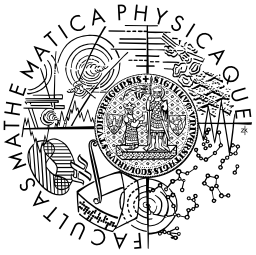


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

David Petera

Transformace hierarchických dat

Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Petr Škoda, Ph.D.

Studijní program: Informatika (B0613A140006)

Studijní obor: IPP (0613RA1400060000)

Praha 2024

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Chtěl bych poděkovat svému vedoucímu Mgr. Petru Škodovi, Ph.D. za odbornou pomoc a trpělivost. Děkuji také svým přátelům, partnerstvu a rodině za podporu při psaní této práce.

Název práce: Transformace hierarchických dat

Autor: David Petera

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Petr Škoda, Ph.D., Katedra softwarového inženýrství

Abstrakt: Hierarchická data, tedy data která svou strukturou připomínají strom, jsou často serializovaná různými formáty jako například JSON, XML, RDF nebo CSV. I přes jejich společnou strukturu nejsou transformace či převody mezi nimi vždy jednoduché. V této práci se pokusíme navrhnout přístup, který by hierarchická data různých formátů transformovat dokázal. Součástí řešení je návrh tzv. Unifikované reprezentace, neboli způsobu jak takové formáty společně reprezentovat. Pro samotnou transformaci jsme navrhli vlastní transformační jazyk, který je schopen s reprezentací pracovat. Vytvořili jsme proof-of-concept implementaci navrhovaného řešení a jeho funkčnost zvalidovali pomocí uživatelské a případové studie.

Klíčová slova: Hierarchická data, Transformace, RDF, XML, JSON, CSV

Title: Transformation of hierarchical data

Author: David Petera

Department: Department of Software Engineering

Supervisor: Mgr. Petr Škoda, Ph.D., Department of Software Engineering

Abstract: Hierarchical data, which in structure resemble a tree, are often serialized in various formats such as JSON, XML, RDF, or CSV. Despite their common structure, transformations or conversions between them are not always straightforward. In this thesis, we attempt to propose an approach that would enable the transformation of hierarchical data across different formats. A part of the solution is the design of the so-called Unified Representation, a method for jointly representing such formats. For the actual transformation, we have designed our own transformation language capable of working with this representation. We created a proof-of-concept implementation of the proposed solution and validated its functionality through a user and case study.

Keywords: Hierarchical data, Transformation, RDF, XML, JSON, CSV

Obsah

Úvod

Na mnoho formátů, například XML, JSON, RDF a CSV, lze nahlížet jako na hierarchické, tedy reprezentovatelné stromovým grafem. Tato společná vlastnost umožňuje transformaci dat mezi jednotlivými formáty. V současné době ale transformace mezi těmito formáty nejsou tak přímočaré. Existující transformace jsou většinou omezené pouze na dvojice formátů, nevyužívají specifik daného formátu, či neumožňují dostatečnou kontrolu nad výslednou strukturou. Navíc stávající nástroje a jazyky často poskytují funkcionalitu, která je jen málo kdy využívána, přesto ale přidává na složitosti zápisu transformací. To vytváří prostor pro nový nástroj, který se bude soustředit na jednotnou reprezentaci stromových dat, jednoduchost a rozšiřitelnost.

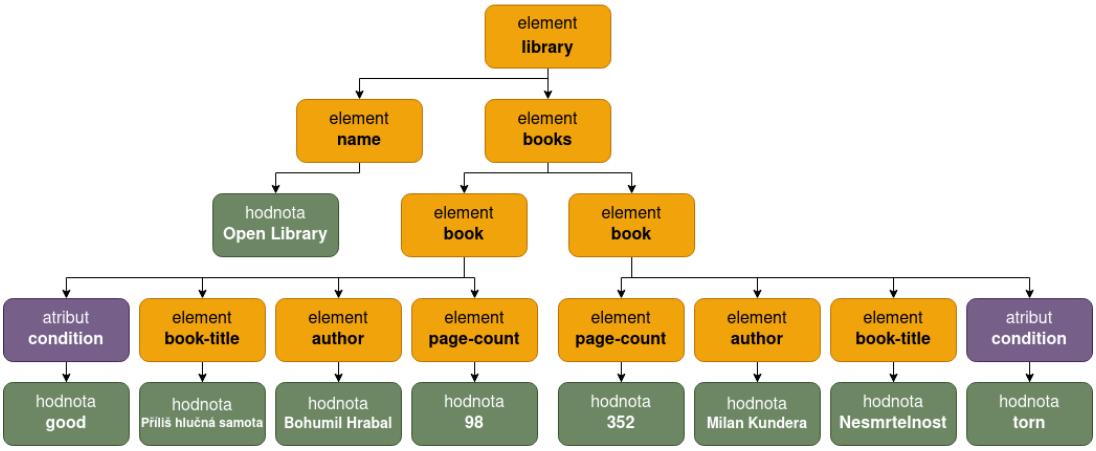
V této práci nejprve zmapujeme existující přístupy pro převod mezi výše uvedenými formáty. V dalším kroku navrhneme řešení pro transformaci s důrazem na kontrolu výsledné struktury. Následně implementujeme proof-of-concept softwarového nástroje, který bude schopen vyhodnocovat dotazy v navrženém jazyce a provádět transformace.

Práce je strukturovaná do kapitol. V kapitole 1 představíme čtenářstvu hierarchické formáty relevantní pro naši práci. V kapitole 2 provedeme rešerši již existujících nástrojů využívaných při transformaci a stručně porovnáme jejich vlastnosti. V kapitole 3 navážeme na kapitolu 2 a provedeme analýzu požadavků na návrh řešení a jeho jednotlivé aspekty. V kapitole 4 detailně popíšeme a vysvětlíme návrh našeho nástroje. Konkrétně se jedná o tzv. Unifikovanou reprezentaci v sekci 4.1, dotazovací jazyk UrPath v sekci 4.2 a transformační jazyk s jeho operacemi v sekci 4.3 a 4.4. Kapitola 5 se věnuje proof-of-concept implementaci navrhovaného nástroje a jeho dokumentaci. Uživatelská dokumentace v sekci 5.2 slouží i jako shrnutí nejdůležitejších konceptů práce, potřebných pro používání nástroje. V poslední kapitole 6 náš návrh zhodnotíme pomocí uživatelské a případové studie a nastíníme možnosti budoucí práce.

V příloze se nachází hledané fráze, script i výsledky hledání využívané při rešerši nástrojů, zdrojový kód naší implementace i s jeho zkompilovanou verzí a podklady pro uživatelskou i případovou studii.

1. Hierarchické datové formáty

Představme ve stručnosti konkrétní datové formáty se kterými budeme pracovat. V první kapitole se postupně budeme věnovat obecně formátům XML, JSON, RDF a nakonec CSV. Pro lepší ilustraci jednotlivých formátů využijeme společného datového modelu jednoduché knihovny s knížkami viz Obrázek 1.



Ukázkový datový model jednoduché knihovny.

XML

Velmi používaným a pravděpodobně nejznámějším hierarchickým datovým formátem je XML neboli eXtensible Markup Language. Jedná se o značkovací jazyk do kterého se hierarchie vnáší vnořováním značek do sebe. Odlišností XML od ostatních heirarchických formátů zmíněných v dalších podkapitolách je možnost využití tzv. atributů, dodatečných informací vložených přímo dovnitř značek formou nazevAtributu="hodnotaAtributu" [1]. Ty lze využívat například pro ukládání metadat. XML má také velké množství odvozených formátů, které si definují značky s vlastním významem. Jedná se například o formát HTML, SVG nebo třeba ODF.

Příklad:

```
<?xml version="1.0" encoding="UTF-8"?>
<library>
  <name>Open Library</name>
  <books>
    <book condition="good">
      <book-title>Příliš hlučná samota</book-title>
      <author>Bohumil Hrabal</author>
      <page-count>98</page-count>
    </book>
    <book contition="torn">
      <book-title>Nesmrtelnost</book-title>
      <author>Milan Kundera</author>
      <page-count>352</page-count>
    </book>
```

```
    </books>
  </library>
```

JSON

JavaScript Object Notation neboli JSON je hierarchický formát původně odvozený z jazyka Javascript. Hlavními konstrukty jazyka jsou pole a dvojice KLÍČ - HODNOTA. JSON bývá považován za přehlednější formát než XML, ale narozdíl od XML nepodporuje koncept atributů, a dokonce ani komentářů[1].

Příklad:

```
{
  "library": {
    "name": "Open Library",
    "books": [{
      "attributes": {
        "condition": "good"
      }
      "book-title": "Příliš hlučná samota",
      "author": "Bohumil Hrabal",
      "page-count": 98
    }, {
     "attributes": {
        "condition": "torn"
      }
      "book-title": "Nesmrtelnost",
      "author": "Milan Kundera",
      "page-count": 352
    }]
  }
}
```

RDF

Původně vytvořený pro popis metadat, Resource Description Framework neboli RDF dokáže popsat v podstatě libovolnou grafovou strukturu. Dokáže to pomocí trojic SUBJEKT - PREDIKÁT - OBJEKT. Trojice vychází ze struktury prohlášení, že nějaký zdroj (subjekt) má nějaký vztah (predikát) s nějakou vlastností nebo jiným zdrojem (objekt). Většinou se v RDF dokumentech jako subjekty, predikáty i objekty využívají globálně jednoznačná URI pro zachování významu nezávisle na kontextu [2]. Dalo by se namítnout, že RDF není čistě hierarchický formát, když může popisovat libovolné grafové struktury, ale po určení

[1] urlhttps://www.json.org/json-en.html

kořenového elementu se tak na něj lze snadno dívat. RDF má mnoho serializací např. N-Triples[2], RDF/XML[3] nebo JSON-LD[4], ale velmi častá je serializace Turtle[5] v které je i následující příklad.

Příklad:

```
<https://example.net/library>
  <https://example.net/name> "Open Library" ;
  <https://example.net/books> [
    <https://example.net/attributes> [
      <https://example.net/condition> "good"
    ] ;
    <https://example.net/book-title> "Příliš hlučná samota"@cs ;
    <https://example.net/author> "Bohumil Hrabal" ;
    <https://example.net/page-count> 98
  ] , [
    <https://example.net/attributes> [
      <https://example.net/condition> "torn"
    ] ;
    <https://example.net/book-title> "Nesmrtelnost"@cs ;
    <https://example.net/author> "Milan Kundera" ;
    <https://example.net/page-count> 352
  ] .
```

CSV

Comma-seperated Values neboli CSV je jednoduchý, ale používaný formát pro ukládání tabulkových dat [3]. CSV je určeno spíše pro relační data, ale na každý záznam se ale lze dívat jako na objekt. Také například ve spojení s cizími klíči o hierarchické struktuře mluvit lze. A takováto abstrakce se nám bude hodit, jak uvidíme v pozdějších kapitolách.

Příklad:

```
library,condition,book-title,author,page-count
Open Library,good,Příliš hlučná samota,Bohumil Hrabal,98
Open Library,torn,Nesmrtelnost,Milan Kundera,352
```

[2] https://www.w3.org/TR/n-triples/

[3] urlhttps://www.w3.org/TR/REC-rdf-syntax/

[4] https://www.w3.org/TR/json-ld/

[5] https://www.w3.org/TR/turtle/

2. Stávající nástroje využívané při transformaci

V předchozí kapitole jsme si představili formáty pro uložení hierarchických dat. Tyto formáty můžeme využít například pro serializování objektů z programovacích jazyků, uložení a načtení webových stránek nebo dat v databázi. V případě, že chceme data změnit, můžeme tak učinit uvnitř aplikační logiky a nebo provést transformaci přímo nad konkrétním formátem. Pro druhou z těchto možností existuje řada nástrojů, které takové transformace umožňují, a které představíme v této kapitole.

Jako nástroj v této kapitole bereme nejenom implementované nástroje, ale i návrhy a speficikace jazyků určených k transformaci nebo nějaké její části.

Pro vyhledání již existujících nástrojů na transformaci hierarchických dat nebo jiných zdrojů relevantních pro téma naší práce jsme použili klíčové fráze psané v angličtině. Doufáme, že jsme tím zvýšili relevanci výsledků.

Klíčové fráze jsme vyhledávali pomocí vyhledávače DuckDuckGo a vzhledem k množství frází jsme si napsali pomocný skript v jazyce Python s využitím knihovny `duckduckgo-search` [1]. U každé klíčové fráze jsme se omezili na maximálně prvních patnáct vyhledaných odkazů. Všechna hlednání proběhla 17. 4. 2023.

Jak hledaná klíčová slova, tak i výsledky a pomocný skript jsou uloženy v hlavní příloze práce.

Nástroje jsme rozdělili do tří kategorií. Dotazovací jazyky, tranformační jazyky a transformační jazyky napříč formáty. Následující podkapitoly kopírují toto rozdělení a nástroje v krátkosti představí a u každého uvedou příklad.

2.1 Dotazovací jazky

Každý transformační jazyk potřebuje způsob, kterým se může jednoduše dotazovat nad daty. Ať už jde o základní referenci na záznam a nebo složitější filtrace dat, které už samy o sobě mohou data předzpracovat a transformovat, to je úkolem dotazovacích jazyků představených v této podkapitole.

2.1.1 XPath

Výrazový jazyk XPath slouží primárně k dotazování nad stromem zapsaném původně pouze ve formátu XML. V novějších verzích je však podporovaný i formát JSON [4]. XPath využívá datový model zvaný "XQuery and XPath Data Model", který rozšiřuje XML o definice hodnotových typů a definuje způsob odkazování na informace korektního XML dokumentu [4].

Struktura dotazu používá dopředná lomítka následovaná názvem elementu pro vyjádření cesty v hierarchii datového modelu. Cesty mohou být absolutní, pak musí začínat dopředným lomítkem, a nebo i relativní, pokud je známý kontext dotazu. Dále cesta může obsahovat možnosti přístupu ke speciálním hodotám

[1] `https://pypi.org/project/duckduckgo-search/`

např. `text()` pro přístup k textovým uzlům datového modelu neboli textu XML elementu nebo `position()` pro přístup k pozici uzlu v modelu. Specifický symbol je `*`, který označuje libovolný uzel modelu. Pomocí symbolu `@` lze přistupovat k atributům. Také je možno v hranatých závorkách specifikovat predikát, kterým lze filtrovat výsledky dotazu [4].

Následující dotaz vybere textovou hodnotu elementu `book-title` jen pokud je v elementu `book` autor Buhumil Hrabal a počet stránek je vyšší než 42.

```
library/books/book[author === "Bohumil Hrabal" \
    && page-count > 42)]/book-title/text()
```

2.1.2 JSONPath a JSONPath Plus

JSONPath je krátká specifikace a implementace dotazovancího jazyka nad JSONem velmi inspirovaná jazykem XPath 1.0. Byla navržena v roce 2007, kdy nebylo možné využít XPath pro JSON [5]. Další motivací bylo vytvořit dotazovací jazyk, který bude respektovat notaci přirozenou pro programovací jazyky z rodiny C, takže pro přístup k potomkům uzlu lze využít tzv. "tečkovou notaci"nebo notaci typickou pro přístup k elementům pole tj. pomocí hranatých závorek [5]. Dále např. díky tomu, že ve formátu JSON nejsou atributy, může JSONPath využívat symbol "@"pro odkaz na aktuální element.

JSONPath Plus je pak implementace a rozšíření JSONPath, které specifikuje nejasnosti v JSONPath, přidává nové operátory a základní datové typy[2].

Následující příklad filtruje knihy podle autora a počtu stránek a v případě splnění predikátu vrátí název knihy.

```
$.library.books[?(@.author === "Bohumil Hrabal" \
    && @.page-count > 42)].book-title
```

2.1.3 JSON Pointer

JSON Pointer je, spíše než dotazovací jazyk, řetězcová syntaxe pro identifikování konkrétní hodnoty formátu JSON. Jazyk tedy neumožňuje žádné filtrování. Primárně byl vytvořen za účelem použití ve fragmentech Uniform Resource Identifier (URI) a v JSON hodnotových řetězcích. Důraz je kladen na escapování symbolů s významem v URI adresách [6].

Následující příklad přistoupí na hodnotu klíče `book-title` v elementu s indexem jedna v poli `books`.

```
/library/books/1/book-title
```

[2] `https://github.com/JSONPath-Plus/JSONPath`

2.1.4 JSPath

JSPath je doménově specifický dotazovací jazyk nad formátem JSON. Je podobný jazyku JSONPath Plus představenému v podsekci 2.1.2 v tom, že také používá "tečkovou notaci"(narozdíl od JSONPath výlučně) pro přístup k potomkům objektu. Symbol tečky ale narozdíl od JSONPath používá i pro kořenový objekt. Pro označení predikátů se používají složené závorky[3].

