VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

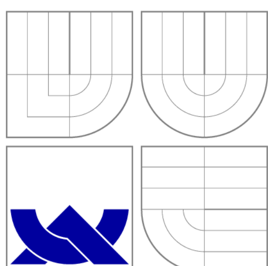
GENEROVÁNÍ VALIDAČNÍCH SCHÉMAT PRO XML

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

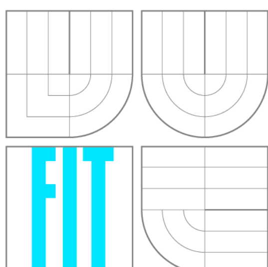
AUTOR PRÁCE
AUTHOR

BOHUSLAV HLOŽEK

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GENEROVÁNÍ VALIDAČNÍCH SCHÉMAT PRO XML

GENERATING VALIDATION SCHEMATA FOR XML

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — BOHUSLAV HLOŽEK
AUTHOR

VEDOUCÍ PRÁCE — PAVEL SMRŽ, doc., RNDr., Ph.D.,
SUPERVISOR

BRNO 2015

Abstrakt
Projekt se zabývá analýzou XML LMF slovníku. Je tvořeno schéma pro schematron a XML s popisem hodnot atributů. Projekt je zpracován v jazyce Python 3.4.

Abstract
The project analyses XML LFM dictionary. Schematron schema is created and XML with description of attribute values is created. The project is coded in Python 3.4.

Klíčová slova
LMF, Lexical Markup Framework, slovník, XML, DTD, XSD, RelaxNG, Schematron, XPATH, XSLT, Python

Keywords
LMF, Lexical Markup Framework, dictionary, XML, DTD, XSD, RelaxNG, Schematron, XPATH, XSLT, Python

Citace
Bohuslav Hložek: Generování validačních schémat pro XML, bakalářská práce, Brno, FIT VUT v Brně, 2015

Generování validačních schémat pro XML

Prohlášení

Prohlašuji, že jsem tento semestrální projekt vypracoval samostatně pod vedením pana Pavla Smrže. Uvedl jsem všechny prameny, ze kterých jsem čerpal.

....
Bohuslav Hložek
10. května 2015

© Bohuslav Hložek, 2015.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 O práci **2**
1.1 Motivace 2

2 Teoretický úvod **3**
2.1 XML 3
2.2 Schéma obecně 5
2.3 DTD 5
2.4 XPATH 7
2.5 XSLT 9
2.6 XSD 12
2.7 RelaxNG 15
2.8 Schematron 18

3 Řešení problému **21**
3.1 LMF 21
3.2 Generování schématu 23
3.3 Analýza elementu feat 25

4 Skript **28**
4.1 Princip činnosti 28
4.2 Užití 30

5 Shrnutí **31**

A Obsah DVD **34**

Kapitola 1

O práci

Tato práce pojednává o generování validačních schémat pro XML ze zadaných dat. V této kapitole je představeno členění práce pro lepší orientaci čtenáře.

V sekci *Motivace* je představen problém, jímž se práce zabývá. Čtenář se dozví, co je potřeba s problémem udělat.

Kapitola *Teoretický úvod* si klade za cíl objasnit generickou problematiku XML a schémat pro ně. Jedná se o teoretickou část, kterou může zkušený čtenář přeskočit, pokud jsou mu názvy všech podkapitol zcela známy. V opačném případě doporučuji kapitolu pročíst jako první, neboť může napomoci se sjednocením mého a čtenářova pohledu na celou problematiku.

Kapitola *Řešení problému* objasňuje, jak problém skutečně řeším (za využití prostředků popsaných v kapitole *Teoretický úvod*). Čtenář se zde dozví, s čím jsem přesně pracoval (zejména bude poučen o formátu `LMF`) a jaká řešení jsem navrhl.

Kapitola *Skript* čtenáři představuji řešení na výsledném skriptu. Kapitola specifikuje požadavky pro jeho spuštění a vysvětluje, jak je dosaženo řešení navrženého v kapitole *Řešení problému*. Rovněž se zde čtenář stručně dozví, jak skript používat.

Kapitola *Shrnutí* stručně shrnuje výsledky práce a mé postoje k dané problematice. Poskytuji zde zamyšlení nad dalším využitím této práce včetně příkladů.

1.1 Motivace

Cílem projektu je analyzovat uživatelem vybraný slovník. Slovník bude ve formátu XML a bude dodržovat standard LMF. Já budu vytvářet schéma, ze kterého uživatel pochopí, jak je slovník strukturovaný. Práce pojednává o výběru potřebných informací, volbě vhodného schématu, způsobu analyzovaní slovníku a ukládání požadovaných informací do schématu.

Kapitola 2

Teoretický úvod

V této kapitole představuji čtenáři problematiku, jež je nutná k pochopení práce a výsledného skriptu.

2.1 XML

Extensible Markup Language (dále jen XML) je zcela zásadním jazykem práce – dostal se i do názvu práce. To naznačuje, že jeho pochopení je zcela zásadní.

Jazyk XML vznikl jako zjednodušení a zpřísnění jazyka SGML. Cílem bylo vytvořit jazyk vhodný pro ukládání a přenos dat, který by byl srozumitelný nejen pro programátory, ale i pro běžné lidi.

XML patří mezi značkovací jazyky (stejně jako SGML). Značkovací jazyk znamená, že implicitně je při psaní v tomto jazyce psán běžný text – programové konstrukce musí být odděleny zvlášť pomocí vyhrazených značek. To je opačný přístup než u běžných programovacích jazyků, kde je implicitně psán kód a běžný text musí být uveden ve speciálních značkách, např. uvozovkách. Značky v XML jsou zásadní zejména dvě. Pro začátek kódu je použit symbol *menší než* < a pro ukončení části s kódem symbol *větší než* >. Celé části mezi *menší než* a *větší než* (včetně zmíněných dvou krajních symbolů) budu dále v práci říkat *tag*.

Názvem tagu bude rozuměno první slovo v něm obsažené. V jiných značkovacích jazycích jsou již pevně dané předpřipravené názvy s předem definovanou sémantikou. V jazyce *html* tag

```
<table>
```

vždy udává, že bude následovat tabulka. XML je v tomto ohledu volnější a poskytuje možnost definovat si vlastní tagy, přičemž je dle standardu [1] povolen znak z ISO/IEC 10646 (s omezením na identifikátory). Lidštěji přeloženo, lze užít platné znaky v Unicode. Vytvořím si příklad, na kterém budu demonstrovat vše, co zde vysvětluji. Nechť je dána situace, ve které si chci pamatovat informace o přidělených úkolech. Vytvořím tag

```
<úkol>
```

Mohu si dovolit napsat *ú*, neboť čeština je obsažena ve zmíněném standardu. V praxi je bezpečnější užívat písmena anglické abecedy, neboť mnozí programátoři z celého světa nemusí mít češtinu na paměti při tvorbě programů pracujících s XML. Dále budu užívat tag

```
<task>
```

Jazyk XML dále vyžaduje, aby byly tagy párové. To znamená, že každý tag musí mít i svůj ukončovací tag, jenž následuje za ním a obsahuje původní název tagu předcházený lomítkem. V našem případě bude existovat

```
<task></task>
```

Přípustná je i zkrácená forma, ve které je koncový tag reprezentován lomítkem na konci tagu počátečního, tj.

```
<task />
```

Části v dokumentu začínající začátečním tagem XY a končící koncovým tagem XY budu říkat *element* XY (XY je název tagu, v našem případě element *task*).

Obsahem elementu budu rozumět vše, co se nachází v textové části mezi tagy daného elementu.

```
<task>Buy a new radio.</task>
```

Obsahem elementu *task* je v mém případě věta "Buy a new radio." Z toho vyplývá, že použití zkrácené verze koncového tagu implikuje prázdný obsah elementu. Obsahem elementu může být i další element, ten nazývám *vnořeným elementem.*

```
<task>
<description>Buy a new radio.</description><reason>Old is broken.</reason>
</task>
```

Na příkladu má element *task* dva vnořené elementy – *description* a *reason.* Zejména je nutné si uvědomit, že element zahrnuje jak počáteční, tak koncový tag. Konec vnořeného elementu musí předcházet konci elementu jemu nadřazenému. Nesmí dojít ke křížení.

```
<task><description>Buy a new radio.</task></description> IS INVALID XML
```

Celý XML soubor musí být zapouzdřen do jednoho elementu. Tomuto elementu říkám *kořenový element.*

```
<task>sleep</task><task>wake up</task> IS INVALID XML
```

```
<root><task>sleep</task><task>wake up</task></root> IS VALID XML
```

Elementy mohou rovněž obsahovat *atributy.* Atributy se nacházejí v počátečním tagu elementu. Každý atribut se zapisuje jako název atributu následovaný znakem *rovná se* =, poté se píše hodnota atributu umístěná do uvozovek (nebo apostrofů). Atributy mohou sloužit k ukládání dat, ale zejména se užívají k vyhledávání dat (např. technologií XPATH). Element může obsahovat i více atributů (ovšem unikátních názvů).

```
<task state="finished" worker="John">Buy a new radio.</task>
```

Na příkladu má element *task* dva atributy – *state* a *worker.*

V praxi se stává, že na XML pracuje více osob. Případně očekáváme, že budou použity cizí xml dokumenty. V tomto případě by mohlo dojít k problematickému jevu, kdy si více autorů pojmenuje element stejně, a přitom se sémantika ve verzi každého autora liší. To je důvodem, proč XML podporuje *jmenné prostory.* Tyto jmenné prostory se píší před název elementu, oddělené dvojtečkou se zbytkem názvu. Na každý jmenný prostor by měl existovat odkaz, který odkazuje užitý standard (aby nedošlo ke kolizi jmenných prostorů a bylo zřejmé, k čemu jmenný prostor slouží). Tento odkaz se píše jako hodnota atributu *xmlns* buď do kořenového elementu, nebo do prvního elementu z daného jmenného prostoru.

```
<a:table xmlns:a="http://www.w3.org/TR/html4/">
<a:tr><a:td><task>Buy a new radio.</task></a:tr></a:td>
</a:table>
```

Na začátek XML souboru je umisťována hlavička, ve které se specifikuje verze XML a kódování zdrojového souboru. Poznámky rovněž v XML existují a využívají obvyklé SGML notace. Pokud dám dohromady vše, co jsem vysvětlil, mohu ukázat komplexní příklad

```
<?xml version="1.0" encoding="utf-8"?>
<root xmlns:a="http://www.w3.org/TR/html4/">
<!-- toto je poznámka -->
<a:table><a:tr>
<a:td><task status="completed">Buy a new radio.</task></a:td>
<a:td><task status="pending">Change bulb.</task></a:td>
</a:tr></a:table>
</root>
```

To je vše, co bude z XML v projektu potřeba. Pro více informací doporučuji pročíst již zmíněný standard. [1]

2.2 Schéma obecně

V předchozí sekci o XLM jsem zmínil, že je možné volně definovat jména elementů a atributů. Záleží jen na autorovi, co chce vyjádřit. Problémem ovšem je, pokud se více autorů snaží vyjádřit podobné jevy a každý z nich to udělá trochu jinak. Z tohoto důvodu se různé obory snaží o sjednocení syntaxe a částečně i sémantiky použité v XML (kupř. standard LMF pro slovníky, se kterými se v tomto projektu pracuje). Vznikají schémata, která slouží pro definici pravidel (omezení na vnořené elementy, jména, hodnoty, apod.) daného XML. Tato schémata používá syntaktický analyzátor (slangově znám jako *parser*) XML i pro jejich validaci. Jedním z nejstarších schémat je DTD (ještě z dob SGML), podle něhož je definovaný formát LMF, kterým se tato práce zabývá. Je tedy nutné tento formát objasnit.

