Transformace XSLT 2.0

Bakalářská práce

Jiří Koch

Pedagogická fakulta Jihočeské univerzity
Katedra informatiky

Transformace XSLT 2.0

bakalářská práce

Autor: Jiří Koch

Vedoucí diplomové práce: Ing. Václav Novák CSc.

České Budějovice 2010

PROHLÁŠENÍ

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách.

V Českých Budějovicích dne 20. dubna 2010

ANOTACE

Abstrakt: XSLT je jazykem pro transformaci dokumentů XML, které jsou velmi rozšířené díky své nezávislosti na platformě a hlavně možnosti oddělit obsah dokumentu od jeho formy. V této práci bych chtěl čtenáře provést po základech XSLT 2.0, konkrétně pak na příkladech ukázat změny, které přinesla nejnovější verze 2.0 oproti verzi předchozí a zmínit jaké problémy mohou nastat při přechodu na tuto verzi. Dále bych chtěl představit některé způsoby použití této technologie, se kterými jsem se v praxi setkal.

ANNOTATION

Abstract: XSLT is a language for XML documents transformation, that are widespread thank to its´ independence on platform and important possibility to separate content of the document from his form. In this thesis, I would like to introduce the basis of XSLT 2.0 and their using to users, concretely show changes that brought latest version 2.0. on examples, and note what problems might arise in the transition to this version. I would also like to introduce some ways to use this technology, with whom I have encountered in practice.

PODĚKOVÁNÍ

Rád bych poděkoval všem, kteří mě při tvorbě této práce podporovali. Nejvíce tedy své rodině, dále kolegům a kamarádům.

OBSAH

ÚVOD

Svět XML technologií se velmi rychle rozšířil kolem nás a dnes se s tímto druhem zpracování dat setkáváme prakticky neustále, a to aniž bychom si to vůbec uvědomovali. Proč se použití této technologie v poslední době tolik rozrůstá a proč je XML výborným řešením pro zajištění výměny strukturovaných textových dat?

Důvodů je hned několik. Důležitá je jednoduchost a srozumitelnost, kterou v sobě tyto dokumenty spojují a to díky tomu, že jsou vytvářeny samotným člověkem podle jeho úsudku a požadavků. Přehlednost a čtení je proto daleko srozumitelnější i při použití v obyčejných textových editorech. Dalším neopomenutelným důvodem je nezávislost na platformě, neboť jej lze bez problémů zpracovávat nejen v jakémkoliv operačním systému, i bez potřeby instalace speciálních programů, tedy v jakémkoliv obyčejném textovém editoru. Tím, že jde o textový soubor bez jakýchkoliv grafických prvků, je zaručena jeho relativně malá velikost, což může být výhodou hlavně při jeho přenosu například mezi softwarovými systémy, stejně tak i při zpracování a jeho zálohování a skladování. Právě oddělení formy a obsahu je hlavním pozitivem tohoto značkovacího jazyka a zaručuje nám, že při budoucím rozšíření či jakýchkoliv úpravách, bude pro nás tato činnost jednodušší.

ZOBRAZENÍ XML DOKUMENTŮ

K tomu, abychom obsah XML mohli prezentovat, potřebujeme nějaký styl a aplikaci, která ho dokáže zpracovat. Ve velké míře se k úpravě zobrazení používá buď jazyk kaskádových stylů (CSS), nebo jazyk XSL (eXtensible Stylesheet Language).

PREZENTACE XML DAT POMOCÍ CSS

Pokud se podíváme na velmi rozšířený jazyk HTML, zjistíme, že spousta editorů a hlavně webových prohlížečů tomuto jazyku „rozumí“ a nepotřebují nic dalšího k tomu, aby obsah HTML dokumentu dokázaly vizuálně prezentovat. Pokud například parser prohlížeče uvidí element <TABLE>, bude vědět, že tento definuje tabulku, nebo element <P> značí odstavec. Prohlížeč zná jasná pravidla, podle kterých bude onen element zobrazen. Pokud chceme u daného HTML elementu upřesnit vizuální podobu, používáme k tomu kaskádové styly (CSS).

Opakem toho jsou elementy v jazyce XML. Například element <auto>, který patří do XML autobazar, neřekne vůbec nic, protože se neví jak ho zobrazit. K tomu můžeme použít třeba již zmiňované CSS.

Vezměme následující příklad, který představuje jednoduchý XML soubor a k němu přiřazený styl CSS. XML otevřeme ve webovém prohlížeči a získáme graficky upravený výstup. Samotnému přiřazení stylu, můžeme vidět na druhé řádce.

Příklad1a:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/css" href="autobazar.css"?>
<autobazar>
    <auto vyrobeno="CZ">
        <znacka>Škoda</znacka>
        <typ>Favorit</typ>
        <rokvyroby>1992</rokvyroby>
        <popis>toto je velmi staré auto</popis>
    </auto>
    <auto vyrobeno="D">
        <znacka>Audi</znacka>
        <typ>Q7</typ>
        <rokvyroby>2008</rokvyroby>
        <popis>toto je nové a velmi drahé auto</popis>
    </auto>
</autobazar>
```

Příklad1b:

```
autobazar {
font-family: Verdana, sans-serif;
font-size: small;
}
znacka {
display: block;
padding: 0.5em;
background: red;
color: white;
font-weight: bold;
font-size: 1.80em;
}
typ{margin-left: 2em;
}
popis{margin-left: 2em;
color: blue;
display: block;
font-size: smaller;
font-style:italic;
padding: 0.5em;
}
```

Příklad1c:

Jak je na příkladu patrné, kaskádové styly klidně poslouží pro grafické zobrazení XML. Co je ale tedy největší slabinou CSS při zobrazení XML? Ona podstatná skutečnost, že nedovedou změnit jeho datovou strukturu!

- CSS zobrazí obsah XML pouze v pořadí v jakém je v souboru zapsán, nelze tedy zobrazit například popis auta před značkou.
- CSS nepodporuje dynamický obsah, tzn., že nedokáže vytvořit na našich stránkách žádné odkazy, ani používat skriptování
- CSS nedovedou zpracovat obsah z jiného dokumentu, který je připojen k našemu XML. Pokud budeme mít připojený jiný XML soubor obsahující například detaily o motoru auta, CSS nejsou schopny k těmto informacím přistoupit a zobrazit je.
- CSS neumí zobrazit atributy elementů, náš atribut vyrobeno se tedy na výstupu nezobrazí

Právě tyto důvody jsou výsledkem toho, že se ve světě XML dokumentů obrací veliká pozornost k použití jazyka XSL.

CÍLE PRÁCE

Cílem mé práce je seznámit čtenáře s hlavními změnami jazyka XSLT 2.0 oproti verzi předcházející, popsat postupy pro přechod na tuto verzi a shrnout využití technologie XSLT v praktických situacích (ODF, OOXML, vícevrstvá aplikace, sharepoint, atd.), se kterými jsem měl možnost se setkat.

CO JE XSL?

Máme-li data uložená v XML souboru, neznamená, že to pro nás bude nějak užitečné, protože pořád potřebujeme napsat program, který bude s daty nějakým způsobem manipulovat. Nejčastěji je manipulací myšleno prezentování těchto dat - jako HTML stránku na webu, jako text posílaný e-mailem, nebo například PDF dokument sloužící k tisku.

Konsorcium W3C začalo vyvíjet jazyk právě pro prezentaci v roce 1998, tedy v době, kdy byl dokončován vývoj XML. Tento jazyk byl pojmenován eXtensible Stylesheet Language (XSL). Cílem vývoje XSL bylo, dostat stylový jazyk, který by překonal limitace jazyka CSS, z nichž některé jsem zmiňoval v úvodu, tzn. vytvořit jazyk nejen pro změnu vizualizace, ale i struktury XML. XSL si tak propůjčilo základy DSSSL (Document Style Semantics and Specification Language), které je stylovým jazykem používaným v SGML aplikacích. Jako další bylo rozhodnuto, že XSL bude využívat syntaxi XML - XSL je tak značkovacím jazykem založeném na XML.

XSL jako takové se skládá ze dvou značkovacích a jednoho textově založeného jazyka. První značkovací jazyk je čistě prezentační a popisuje, jak mají být jednotlivé formátované objekty rozmístěny a zobrazeny na stránce. Tomuto jazyku se tak říká XSL Formatting Objects (XSL-FO). Druhý značkovací jazyk, který je tématem mé práce, definuje jak libovolný soubor (xml, xsl…) transformovat do jiného formátu. Tento jazyk se nazývá XSL Transformation (XSLT). Třetím pak je jazyk, který umí adresovat jednotlivé části XML dokumentů. Ten je označován jako XML Path (XPath) a je často používán třeba s právě zmiňovaným XSLT.

HISTORIE XSLT

XSLT, Extensible Stylesheet Language for Transformation, je jazyk sloužící k transformaci struktury a obsahu XML dokumentů.

XSLT a XPath ve verzi 1.0, které se byly velice úspěšnými, se staly doporučením W3C v listopadu roku 1999. V prosinci roku 2000 bylo XSLT 1.0 následováno konceptem verze 1.1, která obsahovala některé menší změny jako například podporu vícenásobného výstupu dokumentů (<xsl:document>, které je v XSLT 2.0 upraveno a používáno jako <xsl:result-document>) a uživatelsky definovaných funkcí. Jak se ale ukázalo, tak tyto změny byly v konečném důsledku více pracnými či spornými. Proto byl vývoj této verze oficiálně přerušen v srpnu 2001 a nikdy tak nedošlo k jejímu dokončení.

V únoru roku 2001 vydalo W3C seznam požadavků pro verze XSLT 2.0 a XPath 2.0 a od této doby se započaly práce na jejich vývoji. Většina změn v XSLT 2.0 měla vést k zjednodušení věcí a funkcí, které byly v XSLT 1.0 obtížné, jako příklad lze uvést např. grupování.

Dlouhé očekávání se naplnilo nakonec až 23. ledna roku 2007, kdy se XSLT 2.0 stalo oficiálním doporučením organizace W3C.

Dne 21. dubna 2009 pak vyšel návrh úpravy tohoto doporučení pojmenovaný jako XSLT 2.0 (Second Edition). Pokud tento návrh splní podmínky dané konzorciem W3C, stane se poté oficiálním doporučením.

XSLT

XSLT dokument

Dokument XML, který je psaný v jazyce XSLT, je nazýván XSLT styl (XSLT stylesheet) a označován příponou .xsl. Každý takový stylový dokument popisuje, jak mají být **zdrojové dokumenty** konvertovány do **výstupních dokumentů.** Jak už jsem jednou zmínil, výstupem po zpracování XML souboru pomocí stylů XSL může být široká škála formátů. Nejčastěji se jedná o jiný XML dokument, o HTML dokument nebo obyčejný textový dokument. Při použití XSL-FO (XSL – Formátovacích Objektů) pak můžeme jako výstup získat například soubor PDF (Portable Document Format), soubor popisující dvojrozměrnou grafiku - SVG (Scalable Vector Graphics), soubor popisující trojrozměrnou grafiku - VRML (Virtual Reality Modeling Language) a mnoho dalších formátů.

K vykonání transformace je důležité označit ve zdrojovém dokumentu informace, které mají být zpracovány. Hlavní roli v navigaci, shromažďování informací a odkazů v dokumentu zastává jazyk XPath, který je, stejně jako jazyk XSLT, ve verzi 2.0 od 27. ledna 2007 doporučením W3C.

XSLT dokument musí začínat specifikací. Jelikož se vlastně jedná o XML dokument, je nutné, aby obsahoval jeden root element (kořenový element). Ve specifikaci XSLT je kořenový element určený identifikátorem „stylesheet“. Abychom specifikaci XSLT mohli používat, je nutné v elementu stylesheet definovat jmenný prostor a verzi XSLT. Adresa DTD pro specifikaci jmenného prostoru je `http://www.w3.org/1999/XSL/Transform` a jako identifikátor se používá akronym „xsl“. Verze je pak v našem případě uváděna „2.0“. Jak bude celý zápis tohoto kořenového elementu vypadat, můžete vidět níže. Toto je standardní tvar kořenového elementu každého XSLT 2.0 dokumentu.

```
<xsl:stylesheet
version="2.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
</xsl:stylesheet>
```

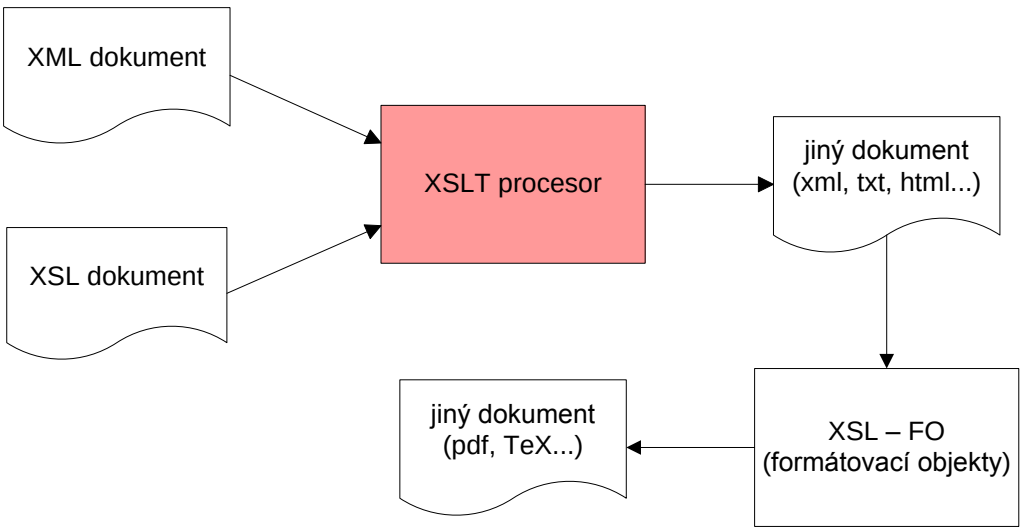
Podle tohoto kořenového elementu pak procesor, chystající se použít tuto šablonu, zjišťuje, o jakou verzi se jedná a následně ji podle toho zpracovává. Způsoby, jakými se procesor zachová, se budeme zabývat v dalších kapitolách.

Transformace

Námi napsaný XSLT styl definuje pravidla (templates), jakými má být XML dokument transformován. Pravidlo (template) vlastně říká něco ve smyslu:

„Pokud najdeš část dokumentu shodnou s tímto, tak zde máš popsáno jakým způsobem ji transformovat.“

O samotnou transformaci se pak stará XSLT procesor, který právě podle pravidel daných v XSL převádí vstupní XML na výstupní dokument. Na obrázku je vidět, že dále můžeme vytvořený dokument transformovat podle formátovacích objektů (XSLT-FO) a získat tak na výstupu mnoho dalších formátů. Můžeme získat stránky HMTL i XHTML, soubory PDF, soubory formátů TeX a mnoho dalších. Možnosti výstupů jsou opravdu bohaté a to je jedna z věcí, která dělá jazyk XSL tolik mnohotvárný a užitečný při praktických činnostech, kdy vytváříme obdobné výstupy za použití měnících se vstupních dat.



Procesory

XSLT 2.0 přidává spoustu nových možností a prvků do XSLT 1.0. Zavádí nové elementy jako například <xsl:for-each-group> nebo <xsl:analyze-string>, dále pak také přidává nové atributy pro XSLT 1.0 elementy. Je pochopitelné, že tyto novinky v XSLT 2.0 nejsou podporovány v procesorech XSLT 1.0. Pokud tedy chceme napsat styly, které budou pracovat jak s procesory XSLT 2.0, tak i s procesory XSLT 1.0, musíme být schopni si zjistit, jakou verzi náš procesor vlastně podporuje a poskytnout mu alternativní kód, který se použije v případě, kdy budeme XSLT 2.0 šablony zpracovávat procesorem XSLT 1.0.

Informace o verzi procesoru jsme schopni získat přímo za pomoci stylového xsl dokumentu, konkrétně použitím funkce system-property(), která vrací string odpovídající hodnotě argumentu použitého v této funkci.

Pro naše účely je nejdůležitější argument xsl:version, jenž vrací údaj o verzi procesoru. Tím jsme schopni analogicky určit jaká verze jazyka XSLT je jím podporována. Konkrétně vrací string s hodnotou „2.0“ pro procesory XSLT 2.0 a číslo 1.0 pro procesory XSLT 1.0. Zde je třeba si povšimnout, že výstupem funkce system-property() není pokaždé hodnota typu string, ale například při použití již zmiňovaného argumentu xsl:version vrací procesory XSLT 1.0 výstup číselného typu.

V následující tabulce jsou ukázány atributy, které může funkce system-property() obsahovat. Ne všechny jsou však podporovány XSLT 1.0 procesory, což můžeme vidět v tabulce ve třetím sloupci. V dokumentaci každého procesoru by však mělo být možné najít přesné informace o tom, které z atributů jsou konkrétně pro něj podporovány. Pokud atribut procesorem podporován není (například máme-li XSLT 1.0 procesor a chtěli bychom získat výstup použitím atributu xsl:is-schema-aware), dostaneme na výstupu prázdný string.

ATRIBUTY	POPIS ATRIBUTU	XSLT 1.0
xsl:version	*verze XSLT podporovaná procesorem*	ano
xsl:vendor	*identifikace výrobce procesoru*	ano
xsl:vendor-url	*URL adresa procesoru*	ano
xsl:product-name	*jméno XSLT procesoru*	
xsl:product-version	*verze procesoru*	
xsl:is-schema-aware	*výstup „yes“ pokud platí, „no“ pokud neplatí*	
xsl:supports-serialization	*výstup „yes“ pokud procesor umí serializovat výstup, „no“ pokud neumí*	
xsl:support-backwards-compatibility	*výstup „yes“ pokud procesor umí pracovat ve zpětně kompatibilním módu, „no“ pokud neumí*	

Styl, který vytváříme pro otestování vlastností procesoru, může obsahovat pouze jediný element template. Tento element navíc může obsahovat v atributu match jako vstup kořenový uzel XML dokumentu. To proto, že ze samotného XML dokumentu nebudeme v tomto případě používat vůbec žádné informace.

Abychom se dozvěděli základní vlastnosti námi použitého procesoru, vytvoříme si teď jednoduchou ukázkovou šablonu, která nám prozradí všechny informace, jež nás zajímají.

Začneme klasickou hlavičkou pro XSLT dokument zmiňovanou výše. Jako výstup nám postačí obyčejný text. Hlavička XSLT 2.0 dokumentu v tomto případě tak bude následující:

