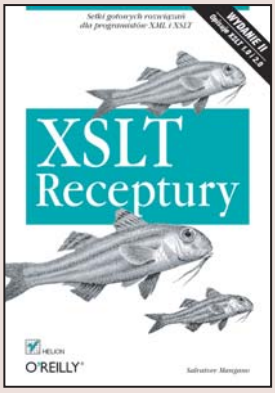
XSLT. Receptury. Wydanie II

Autor: Salvatore Mangano
Tłumaczenie: Anna Trojan
ISBN: 83-246-0461-8
Tytuł oryginału: XSLT Cookbook
Format: B5, stron: 664



Setki gotowych rozwiązań dla programistów XML i XSLT

Język XSLT to jedna z najważniejszych technologii służących do przekształcania dokumentów XML. Za pomocą tego języka można pobierać dane XML, przekształcać je na strony HTML, a nawet generować na ich podstawie wykresy w formacie SVG. Niniejsza książka to praktyczny przewodnika po tych oraz wielu innych funkcjach języka XSLT, a przedstawiony w niej materiał obejmuje także rozbudowane możliwości najnowszej wersji – XSLT 2.0.

Książka „XSLT. Receptury. Wydanie II" zawiera setki gotowych rozwiązań problemów stojących przed programistami używającymi XSLT. Znajdziesz tu sposoby wykonania różnych zadań związanych z transformacją danych XML, zarówno tych podstawowych, jak i skomplikowanych. Poznasz rozmaite techniki przetwarzania dokumentów XML bazujące na obu wersjach języka XSLT. Zrozumiesz także praktyczne zagadnienia związane z wydajnością tworzonych rozwiązań i wygodą ich stosowania. W wielu recepturach znajdziesz alternatywne rozwiązania problemów, dzięki czemu będziesz mógł wybrać technikę najbardziej odpowiadającą wykonywanemu przez Ciebie zadaniu.

- Opis XSLT 2.0
- Wprowadzenie do języka XPath
- Wyszukiwanie danych w dokumentach XML
- Przekształcanie danych XML na różne formaty (zwykły tekst, HTML, SVG, XML)
- Przetwarzanie łańcuchów znaków i wyrażeń matematycznych
- Obsługa dat i czasu
- Obsługa zapytań XML
- Obsługa XSLT w innych językach
- Generowanie kodu
- Zaawansowane zastosowania XSLT
- Testowanie arkuszy XSLT

Gotowe rozwiązania przedstawione w tej książce pomogą Ci w szybkim tworzeniu niezawodnych programów

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 1.

XPath

Neo, prędzej czy później zorientujesz się, tak jak ja wcześniej,
że jest różnica pomiędzy poznaniem ścieżki a kroczeniem po niej.

— Morpheus („Matrix")

1.0. Wprowadzenie

XPath jest językiem wyrażeń, który jest fundamentalny dla przetwarzania XML. Osiągnięcie mistrzostwa w posługiwaniu się XSLT bez biegłości w XPath jest tak samo prawdopodobne, jak osiągnięcie mistrzostwa w posługiwaniu się językiem polskim bez znajomości alfabetu. Niektórzy czytelnicy pierwszego wydania *XSLT. Receptury* zrugali mnie za pominięcie XPath. Ten rozdział został dodany częściowo, by zaspokoić ich oczekiwania, a częściowo ze względu na coraz większe możliwości najnowszych specyfikacji XPath 2.0. Wiele z podanych tutaj receptur da się jednak zastosować także do XPath 1.0.

W XSLT 1.0 XPath odgrywa trzy kluczowe role. Po pierwsze, jest używany w szablonach do adresowania wewnątrz dokumentu podczas jego przetwarzania (w celu ekstrakcji danych). Po drugie, składnia XPath używana jest jako język wzorców w regułach dopasowania dla szablonów. Po trzecie, używany jest do prostych obliczeń matematycznych i manipulacji łańcuchami znaków poprzez wbudowane w XPath operatory i funkcje.

XSLT 2.0 utrzymuje i wzmacnia zależność od XPath 2.0 poprzez korzystanie z nowych możliwości obliczeniowych XPath 2.0. Właściwie można nawet postawić tezę, że rozszerzone możliwości XSLT 2.0 wynikają w dużej mierze z rozwoju w XPath 2.0. Nowe opcje XPath 2.0 obejmują sekwencje, wyrażenia regularne, wyrażenia warunkowe i iteracyjne, a także ulepszony system typów zgodnych z XML Schema wraz z dużą ilością nowych, wbudowanych funkcji.

Każda receptura w tym rozdziale jest zbiorem minireceptur do rozwiązywania pewnych klas problemów z XPath, które często pojawiają się przy używaniu XSLT. Oznaczamy każde wyrażenie XPath zgodnie z konwencją komentarzy XPath 2.0 `(: komentarz :)`, jednak użytkownicy XPath/ XSLT 1.0 powinni mieć świadomość, że komentarze takie nie są dozwoloną składnią. Pokazując rezultat obliczeń XPath, który jest pusty, używamy `()`, co jest jednocześnie metodą zapisu pustej sekwencji w XPath 2.0.

1.1. Efektywne używanie osi

Problem

Należy wybrać węzły w drzewie XML w sposób uwzględniający skomplikowane relacje wewnątrz struktury hierarchicznej.

Rozwiązanie

Każde z poniższych rozwiązań zorganizowane jest wokół powiązanych zbiorów osi. Dla każdej grupy przedstawiany jest przykładowy dokument XML wraz z węzłem kontekstowym (ang. *context node*) opisanym czcionką pogrubioną. Załączono także wyjaśnienie efektu wyznaczania ścieżki wraz z oznaczeniem węzłów, które zostaną zaznaczone w wyróżnionym kontekście. W niektórych przypadkach w rozwiązaniu inne węzły zostaną potraktowane jako kontekst służący zilustrowaniu finezyjności danej ścieżki wyrażenia.