2.3 DTD

Document Type Definition (dále jen DTD) je poměrně jednoduchým jazykem, jenž může sloužit pro popis schématu XML. Jednotlivá pravidla se píší do značek. Úvodní značka je symbol *menší než* a *vykřičník* `<!`, koncovou značkou je *větší než* `>`. Vzniklý tag není párový, neboť veškeré informace se píší do tohoto tagu. Typ pravidla je určen prvním slovem v tagu.

Pravidlo pro vnořené elementy je definováno slovem ELEMENT. Na druhém místě je název elementu, kterého se pravidlo týká. Na třetím místě je umístěn povinný vnořený element. Pravidlo se definuje pro každý platný element.

```
<!ELEMENT root table>
```

Pravidlo říká, že element *root* musí obsahovat právě jeden element *table* (zejména neobsahuje žádný jiný element). Pokud bych chtěl povolit více elementů, umístím jejich názvy do kulatých závorek, oddělených čárkou.

```
<!ELEMENT root (table, chair, bed)>
```

Tím jsem vytvořil pravidlo, kdy element *root* musí obsahovat právě jeden po každém z elementů *table*, *chair* a *bed*. Množství výskytů lze specifikovat pomocí symbolů za názvy elementů. *Otazník* značí nejvýše jeden výskyt. *Hvězdička* značí libovolný počet výskytů a znaménko *plus* značí alespoň jeden výskyt.

```
<!ELEMENT root (table+, chair*, bed?)>
```

Pravidlo říká, že element *root* má 0-1 *elementů bed*, 1-inf elementů *chair* a 0-inf elementů *chair*. Jiná množství DTD specifikovat neumí (např. 3-5 elementů *bed*). Pokud chci na výběr jednu z možností, použiji *svislici*.

```
<!ELEMENT root ((table|chair),bed)>
```

Pravidlo říká, že element root obsahuje právě jeden element *bed* a právě jeden z elementů *table* nebo *chair*. Pár přidaných závorek stojí za povšimnutí, protože bez nich by syntaktický analyzátor nevěděl, jak postupovat. Čárka a svislice nemají v DTD definovanou prioritu operací [6]. Pokud chceme uvést, že element obsahuje textové informace, napíšeme konstantu #PCDATA (znaková data, probíhá syntaktická analýza) nebo #CDATA (neprobíhá syntaktická analýza) [11]. Samotná PCDATA procházejí analyzátorem, a proto se musí užívat entity. Kupříkladu nelze napsat symbol uvozovek a musíme místo něj použít & (zápisu znaku pomocí & říkám *entita*). DTD umí definovat i vlastní entity, ale ty nejsou k pochopení standartu LMF třeba, a proto pro zájemce odkazuji na citované zdroje z této sekce.

```
<!ELEMENT root (#PCDATA|table)*>
```

Pravidlo říká, že element *root* může (ale nemusí) obsahovat textová data a může (ale nemusí) obsahovat elementy *table*. Posledním, co bych chtěl u elementů zmínit, jsou speciální konstanty, jež lze použít jako pravidlo. EMPTY říká, že element musí být prázdný. ANY definuje, že obsahem elementu může být cokoliv – to se moc často nepoužívá, neboť to jde proti původnímu cíli pevně stanovit pravidla.

Druhým pravidlem (a zároveň posledním, které budu potřebovat) je pravidlo pro atributy. To se píše ve tvaru ATTLIST, následuje název elementu, pro které pravidlo tvoříme, a poté následuje dvojice název atributu a přípustná hodnota atributu.

```
<!ATTLIST task status (completed|pending)>
```

Toto pravidlo říká, že element *task* může mít atribut status, jenž musí nabývat hodnoty *completed* nebo *pending*. Pokud bych chtěl specifikovat, že atribut je v elementu vždy povinný, uvedu na konec #REQUIRED. Z příkladu je rovněž patrné, že lze použít obdobné konstrukce jako u pravidel pro elementy. Přípustné konstanty jsou opět CDATA, ale i ID a IDREF(S). CDATA slouží k obecnému znakovému textu, ID značí, že hodnota v daném atributu je jedinečná (v rámci všech stejných atributů daného elementu) a IDREF je hodnota, která jinde v souboru XML existuje jako ID (z hlediska databází by se jednalo o klíč a cizí klíč). IDREFS značí více odkazů na ID.

Pokud chci definovat více pravidel typu atribut pro stejný element, je možné každé psát zvlášť (pro přehlednost), nebo příslušné dvojice až entice umisťovat na konec jednoho pravidla. První možnost ukáži na posledním příkladu

```
<!ATTLIST task status (completed|pending)>
<!ATTLIST task worker CDATA #REQUIRED>
<!ATTLIST task tasknumber ID #REQUIRED>
```

Při shrnutí všeho, co jsem probral, lze spatřit několik zásadních nedostatků DTD. DTD neumí nijak podporovat jmenné prostory – nelze vytvořit pravidlo pro element z daného jmenného prostoru. DTD používá svoji vlastní syntaxi, není ve formátu XML (což by mohlo být pro XML pravidlo vhodné). Také neumožňuje definovat složitější pořadí a zanoření elementů. A hlavně nemám k dispozici pokročilejší datové typy. Mohu sice říci, že se jedná o data PCDATA, ale nemohu specifikovat celé číslo, datum, řetězec apod. Tyto problémy vedly k vytvoření pokročilejších schémat, a i o nich budou následující sekce.

2.4 XPATH

Cílem je tvořit pokročilejší pravidla v XML schématech. Pokročilejší než nabízí DTD. Zároveň by bylo vhodné mít lepší možnost výběru, kterých přesně elementů se bude pravidlo týkat. Je potřeba technologie, která je schopná jednoznačně popsat množinu elementů či atributů. Touto technologií je *XML Path Language*, dále jen XPATH.

XPATH si reprezentuje XML v podobě stromové struktury. Vhodné je přirovnání k adresářovému systému. Disková jednotka se připodobní ke kořenovému elementu. Každý element může mít více dětí (v této práci nazývaných jako vnořených elementů), stejně jako každý adresář může mít více podadresářů. Ke každému elementu existuje jen jeden rodič (element nadřazený). Rodičem kořenového elementu je kořen dokumentu. Textový obsah elementu je nazýván atomem. Každému elementu se rovněž říká uzel.

```
<root>                         <!-- otcem je kořen dokumentu -->
  <prvni_s_r>                  <!-- otcem je root -->
    <prvni_syn_prvniho_s_r />  <!-- otcem je prvni_s_r -->
  </prvni_s_r>
  <druhy_syn_root>             <!-- otcem je root -->
  hello world!                 <!-- textový atom -->
  </druhy_syn_root>
</root>
```

Princip činnosti XPATH je následující. Tazatel vytvoří dotaz a XPATH mu vrátí množinu vyhovujících objektů (elementů, atributů, obsahů). Nejpodstatnější je, že XPATH implicitně vrací všechny objekty vyhovující dotazu. Následně proberu nejdůležitější části syntaxe a sémantiky. Dotaz je tvořen sestavením XPATH výrazu. Ten se skládá z následujících výrazů

```
osa::test_uzlu[predikát]
```

Osa určuje, kterým směrem bude XPATH postupovat (v připodobnění: klikne na podsložku, vyjede úplně nahoru na disk C, apod.) Test uzlu určuje, který uzel musí být po cestě na ose k dispozici, určuje se jménem. Predikát stanovuje další podmínky na uzel (musí být první, musí mít konkrétní atribut aj.) Jednotlivé výrazy se oddělují lomítkem.

XPATH má i implicitní chování. Pokud neuvedu osu, implicitně se použije osa child, která odpovídá zanoření na přímého potomka (připodobnění: vejití do podsložky). Pokud neuvedu predikát, žádné omezující podmínky nejsou. Nejdůležitější věci předvedu na příkladech. Tyto příklady sice předvedou nejdůležitější části XPATH, ale zdaleka nepopisují vše, co XPATH nabízí. Proto doporučuji projít standard [3], ze kterého vycházím (případně alespoň tutoriál [12] od stejné společnosti na w3schools). Pro další výklad budu používat následující XML

```
<r>
  <a id="30">
    <c>1</c>
    <c id="40">2</c>
    <d>
      <c>3</c>
    </d>
  </a>
  <b>
    <c>4</c>
  </b>
  <a id="50">
    <c>5</c>
  </a>
</r>
```

XPATH začíná znakem / sloužícím pro definovaní absolutní cesty v dokumentu (od začátku, od kořene dokumentu dolů). Pokud se tento znak neuvede, jedná se o relativní cestu od současného uzlu, ale o tom budou až následující sekce. XPATH _/r/a/c_ nalezne tři elementy (elementy jsou vráceny včetně obsahu), a to elementy `c(1)`, `c(2)` a `c(5)`, neboť všechny jsou od kořenového elementu zanořeny v elementu `a`.

Pokud chci nalézt element v jakémkoliv zanoření, použiji // místo / (zkrácený zápis pro osu _descendant_). Výraz _/r/a//c_ nalezne čtyři elementy `c(1,2,3,5)`, neboť od kořene musí být zanořeny v elementu `a` a poté mohou být v libovolném dalším zanoření (i žádném).

V XPATH existují i zástupné znaky. _Tečka_ je odkaz na aktuální uzel a _dvě tečky_ na rodičovský uzel. _Hvězdička_ odkazuje na libovolný název. Výraz _/r/*/*/c/.._ vrátí element `d`, neboť je hledán rodič elementu `c`, jenž je třikrát zanořen, a to splňuje právě jen `c(3)`.