```
<xsl:stylesheet version="2.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:output method="text" />

...

</xsl:stylesheet>
```

Otestujeme tedy některé z vlastností XSLT procesoru. Samotné tělo námi vytvářeného dokumentu může mít pak například takovouto strukturu:

```
<xsl:template match="/">
<xsl:choose>
<xsl:when test="number(system property('xsl:version')) >
1.0">
<xsl:text>Procesor XSLT verze </xsl:text>
<xsl:value-of select="system property('xsl:version')" />
<xsl:text>&#xA;Jméno procesoru: </xsl:text>
<xsl:value-of select="system property('xsl:product
name')" />
<xsl:text>&#xA;Verze procesoru: </xsl:text>
<xsl:value-of select="system property('xsl:product
version')" />
<xsl:text>&#xA;URL adresa procesoru: </xsl:text>
<xsl:value-of select="system property('xsl:vendor url')"
/>
</xsl:when>
<xsl:otherwise>
<xsl:text>Procesor XSLT verze 1.0</xsl:text>
<xsl:text>&#xA; Výrobce procesoru: </xsl:text>
<xsl:value-of select="system property('xsl:vendor')" />
<xsl:text>&#xA;URL adresa procesoru: </xsl:text>
<xsl:value-of select="system property('xsl:vendor-url')"
/>
</xsl:otherwise>
</xsl:choose>
</xsl:template>
```

Na výstupu získáme při použití procesoru SAXON 8.4B, což je verze, která již podporuje XSLT 2.0, následující text s námi požadovanými informacemi:

Procesor XSLT verze 2.0
Jméno procesoru: SAXON
Verze procesoru: 8.4
URL adresa procesoru: http://www.saxonica.com/

Při použití procesoru MSXML3, s podporou pouze XSLT 1.0, dostaneme tyto informace:

Procesor XSLT verze 1.0
Výrobce procesoru: Microsoft
URL adresa procesoru: http://www.microsoft.com/

Jak můžeme vidět, zjištění základních informací není vůbec nic složitého a přitom je tato věc velmi důležitá při naší další práci s XSLT šablonami a volbou správného procesoru. Dalšími funkcemi, které bychom při zjišťování dovedností našeho procesoru mohli použít, jsou například function-available() nebo element-available().

Upgrade stylu XSLT 1.0 na XSLT 2.0

Vezměme si situaci, kdy již máme nějaké své stylové šablony ve verzi XSLT 1.0 vytvořené a rádi bychom je převedli na verzi 2.0. Prvním krokem je začít používat procesor XSLT 2.0 a druhým pak samotné převedení šablony na verzi XSLT 2.0

Použití XSLT 1.0 stylu v procesoru XSLT 2.0

Prvním krokem upgradu XSLT 1.0 stylu, je začít používat procesor XSLT 2.0. Jakou verzi XSLT stylu používáme, zjistí procesor díky atributu *version* v kořenovém elementu *<xsl:stylesheet>*. Tento atribut nám tedy říká jaká verze XSLT bude použita bez ohledu na to, jaké prvky budeme do stylu vkládat. Nic nám tedy nebrání v tom, abychom měli části stylu, které používají XSLT 1.0 a části jiné, které používají XSLT 2.0.

Pokud XSLT 2.0 procesor narazí na nějakou část stylu, která používá XSLT 1.0, snaží se automaticky pracovat ve zpětně-kompatibilním (backwards-compatible) režimu, to znamená, že se chová jako procesor XSLT 1.0. Tady je ale třeba dávat pozor, neboť ne všechny procesory tento režim podporují. V dřívější době, tedy kolem vzniku a po vzniku XSLT 2.0 specifikace, byly vytvářeny jako upgrady existujících procesorů XSTL 1.0. Je ale dost nepravděpodobné, že nové procesory psané pro XSLT 2.0, budou doplněny kódem pro podporu zastaralých funkcí. Jak jsme si ale ukazovali v kapitole Procesory, není zase až tak těžké si podporu zpětně-kompatibilního režimu u procesoru ověřit pomocí funkce *system-property()* a atributu *xsl:support-backwards-compatibility*.

Je ale velmi důležité si uvědomit, že každá verze procesoru pracuje odlišně. XSLT 2.0 procesory mají tendenci být striktnější než procesory starší.

Jednoduše si to můžeme vysvětlit třeba na elementu *<xsl:template>*, který by neobsahoval atribut *match*, ale pouze atributy *mode* a *priority*. Procesor XSLT 2.0 by v případě, že by narazil na takovou část šablony, ohlásil chybu (error), jelikož atributy *mode* a *priority* jsou bez atributu *match*, který nám říká na jaký uzel má být šablona použita, irelevantní. Navzdory tomu procesor XSTL 1.0 šablonu akceptuje.

Pokud ale procesor zpětně-kompatibilní režim podporuje, nemusíme se bát žádných velkých problémů a měl by šablonu ve verzi 1.0 v pořádku zpracovat. Pokud přeci jen k nějakým potížím dojde, bude se jednat buď o výše uvedený problém, kdy XSLT 1.0 procesor akceptuje části šablony, ve kterých XSLT 2.0 procesor bude hlásit chybu nebo prostě o problém se špatně uživatelem napsanou šablonou.

Převedení šablony na XSLT 2.0

Pokud tedy máme styl verze 1.0, který pracuje v XSLT 2.0 procesoru (za použití zpětně-kompatibilního režimu), můžeme provést druhý krok a to samotné převedení stylu. Začneme jednoduše tím, že změníme hodnotu atributu *version* v kořenovém elementu *<xsl:stylesheet>* na hodnotu *2.0*. Toto velmi výrazně ovlivní naší snahu o upgrade našeho stylu.

Další změnou, kterou musíme provést je přetypování. XPath 1.0 podporoval implicitně změny typů prvků, tzn., pokud jsme zadali jako hodnotu určité funkce, která má obsahovat hodnotu typu string, hodnotu číselnou, převedl ji sám automaticky na string. XPath 2.0 nám v takovém případě zobrazí chybu, že jsme zadali špatný typ hodnoty. Tyto změny se týkají hlavně jazyka XPath.

Drobné změny najdeme i u zpracování prázdných sekvencí v jazyce XPath 2.0. Pokud v XPath 1.0 použijeme prázdnou sekvenci v momentě, když je očekávaná hodnota typu *number*, je to stejné, jako kdybychom místo prázdné sekvence použili *NaN* (nečíslo). Naopak pokud v XPath 2.0 použijeme

pro číselnou funkci nebo operátor prázdnou sekvenci, na výsledku dostaneme také prázdnou sekvenci. Podívejme se na příklad u numerické funkce *floor()*, která na výstupu dává nejbližší menší celé číslo. Nepoužijeme-li žádný atribut *rating*, pak

```
floor(@rating)
```

vrací *NaN* pro XPath 1.0 a prázdnou sekvenci pro XPath 2.0. Řešení této nekompatibility, v momentě kdy chceme v XPath 2.0 získat na výstupu *NaN*, je takové, že explicitně zkonvertujeme sekvenci hodnotu typu *number* pomocí funkce *number()*. Výsledek bude vypadat následovně:

```
floor(number(@rating)) = 'NaN'
```

Hlavní změna, na kterou si musíme dát pozor v XSLT, nastane v případě, kdy použijeme instrukce, které očekávají výběr jediného uzlu, třeba u atribut *select* u elementu <*xsl:sort*> k výběru více uzlů. V takovém momentě se v XSLT 1.0 použije k řazení první z vybraných uzlů, v XSLT 2.0 tím však vyvoláme chybu. XSLT 2.0 se v takových situacích snaží použít všechny vybrané položky.

Když vybereme sekvenci uzlů použitím <*xsl:value-of*> dostaneme v XSLT 1.0 hodnotu prvního uzlu, v XSLT 2.0 budou na výstupu mezerami oddělené hodnoty všech vybraných uzlů.

Pokud budeme vybírat sekvenci uzlů pomocí atributu *value* v instrukci <*xsl:number*>, pro XSLT 1.0 bude opět výstupem hodnota prvního uzlu, pro XSLT 2.0 to bude víceúrovňové číslo (např.: 5.4.3) složené ze všech vybraných hodnot.

Kdyby pro nás bylo z principu chování XSLT 1.0, tedy výběr pouze prvního prvku sekvence, žádoucí, můžeme na konci výrazu využít predikát *[1]*. Pokud

tedy budeme mít takovýto element:

```
<xsl:value-of select="AUTA | MOTOCYKLY | KOLA" />
```

přepsáním na následující tvar získáme chování XSLT 1.0:

```
<xsl:value-of select="(AUTA | MOTOCYKLY | KOLA)[1]"
                                    />
```

Nejjednodušší způsob jak zjistit místa v kódu, která bude třeba přetypovat, je prosté spuštěním šablony a sledování chyb, jež se v průběhu jejího zpracování objeví. Budeme muset provést některé z následujících změn:

- Využít predikát *[1]* v situacích, kdy chceme vybrat první hodnotu sekvence
- Převedení hodnot typu *number* a *Boolean* na *string* pomocí funkce *string()*
- Převedení hodnot typu *string* a *number* na *Boolean* pomocí funkce *boolean()*
- Převedení hodnot typu *string* a *Boolean* na *number* pomocí funkce *number()*

XSLT 2.0 styly v procesorech XSLT 1.0

V předchozích částech jsme se zabývali tím, jak procesor XSLT 2.0 zpracovává stylový dokument ve verzi 1.0 a jaké je třeba udělat změny v tomto dokumentu, aby byl běh transformace bezproblémový. Nyní se budeme věnovat situaci přesně opačné. Jaké jsou způsoby zajištění správného běhu vašeho XSLT 2.0 stylu v případě, že bude spouštěn v procesorech XSLT 1.0.

Stejně jako umí procesory fungovat ve zpětně-kompatibilním režimu, aby tak mohli zpracovávat styly z dřívějších verzí XSTL, umí také procesory fungovat v režimu opačném, tedy v dopředně-kompatibilním (forwards-compatible). Tento režim slouží analogicky k zpracovávání stylů z novějších verzí XSLT a při jeho použití jsou jednoduše ignorovány tyto syntaktické problémy:

- Top-level elementy, které procesor nerozpozná
- Atributy, které v elementech nerozpozná
- Hodnoty atributů, které nerozpozná a které mohou být z hlediska jazyka ignorovány

Procesor bude dále ignorovat jakékoliv volání funkcí nebo syntaxí XPath, které nebude znát. To vše samozřejmě v případě, kdy není očekáváno provedení těchto výrazů. Pokud je však provedení nutné, je možné použít podmíněnou logiku k přesnému definování možností výstupu. Příkladem nám může být například převod textu malými písmeny na text s VELKÝMI písmeny. Při použití procesoru XSLT 2.0 a novějšího na to využijeme jednoduchou funkci *upper-case()*, při použití starší verze procesoru trochu složitější funkci *translate()* z XSLT 1.0.

```
<xsl:choose>
  <xsl:when test="number(system-
  property('xsl:version')) >= 2.0">
    <xsl:value-of select="upper-case(.)" />
  </xsl:when>
  <xsl:otherwise>
     <xsl:value-of select="translate(.,
     'abcdefghijklmnopqrstuvwxyz',
     'ABCDEFGHIJKLMNOPQRSTUVWXYZ')" />
  </xsl:otherwise>
</xsl:choose>
```

Ve chvíli, kdy bychom potřebovali funkci *upper-case()* vykonat, ale procesor by ji nerozpoznal, vyvolali bychom chybu. Takto má ale procesor logickou možnost, vykonat alternativní část kódu.

To samé platí i u instrukcí, které nejsou rozpoznány, ale měly by být zpracovány. Použijeme opět logickou podmínku a dáme tak procesoru možnost vykonat instrukci s alternativním kódem. Například chceme-li vytvořit grupovaný výstup pomocí XSLT 2.0 elementu *<xsl:for-each-group>*, ale nižší verze procesoru ho nezná, nabídneme alespoň kód s výstupem negrupovaným:

```
<xsl:choose>
  <xsl:when test="number(system-
property('xsl:version')) >= 2.0">
    <xsl:for-each-group select="/autobazar/auto"
    group-by="značka">
    </xsl:for-each-group>
  </xsl:when>
  <xsl:otherwise>
    <xsl:apply-templates select="/autobazar/auto" />
  </xsl:otherwise>
</xsl:choose>
```

Využití elementu <xsl:fallback>

V předchozí části jsme zmiňovali použití podmíněné logiky v případech, kdy procesor XSLT 1.0 narazí na nepodporovanou část kódu (instrukce, funkce…). To ale není jediná možnost jak se s tímto vypořádat. Dalším způsobem poskytnutí alternativního kódu je použití elementu <*xsl:fallback*>. Tento element se vkládá do těla instrukce, které chceme poskytnout alternativní kód. Pokud procesor, procházející instrukci, neví co s ní dělat, neboť ji nezná, zpracuje obsah elementu <*xsl:fallback*> namísto toho, aby nám ohlásil chybu.

Vezměme si příklad s instrukcí <*xsl:for-each-group*>, zmiňovaný o pár řádků výše. Abychom se nemuseli zbytečně zatěžovat zjišťováním, o jakou verzi procesoru se jedná a teprve podle tohoto zjištění poskytovat náhradní kód, využijeme element <*xsl:fallback*>. Procesor tak dojde k instrukci <*xsl:for-each-group*>, zjistí, že ji nezná a zpracuje alternativní kód, jež jsme mu poskytli.