Uveďme příklad stejného dotazu jako u nástroje JSONPath z podsekce 2.1.2.

```
.library.books{.author === "Bohumil Hrabal"  \
    && .page-count > 42}.book-title
```

2.1.5 JSONiq

JSONiq se odlišuje od výše zmíněných dotazovacích jazyků nad formátem JSON. Syntaxe je totiž velmi podobná SQL dotazům i svou škálou selekce, filtrace, projekce atd. Takže pomocí nástroje lze uskutečnovat i základní transformace [7].

Následující příklad vrátí pole titulů knih od autora Bohumila Hrabala s počtem stránek vyšším než 42.

```
let $books = json-file("example-chapter-one.json").library.books
return [
    for $book in $books[]
    where $book."author" = "Bohumil Hrabal" \
           and $book."page-count" gt 42
    return $book."book-title"
]
```

Nástroj si lze vyzkoušet online[4].

2.1.6 JMESPath

Dalším dotazovacím jazykem nad formátem JSON s možnostmi základní transformace je JMESPath. Pro přístup k potomkům objektu používá výlučně "tečkovou notaci"a je velmi podobný jazykům JSONPath Plus a JSPath. Co ale přidává navíc, jsou tzv. pipes (symbol |), které používá pro řetězení dotazů za sebe. To například umožnuje vrátit výsledek dotazu zasazený do strukturovaného JSON objektu. Pro přistoupení k prvkům pole používá hranaté závorky. [8]

Ukažme znovu příklad s filtrováním autora a počtu stránek.

```
library.books[?author==’Bohumil Hrabal’ \
    && ?page-count > ’42’].book-title
```

[3] https://github.com/dfilatov/jspath

[4] https://colab.research.google.com/github/RumbleDB/rumble/blob/master/RumbleSandbox.ipynb

2.2 Transformační nástroje

Následující nástroje slouží primárně pro transformaci hierarchických dat v rámci jednoho formátu. Převod do libovolného textového dokumentu nevnímáme jako plnohodnotnou transformaci napříč formáty, a tak jsme takové nástroje také umístili do této sekce.

2.2.1 XSLT

Extensible Stylesheet Language Transformations (XSLT) je velmi rozsáhlý transformační jazyk navržený primárně pro transformace z XML dokumentů do XML dokumentů. Výstupem však může být v podstatě jakýkoliv textový dokument, tj. i JSON, RDF, CSV, atd.

Každá XSLT transformace je validní XML dokument, který musí obsahovat tzv. stylesheet kořenový element (`<xsl:stylesheet>`) s atributem verze XSLT. Stylesheet element poté obsahuje typicky více šablonových elementů, které určují formát výstupu. Elementy šablony se značí `<xsl:template>`. Šablonové elementy používají dotazovací jazyk XPath, představený v podsekci 2.1.1, pro vyhledání odpovídajících dat ve vstupním XML dokumentu. Šablony využívájí princip "namatchování"se (atribut `match`) na příslušné XML elementy, na které se poté aplikuje daná transformace definovaná v šabloně. "Namatchovaný"element slouží jako kontext pro ostaní XPath dotazy uvnitř šablony [4].

Na následujícím příkladu vidíme transformaci z dat XML dokumentu do jednoduché HTML stránky s tabulkou knih.

```
<?xml version="1.0" encoding="UTF-8"?>

<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <html>
  <body>
  <h2>Library books</h2>
  <table border="1">
    <tr bgcolor="#9acd32">
      <th>Title</th>
      <th>Author</th>
    </tr>
    <xsl:for-each select="library/books/book">
    <tr>
      <td><xsl:value-of select="book-title"/></td>
      <td><xsl:value-of select="author"/></td>
    </tr>
    </xsl:for-each>
  </table>
  </body>
  </html>
</xsl:template>
```

```
</xsl:stylesheet>
```

Šablony také lze zanořovat do sebe. XSLT podporuje i lokální a globální proměnné a generické šablony. Ve verzi XSLT 3.0 lze také transformovat vstupní dokument streamovaně a podporuje také i jiné formáty než XML, například CSV nebo JSON [9].

2.2.2 jq

Nástroj jq na transformace formátu JSON, umožňuje uživatelům jednoduše zpracovávat dokumenty přímo na příkazové řádce. Jq používá pro příkazovou řádku přirozený symbol `|` pro řetězení dotazů. Obecně je jak z hlediska syntaxe dotazování, tak z hlediska možností transformací velmi podobný výše zmíněnému nástroji JMESPath. Nástroj dále podporuje streamování vstupních dat [10].

```
jq '.library.books[] | {name: .book-title}'
```

2.2.3 jj

Jj je utilita na příkazovém řádku umožnující streamované transformace vstupu ve formátu JSON. Používá se k vybírání části vstupu, přidání JSON objektů před vstup, nahrazení nebo smazání konkrétních hodnot nebo rozšíření polí o hodnoty a objekty. Nástroj používá přepínače a optiony k vyjádření jakou transformaci uživatel chce provést. Dotazování využívá "tečkovou notaci"pro přístup k potomkům objektu [11].

Následující příklad přepínačem `-v` (podle value) přepíše prvku na indexu 1 v poli `books` hodnotu `book-title` na `GenerickyNazev`.

```
jj -v GenerickyNazev library.books.1.book-title
```

2.2.4 jl

JSON Lambda je malý funkcionální jazyk pro vyhledávání a manipulování s JSONem.

Pro lambda výrazy se používají šipky (`\x -> y`), pro funkce `nazev_funkce parametr1 parametr2`, např. pro přístup ke klíči k objektu o (`get "k" o`), přičemž tuto konkrétní funkci lze napsat i zkráceně "tečkovou notací"(`o.f`). Dále pak lze využít ve funkcích řídící struktury `if x then y else z` a pro řetězení funkcí se využívají pipes (symbol .|.) [12].

Následuje příklad jednoduchého mapování vstupu na výstup.

```
jl 'map $ \o -> {library-name: o.library.name}'
```

2.2.5 Jolt

Jolt je transformační nástroj pro formát JSON, který transformace vyjadřuje deklarativně také pomocí JSON dokumentu. Zajímavé na něm je, že nepoužívá při transformaci šablony, jako jiné transformační jazyky v této kategorii, ale transformuje data tzv. "od vstupu k výstupu" [13].

Pomocí operací, které mají každá vlastní doménově specifický jazyk zápisu specifikace, nástroj postupně transformuje data přímo z podoby vstupu do cílené podoby na výstup [13].

Transformace funguje iterativně, kdy se postupně jedna po druhé aplikují jednotlivé operace na vstupní dat. Výstup předchozí operace je vstup té následující a výstup poslední je poté celkový výstup transformace [13].

Kostra transformace poté vypadá třeba takto.

```
[
  {
    "operation": "shift",
    "spec" : { // specifikace pro operaci shiftr  }
  },
  {
    "operation": "sort"  //
  },
  ...
]
```

2.2.6 JSLT

JSLT je dotazovací a transformační jazyk formátu JSON inspirovaný nástroji jako dříve zmíněné jq, XPath a XQuery. Pro přístup ke hodnotám objektu využívá "tečkovou notaci"a indexování do pole zprostředkovávají typické hranaté závorky (symboly `[]`). Validní JSON dokument uvnitř transformačního výrazu je ponechán na místě. Dále jazyk podporuje funkce (jako např. `size()`, `string()` aj.), proměnné, řídící struktury a řetěžení výrazů pomocí pipes (symbol `|`) [14].

Příklad jednoduché transformace můžeme vidět zde.

```
[for (.library.books)
  {
    "nazev" : .title ,
    "pocet-stran" : .page-count
  }
]
```

Nástroj si lze také vyzkoušet online[5].

[5] `https://www.garshol.priv.no/jslt-demo`

2.2.7 JSONata

JSONata je další z transformačních jazyků formátu JSON. Inspirovaný XPath 3.1, klade důraz na kompaktní a intuitivní zápis transformace, takže i u něho najdeme "tečkovou notaci"pro přístup k hodnotám a typický přístup k prvkům pole pomocí hranatých závorek (symboly `[]`). Přímo do hranatých závorek lze také místo indexu prvku psát predikáty pro filtrování obsahu. Jazyk také podporuje využívání funkcí, a to i uživatelsky definovaných [15].

Uveďme jednoduchý příklad.

```
library.books[author='Laszlo Krasznahorkai'].title
```

Nástroj lze také vyzkoušet online[6].

JSON Transforms

Posledním zajímavým transformačním nástrojem pro formát JSON je nástroj JSON Transforms. Svým rekurzivním přístupem a "matchováním"JSON dat připomíná jazyk XSTL zmíněný výše. Podobně jako XSLT využívá dotazovací jazyk XPath, využívá JSON Transforms pro dotazování JSPath (také zmíněný výše).

Nástroj je psaný pro JavaScript a jeho použití spočívá v napsání jednotlivých transformací do pole, které se následně společně se vstupním JSONem předá funkci transform, která provede samotnou transformaci. Nejzákladnější transformací je identita (v příkladu níže `jsont.identity`), kterou je dobré využít na konci každého pole transformací. Zajistí totiž, že se každý objekt projde alespoň jednou a zavolá se na něm rekurzivně pole transformací. Za zmínku v příkladu níže stojí funkce `runner`, pomocí které se spouští rekurze, jež umožňuje větší složitou transformaci rozdělit na více jednodušších [16].

Specifikace transformace může vypdat například takto.

```
jsont = require('json-transforms');
const rules = [
  jsont.pathRule(
    '.authors{.name === "Laszlo Krasznahorkai"}', d => ({
      Recent books: d.runner()
    })
  ),
  jsont.pathRule(
    '.books{.year > 2000}.title', d => ({
        title: d.context.title
    })
  ),
  jsont.identity
];
```

[6] https://try.jsonata.org/

```
var transformed  = jsont.transform(json, rules);
```

2.3 Nástroje pro transformaci napříč formáty

Nástroje a jazyky sloužící primárně k mapování jednoho hierarchického formátu na druhý a případně k jeho transformaci jsou jako poslední stručně představeny v této podkapitole. U každého také ukážeme jednoduchý příklad pro ilustraci.

2.3.1 JSON/CSV Cruncher

JSON/CSV Cruncher je nástroj, který umožnuje vykonávat SQL `Select` dotazy, jak nad CSV, tak nad JSON dokumenty a výsledek uložit do jakéhokoliv z obou formátů. Na oba formáty se totiž dívá jako na SQL tabulky. Při vybrání celého dokumentu nám nástroj umožňuje převést jeden formát do druhého, a to i bez dodatečných schémat [17].

Na příkazové řádce může použití JSON/CSV Cruncheru vypadat následovně.

```
cruncher/crunch -in authors.json -out recent_books.csv \
   -sql 'SELECT books.title \
         FROM $table \
         WHERE name = 'Laszlo Krasznahorkai'  \
         ORDER BY books.year DESC LIMIT 3'
```

2.3.2 RML

Pomocí RDF Mapping Language specifikujeme mapování mezi formáty jako např. JSON, CSV, XML a formátem RDF. Jedná se pouze o specifikaci bez implementace. Mapa trojic (`Triples Map`) obsahuje definice pravidel vždy pro jeden subjekt RDF trojic. Je psaná jako Trutle dokument, což znamená, že se jedná o validní RDF. Skládá se ze třech hlavních částí, `Logical Source`, `Subject Map` a `Predicate Object Map` [18].

Logical Source musí také obsahovat tři části, kterými jsou odkaz na zdroj vstupních dat (může jich být i víc), tzv. `Reference Formulation`, která určuje jakým způsobem se bude dotazovat nad daty a iterátor, který určuje jak se budou vstupní data procházet [18].

Logical Source může vypadat například takto.

```
<#AuthorMapping>
  rml:logicalSource [
    rml:source "Authors.json";
    rml:referenceFormulation ql:JSONPath;
    rml:iterator "$.[*].Authors" ].
```

Subject Map obsahuje vzor URI pro každý subjekt a volitelně jeho typ. Tedy například.

```
<#AuthorMapping>
  rr:subjectMap [
    rr:template "http://ex.com/Author/{name}_{surname}";
    rr:class ex:Author ].
```

Výsledné trojice jsou generovány díky Predicate Object Map, která sestává z predikátu a objektu definované následujícím způsobem.

```
<#AuthorMapping>
  rr:predicateObjectMap [
    rr:predicate ex:name;
    rr:objectMap [ rml:reference "name" ] ].
```

2.3.3 JSON-LD

JSON-LD (JSON for Linked Data, česky JSON pro Propojená data) je specifikace, která umožňuje namapovat JSON data na formát RDF. Lze říci, že JSON-LD je jedna ze serializací RDF [19].

JSON-LD si rezervuje klíče `@context`, `@id`, `@type` a další jako klíčová slova s významem, která používá pro mapování. Abychom z JSON dokumentu udělali JSON-LD dokument, přidáme do entit klíč `@context` s mapováním všech původních klíčů na jednoznačné URL s jejich významem (nebo na `Null` pokud chceme klíč z mapování na RDF vyřadit) [19]. V případě potřeby doplníme patřičná klíčová slova, kterými se lépe propojí naše data s ostatními datasety. Tento způsob převodu je pro vývojáře příjemný, protože vyžaduje jen malou modifikaci již existujících JSON dokumentů pro vytvoření propojených dat.

```
{
  "@context": {
    "name": "https://example-vocab.net/name"
    "born": "https://example-vocab.net/born"
    "spouse": "https://example-vocab.net/spouse"
  },
  "@id": "https://dbpedia.org/page/Laszlo_Krasznahorkai",
  "name": "Laszlo Krasznahorkai",
  "born": "1954-01-05",
  "spouse": "Dóra Kopcsányi"
}
```

2.3.4 CSVW

CSVW (CSV on the Web) přináší specifikaci potřebnou pro anotaci popř. transformaci tabulkových dat do propojených dat, jako je například RDF. Specifikace podporuje i mapování do jiných formátů jako jsou XML či JSON. K anotaci CSV tabulek se využívá tzv. JSON-LD deskriptor, který se následně distribuuje spolu s tabulkami. Jedná se skutečně o validní JSON-LD dokument zmíněný v předchozí podsekci 2.3.3. Musí tedy obsahovat alespoň klíč `@context` s odkazem na kontext CSVW [20].

Takže například.

```
{
    "@context": "http://www.w3.org/ns/csvw",
    "url": "https://example.net/authors.csv"
}
```

V případě, že chceme přidat k tabulkovým datům dokumentaci pro potenciální transformaci do RDF nebo JSONu, musí JSON-LD deskriptor obsahovat více informací o struktuře dat. Například informace o používaném jazyku, schématu skupiny tabulek nebo samotných tabulkách (např. názvy sloupců a typy hodnot v jednotlivých buňkách). K tomu slouží např. klíče `tableSchema`, `colums`, `datatype` atd. [20]

Jednoduché schéma tabulky může vypadat třeba takto.

```
{
    "@context": [
        "http://www.w3.org/ns/csvw",
        {"@language": "en"}
    ],
    "@type": "Table",
    "@id": "https://example.org/authors",
    "url": "https://example.net/authors.csv",
    "tableSchema": {
        "columns": [{
            "name": "name",
            "titles": {
                "en": ["name", "fullName"],
                "cs": "jméno"
            },
            "datatype": "string"
        }, {
            "name": "born",
            "titles": {
                "en": "born",
                "cs": "narozen"
            },
            "datatype": "datetime"
        }]
```

```
        }
    }
```

2.4 Porovnání nástrojů

V této sekci bychom představené transformační nástroje stručně porovnali v relevantních kritériích a vyvodili z této kapitoly závěry pro náš navrhovaný transformační nástroj. Kritéria byla vybírána na základě toho, co nám přišlo důležité vzhledem k tématu práce. Nástroje porovnáváme ve formě tabulek v rámci svých kategorií a následně společně vyvodíme závěry.

Dotazovací jazyky

Následující kriteria dotazovacích jazyků srovnává Tabulka 2.1.

Umožňuje základní transformace - zda-li jazyk podporuje základní transformace dat jako jeich filtraci a další podobně složité funkce.

Vlastní datový model - zda-li jazyk vytváří pro dotazování vlastní datový model jako abstrakci konkrétního formátu.