Atributy se hledají pomocí symbolu _at_. Výraz _//@id_ vrátí seznam tří vyhovujících atributů `id(30,40,50)`. Pokud potřebuji hledat element mající konkrétní atribut, dám atribut do predikátu. Výraz _//c[@id]_ nalezne `c(2)`, neboť ten má jako jediný atribut _id_.

XPATH podporuje i běžné operátory, lze vytvořit výraz _//*[@id>35]_ . Výsledkem je element `c(2)` a element `a` mající atribut _id_ padesát. Číslo v predikátu implicitně určuje pořadí. Mírně obměněný první příklad _/r/a[1]/c_ vrátí pouze `c(1,2)`, protože `c(5)` se nenachází v prvním nalezeném elementu `a`, jenž je zanořený v `r`.

Do predikátu lze psát i funkce, kterých jsou desítky. Jednou z nich je i _last()_, která vrací poslední uzel ze současně vybraných. Výraz _/r/a[last()]/c_ vrátí `c(5)`, neboť se hledá v posledním `a` zanořeném v `r`. Kdyby tento element neobsahoval žádné `c`, nic by nebylo vráceno. Další užitečnou funkcí je například _position()_, kterou lze s výhodou využít spolu se znaménky pro porovnání. Funkce _text()_ vrací atomický obsah elementu současného uzlu. Výraz _//c[position()>1]_ vrací `c(2)`.

Více XPATH výrazů lze sloučit pomocí svislice. Pak budou nalezeny všechny výskyty ze všech výrazů. Z toho plyne, že _/r/b/c | /r/a[1]/c[1] | /r/*/*/c_ nalezne `c(1,3,4)`. V predikátech lze užit i logické podmínky. Výraz _//*[@id<50 and not(@id=30)]_ vrací `c(2)`.

Posledním, co mi zbývá vysvětlit, jsou osy [8]. Implicitní osa je _child_. Ta jde o jednoho potomka níže. Osa _self_ ukazuje sama na sebe (ekvivalentem je symbol _tečka_). Osa _parent_ ukazuje na rodiče (ekvivalentem jsou _dvě tečky_). _Descendant_ je potomek po přímé ose (v připodobnění se do složky dostanu, aniž bych se někdy vracel do nadřazené složky). Již jsem zmínil, že toto lze značit jako //. Osa _ancestor_ je předek (pouze ty složky, do kterých

se vracíme). *Descendant-or-self* a *ancestor-or-self* vrací totéž co *descendant* a *ancestor* plus současný uzel. *Following-sibling* a *preceding-sibling* vrací uzly sousedící se současným na stejné úrovni. Sekci završím komplexnějším příkladem, který mi umožní předvést několik os a jejich možné využití pro vyhledávání.

```
<r>
  <a>
    <b>THIS</b>
    <c>10</c>
    <d></d>
  </a>
  <a>
    <c>20</c>
  </a>
  <a>
    <extra><b>THIS</b></extra>
    <c>30</c>
    <d>FIRST</d>
    <d>SECOND<d>THIRD</d></d>
  </a>
</r>
```

V tomto případě mám XML databázi (která může být mnohem delší než na ukázce). Databáze je tvořena položkami `a`. Data uvnitř položek jsou rozmístěna náhodně. Úkolem je vybrat všechna užitečná data elementu `d` z označených záznamů (značkou THIS v elementu `b`), které mají hodnotu elementu `c` větší než patnáct. Výsledek může vypadat například následovně.

```
//descendant-or-self::b[.="THIS"]/ancestor::a//descendant::c[text()>15]
+//ancestor::a//d/text()
```

Naleznu všechny položky `b` v souboru, jejichž obsah je THIS (zde ukazuji, že lze použít zástupný znak *tečka*). Vrátím se k jejich záznamům. Tím ukazuji na první a třetí element `a`. Naleznu v nich element `c` a zkontroluji podmínku. Té vyhoví pouze element `c` u třetího záznamu. Vrátím se k celému třetímu záznamu. Vyberu z nich veškeré elementy `d`. Výsledkem bude FIRST SECOND THIRD.

2.5 XSLT

Další technologií, kterou zde budu dále rozebírat, je `eXtensible Stylesheet Language Transformations`, dále jen XSLT. XSLT se používá k transformaci XML dokumentů na jiné. Transformace probíhá díky XSLT procesoru, který vezme vstupní XML a XSLT šablonu, ze kterých vytvoří výsledný dokument. Šablona je rovněž ve formátu XML a udává, jak má výsledek vypadat, přičemž může používat libovolná data vstupního XML. Pro přístup k těmto datům se používá nám již známý XPATH.

V následující části popíši základy práce se šablonovým dokumentem. Šablona vždy začíná řádkem

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

Ze sekce o XML lze vyvodit, že příkazy XSLT užívané v souboru budou ze jmenného prostoru *xsl*. XSLT zde funguje tak, že volá šablony pro zpracování uzlů ve vstupním XML (implicitně postupuje systematicky po XML souboru, nezáleží na pořadí definice šablon). Pokud šablona pro daný uzel neexistuje, zavolají se implicitní šablony [7]. Jedna implicitní šablona postupně prochází XML (prochází od kořene souboru a volá potomky) a snaží se na každém uzlu předat řízení šabloně, kterou definoval uživatel. Druhá implicitní šablona dává na výstup textový atomický obsah právě zpracovaného uzlu. Jen připomínám, že na právě zpracovávaný uzel se lze s výhodou odvolat v XPATH znakem *tečka* – lze tvořit relativní XPATH výrazy. Právě zpracovávaný uzel je obvykle (výjimka je cyklus iterace) ten, jehož šablona je právě aktivní.

Typickým postupem je vytvoření šablony pro kořen dokumentu, tj. /. Tím mi implicitní šablona při první příležitosti předá řízení a já si mohu veškeré zpracování nastavit sám. Pokud budu chtít zpracovat jiný uzel, tak pro něj vytvořím šablonu a v současné šabloně ji zavolám pomocí funkce *xsl:apply-templates*. Toto volání se implicitně provádí pro každého potomka současného uzlu nebo pro všechny elementy, jež vyhovují volitelnému atributu *select*, jehož hodnotou je XPATH výraz (pokud není zadán absolutně, vztahuje se k relativní pozici současného uzlu). Tato volání se provádějí rekurzivně (dochází k návratu). Mějme XML

```
<root>
<a><b>PRVNI</b></a>
<a><b>DRUHY</b></a>
<c><b>TRETI</b></c>
</root>
```

a mějme XSLT transformační soubor

```
 1.<xsl:stylesheet version="1.0" xmlns:xsl="www.w3.org/1999/XSL/Transform">
 2.  <xsl:template match="/">        <!-- šablona pro kořen -->
 3.   <xsl:apply-templates/>
 4. </xsl:template>
 5. <xsl:template match="a">         <!-- šablona pro element a -->
 6.   <xsl:copy-of select="b" />
 7. </xsl:template>
 8.</xsl:stylesheet>
```

Výsledkem bude

```
<b>PRVNI</b>
<b>DRUHY</b>
TRETI
```

Funguje to tak, že implicitní šablona začne postupovat od kořene a snaží se najít vyhovující šablonu, jež jsem definoval. Tou je hned šablona pro kořen. V ní se zavolá šablona pro každého potomka. Jediným potomkem kořene dokumentu je `root` (neplést kořen dokumentu s kořenovým elementem), pro kterého šablona neexistuje, a proto se zavolá implicitní šablona, která postupně zavolá šablony všech děti. Pro uzly `a` existuje šablona, která zkopíruje element `b` přímo v něm obsažený. Za povšimnutí stojí název funkce pro kopii *copy-of* a použití relativního XPATH výrazu v atributu *select* (relativně vztaženo k příslušnému elementu

a). Pro uzel `c` šablona neexistuje, a proto se implicitně zavolají šablony pro potomky. Jediným potomkem je uzel `b`. Ten rovněž šablonu nemá, a proto se aplikuje implicitní šablona, která vypíše atomický obsah TRETI. Na tomto příkladu demonstruji, že ačkoliv se výsledek zdá téměř správný (pokud bych chtěl vypsat elementy `b` v elementech `a`), tak ve skutečnosti měli velkou nadvládu nad průchodem implicitní šablony, a proto je třeba hlídat možná volání šablon nedefinovaných uzlů. Opravím třetí řádek tím, že přesně specifikuji, aby se volaly pouze šablony pro uzel `a`

```
3. <xsl:apply-templates select="root/a"/>
```

Druhým přístupem je iterování v šabloně a přístup k datům. Pří iteracích se mění aktuální uzel. XML i výstup se zachovávají z předchozího příkladu, změna bude u XSLT souboru.

```
1.<xsl:stylesheet version="1.0" xmlns:xsl="www.w3.org/1999/XSL/Transform">
2.  <xsl:template match="/">
3.    <xsl:for-each select="root/a">
4.      <xsl:copy-of select="./b"/>
5.    </xsl:for-each>
6.    <xsl:value-of select="/root/c/b"/>
7.  </xsl:template>
8.</xsl:stylesheet>
```

Na příkladu lze zpozorovat, že iterační cyklus se provádí funkcí *xsl:for-each*. Atributem je *select*, jenž muže obsahovat relativní nebo absolutní XPATH cestu. Tečkou na čtvrtém řádku zdůrazňuji, že se mění aktuální uzel (šlo by jednoduše napsat `b`). Na šestém řádku představuji funkci pro vypsání hodnoty *value-of*.

XPATH podporuje i podmíněné zpracování. Funkce *xsl:if* ovšem nemá větev else, a proto se používá i *xsl:choose*. Ta má jednu a více části *xsl:when* (nahrazení elsif) a část *xsl:otherwise* (náhrada else). Demonstruji na příkladu. Mějme XML

```
<shop>
<item price="25">Blue item</item>
<item price="80">Green item</item>
<item price="75">Yellow item</item>
</shop>
```

a XSLT transformaci

```
<xsl:stylesheet version="1.0" xmlns:xsl="www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:apply-templates select="shop/item"/>
  </xsl:template>
  <xsl:template match="item">
    <xsl:choose>
      <xsl:when test="@price>50">
        We recommend <xsl:value-of select="."/>,
        only for <xsl:value-of select="@price"/> Kč,-
      </xsl:when>
```

```
      <xsl:otherwise>
        We do not recommend <xsl:value-of select="."/>
      </xsl:otherwise>
</xsl:choose></xsl:template></xsl:stylesheet>
```

s výsledkem

```
We do not recommend Blue item
We recommend Green item, only for 80 Kč,-
We recommend Yellow item, only for 75 Kč,-
```

Šablona kořene volá šablony pro každý uzel `item`. Tato šablona je definovaná a na základě podmínky *test* v části *xsl:when* se provede příslušná část (pokud není žádná podmínka when splněna, provede se část *xsl:otherwise*). Příklad zdůrazňuje, že prostý text v šabloně, jenž není ze jmenného prostoru XSL, se kopíruje na výstup při každém jeho projití. Příklad rovněž připomíná, že XPATH výrazy umí pracovat i s atributy.