```
<xsl:for-each-group select="/autobazar/auto"
group-by="značka">
  ...
  <xsl:fallback>
    <xsl:apply-templates select="/autobazar/auto" />
  </xsl:fallback>
</xsl:for-each-group>
```

Využívání <*xsl:fallback*> je určitě velmi šikovné a ušetří nám práci se sestavováním podmínek, ale to je také zároveň jeho nevýhoda, jelikož pro testování se nám může hodit více právě <*xsl:choose*> a <*xsl:when*>. Můžeme totiž díky nim testovat více položek a instrukcí. Rozhodnutí, co spíše používat, tak nechám raději na vás.

Využití elementu <xsl:message>

Někdy můžeme požadovat, aby došlo k ukončení zpracovávání šablony a zobrazila se nám zpráva, pokud se nebude jednat o XSLT 2.0 procesor. K tomu přesně využijeme instrukci *<xsl:message>* a nepovinný atribut *terminate*. Výstupem této instrukce může být nejenom text, ale například i jiné instrukce apod. Získáme tak třeba uživatelsky přijatelnější výpis chyby, než jen ten defaultně vypisovaný procesorem. Atribut *terminate* nabývá hodnoty *yes* nebo *no*. *Yes* pro ukončení dalšího zpracovávání, *no* pro pokračování. Podívejme se na jednoduchý příklad:

```
<xsl:template match="/">
  <xsl:if test="number(system-
property('xsl:version')) &lt; 2.0">
    <xsl:message terminate="yes">
      <xsl:text>Pro tento styl je vyžadován XSLT 2.0
procesor.</xsl:text>
    </xsl:message>
  </xsl:if>
<html>
...
</html>
</xsl:template>
```

Nové prvky XSLT 2.0

V této části práce naleznete abecedně seřazený seznam elementů a atributů, které se v XSLT používají. Ten je rozdělen na elementy a atributy, které jsou ve specifikaci od verze 1.0 a na ty, které jsou nově ve verzi 2.0. Jelikož bylo mým cílem popsat a vysvětlit vlastnosti jednotlivých funkcí verze XSLT 2.0, podrobněji jsem se věnoval těmto novým prvkům jazyka.

U každého elementu jsou popsány jeho možné atributy, případně i synovské elementy. Dále k čemu tento element slouží a popsáno jakým způsobem pracuje. Nakonec je většina instrukcí a jejich použití demonstrována na jednoduché ukázce, potažmo ukázkách.

Seznam elementů XSLT 1.0:

- *<xsl:apply-imports>*
- *<xsl:apply-templates>*
- *<xsl:attribute>*
- *<xsl:attribute-set>*
- *<xsl:call-template>*
- *<xsl:choose>*
- *<xsl:comment>*
- *<xsl:copy>*
- *<xsl:copy-of>*
- *<xsl:decimal-format>*
- *<xsl:element>*
- *<xsl:fallback>*
- *<xsl:for-each>*
- *<xsl:if>*
- *<xsl:import>*
- *<xsl:include>*
- *<xsl:key>*
- *<xsl:message>*
- *<xsl:namespace-alias>*
- *<xsl:number>*
- *<xsl:otherwise>*
- *<xsl:output>*
- *<xsl:param>*
- *<xsl:preserve-space>*
- *<xsl:processing-instruction>*
- *<xsl:sort>*
- *<xsl:strip-space>*
- *<xsl:stylesheet>*
- *<xsl:template>*
- *<xsl:text>*
- *<xsl:transform>*
- *<xsl:value-of>*
- *<xsl:variable>*
- *<xsl:when>*
- *<xsl:with-param>*

Seznam atributů XSLT 1.0:

- *xsl:extension-element-prefixes*
- *xsl:exclude-result-prefixes*
- *xsl:use-attribute-sets*
- *xsl:version*

Seznam nových elementů XSLT 2.0:

- *<xsl:analyze-string>*
- *<xsl:character-map>*
- *<xsl:document>*
- *<xsl:for-each-group>*
- *<xsl:function>*
- *<xsl:import-schema>*
- *<xsl:matching-substring>*
- *<xsl:namespace>*
- *<xsl:next-match>*
- *<xsl:non-matching-substring>*
- *<xsl:output-character>*
- *<xsl:perform-sort>*
- *<xsl:result-document>*
- *<xsl:sequence>*

Seznam nových atributů XSLT 2.0:

- *xsl:default-collation*
- *xsl:use-when*
- *xsl:xpath-default-namespace*

Seznam nových funkcí XSLT 2.0:

- *current-group()*
- *current-grouping-key()*
- *format-date()*
- *format-dateTime()*
- *format-time()*
- *regex-group()*
- *unparsed-entity-public-id()*
- *unparsed-text()*

xsl:analyze-string

Instrukce <*xsl:analyze-string*> je používána ke zpracování vstupního řetězce za použití regulárních výrazů – *regex* (regular expression). Velmi užitečná je v případech, kdy potřebujeme zpracovat dokument, který má svou vlastní odlišnou syntaxi, jež není syntaxí jazyka XML. Jako jednoduchý příklad lze uvést třeba atribut, jehož hodnotou je seznam čísel oddělených čárkou. Přesně v takové chvíli velmi oceníme tuto instrukci, která do jazyka XSLT vnáší možnost použití regulárních výrazů, která v předchozí verzi nebyla implementována.

```
<xsl:analyze-string
  select = výraz
  regex = { string }
  flags? = { string }>
  <!-- Tělo: (xsl:matching-substring?,
              xsl:non-matching-substring?,
              xsl:fallback*) -->
<xsl:analyze-string>
```

Jak vidíme, tento element má 3 možné atributy – *select, regex, flags*.

První a zároveň povinný atribut, *select,* je výraz jazyka XPath, jehož hodnotou je řetězec, který bude analyzován. *Regex* je volitelný atribut obsahující regulární výraz, který ovšem nesmí nabývat nulové délky, tzn., že jsou vyloučeny hodnoty jako «regex=""» nebo «regex="[0-9]*"». *Flags* je také volitelný a představuje jeden nebo více prvků „flags“ (používaných v jazyce Perl), které umožňují kontrolovat způsob, jakým je porovnávání (matching) regulárního výrazu prováděno, např. hodnota „m“ znamená multi-line mód.

Samotný proces pak probíhá tak, že je každý výraz rozdělen na sekvenci

výrazů (substrings), které jsou klasifikovány podle toho, zda se shodují s regulárním výrazem – matching substrings, nebo zda se neshodují – non-matching substrings. Tyto výrazy jsou poté jednotlivě zpracovávány v těle elementu `xsl:analyze-string`, buď elementem `xsl:matching-substring` (pokud byly klasifikovány jako matching substring) nebo analogicky elementem `xsl:non-matching-substring` (pokud byly klasifikovány jako non-matching substring). Pokud není nějaký z těchto elementů použit, odpovídající výrazy (substrings) nejsou zpracovány. Ještě dodám, že tyto elementy neobsahují žádné atributy.

Na následujícím příkladě vytvětluji jednoduché použití instrukce `xsl:analyze-string`. Ukazuje nám, jak v elementu <*body*> najít text v hranatých závorkách a tento text vložit, namísto do závorek, do nového elementu <*citace*>.

Příklad1:

```
<xsl:analyze-string select="body" regex="\[(.*?)\]">
   <xsl:matching-substring>
      <citace><xsl:value-of select="regex-
group(1)"/></citace>
   </xsl:matching-substring>
   <xsl:non-matching-substring>
      <xsl:value-of select="."/>
   </xsl:non-matching-substring>
</xsl:analyze-string>
```

Na dalším příkladě vidíme také využití elementu `xsl:analyze-string` pro rozdělení stringu do uzlů a to právě podle regulárního výrazu. Element `xsl:matching-substring` pak generuje uzel dále od zadaného stringu, který se shoduje s regulárním výrazem (`xsl:non-matching-substring` analogicky rozdělí string na části podle regex). Tímto způsobem lze nově nahradit element `xsl:for-each` s funkcí `tokenize()`.

Příklad2a:

```
<xsl:analyze-string  select="body" regex="\S+">
     <xsl:matching-substring>
        <slovo>
            <xsl:value-of  select="."/>
        </slovo>
     </xsl:matching-substring>
  </xsl:analyze-string>
```

Příklad2b (xsl:analyze-string):

```
  <xsl:analyze-string  select="body" regex="\s">
     <xsl:non-matching-substring>
        <slovo>
            <xsl:value-of  select="."/>
        </slovo>
     </xsl:non-matching-substring>
  </xsl:analyze-string>
```

xsl:character-map

Tato instrukce definuje mapu znaků Unicode, které mají být při serializaci výsledného stromu nahrazeny určitými hodnotami typu string. Jde tedy o jednoduchý typ substituce. Využívá se v případě, že chceme mít plnou kontrolu nad znaky na výstupu transformace. Element <*xsl:character-map*> je v XSLT 2.0 navržen jako náhrada za element <*disable-output-escaping*>, který byl používán v předchozích verzích jazyka.

```
<xsl:character-map
  name = qname
  use-character-maps? = qnames>
  <!-- Tělo: (xsl:output-character*) -->
</xsl:character-map>
```

<*xsl:character-map*> je deklarován jako top-level element, to znamená, že musí být vždy díťetem kořenového elementu <*xsl:stylesheet*>. Uvnitř elementu se obvykle vyskytují elementy <*xsl:output-character*>. Element samotný má 2 možné atributy – *name, use-character-maps*.

Atribut *name* je povinným a jak je zřejmé, definuje jméno mapy znaků. Podle tohoto jména může být mapa volána při dalších operacích. Atribut *use-character-maps* je oproti předchozímu volitelný. Obsahuje mezerami oddělený seznam jmen jiných map znaků. Mapování znaků z těchto map pak bude do této mapy znaků nepřímo přidáno.

Element <*xsl:character-map*> obsahuje žádný nebo i několik elementů <*xsl:output-character*>. Element <*xsl:output-character*> definuje mapování mezi jedním Unicode znakem a prvky typu string, které jsou použity k reprezentování znaku v serializovaném výstupu.

Následující příklad nám ukazuje vytvoření dvou map znaků a jejich následné společné využití za pomoci atributu *use-character-maps*.

```
<xsl:character-map name="mezera">
   <xsl:output-character char=" "
   string="&amp;nbsp;"/>
</xsl:character-map>

<xsl:character-map name="spec-znaky-latin">
   <xsl:output-character char="&#161;"
   string="&amp;iexcl;"/>
   <xsl:output-character char="&#162;"
   string="&amp;cent;"/>
   <xsl:output-character char="&#163;"
   string="&amp;pound;"/>
   <xsl:output-character char="&#164;"
   string="&amp;curren;"/>
   ...
</xsl:character-map>

<xsl:character-map name="mapa-znaku"
   use-character-maps="mezera
                       spec-znaky-latin"/>
```

xsl:document

Jedná se instrukci, která slouží převážně k validaci na úrovni dokumentu. Validace probíhá na dočasně vytvořeném stromu a v nově vytvořeném uzlu dokumentu, který vzniká zpracováním konstruktoru uvnitř <*xsl:document*> elementu.

```
<xsl:document
  validation? = "strict" | "lax" | "preserve" | "strip"
  type? = qname>
  <!-- Tělo: sequence-constructor -->
</xsl:document>
```

V elementu <*xsl:document*> je možné použít jeden nebo žádný z těcho 2 atributů – *validation, type*. Tyto atributy nám určují, jakým způsobem bude validace probíhat. Tyto atributy jsou dostupné v případě, že procesor, který používáme je schema-aware (toto lze ověřit pomocí funkce *system-property()*, viz kapitola Procesory).

Atribut *validation* může nabývat těchto hodnot: *strip*, *preserve*, *strict*, *lax*. Pokud zde tento atribut není přítomný, provádí se atribut *default-validation* s hodnotou *strip* v elementu <*xls:stylesheet*>. Hodnota *strip* znamená, že jsou odstraněny všechny anotace a nahrazuje je *xs:untyped* pro elementy a *xs:untypedAtomic* pro atributy. Hodtona *preserve* ponechává anotace tak jak jsou a ponechává ověření na základě hodnot instrukce, která je vytvořila. Hodnoty *strict* a *lax* nejdříve kontrolují, zda je strom well-formed, poté dítě uzlu dokumentu zahrnuje právě jeden uzel elementu, nezahrnuje komentáře, textové uzly, ani zpracovávané instrukce. Top-level element je v případě použití hodnoty *lax* validován oproti definici schématu pokud je tato definice nalezena, *strict* naopak ohlásí chybu, pokud tuto definici nenalezne.

Atribut *type* nám říká, že validace elementů bude probíhat podle zadaných

jmenných typů. Například, definujeme-li *type="mf:invoiceType"*, pak bude validace probíhat proti typu schématu *mf:invoiceType*.

Ukažme si použití <xsl:document> pro validaci na základě typu schématu *mf:invoiceType*:

```
<xsl:variable name="temp">
  <xsl:document type="mf:invoiceType">
    <invoice>
      <xsl:call-template name="build-invoice"/>
    </invoice>
  </xsl:document>
</xsl:variable>
```

xsl:for-each-group

Element <xsl:for-each-group> slouží pro výběr několika položek, které následně spojuje do skupin na základě určitých hodnot či jiných kriterií a s takto vytvořenými skupinami pak dále umožňuje pracovat.