SQL-like - využívá-li jazyk při zápisu dotazování syntaxi SQL nebo syntaxi SQL podobnou.

Transformační nástroje

Následující kriteria tranformačních nástrojů srovnává Tabulka 2.2.

Jednoduché základní transformace - zde hodnotíme jestli je nástroj jednoduchý na používání pro základní transformace (např. vypsání hodnot při známé cestě a podobně).

Komplexní transformace - zda-li nástroj umožňuje komplexní transformace vstupních dat typu změna jednoho strukturovaného dokumentu na jiný strukturovaný dokument využívající stejná data.

Nástroj	Základní transformace	Vlastní datový model	SQL-like
XPath	✓	✓	×
JSONPath (Plus)	✓	×	×
JSON Pointer	×	×	×
JSPath	✓	×	×
JSONiq	✓	✓	✓
JAMESPath	✓	×	×

Pozn: ✓ → *splňuje*, × → *nesplňuje*, − → *částečněsplňuje*, ? → *nelzerozhodnout*

Tabulka 2.1: Srovnání dotazovacích jazyků v relevantních aspektech.

Deklarativní - zda-li je způsob zapsání transformace spíše deklarativní.

Impertivní - zda-li je způsob zapsání transformace spíše imperativní.

Streamování - umožňuje-li nástroj transformovat vstupní data i bez nutnosti je celá předem načíst do operační paměti (užitečné primárně při transformaci velkých vstupních nebo dat, která se teprve generují v průběhu výpočtu).

Komentáře - umožňuje-li nástroj k transformacím psát komentáře.

Nástroje pro transforamci napříč formáty

Nástroje pro transformaci napříč formáty srovnává Tabulka 2.3 v následujících kriteriích.

Podporuje XML - umožňuje-li nástroj transformovat vstupní data z a do formátu XML.

Podporuje JSON - umožňuje-li nástroj transformovat vstupní data z a do formátu JSON.

Podporuje RDF - umožňuje-li nástroj transformovat vstupní data z a do formátu RDF.

Podporuje CSV - umožňuje-li nástroj transformovat vstupní data z a do formátu CSV.

Nutnost schématu - umožňuje-li nástroj transformovat vstupní data napříč zvolenými formáty bez nutnosti dodání informací nad rámec samotných dat (schéma, atd.)

SQL-like - využívá-li nástroj při zápisu transformace syntaxi SQL nebo syntaxi SQL podobnou.

Nástroj	Jednoduché základní transform.	Komplexní transform.	Deklarativní	Imperativní	Streamování	Komentáře
XSLT	-	✓	✓	×	✓	✓
jq	✓	✓	✓	×	✓	×
jj	✓	✓	-	-	✓	×
jl	✓	✓	✓	×	?	×
Jolt	✓	✓	✓	×	×	×
JSLT	-	✓	-	-	?	✓
JSONata	✓	✓	✓	×	?	✓
JSON Transforms	-	✓	✓	-	?	✓

Pozn: ✓→ splňuje, × → nesplňuje, – → částečněsplňuje, ? → nelzerozhodnout

Tabulka 2.2: Srovnání transformačních nástrojů v relevantních aspektech.

Nástroj	Podporuje XML	Podporuje JSON	Podporuje RDF	Podporuje CSV	Nutnost schémat pro transform.	SQL-like
JSON/CSV Cruncher	×	✓	×	✓	×	✓
RML	✓	✓	✓	✓	✓	×
JSON-LD	×	✓	✓	×	✓	?
CSVW	×	×	✓	✓	✓	×

Pozn: ✓→ splňuje, × → nesplňuje, – → částečněsplňuje, ? → nelzerozhodnout

Tabulka 2.3: Srovnání nástrojů pro transformaci napříč formáty v relevantních aspektech.

Závěry

Z kapitoly i srovnání vyplývá, že existuje mnoho nástrojů na transformaci hierarchických dat v rámci jednoho formátu. Také existuje mnoho mapování formátů XML, JSON i CSV na formát RDF. Mapování opačným směrem z RDF do zmíněných formátů nebo mezi zmíněnými formáty ale běžná nejsou. Většina z nástrojů pro převod mezi formáty navíc uživatele nutí poskytnout nějaký typ schématu.

Mezi transformačními nástroji jsme si mohli všimnout převahy deklarativního způsobu vyjádření transformace. Podpora komentářů u transformace či podpora streamování příliš častá mezi nástroji není. Dále jsme si mohli všimnout, že XSLT je velmi mocný transformační nástroj a množství transformačních nástrojů z něho vychází. Jeho hlavní problémy ale jsou prioritizace formátu XML (transformuje se z XML), poměrně velká složitost i jednoduchých transformací a výstup transformace založený na textu.

Transformační jazyky se při transformaci potřebují dotazovat nad daty ze vstupu. Časté bylo tuto potřebu uspokojit nějakým již existujícím dotazovacím jazykem. Pro zápis transformace velká část nástrojů využívá šablony, do kterých pomocí dotazovacích jazyků dosazuje data. Využívají při tom pokročilých schopností dotazovacích jazyků, data ze vstupu již předfiltrovat nebo jinak předtransformovat. V tomto je specifický nástroj Jolt, který k transformaci přistupoval trochu jiným způsobem tzv. "od vstupu k výstupu". Nepoužívá šablony, do kterých pomocí dotazovacích jazyků filtruje data, ale přistupuje k transformaci opačně od dat vstupních. Také nám u Joltu přišla zajímavá flexibilita rozdělení transformace na operace s vlastním doménově specifickým jazykem.

U samotných dotazovacích jazyků je zajímavé, jak moc se přibližují k transformačním jazykům svými schopnostmi. I když SQL je používaný jazyk u relačních dat, u hierarchických jeho způsob zápisu zas tak častý není.

V následující kapitole navážeme na poznatky z této sekce analýzou požadavků na náš transformační nástroj a jeho části.

3. Analýza požadavků

V této kapitole se pokusíme na základě cílů práce a znalostí nabytých v minulé kapitole definovat požadavky, které by měl náš návrh splňovat. Zejména se zamyslíme, jak nejlépe na sebe zvolené formáty navzájem mapovat, abychom v souladu se zadáním práce, zajistli co největší kontrolu. Dále zhodnotíme přístupy nástrojů k transformaci a představíme ve většímm detailu nástroj, který autory této práce zaujal svým netradičním přístupem. Nakonec se zamyslíme nad možnostmi zápisu dotazovacích jazyků.

3.1 Reprezentace formátů

V otázce mapování formátů máme v zásadě tři možnosti. Řešení, které nás napadne jako první je, že můžeme všechny formáty namapovat na jeden společný formát, se kterým bychom poté pracovali při transformacích. Tento přístup má nespornou výhodu prakticky úplné unifikace, tedy sjednocení. Následné transformace by také byly generické pro všechny mapované formáty, což je velmi pozitivní vlastnost. Problémy ale nastávají, když jednotlivé formáty mají výrazné odlišnosti, nebo dokonce úplně jiné datové modely. Připomeňme, že formáty se kterými pracujeme mají datový model hierarchický v případě XML a JSON, relační v případě CSV a grafový v případě RDF. Namapovat tyto odlišné datové modely do jediné sdílené reprezentace, by bylo velmi obtížné a určitě bychom v procesu přišli o část specifik jednotlivých formátů.

Další možnost by bylo téměř rezignovat na společné mapování a umožnit nástroji výstup do libovolného textu. Podobně to dělá např. nástroj XSLT viz sekce 2.2.1. Tím bychom více než uspokojili požadavek na kontrolu nad formátem výstupu, neboť bychom snadno kontrolovali i odsazení atd., ale zaplatili bychom za to mimo jiné složitým vymáháním korektnosti výstupných formátů a slabého využití společných rysů daných formátů.

Třetí možností je nemapovat formáty přímo na jeden společný formát, ale umožnit jim různá mapování v rámci nějaké společné reprezentace. To znamená, že by mapování sice zůstala specifická pro jednotlivé formáty, ale reprezentace tohoto mapování by byla společná. Nad touto společnou reprezentací by poté mohl fungovat náš transformační nástroj. Výhoda je, že jsme tímto způsobem schopni reprezentovat specifika jednotlivých formátů a pracovat s nimi dále při transformaci. A tím si vlastně udržet velkou kontrolu nad výsledkem i se zachováním možnosti kontroly správnosti daných formátů, pomocí porovnání s předem stanoveným mapováním. Nevýhodou je, že transformace nebudou moci být stejné pro všechny mapované formáty, což by nám zvýšilo složitost transformací a omezilo znovupoužitelnost již napsaných specifikací transformace.

První z přístupů pro nás není příliš praktický, neboť máme příliš odlišné datové modely, druhý přístup by nám při transformaci nedával přiliš jistot a navíc ho už používá populární XSLT. Pro naši práci tedy využijeme třetí z možností.

Tento přístup nám také při správném návrhu mapování umožní definovat (polo)automatické převody mezi formáty, což bychom také od naše nástroje chtěli.

3.2 Transformace

U transformačních nástrojů jsme v minulé kapitole identifikovali množství relevantních aspektů. V této sekci se pokusíme vybrat vhodné vlastnosti pro náš transformační nástroj.

První vyberme mezi deklarativním, nebo imperativním přístupem k zápisu transformace. V deklarativním přistupu se soustředíme zejména na to, co se má stát, a ne nutně jak se to má stát. V přístupu imperativním naopak programaticky popisuje jak se mají data projít a transformovat. V srovnávací tabulce 2.2 jsme mohli vidět, že většina nástrojů je deklarativních a i autorům této práce tento přístup připadá pro transformace dat vhodnější.

Dále bychom po našem nástroji chtěli, aby byl flexibilní a rozšiřitelný a zvládal i komplexní transformace. Na druhou stranu bychom byli rádi, kdyby se podařilo navrhnout nástroj dostatečně intuitivně, aby i nováčci byli schopni poměrně rychle vykonat jednoduché transformace.

I když streamování by bylo dobré mít, uvědomujeme si, že to může omezovat návrh a přidávat složitost implementaci, takže na něm netrvnáme. Na čem ale trváme, je podpora komentářů, neboť pomocí komentářů lze jednoduše vyjasnit spoustu rozhodutí při transformaci.

Posledním aspekt, který nás bude zajímat, je způsob přístupu k transformaci. V minulé kapitole jsme zaznamenali zejména dva odlišné přístupy. První z nich by se dal označit jako "od vstupu k výstupu", protože transformace vychází z dat na vstupu a nějakým způsobem se k nim staví - např. je maže nebo posouvá na jiné místo atd. Druhý přístup, řekněme "od výstupu ke vstupu", začíná od struktury výstupních dat, kterou popisuje a do zvolených klíčů dosazuje, pomocí dotazovacího jazyka, hodnoty ze vstupu. Nám přijde mnohem přirozenější a intuitivnější přístup "od vstupu k výstupu"a chtěli bychom jeho využití v této práci prozkoumat.

Inspiraci čerpáme i z nástroje Jolt pro formát JSON, který dokonce ve své spcifikaci celý vstup opisuje. Podívejme se na něj z blízka v následující sekci.

3.3 Inspirace nástrojem Jolt

Transformační nástroj Jolt nám slouží jako nemalá inspirace pro náš návrh, tudíž mu zde věnujeme celou sekci. Autorstvo Joltu nástroj popisuje jako transformační knihovnu pro formát JSON psanou v jazyce Java, u kterého je specifikace transformace také JSON dokument [13]. Zápis transformace samotný je tedy validní JSON dokument, což naznačuje, že se jedná o deklarativní transformační jazyk.

Popis pokračuje výčtem hlavních charakteristik, jako jsou nástrojem poskytované jednodušší transformace, které lze **řetězit za sebe** do transformací složitějších, **soustředění se na strukturu** dat místo konkrétních hodnot a práce výhradně s už načteným JSONem v paměti (jak doplňování hodnot, tak deserializaci a serializaci si uživatel má udělat sám) [13].

Nás na nástroji zaujala zejména **flexibilita** a **přehlednost** řetězitelných operací, které používají svůj **vlastní doménově specifický jazyk**. A přístup "**od vstupu k výstupu**", který tyto jazyky operací využívají.

Operace vlastně opisují strukturu vstupních dat a u relevantních entit provedou patřičnou transformaci, jako například přesun podstromu na jiné místo na výstup nebo smazání entity. Tento přístup, vlastně opačný k nástrojům jdoucím "od výstupu k vstupu"využívaný v jazycích jako např. XSLT [9] a jiných, může podle autorstva zjednodušit a zpřehlednit složitější transformace [13]. Jedná se tedy o méně rozšířený přístup k transformaci, který by ale měl mít velmi **podobnou sílu vyjádření** a potenciálně by mohl vést k **jednodušším zápisům** transformací.

Nástroj poskytuje sadu základních operací jako jsou `shiftr`, `default`, `remove`, `sort`, `cardinality`. U většiny z nich lze podle názvu snadno odvodit co dělají. My se zde budeme stručně věnovat jen "hlavní"operaci `shift`. Píšeme, že jde o hlavní operaci, neboť zodpovídá za největší strukturální změny v transformaci. Přesouvá totiž ze vstupu na výstup hodnoty nebo celé podstromy [13].

Uvěďme jednoduchý příklad s následujícím vstupem.

```
{
  "library": {
    "name": "Open Library"
  }
}
```

Specifikace operace `shiftr` začne s opsáním celé struktury vstupu a místo hodnoty u klíče `name` zapíše cestu kam se má hodnota přesunout na výstup.

```
{
  "library": {
    "name": "output-object.library.called"
  }
}
```

Je nutné popsat celou strukturu, protože Jolt poté vykoná paralelní průchod specifikací i daty na vstupu, a když narazí ve specifikaci na cestu přesune hodnotu pomocí této cesty na výstup [13]. Výsledek je potom následující.

```
{
  "output-object": {
    "library": {
      "called": "Open Library"
    }
  }
}
```

Pouze takto by se ale pokaždé musel opsat **celý** vstup, což je jednak nepraktické pro velké vstupy a jednak nemožné v případě, že nevíme jak přesně bude vstup vypadat. Autorstvo tento problém řeší podporou tzv. **wildcards** neboli **divokých karet** [21] (podobné třeba těm na příkazovém řádku operačního systému linux).

Symbol `*` použitý ve specifikaci místo (části) klíče funguje jako proměnná, za kterou se může dosadit jakýkoli klíč (nebo jeho část) ze vstupu, který splňuje zbytek cesty. Na hodnotu dosazené proměnné se lze odkázat pomocí symbolu `&` s indexem podle toho, jak vzdálený v cestě je od entity kterou posouváme [21].

Stejného výsledku lze tedy dosáhnout i následující specifikací využívající divoké karty. Tato transformace by se ale narozdíl od té první vykonala na jakémkoli objektu v kořeni s klíčem `name`.

```
{
  "*": {
    "name": "output-object.&1.called"
  }
}
```

Už jen operace `shiftr` s divokými kartami podle autorstva Joltu dokáže uspokojit drtivou většinu požadovaných transformací [21]. Ještě zmíníme, že pokud nějaké hodnoty z vstupu na výstupu nechceme, tak je musíme přesunout do nějaké společné entity a tu poté pomocí operace remove z výstupu odstranit.

Na operaci `shiftr` by se dalo popsat ještě více detailů a funkcionality (např. použití pole výstupních cest místo jedné cesty atd.), ale to už není pro naši práci tolik podstatné.

Také doufáme, že si čtenářstvo snadno dokáže představit, jak přibližně fungují ostatní operace, které mají za úkol nastavit výchozí hodnotu entity nebo entitu ze vstupu odstranit. V zásadě je to totiž velmi podobné operaci `shiftr`, jen pod daným klíčem není cesta na výstup, ale třeba hodnota k nastavení nebo akce smazání.

Více informací k nástroji lze najít v readme [1] projektu. Nástroj si také lze vyzkoušet online [2].

Celkově tedy vidíme, že se jedná o netradiční a zajímavý nástroj, který ale funguje pouze pro formát JSON, a to ještě pouze pro jazyk Java. Také se možná čtenářstvu nemusí tolik zamlouvat opisování celého vstupu ve specifikaci a ne příliš intuitivní a přehledný zápis divokých karet.