To, co jsem zde řekl o XSLT je pouze špičkou ledovce. XSLT nabízí daleko víc – zejména generovaní elementů/atributů předem neznámých jmen, stanovování priorit a módů pro vícenásobné zpracování uzlů – pro zájemce doporučuji standard [2].

2.6 XSD

XML schema, také známé jako *XML Schema Definition* (dále jen XSD) je další z technologií pro tvorbu XML schémat. Cílem je vyřešit nedostatky z DTD. Pro XSD existuje i doporučení [5], ze kterého budu v této sekci vycházet.

Jedním z problémů DTD byl fakt, že samotné DTD schéma nebylo validním XML, a bylo nutné učit se novou syntaxi. Naproti tomu je XSD validním XML ze jmenného prostoru *xs*. Zapouzdřovací formát XSD schématu je

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.w3.org/1999/xhtml">
<!-- pravidla zde -->
</xs:schema>
```

Kořenovým elementem je element `schema` z již zmíněného jmenného prostoru `xs`. Jmenný prostor je rovněž v kořenovém elementu standardně definován. Užitý volitelný atribut `targetNamespace` říká, ze kterého jmenného prostoru jsou elementy, ke kterým jsou tvořena pravidla (v mém příkladu se XML schéma bude zřejmě zabývat zpřísněním pravidel pro *xhtlm*). Z toho vyplývá, že XSD je (na rozdíl od DTD) schopno pracovat se jmennými prostory.

Základními pojmy v XSD jsou pravidla pro jednoduché a komplexní typy. Jednoduchých typem se rozumí element, atribut a omezení (jak přesně je to myšleno bude vysvětleno dále). Komplexním typem jsou prázdné elementy, elementy s vnořenými elementy, elementy s textem a elementy s textem i vnořenými elementy.

Jednoduché typy slouží zejména k použití v komplexních typech pro dále již nerozložitelné části.

Jednoduchý typ `element` slouží pro element, který nemá žádné atributy (pouze elementy komplexního typu mohou mít atributy) a nemá žádné vnořené elementy. Základní syntaxe je

```
<xs:element name="jmenoelementu" type="xs:datovytyp" />
```

Z výše uvedené syntaxe lze zpozorovat, že XSD podporuje datové typy (celé číslo, datum, řetězec, uri aj.) Dalším jednoduchým typem je element `attribute` reprezentující atribut. Jeho základní syntaxe je

```
<xs:attribute name="jmenoatributu" type="xs:datovytyp" use="required" />
```

Velmi podobné jako u elementu. Možnosti XSD jsou rozsáhlejší, než je ukázáno v této práci. To znázorňuji na jednom z možných volitelných atributů `use`, který udává, že atribut je povinný. Posledním jednoduchým typem je omezení (restriction). Tím se rozumí definice omezení nad některým ze základních datových typů.

```
<xs:simpleType name="mujtyp">             <!-- název omezeného typu -->
  <xs:restriction base="xs:string">       <!-- určení omezovaného typu -->
    <xs:enumeration value="Completed"/>  <!-- omezení výčtem -->
    <xs:enumeration value="Pending"/>    <!-- omezení výčtem -->
  </xs:restriction>
</xs:simpleType>
<!-- užití v jiné definici -->
<xs:attribute name="status" type="xs:mujtyp" use="required" />
```

Příklad ukazuje definici podtypu `mujtyp` a jeho následné použití při definici pravidla atributu. Omezení je definováno výčtem, ale to není jediná možnost. XSD poskytuje více možností, jak omezení provést. Poskytovány jsou i regulární výrazy (syntakticky jsou obdobné jazyku Perl), které se umisťují do elementu `pattern`. Místo výčtů lze napsat

```
<xs:pattern value="Completed|Pending" />
```

Základními datovými typy, jež jsou v XSD poskytovány, jsou čísla, řetězce, data (od slova datum) a několik dalších bez vlastní kategorie. Hlavním typem pro čísla je typ `decimal`, který značí desetinné číslo (např. -63.57). Pro celá čísla se používá `integer`. Existují i mnohé další typy jako `negativeInteger` či `nonPositiveInteger`, ale s pomocí omezení a regulárních výrazů by bylo možné tyto podtypy velmi rychle vytvořit. Typ `string` je řetězec, který zpracovává všechny znaky. Poskytnut je i typ `normalizedString`, který nezpracovává znaky CL, RF a tabulátory. Typ `token` kromě těchto tří znaků automaticky zpracovává i vícenásobné mezery v řetězci a mezery na začátku a konci řetězce. Typ `ID` slouží pro klíč, typy `IDREF` a `IDREFS` slouží pro cizí klíče. Poskytnuty jsou z důvodu kompatibility s DTD i další speciální znakové podtypy. Datové typy se vyznačují nutností přísně dodržovat specifikaci. Typ `DATE` vyžaduje datum ve formátu RRRR-MM-DD (R-rok, M-měsíc, D-den) a nic jiného není přípustné. Typ `time` se specifikuje jako HH:MM:SS (H-hodina, M-minuta, S-sekunda). Během užívání těchto typů je vždy nutné mít nastudovanou správnou specifikaci. Mezi další typy patří například typ `anyURI` pro uri, typ `hexBinary` pro binární data v šestnáctkové soustavě či typ `boolean` pro pravdivostní hodnoty.

Ve specifikaci se lze dočíst, jaká omezení lze nad daným typem provádět. Již jsem použil výčet a regulární výraz. Tyto možnosti omezení může využívat většina typů (avšak ne všechny, například typy `IDREF` či `boolean`). Existují omezení *totalDigits* a *fractionDigits* pro omezení počtu cifer (celkový počet a počet za řádovou čárkou). Omezení *length*, *minlength* a *maxlength* slouží k určení délky řetězce. Omezení *maxExclusive*, *minExclusive*,

maxInclusive, *minInclusive* slouží pro určení největšího a nejmenšího čísla (Inclusive připouští i mezní číslo). Následně ukáži definici celého čísla dělitelného pěti, jež je menší než sedmdesát.

```
<xs:simpleType name="mojecislo">
  <xs:restriction base="xs:int">
    <xs:pattern value=".*0"/>
    <xs:pattern value=".*5"/>
    <xs:maxInclusive value="70"/>
  </xs:restriction>
</xs:simpleType>
```

Nejdůležitější částí v XSD jsou komplexní typy, které jsou schopny zachytit strukturu XML dokumentu. Tvoří se stromová struktura, která obvykle končí jednoduchými typy. Každý komplexní element (není to jen atomický text) se zabaluje do elementu `ComplexType`. Uvnitř se definují jeho atributy a další vnořené elementy. Ty se zabalují do možností `all` (každý z vnořených elementů právě jednou v jakémkoliv pořadí), `sequence` (každý z elementů implicitně jednou, v pevně daném pořadí) nebo `choice` (pouze jedna z následných možností). Pro popis dokumentů s volnou strukturou bude bohužel nutné použít `choice` a vyjmenovat všechny možné sekvence [6]. Počet výskytů daného elementu lze omezit atributy `minOccurs` a `maxOccurs`. Konstanta `unbounded` slouží pro nekonečno. Pokud bych chtěl v elementu povolit i textový obsah, uvedu na začátek definice pravidla atribut `mixed` s hodnotou `true`. Pokud je třeba definovat pouze textový obsah, užije se `simpleContent`. Element `any` a `anyAttribute` slouží pro vložení libovolného elementu či atributu.

```
<xs:element name="task" mixed="true">
  <xs:complexType>
    <xs:sequence>
      <xs:choice>
       <xs:sequence>
        <xs:element name="note"
                    type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
        <xs:element name="description" type="xs:string"/>
       </xs:sequence>
       <xs:sequence>
        <xs:element name="description" type="xs:string" />
        <xs:element name="note"
                    type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
       </xs:sequence>
      </xs:choice>
    </xs:sequence>
    <xs:attribute name="status" type="xs:mujtyp" use="required"/>
  </xs:complexType>
</xs:element>
```

Na příkladu je definováno, že element jménem `task` je komplexního typu. Povinný je atribut `status`, jehož typ byl definován dříve. V elementu se mohou vyskytovat libovolné textové řetězce (díky atributu `mixed`). Uvnitř se buď na začátku, nebo na konci nachází element `description` a libovolné množství elementů `note`. Problémem je, že nelze vyjádřit,

že element `description` by mohl být umístěn libovolně mezi elementy `note`. O pokročilejší práci s pozicemi se bude snažit další technologie, RelaxNG.

2.7 RelaxNG

Technologie *Regular language for XML Next Generation* (dále jen RelaxNG) je další z možných náhrad DTD [4]. Vznikala souběžně s XSD a poskytuje obdobné možnosti. Samozřejmě má své výhody i nevýhody. I o nich bude následující sekce.

RelaxNG schéma je validní XML ze jmenného prostoru `rng`. Zapouzdření může probíhat více způsoby, nejjednodušší je užití elementu `element` definující kořenový element XML.

```
<element xmlns="http://relaxng.org/ns/structure/1.0"
         name="task">
     <text/>
</element>
```

Atribut `xmlns` mi umožňuje definovat celý jmenný prostor jako *rng*. To mi ušetří místo, neboť většina elementů bude z tohoto jmenného prostoru. Atribut `name` určuje název elementu (v tomto případě kořenového), element `text` značí, že se uvnitř elementu `task` mohou vyskytovat data. Pokud bych chtěl říci, že element je prázdný, použil bych element `empty`. Uvnitř elementu mohu definovat, co se v něm objeví (vnořené elementy, text, atributy aj.) Element `attribute` slouží pro definici atributu.