```
<xsl:for-each-group
  select = výraz
  group-by? = výraz
  group-adjacent? = výraz
  group-starting-with? = vzor
  group-ending-with? = vzor
  collation? = { uri }>
  <!-- Tělo: (xsl:sort*, sequence-constructor) -->
</xsl:for-each-group>
```

Tento element má jeden povinný atribut a pět atributů volitelných – *select, group-by, group-adjacent, group-starting-with, group-ending-with, collation*.

Atribut *select* je povinným a říká, jaké prvky budou vybrány pro seskupení. Volitelný atribut *group-by* definuje klíč, který určuje, podle jakých hodnot budou položky řazeny do stejné skupiny. Atribut *group-adjacent* definuje klíč na základě sousedních prvků. *Group-starting-with* začne vytvářet skupinu vždy od každého prvku, který se shoduje se zadaným výrazem. *Group-ending-with* vytváří skupinu vždy od nadcházejícího prvku po prvek shodného se zadaným výrazem, neboli ukončuje skupinu vždy každým prvkem, který se shoduje se zadaným výrazem. Pomocí *collation* definujeme skupinu pravidel pro porovnání hodnot stringů.

Podívejme se na příklad využití tohoto elementu na ukázkovém xml, který byl použit v ukázce CSS v úvodu této práce:

```
<xsl:for-each-group select="/autobazar/auto" group-
by="rokvyroby">
    <xsl:sort select="current-grouping-key()" />
    <p style="background-color:red; font-size:large" >Rok
<xsl:value-of select="current-grouping-key()"/></p>
       <table>
          <xsl:for-each select="current-group()">
     <tr><td><xsl:value-of select="znacka, typ
     "/></td><td>---> <i><xsl:value-of select="popis
     "/></i></td></tr>
                  </xsl:for-each>
        </table>
</xsl:for-each-group>
```

xsl:function

Pomocí elementu <*xsl:function*> je možné vytvářet vlastní, uživatelsky definované funkce, které se využívají stejně jako funkce standardní.

```
<xsl:function
  name = qname
  as? = sequence-type
  override? = "yes" | "no">
  <!-- Tělo: (xsl:param*, sequence-constructor) -->
</xsl:function>
```

<*xsl:function*> je deklarován jako top-level element, to znamená, že musí být vždy dítětem kořenového elementu <*xsl:stylesheet*>. Tento element má jeden povinný atribut a dva atributy volitelné – *name*, *as*, *override*. Elementy <*xsl:param*>, uvedené v těle, specifikují parametry funkce.

Atribut *name* je povinným a určuje jméno funkce. Tímto jménem pak může být funkce volána v dalších částech šablony. Jméno musí mít zadaný prefix, aby se předešlo případným kolizím s funkcemi definovanými v defaultních jmenných prostorech.

Atribut *as* určuje, jakého typu bude hodnota na výstupu. Pokud typ neodpovídá, zahlásí procesor „type error“.

Pokud procesor zjistí, že funkci se stejným názvem a stejným počtem parametrů již zná a atribut *override* je nastaven na hodnotu *no*, zpracuje namísto ní funkci definovanou v základním jmenném prostoru. Pokud bude nabývat hodnoty *yes*, zpracuje tuto funkci.

Příklad1:

```
<xsl:function  name="my:printHelloWorld">
      <xsl:text>Hello World</xsl:text>
</xsl:function>
<xsl:template  match="/aaa">
      <xsl:value-of  select="my:printHelloWorld()"/>
      <xsl:text> :-)</xsl:text>
</xsl:template>
```

Příklad2:

```
<xsl:function  name="my:nasobeni">
      <xsl:param  name="a"/>
      <xsl:param  name="b"/>
      <xsl:value-of  select="$a * $b"/>
</xsl:function>
<xsl:template  match="*">
      <xsl:value-of  select="my:nasobeni(2,3)"/>
</xsl:template>
```

xsl:import-schema

Element <*xsl:import-schema*> slouží k identifikaci schématu, obsahujícího definice datových typů, které jsou uvedené ve stylu. Pozor! Tato instrukce je funkční pro procesory podporující schema-aware. Opět můžeme ověřit pomocí funkce *system-property()*.

```
<xsl:import-schema
  namespace? = uri-reference
  schema-location? = uri-reference />
```

Jedná se o top-level atribut, to znamená, že musí být dítětem <*xsl:stylesheet*>. Tento element má dva volitelné atributy – *namespace*, *schema-location*.

Atribut *namespace* určuje adresu jmenného prostoru, který definuje požadované schéma.

Atribut *schema-location* vystihuje přesné umístění kopie schématu.

Můžeme využít oba tyto atributy naráz, pak ale musí adresa jmenného prostoru korespondovat s importovaným schématem. Pokud uvedeme jen *namespace*, předpokládá se, že procesor bude schopný získat schéma z umístění jmenného prostoru. Jakákoliv chyba během importu schématu (chyba v umístění schématu, chyba v samotném schématu…) bude zobrazena jako compile-time error.

Příklad1:

```
<xsl:import-schema
    namespace = "http://mojeadresa.cz/ns"
    schema-location =
"http://mojeadresa.cz/ns/schema.xsd"/>
```

Příklad2:

```
<xsl:import-schema
    schema-location = "schema.xsd"/>
```

Příklad3:

```
<xsl:import-schema
    namespace = "http://www.w3.org/1999/XSL/Transform"/>
```

xsl:matching-substring

O elementu <*xsl:matching-substring*> jsem se zmiňoval již v úvodu tohoto přehledu nových elementů, konkrétně u elementu <*xsl:analyze-string*>. Můžeme ho využít výhradně jako dítě tohoto elementu. Je používán pro zpracování sekvencí výrazů (substrings), které se shodují s regulárním výrazem uvedeným v atributu *regex*.

```
<xsl:matching-substring>
  <!-- Tělo: sequence-constructor -->
</xsl:matching-substring>
```

xsl:namespace

Tato instrukce není sice příliš často využívaná, ovšem jsou situace, kdy najde uplatnění. Vytváří pro požadovaný element uzel jmenného prostoru. Deklarace tohoto uzlu musí být v elementu umístěna před ostatními instrukcemi.

```
<xsl:namespace
  name = { string }
  select? = expression
  <!-- Tělo: sequence-constructor -->
</xsl:namespace>
```

Jedná se o instrukci, která musí být vždy součástí sekvenčního konstruktoru.

Atribut *name* určuje jmého uzlu, které reprezentuje prefix jmenného prostoru. Ten lze určovat dynamicky.

Atribut *select* nebo obsah elementu <*xsl:namespace*> (může být pouze jedno nebo druhé) nám říkají jaká je URI adresa jmenného prostoru. Hodnotou atributu *select* nesmí být prázdný string, jinak bude zobrazena chyba.

```
<price xsi:type="xs:decimal">
   <xsl:namespace name="xs"
    select="'http://www.w3.org/2001/XMLSchema'"/>
   <xsl:value-of select="23.50"/>
</price>
```

xsl:next-match

Instrukce <*xsl:next-match*> nám dovolí použít více než jedno pravidlo (template) na ten samý uzel ve zdrojovém dokumentu. Díky tomu můžeme vybrat další pravidlo, které bude aplikováno a to i pravidlo ze stejného stylového dokumentu. Tato instrukce musí být umístěna v elementu <*xsl:template*> a nesmí být dítětem jiného <*xsl:next-match*>.

```
<xsl:next-match>
  ( <xsl:with-param> | <xsl:fallback> ) *
</xsl:next-match>
```

Element může být buď prázdný, nebo může obsahovat jeden či více elementů <*xsl:with-param*> a <*xsl:fallback*>. Jak víme z předchozích kapitol, instrukce <*xsl:fallback*> se zpracovává v dopředně-kompatibilním režimu procesoru, tzn., pokud budeme zpracovávat styl procesorem XSLT 2.0, bude tato instrukce ignorována.

Příklad1:

```
<xsl:template match="Neco" priority="2">
  <span class="neco">
  <xsl:next-match />
  </span>
</xsl:template>
```

Příklad2:

```
<xsl:template match="text()" mode="eg:collectTextNodes"
priority="1">
  <xsl:sequence select="." />
  <xsl:next-match />
</xsl:template>
```

xsl:non-matching-substring

O elementu <*xsl:non-matching-substring*> jsem se zmiňoval již v úvodu tohoto přehledu nových elementů, konkrétně u elementu <*xsl:analyze-string*>. Můžeme ho využít výhradně jako dítě tohoto elementu. Je používán pro zpracování sekvencí výrazů (substrings), které se neshodují s regulárním výrazem uvedeným v atributu *regex*. Jedná se analogicky o opačný způsob použití oproti elementu <*xsl:matching-substring*>.

```
<xsl:non-matching-substring>
  <!-- Tělo: sequence-constructor -->
</xsl:non-matching-substring>
```

xsl:output-character

Element <*xsl:output-character*> byl zmíněn již v úvodu tohoto přehledu nových elementů, konkrétně u elementu <*xsl:character-map*>. Definuje mapování mezi znakem a prvky typu string, které jsou použity k reprezentování znaku v serializovaném výstupu.

```
<xsl:output-character
character="character" string="string" />
```

Obsahuje dva povinné atributy – *character*, *string*.

Atribut *character* označuje znak, který bude nahrazen během serializace.

Atribut *string* určuje jakýkoliv textový řetězec, jenž nahradí během serializace na výstupu znak v atributu *character*.

```
<xsl:output-character character="&amp;"
string="&amp;ampersand;"/>

<xsl:output-character  character="@" string=" at "/>
```

xsl:perform-sort

Instrukce <*xsl:perform-sort*> slouží k řazení sekvencí. Jde o analogické řazení, jako provádí instrukce <*xsl:for-each*>, s tím rozdílem, že v tomto případě nejsou již prvky uvnitř sekvence zpracovávány individuálně (prvky jsou pouze seřazeny).

```
<xsl:perform-sort
  select? = výraz
  <!-- Tělo: (xsl:sort+, sequence-constructor) -->
</xsl:sequence>
```

Volitelný atribut *select* zajišťuje výraz či hodnotu, podle níž budou sekvence rozřazeny.

V těle instrukce se pak vyskytuje minimálně jeden element <*xsl:sort*> určující přesná pravidla sortování.

Příklad1:

```
<xsl:perform-sort  select="9,3,5,1">
    <xsl:sort/>
</xsl:perform-sort>
```

Příklad2:

```
<xsl:param name="in" as="xs:date*"/>
   <xsl:variable name="sorted-dates" as="xs:date*">
      <xsl:perform-sort select="$in">
         <xsl:sort select="."/>
      </xsl:perform-sort>
   </xsl:variable>
```

xsl:result-document

Instrukce vytváří nový uzel dokumentu a umožňuje specifikovat jak má být serializován. Díky tomu, můžeme vytvářet několikanásobné výstupy, například vytvořit z jednoho velkého XML souboru více malých XML. Ty pak lze zpracovat třeba do HTML propojenými odkazy.

```
<xsl:result-document
  format? = qname
  href? = { uri-reference }
  validation? = "strict" | "lax" | "preserve" | "strip"
  as? = sequence-type>
  <!-- Tělo: sequence-constructor -->
</xsl:result-document>
```

Jedná se o instrukci, která se může objevit kdekoliv v sekvenčním konstruktoru. Obsahuje čtyři volitelné atributy – *format, href, validation, type.*

Atribut *format* definuje požadovaný formát výstupu.

Atribut *href* určuje umístění výstupního dokumentu.

Atribut *validation* může nabývat těchto hodnot: *strip, preserve, strict, lax*. Pokud zde tento atribut není přítomný, provádí se atribut *default-validation* s hodnotou *strip* v elementu *<xls:stylesheet>*. Hodnota *strip* znamená, že jsou odstraněny všechny anotace a nahrazuje je *xs:untyped* pro elementy a *xs:untypedAtomic* pro atributy. Hodnota *preserve* ponechává anotace tak jak jsou, ověření proběhne na základě hodnot instrukce, která je vytvořila. Hodnoty *strict* a *lax* nejdříve kontrolují, zda je strom well-formed, poté dítě uzlu dokumentu zahrnuje právě jeden uzel elementu, nezahrnuje komentáře, textové uzly, ani zpracovávané instrukce. Top-level element je v případě použití hodnoty *lax* validován oproti definici schématu pokud je tato definice nalezena, *strict* naopak ohlásí chybu, pokud tuto definici nenalezne.

Příklad1:

```
<xsl:template  match="/">
      <xsl:copy-of  select="*"/>
      <xsl:result-document  href="dokument.xml"
                            use-character-maps="mapa">
            <xsl:copy-of  select="*"/>
      </xsl:result-document>
</xsl:template>
```

Příklad2:

```
<xsl:result-document href="{$filename}" format="html">
    <html><body>
        <xsl:value-of select="značka"/>
    </body></html>
</xsl:result-document>
```

xsl:sequence

Element <*xsl:sequence*> se chováním téměř rovná elementu copy-of. Narozdíl od elementu copy-of však nemusí nutně vytvářet nové elementy. Pokud kopírujeme uzly pomocí copy-of, vždy se vytvoří jejich kopie. To zbytečně spotřebovává paměť a klade tudíž větší nároky na stroj při transformaci větších souborů. Na rozdíl od toho element sequence v tomto případě vrací existující uzly. Konstrukce je totožná s elementem copy-of, kde se prvky sequence definují v atributu *select*.

```
<xsl:sequence
  select? = výraz>
  <!-- Tělo: <xsl:fallback>* -->
</xsl:sequence>
```

Jediným možným obsahem tohoto elementu je element <*xsl:fallback*>, ale ten lze zpracovat jen při použití XSLT 1.0 procesoru. Volitelný atribut *select* určuje hodnoty jaké bude instrukce vracet. Ty lze buď vytvářet, nebo využít již existující hodnoty.