3.4 Dotazovací jazyky

Transformační nástroje často potřebují způsob, jak se dotazovat nad právě transformovanými daty. K tomu využívají právě dotazovací jazyky. Dotazovací jazyky jako takové můžou být samy o sobě poměrně silné transformační nástroje, jak jsme viděli v minulé kapitole. Silné dotazovací jazyky typicky využívají nástroje

[1] urlhttps://github.com/bazaarvoice/jolt/blob/master/README.md

[2] `https://jolt-demo.appspot.com/#inception`

fungující "od výstupu ke vstupu", protože potřebují silný způsob, jak "matchovat"data ze vstupu. My ale budeme využívat přístup "od vstupu k výstupu"a pro takové transformační jazyky není typicky silný dotazovací jazyk potřeba. Stačí jim pouze jednoduchý ukazovací jazyk, jako je třeba JSONPointer, který bude schopen ukazovat do entit vstupu.

3.5 Shrnutí požadavků

Na závěr kapitoly shrňme požadavky na náš návrh. Jak bylo zmíněno v sekci 3.1, chtěli bychom využít různých mapování formátů do společné reprezentace, neboť nám to umožní zachovat kontrolu nad výstupem i se zachováním garancí korektního výstupu. Tuto společnou reprezentaci bychom chtěli transformovat pomocí jazyka, který je deklarativní, flexibilní a rozšiřitelný, podporuje komentáře, je silný a zároveň intuitivní a nemusí podporovat streamování. Dále by měl fungovat na principu "od vstupu k výstupu", protože nám tento přístup přijde přirozenější a přehlednější. Následně jsme si uvědomili, že díky tomuto přístupu nemusí být dotazovací jazyk využívaný při transformaci příliš silný,

4. Návrh

V této kapitole čtenářstvu přiblížíme návrh a funkcionalitu našeho transformačního jazyka a nástroje. Návrh stojí na třech stěžejních konceptech, které podrobně rozebereme v následujících podkapitolách. Prvním je Unifikovaná reprezentace hierarchických dat (zkráceně Ur) umožňující sjednocení vstupních a výstupních formátů i se zachováním jejich specifik. Zjednodušeně si lze Ur představit jako JSON s výhradně řetězcovými hodnotami. Druhým konceptem je ukazovací jazyk UrPath, který slouží k odkazování na existující i neexistující - právě tvořené - entity Unifikované reprezentace. Třetím konceptem je pak transformace pracující s Unifikovanou reprezentací na svém vstupu i výstupu hojně využívající ukazovací jazyk UrPath ve svých operacích. Pro celou transformaci tedy nejdříve data ve vstupním formátu převedeme do Ur, vykonáme transformaci a následně výstup exportujeme z Ur do zvoleného formátu.

4.1 Unifikovaná reprezentace

K sjednocení hierarchických datových formátů představujeme tzv. Unifikovanou reprezentaci neboli Ur. Primární motivací za Unifikovanou reprezentací je možnost mít kontrolu nad reprezentací datových modelů jednotlivých formátů a zároveň poskytnutí určitým způsobem jednotného prostředí transformacím dále v procesu. Kontrola pro jednotlivé formáty spočívá v možnosti různého mapování na Ur podle potřeb daného formátu. Jednotného prostředí je docíleno využíváním společných stavebních bloků typických pro hierarchické formáty, jako jsou entity s klíči a pole.

V následující podsekci definujeme Unifikovanou reprezentaci a vysvětlíme racionalitu za jejím návrhem. Dále potom na příkladech předvedeme jednotlivá mapování pro všechny hierarchické formáty, kterými se naše práce zabývá.

4.1.1 Definice Unifikované reprezentace

Jak už bylo řečeno výše, jednoduše si lze Ur představit jako JSON pouze s řetězcovými hodnotami.

Hlavními částmi Unifikované reprezentace jsou entity, pole a kompozitní hodnoty. Primitivní hodnoty jsou vždy řetězce a jejich typ je ukládán po boku hodnoty, také jako řetězec. Pro řídící značky, které jsou často specifické pro jednotlivý formát, používáme symbol `@`, přičemž inspiraci čerpáme z RDF serializace JSON-LD [19]. Pro hodnotu využíváme klíč `@value` a pro typ klíč `@type`. Hodnota každého klíče v Ur je vždy pole. Pole se využívají k uchování potenciálně více hodnot pro daný klíč entity. Pro jednotný přístup je každá hodnota obalena polem, aby mohla poteciálně mít více hodnot. Koncept polí tedy využíváme zejména pro jednotný přístup, a tudíž pro pole jako kolekce hodnot využíváme pro přehlednost entity. Takže entity v Ur reprezentují jak objekty, tak i kolekce hodnot. Objekty jsou kolekce dvojic klíč a hodnota. Podobně tak pole jsou reprezentované jako kolekce dvojic index a hodnota.

Striktně řetězcové hodnoty využíváme pro přesné zachování vstupní hodnoty. Při jakékoliv konverzi ze vstupu totiž může docházet k nechtěnému zkreslení. U

numerických hodnot např. k zakrouhlení nebo změně přesnosti. Tohoto se vyvarujeme právě zachováním přímo řetězcové hodnoty ze vstupu.

Veškerá funkcionalita hierarchických formátů bohužel jen takto jednoduše reprezentovat nejde, a proto i Unifikovaná reprezentace je ve skutečnosti specifická pro jednotlivé formáty a jejich potřeby.

4.1.2 Serializace Ur do formátu JSON

Přestože Unifikovaná reprezentace může být serializována i do jiných formátů, poměrně přímočaře pracuje s formátem JSON. Ur entity nám tvoří JSON objekty, Ur pole nám tvoří JSON pole a pro řídící značky využijeme konkrétní JSON klíče.

S touto serializací nyní budeme představovat mapování jednotlivých formátů do Unifikované reprezentace.

4.1.3 Mapování formátů do Ur

V této podsekci představíme mapování do Unifikované reprezentace a to konkrétně návrhy autorů této práce pro formáty JSON, XML, CSV a RDF Turtle. V souladu se zadáním práce se věnujeme právě těmto čtyřem typicky užívaným formátům. Upozorněme ještě čtenářstvo, že se jedná pouze o naše návrhy Ur pro tyto formáty. V případě potřeby lze navrhnout i více Unifikovaných reprezentací pro jeden a tentýž formát. Také lze navrhnout a následně pracovat s Unifikovanými reprezentacemi dalších hierarchických formátů, např. jiných serializací RDF.

V tuto chvíli platí pro všechna navržená mapování do Unifikované reprezentace, že Ur pole obsahuje vždy jen jednu hodnotu. Jak už bylo řečeno výše, hodnoty jsou uzavřené v Ur poli z důvodu sjednocení zápisu všech druhů entit pro obecné mapování. Dodržováním tohoto pravidla si ale můžeme například zjednodušit implementaci.

Mapování mají také společné to, že primitivní hodnoty jsou entitami s řídící značkou `@type` a `value`. Jejich podporované typy už se ale můžou lišit. Také jsme v mapováních sjednotili skutečnost, že každá entita má typ - mimo primitivní hodnoty typicky `object` a `array`.

Začněme prezentaci konkrétních mapování formátem JSON.

JSON

Mapování formátu JSON je poměrně jednoduché a nijak nevybočuje z již nastíněných společných vlastností mapování. Objekty jsou mapované jako Ur entity s typem `object`. Primitivní hodnoty jako Ur entita s typem a hodnotou. Pole jsou mapována jako Ur entita s řetězcovými indexy, číslovanými od hodnoty nula.

Předveďme mapování pro následující JSON dokument zobrazující část JSON dokumentu ze sekce 1.

```
{
  "library": {
    "name": "Open Library",
    "books": [{
      "attributes": {
```

```
        "condition": "good"
      }
      "book-title": "Příliš hlučná samota",
      "page-count": 98
    }]
  }
}
```

Mapování do Ur pak vypadá následovně.

```
{
  "@type": ["object"],
  "library": [{
    "@type": ["object"],
    "name": [{
      "@type": ["string"],
      "@value": ["Open Library"]
    }],
    "books": [{
      "@type": ["array"],
      "0": [{
        "@type": ["object"],
        "attributes": [{
          "@type": ["object"],
          "condition": [{
            "@type": ["string"],
            "@value": ["good"]
          }]
        }],
        "book-title": [{
          "@type": ["string"],
          "@value": ["Příliš hlučná samota"]
        }],
        "page-count": [{
          "@type": ["number"],
          "@value": ["98"]
        }]
      }]
    }]
  }]
}
```

XML

V zásadě vypadá mapování velmi podobně jako mapování pro JSON, ale máme zde pár rozdílů. Pro XML hlavičku zavádíme dvě speciální nepovinné řídící značky `@version` a `@encoding`. Ty musí být v kořenu dokumentu a mapují se jako primitivní řetězcová hodnota, tedy jako Ur entita s typem `string` a hodnotou. Takové

mapování jsme zvolili z důvodu v zásadě automatického mapování na primitivní řetězcovou hodnotu u jiných formátů, se kterou lze pak dále pracovat v operacích, které představíme v sekci 4.3.2.

Další malý rozdíl je v zápisu polí. Formát XML pole přímo nemá a využívá pro ně objekty s více entitami stejné značky. Dva klíče stejné hodnoty ale naopak nedovoluje v jednom objektu formát JSON, který používáme jako hostující formát pro Unifikovanou reprezentaci. V takových situacích sdružíme tyto prvky do indexovaného pole pod jeden klíč s názvem značky v pořadí, v jakém byly v mateřském elementu na vstupu.

Posledním rozdílem je řídící značka `@attributes`, která umožňuje do Ur zapsat atributy pomocí objektu s dvojicemi `atribut` a `hodnota`.

Předveďme mapování Unifikované reprezentace na následující části XML dokumentu ze sekce 1.

```
<?xml version="1.0" encoding="UTF-8"?>
<library>
  <name>Open Library</name>
  <books>
    <book condition="good">
      <book-title>Příliš hlučná samota</book-title>
    </book>
    <book>
      <book-title>Nesmrtelnost</book-title>
    </book>
  </books>
</library>
```

Mapování potom bude následující.

```
{
  "@version": [{ "@type": ["string"], "@value": ["1.0"] }],
  "@encoding": [{ "@type": ["string"], "@value": ["UTF-8"] }],
  "library": [{
    "@type": ["object"],
    "name": [{
      "@type": ["string"],
      "@value": ["Open Library"]
    }],
    "books": [{
      "@type": ["object"],
      "book": [{
        "@type": ["array"],
        "0": [{
          "@attributes": [{
            "condition": [{
              "@type": ["string"],
              "@value": ["good"]
```

```
            }]
          }],
          "@type": ["object"],
          "book-title": [{
            "@type": ["string"],
            "@value": ["Příliš hlučná samota"]
          }]
        }],
        "1": [{
          "@type": ["object"],
          "book-title": [{
            "@type": ["string"],
            "@value": ["Nesmrtelnost"]
          }]
        }]
      }]
    }]
  }
```

Uvědomme si rozdíl mezi minulým příkladem JSON mapováním do Ur, kde klíč `attributes` byl pouze klasický klíč pro JSON objekt, a ne řídící značka `@attributes` jako zde. Dokážeme si ale jednoduše představit, jak snadno půjde mezi jednotlivými reprezentacemi převádět.

Upozorněme také, že narozdíl od JSONu musí mít validní Ur pro XML, mimo řídící značky `@encoding` a `@version`, pouze jeden kořenový element (klíč) dokumentu. Toto je důležité zejména pro převod z Unifikované reprezentace, která například vznikla transformací z Ur mapování jiného formátu, zpátky do formátu XML.

CSV

Narozdíl od formátů JSON a XML je CSV spíše relační nežli hierarchický formát. To nám ale nevadí a můžeme ho namapovat do Unifikované reprezentace stejně. Řádky mapujeme jako entity s klíči z hlavičky tabulky a jejich příslušnými primitivními hodnotami. Řádky mapujeme do entity s indexy řádků. Vytvoříme tak vlastně pole řádků, a tak entitě přidáme typ `array` pro snazší převádění do zbylých formátů. Pro toto pole zavádíme novou řídící značku `@rows`. Z celé hlavičky ještě uděláme entitu typu pole s řetězcovými primitivními hodnotami obsahujícími tokeny hlavičky u příslušných indexů. A tuto entitu uschováme pod nový speciální klíč `@header`. Nakonec, znovu pro snazší mapování na ostatní formáty, přidáme do kořenové entity klíč typu s hodntou `object`.

Ukažme mapování následujícího CSV dokumentu.

```
library,book-title
Open Library,Příliš hlučná samota
Open Library,Nesmrtelnost
```

Mapování potom vypadá takto.

```
{
  "@type": ["object"],
  "@header": [{
    "@type": ["array"],
    "0": [{
      "@type": ["string"],
      "@value": ["library"]
    }],
    "1": [{
      "@type": ["string"],
      "@value": ["book-title"]
    }]
  }],
  "@rows": [{
    "0": [{
      "@type": ["object"],
      "library": [{
        "@type": ["string"],
        "@value": ["Open Library"]
      }],
      "book-title": [{
        "@type": ["string"],
        "@value": ["Příliš hlučná samota"]
      }]
    }],
    "1": [{
      "@type": ["object"],
      "library": [{
        "@type": ["string"],
        "@value": ["Open Library"]
      }],
      "book-title": [{
        "@type": ["string"],
        "@value": ["Nesmrtelnost"]
      }]
    }]
  }]
}
```

RDF Turtle

Mapování RDF Turtle bylo trochu složitější, neboť se nejedná o hierachický datový model, ale jde o model grafový, jako takový nemá žádný kořen. Pro mapování jsme buďto mohli zvolit nějaký uzel jako kořen a nebo vytvořit nějaký "umělý"kořenový element, podobně jako tomu bylo u formátu CSV. Zvolili jsme druhou z možností.

V kořenové entitě jsou uchovány zdroje vyskytující se v RDF trojici na pozici subjektu. Zdroje tvoří Ur entity a predikáty jsou v nich uloženy jako klíče. Pod klíči musíme být schopni uložit všechno, co se může vyskytovat v RDF trojici na pozici objektu (zde nemyslíme objekt jako entitu, ale objekt z hlediska RDF trojice, viz sekce 1). To znamená primitivní hodnotu, anonymní objekt, jiný identifikovatelný zdroj a nebo pole všech zmíněných.

Převod primitivní hodnoty je totožný s ostatními formáty tedy kompozitní řetězcovou hodnotou s typem. Uveďme jednoduchý příklad s následujícím vstupem.

```
<https://example.net/library>
  <https://example.net/name> "Open Library" .
```

Výstup bude takovýto.

```
{
  "@type": ["object"],
  "https://example.net/library": [{
    "@type": ["object"],
    "https://example.net/name": [{
      "@value": ["Open Library"],
      "@type": ["http://www.w3.org/2001/XMLSchema#string"]
    }],
    "@id": [{
      "@value": ["https://example.net/library"],
      "@type": ["http://www.w3.org/2001/XMLSchema#string"]
    }]
  }]
}
```

Pokračujme s identifikovatelným zdrojem na pozici objektu v RDF trojici. Ten se namapuje jako Ur entita s typem `objekt` a novou řídící značkou `@id`. Řídící značka `@id` reprezentuje identifikátor zdroje a je uložená jako řetězcová primitivní hodnota pro jednodušší automatický převod do jiných formátů. Mají ji všechny neanonymní zdroje a tedy i ty v kořenové entitě.

Značka `@id` zároveň slouží jako odkaz v rámci dokumentu i mimo něj. Pokud se v něm zdroje vyskytují i samostatně, budou zmíněny jako samostatný klíč v kořenové entitě i se svými predikáty. Tímto způsobem vlastně reprezentujeme graf v jednodušší hierarchické struktuře, přičemž zmiňme, že jsme čerpali inspiraci v tzv. Flattened reprezentaci u JSON-LD [22].

Ilustrujme tento případ následujícím vstupem.

```
<https://example.net/library>
  <https://example.net/name> "Open Library" ;
  <https://example.net/friend> <https://example.net/bookstore> .
```

```
    <https://example.net/bookstore>
      <https://example.net/name> "Open Bookstore" .
```

S následujícím výstupem.

```
{
  "@type": ["object"],
  "https://example.net/library": [{
    "@type": ["object"],
    "https://example.net/name": [{
      "@value": ["Open Library"],
      "@type": ["http://www.w3.org/2001/XMLSchema#string"]
    }],
    "https://example.net/friend": [{
      "@id": [{
        "@value": ["https://example.net/bookstore"],
        "@type": ["http://www.w3.org/2001/XMLSchema#string"]
      }],
      "@type": ["object"]
    }],
    "@id": [{
      "@value": ["https://example.net/library"],
      "@type": ["http://www.w3.org/2001/XMLSchema#string"]
    }]
  }],
  "https://example.net/bookstore": [{
    "@type": ["object"],
    "https://example.net/name": [{
      "@value": ["Open Bookstore"],
      "@type": ["http://www.w3.org/2001/XMLSchema#string"]
    }],
    "@id": [{
      "@value": ["https://example.net/bookstore"],
      "@type": ["http://www.w3.org/2001/XMLSchema#string"]
    }]
  }]
}
```

Anonymní objekt můžeme namapovat jako Ur entitu s typem `objekt` a dále pro něj platí rekurzivně to co pro zdroje v kořenovém elementu, s tím rozdílem, že nemá řídící značku `@id`.

