Folder src

10 printable files

(file list disabled)

src/console-reader/index.ts

```ts
import readlineP from "readline/promises";
import { IReader } from "../interface";

export class ConsoleReader implements IReader {
    constructor() {
        this.iface = readlineP.createInterface({
            input: process.stdin
        });
    }

    async readString(): Promise<string> {
        return new Promise<string>((resolve) => {
            this.iface.once("line", resolve);
        });

    }

    async readRegex(): Promise<string> {
        return new Promise<string>((resolve) => {
            this.iface.once("line", resolve);
        });
    }

    async [Symbol.asyncDispose]() {
        this.iface.close();
    }

    private iface: readlineP.Interface;
}
```

src/file-reader/index.ts

```ts
import fsP from "fs/promises";
import readlineP from "readline/promises";
import { IReader } from "../interface";

export class FileReader implements IReader {
    constructor(filePath: string) {
        this.filePromise = fsP.open(filePath);
    }

    async readString(): Promise<string> {
        const file = await this.filePromise;
        const iface = readlineP.createInterface({
            input: file.createReadStream()
        });

        return new Promise<string>((resolve) => {
            iface.once("line", line => {
                iface.close();
```

```
                resolve(line);
            });
        });
    }

    async readRegex(): Promise<string> {
        const file = await this.filePromise;
        return (await file.readFile({encoding: 'utf-8'})).trim();
    }

    async [Symbol.asyncDispose]() {
        const file = await this.filePromise;
        await file.close();
    }

    private filePromise: Promise<fsP.FileHandle>;
}