```
<xsl:template match="/aaa">
  <cisla>
    <xsl:sequence select="1 to 100"/>
  </cisla>
  <mismatch>
 <!-- vytvoreni hybridniho seznamu pomoci operatoru to-->
    <xsl:sequence select="1 to 5,'a',6,'b'"/>
  </mismatch>
 <!-- použití exitujicich hodnot -->
  <copy>
    <xsl:sequence  select="bbb"/>
  </copy>
</xsl:template>
```

JAZYK XPath 2.0[1]

XPath je jazyk od konsorcia W3C. Jazyk XPath pracuje s abstraktním datovým modelem XML dokumentu. V paměti počítače je pro XML dokument vytvořen strom, nad kterým může XPath vyhodnocovat své výrazy. Pro procházení stromu jsou využívány tzv. osy, pomocí kterých lze definovat cestu v XML dokumentu. V současné době je jazyk XPath ve verzi 2.0 a oproti verzi 1.0 se liší hlavně přidáním podpory více datových typů. Může využít i informace o daném typu s příslušného XML schématu, patřícímu k XML dokumentu. Jazyk XPath je velice často využíván v ostatních dotazovacích jazycích pro XML, které jsou mnohdy jen jakousi nadstavbou jazyka XPath.

Jazyk XPath 1.0 a Jazyk XPath 2.0

V současné době se používají standarty XPath 1.0 i XPath 2.0, které jsou si velmi podobné. XPath 2.0 navazuje na silné stránky XPath 1.0, ke kterým je navíc přidáno několik nových možností.

Omezení Jazyka XPath 1.0

Ačkoli specifikace XPath 1.0 přinesla řadu zjednodušení při práci s XML daty, existuje v ní několik nejasností, které bylo třeba vylepšit. Konsorcium W3C se snaží XPath zdokonalit tak, aby lépe podporoval ostatní standarty jako XQuery, XML Schema, XSLT 2.0. Ve specifikaci XPath 1.0 chybí hlavně podpora různých typů, a proto vznikla specifikace XPath 2.0, která lépe vyhoví požadavkům ostatních standardů.

[1] MAREŠ, Vladimír. *Dotazovací jazyky pro XML a nativní XML databáze* [online]. České Budějovice : Jihočeská univerzita, 2005. 98 s. Bakalářská práce. Pedagogická fakulta Jihočeské univerzity, Katedra informatiky. Dostupné z WWW: <http://home.pf.jcu.cz/~pepe/Diplomky/mares.pdf>.

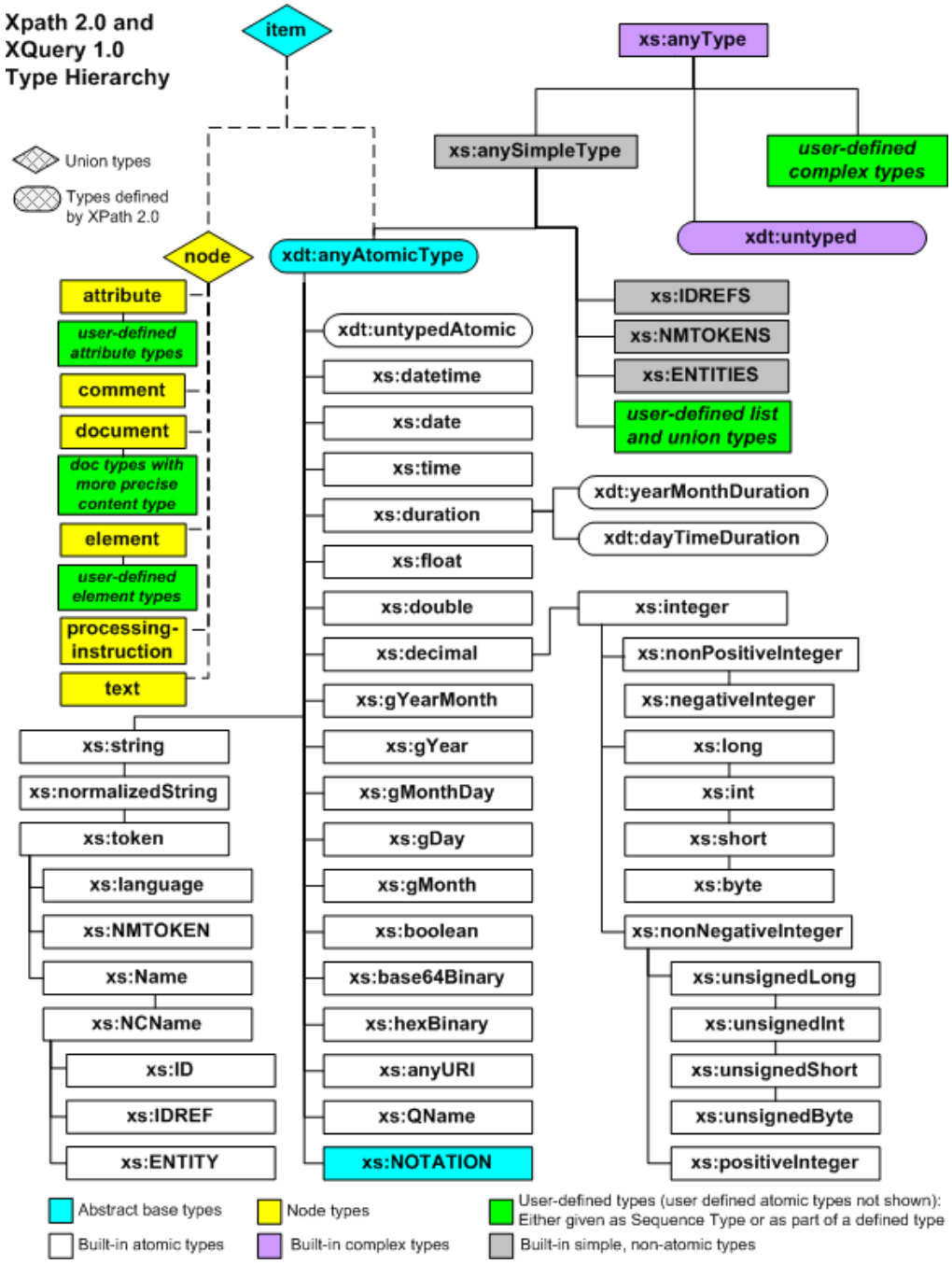
Jazyk XPath 2.0

Základní seznam požadavků na specifikaci XPath 2.0 je následující:

- měla by udržovat zpětnou kompatibilitu
- měla by zjednodušit svoje používání
- měla by lépe podporovat manipulaci s řetězci
- podpora ostatních standardů (XSLT 2.0, XQuery 1.0)
- lepší podpora XML schéma (jednoduché a komplexní typy)

Tyto požadavky specifikace XPath 2.0 celkem dobře splňuje. Zpětná kompatibilita není sice stoprocentní, protože se při vývoji muselo vyhovět ostatním požadavkům, které jsou zřejmě důležitější pro současné použití a budoucí vývoj XPath a ostatních standardů, které XPath využívají. Hlavní změna v XPath 2.0 je ve specifikaci datového modelu, který je založen na infosetu (stromová reprezentace XML dokumentu) s nezbytnou podporou XML schéma. Datový model definuje 7 typů uzlů v XML dokumentu (dokument, element, text, atribut, jmenný prostor, instrukce pro zpracování a komentáře). Uzly v datovém modelu XPath 2.0 jsou velmi podobné jako XPath 1.0, ale v případě elementů a atributů je rozšířena možnost typové informace v XML schématu pro správnou validaci XML dokumentu. Výsledný „typový“ datový model je označován jako „Post schema validation infoset“ (PSVI).

Typy v jazyce Xpath 2.0 a XQuery



Syntaxe XPath

XML dokument může být reprezentován jako strom uzlů (je to podobné adresářové struktuře na vašem PC).

XPath využívá cesty vyjádření k identifikaci uzlů v XML dokumentu. Cesta se skládá z lomítka oddělujícího seznam *child* elemntů vyjadřujících cestu XML dokumentem. Příkaz vybere elementy shodné se zadanou cestou.

Výběr elementů

Následující XPath vyjádření vybere všechny elementy *cena* ze všech elementů *CD* z elementu *katalog*: „`/katalog/cd/cena`“.

Jestliže cesta začíná lomítkem „*/*“, reprezentuje absolutní cestu k elementu a jestliže cesta začíná dvěma lomítky „*//*“, tak platí pro celý dokument.

Následující XPath výraz vybere všechny elementy *cd* v dokumentu: „`//cd`“.

K vybrání neznámých elementů se používá znaménko „*****“.
Následující XPath výraz vybere všechny *child* (dětské) elementy z elementu *cd* z elementu *katalog*: „`/katalog/cd/*`“.

Následující XPath výraz vybere všechny elementy *cena*, které jsou „grandchild“ elementu *katalog*: „`/katalog/*/cena`“.

Následující XPath výraz vybere všechny elementy *cena*, které mají 2 předky: „`/*/*/cena`“.

Následující XPaht výraz vybere všechny elemnty z dokumentu: „`/*`“.

Použitím hranatých závorek můžeme specifikovat další element.
Následující XPath výraz vybere první „child“ element *CD* z elementu *katalog*: „`/katalog/cd[1]`“.

Následující XPath výraz vybere poslední „child“ element *cd* z elementu *katalog*:
„`/katalog/cd[last()]`“, (funkce first() neexistuje).
Následující XPaht výraz vybere všechny elementy *cd*, které obsahují element cena: „`/katlog/cd[cena]`“.

Následující XPath výraz vybere všechny elementy *cd* z elementu *katalog*, kde má element *cena* hodnotu 500: „`/katalog/cd[cena=500]`“ .

Následující XPath výraz vybere všechny elementy *cd* z elementu *katalog*, kde má element *cena* hodnotu 600: „`/katalog/cd[cena=600]/cena`“.

Výběr atributů

V XPath jsou všechny atributy specifikovány prefixem @.
„`//@stat`“ – vybere všechny atributy s názvem *stat.*
„`//cd[@stat]`“ – všechny elementy *cd*, které mají atribut s názvem *stat.*
„`//titul | //umelec`“ – vybere všechny elementy *titul* a *umelec* z dokumentu.
„`//cd[@*]`“ – vybere všechny elementy *cd* s nějakým atributem.

„`//cd[@stat='UK']`“ – vybere všechny elementy *cd* s atributem *stat* s hodnotou *UK*.

Cesta k umístění

Cesta k umístění může být absolutní nebo relativní. Absolutní cesta k umístění začíná lomítekem (/) a relativní cesta lomítkem nezačíná. Oba případy obsahují jeden nebo více částí umístění, všechny odděleny lomítkem.
Absolutní cesta: `/step/step/...`
Relativní cesta: `step/step/...`

Cesta k umístění je ohodnocena pouze jednou, z leva do prava. Každý krok vyjadřuje jeden uzel z aktuální množiny uzlů. Jestliže je cesta absolutní, tak aktuální množina uzlů obsahuje hlavní uzel root. Jestliže je cesta relativní, obsahuje aktuální množina uzlů uzly, které výraz obsahuje. Části cesty jsou složeny z osy, která specifikuje strom vztahů mezi uzly vybranými aktuálním krokem cesty a aktuálním uzlem.

Výběr několika cest

S použitím operátoru | v XPath výrazu můžeme vybrat několik cest.
Výběr elementů *titul* a *umelec* z elementu *cd* z elementu *katalog:*
„`/katalog/cd/titul | /katalog/cd/umelec`“.
Výběr všech elementů *titul* z elementu *cd* z elementu *katalog* a všechny elementy *umelec* z dokumentu: *„`/katalog/cd/titul | //umelec`“*.

XPath osy

Osa definuje množinu uzlů relativně k aktuálnímu uzlu. Test uzlu se používá k identifikaci uzlu uvnitř osy. Můžeme vykonávat test uzlu podle jména nebo typu.

Jména os

„`ancestor`“(předek) – obsahuje všechny předky (*parent, grandparent* atd.) aktuálního uzlu a vždy zahrne *root* uzel, jestliže aktuální uzel není *root.*

„`ancestor-or-self` “ – obsahuje aktuální uzel + všechny předky.

„`attribute`“ – obsahuje všechny atributy aktuálního uzlu.

„`child`“ – obsahuje všechny děti aktuálního uzlu.

„`descendant`“ – obsahuje všechny potomky aktuálního uzlu neobsahuje atributy a jmenné prostory uzlů.

„`descendant-or-self` “– aktuální uzel + všichni potomci.

„`following`“ – obsahuje všechno v dokumentu po uzavíracím tagu aktuálního dokumentu.

„`following-sibling`“ – obsahuje všechny sourozence po aktuálním uzlu jestliže je aktuální uzel atribut uzel nebo *namespace* uzel, bude tato osa prázdná.

„`namespace`“ – obsahuje všechny *namespace* uzly aktuálního uzlu.

„`parent`“ – obsahuje rodiče aktuálního uzlu.

„`preceding`“ – obsahuje vše v dokumentu, co senachází před otevíracím tagem aktuálního uzlu.

„`preceding-sibling`“ – všechny sourozence před aktuálním uzlem. Jestliže je aktuální uzel atribut nebo jmenný prostor, tak je výsledná množina prázdná.

„`self` “ – obsahuje aktuální uzel.

„`child::cd`“ – vybere všechny elementy *cd*, které jsou dětmi aktuálního uzlu. Jestliže nemá žádné děti, je výsledná množina prázdná.

„`attribute::src`“ – vybere *src* atribut aktuálního uzlu. Pokud uzel nemá tento atribut, vrací prázdnou množinu.

„`child::*`“ – vybere všechny „děti“ elementy aktuálního uzlu.

„`attribute::*`“ – vybere všechny atributy aktuálního uzlu.

„child::text()“ – vybere text dětského uzlu aktuálního uzlu.

„child::node()“ – vybere všechny „děti“ aktuálního uzlu.

„descendant::cd“ – vybere všechny elementy *cd*, které jsou potomky aktuálního uzlu.

„ancestor-or-self::cd“ – vybere všechny elementy *cd*, které jsou potomky aktuálního uzlu a jestliže je aktuální uzel element *cd*, tak také ten.

„child::*/child::cena“ – všechny ceny „vnoučat“ aktuálního uzlu.

„/“ – vybere kořen dokumentu.

Predikáty

Predikát filtruje množinu uzlů na novou množinu uzlů. Predikát je uvnitř hranatých závorek.

„child::cena[cena=400]“ – vybere všechny price elementy, které jsou dětmi aktuálního uzlu, kde se cenový prvek rovná 400.

„child::cd[position()=1]“ – vybere první „child“ element *cd* aktuálního uzlu.

„child::cd[position()=last()]“ – vybere poslední „child“ element *cd* aktuálního uzlu.

„/descendant::cd[position()=7]“ –vybere sedmý element *cd*, který je potomkem aktuálního uzlu.

„child::cd[attribute::type="rock"]“ – vybere všechny „child“ elementy cd s atributem classic aktuálního uzlu.

„child::cd[position()=last()-1]“ – vybere předposlední „child“ element cd aktuálního uzlu.

„child::cd[position()<6]“ – vybere prvních 5 „child“ elementů *cd* aktuálního uzlu.

Zkrácený zápis

Zkratka „“ (nic neuvedeme) znamená „child::“.

Např.: „cd“ je zkratka pro „child::cd“.

Zkratka „@“ znamená „attribute::“.
Např. „cd[@type="rock"]“ je zkratka pro
„child::cd[attribute::type="rock"]“

Zkratka „.“ znamená „self::node()“.
Např.: „.//cd“ je zkratka pro
„self::node()/descendant-or self::node()/child::cd“.

Zkratka „..“ znamená „parent::node()“.
Např.: „../cd“ je zkratka pro „parent::node()/child::cd“.

Zkratka „//“ znamená „/descendant-or-self::node()/“ .
Např.: „//cd“ je zkratka pro
„/descendant-or-self::node()/child::cd“.

Pokud uvedeme název uzlu, získáme všechny elementy s tímto názvem, které jsou dětmi aktuálního uzlu.
Např.: „kniha“ – vybere všechny dětské uzly *kniha* aktuálního uzlu

„*“ – vybere všechny dětské uzly aktuálního uzlu.
„text()“ – vybere všechny textové dětské uzly aktuálního uzlu.
„@src“ – vybere všechny atributy *src* aktuálního uzlu.
„@*“ – vybere všechny atributy aktuálního uzlu.
„kniha[1]“ – vybere první dětský uzel *kniha* aktuálního uzlu.
„kniha[last()]“ – vybere poslední dětský uzel *kniha* aktuálního uzlu.
„*/kniha“ – vybere všechny „vnukovské“ (*grandchildren*) uzly *kniha*

aktuálního uzlu.

„`/kniha/kapitola[3]/odstavec[1]“ – vybere první odstavec ze třetí kapitoly z uzlu kniha.

„`//kniha“ – vybere všechny uzly *kniha,* které jsou potomky kořenového uzlu a potom všechny uzly *kniha,* které jsou potomky aktuálního uzlu.

„`.“ – vybere aktuální uzel.

„`.//kniha“ – vybere všechny uzly *kniha*, které jsou potomky aktuálního uzlu.

„`..“ – vybere rodičovský element aktuálního uzlu.

„`../@src“ – vybere atributy *src* rodičovského uzlu.

„`kniha[@type='klasika']“ – vybere všechny dětské uzly *kniha* aktuálního uzlu, které mají atribut s hodnotu klasika.

„`kniha[@type='klasika'][5]“ – vybere pátý dětský uzel *kniha* aktuálního uzlu, který má atribut s hodnotou klasika.

„`kniha[5][@type='klasika']“ vybere pátý dětský uzel *kniha* aktuálního uzlu, který má atribut s hodnotou klasika.

„`kniha[@type and @isbn]“ – vybere všechny dětské uzly aktuálního uzlu, které mají oba uvedené parametry.

Výrazy

Výrazy v jazyce XPath 2.0 jsou stejné jako v jazyce XQuery, a proto už v části o jazyku XQuery nebudou příliš rozebírány.

Matematické výrazy

Matematické výrazy s příkladem použití jsou uvedeny v tabulce č. 1.

Tabulka 1. – matematické výrazy

Operátor	Význam	Příklad	Výsledek
+	sčítání	6 + 4	10
–	odečítání	6 – 4	2
*	násobení	6 * 4	24
div	dělení	8 div 4	2
mod	zbytek po dělení	5 mod 2	1

Porovnávací výrazy

Porovnávací výrazy s příkladem použití jsou uvedeny v tabulce č. 2.

Tabulka 2. – porovnovácí výrazy

Operátor	Význam	Příklad	Výsledek
=	rovno	cena = 10	true (jestliže je cena rovna 10)
!=	nerovno	cena != 10	true (jestliže je cena není rovna 10)
>	větší	cena > 10	true (jestliže je cena větší než 10)
>=	větší nebo rovno	cena >= 10	true (jestliže je cena větší nebo rovna 10)
<	menší	cena < 10	true (jestliže je cena menší než 10)
<=	menší nebo rovno	cena <=10	true (jestliže je cena menší nebo rovna 10)

Funkce

Funkce v jazyce XPath 2.0 jsou stejné jako v jazyce XQuery.

Číselné funkce

Číselné funkce s příkladem použití jsou uvedeny v tabulce č. 3.

Tabulka 3. – číselné funkce

Funkce	Příklad	Výsledek
abs()	abs(55.5)	55.5
	abs(–55.5)	55.5
ceiling()	ceiling (55.5)	56
	ceiling (–55.5)	–55
floor()	floor (55.5)	55
	floor (–55.5)	–56
round()	round(55.5)	56
	round(–55.5)	55

Funkce na řetězcích

Funkce na řetězcích příkladem použití jsou uvedeny v tabulce č. 4.

***Tabulka 4.* – funkce na řetězcích**

Funkce	Příklad	Výsledek
concat()	concat(‘ne’,’bezbečný’)	nebezpečný
string–join()	string–join(‘Ahoj,’, ‘jak’, ‘to’, ‘…’)	Ahoj, jak to …
substring()	substring(‘123456’,2,3)	234
string–length()	string–length(‘12345‘)	5
normalize–space()	normalize–space(‘Ahoj, jak to jde?)	Ahoj, jak to jde?

upper–case()	upper–case(‘aBcdE’)	ABCDE
lower–case()	lower–case(‘aBcdE’)	abcde
translate()	translate(‘Ahoj‘, ‘A’,’a’)	ahoj

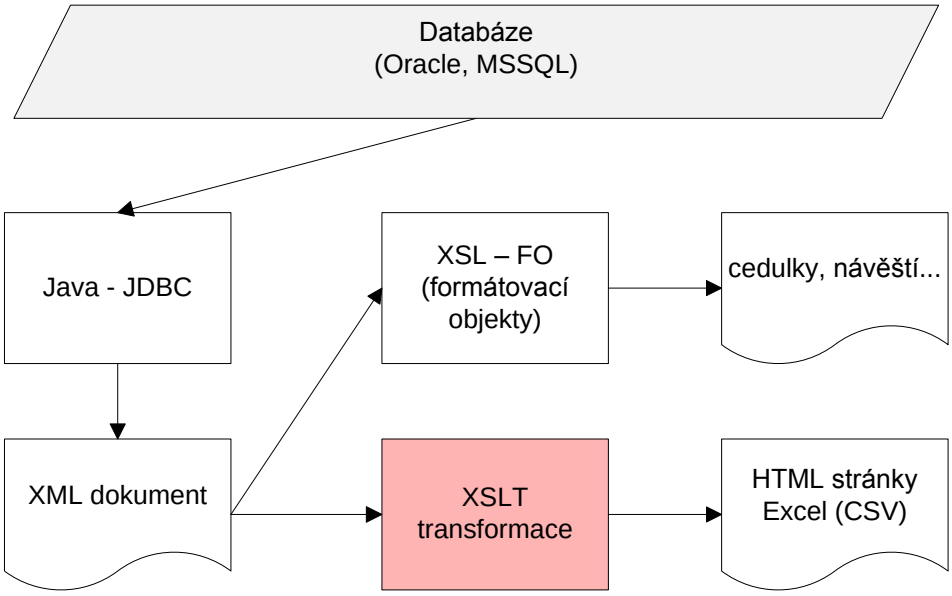
Pro jazyk XPath 2.0 a jazyk XQuery existuje ještě celá řada funkcí, které zde nebudu všechny popisovat. Funkce, které jsem uvedl, slouží pro lepší představu o tom, co vše jazyky XPath 2.0 a XQuery umožňují. Nezmínil jsem zde ani funkce pro práci s datumem a časem, kterých je také celá řada a pokud se s nimi chcete seznámit, podívejte se na http://www.w3.org/TR/xpath–functions/.

XSLT V PRAXI

Vícevrstvé aplikace

S použitím XSLT jsem se setkal několikrát před lety ve firemním prostředí a byť se jednalo o krátká setkání, získal jsem alespoň malou představu o jeho použití ve větším měřítku, konkrétně pak o využití jazyka XSLT ve vícevrstvých aplikacích.

Prvním typem byla třívrstvá aplikace založená na použití databázových systémů (Oracle nebo MSSQL), Java aplikace s JDBC (Java DataBase Connector) a XSL transformací. Data uložená v databázi se pomocí specializovaného aplikačního rozhraní JDBC, které je součástí programovacího jazyka Java, načítala a byla převáděna do XML. Pak přicházela na řadu XSL transformace, která zajišťovala výstup takto získaných dat do HTML stránek, exporty do excelu (CSV) apod. Přes XSL-FO se pak vytvářely reporty v PDF, nebo třeba jmenovky zaměstnancům na přepážky, informační cedulky s obsazením kanceláří lidmi, k patrovým návěstím s přehledem v kterém křídle jsou jaké kanceláře… atd. Využití transformace zde bylo široké a velmi praktické.



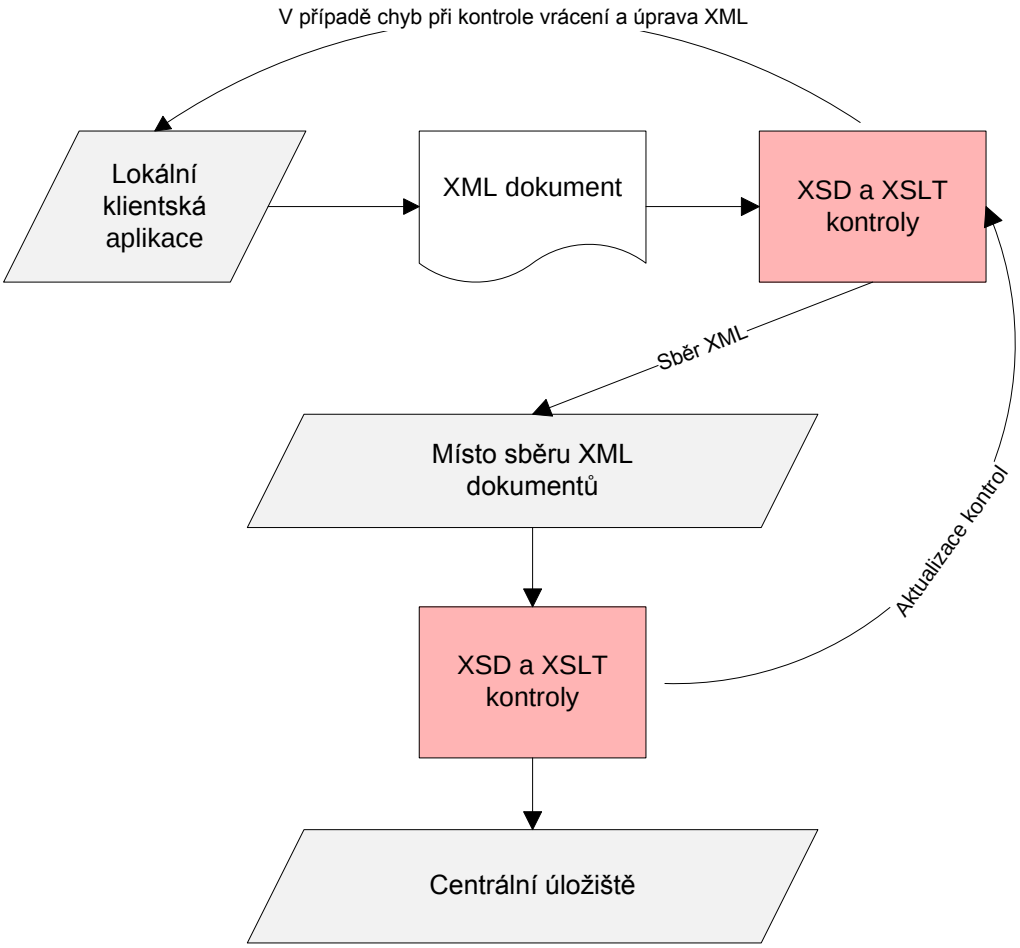
Druhým typem byla třívrstvá aplikace fungující jako informační systém, který shromažďoval data z různorodých pobočkových míst, což byly opravdu odlišné programy od různých dodavatelů, tedy cca desítky (konkrétně více než 50) nesourodých systémů, jejichž data bylo třeba posbírat na jedno centrální místo pro globální vyhodnocení. Toto může být třeba případ mezinárodní firmy, například cestovní kanceláře, která skupuje v jednotlivých zemích menší cestovní kanceláře a chtěla by od nich sbírat data o tom, kolik čeho prodaly, seznamy klientů, personální informace o zaměstnancích, příjmy, výdaje, zisky, atd. Těmto pobočkám nechá firma jejich používaný software, protože by bylo zbytečné pořizovat všem nový, lokalizovat jej, patřičně zaškolovat zaměstnance, převádět data, atd. Z tohoto důvodu firma pouze definuje XML, které musí umět každý z těchto programů vygenerovat. Následně tato XML posbírá, zkontroluje jejich správnost a importuje jednoduše do centrálního úložiště, kde se budou data různě vyhodnocovat a zpracovávat.

V tomto konkrétním případě se jednalo o klientský Java program, který uměl načíst XML, zkontrolovat jeho strukturu přes XSD, provést logické a

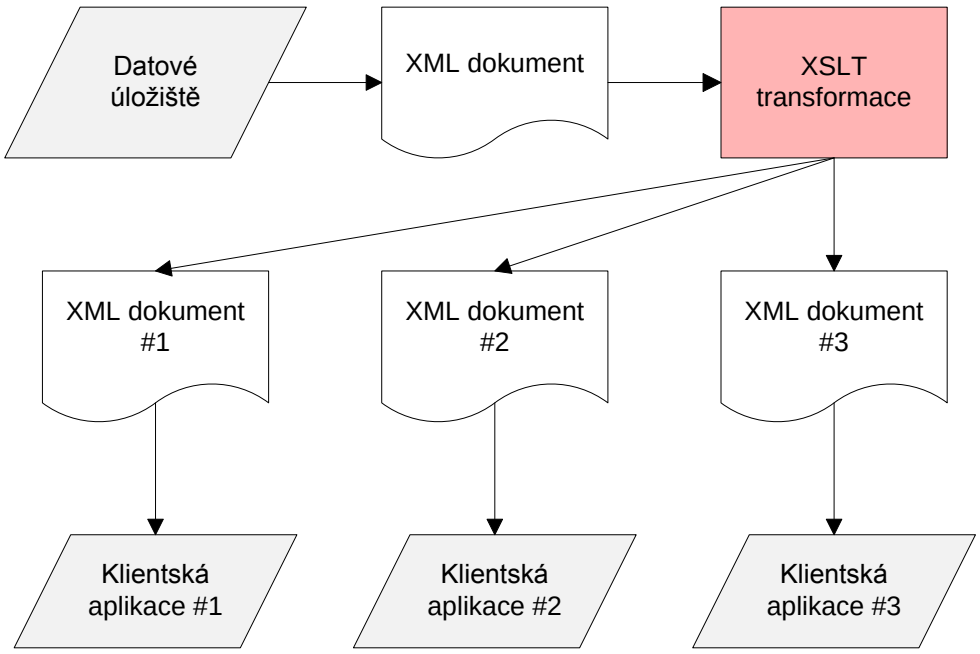
číselníkové kontroly přes XSLT a vypsat výstup chyb či varování v případě, že data nebyla správná. Obrovská výhoda byla v tom, že se řeklo, jak má vypadat XML, definovaly se jedny kontroly v XSLT a tyto kontroly si mohly udělat jednotlivé pobočky již před odesláním XML na centrálu, čímž si mohly snadno zjistit, zda jejich výstup bude přijat v centrále. Odpadalo tedy zdlouhavé posílání na server, čekání na noční zpracování, vrácení seznamu chyb, a X dalších koleček dokud vše nebylo podle toho, jak to potřeboval server.

Server samozřejmě používal stejné XSLT kontroly jako pobočky. Pokud došlo k úpravě XSLT kontrol, standardními distribučními kanály (stejně jako updaty SW) se tyto změny dostaly ke klientským programům a bylo tak zajištěno jejich aktuálně správné provedení.

Níže se ještě podíváme na některé ukázky kódu, které mohou být použity v takovéto aplikaci, abychom si tak mohli udělat dokonalejší představu o jejím fungování.



Třetí typ třívrstvé aplikace se používal opačně než posledně zmiňovaný. Z jednoho místa se XML soubory distribuovaly do dalších různých aplikací. XSLT v tomto případě plnil roli překladatele určitých XML souborů na jiné XML, podle směrování na aplikace. Struktura XML souborů se tak upravovala přesně podle potřeby té které aplikace. Ta poté mohla přijatý XML soubor bezproblémově zpracovat, protože již byl ve tvaru, jaký pro ni byl potřebný. Takováto centrální úprava pomocí XSLT, před samotnou distribucí, snižovala zatížení klientských aplikací, které by musely přijatý soubor samy transformovat.



Když jsem zmiňoval druhý typ třívrstvé aplikace, používající XSTL transformaci pro logické a číselníkové kontroly, sliboval jsem ukázky kódů, které tato aplikace může například využívat. Z některých kontrol prováděných na lokálních aplikacích a poté i na serveru, se jako první podíváme na jednoduchou kontrolu vztahu přesčasových a skutečně odpracovaných hodin zaměstnance. Jde o situaci, kdy potřebujeme ověřit jednoduchou logickou podmínku, že pokud zaměstnanec odpracoval nějaké skutečné hodiny a zároveň také přesčasové hodiny, nemůže být hodnota přesčasových hodin větší než hodnota skutečně odpracovaných hodin. K takovéto chybě může dojít překlepem nebo chybným výpočtem.