Další novou řídící značkou je značka `@language`, která uchovává informaci o jazyce v případě, že primitivní hodnota je typu `langString`. Také je uložená ve formě řetězcové primitivní hodnoty, ale s tím rozdílem, že zde je zanořená v již existující reprezentaci primitivní hodnoty. Výstupní mapování z Ur se poté může rozhodnout jestli informaci o jazyku využít, nebo ignorovat.

Přestože v RDF Turtle pole (neboli více hodnot u jednoho predikátu) nemají žádné pevné pořadí, v Ur jsou reprezentované stejnou entitou jako pole v Unifikované reprezentaci pro formát JSON. Tedy pomocí entity s typem `array` obsahující

dvojice indexů a hodnot. Cílem tohoto rozhodnutí bylo co nejvíce sjednotit reprezentaci s ostatními formáty. Uživatel případně může toto pořadí ignorovat.

Na následujícím zkráceném příkladu z sekce 1 můžeme vidět mapování pro výše zmíněné jevy.

```
{
  "@type": ["object"],
  "https://example.net/library": [{
    "@type": ["object"],
    "https://example.net/name": [{
      "@value": ["Open Library"],
      "@type": ["http://www.w3.org/2001/XMLSchema#string"]
    }],
    "@id": [{
      "@value": ["https://example.net/library"],
      "@type": ["http://www.w3.org/2001/XMLSchema#string"]
    }],
    "https://example.net/books": [{
      "@type": ["array"],
      "0": [{
        "https://example.net/author": [{
          "@value": ["Milan Kundera"],
          "@type": [http://www.w3.org/2001/XMLSchema#string"]
        }],
        "https://example.net/attributes": [{
          "https://example.net/condition": [{
            "@value": ["torn"],
            "@type": ["http://www.w3.org/2001/XMLSchema#string"]
          }],
          "@type": ["object"]
        }],
        "@type": ["object"],
        "https://example.net/page-count": [{
          "@value": ["352"],
          "@type": ["http://www.w3.org/2001/XMLSchema#integer"]
        }],
        "https://example.net/book-title": [{
          "@value": ["Nesmrtelnost"],
          "@type": [
            "http://www.w3.org/1999/02/22-rdf-syntax-ns#langString"
          ],
          "@language": [{
            "@value": ["cs"],
            "@type: ["string"]
          }]
        }]
      }],
      "1": [{
        ...
      }]
    }]
```

```
  }]
}
```

Tímto jsme ukončili sekci o mapování jednotlivých formátů do Ur a následuje sekce o ukazovacím jazyku pro Unifikovanou reprezentaci UrPath.

4.2 Ukazovací jazyk UrPath

Operace našeho transformačního jazyka potřebují často ukazovat na entity Unifikované reprezentace nebo nové entity v Ur tvořit. K tomuto účelu slouží UrPath, kterou představíme v této části. Píšeme sice o ukazovacím jazyku, ale ve skutečnosti potřebujeme ukazovací jazyk rozšířit, aby mohl "ukazovat"i do neexistujících (právě tvořených) entit. Začněme jednoduchou ukázkou.

```
["library", "books", "[0]"]
```

Cesta ukazuje na nultý prvek entity typu pole `books` v entitě typu objektu `library`. Vidíme, že přímo využíváme pole hostitelského formátu naší speficikace (tedy formátu JSON) pro seřazení jednotlivých tokenů cesty. Zleva do prava postupujeme od kořene hlouběji do větví stromu hierarchické struktury. Prázdné pole označuje kořen Unifikované reprezentace.

Jednotlivé řetězcové tokeny poté můžou obsahovat klíč objektu nebo otevírající a zavírající hranatou závorku, potenciálně s číslem mezi nimi. Závorky ukazují, že se jedná o Ur entitu typu pole a číslo mezi nimi je index do tohoto pole. Prázdné závorky ukazují na prvek těsně po posledním prvku pole a používají se jen ve výstupních cestách při tvoření nových prvků pole. (TODO možná zmínit inspiraci jazykem PHP) V případě použití prázdných závorek ve vstupní cestě se jedná vždy o chybu, neboť prvek po posledním prvku pole neexistuje. V případě, že součástí řetězcového tokenu je otevírající nebo zavírající hranatá závorka musí se nahradit dvojicí symbolů `~1` resp. `~2`. S tímto druhem escapování musíme ve vstupech nahrazovat i znak `~`, a to dvojicí `~0`. Pro správné dekódování tokenů se nejprve nahradí `~1` a `~2` a až poté `~0`. Vyvarujeme se tak chybného dekódování. Jako inspirace pro způsob excapování řídících značek nám sloužil JSONPointer [6].

V následující sekci se již budeme věnovat návrhu samotného transformačního jazyka.

4.3 Transformační jazyk

Jak už bylo řečeno výše, náš navrhovaný transformační jazyk je těsně provázaný s Unifikovanou reprezentací. Pracuje totiž s Ur na svém vstupu i výstupu, což nám umožní určitou míru sjednocení transformačních operací. Píšeme pouze o určité míře sjednocení, neboť z podkapitoly 4.1 víme, že Unifikovaná reprezentace je specifická pro konkrétní formát. Z toho vyplývá, že i zápis transformací

bude muset být závislý na vstupních a výstupních formátech. Přesněji pouze na jejich Unifikovaných reprezentacích.

V návrhu našeho jazyka jsme se inspirovali jazykem používaným již existujícím transformačním nástrojem Jolt, který jsme podrobně rozebrali v sekci 3.3. Zejména jeho přístupem k transformaci tzv. od vstupu k výstupu, oddělenými operacemi s vlastními doménově specifickými jazyky a proměnnými ve specifikacích transformace. Při vysvětlování hlavních principů fungování našeho transformačního jazyka se budeme tedy často vztahovat právě k němu.

Nejprve obecně popíšeme definici transformačního jazyka a jeho struktury bez konkrétní serializace, poté představíme význam a principy základních operací a nakonec ukážeme konkrétní serializaci jazyka do formátu JSON.

4.3.1 Definice transformačního jazyka

Podobně jako u nástroje Jolt je náš jazyk **deklarativní**. Celá specifikace transformace lze zapsat pomocí jakéhokoliv hierarchického formátu, který podporuje zápis objektů s klíči a uspořádaná pole. I když si v této sekci budeme vypomáhat příklady v JSONu, jazyk jako takový lze popsat obecně bez konkrétní serializace.

Vzhledem k více podporovaným formátům a potenciálně více mapováním z formátu na jeho Ur, specifikace transformace musí obsahovat výběr vstupního a výstupního mapování (konvertoru) mezi Unifikovanou reprezentací a daným formátem.

V základu podporuje jazyk vstupní a výstupní převody pro formáty JSON, XML, CSV a RDF Turtle popsané v sekci 4.1.3. Základní konvertory se označují pomocí přípony `default` např. `rdf-ttl-default`, `csv-default` atd. Jazyk a jeho implementace by také měly podporovat vypsání výstupu v samotné Unifikované reprezentaci pomocí výstupního konvertoru `ur-inner`.

S Joltem jsme dále konzistentní v tom, že se celá transformace skládá z několika menších navazujících transformací, kterým říkáme **operace**. Počítáme, že nástroj, který náš jazyk využívá, vykonává operace sekvenčně jednu po druhé, s tím, že výstup předcházející operace je vstupem té následující. Výstup poslední operace je potom výstupem celé transformace.

Každá operace má také svůj doménově specifický jazyk, kterým uživatelstvo zaznamenává svůj záměr.

Celá specifikace transformace se tedy musí v kořenové entitě typu objekt ukládat klíče pro vstupní a výstupní konvertery s jejich řetězcovými hodnotami a pole operací pod klíčem `operations`. Pole s operacemi se poté musí skládat z objektů jednotlivých specifikací transformací každé operace.

Nyní ukažme jednoduchý příklad kostry takové specifikace v serializaci JSON. Znovu ale připoměňme, že by to mohl být i jiný hierarchický formát, např. yaml.

```
{
  "input-converter": "json-default",
  "output-converter": "xml-default",
  "operations": [
    {
      "operation": "remove",
```

```
        "comment": "Komentář k operaci remove",
        "specs": [
          ...
        ]
      },
      {
        "operation": "shift",
        "comment": "Komentář k operaci shift",
        "specs": [
          ...
        ]
      }
    ]
  }
```

Všimněme si možnosti přidávat komentáře k jednotlivým operacím, které jinak ve formátu JSON chybí.

Operace jazyka musí obsahovat svůj jedinečný název, specifikaci a případně komentář. Pozornější čtenářstvo si již také mohlo všimnout toho, že specifikace jednotlivých operací jsou pole a ne objekty.

Rozhodli jsme se totiž, že nejenom transformace budou složeny z množství menších operací, ale i operace samotné budou specifikovány **množstvím transformací**, které se budou týkat vždy jedné entity ve vstupu. Toto rozhodutí pramení z pozorování, že všechny operace jsou ve své podstatě **cesty do vstupní entity zkombinované s nějakou akcí či hodnotou** (např. přesun podle výstupní cesty, výchozí hodnota k nastavení nebo predikát ve filtru). Tento přístup také řeší problém s opisováním celého vstupu v transformaci, který jsme identifikovali v sekci 3.3 u nástroje Jolt.

Jak bylo už řečeno, specifikace operací se skládá z množství objektů sestávající minimálně z cesty do vstupní Ur. Ukazovací jazyk UrPath, který využívají operace k odkazům na entity v Ur, už jsme viděli v sekci 4.2. Taková cesta může být buďto celá z konkrétních hodnot, a nebo může obsahovat **pojmenované proměnné**. Ty jsme zavedli jako alternativu k divokým kartám nástroje Jolt popsaném v sekci 3.3. Pojmenované proměnné fungují prakticky stejně, tj. může se za ně doplnit jakýkoli klíč ve vstupu, který je validní se zbytkem cesty (části klíčů pro tuto chvíli nepodporujeme). Pojmenované proměnné jsou ale, dle našeho názoru, přehlednější a jejich používaní by mělo být více intuitivní, než pouhé pracování se symboly. Jsou to pojmenované proměnné, neboť si uživatel může zvolit jméno libovolné proměnné a poté se na ně v rámci jedné transformace v operaci může odkazovat, jak uvidíme později.

Operacím se podrobně budeme věnovat v následující sekci. Uveďme zde rovnou poznámku, že následující operace jsou jen základ, který by měla každá implementace jazyka poskytovat, ale stejně jako mapování formátů do Ur je lze rozšířit.

4.3.2 Transformační operace

Jak už bylo zmíněno, celá transformace v našem jazyce se skládá z více menších a jednodušších transformací jdoucích po sobě. Takovým jednoduchým transformacím říkáme **operace**. Výstup jedné operace je vstupem té následující a výstup poslední je výstup celé transformace. Tento přístup nám pomáhá zpřehlednit složitější transformace, a tím je potencálně zjednodušit.

Každá operace má **vlastní doménově specifický jazyk**, který je ale záměrně držen co nejvíce jednotný pro všechny navrhované operace, aby byl jazyk jako celek co nejvíce intuitivní. Doménově specifický jazyk operace zajišťuje také výraznou flexibilitu celého nástroje, neboť je poměrně jednoduché přidat implementaci nějaké operace, kterou bychom v budoucnu potřebovali viz Programátorská dokumentace v sekci 5.3.

Doménově specifické jazyky operací definovaných autory této práce navazují na princip transformace "**od vstupu k výstupu**"popsaném v sekci 3.3. Mimo to byly ale zásadně změněny a spolu s nimi byla upravena i sémantika operací. Jazyk podporuje kromě operací `shift`, `remove` a `default` i nové operace `filter`, `array-map` a `replace`.

Návrh operací byl iterativní, neboť jsme ze začátku nevěděli jaké všechny operace budeme potřebovat. Začali jsme se základními operacemi z Joltu s přidáním operace `filter` a zkoušeli jsme tím transformovat data. Na základě toho jsme pak přidávali další operace podle potřeby.

Představme nyní jednotlivé operace a jejich sémantiku obecně. V následující sekci poté představíme konkrétní zápis s příklady použití formou jednoduchého tutoriálu.

- Operace `shift` přesouvá zmíněné vstupní entity na výstup. V transformacích typicky zodpovídá za většinu změny struktury, a jde tedy o stěžejní operaci. Výstup operace jsou všechny přesunuté enitity.
- Operace `remove` jednoduše odstraní hodnotu, kterou na výstupu nechceme.
- Operace `default` inicializuje entity, které nebyly na vstupu zvolenou hodnotou. Na výstupu jsou tedy všechny entity vstupu rozšířené o hodnoty zmíněné v operaci. Využíváme ji například v případě, že chceme vyfiltrovat nějaké entity podle daného klíče, ale klíč mají jenom některé z nich. V takovém případě můžeme nastavit hodnotu klíče na nějakou nevalidní hodnotu a až následně vyfiltrovat. Základní použití je ale samozřejmě nastavení chtěné hodnoty.
- Operace `filter` přidává způsob jak pracovat se strukturou dokumentu i na základě samotných hodnot. Vybere si entitu k testování a entitu kterou v případě nesplnění testu vyfiltruje. Výstup operace jsou vstupní entity bez těch, které nesplnili podmínku.
- Operace `array-map` dokáže převést entitu typu objekt a entitu typu pole, tím že zahodí klíče entity a nahradí je za indexy. Pomáhá nám při transformaci sdružovat klíče do společné enitity typu objekt, který potom v případě potřeby lze převést na pole. Jinak bychom při operaci nevěděli, na jaký index pole máme data ukládat. Také dokáže mapovat pole na pole a smazat tak potenciální díry v indexech, které mohly vzniknout při transformaci.

- Operace `replace` dokáže na základě identifikátoru uloženého v primitivní hodnotě na svoje místo zkopírovat entitu z jiné části dokumentu. Byla vytvořena pro kopírování entit známých zdrojů v RDF na místo, kde je na ně v dokumentu odkazováno. Identifikátor ale ve skutečnosti může být jakýkoliv validní klíč.

4.4 Tutoriál k transformačním operacím

V této sekci předvedeme všechny schopnosti operací formou tutoriálu. Připoměňme ještě, že operace se dají řetězit a výstup předešlé operace je vstup té následující. V rámci jedné operace, ale všechny transformace ve specifikaci pracují s jedním vstupem a upravují jeden výstup. To znamená, že se navzájem neovlivňují. Pokud tedy chceme např. posunout entitu na výstup a následně s ní hned pracovat, musíme operaci `shift` rozdělit na dvě.

Jako vstup pro většinu příkladů budeme využívat ukázkový JSON dokument z první kapitoly, který zde pro pohodlí čtenáře znovu uvádíme.

```
{
  "library": {
    "name": "Open Library",
    "books": [{
      "attributes": {
        "condition": "good"
      }
      "book-title": "Příliš hlučná samota",
      "author": "Bohumil Hrabal",
      "page-count": 98
    }, {
     "attributes": {
        "condition": "torn"
      }
      "book-title": "Nesmrtelnost",
      "author": "Milan Kundera",
      "page-count": 352
    }]
  }
}
```

4.4.1 shift

Tato operace se využívá pro **posun** konkrétních **hodnot** nebo celých **podstromů na výstup**. Skládá se z pole objektů, které mají klíče `input-path` a klíč `output-path`. Pod klíčem `input-path` je uložená UrPath k hodnotě na vstupu a `output-path` obsahuje buďto cestu nebo pole cest ve výstupním objektu, do kterých se posune entita ze vstupu. Už zde vidíme výhodu oproti Joltu, který by musel opsat celý vstup, i kdyby chtěl přesunou pouze jednu entitu. Nezmíněné entity se na výstup nekopírují.