Osie dziecka i potomka

Oś dziecka jest domyślną osią w XPath. Oznacza to, że nie ma potrzeby używania specyfikacji osi `child::`, choć osoby pedantyczne mogą tak uczynić. Można sięgnąć w głąb drzewa XML, korzystając z osi `descendant::` i `descendant-or-self::`. Pierwsza z nich wyklucza węzeł kontekstowy, natomiast druga — uwzględnia go.

```
<Test id="descendants">
   <parent>
      <X id="1"/>
      <X id="2"/>
      <Y id="3">
         <X id="3-1"/>
         <Y id="3-2"/>
         <X id="3-3"/>
      </Y>
      <X id="4"/>
      <Y id="5"/>
      <Z id="6"/>
      <X id="7"/>
      <X id="8"/>
      <Y id="9"/>
   </parent>
</Test>
```

```
(: Wybierz wszystkie dzieci o nazwie X :)
X     (: to samo co child::X :)

Wynik: <X id="1"/> <X id="2"/> <X id="4"/> <X id="7"/> <X id="8"/>

(: Wybierz pierwsze dziecko X :)

X[1]

Wynik: <X id="1"/>

(: Wybierz ostatnie dziecko X :)

X[last( )]

Wynik: <X id="8"/>
```

(: Wybierz pierwszy element, pod warunkiem że jest to X. W przeciwnym wypadku zwróć pusty. :)

```
*[1][self::X]
```

```
Wynik: <X id="1"/>
```

(: Wybierz ostatnie dziecko, pod warunkiem że jest to X. W przeciwnym wypadku zwróć pusty. :)

```
*[last( )][self::X]
```

```
Wynik: ( )
```

```
*[last( )][self::Y]
```

```
Wynik: <Y id="9"/>
```

(: Wybierz wszystkich potomków o nazwie X. :)

```
descendant::X
```

```
Wynik: <X id="1"/> <X id="2"/> <X id="3-1"/> <X id="3-3"/> <X id="4"/> <X id="7"/>
<X id="8"/>
```

(: Wybierz węzeł kontekstowy, pod warunkiem że jest to X, i wszystkich potomków o nazwie X. :)

```
descendant-or-self::X
```

```
Wynik: <X id="1"/> <X id="2"/> <X id="3-1"/> <X id="3-3"/> <X id="4"/> <X id="7"/>
<X id="8"/>
```

(: Wybierz węzeł kontekstowy i wszystkich potomków. :)

```
descendant-or-self::*
```

```
Wynik: <parent> <X id="1"/> <X id="2"/> <Y id="3"> <X id="3-1"/> <Y id="3-2"/> <X
id="3-3"/> </Y> <X id="4"/> <Y id="5"/> <Z id="6"/> <X id="7"/> <X id="8"/> <Y
id="9"/> </parent> <X id="1"/> <X id="2"/> <Y id="3"> <X id="3-1"/> <Y id="3-2"/> <X
id="3-3"/> </Y> <X id="3-1"/> <Y id="3-2"/> <X id="3-3"/> <X id="4"/> <Y id="5"/> <Z
id="6"/> <X id="7"/> <X id="8"/> <Y id="9"/>
```

Osie rodzeństwa

Osie rodzeństwa obejmują `preceding-sibling::` i `following-sibling::`. Zgodnie z nazwą oś `preceding-sibling` zawiera rodzeństwo poprzedzające węzeł kontekstowy, z kolei `following-sibling` — następujące po nim. *Rodzeństwo* to oczywiście węzły dzieci mających tego samego rodzica. Większość przykładów poniżej korzysta z `preceding-sibling::`, jednak bez większych problemów można także obliczyć wynik dla `following-sibling::`.

Należy pamiętać, że użycie pozycyjnego wyrażenia ścieżkowego w formie `preceding-sibling::*[1]` oznacza odniesienie się do węzła rodzeństwa bezpośrednio poprzedzającego węzeł kontekstowy, a nie do pierwszego węzła rodzeństwa w kolejności dokumentu. Niektóre osoby mogą być zdziwione, gdyż wyniki zwracane są w sekwencji zgodnej z kolejnością w dokumencie bez względu na to, czy używa się `preceding-sibling::` czy `following-sibling::`. Wyrażenie `../X`, nie będąc w istocie wyrażeniem dotyczącym osi, w praktyce jest wykorzystywane do wybrania zarówno poprzedniego, jak i następnego węzła rodzeństwa o nazwie X wraz z węzłem kontekstowym, jeśli nosiłby on nazwę X. Z formalnego punktu widzenia jest to skrót od `parent::node()/X`. Należy zauważyć, że `(preceding-sibling::*)[1]` oraz `(following-sibling::*)[1]` wybiorą pierwszy poprzedni lub następny węzeł rodzeństwa zgodnie z kolejnością w dokumencie.

```
<!-- Przykładowy dokument z zaznaczonym węzłem kontekstowym. -->
<Test id="preceding-siblings">
   <A id="1"/>
   <A id="2"/>
   <B id="3"/>
   <A id="4"/>
   <B id="5"/>
   <C id="6"/>
   <A id="7"/>
   <A id="8"/>
   <B id="9"/>
</Test>
```

(: Wybierz wszystkie węzły rodzeństwa A poprzedzające węzeł kontekstowy. :)

```
preceding-sibling::A
```

```
Wynik: <A id="1"/> <A id="2"/> <A id="4"/>
```

(: Wybierz wszystkie węzły rodzeństwa A następujące po węźle kontekstowym. :)

```
following-sibling::A
```

```
Wynik: <A id="8"/>
```

(: Wybierz wszystkie węzły rodzeństwa poprzedzające węzeł kontekstowy. :)

```
preceding-sibling::*
```

```
Wynik: <A id="1"/> <A id="2"/> <B id="3"/> <A id="4"/> <B id="5"/> <C id="6"/>
```

(: Wybierz pierwszy poprzedzający węzeł rodzeństwa o nazwie A, licząc od węzła kontekstowego (bieżącego). :)

```
preceding-sibling::A[1]
```

```
Wynik: <A id="4"/>
```