```
<!-- skutecne odpracovane hodiny -->
<xsl:variable name="oh" select="normalize-
space(odpr_neodpr_doba/AA0132)"/>
<!-- prescasove hodiny -->
<xsl:variable name="ph" select="normalize-
space(odpr_neodpr_doba/AA0133)"/>
<!-- podmínka pro porovnání odpacovaných a přesčasových hodin -
->
<xsl:if test="string-length($oh) &gt; 0 and string-length($ph)
&gt; 0 and $oh &lt; $ph">
<!-- pokud nastane podmínka, získáme hlášení o chybě -->
  <xsl:element name="chyba">
    <xsl:attribute name="typ"><xsl:value-of
select="$error"/></xsl:attribute>
    <xsl:attribute name="radka"><xsl:value-of select="-
1"/></xsl:attribute>
    <xsl:attribute name="oblast"><xsl:value-of
select="$oblast"/></xsl:attribute>
    <xsl:text>A03 - Zamestnanec musi mit odpracovano mene
prescasovych hodin (AA0133), nez skutecne odpracovanych
(AA0132) !</xsl:text>
  </xsl:element><xsl:text>&#xA;</xsl:text>
</xsl:if>
```

Ukázkový příklad logické kontroly odpracovaných hodin je jednoduchým zjištěním, zda nenastala chyba v zadaných datech. Další takovou důležitou kontrolou u položek zaměstnanců, která se běžně provádí, je kontrola správnosti rodného čísla. Tato kontrola je opět prováděna na daných datech a ověřuje, zda nedošlo k překlepu při zadávání obsluhou nebo zda takové rodné číslo vůbec existuje. K této kontrole je použito modulo 11, i když u starších rodných čísel toto pravidlo platit nemusí. Výstupem nám je seznam chyb, kde X je označení pro chybu a W pro varování. Pro nejistotu v použití modulo 11 u starších rodných čísel je zobrazováno pouze varování W v případě jeho nesplnění. Pojďme se tedy podívat na samotný kód.

