Slovenská Technická Univerzita v Bratislave

Fakulta informatiky a informačných technológií

Dopredný produkčný systém

Zadanie 4

Predmet: Umelá inteligencia

Obdobie: Letný semester 2019/2020

Cvičiaci: Ing. Ivan Kapustík

Študent: Bc. František Gič

Obsah

Zadanie

Úlohou je vytvoriť jednoduchý dopredný produkčný systém, s prípadnými rozšíreniami, napríklad o kladenie otázok používateľovi alebo vyhodnocovanie matematických výrazov.

Produkčný systém patrí medzi znalostné systémy, teda medzi systémy, ktoré so svojimi údajmi narábajú ako so znalosťami. Znalosti vyjadrujú nielen informácie o nejakom objekte, ale aj súvislosti medzi objektami, vlastnosti zvolených problémov a spôsoby hľadania ich riešenia. Znalostný systém je teda v najjednoduchšom prípade dvojica – program, ktorý dokáže všeobecne manipulovať so znalosťami a báza znalostí, ktorá opisuje problém a vzťahy, ktoré tam platia. Znalosti majú definovanú nejakú štruktúru a spôsob narábania s touto štruktúrou – to sa nazýva formalizmus reprezentácie znalostí. Program vie pracovať s týmto formalizmom, ale nesmie byť závislý od toho, aké konkrétne znalosti spracováva, inak by to už nebol systém, kde riešenie úlohy je dané použitými údajmi.

Implementácia

Riešenie zadania som vypracoval v jazyku *JavaScript*, lokálnom environmente - *Node.js* s použitím supersetu *Typescript* pre striktné otypovanie.

Inštalácia

Prerekvizity:

- Node.js (https://www.nodejs.org/)
- Node package manager (https://www.npmjs.com/)

V root adresári spustite nasledovné príkazy:

```
npm install
```

A následne, pre každú transpiláciu typescriptového kódu na javascript a spustenie kódu v node.js:

```
npm run dev
```

Reprezentácia údajov

Vstupné dáta načítavam vo formáte JSON. Sú to primitívne dátové typy, reťazce (string). Pravidlá produkčného systému sú pre zjednodušenie uložené v objekte, aby som nemusel riešiť základné problémy ako koniec riadku alebo nejaké oddelovače.

```
export default {
    facts: [
        'Peter je rodic Jano',
        'Peter je rodic Vlado',
        'manzelia Peter Eva',
        'Vlado je rodic Maria',
        'Vlado je rodic Viera',
        'muz Peter',
        'muz Jano',
        'muz Vlado',
        'zena Maria',
        'zena Viera',
        'zena Eva',
    ],
    rules: [
        {
            name: 'DruhyRodic1',
            if: ['?X je rodic ?Y', 'manzelia ?X ?Z'],
            then: ['pridaj ?Z je rodic ?Y'],
        },
```

Fakty sa následne sparsujú do objektu typu `Relationship`. Tento objekt obsahuje dáta (aktérov daného vzťahu) a ich vzťah v reťazci – rodič, muž atď.

```
Facts:  [
  { data: [ 'Peter', 'Jano' ], name: 'rodic' },
  { data: [ 'Peter', 'Vlado' ], name: 'rodic' },
  { data: [ 'Peter', 'Eva' ], name: 'manzelia' },
  { data: [ 'Vlado', 'Maria' ], name: 'rodic' },
  { data: [ 'Vlado', 'Viera' ], name: 'rodic' },
  { data: [ 'Peter' ], name: 'muz' },
  { data: [ 'Jano' ], name: 'muz' },
  { data: [ 'Vlado' ], name: 'muz' },
```

Pravidlá, tie sa sparsujú na objekt typu `Rule`. Ten obsahuje názov (`name`), `if` – akcie ktoré su podmienkami – je to pole typu `Relationship` a `then`, čo je pole typu `Action` – názov akcie (pridaj, vymaž, správa) a vzťah z ktorým sa pracuje.

```
{
    name: "DruhyRodic2",
    if: [
        { name: "rodic", data: ["?X", "?Y"] },
        { name: "manzelia", data: ["?Z", "?X"] },
    ],
    then: [
        { name: "pridaj", relationship: { name: "rodic", data: ["?Z", "?Y"] } },
    ],
},
{
    name: "Otec",
    if: [
        { name: "rodic", data: ["?X", "?Y"] },
        { name: "muz", data: ["?X"] },
    ],
    then: [
        { name: "pridaj", relationship: { name: "otec", data: ["?X", "?Y"] } },
    ],
},
```

Algoritmus

Samotný hlavné telo programu je pomerne jednoduché. V každej iterácií cyklu prechádzame všetky pravidlá. Ku každému pravidlu získame `Bindingy` – možné aplikovateľné pravidlá ktoré spĺňajú jednotlivé podmienky – pole výsledkov z prvej podmienky, druhé pole z druhej a pod... Ak je teda podmienka, že X je rodič Y, vráti to pole vzťahov – napr Peter -> Jano, lebo ich vzťah je typu rodič.