Základní shift operace může vypadat následovně.

```
{
  "operation": "shift",
  "comment": "Posuň hodnotu name z objektu 'library'
              do objektu 'org/lib'",
  "specs": [
    {
      "input-path": ["library", "name"],
      "output-path": ["org", "lib", "name"]
    }
  ]
}
```

Výstup operace bude potom následující.

```
{
  "org": {
    "lib": {
      "name": "Open Library"
    }
  }
}
```

Jak už bylo zmíněno výše, pro vložení entit na konec pole se v cestě využívá symbol `[]`. Skutečná síla operace se projeví až po využití pojmenovaných proměnných. Ty se automaticky nahradí jakýmkoliv klíčem, pro který bude po nahrazení cesta do vstupní entity platná. Proměnné platí jen v rámci jednoho objektu v poli specifikací. To znamená, že nelze využívat hodnotu proměnné v jiném objektu, i když má stejné jméno. Z pohledu jazyka se jedná o dvě různé proměnné.

```
{
  "operation": "shift",
  "comment": "Posuň hodnotu 'name' z jakéhokoli objektu
              do pole 'names' v objektu 'entity'.
              Dále posuň informaci o autorovi
              do jednoho společného pole 'authors'",
  "specs": [
    {
      "input-path": ["@var:named-entity:", "name"],
      "output-path": ["entity", "names", "[]"]
    },
    {
      "input-path": [
```

```
          "@var:named-entity:",
          "books",
          "[@var:index:]",
          "author"
        ],
        "output-path": ["entity", "authors", "[]"]
      }
    ]
  }
```

Na výstupu bude poté následující objekt.

```
  {
    "entity": {
      "names": ["Open Library"]
      "authors": ["Bohumil Hrabal", "Milan Kundera"]
    }
  }
```

Následně lze také hodnota nahrazené proměnné využít i v cestách ve výstupní entitě.

```
  {
    "operation": "shift",
    "comment": "Posuň hodnotu name z jakéhokoli objektu
                do objektu stejného jména a klíče called",
    "specs": [
      {
        "input-path": ["@var:named-entity:", "name"],
        "output-path": ["@var:named-entity:", "called"]
      }
    ]
  }
```

Výstup bude vypadat takto.

```
  {
    "library": {
      "called": "Open Library"
    }
  }
```

V případě, že se pojmenovaná proměnná zmíněná ve výstupní cestě zároveň nenachází v cestě vstupní, jedná o chybu, neboť není žádná validní hodnota, kterou bychom mohli proměnnou nahradit.

4.4.2 filter

Jak název napovídá, tato operace slouží k **filtrování hodnot**. Operace se zapisuje jako pole objektů s klíči `tested-path`, `filtered-path` a `predicate`. Klíč `tested-path` uchovává cestu ve vstupu k entitě (většinou primitivní hodnotě), která se bude testovat pomocí predikátu uloženého pod klíčem `predicate`. Ten uchovává podmínku (predikát), kterou musí testovaná entita splňovat, aby se filtrované entita (typicky rozdílná, než ta testovací) ocitla na výstupu. Klíč `filtered-path` potom uchovává cestu k filtorvané entitě. Výstup operace je tedy entita s všemi filtrovanými klíči a hodnotami, které splňují podmínky predikátů a nebo nebyly v operaci nijak zmíněny.

Predikáty jsou dvojího typu. Buďto se vztahují k samotné hodnotě a nebo k jejímu typu. Značí se stejně jako v Unifikované reprezentaci `@value` respektive `@type`. V zápisu predikátu po nich vždy následuje znaménko `==` nebo `!=`, které porovnává hodnoty na bázi řetězcového porovnání. Na pravé straně znaménka je potom samotná hodnota, vůči které porovnáváme. Zmíněné tokeny musí být od sebe v predikátu odděleny mezerou.

Základní operace `filter` může vypadat například takto.

```
{
  "operation": "filter",
  "comment": "Filtrování literálu podle hodnoty",
  "specs": [
    {
      "tested-path": ["library", "name"],
      "filtered-path": ["library"],
      "predicate": "@value == Open Library"
    }
  ]
}
```

Predikát se nemusí vyskytovat jen u literálu. Porovnávat hodnotu u jiných než literálních hodnot ale povoleno není.

```
{
  "operation": "filter",
  "comment": "Filtrování s porovnáním typu entity u neliterálu",
  "specs": [
    {
      "tested-path": ["library"],
      "filtered-path": ["library"],
      "predicate": "@type == object"
    }
  ]
}
```

Výstup prvních dvou operací bude stejný jako vstup, neboť predikáty byly splněny. Operace `filter` také podporuje pojmenované proměnné.

```
{
  "operation": "filter",
  "comment": "Filtrování s porovnáním typu entity u literálu
              a využitím pojmenované proměnné",
  "specs": [
    {
      "tested-path": ["library", "books", "[@var:i:]", "author"],
      "filtered-path": ["library", "books", "[@var:i:]"],
      "predicate": "@value == Piotr Kropotkin"
    }
  ]
}
```

Výstup poslední operace bude vstup s prázdným polem `books`, neboť žádná entita nesplnila predikát.

```
{
  "library": {
    "name": "Open Library",
    "books": []
  }
}
```

Operace `filter` se často používá v kombinaci s operacemi `shift` a `array-map`. Třeba, když potřebujeme rozdělit entity z jednoho pole do dvou, ve kterých jsou už jen entity stejného typu, např. se stejnou hodnotou klíče `@language`. Přesuneme pomocí operace `shift` všechny prvky do obou polí a následně je vyfiltrujeme. Nakonec už jen použijeme operaci `array-map` na přemapování indexů.

4.4.3 remove

Tato operace slouží k **odstranění** klíčů a jejich hodnot, ať už literálů, či celých objektů. Zapisuje se jako pole objektů s klíčem `path` držící cestu ke klíči k odstranění ve vstupní entitě. Výstupem je tedy vstupní entita bez všech klíčů (a jejich hodnot) zmíněných v operaci. Když se odstraní poslední klíč z objektu, tak na výstupu objekt zůstává (jen je prázdný).

Základní remove operace může vypadat třeba takto.

```
{
  "operation": "remove",
  "comment": "Odstraň klíč 'name' v objektu 'library'",
  "specs": [
```

```
    {
      "path": ["library", "name"]
    }
  ]
}
```

Výstupem by byl původní objekt `library` bez klíče `name`.

Operace remove jako ostatní základní operace podporuje pojmenované proměnné.

```
{
  "operation": "remove",
  "comment": "Odstraň 'name' v každém objektu splňujícím cestu",
  "specs": [
    {
      "path": ["@var:named-entity:", "name"]
    }
  ]
}
```

Operace `remove` je velice jednoduchá, a tak rovnou přejděme na následující.

4.4.4 default

Tato operace slouží k **nastavení výchozích hodnot** pro nové klíče. Pokud ve vstupní entitě už hodnota pro tento klíč existuje, nic se nenastavuje. Operace je pole objektů s klíči `path` a `value`. Klíč `path` obsahuje cestu ke klíči, kterému chce uživatel nastavit hodnotu `value`. Výstupem operace je tedy vstupní entita rozšířená o výchozí hodnoty nastavené při operaci.

Základní default operace může vypadat třeba takto.

```
{
  "operation": "default",
  "comment": "Nastav výchozí hodnotu klíče 'nick' v 'library'",
  "specs": [
    {
      "path": ["library", "nick"],
      "value": "Libertad"
    }
  ]
}
```

Jako ostatní operace i tato podporuje pojmenované proměnné, a je tedy schopna na základě jedné cesty ve specifikaci nastavit defaultní hodnotu více

cestám, které splňují zápis. Hodnotu proměnných nelze využít v defaultní hodnotě. Následující příklad nastaví hodnotu `base-name` klíči `name` všem objektům kořenové entity, které neobsahují daný klíč.

```
{
  "operation": "default",
  "comment": "Nastav výchozí hodnotu pro klíč 'name'
              v objektu splňujícím cestu",
  "specs": [
    {
      "path": ["@var:named-entity:", "name"],
      "value": "base-name"
    }
  ]
}
```

4.4.5 array-map

Tato operace se používá pro **namapování objektu** s klíči **na pole** s indexy. Prakticky se klíče v objektu zahodí a jejich obsah se přidá pod nové indexy pole (s negarantovaným pořadím) a typ entity se změní na pole. Tato operace je potřebná pro případy, kdy chceme do polí na výstupu dávat složitější objekty, které teprve postupně tvoříme. Nejprve je tedy vytvoříme v objektu s klíči jako identifikátory a poté objekt namapujeme na pole operací array-map. Pro pole s primitivními hodnotami toto nepotřebujeme a stačilo by nám pouze přidat primitivní hodnotu jako nový prvek pole pomocí prázdných `[]` v UrPath.

Druhé využití této operace je i pro samotná pole. I když se může mapování pole na pole zdát zbytečné, má svoje opodstatnění. Například, když nám po profiltrování prvků pole, mezi indexy zbydou mezery. Tyto mezery odstraní právě operace array-map, neboť ta zahodí původní indexy a (v potenciálně různém pořadí!) je namapuje na nové hned za sebe.

Specifikace je opět pole objektů s klíčem `path` obsahujícím UrPath k objektu, který chceme namapovat. Základní array-map operace vypadá následovně.

```
{
  "operation": "array-map",
  "comment": "Namapuj první objekt v poli books na pole",
  "specs": [
    {
      "path": ["library", "books", "[0]"]
    }
  ]
}
```

S výsledkem.

```
{
  "library": {
    "name": "Open Library",
    "books": [[
      {"condition": "good"},
      "Příliš hlučná samota",
      "Bohumil Hrabal",
      98
    ], {
     "attributes": {
        "condition": "torn"
      }
      "book-title": "Nesmrtelnost",
      "author": "Milan Kundera",
      "page-count": 352
    }]
  }
}
```

Operace také podporuje pojmenované proměnné. Jejich používání si určitě snadno dokážete představit.

4.4.6 replace

Poslední operací, kterou představíme je operace `replace`. Tato operace byla vytvořena pro **kopírování** entit známých **zdrojů** v RDF na místo kde je na ně **v dokumentu** odkazováno. Identifikátor ale ve skutečnosti může být jakýkoliv validní klíč. Prakticky vezme operace cestu k řetězcové hodnotě s identifikátorem a cestu k entitě, kde má element ke kopii s daným id hledat. V případě, že je element nalezen, nahradí jeho kopií operace entitu, ve které byl původně identifikátor. Specifikace je pole objektů a cesty k identifikátoru a k potenciálnímu rodiči elementu ke kopii jsou v objektech uloženy pod klíčem `id-path` respektive `entity-parent-path`.

Ukažme příklad operace na následujícím vstupu.

```
{
  "libraries": {
    "open-library" {
      "name": "Open Library",
      "friend-library": {
        "id": "radical-library"
      }
    },
    "radical-library" {
      "name": "Radical Library",
      "cool-literature": true
```

```
      }
    }
  }
```

Specifikace, která chce přesunout obsah objektu `radical-library` pod klíč `friend-library` vypadá následovně.

```
{
  "operation": "replace",
  "specs": [
    {
      "id-path": [
        "libraries",
        "open-library",
        "friend-library",
        "id"
      ],
      "entity-parent-path": ["libraries"]
    }
  ]
}
```

Výsledek operace poté vypadá takto. Zkopírovaná entita na výstupu samozřejmě zůstává.

```
{
  "libraries": {
    "open-library" {
      "name": "Open Library",
      "friend-library": {
        "name": "Radical Library",
        "cool-literature": true
      }
    },
    "radical-library" {
      "name": "Radical Library",
      "cool-literature": true
    }
  }
}
```

Pojmenované proměnné v této operaci fungují stejně jako ve všech ostatních.

4.5 Návrh transformačního nástroje

Na konci této kapitoly se ještě vyjádřeme k návrhu samotného nástroje, který bude transformační jazyk využívat.

Jako hostující formát pro deklarativní zápis specifikace transformace jsme zvolili **JSON**, ale jak bylo řečeno výše, jakýkoli hierarchický formát poskytující podobné druhy entit by fungoval stejně dobře.

Jako jazyk implementace nástroje jsme si vybrali **Javu**, primárně pro její velký a fungující ekosystém knihoven, velký počet uživatelů, kteří by potenciálně náš nástroj chtěli používat a snadnou distribuci výsledných programů.

Také jsme se rozhodovali, jak bude vypadat rozhraní mezi uživatelem a programem. Myslíme si, že pro transformaci dat jsou uživatelé zvyklí na práci s příkazovou řádkou a její výhody. Například možnost výstup programu řetězit s jinými programy, což je určitě velké zjednodušení při nutnosti kombinovat transformace různého druhu. Nakonec jsme se tedy rozhodli pro program s jednoduchým rozhraním **na příkazové řádce**.

Konkrétně program bude brát jako svůj vstup dva argumenty. První argument bude cesta ke vstupnímu souboru s daty. Druhý argument bude obsahovat cestu ke specifikaci transformace v jazyce popsaném výše.

Při návrhu transformančího jazyka jsme nepočítali se streamovanou exekucí, a tudíž náš nástroj ani tuto vlastnost splňovat nemusí.

Nástroj by měl také umožnovat, v rámci možností snadné, přidávání nových operací a mapování formátů do Unifikované reprezentace, neboť i v tom částečně spočívá síla našeho návrhu. Jakým způsobem tento požadavek bude uspokojen už necháváme na konkrétní implementaci.

5. Implementace

V této kapitole se budeme věnovat implementaci nástroje a rozhodnutím, která jsme museli při implementaci učinit. Naší implementaci navrhovaného transformančího jazyka bereme jako proof-of-concept. To znamená, že má sloužit hlavně jako ukázka toho, že takový jazyk lze implementovat i používat. Důraz tedy nebyl kladen na časovou či prostorovou optimalizaci. Nástroj jsme navrhovali s myšlenkou extenzibility a logického oddělení komponent a to jsme se snažili zachovat i v implementaci.

Pro načítání dat konkrétních formátů z jejich textové podoby a jejich následné ukládání jsme v implementaci používali knihovny třetích stran. Konkrétně pro formát CSV knihovnu OpenCSV [1], pro formát JSON knihovnu JSON.org [2] a pro formát RDF Turtle knihovnu Apache Jena [3]. Pro formát XML má Java nativní podporu. Toto není ideální, neboť nástroje často nenačítají hodnoty ze vstupu jako řetězec, a tak může dojít k např. zaokrouhlení, kterému jsme se snažili vyhnout. Pro proof-of-concept implementaci nám to ale příjde jako kompromis, který jsme ochotni udělat.

Všechny ostatní části nástroje už jsme implementovali sami.

Návrh jazyka ve své podstatě počítá s tím, že se transformace bude vykonávat s celými načtenými daty v paměti. Nástroj tedy nepodporuje vykonávání transformací streamovaně a pokud uživatelstvo chce transformovat velká data, musí na to mít i patřičně velkou operační pamět. V naší implementaci se může stát, že jsou vstupní data v paměti načtená i dvakrát vedle sebe. Z toho vyplývá i poměrně velký tlak na garbage collector při vykonávání transformací.

V administrátorské dokumentaci popíšeme jak nástroj zprovoznit. V uživatelská dokumentace se bude zabývat používáním nástroje a stručně znovu vysvětlí používané koncepty jako pro úplné nováčky. V dokumentaci programátorské popíšeme návrh našeho nástroje a způsoby rozšíření nástroje o operace nebo mapování na další formáty.

5.1 Administrátorská dokumentace

Vzhledem k tomu, že je nástroj myšlený v tuto chvíli jako proof-of-concept aplikace, vybrali jsme pro distribuci ten nejjednodušší způsob, který funguje. Program je psaný v Javě, a tak poskytujeme `.jar` archiv, který potřebuje pouze JVM ke spuštění. Program je pro verzi Javy 17. Spuštění programu je poté pouze následujícím příkazem.

```
java -jar jmeno_archivu.jar vstupni_data.in specifikace.json
```

První argument je cesta k souboru se vstupními daty a druhý cesta k specifikaci transformace. Výstup transformace bude zapsán na standardní výstup, aby si ho uživatelstvo mohlo přesměrovat, kam bude potřebovat - např. do souboru nebo na vstup dalších programů.

[1] `https://opencsv.sourceforge.net`

[2] `https://javadoc.io/doc/org.json/json/latest/org/json/package-summary.html`

[3] `https://jena.apache.org/index.html`

5.2 Uživatelská dokumentace

Nástroj slouží k transformacím hierarchických dat a převodům mezi datovými formáty, na které lze nahlížet jako hierarchické. Konkrétně jsou podporované formáty JSON, RDF (serializace Turtle), XML a CSV. Abychom mohli formáty transformovat společným jazykem, musíme je umět jednotně uchopit. Hlavní roli v sjednocování datových modelů jednotlivých formátů hraje koncept **Unifikované reprezentace**, se kterou přicházíme a které je potřeba pro používání nástroje porozumět. Princip celé transformace je totiž převod vstupních dat do Unifikované reprezentace, transformace dat v Unifikované reprezentaci a poté převod zpátky do nějakého z podporovaných formátů (potenciálně jiného, než jaký byl na vstupu).