(: Element bezpośrednio poprzedzający węzeł kontekstowy (bieżący), pod warunkiem że węzłem tym jest A. :)

```
preceding-sibling::*[1][self::A]
```

```
Wynik: ( )
```

(: Gdyby węzłem kontekstowym był <A id="8"/>, wynikiem byłoby <A id="7"/>. :)

(: Wszystkie poprzedzające węzły rodzeństwa niebędące A. :)

```
preceding-sibling::*[not(self::A)]
```

```
Wynik: <B id="3"/> <B id="5"/> <C id="6"/>
```

(: Dla następnych receptur należy użyć tego dokumentu. :)

```
<Test id="preceding-siblings">
   <A id="1">
      <A/>
   </A>
   <A id="2"/>
   <B id="3">
      <A/>
   </B>
   <A id="4"/>
   <B id="5"/>
   <C id="6"/>
   <A id="7"/>
   <A id="8"/>
   <B id="9"/>
</Test>
```

(: Element bezpośrednio poprzedzający węzeł kontekstowy, pod warunkiem że ma on dziecko A. :)

```
preceding-sibling::*[1][A]
```

```
Wynik: ( )
```

(: Pierwszy z elementów poprzedzających węzeł kontekstowy, mających dziecko A. :)
```
preceding-sibling::*[A][1]
```

```
Wynik: <B id="3"> ...
```

(: XPath 2.0 pozwala na większą elastyczność w wyborze elementów w odniesieniu do ich przestrzeni nazw. W następnych recepturach wykorzystywany jest następujący dokument XML. :)

```
<Test xmlns:NS="http://helion.pl/ksiazki/xsltre/ns/1"
xmlns:NS2="http://helion.pl/ksiazki/xsltre/ns/2">
   <NS:A id="1"/>
   <NS2:A id="2"/>
   <NS:B id="3"/>
   <NS2:B id="3"/>
</Test>
```

(: Wybierz wszystkie elementy rodzeństwa poprzedzające węzeł kontekstowy, których przestrzeń nazw jest przestrzenią skojarzoną z prefiksem NS. :)
```
preceding-sibling::NS:*
```

```
Wynik: <NS:A id="1"/>
```

(: Wybierz wszystkie węzły rodzeństwa poprzedzające węzeł kontekstowy, których lokalną nazwą jest A. :)
```
preceding-sibling::*:A
```

```
Wynik: <NS:A id="1"/>, <NS2:A id="2"/>
```

Osie rodzica i przodków

Oś rodzica (`parent::`) odnosi się do rodzica węzła kontekstowego. Wyrażenie `parent::X` nie powinno być mylone z `../X`. To pierwsze utworzy sekwencję z dokładnie jednym elementem, pod warunkiem że rodzicem węzła kontekstowego będzie X, bądź zwróci pusty element. To drugie jest skróconym zapisem wyrażenia `parent::node()/X`, które wybierze wszystkie węzły rodzeństwa o nazwie X dla danego węzła kontekstowego, włącznie z samym węzłem kontekstowym, o ile nosi on nazwę X.

By przejść do wyższych poziomów drzewa XML (dziadków, pradziadków, prapradziadków i tak dalej), należy użyć wyrażenia `ancestor::` lub `ancestor-or-self::`. To pierwsze pomija węzeł kontekstowy, to drugie zawiera go.

(: Wybierz rodzica węzła kontekstowego, pod warunkiem że jest nim element X. W przeciwnym wypadku zwróć pusty element. :)
```
parent::X
```

(: Wybierz rodzica węzła kontekstowego. Może być pusty, tylko jeśli węzeł kontekstowy jest elementem z najwyższego poziomu. :)
```
parent::*
```

(: Wybierz rodzica, jeśli znajduje się on w przestrzeni nazw skojarzonych z prefiksem NS. Prefiks musi być zdefiniowany, w przeciwnym razie jest to błąd. :)
```
parent::NS:*
```

(: Wybierz rodzica bez względu na jego przestrzeń nazw, pod warunkiem że jego lokalna nazwa to X. :)
```
parent::*:X
```

(: Wybierz wszystkich przodków (z rodzicem włącznie) o nazwie X. :)
```
ancestor::X
```

(: Wybierz wszystkich przodków węzła kontekstowego o nazwie X oraz sam węzeł kontekstowy, jeśli nazywa się X :)
```
ancestor-or-self::X
```

Osie poprzedzające i następujące

Osie poprzedzające i następujące mają potencjalnie możliwość wyboru dużej liczby węzłów, gdyż biorą one pod uwagę wszystkie elementy poprzedzające węzeł kontekstowy lub po nim następujące w kolejności dokumentu, z wyłączeniem węzłów przodków lub potomków. Oś następująca pomija potomków, a oś poprzedzająca — przodków. Nie należy zapomnieć, że obie osie pomijają węzły przestrzeni nazw i ich atrybuty.

```
(: Wszystkie węzły poprzedzające o nazwie X. :)
preceding::X

(: Najbliższy węzeł poprzedzający o nazwie X. :)
preceding::X[1]

(: Najdalszy węzeł następujący o nazwie X. :)
following::X[last( )]
```

Analiza

XPath używa pojęcia osi do podzielenia drzewa dokumentu na podzbiory względem pewnego węzła zwanego węzłem kontekstowym. Ogólnie rzecz biorąc, podzbiory te nakładają się na siebie; jednak osie przodka, potomka, poprzedzające i następujące dzielą dokument (ignorując węzły przestrzeni nazw i atrybutów): nie nakładają się na siebie i razem zawierają wszystkie węzły dokumentu. Węzeł kontekstowy jest ustalany przez język zawierający XPath. W XSLT kontekst ustala się poprzez:

- dopasowanie szablonu (`<xsl:template match="x"> ... </xsl:template>`),
- `xsl:for-each`,
- `xsl:apply-templates`.