```

src/fsm/index.ts

```
import { IFiniteStateMachineBuilder, IFSM, INodeType, ITree, ITreeFuncs } from "../interface";

export class FSMBuilder implements IFiniteStateMachineBuilder {
    async buildFSM(tree: ITree, funcs: ITreeFuncs, alphabet: string = "abcdefghijklmnopqrstuvwxyz"): Promise<IFSM> {
        const states: IFSM['states'] = [];
        const transitionFunction: IFSM['transitionFunction'] = {};

        let stateId = -1;

        states.push({id: ++stateId, positions: new Set<number>([...funcs.firstpos[funcs.root]])});

        const initialState = stateId;
        const finalStates: number[] = [];
        const unmarkedStates = [states[0]];

        for(const pos of states[0].positions) {
            const node = tree.nodes[pos];
            if (node.type === INodeType.NODE_CHAR && node.content === "#") {
                finalStates.push(states[0].id);
            }
        }

        while(unmarkedStates.length > 0) {
            const s = unmarkedStates.pop()!;

            for(const a of alphabet) {
                const u = new Set<number>();

                for(const p of s.positions) {
                    const node = tree.nodes[p];

                    if (node.type === INodeType.NODE_CHAR && node.content === a) {
                        for(const pos of funcs.followpos[p]) {
                            u.add(pos);
                        }
```

```
                    }
                }

                let id = -1;

                for(const st of states) {
                    if (this.setsEqual(st.positions, u)) {
                        id = st.id;
                    }
                }

                if (id === -1) {
                    const newSt = {id: ++stateId, positions: u};

                    states.push(newSt);
                    unmarkedStates.push(newSt);

                    for(const pos of newSt.positions) {
                        const node = tree.nodes[pos];
                        if (node.type === INodeType.NODE_CHAR && node.content === "#") {
                            finalStates.push(newSt.id);
                        }
                    }

                    id = stateId;
                }

                transitionFunction[s.id] = {...transitionFunction[s.id], [a]: id};
            }
        }

        return {
            states,
            transitionFunction,
            initialState,
            finalStates,
            alphabet
        };
    }

    async buildMinifiedFSM(fsm: IFSM): Promise<IFSM> {
        if (fsm.states.length <= 1) {
            // Nothing to minimize
            return {...fsm};
        }

        const {states, finalStates, transitionFunction: d, alphabet: S} = fsm;
        const Q = new Set<number>(states.map((s) => s.id));
        const F = new Set<number>(finalStates);
        const P = [F, this.setDifference(Q, F)];
        const Class: Record<number, number> = {};
        for(const q of F) {
            Class[q] = 0;
        }
        for(const q of this.setDifference(Q, F)) {
            Class[q] = 1;
        }
        const Inv: Record<number, Record<string, number[]>> = {};
```

```
94
95          for(const [sourceState, transitions] of Object.entries(d)) {
96              for(const [sym, targetState] of Object.entries(transitions)) {
97                  if (Inv[targetState] == null) {
98                      Inv[targetState] = {};
99                  }
100                 if (Inv[targetState][sym] == null) {
101                     Inv[targetState][sym] = [];
102                 }
103                 Inv[targetState][sym].push(parseInt(sourceState));
104             }
105         }
106
107         const Queue: Array<[Set<number>, string]> = [];
108
109         for(const c of S) {
110             Queue.push([F, c]);
111             Queue.push([this.setDifference(Q, F), c]);
112         }
113
114         while(Queue.length > 0) {
115             const [C, a] = Queue[0];
116             Queue.splice(0, 1);
117             const Involved: Record<number, number[]> = {};
118
119             for (const q of C) {
120                 for (const r of Inv[q]?.[a] ?? []) {
121                     const i = Class[r];
122                     if (Involved[i] == null) {
123                         Involved[i] = [];
124                     }
125                     Involved[i].push(r);
126                 }
127             }
128
129             for (const i of Object.keys(Involved)) {
130                 if (Involved[i].length < P[i].size) {
131                     const j = P.push(new Set<number>()) - 1;
132                     for(const r of Involved[i]) {
133                         P[i].delete(r);
134                         P[j].add(r);
135                     }
136
137                     if (P[j].size > P[i].size) {
138                         const tmp = P[i];
139                         P[i] = P[j];
140                         P[j] = tmp;
141                     }
142
143                     for(const r of P[j]) {
144                         Class[r] = j;
145                     }
146
147                     for(const c of S) {
148                         Queue.push([P[j], c]);
149                     }
150                 }
151             }
```

```
152         }
153
154         const minStates = P.map((el, i) => ({id: i, positions: el}));
155         const minFinalStates = minStates.filter((el) => this.setIntersection(el.positions,
    new Set<number>(fsm.finalStates)).size > 0);
156         const minInitialStates = minStates.filter((el) =>
    this.setIntersection(el.positions, new Set<number>([fsm.initialState])).size > 0);
157
158         const minTransitionFunction: IFSM['transitionFunction'] = {};
159
160         for(const sourceState of minStates) {
161             for (const targetState of minStates) {
162                 const sId = sourceState.id;
163                 const sPos = [...sourceState.positions];
164                 const tId = targetState.id;
165                 const tPos = [...targetState.positions];
166
167                 if (minTransitionFunction[sId] == null) {
168                     minTransitionFunction[sId] = {};
169                 }
170
171                 for(const a of fsm.alphabet) {
172                     if (sPos.some(p => tPos.includes(d[p]?.[a] ?? -1))) {
173                         minTransitionFunction[sId][a] = tId;
174                     }
175                 }
176             }
177         }
178
179         return {
180             alphabet: fsm.alphabet,
181             states: minStates,
182             finalStates: minFinalStates.map(el => el.id),
183             initialState: minInitialStates[0]?.id,
184             transitionFunction: minTransitionFunction
185         }
186     }
187
188     protected setsEqual<T = number>(a: Set<T>, b: Set<T>): boolean {
189         for(const aEl of a) {
190             if (!b.has(aEl)) {
191                 return false;
192             }
193         }
194
195         for(const bEl of b) {
196             if (!a.has(bEl)) {
197                 return false;
198             }
199         }
200
201         return true;
202     }
203
204     protected setDifference<T = number>(a: Set<T>, b: Set<T>, eq: (x: T, y: T) => boolean =
    (x, y) => x === y): Set<T> {
205         const result: Set<T> = new Set<T>();
206
```

```
        for(const aEl of a) {
            let isInB: boolean = false;

            for(const bEl of b) {
                if (eq(aEl, bEl)) {
                    isInB = true;
                    break;
                }
            }

            if (!isInB) {
                result.add(aEl);
            }
        }

        return result;
    }

    protected setIntersection<T = number>(a: Set<T>, b: Set<T>): Set<T> {
        const result = new Set<T>();

        for(const aEl of a) {
            if (b.has(aEl)) {
                result.add(aEl);
            }
        }

        return result;
    }
}