```
<element name="task">
  <element name="placeholder">
    <empty/>
  </element>
  <element name="description">
    <attribute name="writer">
      <text/>
    </attribute>
    <text/>
  </element>
</element>
```

Na příkladu je kořenový element `task`. Ten obsahuje právě jeden prázdný element `placeholder` a právě jeden element `description`. Tento element má atribut `writer` a obsahuje libovolná textová data. Jak lze vidět, bylo by vhodné umět lépe kontrolovat množství i pořadí elementů.

Element `group` definuje přesné pořadí elementů v něm obsažených a všechny zapouzdřuje. Element `choice` definuje, že se může použít právě jeden ze vzorů v něm. Element `interleave` definuje, že vzory v něm se mohou použít v libovolném pořadí. Element `optional` uvádí vzor nepovinný. Element `oneOrMore` slouží pro opakovaní vzoru (alespoň jedno opakování). Existuje i varianta pro libovolné množství opakování se jménem `zeroOrMore`. V RelaxNG bohužel nelze specifikovat přesný počet elementů jako například v XSD pomocí `minOccurs` a `maxOccurs`. Na druhou stranu je možné díky elementu `interleave` definovat dokumenty s velmi volnými pravidly [6]. Použití některých z popsaných elementů demonstruji na následujícím příkladu.

```
01.<element name="task">
02.  <interleave>
03.    <group>
04.        <choice>
05.          <element name="short">
06.            <text/>
07.          </element>
08.          <group>
09.            <element name="longA">
10.              <text/>
11.            </element>
12.            <element name="longB">
13.              <text/>
14.            </element>
15.          </group>
16.        </choice>
17.        <optional>
18.          <element name="worker">
19.            <text/>
20.          </element>
21.        </optional>
22.    </group>
23.    <zeroOrMore>
24.      <element name="note">
25.        <text/>
26.      </element>
27.    </zeroOrMore>
28.  </interleave>
29.</element>
```

Na příkladu je ukázán element `task`. V něm je povinná skupina elementů v daném pořadí (od řádku 3 do řádku 22). Nejprve musí být jeden z následujících. Buď element `short`, nebo zapouzdřená dvojice elementů `longA` a `longB` – přesně v tomto pořadí (zapouzdření začíná řádkem 8). Následně může být přítomen volitelný element `worker`. Všechny elementy v této části (řádky 3 až 22) mohou být promíchány s obsahem na řádcích 23 až 28, neboť to je další z možností v `interleave`. Ve výsledku se v kterémkoliv elementu (uvnitř kořenového) může na libovolném místě objevit libovolný počet (řádek 23) elementů `note`.

Problémem DTD byla absence datových typů. Ta je zde vyřešena. Pro obecný obsah byl zatím v této práci používán pouze element `text`, avšak RelaxNG podporuje daleko více. Pro výčet povolených hodnot se užívá element `value`. Tento element lze užít i v elementu `except` pro vyřazení některých hodnot z povolených. Pro použití datového typu se užívá element `data`, jehož atribut `type` definuje zvolený typ. Dále se užívá atribut `datatypeLibrary`, který indikuje, odkud datový typ pochází. Implicitně se používají stejné datové typy jako u XSD. Omezení datového typu lze vytvořit uvedením elementu `param` do příslušného elementu `data`. Element `param` má atribut `name`, který uvádí název omezení (v případě implicitního XSD např. `pattern` pro regulární výraz). Samotné omezení je definováno atomem elementu `param`.

```
<element name="root"
         xmlns="http://relaxng.org/ns/structure/1.0"
         datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <zeroOrMore>
    <element name="Word">
      <data type="token">
        <param name="pattern">[AaÁá].*</param>
      </data>
    </element>
  </zeroOrMore>
  <attribute name="Starts">
    <choice>
      <value>A</value>
      <value>Letter A</value>
    </choice>
  </attribute>

</element>
```

Na příkladu je v elementu `root` libovolný počet elementů `Word`. Element `root` má povinný atribut `Starts`, který připouští právě jednu ze dvou uvedených hodnot. Element `Word` obsahuje řetězce začínající na první písmeno abecedy.

V úvodu jsem naznačil, že zabalení dokumentu RelaxNG lze provádět i jiným způsobem. Tím jsou definice. Je možné definovat si šablony, které se často opakují a následně se na ně odkazovat. V tomto případě je kořenovým elementem `grammar`. Část s pravidly se umisťuje do elementu `start`. Jednotlivé definice se dávají do elementu `define`. Na definici se odkazuje elementem `ref`.

```
<grammar xmlns="http://relaxng.org/ns/structure/1.0">
  <start>
    <element name="tasks">
      <zeroOrMore>
      <element name="task">
        <group>
          <element name = "problem">
            <ref name="goodtoknow"/>
          </element>
          <element name = "soultion">
            <ref name="goodtoknow"/>
          </element>
        </group>
      </element>
      </zeroOrMore>
  </start>

  <define name="goodtoknow">
    <element name="When">
      <text/>
    </element>
```

```
    <element name="How">
      <text/>
    </element>
  </define>
</grammar>
```

V tomto případě bude v elementu `problem` i elementu `solution` požadována dvojice elementů `When` a `How`. Toto by mělo pro základní pochopení RelaxNG stačit.

2.8 Schematron

Jazyk Schematron je technologií, která umožňuje tvořit schéma XML pomocí sady pravidel [6]. Využívá k tomu nástroje XPATH, kdy pro elementy či atributy nalezené jedním XPATH musí platit jiná podmínka XPATH. Schematron následně tyto podmínky využije k sestavení XSL transformace, která uživateli vypíše hlášení pro jednotlivá definovaná pravidla. V souhrnu se jedná o opačný přístup než předchozí technologie. Schematron povoluje v XML vše, co není pravidly zakázáno.

Schematronové schéma je validní XML, jehož kořenovým elementem je element `Schema`. Jmenný prostor schematronu je *sch*. Jelikož jsou všechny elementy ve schematronovém schématu z tohoto jmenného prostoru, mohu na začátek uvést

```
<schema xmlns="http://www.ascc.net/xml/schematron">
<!-- Pravidla zde -->
</schema>
```

Existuje i volně dostupný standard ISO/IEC 19757-3 pro schematron. Pro plné využití tohoto standardu se doporučuje [9] začínat

```
<?xml version="1.0" encoding="utf-8"?>
<iso:schema xmlns="http://www.ascc.net/xml/schematron"
            xmlns:iso="http://purl.oclc.org/dsdl/schematron"
            queryBinding='xslt2'
            schemaVersion='ISO19757-3'>
<iso:ns prefix='xhtml' uri='http://www.w3.org/1999/xhtml'/>
<!-- Pravidla zde -->
</iso:schema>
```

Tím je možné používat klasický i standardizovaný Schematron, transformovat se bude pomocí XSLT2.0, přesné ISO je uvedeno v atributu `schemaVersion`. Dále jsem předvedl, že se dá definovat platný jmenný prostor zkoumaného XML pomocí elementu `ns`. V dalším budu používat pouze klasický schematron z prvního příkladu.

Schéma je tvořeno pravidly. Každé pravidlo je zabaleno do elementu `pattern` a má atribut `name`. Do něj programátor uvádí název, popř. popis daného pravidla. Pravidlo se skládá z již zmíněných XPATH výrazů. Elementy či atributy, kterých se část pravidla týká, se zabalují do elementu `rule`. Samotný XPATH výraz pro vyhledávání příslušných údajů se nachází v atributu `context`. Uvnitř pravidla se nachází testy pro vybrané prvky. Pravidla jsou umístěna v elementech `assert` a `report`. Test je umístěn do atributu `test`. Obsah elementu `assert` je vypsán, pokud test selže. Komplementárně k tomu je obsah elementu

`report` vypsán, pokud test uspěje. Pro výpis údajů z XML je možné užít element `name` – údaje jsou opět vybírány XPATH výrazem v atributu `path`. Uvedu příklad. Nechť je dáno XML *dokument.xml* ve tvaru

```
01.<?xml version="1.0" encoding="utf-8"?>
02.<root>
03.        <task id="1">
04.          Buy a green item
05.          <note>Best value</note>
06.        </task>
07.        <task id="2">
08.          Buy a red item
09.          <tasked></tasked>
10.        </task>
11.          <task id="1">
12.          <tasked>John</tasked>
13.          Ignore a blue item
14.          <extra>Too cheap</extra>
15.        </task>
16.</root>
```

a schematronové schéma ve tvaru

```
<schema xmlns="http://www.ascc.net/xml/schematron" >
  <pattern name="Valid elements">
     <rule context="task">
       <assert test="count(note|tasked) = count(*)">
       Invalid element
       <name path="./*[name()!='tasked' and name()!='note']"/>
       detected
       </assert>
     </rule>
     <rule context="tasked">
       <report test="normalize-space(.) = ''">
       Empty tasked detected
       </report>
     </rule>
  </pattern>

  <pattern name="ID check">
     <rule context="task">
       <report test="count(../task[@id = current()/@id]) &gt; 1">
       ID duplicity!
       </report>
     </rule>
  </pattern>
</schema>
```

Výsledkem validace bude výstup

```
dokument.xml:3: error: report:
  ID duplicity!
dokument.xml:9: error: report:
  Empty tasked detected
dokument.xml:11: error: assertion failed:
  Invalid element extra detected
dokument.xml:11: error: report:
  ID duplicity!
```

Schéma definuje dvě sady pravidel. První sada slouží k určení přípustných elementů. Obsahuje dvě pravidla. První pravidlo kontroluje, že pro každý element `task` existují pouze vnořené elementy se jmény `note` a `tasked`. Pokud bude přítomen nějaký jiný element, vypíše se jeho jméno. Schéma při validaci XML detekovalo neplatný element `extra`, který se nachází v kontextu elementu `task` na řádku 11. Druhým pravidlem v této sadě je pravidlo, které říká, že každý element `tasked` něco obsahuje (text nebo jiné elementy, nesmí být prázdný). Připomínám, že tento relativní XPATH projde všechny výskyty. Alternativně by bylo možné napsat XPATH výraz *//tasked*. Při validaci byl nalezen prázdný element `tasked` v kontextu elementu `tasked` na řádku 9. Druhou sadu pravidel jménem `ID check` tvoří jediné pravidlo. Toto pravidlo zajišťuje, že pokud má element `task` atribut `id` (existenci žádné pravidlo nezaručuje), pak je tento atribut ve všech elementech `task` v souboru jedinečný. Validace zjistila, že nastala duplicita v kontextu `task` na řádcích 3 a 11 (tyto dva elementy `task` mají stejné ID).

Jak lze vidět, schematron je v zásadě velmi jednoduchý nástroj, který se podobá mnohým nástrojům pro tvorbu automatických testů. Nejdůležitější částí jsou XPATH výrazy, jejichž dobrá znalost je základem pro práci se schematronem. Schematron se často používá jako doplnění jiné technologie, jako je např. RelaxNG. V každé části se definují ty pravidla, který daný jazyk zvládá lépe. RelaxNG si skvěle poradí s datovými typy a přesným pořadím elementů, schematron se postará o přesné počty elementů a jejich obsah.

Kapitola 3

Řešení problému

V této kapitole nejprve stručně představím formát LMF, pro které práci tvořím.

3.1 LMF

`Lexical markup framework` (dále jen LMF) je dnes již standardem (ISO 24613:2008), který slouží pro uchovávání slovníkových dat ve formátu XML. Jeho návrh vychází z DTD [10]. Dále budu předpokládat verzi 16 tohoto standardu.

LMF má za kořenový element `LexicalResource`. V něm se mohou vyskytovat globální informace, které se vztahují na celý soubor. Má jediný atribut jménem `LexicalResource`, který udává číslo použité verze LMF (v mém případě 16). Soubor dále obsahuje alespoň jeden slovník značený elementem `Lexicon`. V praxi jsou slovníky velmi rozsáhlé a v souboru se z toho důvodu vyskytuje pouze jeden `Lexicon`. Samotný slovník se skládá z hesel (záznamů), které se umisťují do elementů `LexicalEntry`. Každý záznam musí obsahovat právě jeden element `Lemma`, který podle standardu obsahuje výraz, podle něhož vyhledávám. Výsledek, který pro hledaný výraz vyhledávám, se umisťuje do různých elementů v závislosti na typu slovníku. Nejčastěji jsem se setkal s elementem `Sense`. Tím samozřejmě standard nekončí. Ve zmíněných elementech může být zanořeno velké množství dalších elementů, které mají blíže specifikovat výslednou sémantiku, jež se na konci nachází.

Data se do LMF XML slovníku nikdy neumisťují jako atomický text, ale musí být vždy umistěna v atributech elementu `feat`, viz. výtah z DTD