Efektywne korzystanie z tego typu wyrażeń ścieżkowych stanowi klucz do wykonywania zarówno prostych, jak i bardziej skomplikowanych transformacji. Doświadczenie z tradycyjnymi językami programowania czasami dezorientuje i prowadzi do błędów przy użyciu XPath. Ja na przykład często łapię się na pisaniu czegoś w stylu `<xsl:if test="preceding-sibling::X[1]"> </xsl:if>`, kiedy naprawdę mam na myśli `<xsl:if test="preceding-sibling::*[1][self::X]"> </xsl:if>`. Prawdopodobnie dzieje się tak, ponieważ to drugie jest mało intuicyjnym sposobem powiedzenia „sprawdź, czy bezpośrednio następujący węzeł rodzeństwa jest X”.

Oczywiście niemożliwością jest przedstawienie każdej przydatnej kombinacji wyrażeń ścieżkowych z użyciem osi. Jednak jeśli zrozumie się części składowe przedstawione wcześniej, jest się na najlepszej drodze do zrozumienia znaczenia konstrukcji takich jak `preceding-sibling::X[1]/descendant::Z[A/B]` lub jeszcze bardziej skomplikowanych.

1.2. Filtrowanie węzłów

Problem

Należy wybrać węzły w oparciu o dane, które zawierają, a nie ich nazwę lub pozycję.

Rozwiązanie

Wiele z minireceptur z Receptury 1.1 używa predykatów do filtrowania węzłów, jednak te predykaty oparte były ściśle na pozycji węzła lub jego nazwie. Tutaj rozważymy różne predykaty, które filtrują na podstawie zawartości danych. W poniższych przykładach używamy prostej ścieżki elementu dziecka X przed każdym predykatem, jednak można w miejsce X wstawić dowolne inne wyrażenie ścieżkowe, łącznie z tymi wymienionymi w Recepturze 1.1.

W poniższych przykładach używamy operatorów porównania z XPath 2.0 (`eq`, `ne`, `lt`, `le`, `gt` i `ge`) w miejsce operatorów (=, !=, <, <=, > i >=). Dzieje się tak, ponieważ przy porównywaniu wartości niepodzielnych preferowane są nowe operatory. W XPath 1.0 do dyspozycji jest tylko ten drugi zestaw operatorów, należy zatem je odpowiednio zastąpić. Nowe operatory zostały wprowadzone w XPath 2.0, gdyż mają prostszą semantykę i w rezultacie powinny być bardziej wydajne. Złożoność wcześniejszych operatorów pojawia się w przypadkach, gdy po którejś stronie porównania znajduje się sekwencja. Więcej na ten temat znajduje się w Recepturze 1.8.

Jeszcze jedna uwaga odnosi się do XPath 2.0 i związana jest z faktem, że wersja ta dołącza informację o typie tam, gdzie dostępny jest schemat. Może to sprawić, że w poniższych wyrażeniach pojawią się błędy typu. Na przykład `X[@a = 10]` nie jest tym samym, co `X[@a = '10']`, jeśli atrybut `a` jest liczbą całkowitą. Tutaj zakładamy, że schemat nie występuje i dlatego wszystkie wartości niepodzielne są typu `untypedAtomic`. Więcej na ten temat znajduje się w Recepturach 1.9 i 1.10.

(: Wybierz dzieci X, które mają atrybut a. :)
```
X[@a]
```

(: Wybierz dzieci X, które mają przynajmniej jeden atrybut. :)
```
X[@*]
```

(: Wybierz dzieci X, które mają przynajmniej trzy atrybuty. :)
```
X[count(@*) > 2]
```

(: Wybierz dzieci X, których atrybuty sumują się do wartości mniejszej niż 7. :)
```
X[sum(for $a in @* return number($a)) < 7]
```
(: W XSLT 1.0 należy użyć sum(@) < 7 :)*

(: Wybierz dzieci X, które nie mają atrybutu o nazwie a. :)
```
X[not(@a)]
```

(: Wybierz dzieci X, które nie mają żadnych atrybutów. :)
```
X[not(@*)]
```

(: Wybierz dzieci X, które mają atrybut o nazwie a i wartości '10'. :)
```
X[@a eq '10']
```

(: Wybierz dzieci X, które mają dziecko o nazwie Z i o wartości '10'. :)
```
X[Z eq '10']
```

(: Wybierz dzieci X mające dziecko o nazwie Z i wartości, która nie jest równa '10'. :)
```
X[Z ne '10']
```

(: Wybierz dzieci X, jeśli mają one przynajmniej jedno dziecko będące węzłem tekstowym. :)
```
X[text( )]
```

(: Wybierz dzieci X, jeśli mają węzeł tekstowy z przynajmniej jednym znakiem niebędącym znakiem białym (spacja, tabulator itd.). :)
```
X[text( )[normalize-space(.)]]
```

```
(: Wybierz dzieci X, jeśli mają one jakieś dziecko. :)
X[node( )]

(: Wybierz dzieci X, jeśli zawierają one węzeł komentarza. :)
X[comment( )]

(: Wybierz dzieci X, jeśli zawierają one @a, którego wartość numeryczna jest mniejsza od 10. Wyrażenie to będzie działało
zarówno w XPath 1.0, jak i 2.0, bez względu na to, czy @a jest łańcuchem znaków czy liczbą. :)

X[number(@a) < 10]

(: Wybierz X, jeśli ma co najmniej jeden poprzedzający go węzeł rodzeństwa o nazwie Z z atrybutem y, który nie jest
równy 10. :)

X[preceding-sibling::Z/@y ne '10']

(: Wybierz dzieci X, których wartość łańcucha znaków składa się z jednego znaku spacji. :)
X[. = ' ']

(: Dziwny sposób uzyskania pustej sekwencji! :)
X[false( )]

(: To samo co użycie wyrażenia X. :)
X[true( )]

(: Elementy X z dokładnie 5 dzieci. :)
X[count(*) eq 5]

(: Elementy X z dokładnie 5 dzieci (włączając w to węzły elementów, tekstowe, komentarzy i PI, jednak nie węzły atrybutów). :)
X[count(node( )) eq 5]

(: Elementy X, które mają dokładnie 5 węzłów dowolnego typu. :)
X[count(@* | node( )) eq 5]

(: Pierwsze dziecko X, jeśli ma wartość 'some text'; w przeciwnym wypadku zwraca pusty element. :)
X[1][. eq 'some text']

(: Wybierz wszystkie dzieci X o wartości 'some text' i zwróć pierwsze z nich lub pusty element, jeśli takie dziecko nie występuje.
Ujmując to w prostszy sposób, pierwsze dziecko, które ma wartość łańcucha znaków 'some text'. :)
X[. eq 'some text'][1]
```