```

src/index.ts

```
export * from "./fsm";
export * from "./printer";
export * from "./file-reader";
export * from "./simulator";
export * from "./tree";
export * from "./tree-funcs";
export * from "./interface";
```

src/interface.ts

```
export interface IReader {
    readString(): Promise<string>;
    readRegex(): Promise<string>;
};

export enum INodeType {
    NODE_CHAR,
    NODE_CONCAT,
    NODE_ALT,
    NODE_ITER,
    NODE_ZITER,
    NODE_OPENING_BRACE,
    NODE_CLOSING_BRACE
};

```

```typescript
export interface INode {
    id: number;
    type: INodeType;
    content?: string;
};

export interface ITree {
    nodes: Record<number, INode>;
    parents: Record<number, [number, boolean] | undefined>;
};

export interface ITreeFuncs {
    nullable: Record<number, boolean>;
    firstpos: Record<number, Set<number>>;
    lastpos: Record<number, Set<number>>;
    followpos: Record<number, Set<number>>;
    root: number;
}

export interface IFSMState {
    id: number;
    positions: Set<number>;
}

export interface IFSM {
    states: Array<IFSMState>;
    transitionFunction: Record<number, Record<string, number>>;
    initialState: number;
    finalStates: number[];
    alphabet: string;
}

export interface IFSMSimResult {
    accepted: boolean;
    error?: Error;
    steps: Array<{id: number; curStateId: number; char: string; isInitial: boolean; isFinal: boolean}>;
}

export interface ITreeBuilder {
    buildTree(regex: string): Promise<ITree>;
};

export interface IPrinter {
    printTree(tree: ITree): Promise<void>;
    printFSM(fsm: IFSM): Promise<void>;
}

export interface ITreeFuncComputer {
    computeTreeFuncs(tree: ITree): Promise<ITreeFuncs>;
}

export interface IFiniteStateMachineBuilder {
    buildFSM(tree: ITree, funcs: ITreeFuncs, alphabet: string): Promise<IFSM>;
    buildMinifiedFSM(fsm: IFSM): Promise<IFSM>;
}

export interface IFiniteStateMachineSimulator {
```

```
    simulateFSM(fsm: IFSM, input: string): Promise<IFSMSimResult>;
}
```