```
<!ELEMENT feat EMPTY>
<!ATTLIST feat
    att      CDATA #REQUIRED
    val      CDATA #REQUIRED>
```

Element `feat` nikdy neobsahuje vnořené elementy a vždy má atributy `att` a `val`. Do atributu `att` se píše typ údaje a do atributu `val` se umisťuje hodnota údaje. Uvedu příklady elementů `feat` ze skutečných slovníků.

```
<feat att="partOfSpeech" val="commonNoun"/>
<feat att="language" val="en"/>
<feat att="grammaticalGender" val="masculine"/>
<feat att="grammaticalGender" val="feminine"/>
```

Lze si všimnout, že třetí a čtvrtý element `feat` nejspíše označují stejnou skutečnost (záznam je rodu ženského). To je velký problém LMF, neboť není vytvořen standard pro hodnoty atributů `att` a `val`. To znamená, že každý výrobce slovníku si do těchto atributů může napsat cokoliv chce, a přitom dodržovat standard. Nikdo také nekontroluje sémantiku ostatních elementů (nezbývá než doufat, že jsou užity správně). Některé z těchto elementů umožňují vzájemné odkazování přes atributy mnoha jmen, ale vždy typů ID a IDREF(S). Následuje příklad, který dodržuje syntaxi LMF.

```
01.<?xml version="1.0" encoding="UTF-8"?>
02.<LexicalResource dtdVersion="16">
03.  <GlobalInformation>
04.    <feat att="languageCoding" val="ISO 639-3"/>
05.    <feat att="fromLanguage" val="cz"/>
06.  </GlobalInformation>
07.  <Lexicon>
08.    <feat att="tolanguage" val="en"/>
09.    <LexicalEntry id="5br79hash2L">
10.      <feat att="partOfSpeech" val="adjective" />
11.      <Lemma>
12.        <feat att="writtenForm" val="ahoj"/>
13.      </Lemma>
14.      <WordForm>
15.        <feat att="language" val="en"/>
16.        <feat att="pronunciation" val="he-lou"/>
17.      </WordForm>
18.      <Sense>
19.        <Equivalent>
20.          <feat att="language" val="cz"/>
21.          <feat att="writtenForm" val="ahoj"/>
22.        </Equivalent>
23.        <SenseExample id="1">
24. <feat att="writtenForm" val="Ahoj světe!">
25. <Equivalent>
26. <feat att="writtenForm" val="Hello World!">
27.           <feat att="note" val="my favourite line, lol">
28. </Equivalent>
29. </SenseExample>
30.      </Sense>
31.     </LexicalEntry>
32.  </Lexicon>
33.</LexicalResource>
```

Můj soubor (LMF slovník) dodržuje standardní dtd verze 16 (řádek 2). Globální informace nejspíše říkají, že slovníky budou české a specificky kódované (řádky 5 a 4). Soubor obsahuje jeden slovník s jedním záznamem. Globální informací slovníku je nejspíše, že bude překladovým slovníkem do angličtiny (řádek 8). V záznamu směle několikrát zopakuji, o jakých jazycích je řeč (řádek 15 a řádek 20). Jako programátor si na řádek 27 přidám informaci indikující, že mám rád uvedený příklad užití). Standard mi v tom nijak nebrání. Údaj, že

se jedná o přídavné jméno (řádek 10), je také chybný, ale to standard samozřejmě neošetřuje. Části hesla umisťuji sémanticky zhruba správně. Hledané heslo do elementu `Lemma`. Hledaný význam do elementu `Sense` včetně příkladu užití v elementu `Equivalent` a další informace jsem rozmístil volně do okolí, např. výslovnost do elementu `WordForm`. Ačkoliv bych se slovníkem mohl pyšně běžet pro razítko standartu, lze vidět, že návrh mohl být i jiný, ideálně lepší. Toto je důvodem, proč se v projektu snažím zjistit skutečné schéma slovníku a pokouším se analyzovat data v elementech `feat`. Potkal jsem slovník, který o sobě prohlašuje, že dodržuje dtd verzi 16 a přitom používá místo elementu `SenseExample` element `Collocation`, který toto dtd nezná.

3.2 Generování schématu

Mám XML soubor, který obsahuje LMF slovník. Obvykle se jedná o statisíce až miliony řádků. Úkolem je tento soubor analyzovat tak, aby bylo jasné, jak vypadá vnitřní struktura. K tomu budu vytvářet schéma, které bude mít co nejvíce omezující podmínky. Jak lze vyvodit z teoretické části, uvažoval jsem nad technologiemi XSD, RelaxNG a Schematron. Zvolil jsem Schematron. XSD a RelaxNG mají dva přístupy ke tvorbě schématu. Vychází z principu, že dovoleno je jen to, co programátor určí. První přístup postupně kopíruje maximální možnou strukturu dokumentu. To vede k tvorbě horizontálně náročného schématu, ve kterém čtenář snadno ztratí přehled o úrovni zanoření. To obzvláště platí v případě mých slovníků, kde se běžně vyskytují soubory se sedmi a více stupni zanoření. Druhý přístup nejprve detekuje podtypy a definuje je zvlášť. Tento přístup pro čtenáře zjednodušuje stupně zanoření, ale na druhou stranu má velmi velké nároky na čtenářovu paměť, neboť vzniká velké množství těchto podtypů a čtenář se bude pravděpodobně muset k jejich definicím často vracet. Vzniká tím schéma, které je příliš vertikálně náročné. Oproti tomu schematron povoluje vše a já pouze specifikuji, co nastat nesmí. Vhodným přístupem k tvorbě pravidel ve schematronu je tvorba sady podmínek pro každou množinu elementů definovanou co nejjednodušším XPATH výrazem. Nevýhodou samozřejmě je, že čtenář musí znát XPATH, ale to není o moc náročnější, než se naučit syntaxi XSD či RelaxNG. Jádrem bude

```
<pattern name="Identifikátor">
  <rule context="XPATH výběr">
    <assert test="XPATH podmínka">
      Popis testu
    </assert>
    <!-- další pravidla zde -->
  </rule>
</pattern>
```

Toto jádro se zopakuje tolikrát, kolik různých XPATH cest v dokumentu existuje. Tím se pokryjí pravidla pro všechny elementy. Zbývá vytvořit samotná pravidla. Čtenář pravděpodobně schéma netvoří proto, aby jej užil na soubor, ze kterého jej vytvořil. Schéma nejspíše chce proto, aby co nejlépe dané xml pochopil, a proto by samotná pravidla měla jasně reflektovat informace, které pro něj mohou být užitečné.

První informací, která čtenáře může zajímat, jsou názvy vnořených elementů pro daný element (specifikovaný kontextem XPATH). Vytvořím podmínku, která bude říkat, že vnořené elementy mohou mít jména pouze těch elementů, které byly nalezeny jako vnořené. Tato podmínka se vytvoří dle následujícího vzoru

```
<assert test="count(*) = count(aaa|bbb|ccc)">
  Warning - unexpected child element inside - /root
  <!-- context je /root -->
</assert>
```

Podmínka říká, že uvnitř specifikovaného kontextu se mohou nacházet pouze vnořené elementy se jmény `aaa`, `bbb` a `ccc`. Toho se docílí podmínkou v atributu `test` elementu `assert`. Část `count(*)` spočítá všechny vnořené elementy (hvězdička odpovídá libovolnému názvu v jednom stupni zanoření relativně od současného elementu). Následná část `count(aaa|bbb|ccc)` spočítá počty všech povolených elementů (jednotlivé názvy se oddělují svislicí). Tyto dvě části si musí být rovny. To znamená, že všechny vnořené elementy musí mít daný název. Pokud element nemá mít žádné vnořené elementy, položí se pravá strana rovnice nule.

Další informací je informace o povoleném počtu jednotlivých elementů. Stále se snažím o co nejlepší čitelnost pro čtenáře, a proto informaci transformuji na povolený počet stejně pojmenovaných sourozenců. Tím se ze struktury, která tvoří různý počet pravidel (v předchozím případě tři, pravidlo pro počet elementů `aaa`, `bbb` a `ccc`), stane struktura, která má vždy právě jedno pravidlo pro sebe sama. Podmínka pravidla bude vypadat následovně

```
<assert test="count(../bbb) &gt;= 2 and count(../bbb) &lt;= 4">
        Warning - unprecedent number of sibling elements
        - bbb - withnin context - /root/bbb
        <!-- context je /root/bbb -->
</assert>
```

Uvedená podmínka specifikuje, že na uvedeném XPATH musí vedle sebe být 2, 3 nebo 4 elementy `bbb`. Podmínka se skládá z konjunkce dvou tvrzení. První říká, že počet specifikovaného elementu v rodiči (zastoupeno dvěma tečkami) musí být větší než nebo rovno danému číslu. Druhá část podmínky je symetricky vytvořena pro menší a rovno. Za povšimnutí stojí, že znaky *menší než* a *větší než* musí být psány pomocí znakových entit, neboť patří mezi řídící značky. Pokud je minimální počet nula elementů, lze část podmínky vynechat. Pokud je počet pevně dané číslo, pak lze podmínku zjednodušit na rovnost danému číslu