Analiza

Tak jak w przypadku Receptury 1.1 niemożliwe jest przedstawienie wszystkich interesujących kombinacji predykatów filtrujących. Jednakże opanowanie podanych wyżej przykładów powinno pomóc w późniejszym napisaniu dowolnych innych wyrażeń filtrujących. Należy też zauważyć, że można tworzyć jeszcze bardziej skomplikowane warunki, używając operatorów logicznych `and`, `or` i funkcji `not()`.

```
number(@a) > 5 and X[number(@a) < 10]
```

Jeśli używa się predykatów ze skomplikowanymi wyrażeniami ścieżkowymi, należy rozumieć efekt zastosowania nawiasów.

```
(: Wybierz pierwsze dziecko Y każdego dziecka X węzła kontekstowego. Wyrażenie to może skutkować zwróceniem sekwencji
więcej niż jednego Y. :)
X/Y[1]

(: Wybierz sekwencję węzłów X/Y, a następnie weź pierwszy z nich. To wyrażenie zwraca maksymalnie jeden Y. :)
(X/Y)[1]
```

Informatyk powiedziałby, że operator warunkowy `[]` wiąże bardziej, niż operator ścieżkowy `/`.

1.3. Praca z sekwencjami

Problem

Należy przetworzyć zbiory dowolnych węzłów i niepodzielnych wartości pochodzących z dokumentu bądź dokumentów XML.

Rozwiązanie

XPath 1.0

W XPath 1.0 nie istnieje pojęcie sekwencji i dlatego te receptury nie mają zastosowania. XPath 1.0 ma zbiory węzłów. Istnieje jednak idiomatyczny sposób, by skonstruować pusty zbiór węzłów z użyciem XSLT 1.0.

```
(: Pusty zbiór węzłów. :)
/..
```

XPath 2.0

```
(: Konstruktor pustej sekwencji. :)
( )

(: Sekwencja składająca się z jednego niepodzielnego elementu 1. :)
1

(: Należy skorzystać z przecinka, by skonstruować sekwencję. Tutaj budujemy sekwencję wszystkich dzieci X danego kontekstu,
następnie wszystkich dzieci Y oraz wszystkich dzieci Z. :)
X, Y, Z

(: Należy skorzystać z operatora "to", by skonstruować przedziały. :)
1 to 10

(: Tutaj łączymy przecinek z kilkoma przedziałami. :)
1 to 10, 100 to 110, 17, 19, 23

(: Zmienne i funkcje także mogą być wykorzystane. :)
1 to $x

1 to count(para)

(: Sekwencje nie zagnieżdżają się, zatem poniższe dwie sekwencje są takie same. :)

((1,2,3), (4,5, (6,7), 8, 9, 10))

1,2,3,4,5,6,7,8,9,10

(: Operator "to" nie może stworzyć bezpośrednio sekwencji malejącej. :)
10 to 1 (: Ta sekwencja jest pusta! :)

(: Zamierzony efekt da się jednak osiągnąć w następujący sposób. :)
for $n in 1 to 10 return 11 - $n

(: Usuń z sekwencji duplikaty. :)
distinct-values($seq)

(: Zwróć wielkość sekwencji. :)
count($seq)
```

```
(: Sprawdź, czy sekwencja jest pusta. :)
empty($seq) (: lepsze niż użycie count($seq) eq 0 :)

(: Znajdź wszystkie pozycje elementu w sekwencji. "index-of" zwraca indeksy jako sekwencję liczb całkowitych
odpowiadających każdemu elementowi sekwencji podanej jako pierwszy argument, który jest równy z elementem
podanym jako drugi argument. :)
index-of($seq, $item)

(: Określanie podzbiorów. :)

(: Do 3 elementów z $seq, rozpoczynając od drugiego. :)
subsequence($seq, 2, 3)

(: Wszystkie elementy z $seq na pozycji od 3 od końca sekwencji. :)
subsequence($seq, 3)

(: Wstaw sekwencję $seq2 przed trzeci element sekwencji początkowej, $seq1. :)
insert-before($seq1, 3, $seq2)

(: Skonstruuj nową sekwencję, która zawiera wszystkie elementy sekwencji $seq1 z wyjątkiem trzeciego. :)
remove($seq1, 3)

(: Jeśli należy usunąć kilka elementów, można rozważyć następujące wyrażenia. :)

$seq1[not(position( ) = (1,3,5))]

$seq1[position( ) gt 3 and position( ) lt 7]
```

Analiza

W XPath 2.0 każdy element danych (wartość) jest sekwencją. Dlatego niepodzielna wartość 1 jest taką samą sekwencją jak wynik wyrażenia `(1 to 10)`. Innym sposobem przedstawienia tego faktu jest powiedzenie, że każde wyrażenie XPath 2.0 jest przekształcane na sekwencję. Sekwencja może zawierać zero lub więcej wartości i te wartości mogą być węzłami, wartościami niepodzielnymi lub mieszanką obu. Odniesienie się do pojedynczych elementów sekwencji rozpoczyna się od pozycji 1 (nie 0, jak mogłaby oczekiwać osoba znająca języki C lub Java).