src/main.ts

```
import { FSMBuilder } from "./fsm";
import { DotFilePrinter } from "./printer";
import { FileReader } from "./file-reader";
import { TreeBuilder } from "./tree";
import { TreeFuncComputer } from "./tree-funcs";
import { ConsoleReader } from "./console-reader";
import { IFSM } from "./interface";
import { FSMSimulator } from "./simulator";

interface IState {
    input: () => Promise<string>;
    regex?: string;
    fsm?: IFSM;
    minFsm?: IFSM;
};

interface ICommand {
    help: string;
    invoke(state: IState): Promise<IState>;
}

const commands: ICommand[] = [
    {
        help: 'exit',
        invoke: async (state) => {
            process.exit(0);

            return state;
        }
    },
    {
        help: 'read regular expression',
        invoke: async (state) => {
            console.log('Enter file path or "-" to read regex from stdin');

            const filePath = await state.input();
            const regexReader = filePath === "-" ? new ConsoleReader() : new
FileReader(filePath);

            if (filePath === "-") {
                console.log('Enter regex, operations supported: (), |, +, *');
            }

            const regex = await regexReader.readRegex();
            state.regex = regex;

            console.log(`Read regex: "${regex}"`);

            return state;
        }
    },
    {
        help: 'build and save syntax tree in .dot format',
```

```
        invoke: async (state) => {
            if (state.regex == null) {
                console.error('No regular expression entered, skipping...');
                return state;
            }

            console.log('Enter file path to save the syntax tree');
            const filePath = await state.input();

            const builder = new TreeBuilder();
            const printer = new DotFilePrinter(filePath);

            const tree = await builder.buildTree(`(${state.regex})#`);

            await printer.printTree(tree);

            console.log(`Saved tree to file: ${filePath}`);

            return state;
        }
    },
    {
        help: 'build and save determined finite-state machine in .dot format',
        invoke: async (state) => {
            if (state.regex == null) {
                console.error('No regular expression entered, skipping...');
                return state;
            }

            console.log('Enter file path to save the finite-state machine or press ENTER to
skip saving');
            const filePath = await state.input();

            console.log('Enter the alphabet or press ENTER to guess it from the regular
expression');
            let alphabet = (await state.input()).trim();

            if (alphabet.length === 0) {
                alphabet = state.regex.split("").filter(ch => !"()+*#|
ɘ".includes(ch)).join("");
            }

            const builder = new TreeBuilder();
            const computer = new TreeFuncComputer();
            const fsmBuilder = new FSMBuilder();

            const tree = await builder.buildTree(`(${state.regex})#`);
            const funcs = await computer.computeTreeFuncs(tree);
            const fsm = await fsmBuilder.buildFSM(tree, funcs, alphabet);

            console.log('Built finite-state machine');

            if (filePath.length > 0) {
                const printer = new DotFilePrinter(filePath);
                await printer.printFSM(fsm);

                console.log(`Saved finite-state machine to file: ${filePath}`);
            }
```

```
108
109             state.fsm = fsm;
110             return state;
111         }
112     },
113     {
114         help: 'minimize and save pre-built determined finite-state machine in .dot format',
115         invoke: async (state) => {
116             if (state.fsm == null) {
117                 console.error('No built finite-state machine, skipping...');
118                 return state;
119             }
120
121             console.log('Enter file path to save the finite-state machine or press ENTER to
    skip saving');
122             const filePath = await state.input();
123
124             const fsmBuilder = new FSMBuilder();
125             const minFsm = await fsmBuilder.buildMinifiedFSM(state.fsm);
126
127             console.log('Minimized finite-state machine');
128
129             if (filePath.length > 0) {
130                 const printer = new DotFilePrinter(filePath);
131                 await printer.printFSM(minFsm);
132
133                 console.log(`Saved minimized finite-state machine to file: ${filePath}`);
134             }
135
136             state.minFsm = minFsm;
137             return state;
138         }
139     },
140     {
141         help: 'simulate minimized finite-state machine',
142         invoke: async (state) => {
143             if (state.minFsm == null) {
144                 console.error('Minimize finite-state machine before simulating,
    skipping...');
145                 return state;
146             }
147
148             const simulator = new FSMSimulator();
149
150             do {
151                 console.log('Enter input string or press ENTER to exit');
152                 const input = await state.input();
153
154                 if (input.length === 0) {
155                     return state;
156                 }
157
158                 const result = await simulator.simulateFSM(state.minFsm, input);
159
160                 console.log('Simulation Result');
161                 console.log('Input Accepted: ', result.accepted);
162                 console.log('Steps:');
163
```

```
                for(const step of result.steps) {
                    console.log(`>>> Step #${step.id}`);
                    console.log(`Current State: ${step.curStateId}`);
                    console.log(`Character just read: ${step.char}`);
                    console.log(`State is initial: ${step.isInitial}; state is final: ${step.isFinal}`);
                }

                console.log('Simulation finished');

            } while (true);
        }
    }
]

async function main() {
    const cmdReader = new ConsoleReader();

    let state = {
        input: cmdReader.readString.bind(cmdReader)
    };

    let cmd = -1;

    do {
        console.log('Menu:');

        for(let i = 0; i < commands.length; ++i) {
            console.log(`${i}: ${commands[i].help}`);
        }

        console.log('Enter command:');
        const cmdStr = await cmdReader.readString();
        cmd = parseInt(cmdStr.trim(), 10);

        if (cmd >= 0 && cmd < commands.length) {
            state = await commands[cmd].invoke(state);
        } else {
            console.error('Invalid command!', cmdStr);
        }
    } while (true);
}

main();
```