```
<assert test="count(../bbb) = 3 >
```

Dále by čtenáře mohly zajímat atributy, jež se v kontextu daného elementu mohou nacházet. Cílem bude vypsat všechny povolené kombinace, jež byly zjištěny. Opět bude u každé kombinace nutno specifikovat počet elementů v této kombinaci (kvůli principu vše implicitně povoleno). Vzniklé pravidlo bude mít tvar

```
<assert test="count(./@*) = 0 or ./@ba and ./@bb and count(./@*) = 2">
  Warning - Unexpected set of attributes inside - /root/first
  <!-- context je /root/bbb -->
</assert>
```

Na ukázce je stanoveno, že element buď nemá žádný atribut, nebo má právě dva atributy jmény `ba` a `bb`. Celá podmínka se skládá z disjunkcí dílčích podmínek pro každou množinu povolených atributů. Dílčí podmínka je tvořena z konjunkcí podmínek pro názvy atributů

a jedné podmínky pro počet atributů. V případě žádného atributu z dílčí podmínky zůstane jen část o počtu atributů, a to nula (ukázáno na příkladu). Samotná podmínka pro jméno je velmi jednoduchá. Jedná se o relativní XPATH, které ze současného uzlu (znak tečka) požaduje daný atribut. Podmínka pro počet atributů vybírá z uzlu atributy libovolného jména (znak hvězdička).

Zbývá zabudovat pravidlo, že všechna data v LMF musí být umístěna v atributech a ne jako atomický text uvnitř elementu. To vyřeší pravidlo, které jako konstantu umístím vždy do kořenového elementu.

```
<assert test="normalize-space(.) = ''">
  Warning - file contains atomic text
</assert>
```

Vznikne schematronové schéma, ve kterém čtenář snadno nalezne informace o atributech a elementech. Další věcí, kterou by bylo rozumné sledovat, jsou samotná data. Data se vždy nachází v atributech `att` a `val` elementu `feat`, a právě o této záležitosti bude následující sekce.

3.3 Analýza elementu feat

Úkolem je prezentovat čtenáři data v elementu `feat` tak, aby se v nich vyznal, a mohl je dále používat. Další podmínky pro schematron nepovažuji za rozumné, neboť by zásadním způsobem zvětšily obsah každého elementu `pattern` a opět by nastal problém s příliš dlouhým kódem, ve kterém by se čtenář ztrácel. To je důvodem, proč budu analýzu elementu `feat` provádět do zvláštního souboru. Pro další použitelnost této analýzy opět volím formát XML. Předpokládám, že čtenáře budou zajímat hodnoty atributu `val` pro daný atribut `att` v daném kontextu. XML bude mít formát

```
<?xml version="1.0" encoding="utf-8"?>
<analyzed maxchoices="NUM">
  <feat path="XPATH VÝRAZ" valueofatt="TEXT" choices="NUM">
    <valueofval samples="NUM">TEXT</valueofval>
  </feat>
</analyzed>
```

Kořenovým elementem je element `analyzed`. V tom se pro každý klíč – tvořený XPATH cestou k danému elementu `feat` ve zkoumaném slovníku a hodnotou atributu `att` v daném XPATH – vytvoří element `feat`. Dvojice, která je tomuto elementu unikátním klíčem, se uchovává v atributech `path` a `valueofatt`. Atribut `choices` čtenáři prozrazuje, kolik různých hodnot atributu `val` bylo pro daný klíč nalezeno. Jednotlivé hodnoty `val` jsou vypsány pod sebou jako obsah elementu `valueofval`. Tento element má atribut `samples`. Ten určuje, kolikrát se tato hodnota (v kontextu klíče) vyskytla v souboru. Jelikož by se implicitně vypsal celý slovník, je nutné omezit výpisy u těch klíčů, které mají příliš mnoho možností. K tomu slouží atribut `maxchoices` u kořenového elementu. Pokud je u některého elementu `feat` větší množství možností `choices` než je číslo `maxchoices`, pak se pro něj nebudou vypisovat elementy `valueofval`. Hodnotu `valueofval` lze nastavit na `inf`, a poté se vypisuje vše.

Celou analýzu v praxi předvedu na příkladu. Nechť je dáno XML

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <feat val="global" att="type" />
  <first>
    <feat val="first" att="type" />
  </first>
  <first firstat="value">
    <second>
      <feat att="secondA" val="A1" />
      <feat att="secondB" val="B1" />
    </second>
  </first>
  <first>
    <feat att="type" val="first" />
    <second>
      <feat att="secondA" val="A1" />
      <feat att="secondA" val="A2" />
      <feat att="secondA" val="A3" />
      <feat att="secondB" val="B2" />
    </second>
  </first>
</root>
```

Následné schematronové schéma je zkráceno. Z šesti pravidel jsou ukázány dvě a jsou ořezány výpisy při nesplnění podmínek.

```
<schema xmlns="http://www.ascc.net/xml/schematron" >
  <pattern name="pattern1">
    <rule context="/root">
      <assert test="count(../root) = 1">Warning</assert>
      <assert test="count(*) = count(feat|first)">Warning</assert>
      <assert test="count(./@*) = 0">Warning</assert>
      <assert test="normalize-space(.) = ''">Warning</assert>
    </rule>
  </pattern>
...
  <pattern name="pattern5">
    <rule context="/root/first/second/feat">
      <assert test="count(../feat) &gt;= 2 and count(../feat) &lt;= 4">
      Warning</assert>
      <assert test="count(*) = 0">Warning</assert>
      <assert test="./@val and ./@att and count(./@*) = 2">
      Warning</assert>
    </rule>
  </pattern>
...
</schema>
```

Analýza elementu `feat` dopadne následovně

```
<?xml version="1.0" encoding="utf-8"?>
<analyzed maxchoices="inf">

  <feat path="/root/feat" valueofatt="type" choices="1">
    <valueofval samples="1">global</valueofval>
  </feat>

  <feat path="/root/first/feat" valueofatt="type" choices="1">
    <valueofval samples="2">first</valueofval>
  </feat>

  <feat path="/root/first/second/feat" valueofatt="secondA" choices="3">
    <valueofval samples="1">A2</valueofval>
    <valueofval samples="1">A3</valueofval>
    <valueofval samples="2">A1</valueofval>
  </feat>

  <feat path="/root/first/second/feat" valueofatt="secondB" choices="2">
    <valueofval samples="1">B2</valueofval>
    <valueofval samples="1">B1</valueofval>
  </feat>
</analyzed>
```

Čím složitější strukturu má vstupní soubor, tím delší je schematronové schéma. Čím delší je vstupní soubor, tím delší je XML s analýzou. Např. testovací slovník o 208496 řádcích má schéma o 314 řádcích.

Kapitola 4

Skript

V této kapitole se zabývám výsledným skriptem. Nejprve popíši, jak byl skript vytvořen.

4.1 Princip činnosti

Pro tvorbu skriptu jsem použil jazyk `Python3.4`. Kostru skriptu tvoří následující funkce – *ParseArg*, *CheckFiles*, *Repair*, *ParseXML*, *GenSch* a *Analyze*.

Funkce *ParseArg* slouží ke zpracování parametrů z příkazové řádky. Používám k tomu modul `argparse`. Funguje na principu, že vytvořím analyzátor metodou `ArgumentParser()`, nastavím přijímané přepínače metodou `add_argument()` a spustím zpracování příkazové řádky metodou `parse_args(argv)`.

Funkce *CheckFiles* kontroluje existenci uživatelem zadaného souboru a otevírá jej. Uživatel může požádat o opravu častých chyb v XML. To obstarává funkce *Repair*. *Repair* je ovšem nutné užívat opatrně, neboť odstraňuje znaky z private use area UTF-8 (např. znaky pro výslovnost, které uživatel nepodporuje). Rovněž je odstraněna informace o dodržovaném DTD (např. uživatel nemá LMF 16 k dispozici). Upravený soubor se jmenuje `temp_file.xml` a existuje po zbylou dobu běhu skriptu. Obecně ale platí, že by na vstupu mělo být validní (well-formed) XML. Nejčastějšími prohřešky proti validitě bývají neukončené prázdné elementy (kupř. `feat`) a nepoužívání znakových entit uvnitř hodnot atributů.

Funkce *ParseXML* je nejdůležitější část skriptu, která analyzuje soubor a ukládá si data. Pro analýzu XML jsem zvolil modul `xml.sax`. Tento modul vytváří syntaktický analyzátor metodou `xml.sax.make_parser()`. Jedná se o nepostradatelnou část skriptu, a proto by se měl uživatel ujistit, že tvorbu analyzátoru z `xml.sax` podporuje. Analýza probíhá systematickým průchodem po XML souboru, přičemž je mi dovoleno definovat reakce na některé události. Tou může být začátek elementu, konec elementu, detekce atomického textu apod.

Pro ukládání dat ze slovníku jsem vytvořil následující slovník jménem *elements*.

```
elements = {
XPATH: [FEATURE, [ATRIBUTES], LEVEL, CURROCCUR, MAXOCCUR, MINOCCUR],
        ...}
```

Klíčem slovníku je XPATH výraz reprezentující platnou cestu k elementu v daném XML. Existuje záznam pro každou cestu. Hodnotou na tomto klíči je seznam několika hodnot. Hodnota `FEATURE` je pravdivostní hodnota *True* nebo *False*. Indikuje, jestli je popisován element `feat`. To se zjistí tak, že XPATH elementem `feat` končí. Hodnotami seznamu

ATRIBUTES jsou seznamy postihující všechny existující kombinace atributů v elementech daným XPATH. LEVEL poskytuje informaci o zanoření (aby se nemusela pořád znovu a znovu počítat z XPATH). Zanoření kořenového elementu je 1. Čím větší zanoření, tím větší číslo. Informace OCCUR říká, kolik sourozenců stejného XPATH vedle sebe může být. MINOCCUR slouží pro minimální počet, MAXOCCUR pro maximální počet a CURROCCUR je pomocná hodnota při výpočtu předchozích dvou. V praxi může záznam vypadat následovně

```
elements = {
"/a/b/c":[False,[[],[at1,at2],[at3],[at1,at2,at3]], 3, 0, 4, 2],
        ...}
```

Na ukázce je element c zanořený v elementu b, který je zanořený v kořenovém elementu a. Není to feat, musí se vyskytovat po 2-4 sourozencích a je v zanoření číslo tři. Buď nemá žádné atributy, nebo má pouze atribut at3, nebo má právě dva atributy at1 a at2, nebo má právě tři atributy at1, at2 a at3.