XPath 1.0 nie ma sekwencji, a raczej zbiory węzłów. Zbiory węzłów nie są koncepcją tak jasną, jak sekwencje, ale w wielu przypadkach różnice między nimi nie mają znaczenia. Przykładowo, dowolne wyrażenie XPath 1.0 korzystające z funkcji `count()` i `empty()` powinno zachowywać się tak samo w XPath 2.0. Zaletą XPath 2.0 jest to, że sekwencja jest elementem typu „first-class", czyli może być jawnie skonstruowana i przetwarzana z użyciem rozmaitych nowych funkcji XPath 2.0. Receptury w tej części wprowadzają wiele ważnych idiomów sekwencji; kolejne pojawią się w innych recepturach tej książki.

1.4. Skracanie kodu warunków z użyciem wyrażeń „if"

Problem

Przy wyrażaniu prostych warunków „if-then-else" kod XSLT jest zbyt długi i skomplikowany ze względu na użycie rozwlekłego XML.

Rozwiązanie

XPath 1.0

Jest kilka prostych sztuczek, które można zastosować w XPath 1.0, by uniknąć rozwlekłych `xsl:choose` w prostych sytuacjach. Sztuczki te opierają się przede wszystkim na fakcie, że `false` konwertowane jest do 0, a `true` do 1 w przypadku użycia ich w kontekście matematycznym.

Zatem na przykład wartość minimalna, maksymalna i bezwzględna mogą być obliczone bezpośrednio w XPath 1.0. W poniższych przykładach należy założyć, że `$x` i `$y` zawierają liczby całkowite.

```
(: Wartość minimalna. :)
($x <= $y) * $x + ($y < $x) * $y

(: Wartość maksymalna. :)
($x >= $y) * $x + ($y > $x) * $y

(: Wartość bezwzględna. :)
(1 - 2 * ($x < 0)) * $x
```

XPath 2.0

Dla prostych przypadków z rozwiązania dla XPath 1.0 (wartość minimalna, maksymalna, bezwzględna) istnieją teraz wbudowane funkcje XPath. Dla innych prostych warunków należy skorzystać z nowego wyrażenia warunkowego `if`.

```
(: Wartość domyślna brakującego atrybutu jest ustalana na 10. :)
if (@x) then @x else 10

(: Wartość domyślna brakującego elementu jest ustalana na "unauthorized". :)
if (password) then password else 'unauthorized'

(: Ochrona przed dzieleniem przez zero. :)
if ($d ne 0) then $x div $d else 0

(: Tekst elementu będącego argumentem, jeśli zawiera przynajmniej jeden znak inny od znaku białego (spacja, tabulator);
w przeciwnym wypadku pojedyncza spacja. :)
if (normalize-space(para)) then string(para) else ' '
```

Analiza

Większość weteranów programowania w XSLT 1.0 prawdopodobnie kuli się ze strachu za każdym razem, gdy ma dodać do szablonu dodatkowy fragment kodu warunkowego. Ja sam tak robię. Często później z bólem przechodzę przez kolejne zawiłości sposobów dopasowywania do wzorca w XSLT, by tylko zminimalizować ilość kodu z warunkami. Dzieje się tak nawet nie dlatego, że taki kod jest w XSLT bardziej skomplikowany czy niewydajny, a dlatego, że jest tak straszliwie rozwlekły. Proste `xsl:if` nie jest jeszcze takie złe, ale jeśli należy przedstawić logikę „if-then-else", to trzeba skorzystać ze znacznie obszerniejszego `xsl:choose`. Okropność!

W XSLT 2.0 istnieje dla tego alternatywa, jednak pochodzi ona raczej z XPath 2.0, niż z samego XSLT 2.0. Na pierwszy rzut oka można odnieść wrażenie, że XPath został niejako pogwałcony przez wprowadzenie czegoś, co w programowaniu proceduralnym nazywa się ciągiem

instrukcji sterujących. Jednakże jak tylko zacznie się w pełni korzystać z XPath 2.0, wysuwa się wniosek, że zarówno XPath, jak i XSLT zyskały na tych ulepszeniach. Ponadto wyrażenia warunkowe XPath 2.0 nie unieważniają elementu `xsl:if`, a co najwyżej redukują jego użycie w tych przypadkach, gdy jest to niewygodne. Dla zilustrowania powyższych informacji można porównać następujące fragmenty kodu:

```
<!-- XSLT 1.0 -->
<xsl:variable name="size">
   <xsl:choose>
      <xsl:when test="$x &gt; 3">duży</xsl:when>
      <xsl:otherwise>mały</xsl:otherwise>
   </xsl:choose>
</xsl:variable>

<!-- XSLT 2.0 -->
<xsl:variable name="size" select="if ($x gt 3) then 'duży' else 'mały' "/>
```

Sądzę, że większość czytelników będzie wolała rozwiązanie z XPath 2.0, niż to poprzednie, z XSLT 1.0.

Jedną ważną rzeczą w wyrażeniach warunkowych XPath jest to, że `else` nie jest opcjonalne. Programiści języka C docenią ten fakt, porównując go sobie z wyrażeniem `a ? b : c` z tego języka. Często używa się pustego elementu `()` w przypadku, gdy nie ma żadnej sensownej wartości do wstawienia do wyrażenia `else`.

Wyrażenia warunkowe są przydatne do zwracania wartości domyślnych wtedy, gdy nie ma schematu z wartościami domyślnymi.

```
(: Ustalenie wartości domyślnej opcjonalnego atrybutu. :)
if (@optional) then @optional else 'some-default'

(: Ustalenie wartości domyślnej opcjonalnego elementu. :)
if (optional) then optional else 'some-default'
```

Wyrażenia te znajdują swoje zastosowanie także w sytuacjach niezdefiniowanych lub nieoczekiwanych wyników. W poniższym przykładzie mamy potrzebę, by wynikiem było 0, a nie `number('Infinity')`.

```
if ($divisor ne 0) then $dividend div $divisor else 0
```

Oczywiście, można także skonstruować bardziej skomplikowane warunki. Następujący fragment służy do dekodowania numerowanej listy.

```
if (size eq 'XXL') then 50
else if (size eq 'XL') then 45
else if (size eq 'L') then 40
else if (size eq 'M') then 34
else if (size eq 'S') then 32
else if (size eq 'XS') then 29
else -1
```

Jednakże w tym wypadku użycie sekwencji może być ładniejszym rozwiązaniem, szczególnie jeśli zastąpimy dosłowne sekwencje zmiennymi, które mogą być inicjalizowane z zewnętrznego pliku XML.

```
(50,45,40,34,32,29,-1)[(index-of((('XXL', 'XL', 'L', 'M', 'S', 'XS')), size), 7)[1]]
```

Tutaj zakładamy, że węzeł kontekstowy ma tylko jedno dziecko; w przeciwnym razie wyrażenie to jest niedozwolone (choć wtedy można użyć `size[1]`). Polegamy także na fakcie, że jeśli szukany element nie zostaje znaleziony, `index-of` zwraca pustą sekwencję, do której następnie dodajemy 7, by móc obsłużyć przypadek `else`.