```
<xsl:choose>
        <!-- Test zda prvni dvojcisli (rok) je 00-99 -->
        <xsl:when test="not(not(normalize-space(rodne_cislo))
or
                        (number(substring(rodne_cislo,1,2))
&gt; -1  and number(substring(rodne_cislo,1,2)) &lt; 100))">
          <xsl:element name="chyba">Chyba X - První dvojčíslí
rodného čísla (rok) musí být v intervalu 00-99 !</xsl:element>
        </xsl:when>
        <!-- Test zda druhe dvojcisli (mesic) je 1-12 nebo 51-
62 -->
        <xsl:when test="not(not(normalize-space(rodne_cislo))
or
                        (number(substring(rodne_cislo,3,2))
&gt; 0  and number(substring(rodne_cislo,3,2)) &lt; 13) or
                        (number(substring(rodne_cislo,3,2))
&gt; 50 and number(substring(rodne_cislo,3,2)) &lt; 63))">
          <xsl:element name="chyba">Chyba X - Druhé dvojčíslí
rodného čísla (měsíc) musí být v intervalu 1-12 nebo 51-62
!</xsl:element>
        </xsl:when>
        <!-- Test na treti dvojcisli (den) - zda odpovida pocet
dnu v mesici -->
        <xsl:when test="not(not(normalize-
space(.//rodne_cislo)) or
                        (
(number(substring(.//rodne_cislo,3,2)) = 1  or
number(substring(.//rodne_cislo,3,2)) = 3  or

number(substring(.//rodne_cislo,3,2)) = 5  or
number(substring(.//rodne_cislo,3,2)) = 7  or

number(substring(.//rodne_cislo,3,2)) = 8  or
number(substring(.//rodne_cislo,3,2)) = 10 or

number(substring(.//rodne_cislo,3,2)) = 12 or
number(substring(.//rodne_cislo,3,2)) = 51 or

number(substring(.//rodne_cislo,3,2)) = 53 or
number(substring(.//rodne_cislo,3,2)) = 55 or

number(substring(.//rodne_cislo,3,2)) = 57 or
number(substring(.//rodne_cislo,3,2)) = 58 or

number(substring(.//rodne_cislo,3,2)) = 60 or
number(substring(.//rodne_cislo,3,2)) = 62 ) and

(number(substring(.//rodne_cislo,5,2)) &gt; 0 and
number(substring(.//rodne_cislo,5,2)) &lt; 32)
                        ) or
                        (
(number(substring(.//rodne_cislo,3,2)) = 2 or
number(substring(.//rodne_cislo,3,2)) = 52) and
```

```
(number(substring(.//rodne_cislo,5,2)) &gt; 0 and
number(substring(.//rodne_cislo,5,2)) &lt; 30)
                        ) or
                        (
(number(substring(.//rodne_cislo,3,2)) = 4  or
number(substring(.//rodne_cislo,3,2)) = 6  or

number(substring(.//rodne_cislo,3,2)) = 9  or
number(substring(.//rodne_cislo,3,2)) = 11 or

number(substring(.//rodne_cislo,3,2)) = 54 or
number(substring(.//rodne_cislo,3,2)) = 56 or

number(substring(.//rodne_cislo,3,2)) = 59 or
number(substring(.//rodne_cislo,3,2)) = 61) and

(number(substring(.//rodne_cislo,5,2)) &gt; 0 and
number(substring(.//rodne_cislo,5,2)) &lt; 31)
                        ))">
          <xsl:element name="chyba">Chyba X - v třetím
dvojčíslí v rodném čísle. Měsíc <xsl:value-of
select="substring(rodne_cislo,3,2)"/> nemá <xsl:value-of
select="substring(rodne_cislo,5,2)"/> dní !</xsl:element>
        </xsl:when>
        <!-- Test na posledni ctyrcisli -->
        <xsl:when test="not(not(normalize-space(rodne_cislo))
or
                        string-length(rodne_cislo) = 9 or
                        ( (string-length(rodne_cislo) = 10 and
(number(substring(rodne_cislo,1,9)) mod 11) = 10 and
                          number(substring(rodne_cislo,10,1)) =
0) or
                          (string-length(rodne_cislo) = 10 and
(number(substring(rodne_cislo,1,9)) mod 11) != 10 and
                          number(substring(rodne_cislo,10,1)) =
(number(substring(rodne_cislo,1,9)) mod 11))
                        ))">
          <xsl:element name="chyba">Chyba W - Chybné poslední
čtyřčíslí rodného čísla (nesplněn algoritmus modulo 11)
!</xsl:element>
        </xsl:when>

      </xsl:choose>
```

Jak je vidět, zde už se jedná o trochu složitější logické porovnávání. Je třeba brát v potaz správnost každé části rodného čísla (rok, měsíc a den) včetně posledního čtyřčíslí. Porovnání tím máme hotové, ale my samozřejmě chceme mít i nějaký výstup. Pro jeho získání můžeme vyrobit šablonu, která nám

vypíše chyby, jejich součty, atd. do textového (samozřejmě je možné vytvořit třeba i graficky stylovaný výstup do HTML) souboru.

```
<!-- Globální proměnné -->
<xsl:variable name="warn"  select="'W'"/>
<xsl:variable name="error" select="'X'"/>
<xsl:variable name="fatal" select="'E'"/>

<xsl:template match="/">
  <xsl:apply-templates select="kontroly"/>
</xsl:template>

<xsl:template match="kontroly">
  <xsl:apply-templates select="kontrola"/>
  <xsl:call-template name="totalSum"/>
</xsl:template>

<xsl:template match="kontrola">
  <xsl:variable name="d" select="document(.)"/>
  <xsl:value-of
select="$d/kontrola/header"/><xsl:text>&#xA;</xsl:text>
  <xsl:text>-------------------------------------------------
----&#xA;</xsl:text>
  <xsl:choose>
<!-- Pokud existují chyby, vypíšeme o každé z nich informace --
>
    <xsl:when test="count($d/kontrola/chyby/chyba) > 0">
      <xsl:for-each select="$d/kontrola/chyby/chyba">
        <xsl:variable name="pos" select="position()"/>
        <xsl:variable name="oblast" select="@oblast"/>
        <xsl:variable name="last_oblast"
select="ancestor::chyby/chyba[position()=($pos - 1)]/@oblast"/>

        <xsl:if test="($pos = 1 and string-length($oblast) > 0)
or ($oblast != $last_oblast)">
          <xsl:value-of
select="@oblast"/><xsl:text>&#xA;</xsl:text>
        </xsl:if>
        <xsl:text>Chyba </xsl:text><xsl:value-of
select="@typ"/>
        <xsl:if test="number(normalize-space(@radka)) > 0">
          <xsl:text> - radka </xsl:text><xsl:value-of
select="@radka"/>
        </xsl:if>
        <xsl:text> : </xsl:text><xsl:value-of select="."/>
        <xsl:text>&#xA;</xsl:text>
      </xsl:for-each>
      <xsl:text>&#xA;</xsl:text>
      <xsl:text>Pocet varovani : </xsl:text><xsl:value-of
```

```
select="count($d/kontrola/chyby/chyba[@typ=$warn])"/><xsl:text>
&#xA;</xsl:text>
      <xsl:text>Pocet zavaznych chyb : </xsl:text><xsl:value-of
select="count($d/kontrola/chyby/chyba[@typ=$error])"/><xsl:text
>&#xA;</xsl:text>
      <xsl:text>Pocet nezpracovatelnych dat :
</xsl:text><xsl:value-of
select="count($d/kontrola/chyby/chyba[@typ=$fatal])"/><xsl:text
>&#xA;</xsl:text>
      <xsl:text>Celkem chyb : </xsl:text><xsl:value-of
select="count($d/kontrola/chyby/chyba)"/><xsl:text>&#xA;</xsl:t
ext>
    </xsl:when>
<!-- Pokud chyby nejsou -->
    <xsl:otherwise>
      <xsl:text>Bez chyb.</xsl:text><xsl:text>&#xA;</xsl:text>
    </xsl:otherwise>
  </xsl:choose>
  <xsl:text>&#xA;</xsl:text>
</xsl:template>
<!-- Celkové součty -->
<xsl:template name="totalSum">
  <xsl:text>&#xA;</xsl:text>
  <xsl:text>Celkovy seznam
chyb:</xsl:text><xsl:text>&#xA;</xsl:text>
  <xsl:text>--------------------&#xA;</xsl:text>
  <xsl:text>Celkem varovani : </xsl:text><xsl:value-of
select="/kontroly/celkem[@typ=$warn]"/><xsl:text>&#xA;</xsl:tex
t>
  <xsl:text>Celkem zavaznych chyb : </xsl:text><xsl:value-of
select="/kontroly/celkem[@typ=$error]"/><xsl:text>&#xA;</xsl:te
xt>
  <xsl:text>Celkem nezpracovatelnych dat :
</xsl:text><xsl:value-of
select="/kontroly/celkem[@typ=$fatal]"/><xsl:text>&#xA;</xsl:te
xt>
  <xsl:text>Celkem vsech chyb : </xsl:text><xsl:value-of
select="/kontroly/total"/><xsl:text>&#xA;</xsl:text>
  <xsl:text>&#xA;&#xA;</xsl:text>
</xsl:template>
```

EXSLT – moduly pro rozšíření jazyka XSLT

Další z prakticky využívaných oblastí XSLT je open-source projekt EXSLT. Jedná se vlastně o poskytování rozšíření jazyka XSLT. Pro šablony v jazyce XSLT 1.0 tak rázem získáme možnost využívat například funkce data a času, regulární výrazy… atd., tedy funkce, které standardně nejsou dostupné v této starší verzi jazyka XSLT. Tato rozšíření jsou členěna do několika modulů:

Common
Math
Sets
Functions
Date and Times
Strings
Regular Expressions
Dynamic
Random

Výhodou při používání těchto rozšíření tkví například v přenosnosti stylů. Můžeme si stáhnout balíčky rozšíření, které poté použijete k poskytnutí funkcí a šablon pro náš stylový soubor. Abychom nějaké z těchto rozšíření mohli použít, je potřeba definovat jmenný prostor (xslns) pro modul EXSLT. Odkaz na jmenný prostor je standardně v následující podobě:

`http://exslt.org/jméno-modulu`

tzn. například pro použití modulu regulárních výrazů (Regular Expressions) odkaz vypadá takto:

`http://exslt.org/regular-expressions`

Pokud námi používaný procesor tento modul podporuje, není již třeba definovat nic dalšího. Pokud ale podpora od procesoru neexistuje, je nezbytné definovat odkaz na šablonu námi staženého modulu. V našem stylu pak bude definice (v tomto případě pro modul EXSLT - Regular Expressions) vypadat následujícím způsobem:

```
<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:regexp="http://exslt.org/regular-
expressions"
                extension-element-prefixes="regexp">

<xsl:import href="regexp.xsl" />

...

</xsl:stylesheet>
```

Další podrobnější informace je možné se dozvědět přímo z webových stránek projektu na http://exslt.org.

Jak je na první pohled zřejmé, pro programátory zvyklé na starší verzi XSLT může být toto rozšíření velmi praktické a zajisté pomůže i k plynulejšímu přechodu mezi verzemi jazyka XSLT tím, že budou moci postupně používat funkce, které do té doby nepoužívali, ale které již jsou nedílnou součástí verze 2.0.

Použití XSLT v .NET frameworku

Stěžejní třídou pro provádění XSL transformací je v základní knihovně tříd .NET frameworku třída ***XslTransform***, kterou nalezneme ve jmenném prostoru ***System.Xml.Xsl***. Zdroje pro transformaci jsou představovány instancemi tříd ***XMLDocument***, ***XMLDataDocument*** nebo ***XPathDocument***, které všechny implementují rozhraní IXPathNavigable. Jako nejvhodnější pro klasické transformace se jeví použití instancí třídy XPathDocument, které by mělo zajišťovat nejvyšší výkon.

Samotné provedení transformace není v .NET vůbec složité. Po vytvoření instance třídy ***XPathDocument*** na základě XML dokumentu stačí vytvořit instanci třídy ***XslTransform*** pomocí její instanční metody ***Load*** nahrát požadovanou šablonu. A k provedení samostatné transformace použít jednu z mnoha přetížených verzí metody ***Transform***, které poté předat pouze zdroj ve formě instance třídy ***XPathDocument*** a výstupní proud zaobalený do instance třídy ***StreamWriter***.

XSLT ve Windows SharePoint Services

Data uložená v systému Windows SharePoint Services 3.0 mohou být prezentována více způsoby, nejen jednoduchým tabulkovým layoutem, na který jsme ze Sharepointu běžně zvyklí. Použijeme-li aplikaci Microsoft Office SharePoint Designer 2007 a vlastní XSLT můžeme zobrazovat stejná data také v grafické podobě - například jako různé grafy. Web party tak získají zcela nový vzhled, závislý na naší dovednosti úpravy XSL souboru určeného pro tuto transformaci. Schopnost vytvoření editovatelného XSL na několik kliknutí tak dělá z aplikace Sharepoint Designer 2007 výborného pomocníka při kustomizaci intranetových webových stránek. SharePoint Designer 2007 je navíc od konce března roku 2009 volně ke stažení ze stránek Microsoftu, takže není problém vyzkoušet ho ve svém prostředí.

Ze zkušeností ze zaměstnání také vím, že Sharepoint je velmi se rozšiřujícím řešením pro tvorbu a správu podnikového intranetu. Možnost využívat v něm XSLT je určitě skvělá pro vývojáře, kteří chtějí jeho vzhled přizpůsobit firemním požadavkům.

XSLT Data View Web Parts byly a stále zůstávají jednou z nejmocnějších webových částí SharePointu, až se jim dokonce někdy přezdívá „švýcarský armádní nůž SharePoint Web Parts“. S nadcházejícím vydáním Windows SharePoint Services 4.0 a zároveň SharePoint 2010 a SharePoint Designer 2010 bychom se měli dočkat několika výrazných změn. Jednou z nich, pokud bude zavedena, by mohla být také podpora XSLT ve verzi 2.0. Asi největší změnou vzhledem k používání technologie XSLT v SharePointu bude přidání nového typu web part - XSLT List View Web Part, které nahradí stávající defaultní zobrazení pomocí List View Web Part.

Problémem s Data View Web Parts je v současné době to, že jakmile je jednou publikujeme na stránku, nemohou s nimi už koncoví uživatelé

manipulovat. V takovém případě mají dvě možnosti, otevřít je a znovu zpracovat v SharePoint Designeru nebo editovat XSLT přímo ve webovém prohlížeči, což ale není úplně nejjednodušší věc. List View Web Part je v současnosti používáno pro zobrazování různých typů web partů na stránce, ale pro lepší editaci se často převádí na XSLT Data View Web Parts a teprve poté v Designeru zpracovává a rozšiřuje. V novém Sharepoint Designer 2010 bude XSLT List View Web Part kombinovat výhody obou zmiňovaných:

XSLT List View Web Part = List View Web Part + XSLT Data View Web Part