Důležité je zmínit, že analýza bude probíhat ve dvou průchodech. V prvním průchodu se zejména zjišťují přítomné XPATH a v druhém se již správně počítají OCCUR. V jediném průchodu by mohlo docházet k chybám v MINOCCUR – Pokud by byla detekována nepřítomnost elementu jen před tím, než bude samotný uvažovaný element poprvé nalezen. Nechť je příkladem XML

```
<root>
  <aaa></aaa>
  <aaa><bbb /></aaa>
  <aaa><bbb /><bbb /></aaa>
  <aaa><bbb /></aaa>
</root>
```

V jediném průchodu by se ukázalo, že elementů bbb na XPATH */root/aaa/bbb* by mohlo být po jednom až dvou výskytech. Správně jsou ovšem nula až dva výskyty, ale při průchodu prvním elementem aaa ještě neexistovala informace o tom, že v něm existuje element bbb, a proto se jeho MINOCCUR nikdy nenastavila na nulu. To se stane až při druhém průchodu, kdy je */root/aaa/bbb* známé už od začátku. Při druhém průchodu se také začínají sbírat data pro analýzu obsahu elementů feat (ty jsou snadno detekovány pomocí informace FEATURE). Pro elementy feat existuje následující slovník.

```
features = {(XPATH, VAL_ATT):[TYPE, LEN, {VAL_VAL: USAGE}]}
```

Do tohoto slovníku se ukládá přesně to, co bude potřeba při tvorbě XML s analýzou popsanou v předchozí kapitole. Klíč je dvojice (XPATH, VAL_ATT) – to odpovídá XPATH cestě a hodnotě (value) atributu att. TYPE je pomocná informace. Hodnota LEN je předpřipravená délka následujícího slovníku, jehož klíčem VAL_VAL je hodnota atributu VAL a hodnotou je USAGE – počet výskytů dané klíčové hodnoty. Např.

```
features = {
("/a/b/feat","rod"):["NORMAL",3,{"mužský":10,"ženský":13,"střední":8}],
        ...}
```

Na příkladu je element `FEATURE` zanořen v elementu `b`, a ten je zanořen v kořenovém elementu `a`. Záznam se týká těch elementů, jehož atribut `att` má hodnotu *rod*. Nalezeny byly tři možnosti atributu `val`. Desetkrát se našla hodnota *mužský*, třináctkrát hodnota *ženský* a osmkrát hodnota *střední*. Důležité je, že skript vždy očekává element `feat` s právě dvojicí atributů `att` a `val`. Pokud je zjištěný jiný tvar elementu `feat`, nelze XML analýzu vytvořit. Vše se tvoří z událostí pro začátek a konec elementu. Atomický text se nezpracovává, neboť žádný neočekáváme a pravidlo pro něj ve schematronu je konstantní.

Funkce *GenSch* a *Analyze* slouží k vytvoření schematronového schématu a XLM analýzy elementu *feat*. Schéma se píše na stdout, Analýza se umisťuje do souboru specifikovaného uživatelem. Tyto funkce pouze správně interpretují data ve slovnících *elements* a *features* popsané výše.

4.2 Užití

Skript se volá z příkazové řádky pomocí příkazu

```
python lmfsg.py -f FILE [-r] [-a file] [-l NUMBER] [-h]
```

Python je testován na verzi 3.4 s podporou modulů `re`, `sys`, `argparse`, `math`, `xml.sax`, `os` a `unicodedata`, včetně modulů, na kterých jsou zmíněné moduly závislé.

Skript se jmenuje *lmfsg.py*. Povinný parametr je `-f`, případně `--file`, hodnotou je cesta k testovanému XML souboru. Nepovinný přepínač `-r`, případně `--repair` opravuje některé časté problémy s XML, ale za cenu možné ztráty dat. Nepovinný parametr `-a`, případně `--analyze` vytvoří XML soubor s analýzou elementu `feat`. Povinnou součástí tohoto přepínače je i název výstupního XML souboru. Číslo udávající hodnotu `maxchoices` – výpis hodnot `val` jen, pokud jich má méně než toto číslo – lze ovlivnit parametrem `-l`, popř. `--limit`. Přijímáno je přirozené číslo nebo -1, které slouží pro neomezený počet výpisů (toto je implicitní chování při neuvedení parametru). Nepovinný přepínač `-h` zobrazí nápovědu, kterou automaticky generuje *argparse*.

Pro validaci XML souboru proti schematronu jsem používal Jing. Tento nástroj slouží nejen k validaci schematronu, ale i k validaci RelaxNG. Použití nástroje je následovné.

```
java -jar jing.jar SCHEMA XMLFILE
```

Lze vidět, že se jedná o nástroj v Javě (je nutné mít javu nainstalovanou). Parametr `jing.jar` je cesta k balíku Jing na počítači. Parametr `SCHEMA` je cesta k souboru se schématem. Parametr `SCHEMA` je cesta k XML souboru, který je cílem validace. Jing následně na `stdout` vypíše všechny porušené podmínky. Jing ovšem není stavěn na velké soubory, a při testech s velkými slovníky obvykle při validaci došla paměť.

Kapitola 5

Shrnutí

V práci jsem vytvořil skript, který umožňuje analyzovat XML LMF slovníky pomocí schématu. Schematronové schéma slouží k pochopení struktury slovníku a další tvořený XML soubor slouží k výpisům dat z atributu `feat` v daném slovníku. Toto může sloužit uživateli pro další práci se slovníky. Schéma i XML jsou dále zpracovatelné a uživatel jej může využít např. ke tvorbě překladů hodnot některých elementů `feat`. Práce jej naučila, jak to udělat (XSLT). Jedno z možných řešení představím zde. Nechť je dán příklad XML

```
<root>
  <aaa>
    <feat att="001" val="AAA" />
    <feat att="002" val="AAA" />
    <feat att="001" val="BBB" />
  </aaa>
  <feat att="001" val="BBB" />
</root>
```

Pak XSLT pro transformaci může vypadat dle následujícího vzoru

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

<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <!--hlavní šablona -->
  <xsl:template match="@*|node()">
    <xsl:copy>
        <xsl:apply-templates select="@*|node()" />
    </xsl:copy>
  </xsl:template>

  <!-- šablona pro daný XPATH -->
  <xsl:template match="/root/aaa/feat">
    <xsl:choose>
      <xsl:when test="./@att = '001' and ./@val = 'BBB' ">
        <feat>
        <xsl:attribute name="att">
          <xsl:value-of select="'200'" />
        </xsl:attribute>
```

```
        <xsl:attribute name="val">
          <xsl:value-of select="./@val" />
        </xsl:attribute>
        </feat>
      </xsl:when>
      <!-- místo pro další pravidla na daném XPATH -->
      <xsl:otherwise>
        <xsl:copy-of select="." />
      </xsl:otherwise>
    </xsl:choose>
  </xsl:template>

  <!-- místo pro šablony jiného XPATH -->

</xsl:stylesheet>
```

Výsledkem transformace bude

```
<root>
  <aaa>
    <feat att="001" val="AAA"/>
    <feat att="002" val="AAA"/>
    <feat att="200" val="BBB"/>
  </aaa>
  <feat att="001" val="BBB"/>
</root>
```

Na příkladu jsem ukázal, jak lze změnit element `<feat att="001"val="BBB"/>` nacházející se v XPATH výrazu */root/aaa/feat* na element `<feat att="200"val="BBB"/>`. Hodnoty atributů měněného i cílového elementu mohou být brány z mé analýzy. Tím lze navrhnout další projekt, který by pro dva specifikované jazyky dělal automatický návrh těchto změn. Jak je ovšem na přiloženém médiu ukázáno v adresáři `showcase2`, analýza může poodhalit velmi laxní přístup k ukládání informací ve slovnících (téměř všechny podstatné informace mají atribut `att` na hodnotě *hint*). Toto znemožňuje vytvořit obecný překlad dvou slovníků. Skript by musel ze slovníku pochopit celou syntaxi a hlavně sémantiku předem neznámého jazyka včetně jeho výjimek, a proto bych pro budoucí zamyšlení doporučoval jít směrem – skript pro daný typ slovníku daných jazyků, které programátor perfektně zná. V tomto případě by s využitím heuristik mělo být možné návrh vytvořit, ačkoliv autoři slovníků by i zde mohli programátorovi házet klacky pod nohy tím, že by v jednom slovníku byly gramatické jevy zapsány v jiném počtu elementů `feat` než v jiném slovníku, a ideálně by si tito autoři pro sloučené gramatické jevy vytvořili vlastní zkratky.

Literatura

[1] Bray, T.; Paoli, J.; Sperberg-McQueen, C. M.; aj.: Extensible Markup Language (XML) 1.0 (Fifth Edition). online.
URL http://www.w3.org/TR/xml/

[2] Clark, J.: XSL Transformations (XSLT)Version 1.0. online.
URL http://www.w3.org/TR/xslt

[3] Clark, J.; DeRose, S.: XML Path Language (XPath) Version 1.0. online.
URL http://www.w3.org/TR/xpath/

[4] Clark, J.; Makoto, M.: RELAX NG Specification. online.
URL http://relaxng.org/spec-20011203.html

[5] Fallside, D. C.; Walmsley, P.: XML Schema Part 0: Primer Second Edition. online.
URL http://www.w3.org/TR/xmlschema-0/

[6] Kosek, J.: XML schémata. online.
URL http://www.kosek.cz/xml/schema/xmlschema.pdf

[7] Kosek, J.: XSLT v příkladech. online.
URL http://www.kosek.cz/xml/xslt/

[8] Nic, M.; Jirat, J.: XPath Tutorial. online.
URL http://zvon.org/xxl/XPathTutorial/General/examples.html

[9] Pawson, D.; Costello, R.; Georges, F.: ISO Schematron tutorial. online.
URL http://www.dpawson.co.uk/schematron/index.html

[10] Tagmatica: LMF DTD REV 16. online.
URL http://www.tagmatica.fr/lmf/DTD_LMF_REV_16.dtd

[11] w3schools.com: Introduction to DTD. online.
URL http://www.w3schools.com/DTD/dtd_intro.asp

[12] w3schools.com: XPath Tutorial. online.
URL http://www.w3schools.com/xpath/default.asp

Příloha A

Obsah DVD

DVD obsahuje adresář `skript` a adresář `text`.

Adresář `text` obsahuje `projekt.pdf` soubor s textem této práce. Zdrojové soubory tohoto dokumentu jsou umístěny v podadresáři `src`.

Adresář `skript` obsahuje výsledný skript `lmfsg.py`. Soubor `license.html` obsahuje licenci ke skriptu. Podadresáře `showcase1` a `showcase2` demonstrují ukázkové vstupy a výstupy.