1.5. Eliminowanie rekurencji z wyrażeń „for"

Problem

Chcemy przekształcić sekwencję wejściową na sekwencję wyjściową w taki sposób, by każdy element wyjściowej był dowolnie skomplikowaną funkcją wejściowej i by wielkość obu sekwencji niekoniecznie była taka sama.

Rozwiązanie

XPath 1.0

Nie da się zastosować w XPath 1.0. Należy zamiast tego skorzystać z szablonu rekurencyjnego XSLT.

XPath 2.0

Należy w tym wyrażeniu skorzystać z XPath 2.0. Poniżej pokazujemy cztery przypadki, w których wyrażenie `for` może odwzorowywać sekwencje wyjściowe i wejściowe różniące się wielkością.

Agregacja

```
(: Suma kwadratów. :)
sum(for $x in $numbers return $x * $x)

(: Średnia kwadratów. :)
avg(for $x in $numbers return $x * $x)
```

Odwzorowanie

```
(: Odwzorowanie sekwencji słów we wszystkich akapitach na sekwencję długości słów. :)
for $x in //para/tokenize(., ' ') return string-length($x)

(: Odwzorowanie sekwencji słów w akapicie na sekwencję długości słów zawierających więcej niż trzy litery. :)
for $x in //para/tokenize(., ' ') return if (string-length($x) gt 3) the
string-length($x) else ( )

(: To samo, co powyżej, ale z warunkiem dla sekwencji wejściowej. :)
for $x in //para/tokenize(., ' ')[string-length(.) gt 3] return string-length($x)
```

Generowanie

```
(: Generowanie sekwencji kwadratów pierwszych 100 liczb całkowitych. :)
for $i in 1 to 100 return $i * $i

(: Generowanie sekwencji kwadratów w odwrotnej kolejności. :)
for $i in 0 to 10 return  (10 - $i) * (10 - $i)
```

Rozszerzanie

```
(: Odwzorowanie sekwencji akapitów na zduplikowaną sekwencję akapitów. :)
for $x in //para return ($x, $x)

(: Duplikowanie słów. :)
for $x in //para/tokenize(., ' ') return ($x, $x)
```

```
(: Odwzorowanie słów na słowa wraz z ich długością. :)
for $x in //para/tokenize(., ' ') return ($x, string-length($x))
```

Łączenie

```
(: Dla każdego klienta wynikiem jest identyfikator i suma wszystkich jego zamówień. :)
for $cust in doc('customer.xml')/*/customer
    return
     ($cust/id/text( ),
      sum(for $ord in doc('orders.xml')/*/order[custID eq $cust/id]
               return ($ord/total)) )
```

Analiza

Jak zaznaczono w Recepturze 1.4, dodanie instrukcji sterujących do języka wyrażeń takiego jak XPath może na początku wydawać się dziwne lub nawet błędne. Ale gdy tylko odkryje się wyzwalającą moc tych konstrukcji w XPath 2.0, szybko pokonuje się wszelkie wątpliwości. Jest tak w szczególności w przypadku wyrażenia `for` z XPath 2.0.

Siła `for` ujawnia się w pełni, kiedy uświadomi się sobie, jak można wykorzystać to wyrażenie w celu zredukowania wielu skomplikowanych rekurencyjnych rozwiązań XSLT 1.0 do jednej linii wyrażenia XPath 2.0. Rozważmy na przykład problem obliczania sum w XSLT 1.0. Jeśli potrzebna jest tylko prosta suma, nie ma problemu, bo wbudowana funkcja XPath 1.0 `sum` doskonale się sprawdzi. Jednak kiedy potrzebne jest otrzymanie sumy kwadratów, konieczne staje się napisanie większego, niewygodnego i mniej przejrzystego szablonu rekurencyjnego. Właściwie duża część pierwszego wydania tej książki składała się z takich właśnie powtarzalnych rozwiązań do radzenia sobie z rekurencyjną gimnastyką. W XPath 2.0 suma kwadratów to nic innego jak `sum(for $x in $numbers return $x * $x)`, gdzie `$numbers` zawiera sekwencję liczb, które chcemy dodać. Ile drzew mogłem uratować, gdyby to ułatwienie było dostępne w XPath 1.0!

Wyrażenie `for` kryje w sobie jednak jeszcze więcej mocy. Nie jest się już ograniczonym do tylko jednej zmiennej iteracyjnej. Możliwe jest połączenie kilku zmiennych i stworzenie zagnieżdżonych pętli, które mogą tworzyć sekwencje z powiązanych ze sobą w skomplikowanym dokumencie węzłów.

```
(: Zwraca sekwencję składającą się z wartości atrybutu id poszczególnych elementów para oraz elementu ref będącego
dzieckiem danego elementu para. :)
for $s in /*/section,
   $p in $s/para,
   $r in $p/ref
      return ($p/@id, $r)
```

Należy zauważyć, że — poza tym, że jest krótsze — powyższe wyrażenie nie różni się niczym od:

```
for $s in /*/section
   return for $p in $s/para
      return for $r in $p/ref
         return ($p/@id, $r)
```