src/printer/index.ts

```
import fsP from "fs/promises";

import { INodeType, ITree, IPrinter, IFSM } from "../interface";

export class DotFilePrinter implements IPrinter {
    constructor(filePath: string) {
        this.filePromise = fsP.open(filePath, "w");
    }

    async printTree(tree: ITree): Promise<void> {
```

```
        const file = await this.filePromise;

        await file.write("digraph G {\n");

        for(const node of Object.values(tree.nodes)) {
            const nodeLabels = {
                [INodeType.NODE_ALT]: "|",
                [INodeType.NODE_CONCAT]: "@",
                [INodeType.NODE_ITER]: "+",
                [INodeType.NODE_ZITER]: "*",
                [INodeType.NODE_CHAR]: undefined
            };

            const nodeLabel = nodeLabels[node.type] ?? node.content ?? `node_${node.id}`;

            await file.write(`node_${node.id} [shape=circle style=filled label="${node.id}\n${nodeLabel}"];\n`);
        }

        for(const entry of Object.entries(tree.parents)) {
            const [childId, parent] = entry;

            if (parent) {
                const [parentId, isRight] = parent;

                await file.write(`node_${parentId} -> node_${childId} [color=${isRight?'red': 'blue'}];\n`);
            }
        }

        await file.write("}\n");

        await file.close();
    }

    async printFSM(fsm: IFSM): Promise<void> {
        const file = await this.filePromise;

        await file.write("digraph G {\n");

        await file.write('state_fake [label="",shape=none,height=.0,width=.0];\n');

        for(const state of fsm.states) {
            const shape = fsm.finalStates.includes(state.id) ? "doublecircle" : "circle";

            await file.write(`state_${state.id} [shape=${shape} style=filled label="${state.id}\n${[...state.positions]}"];\n`);
        }

        for(const [sourceStateId, transitions] of Object.entries(fsm.transitionFunction)) {
            for(const [sym, targetStateId] of Object.entries(transitions)) {
                await file.write(`state_${sourceStateId} -> state_${targetStateId} [label="${sym}"];\n`)
            }
        }

        await file.write(`state_fake -> state_${fsm.initialState};\n`);

```

```
        await file.write("}\n");

        await file.close();
    }

    private filePromise: Promise<fsP.FileHandle>;
}

```

src/simulator/index.ts

```
import { IFiniteStateMachineSimulator, IFSM, IFSMSimResult } from "../interface";

export class FSMSimulator implements IFiniteStateMachineSimulator {
    async simulateFSM(fsm: IFSM, input: string): Promise<IFSMSimResult> {
        const steps: IFSMSimResult['steps'] = [];
        let curStateId = fsm.initialState;

        steps.push({
            id: steps.length + 1,
            curStateId,
            char: '<NULL>',
            isFinal: fsm.finalStates.includes(curStateId),
            isInitial: fsm.initialState === curStateId
        });

        for(const char of input) {
            try {
                curStateId = fsm.transitionFunction[curStateId][char];
            } catch (err) {
                return {
                    accepted: false,
                    error: err,
                    steps
                };
            }

            steps.push({
                id: steps.length + 1,
                curStateId,
                char,
                isFinal: fsm.finalStates.includes(curStateId),
                isInitial: fsm.initialState === curStateId
            });
        }

        return {
            accepted: fsm.finalStates.includes(curStateId),
            steps
        };
    }
}