Nástroj pracuje se třemi koncepty, kterými jsou Unifikovaná reprezentace, ukazovací jazyk UrPath a poté samotný transformační jazyk s operacemi. Nyní stručně představíme všechny tři zmíněné koncepty.

Unifikovaná reprezentace

Datový model Unifikované reprezentace (zkráceně Ur) implementujeme pomocí formátu JSON. Z uživatelského hlediska si lze Ur představit jako JSON s výhradně řetězcovými hodnotami, který je mapován na zmíněné hierarchické formáty. Primitivní hodnoty si ukládáme v JSON objektech rozdělené na typ a hodnotu uložené jako řetězce. Složitější entity jako jsou objekty a pole si ukládáme v JSON objektu s klíči resp. indexy a informací o typu - `objekt` nebo `array`. Hodnoty všech klíčů jsou obalené do JSON pole pro jednotný přístup k všem druhům entit a snazší implementaci nástroje. Pro uživatelstvo to nemá prakticky žádný vliv.

Příklady mapování Ur na konkrétní formáty můžeme vidět v sekci 4.1.2.

Ukazovací jazyk UrPath

Transformační nástroj potřebuje často ukazovat na entity Unifikované reprezentace nebo nové entity v Ur tvořit. K tomuto účelu slouží jazyk UrPath, který zde představíme. Začněme jednoduchou ukázkou.

```
["library", "books", "[0]"]
```

Cesta ukazuje na nultý prvek pole `books` v objektu `library`. Pro seřazení jednotlivých tokenů cesty využíváme pole hostitelského formátu speficikace (tedy formátu JSON). Zleva do prava postupujeme od kořene hlouběji do větví stromu hierarchické struktury. Prázdné pole označuje kořen Unifikované reprezentace.

Jednotlivé řetězcové tokeny poté můžou obsahovat klíč objektu nebo otevírající a zavírající hranatou závorku, potenciálně s číslem mezi nimi. Závorky ukazují, že se jedná o pole a číslo mezi nimi je index do tohoto pole. Prázdné závorky ukazují na prvek těsně po posledním prvku pole a používají se jen ve výstupních cestách při tvoření nových prvků pole. V případě použití prázdných

závorek ve vstupní cestě se jedná vždy o chybu, neboť prvek po posledním prvku pole neexistuje.

Transformační jazyk a operace

Transformační jazyk je deklarativní a náš nástroj implementuje serializaci specifikace transformace do formátu JSON. Je také těsně provázaný s Unifikovanou reprezentací. Pracuje totiž s Ur na svém vstupu i výstupu, což nám umožní určitou míru sjednocení transformačních operací. Píšeme pouze o určité míře sjednocení, neboť Unifikovaná reprezentace je sice jednotnější, ale stále specifická pro konkrétní formát. Z toho vyplývá, že i zápis transformací bude muset být závislý na vstupních a výstupních formátech. Přesněji pouze na jejich Unifikovaných reprezentacích.

Ve specifikaci transformace tedy vždy musíme uvést vstupní a výstupní mapování (neboli konverzi). Vstupní konverze se zapisuje řetězcovou hodnotou pod klíč `input-converter` a podobně výstupní konverze pod klíč `output-converter` do kořene JSON dokumentu.

Základní podporovaná mapování se označují pomocí přípony `default`, tedy `rdf-ttl-default`, `csv-default`, `xml-default`, `json-default`.

Nástroj také podporuje vypsání výstupu v samotné Unifikované reprezentaci pomocí výstupního konvertoru `ur-inner`. Ta vypíše Ur v JSON serializaci. Je rozumné, zvláště při transformaci mezi různými formáty, si nejdříve nechat vypsat Unifikovaou reprezentaci vstupu a pak už jen přemýšlet nad transformací samotné Ur.

V kořenovém elementu specifikace transformace musí dále už být jen klíč `operations`, který obsahuje pole operací, které provádí samotnou transformaci. Všechny ostatní klíče jsou nástrojem ignorovány a uživatelstvo je může využít k čemukoliv chce - např. komentářům, které v JSONu chybí.

Uvěďme jednoduchou ukázku kostry specifikace transformace.

```
{
  "input-converter": "json-default",
  "output-converter": "xml-default",
  "operations": [
    {
      "operation": "remove",
      "comment": "Komentář k operaci remove",
      "specs": [
        ...
      ]
    },
    {
      "operation": "shift",
      "comment": "Komentář k operaci shift",
      "specs": [
        ...
      ]
    }
  ]
```

```
}
```

Celá transformace se skládá z více menších a jednodušších transformacích jdoucích po sobě. Takovým jednoduchým transformacím říkáme **operace**. Výstup jedné operace je vstupem té následující a výstup poslední je výstup celé transformace. Operace na sobě tedy závisí a vykonávají se jedna po druhé. Tento přístup nám pomáhá zpřehlednit složitější transformace, a tím je potencálně zjednodušit.

Každá operace má **vlastní doménově specifický jazyk**, který je ale záměrně držen co nejvíce jednotný pro všechny navrhované operace, aby byl jazyk co nejvíce intuitivní. Doménově specifický jazyk operace zajišťuje také výraznou flexibilitu celého nástroje, neboť je poměrně jednoduché přidat implementaci nějaké operace, kterou bychom v budoucnu potřebovali. Zde odkážeme na sekci 5.3 obsahující programátorskou dokumentaci s potřebnými informacemi k rozšíření nástroje.

Stejně jako transformace jsou složeny z množství menších operací, i operace samotné jsou specifikovány množstvím transformací, které se budou týkat vždy jedné entity ve vstupu. Toto rozhodutí pramení z pozorování, že všechny operace jsou ve své podstatě cesty do vstupní entity zkombinované s nějakou akcí či hodnotou (např. přesun podle výstupní cesty, výchozí hodnota k nastavení nebo predikát ve filtru).

Ukazovací jazyk UrPath, který využívají operace k odkazům na entity v Ur, už jsme viděli výše. Nezmínili jsme ale, že taková cesta může být buďto celá z konkrétních hodnot a nebo může obsahovat **pojmenované proměnné**. Ty fungují podobně jako divoké karty na příkazové řádce linuxu. Za pojmenované proměnné se doplní jakýkoliv klíč ze vstupu, který je validní se zbytkem cesty (části klíčů pro tuto chvíli nepodporujeme). Pojmenovanné proměnné slouží jednak k obecnějším transformacím a jednak se s hodnotami doplněných proměnných poté může pracovat i dále v operaci např. na výstupní cestě a podobně.

Na závěr této sekce ještě zmiňme, že všechny operace, a tím i celý nástroj funguje na principu "**od vstupu k výstupu**". To znamená, že na rozdíl od jiných transformačních nástrojů, které popíšou strukturu výstupu a do něho se snaží z vstupu doplnit ty správné hodnoty, funguje náš nástroj opačně. Vychází ze struktury vstupu a hodnoty transformuje na správné místo na výstup. Tento přístup není mezi transformačními nástroji tolik zastoupený, ale v principu by měl mít, při správném návrhu, stejnou vyjadřovací schopnost. Dle autorů tohoto nástroje umožnuje uživatelstvu přistupovat k transformaci intuitivněji, tak jak se skutečně děje, a tím vytvářet jednodušší a přehlednější transformace.

Předvedení konkrétních operací formou jednoduchého tutoriálu můžete vidět v sekci 4.4.

5.3 Programátorská dokumentace

V této sekci stručně popíšeme obecný návrh naší implementace a zmíníme, jak by se případně dala rozšířit o nové operace či mapování do Unifikované reprezentace. Nebudeme zde ale rozebírat konkrétní třídy a rozhraní, neboť to není

předmětem této práce. I proto, že třídy jsou patřičně okomentované dokumentačními komentáři přímo ve zdrojovém kódu, který je k nalezení v příloze této práce. Zmínění některých tříd či rozhraní se ale nevyhneme.

Návrh je oddělený na konverzi vstupního souboru do Unifikované reprezentace, samotnou transformaci, která pracuje s Ur na svém vstupu i výstupu a postupně aplikuje na vstup operace a nakonec konverzi do výstupního formátu. Velmi blízko tedy kopíruje návrh transformačního jazyka a jeho částí. Zmiňme ještě, že třída implementující Unifikovanou reprezentaci poskytuje veřejně CRUD rozhraní (tedy operace `vytvoř`, `čti`, `aktualizuj` a `odstraň`), se kterým poté můžou jednotlivé operace pracovat.

Jak operace, tak vstupní a výstupní konvertory jsou při načítaní specifikace tvořeny tzv. factory třídami, tedy třídami, které na základě jména operace vytvoří jeho inicializovanou instanci. Pro přídání podpory do nástroje pro nová mapování nebo operace, je nutné tyto factory třídy nahradit vlastními. Pokud by někdo pouze chtěl operaci nebo mapování přidat, může podědit od základních implementací těchto tříd, dekorovat hlavní metodu svým rozšířením a následně zavolat původní implementaci. Pro větší změny je potřeba factory rozhraní implementovat úplně od začátku. Případné nové mapování nebo operace samozřejmě také musí implementovat patřičná rozhraní. Pro oba dva případy jsou připraveny v nástroji třídy, které sdružují společné rysy tříd daného typu, od kterých jde snadno dědit. S přidáváním mapování i operací se při vývoji počítalo a nemělo by být složité.

Upozorněme ještě čtenářstvo, že přestože lze na úrovni operací implementovat libovolný doménově specifický jazyk, který lze zapsat do hierarchického formátu typu JSON, měly by se rozšíření držet i hlavní myšlenky nástroje tzv. přístupu "od vstupu k výstupu". Toto ale nijak přímo vynucováno implementací není a jedná se spíše o silné doporučení.

6. Zhodnocení

V této kapitole představíme dvě studie, které nám pomůžou zhodnotit návrh našeho nástroje a jeho implementaci. První, uživatelská, studie nám pomůže zhodnotit uživatelskou přívětivost, přístupnost konceptů a intuitivnost používání našeho nástroje. Využijeme totiž zpětné vazby respondentstva. Druhá, případová, studie nám pomůže zhodnotit vybavenost nástroje při složitější transformaci z praxe. V předposlední sekci vyvedeme z obou studií závěry. V poslední sekci poté nastíníme možné upravy a rozšíření návrhu jazyka a nástroje do budoucna.

6.1 Uživatelská studie

Abychom mohli zhodnotit náš návrh i z pohledu nového uživatelstva, oslovili jsme studentstvo z prostředí okolo Matematicko-Fyzikální fakulty Univerzity Karlovy, s prosbou, aby náš nástroj vyzkoušeli na pár jednoduchých transformacích a poskytli nám zpětnou vazbu. V této sekci se primárně snažíme vyhodnotit jednoduchost konceptů na pochopení a intuitivnost používání nástroje při jednoduchých až středně težkých transformacích. Nejprve rámcově představíme úkony a otázky, které měly za úkol participanti vyřešit respektive zodpovědět. Následně shrneme jejich dojmy a hodnocení.

6.1.1 Zadání

Zadání uživatelské studie jsme rozdělili na dvě části. První část měla za úkol co nejjednodušším způsobem předat informace o stěžejních konceptech transformačního nástroje. Prakticky jsme poskytli respondentům lehce upravenou uživatelskou dokumentaci ze sekce 5.2. Spolu s ní jsme předali i administrátorskou dokumentaci ze sekce 5.1, neboť druhá část měla praktický charakter a bylo potřeba nástroj zprovoznit.

Ve druhé části se po respondentstvu chtělo transformovat pět jednoduchých až středně těžkých příkladů. Poskytli jsme vstupy a požadované výstupy a dali časový limit sedm minut na příklad. I pokud by někdo transformaci v požadovaném čase nestihl, je to pro nás cenná zpětná vazba. Také jsme nechtěli ještě více prodlužovat nutný čas pro vyplnění dotazníku. Příklady testovaly jak transformace v rámci jednoho formátu, tak i mezi nimi.

Uveďme příklad jedné testované transformace z formátu JSON do formátu XML. Zbytek testovaných příkladů i s jejich vzorovým řešením je přiložen v hlavní příloze.

Vstupní soubor vypadal následovně.

```
{
  "people": [
    { "name": "Piotr" },
    { "name": "Biotr" }
  ]
}
```

Požadovaný výstupní soubor poté vypadal takto.

```
<?xml version="1.0" encoding="UTF-8"?>
<people>
  <human>
    <called>Piotr</called>
  </human>
  <human>
    <called>Biotr</called>
  </human>
</people>
```

Tento příklad primárně testoval převod mezi různými Unifikovanými reprezentacemi. Také použití operací `default` pro řídící značky XML a `shift` pro změnu celé struktury. Žádali jsme také, aby tvořené transformace byly co nejobecnější, aby si lidé zkusili i práci s pojmenovanými proměnnými.

Nakonec bylo respondentstvo požádáno o odpovědi na následující otázky.

> Jak byste zhodnotili vaši znalost/zkušenost s formáty CSV, XML, JSON, RDF? (stačí "JSON - dobrá, CSV - skoro žádná, ...")
>
> Máte zkušenost s nějakými transformačními nebo dotazovacími jazyky pro tyto formáty? (např. XSLT, jq, SQL, ...) Pokud ano, tak jakou?
>
> Jak hodnotíte provedené transformace? Šlo vše udělat jednoduše a intuitivně, nebo jste narazili na nějaké problémy? Pokud se vyskytly problémy, prosím uveďte jaké. (Ideálně se prosím vyjádřete ke všem pěti transformacím)
>
> Jak hodnotíte obecně návrh jazyka? Nedělá vám problém vidět reprezentaci konkrétního formátu, ale muset přemýšlet v Unifikované reprezentaci? Změnili byste něco? Vyhovuje vám přístup k transformaci "od vstupu k výstupu"?

Odpovědi na tyto otázky vyhodnotíme v následující podkapitole.

6.1.2 Shrnutí odpovědí

Nakonec se nám podařilo získat zpětnou vazbu od tří respondentů. Jejich odpovědi shrneme v této sekci. Kompletní odpovědi na otázky i s řešením jednotlivých úkolů jsou k nalezení v hlavní příloze práce.

Na otázku ohledně zkušeností s formáty prakticky všichni respondenti uváděli, že zkušenost s formáty JSON, XML a CSV mají dobrou nebo velmi dobrou. S formátem RDF naopak skoro všichni neměli zkušenost žádnou a nebo jen lehkou.

Ohledně zkušeností s transformačními a dotazovacími jazyky zmiňovali dotázaní dobrou zkušenost s jazyky SQL a XPath. V některých odpovědích také zazněly odpovědi jako JSONPath, SPARQL nebo CSS Selector.

Odpovědi na otázku ohledně úkolů na transforamce byly odpovědi různé. Respondenti se shodovali, že první dva příklady, které byly v rámci formátu JSON, byly jednoduše vymyslitelné s poskytnutou dokumentací. Dva ze tří respondentů, ale byli překvapeni, že po využití operace `shift` nemohli dále transformovat nezmíněná data. Jinými slovy nebylo jim zřejmé, že se oprace řetězí a výstup předchozí je vstup té následující. Pro většinu bylo náročnější převádět z formátu na formát u úkolů tři a čtyři, ve kterých se převádělo z JSONu do XML a z CSV do JSONu. Všem se ale podařilo dostat se velmi blízko k řešení. U posledního příkladu směřovaného na filtraci dat v rámci formátu JSON respondenti převážně uváděli, že se jim ho podařilo vyřešit.

Na poslední otázku ohledně návrhu celého nástroje a Unifikované reprezentace se respondenti shodovali, že jim nástroj jako takový přijde intuitivní a funkční. Zmiňují, že přístup "od vstupu k výstupu"v zásadě kopíruje jejich myšlenkový proces nad transformací. Také se jim líbil zápis pomocí formátu JSON, pro který už existují nástroje. Někteří ale zmiňovali problémy konkrétních operací. Třeba na chování při mazání prvků v poli, kde po prvcích zůstávájí mezery nebo na malou kontrolu nad pořadím prvků v poli.