Následne sa tieto `bindings` posielajú do funkcie `getRuleMatches` ktorá tieto polia vyhodnotí, nájde zhodu medzi nimi a vráti pole výsledkov.

Následne cez tieto výsledky iterujeme a vykonávame akcie ktoré sú definované v `then` daného pravidla. Napríklad, ak je tam `pridaj`, do stacku sa pridá nové pravidlo (pokiaľ už neexistuje v stacku alebo faktoch).

Na koniec sa do faktov (pracovnej pamäte) priradí prvé pravidlo zo stacku (pomocného výstupu).

Toto sa opakuje pokiaľ v pomocnom výstupe (stacku) existujú nejaké elementy.

```
const rules = parseRules(input.rules);
let facts = parseFacts(input.facts);
const messages: string[] = [];

const stack = [] as any;
do {
    rules.forEach((rule: Rule) => {
        const bindings = getBindings(rule, facts);
        const results = getRuleMatches(bindings);
        results.forEach((result: Binding[]) => {
            apply(rule.then, result, { messages, stack, facts });
        });
    });
    facts.push(stack.shift());
} while (stack.length);

console.log('Facts: ', facts);
```

Testovanie

Riešenie som testoval na príkladovom zadaní, ktorého pravidlá a fakty som vložil s načítal zo súboru.

Výstup obsahoval dve hlášky, podľa pravidiel, a taktiež výpis pracovnej pamäte tak ako v príkladovom projekte na webstránke.

```
Maria ma stryka
Viera ma stryka
Facts:  [
  { data: [ 'Peter', 'Jano' ], name: 'rodic' },
  { data: [ 'Peter', 'Vlado' ], name: 'rodic' },
  { data: [ 'Peter', 'Eva' ], name: 'manzelia' },
  { data: [ 'Vlado', 'Maria' ], name: 'rodic' },
  { data: [ 'Vlado', 'Viera' ], name: 'rodic' },
  { data: [ 'Peter' ], name: 'muz' },
  { data: [ 'Jano' ], name: 'muz' },
  { data: [ 'Vlado' ], name: 'muz' },
  { data: [ 'Eva' ], name: 'zena' },
  { data: [ 'Eva', 'Jano' ], name: 'rodic' },
  { data: [ 'Eva', 'Vlado' ], name: 'rodic' },
  { data: [ 'Peter', 'Jano' ], name: 'otec' },
  { data: [ 'Peter', 'Vlado' ], name: 'otec' },
  { data: [ 'Vlado', 'Maria' ], name: 'otec' },
  { data: [ 'Vlado', 'Viera' ], name: 'otec' },
  { data: [ 'Jano', 'Vlado' ], name: 'surodenci' },
  { data: [ 'Vlado', 'Jano' ], name: 'surodenci' },
  { data: [ 'Maria', 'Viera' ], name: 'surodenci' },
  { data: [ 'Viera', 'Maria' ], name: 'surodenci' },
  { data: [ 'Eva', 'Jano' ], name: 'matka' },
  { data: [ 'Eva', 'Vlado' ], name: 'matka' },
  { data: [ 'Jano', 'Vlado' ], name: 'brat' },
  { data: [ 'Vlado', 'Jano' ], name: 'brat' },
  { data: [ 'Jano', 'Maria' ], name: 'stryko' },
  { data: [ 'Jano', 'Viera' ], name: 'stryko' }
]
```

Zhodnotenie

Riešenie som vypracoval a otestoval na vzorovom a považujem ho za správne.
Najväčším problémom nad ktorým som strávil veľa času bola funkcia `getRuleMatches`, ktorá dostane ako parameter pole polí ktoré spĺňajú jednotlivé podmienky daného pravidla a výstupom by malo byť pole kombinácií (X,Y,Z) ktoré jednotlivé podmienky spĺňa. Pôvodne som daný problém chcel riešiť rekurzívne, ale použil som možnosť spôsob, kde dvojdimenzionálne pole iterujem v kombináciách po dvoch. Vezmem spoločný stĺpec daných dvoch polí a spojím ich podľa kľúča v danom stĺpci. Potom s každým nasledujúcim pravidlom už len robím prienik množín pravidiel. Tieto výsledky sa potom vrátia do hlavnej časti programu ako trojice X Y Z s príslušnými hodnotami a pokračuje sa vyhodnotením funkcie `then`.
Taktiež som mal problémy s neustálym výpisom správ z funkcie `sprava.` Problém som vyriešil evidovaním už vypísaných správ a pred výpisom správy kontrolujem či sa daná správa už neeviduje (čiže nebola už vypísaná predtým).