```

src/tree-funcs/index.ts

```
import { INodeType, ITree, ITreeFuncComputer, ITreeFuncs } from "../interface";

export class TreeFuncComputer implements ITreeFuncComputer {
```

```
    async computeTreeFuncs(tree: ITree): Promise<ITreeFuncs> {
        const nullable: ITreeFuncs['nullable'] = {};
        const firstpos: ITreeFuncs['firstpos'] = {};
        const lastpos: ITreeFuncs['lastpos'] = {};
        const followpos: ITreeFuncs['followpos'] = {};

        const orderedNodeIds: number[] = [];
        const children: Record<number, Record<'l'|'r', number|undefined>> = {};
        let root: number = -1;

        for(const id of Object.keys(tree.nodes)) {
            const iid = parseInt(id);

            if (children[id] == null) {
                children[id] = {l: undefined, r: undefined};
            }

            if (tree.parents[id] != null) {
                const [parentId, isRight] = tree.parents[id];

                if (children[parentId] == null) {
                    children[parentId] = {l: undefined, r: undefined};
                }

                children[parentId][isRight ? 'r' : 'l'] = iid;
            } else {
                root = iid;
            }
        }

        const q = [root];

        while (q.length > 0) {
            const top = q.pop()!;
            const {l, r} = children[top];
            orderedNodeIds.unshift(top);

            if (l != null) {
                q.push(l);
            }

            if (r != null) {
                q.push(r);
            }
        }

        for(const id of orderedNodeIds) {
            const {l, r} = children[id];
            const node = tree.nodes[id];

            if (node.type === INodeType.NODE_CHAR) {
                const isEps = node.content === 'eps';
                nullable[id] = isEps;
                firstpos[id] = new Set<number>();
                lastpos[id] = new Set<number>();
                if (!isEps) {
                    firstpos[id].add(id);
                    lastpos[id].add(id);
```

```
            }

        } else if (node.type === INodeType.NODE_ZITER) {
            nullable[id] = true;
            firstpos[id] = new Set<number>([...firstpos[l!]]);
            lastpos[id] = new Set<number>([...lastpos[l!]]);

            for (const pos of lastpos[id]) {
                if (followpos[pos] == null) {
                    followpos[pos] = new Set<number>();
                }

                for(const rPos of firstpos[id]) {
                    followpos[pos].add(rPos);
                }
            }

        } else if (node.type === INodeType.NODE_ITER) {
            nullable[id] = nullable[l!];
            firstpos[id] = new Set<number>([...firstpos[l!]]);
            lastpos[id] = new Set<number>([...lastpos[l!]]);

            for (const pos of lastpos[id]) {
                if (followpos[pos] == null) {
                    followpos[pos] = new Set<number>();
                }

                for(const rPos of firstpos[id]) {
                    followpos[pos].add(rPos);
                }
            }

        } else if (node.type === INodeType.NODE_ALT) {
            nullable[id] = nullable[l!] || nullable[r!];
            firstpos[id] = new Set<number>([...firstpos[l!], ...firstpos[r!]]);
            lastpos[id] = new Set<number>([...lastpos[l!], ...lastpos[r!]]);

        } else if (node.type === INodeType.NODE_CONCAT) {
            nullable[id] = nullable[l!] && nullable[r!];

            if (nullable[l!]) {
                firstpos[id] = new Set<number>([...firstpos[l!], ...firstpos[r!]]);
            } else {
                firstpos[id] = new Set<number>([...firstpos[l!]]);
            }

            if (nullable[r!]) {
                lastpos[id] = new Set<number>([...lastpos[l!], ...lastpos[r!]]);
            } else {
                lastpos[id] = new Set<number>([...lastpos[r!]]);
            }

            for (const pos of lastpos[l!]) {
                if (followpos[pos] == null) {
                    followpos[pos] = new Set<number>();
                }

                for(const rPos of firstpos[r!]) {
```

```
                        followpos[pos].add(rPos);
                    }
                }
            }
        }

        return {
            nullable,
            firstpos,
            lastpos,
            followpos,
            root
        }
    }
}