```
 </asp:UpdatePanel>
<WebPartPages:XsltListViewWebPart ChromeType="TitleAndBorder"
<XmlDefinition>
    <View Name="{8BE833A3-3388-4298-A3B6-B02A05B6CB60}" Mobile
```

Upravovat XSLT List View Web Part budeme moci přímo v novém SharePoint Designeru 2010, jakmile je poté publikujeme na stránku, budou s nimi moci koncoví uživatelé pracovat se stejnými možnostmi, jako tomu bylo u List View Web Part. Stále ale v SharePoint Designeru 2010 najdeme onen „švýcarský armádní nůž“ - XSLT Data View Web Parts. Navíc bude prostředí SPD designeru doplněno o příjemnou Ribbon lištu, jako ji známe již z Office 2007. Určitě samé příznivé zprávy pro vývojáře pracující s XSLT technologií v prostředí Windows SharePoint Services.

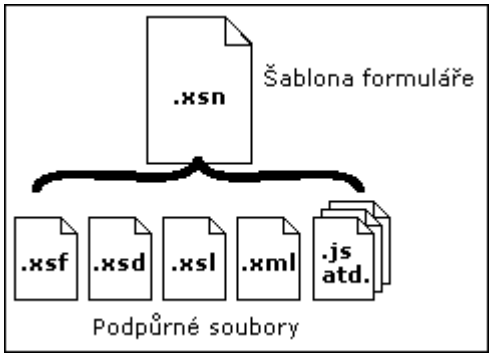
Využití XSLT v MS Office a OpenOffice

Formát Office Open XML (OOXML) od společnosti Microsoft a formát OpenDocument (ODF) od sdružení OASIS jsou dvě, v jádru podobné, specifikace. Obě mají za účel zkvalitnit práci s dokumenty pomocí použití více možností pro editování a formátování, které nám jazyk XML nabízí. Další podstatnou věcí je umožnění práce s těmito formáty v kancelářských aplikacích vyvinutých různými společnostmi, což je také prakticky největší výhoda při používání multiplatformních formátů, jako je kupříkladu XML. Zároveň nám tím tyto typy souborů dávají možnost využívat XSLT transformaci, což je to, co nás právě teď zajímá.

MS OFFICE

Nejnovější formát souborů, který se v MS Office 2007 začal využívat a jenž poznáme podle písmene x na konci přípony (docx – dokument pro textové procesory, xlsx – dokument pro tabulkové kalkulátory, pptx – dokument s prezentací), byl vyvinut jako specifikace pro ukládání dokumentů z kancelářských balíků. Tyto dokumenty jsou vlastně komprimované ZIP soubory, které v sobě ukládají obsah dokumentu v podobě XML a dalších souborů. Díky komprimaci mají tyto soubory menší velikost, dále nabízí více kompatibility a transparentnosti.

Jednou z aplikací z balíčku MS Office, která hojně využívá XSLT transformace, je InfoPath 2007. Tento nástroj slouží ke správě a shromažďování široké škály elektronických formulářů. Šablony těchto formulářů jsou též složeny z několika různých podpůrných souborů. Jde o soubory s grafikou, soubory s programovým kódem, nebo třeba soubory s definicí vzhledu ovládacích prvků formuláře… viz.: obrázek a tabulka s popisem jednotlivých formátů v šabloně.



Typ souboru	Přípona	Popis
Soubor definice formuláře	XSF	Soubor obsahující informace o tom, jak je šablona formuláře sestavena, včetně používaných schémat XML a obsažených souborů prostředků. Aplikace InfoPath automaticky vygeneruje tento soubor při návrhu nové šablony formuláře. Při změnách šablony formuláře během návrhu je tento soubor automaticky aktualizován, aby odrážel provedené změny.
Schéma XML	XSD	Soubor nebo soubory sloužící k omezení a ověření dat v šabloně formuláře. Obsah souborů schématu XML – prvky, atributy a tak dále – je v podokně úloh **Zdroj dat**představován skupinami a poli. Každému zdroji datpřidruženému k šabloně formuláře (včetně hlavního zdroje dat) odpovídá určitý soubor XSD.
Zobrazení	XSL	Soubory transformace XSL (XSLT), které slouží k prezentaci, zobrazení a převodu dat obsažených ve formuláři vyplňovaném uživateli do formátu HTML. Pracujete-li s různými zobrazeními šablony formuláře, díváte se ve skutečnosti na různé reprezentace (transformace) dat ve formuláři do formátu HTML.
Šablona XML	XML	Soubor obsahující data, která se mají zobrazit jako výchozí v určitých ovládacích prvcích. Uživatelé vidí tato data při otevření formuláře a pak mohou v ovládacích prvcích vybrat jiné hodnoty.

Pokud se podíváme na použití XSLT ve Wordu, zjistíme, že i zde se dá transformace velmi dobře využít. Lze například vytvářet dokumenty Word z obyčejného xml dokumentu za pomoci XSLT transformace. Proces převedení xml souboru do formátu docx se provádí nejjednodušším způsobem na základě již vytvořeného vzhledu v existujícím dokumentu aplikace MS Word 2007. Tento vzhled (xml soubor) je možné z docx souboru jednoduše zkopírovat do jiné složky. K tomu nám pomůže přejmenovat tento soubor na soubor s příponou zip (např.: priklad.docx.zip), ten následně v průzkumníku otevřít a ve složce word nalezneme document.xml jehož kód poté použijeme k vytvoření stylového xsl souboru. Pro pokročilého programátora pak není velkým problémem za pomoci Open XML SDK 2.0 (obsahuje knihovny pro práci s Open XML dokumenty) zajistit XSL transformaci do souboru Word docx ze zdrojového XML datového souboru (xml + xslt → docx).

V procesu zautomatizování spouštění aplikace Word nám mohou hodně pomoci přepínače. Jedná se vlastně o parametry, které jsou použity a mění její chování po spuštění příkazu Winword.exe. Přepínač se zadává za tento příkaz, mezeru a lomítko.

V našem případě využijeme přepínač /p*xslt*, který spustí aplikaci Word pomocí definovaného XML souboru a námi zadané XSLT transformace v podobě XSL souboru. Jako příklad by mohlo spuštění s přepínačem /p vypadat takto:

```
winword.exe /p nase_transformace.xsl datovy_soubor.xml
```

Tímto způsobem je možné jednoduše vytvořit například zástupce, který bude automaticky spouštět nový dokument Word, na základě našich vstupních dokumentů.

I aplikace Excel dokáže využívat XSLT transformací. Jazyk XML je výborným nástrojem pro importy prvků do sešitů a šablon aplikace Excel, pro exporty za pomoci mapování klíčových dat, bez ovlivnění ostatních dat v celém sešitu… atd. Mapování XML slouží k navázání obsahu mapovaných buněk k prvkům schématu při importu nebo exportu datových souborů XML. Transformaci XSLT pak využijeme jak při importu dat z XML souboru do xlsx tak i při exportu. Kombinací šablon aplikace Excel, datových XML a transformačních XSL souborů tak získáváme velmi silný nástroj pro správu a tvorbu různých druhů dokumentů, jakými jsou například firemní faktury.

OPEN OFFICE

Pro formát Open Office XML od společnosti Microsoft už ale v době jeho vzniku existoval konkurenční a zcela otevřený formát, OpenDocument Format (ODF) vyvinutý za pomoci mnoha společností a programátorů sdružením OASIS. Vychází ze starších formátů kancelářského balíčku OpenOffice.org, který je sám o sobě založen na XML, což ho předurčuje k používání transformací pomocí eXtensible Stylesheet Language. Slouží pro výměnu dokumentů mezi kancelářskými balíčky a dalšími aplikacemi. Otevřenost je zde úplná, takže není potřeba upínat se k jednomu dodavateli software, a na vývoji a tvorbě aplikací používajících ODF se tak může podílet naprosto kdokoliv. Není proto málo častým jevem, že si jednotlivé společnosti programují své vlastní aplikace, které při práci s ODF využívají. Kompatibilita je díky úplné otevřenosti zaručena, což není u konkurenčního standardu společnosti Microsoft zcela stejné, neboť formát OOXML vychází přeci jen z jejich vlastních uzavřených formátů.

Podobně jako u Microsoft standardu, i zde je použití s XSLT podobné. Šablony jsou využívány při tvorbě ODF souborů z XML datových souborů, k změně vzhledu a stylů, k exportu do XML, HTML, XHTML a dalších.

Kancelářský balík Open Office obsahuje v základním nastavení několik užitečných šablon transformací XSLT, které se nalézají v instalační složce balíku v podložkách Basis\share\xslt. Tyto šablony lze upravovat nebo vytvořit své vlastní a ty pak při práci s dokumenty používat.

Dalším plusem při práci s tímto formátem je velké množství přeprogramovaných filtrů, díky nimž lze provádět činnosti, jako třeba převod z ODF na XHTML automaticky bez složitého vytváření vlastních transformačních souborů.

Co se týče kompatibility mezi těmito konkurenčními formáty, je třeba si dát dobrý pozor v situaci, kdy bychom vytvořený docx soubor chtěli uložit jako odt (formát souboru textového editoru Writer). Všechny funkce pracující s XML v Microsoft Office Word nejsou v OpenOffice.org podporovány (viz tabulka).

Jazyk	Funkce	Podpora
XML	Připojení schématu	Nepodporována
XML	Atributy	Nepodporována
XML	Externí zdroj	Nepodporována
XML	Zahrnutí textu	Nepodporována
XML	Sloučení dokumentů XML	Nepodporována
XML	Objektově orientované uživatelské rozhraní	Nepodporována
XML	Chování při označování	Nepodporována
XML	Stromové zobrazení	Nepodporována
XML	Ověření	Nepodporována
XML	Transformace XSL (XSLT)	Nepodporována

Jak je vidět, aplikace Microsoft Office a OpenOffice.org poskytují celou řadu možností práce s XSLT. Stále se však jedná o konkurenční produkty a vzájemná kompatibilita tak není zcela na místě. Podle mého názoru tak zůstávají karty především v rukou projektu OpenDocument, jelikož právě on je od základů postaven na XML a vzhledem k jeho úplné otevřenosti poskytuje daleko jednodušší a transparentní možnosti použití transformací.

ZÁVĚR

eXtensible Stylesheet Language Transformation je jednou z velmi se rozšiřujících technologií pro zpracování dat uložených v XML a jak jste se na stránkách této práce mohli dočíst, existuje spousta praktických situací, kdy je využití velmi vhodné. Jde jak o firemní prostředí tak i osobní. Samozřejmě by se dalo psát o využití pro RSS, Docbook, PHP a spoustě dalších technologií, kde lze transformaci uplatnit, ale na to již rozsah této práce nestačí. Snažil jsem se tedy zmínit hlavně ty, se kterými jsem alespoň trochu přišel do kontaktu a umožnit tak čtenáři nahlédnout pod pokličku praktickému využití.

Ti, kdo již nějaké zkušenosti s předchozí verzí tohoto jazyka mají, se mohli seznámit s postupy přechodu na verzi 2.0 a novými prvky, které tato verze obsahuje. Pro začínající programátory jsem snad poskytl vodítko a chuť k dalšímu studiu tohoto jazyka.

Nelze nezmínit web zdrojak.cz, kde v Názorech k článku XSLT – Jazyk budoucnosti píše veliký odborník pan Kosek: „XSLT pracuje na odlišném modelu, než konvenční procedurální program. Na školení vždy říkám, že je potřeba si na XSLT zvyknout a přeskupit neurony v mozku tak, aby myslely v duchu XSLT/XPath. ;-) “

Nezbývá než poděkovat, pokud jste mou práci dočetli až sem a popřát ať nám to správně v duchu XSLT myslí.

SEZNAM POUŽITÉ LITERATURY

[1] EL-HATTAB, Ayman In SharePoint Designer 2010 : XSLT List View Web Part = List View Web Part + XSLT Data View Web Part. In . [s.l.] : [s.n.], 24.11.2009 [cit. 2010-03-18]. Dostupné z WWW: <http://www.sharepoint4arabs.com/AymanElHattab/Lists/Posts/Post.aspx?ID=118>.

[2] *EXSLT* [online]. 2002 [cit. 2010-03-19]. Dostupné z WWW: http://exslt.org/

[3] KAY, Michael. *XSLT 2.0 Programmer's Reference, Third Edition*. [s.l.] : Wrox Press, 2004. 955 s. ISBN 0764569090.

[4] MAREŠ, Vladimír. *Dotazovací jazyky pro XML a nativní XML databáze* [online]. České Budějovice : Jihočeská univerzita, 2005. 98 s. Bakalářská práce. Pedagogická fakulta Jihočeské univerzity, Katedra informatiky. Dostupné z WWW: <http://home.pf.jcu.cz/~pepe/Diplomky/mares.pdf>.

[5] *Microsoft Office Online* [online]. 2010 [cit. 2010-03-22]. Microsoft Office InfoPath. Dostupné z WWW: <http://office.microsoft.com/cs-cz/infopath/HA101514381029.aspx?pid=CH100598271029>.

[6] *Microsoft Office Online* [online]. 2010 [cit. 2010-04-07]. Přehled jazyka XML v aplikaci Excel. Dostupné z WWW: <http://office.microsoft.com/cs-cz/excel/HA102063961029.aspx?pid=CH100648521029>.

[7] *MSDN* [online]. 2010 [cit. 2010-03-22]. Using XSLT and the Open XML SDK 2.0 to Create a Word 2007 Document. Dostupné z WWW: <http://msdn.microsoft.com/en-us/library/ee872374.aspx>.

[8] NIC, Miloslav. *XSLT 2.0 Tutorial* [online]. 2005-12-13 [cit. 2010-04-19]. Dostupné z WWW: <http://zvon.org/xxl/XSL-Ref/Tutorials/>

[9] Poznáváme C# a Microsoft.NET 47. díl – použití XSL transformací. *Poznáváme C# a Microsoft.NET* [online]. 4.11.2005, 47, [cit. 2010-03-18]. Dostupný z WWW: <http://www.zive.cz/Clanky/Poznavame-C-a-MicrosoftNET-47-dil--pouziti-XSL-transformaci/sc-3-a-127435/default.aspx>.

[10] TENNISON, Jeni. *Beginning XSLT 2.0 From Novice to Professional*. United States of America : United States of America 9 8 7 6 5 4 3 2 1, 2005. 824 s. Dostupné z WWW: <http://apress.com/book/view/9781590593240>. ISBN 978-1-59059-324-0.