:=$/;

return asignacionRegex.test(token);

}

###### automata\_identificadores.js

export function automata\_identificadores(token) {

const identificadorRegex = /^[a-zA-Z\_][a-zA-Z0-9\_]\*$/;

return identificadorRegex.test(token);

}

###### automata\_reales.js

export function automata\_reales(token) {

const realRegex = /^[+-]?([0-9]\*[.])?[0-9]+$/;

return realRegex.test(token);

}

### Explicación del Proceso de Desarrollo

1. **Identificación de Componentes Léxicos**: Primero, identificamos los diferentes componentes léxicos de Pascal que el analizador debe reconocer, como operadores, palabras reservadas, etc.
2. **Creación de Funciones de Verificación**: Para cada tipo de componente léxico, creamos funciones que verifican si un token pertenece a ese tipo. Estas funciones utilizan conjuntos para una búsqueda rápida y eficiente.
3. **Implementación de Autómatas**: Para identificadores, asignaciones y números reales, implementamos autómatas que utilizan expresiones regulares para la verificación.
4. **Lectura y Análisis del Archivo**: En main.js, leemos el archivo de entrada pascal.txt, lo dividimos en tokens y utilizamos las funciones y autómatas para analizar cada token.

**Cómo Ejecutar el Analizador Léxico**

1. **Instalar Node.js**: Si no lo tienes instalado, descárgalo e instálalo desde [nodejs.org](https://nodejs.org/).
2. **Inicializar el Proyecto**: En el directorio del proyecto, ejecuta npm init -y para crear un package.json.
3. **Instalar Dependencias de Babel**: Para usar import/export, instala Babel:

npm install --save-dev @babel/core @babel/cli @babel/preset-env @babel/node

1. **Configurar Babel**: Crea un archivo .babelrc con el siguiente contenido:

{

"presets": ["@babel/preset-env"]

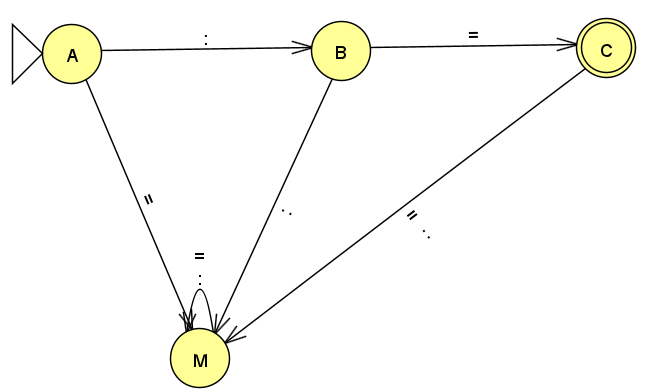
}

1. **Ejecutar el Analizador**: Utiliza el siguiente comando para ejecutar main.js:

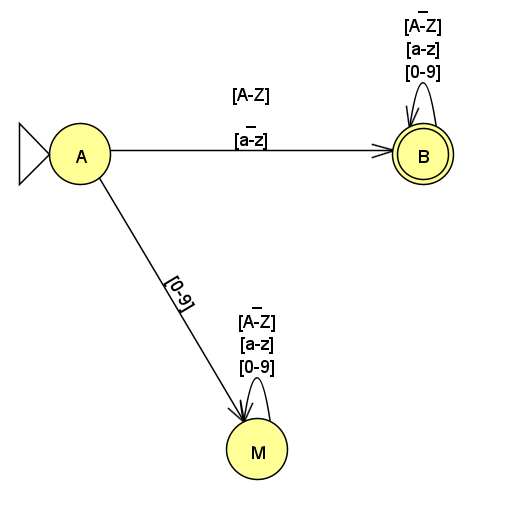
npx babel-node main.js

Este comando leerá el archivo pascal.txt, analizará el contenido y mostrará los resultados en la consola.

Automata asignación



Automata identificadores



Automata Real