```

src/tree/index.ts

```ts
import { INode, INodeType, ITree, ITreeBuilder } from "../interface";

export class TreeBuilder implements ITreeBuilder {
    constructor() {

    }

    async buildTree(regex: string): Promise<ITree> {
        regex = regex.trim();

        const tree: ITree = {
            nodes: {},
            parents: {}
        };

        const stack: INode[] = [];

        let curNodeId = -1;
        let ch = '';
        let nextCh = regex[0];

        for(let i = 0; i < regex.length; ++i) {
            ch = nextCh;

            if (i === regex.length - 1 || '(|@'.includes(ch) || '|+*)'.includes(regex[i + 1])) {
                nextCh = regex[i + 1];
            } else {
                i--;
                nextCh = '@';
            }

            if (ch === '(') {
                stack.push({id: -1, type: INodeType.NODE_OPENING_BRACE});
            } else if (ch === ')') {
                let top: INode | undefined = undefined;
                let prevTop: INode | undefined = undefined;

                while(stack.length > 0 && stack[stack.length - 1].type !== INodeType.NODE_OPENING_BRACE) {
```

```
                    top = stack.pop()!;

                    if (prevTop) {
                        tree.parents[prevTop.id] = [top.id, true];
                    }

                    prevTop = top;
                }

                stack.pop();

                prevTop = stack[stack.length - 1];

                if (top) {
                    if (prevTop) {
                        tree.parents[top.id] = [prevTop.id, true];
                    }

                    stack.push({id: top.id, type: INodeType.NODE_CLOSING_BRACE});
                }
            } else if (ch === '+' || ch === '*') {
                const top = stack.pop()!;
                const prevTop = stack.pop();

                const node = {
                    id: ++curNodeId,
                    type: ch == '+' ? INodeType.NODE_ITER : INodeType.NODE_ZITER
                };

                tree.nodes[node.id] = node;

                if (prevTop) {
                    stack.push(prevTop);
                }
                stack.push(node);
                if (tree.parents[top.id]) {
                    tree.parents[node.id] = tree.parents[top.id];
                }

                tree.parents[top.id] = [node.id, false];
            } else if (ch === '|') {
                const node = {
                    id: ++curNodeId,
                    type: INodeType.NODE_ALT
                };
                tree.nodes[node.id] = node;

                let top: INode | undefined = undefined;
                let prevTop: INode | undefined = undefined;

                while(stack.length > 0 && stack[stack.length - 1].type !==
INodeType.NODE_OPENING_BRACE) {
                    top = stack.pop()!;

                    if (prevTop) {
                        tree.parents[prevTop.id] = [top.id, true];
                    }

```

```
                    prevTop = top;
                }

                if (!top) {
                    throw new Error('Incorrect regex');
                }

                tree.parents[top.id] = [node.id, false];

                stack.push(node);
            } else if (ch === '@') {
                const node = {
                    id: ++curNodeId,
                    type: INodeType.NODE_CONCAT
                };
                tree.nodes[node.id] = node;

                const top = stack.pop();

                if (!top) {
                    throw new Error('Invalid regex, | without left-hand side');
                }

                tree.parents[top.id] = [node.id, false];

                stack.push(node);
            } else {
                const node = {id: ++curNodeId, type: INodeType.NODE_CHAR, content: ch === 'ə' ? 'eps' : ch};
                tree.nodes[node.id] = node;

                stack.push(node);
            }
        }

        let prevTop: INode | undefined = undefined;

        while(stack.length > 0) {
            const top = stack.pop()!;

            if (prevTop) {
                tree.parents[prevTop.id] = [top.id, true];
            }

            prevTop = top;
        }

        return tree;
    }
}

```