subconjuntos.ts

- ## subconjuntos.ts

- §

```
import { Automata } from "../types/automata";
import { Nodo } from "../types/automata";
import { EPSILON } from "./thompson";
```

- §

Función que calcula la cerradura epsilon de un estado.

```
function epsilonCerradura(S: Nodo): Set<Nodo> {
  const cerradura = [S];
  let i = 0;
  while (i < cerradura.length) {
```

- §

Consigue los estados adyacentes alcanzables a través de una arista epsilon.

```
    const vecinosConVacio = cerradura[i].adyacentes[EPSILON] || [];
    for (let nodo of vecinosConVacio) {
```

- §

Agregar los estados que faltan para procesar en una siguiente iteración.

```
      if (!estaNodoEnLista(cerradura, nodo)) {
        cerradura.push(nodo);
      }
    }
    i++;
  }
  return new Set<Nodo>(cerradura);
}
```

- §

Función que calcula la cerradura epsilon a partir de un conjunto de estados
T

```
function epsilonCerraduraT(T: Set<Nodo>): Set<Nodo> {
  let cerradura: Set<Nodo> = new Set<Nodo>();
  T.forEach((estado) => {
```

- §

Calcula la cerradura epsilon de cada estado del conjunto T.

```
    const epsilon = epsilonCerradura(estado);
```

- §

Realiza la unión de la cerradura calculada con el resultado parcial de las cerraduras.

```
    cerradura = unionConjuntos(cerradura, epsilon);
  });
  return cerradura;
}
```

- §

Función que calcula el resultado de aplicar la operación Mover(T,a) donde T es un conjunto de estados y a es un simbolo del alfabeto.

```
function mover(T: Set<Nodo>, a: string): Set<Nodo> {
  const resultado = new Set<Nodo>();
  T.forEach((estado) => {
```

- §

Consigue los estados adyacentes con una arista con la etiqueta a.

```
    let adyacentes = estado.adyacentes[a] || [];
    for (let adyacente of adyacentes) {
```

- §

Agrega los estados adyacentes al resultado de la función.

```
      resultado.add(adyacente);
    }
  });
  return resultado;
}
```

- §

Función utilitaria que calcula la unión de conjuntos

```
const unionConjuntos = (
  conjuntoA: Set<Nodo>,
  conjuntoB: Set<Nodo>
): Set<Nodo> => {
  const resultado = new Set<Nodo>();
```

```
  conjuntoA.forEach((item) => {
    resultado.add(item);
  });
  conjuntoB.forEach((item) => {
    resultado.add(item);
  });
  return resultado;
};
```

- §

  Función utilitaria que determina si un conjunto ya se encuentra en Destados.

```
const estaConjuntoEnLista = (
  lista: Set<Nodo>[],
  conjunto: Set<Nodo>
): boolean => {
  for (let elem of lista) {
    if (compararConjuntos(elem, conjunto)) return true;
  }
  return false;
};
```

- §

  Función utilitaria que retorna la posición del conjunto en Destados.

```
const encontrarConjunto = (lista: Set<Nodo>[], conjunto: Set<Nodo>): number => {
  for (let i = 0; i < lista.length; i++) {
    let elem = lista[i];
    if (compararConjuntos(elem, conjunto)) return i;
  }
  return -1;
};
```

- §

  Función utilitaria que determina si dos conjuntos son iguales.

```
const compararConjuntos = (
  conjuntoA: Set<Nodo>,
  conjuntoB: Set<Nodo>
): boolean => {
  if (conjuntoA.size !== conjuntoB.size) return false;
  let retorno = true;
  conjuntoA.forEach((elem) => {
    if (!conjuntoB.has(elem)) {
      retorno = false;
    }
  });
```

```
  return retorno;
};
```

- §

  Función utilitaria que determina si un estado pertenece a una lista de estados.

```
const estaNodoEnLista = (listaNodos: Nodo[], nodoA: Nodo): boolean => {
  for (let nodoB of listaNodos) {
    if (esIgualNodo(nodoA, nodoB)) return true;
  }
  return false;
};
```

- §

  Función utilitaria que compara dos estados.

```
const esIgualNodo = (a: Nodo, b: Nodo): boolean => {
  return a.etiqueta === b.etiqueta;
};
```

- §

  Función que a partir de un AFN obtiene un AFD.

```
export const getAFD = (afn: Automata): Automata => {
  const Destados: Set<Nodo>[] = [];
  const nodos: Nodo[] = [];
```

- §

  Se inicializa Destados con la cerradura epsilon del nodo inicial

```
  const S = afn.inicio;
  Destados.push(epsilonCerradura(S));
  nodos.push(new Nodo(String(0), false));
  let i = 0;
  const alfabeto = afn.alfabeto;
```

- §

  Mientras se tengan conjuntos que procesar en Destados se procesa el conjunto T

```
  while (i < Destados.length) {
    let T = Destados[i];
    let nodoT = nodos[i];
```

- §

  Para cada carácter del alfabeto

```
    alfabeto.forEach((caracter) => {
```

- §

  Se calcula el estado U. Resultado de aplicar la cerradura epsilon sobre mover(T,carácter)

```
      let U = epsilonCerraduraT(mover(T, caracter));
      if (U.size > 0) {
```

- §

  Si no se encuentra en Destados, agregar U

```
        if (!estaConjuntoEnLista(Destados, U)) {
          nodos.push(new Nodo(String(Destados.length), false));
          Destados.push(U);
        }
        let index = encontrarConjunto(Destados, U);
        let nodoU = nodos[index];
```

- §

  Se agrega la transición T -> U con la etiqueta carácter

```
        nodoT.agregarArista(nodoU, caracter);
      }
    });
    i++;
  }
```

- §

  Se asigna como estado inicial del AFD a aquel que contenga el estado inicial del AFN.

```
  let afd = new Automata(nodos[0]);
  afd.alfabeto = alfabeto;
```

- §

  Se etiquetan los estados finales Para cada conjunto en Destados

```
  for (let j = 0; j < Destados.length; j++) {
    let conjunto: Set<Nodo> = Destados[j];
    nodos[j].representacion = Destados[j]
```

- §

  Para cada estado del conjunto

```
    conjunto.forEach((elem) => {
```

- §

  Si es un estado de aceptación, se le agrega su clase correspondiente.

```
      if (elem.esAceptacion) {
        nodos[j].setAceptacion(true, elem.clase);
      }
    });
  }
  return afd;
};
```