V souvislosti s Unifikovanou reprezentací se respondenti vyjadřovali o mírném zmatení a nebo o "vizuálním šumu", který jim ztěžoval transformace mezi různými formáty. Celkově si ale respondenti chválili užitečnost takového nástroje a jeden uváděl, že by si snadno dokázal představit jeho nasazení v praxi.

V následující sekci popíšeme případovou studii a poté společně vyvodíme závěry a nástroj vyhodnotíme.

6.2 Případová studie

Nástroj jsme chtěli otestovat i na větších datech, se kterými bychom se mohli setkat ve skutečném světě. Konkrétně jsme pro tuto případovou studii zvolili veřejně dostupná data z Národního Katalogu Otevřených Dat České republiky (dále jen NKOD), která si lze stáhnout na odkazu https://data.gov.cz/soubor/nkod.trig. Požadovaná transformace, která se veducímu této práce skutečně vyskytla v praxi a byla nemalým popudem pro vznik této práce, má převést data z původního formátu RDF TriG do specifické podoby formátu JSON, odpovídající schématu GraphQL endpointu na https://data.gov.cz/graphql.

V této sekci nejprve blíže představíme strukturu dat NKODu a schéma požadovaného výstupu. Dále potom popíšeme naší zkušenost s nástrojem při větší transformaci. V následující sekci se poté ve světle této zkušenosti, i zkušeností responentů z uživatelské studie, pokusíme zhodnotit náš návrh a implementaci nástroje.

6.2.1 NKOD

Data NKOD, se kterými jsme pracovali slouží k popisu datových sad zařazených do katalogu otevřených dat. V katalogu se tedy vyskytují informace o jejich názvu, slovním popisu, klíčových slovech, autorech, podmínkách užití, kontaktním bodu atd. Některé informace jsou jednoduché primitivní hodnoty. Jiné jsou ale, jak to v RDF bývá, samostatné zdroje se složitější strukturou a vlastními jednoznačnými identifikátory.

Uvěďme krátký úryvek z dat pro představu.

```
<https://data.gov.cz/zdroj/datové-sady/48135097/3c9...e5> {
<https://data.gov.cz/zdroj/datové-sady/48135097/3c9...e5>
  a dcat:Dataset, typdz:DokumentyDCAT, typdz:LKOD;
  dct:title
    "Ochranné známky - přírůstek 2022-12-18"@cs,
    "Trademarks - increment 2022-12-18"@en;
  dct:description "Datová sada obsahuje data
      o národních ochranných známkách.
      Data jsou publikovaná ve formě ročního
      plného exportu a denních přírůstkových aktualizací."@cs,
      "This dataset contains national trademark data.
      Data is published in form of full export
      and subsequent incremental data."@en;
  dcat:distribution
    <https://data.gov.cz/../distribuce/a6...53b>;
  dct:accrualPeriodicity
    <http://publications.europa.eu/../frequency/DAILY>;
  dcat:keyword
    "ÚPV"@cs,
    "ochranné známky"@cs,
    "známky"@cs,
    "IPO CZ"@en,
    "trade marks"@en,
    "trademarks"@en;
  ...
  dcat:theme theme:GOVE, eurovoc:6416;
  dcat:contactPoint
      <https://data.gov.cz/zdroj/datové-sady/4../../kontaktní-bod>;
  dct:publisher ovm:48135097 .
```

Celý soubor se zkrácenými klíči pomocí slovníků má 82MB a obsahuje bezmála milion sto tisíc řádků.

Formát výstupu je potom JSON s polem objektů, které popisují datové sady, a které mají, jak je u JSONu zvykem, obsah složitějších odkazovaných zdrojů vnořeny přímo ve svém objektu. Také se u některých vnořených entit zbavujeme jednoznačné identifikace.

Konkrétní schéma můžeme vidět v příloze práce.

6.2.2 Průběh transformace

Na začátek se zmiňme o rychlosti běhu transformace a spotřebě paměti. Celá transformace souboru o velikosti 82MB seběhla na notebooku autorů se šestijádrovým procesorem AMD Ryzen 5 4500U a 8GB operační paměti.

Nástroj spotřebovává na transformace velké množství paměti, neboť, jak bylo zmíněno v úvodu kapitoly 5, v jeden čas může být celý strom načtený v paměti hned několikrát. Aby takto velká transformace vůbec seběhla, museli jsme navýšit maximální velikost haldy JVM na 3,5GB.

Časově se transformace pohybuje na našem stroji okolo 32 vteřin. To nepovažujeme za příliš dobré, ale vyhodnocení by vyžadovalo důkladnější srovnání i s jinými nástroji.

Nyní už k samotné transformaci. Hned ze začátku jsme museli vyřešit problém ohledně, do té doby neexistujícího, mapování pro formát RDF TriG do Unifikované reprezentace. S existencí mapování (i konvertoru) RDF Turtle to ale nebylo složité vyřešit. Neboť jak autorstvo RDF Trig píše, formát je prakticky RDF Turtle jen s možností uložit více pojmenovaných grafů [23]. Pro jednoduchost jsme zdroje všech grafů sloučili do jednoho kořenového elementu, neboť to byl jednoduchý způsob, který pro náš problém fungoval.

S tímto mapováním už jsme byli schopni automaticky převést data z RDF TriG do formátu JSON. Následně už zbývalo "jen"transformovat data do správného tvaru. V celé transformaci jsme použili deset operací, z čehož byla jen jedna operace `shift`, tři operace `filter`, dvě operace `replace`, dvě operace `array-map` a jedna operace `default`.

Pracovali jsme se všemi zdroji RDF v kořenovém elementu. Nejprve jsme poměrně dlouhou specifikací operace `shift` přesunuli všechny chtěné hodnoty do správných míst na výstupu. Velmi nám v tom pomáhaly pojmenovanné proměnné. Zachovali jsme si také některé klíče, které využijeme dále v transformaci na filtrování (např. typ).

Za zmínku v této části transformace stojí zejména způsob, jakým jsme transformovali řetězcové hodnoty s konkrétním jazykem (tzv. language string). Na výstupu jsme chtěli hodnoty pro český a anglický jazyk oddělené v polích pod klíči `cs` reps. `en`. Na vstupu jsme je ale měli smíchané v jednom poli. Postupovali jsme následovně. V operaci `shift` jsme si celé pole přesunuli jak pod klíč `cs`, tak pod klíč `en`. To můžeme, neboť nám operace nabízí přesunout jednu entitu na více míst zároveň. Poté jsme v následujích operacích profiltrovali jednotlivá pole podle zadání a následně použili operaci `array-map` pro správné přemapování indexů.

V transformaci jsme pokračovali oprací `replace`, pomocí které jsme nahradili identifikátory zdrojů jejich skutečnými obsahy. Museli jsme tak činit na dvakrát, tedy dvěma operacemi `replace` za sebou, neboť jeden zdroj nejprve sám potřeboval nahradit jeden odkaz a poté až mohl být sám zkopírován na patřičné místo.

V tuto chvíli už jsme pomocí operací `default` a `filter` mohli vyfiltrovat všechny zdroje v kořenové entitě mimo datových sad, které na výstupu chceme. A následně odstranit všechny nepotřebné pomocné klíče v entitách.

Celou transformaci jsme si drželi RDF zdroje pod klíči s jejich identifikátory, ale jak bylo zmíněno výše, požadovaný výsledek má být pole datových sad. Přesně na to jsme na závěr využili operaci `array-map` a měli jsme hotovo. Celou specifikaci transformace můžeme vidět v příloze práce.

Většiny strukturální transformace tedy bylo dosaženo operacemi `shift` a `replace`. A ostatní operace byly důležité zase při výběru a filtraci správných hodnot. Transformace ve výsledku nebyla ani tak složitá, jak se z počátku zdálo a podařilo se nám jí uspěšně provést. Veškerá data k transformaci jsou dostupná v příloze.

6.3 Vyhodnocení

V této sekci pomocí výsledků uživatelské a případové studie zhodnotíme náš návrh transformačního jazyka a jeho částí.

Nejprve zkusme iterpretovat zkušenosti respondentů v uživatelské studii. Pomineme–li problémy spojené s počátečním neporozuměním vykonávání operací, můžeme prohlásit, že pro jednoduché operace v rámci jednoho formátu je náš návrh pro nováčky intuitivní a snadno stravitelný. Způsob transformace "od vstupu k výstupu"se tedy zdá být v tomto ohledu zvolený správně.

Komentáře ohledně jednotlivých operací bychom ale měli vzít v potaz a potenciálně poskytnout i v základních operacích větší kontrolu nad pořadím prvků, například operací `sort`.

Zmatení z Unifikované reprezentace popř. negativnější názory na její přehlednost, lze částečně připisovat malé zkušenosti s takovýmto přístupem nebo s konkrétním mapováním formátů. Můžeme ale říci, že nutnost přemýšlet o transformaci Ur a ne přímo formátu samotného zřejmě přidává určitou složitost navíc. To nás vede k otázce, jestli by se nedala Unifikovaná reprezentace navrhnout přehldněji, abychom co nejvíce usnadnili její pochopení.

Myslíme si, že velkou roli ve zmatení mohlo hrát jednotné uzavírání všech hodnot do pole. Zpětně jsme si uvědomili, že jsme všechna mapování formátů byli schopni vyřešit i s omezením, že v Ur poli bude vždy jen jeden prvek. Případné řídící hodnoty, které by se potřebovali skládat z více hodnot, bychom mohli vyřešit podobně jako u kompozitních klíčů, tedy uložit do řídící značky složitější strukturu.

Náš návrh Unifikované reprezentace, i přes svoji potenciálně nadbytečnou složitost, nám efektivně umožnuje převádět data z formátu do formátu, a také je transformovat. Přesvědčili jsme se o tom i v úspěšné případové studii v sekci 6.2. Po vyzkoušení nástroje na složitější transformaci jsme si všimli, že sice rozdělení na jednodušší operace částečně zpřehledňuje složitější transformace, ale absence jedné složité a mocné operace nás nutí k "úkrokům"při transformaci. Jde např. vytváření pomocného objektu nebo přidání více entit pod jeden klíč s následným vyfiltrováním, což naopak celou transformaci dělá méně intuitivní a přehlednou. Toto může být způsobeno i naším konkrétním zvolením operací a je dost dobře možné, že by jazyk založený na principu "od vstupu k výstupu"s jinými, lépe zvolenými operacemi, mohl tyto problémy mitigovat. Také by se ale dalo argumentovat, že tyto nutné "úkroky"nejsou nutně na závadu a je to jednoduše styl jakým transformace probíhá, neboť podobné jevy vidíme např. při dotazovaní pomocí jazyka SQL.

Jak bylo zmíněno výše, naše implementace nástroj potřebuje na transformace spoustu operační paměti. S tím ale uživatelstvo musí počítat. Jako autoři nástroje to nevnímáme jako příliš velký problém, neboť jak bylo ukázáno v sekci 6.2, i tato naše neoptimalizovaná proof-of-concept implementace je schopna transformovat na poměrně slabém stroji i poměrně velká data.

6.4 Úpravy a rozšíření do budoucna

Zmiňme na závěr možná rozšíření a úpravy naší práce.

V uživatelské studii v sekci 5.2 jsme narazili na požadavky na větší kontrolu nad pořadím prvků v poli. I nás v průběhu návrhu napadala možná rozšíření k jednotlivým operacím. Například operace filter by mohla podporovat složitější predikáty než jen základní podovnání řetězců na rovnost. Přemýšleli jsme i nad podporou jednoduchých funkcí jako např. dálka řetězce a podobně, nebo dokonce nad podporou pro regulární výrazy. V operaci shift by také mohl přibýt způsob, jak pracovat s klíči vstupní cesty ve výstupní hodnotě.

Velkou změnou transformačního nástroje by mohlo být rozšíření lineárního výpočtu na oddělené paralelní větvě, které by se eventuelně dále ve výpočtu mohly znovu spojit. Potenciálně by to mohlo pomoci se složitějšími operacemi, neboť by šlo část transformace delegovat na oddělené větve. A přidávalo by to místa pro paralelizaci výpočtu, což by mohlo pomoci s výkonem.

Poslední změny, které zmíníme jsou pro navrhovanou Unifikovanou reprezentaci, u které by stálo za to prozkoumat možná řešení bez sjednocování všech prvků v poli, což by mohlo pomoci s přehledností a orientací.

Závěr

Cílem práce bylo prozkoumat stávající nástroje využívané pro transformaci zvolených hierarchických formátů a na základě jejich analýzy navrhnout vlastní řešení. Z tímto účelem jsme navrhli koncept Unifikované reprezentace do které mapujeme konkrétní formáty, vlastní dotazovací jazyk UrPath a transformační jazyk, který s Unifikovanou reprezentací pracuje na svém vstupu a výstupu. Podařilo se nám úspešně vytvořit proof-of-concept aplikaci, která navržený jazyk implementuje a následně ho i otestovat v uživatelské a případové studii v kapitole 6.

Na základě uživatelské studie jsme zjistili, že nástroj je pro nováčky intuitivní a snadno použitelný pro jednoduché až středně těžké transformace. Uživatelé ocenili přístup "od vstupu k výstupu", který jim pomáhal lépe pochopit proces transformace. Nicméně, některé specifické operace způsobovaly problémy a vedly k nejasnostem, což naznačuje, že je potřeba zlepšit dokumentaci a možná i samotnou implementaci některých operací, jako je shift a filter.

Případová studie ukázala, že nástroj je schopný provádět složitější transformace na velkých datech, což dokazuje jeho praktickou využitelnost. Nicméně, vysoké nároky na operační paměť a delší doba zpracování naznačují, že je potřeba optimalizace výkonu a efektivnější správa paměti. Transformace složitějších dat ukázala, že současný návrh operací je dostatečně mocný, ale může být dále vylepšen zavedením složitějších predikátů a dalších funkcí, které by uživatelům poskytly větší flexibilitu a kontrolu.

Z hlediska Unifikované reprezentace bylo zjištěno, že naše zpracování konceptu přidává určitou složitost a vizuální šum navíc, což může vést ke zmatení uživatelů. Proto bychom v další iteraci návrhu doporučovali přehlednost zohlednit ve větší míře, i třeba na úkor obecnosti.

Seznam použité literatury

[1] Sperberg-McQueen Maler Yergeau Bray, Paoli. Extensible markup language (xml) 1.0 (fifth edition), 2008. Accesed: 2024-07-17.

[2] Raimond Schreiber. Rdf 1.1 primer, 2014. Accesed: 2024-07-17.

[3] Shafranovich. Common format and mime type for comma-separated values (csv) files, 2005. Accesed: 2024-07-17.

[4] Spiegel Robie, Dyck. Xml path language (xpath) 3.1, 2017. Accesed: 2024-07-17.

[5] Goessner. Jsonpath - xpath for json, 2007. Accesed: 2024-07-17.

[6] Bryan. Javascript object notation (json) pointer, 2013. Accesed: 2024-07-17.

[7] Brantner-Florescu Westmann Zaharioudakis Robie, Fourny. Jsoniq, 2022. Accesed: 2024-07-17.

[8] Saryerwinnie. Jmespath, 2015. Accesed: 2024-07-17.

[9] Kay. Xsl transformations (xslt) version 3.0, 2017. Accesed: 2024-07-17.

[10] jq 1.7 manual, 2023. Accesed: 2024-07-17.

[11] Baker. Jj json stream editor, 2023. Accesed: 2024-07-17.

[12] Done. jl, 2021. Accesed: 2024-07-17.

[13] Bazaarvoice. Jolt, 2019. Accesed: 2024-07-17.

[14] Schibsted. Jslt, 2023. Accesed: 2024-07-17.

[15] Jsonata documentation, 2021. Accesed: 2024-07-17.

[16] Eberhardt. Json transforms, 2016. Accesed: 2024-07-17.

[17] Žižka. Csv + json cruncher - query and process your csv and json files using sql., 2023. Accesed: 2024-07-17.

[18] Delva Meester, Heyvaert. Rdf mapping language (rml), 2024. Accesed: 2024-07-17.

[19] Lehn. Json-ld primer, 2023. Accesed: 2024-07-17.

[20] Tennison. Csv on the web: A primer, 2016. Accesed: 2024-07-17.

[21] Bazaarvoice. Jolt introduction, 2013. Accesed: 2024-07-17.

[22] Kellog-Lanthaler Lindstrōm Sporny, Longley. Json-ld 1.1, a json-based serialization for linked data, 2017. Accessed: 2024-07-17.

[23] Cyganiak Bizer. The trig syntax, 2007. Accessed: 2024-07-17.

Seznam tabulek