Warto także zauważyć, że nie ma potrzeby korzystania z zagnieżdżonych pętli `for`, jeśli bardziej elegancką reprezentację zamierzonej sekwencji przedstawia tradycyjne wyrażenie ścieżkowe.

```
(: To użycie "for" jest tylko rozwlekłym sposobem napisania "/*/section/para/ref". :)
for $s in /*/section,
   $p in $s/para,
   $r in $p/ref return $r
```

Czasami warto znać pozycję każdego z elementów sekwencji w czasie jej przetwarzania. Nie można tu użyć funkcji `position()` — tak, jak by to zrobiono dla `xsl:for-each` — gdyż wyrażenie XPath nie zmienia pozycji kontekstu. Ten sam efekt da się jednak osiągnąć, korzystając z poniższego wyrażenia:

```
for $pos in 1 to count($sequence),
   $item in $sequence[$pos]
      return $item , $pos
```

1.6. Radzenie sobie ze skomplikowaną logiką z użyciem kwantyfikatorów

Problem

Należy przetestować sekwencję pod kątem obecności warunku w kilku bądź wszystkich jej elementach.

Rozwiązanie

XPath 1.0

Jeśli warunek oparty jest na równości, wtedy w zupełności wystarczy semantyka operatorów = i != z XPath 1.0 i 2.0.

```
(: Prawdziwe, jeśli występuje odniesienie do przynajmniej jednej części. :)
//section/@id = //ref/@idref

(: Prawdziwe, jeśli istnieje odniesienie do każdej części. :)
count(//section) = count(//section[@id = //ref/@idref])
```

XPath 2.0

W XPath 2.0 należy skorzystać z wyrażeń `some` i `every`, by osiągnąć ten sam efekt.

```
(: Prawdziwe, jeśli występuje odniesienie do przynajmniej jednej części. :)
some $id in //para/@id satisfies $id = //ref/@idref

(: Prawdziwe, jeśli istnieje odniesienie do każdej części. :)
every $id in //section/@id satisfies $id = //ref/@idref
```

W XPath 2.0 można jednak bez większego wysiłku pójść jeszcze dalej.

```
(: Istnieje jakaś część, która zawiera odniesienie do wszystkich sekcji z wyjątkiem siebie samej. :)
some $s in //section satisfies
   every $id in //section[@id ne $s/@id]/@id satisfies $id = $s/ref/@idref

(: $sequence2 jest częścią $sequence1 :)
count($sequence2) <= count($sequence1) and
every $pos in 1 to count($sequence1),
   $item1 in $sequence1[$pos],
   $item2 in $sequence2[$pos] satisfies $item1 = $item2
```

Po usunięciu sprawdzania zliczania `count` w poprzednim wyrażeniu kod ten zapewnia, że przynajmniej pierwsze `count($sequence1)` elementów w `$sequence2` jest takiej samo, jak te odpowiadające im w `$sequence1`.

Analiza

Semantyka =, !=, <, >, <=, >= w XPath 1.0 i 2.0 czasami zaskakuje niewtajemniczonych, kiedy jeden z operandów jest sekwencją lub zbiorem węzłów XPath 1.0. Dzieje się tak, ponieważ operatory są obliczane do wartości logicznej „prawda", jeśli istnieje co najmniej jedna para wartości z każdej strony wyrażenia, która da się porównać zgodnie z relacją. W XPath 1.0 czasami działa to na korzyść użytkownika, jak pokazaliśmy wcześniej, jednak kwestia ta potrafi niekiedy przyprawić o ból głowy i sprawić, że chce się wrócić do piątej klasy podstawówki, gdzie matematyka jeszcze miała sens. Przykładowo, można domniemywać, że relacja `$x = $x` jest zawsze prawdziwa, a jednak nie jest tak, jeśli `$x` jest pustą sekwencją! Wynika to z faktu, że nie da się znaleźć pary równych elementów wewnątrz dwóch pustych sekwencji.

1.7. Używanie operacji na zbiorach

Problem

Należy przetworzyć sekwencje tak, jakby były one zbiorami matematycznymi.

Rozwiązanie

XPath 1.0

Operator | oznaczający sumę zbiorów na węzłach jest obsługiwany przez XPath 1.0, ale trzeba skorzystać z małych sztuczek, żeby uzyskać część wspólną zbiorów i ich różnicę.

```
(: Suma zbiorów. :)
$set1 | $set2

(: Część wspólna zbiorów. :)
$set1[count(. | $set2) = count($set2)]

(: Różnica zbiorów. :)
$set1[count(. | $set2) != count($set2)]
```

XPath 2.0

Operator | jest nadal obecny w XPath 2.0, jednak dodano do niego także `union` jako jego alias. Dodatkowo dostępne są `intersect` i `except` dla uzyskania odpowiednio: części wspólnej zbiorów i ich różnicy.

```
$set1 union $set2

(: Część wspólna zbiorów. :)
$set1 intersect $set2

(: Różnica zbiorów. :)
$set1 except $set2
```

Analiza

W XPath 2.0 zbiory węzłów są zastąpione sekwencjami. W przeciwieństwie do zbiorów węzłów sekwencje są uporządkowane i mogą zawierać zduplikowane elementy. Jednakże przy

użyciu operacji na zbiorach XPath 2.0 zarówno duplikaty, jak i uporządkowanie są ignorowane, zatem sekwencje zachowują się tak samo, jak zbiory. Wynik operacji na zbiorach nie będzie nigdy zawierał duplikatów, nawet jeśli posiadały je zbiory wejściowe.

Operator `except` jest używany w idiomie XPath 2.0 służącym do wybrania wszystkich atrybutów z wyjątkiem podanego zbioru.

```
(: Wszystkie atrybuty z wyjątkiem @a. :)
@* except @a

(: Wszystkie atrybuty z wyjątkiem @a i @b. :)
@* except @a, @b
```

W XPath 1.0 potrzebne są bardziej niezgrabne wyrażenia, jak poniżej:

```
@*[local-name(.) != 'a' and local-name(.) != 'b']
```

Co ciekawe, XPath pozwala na operacje na zbiorach tylko na sekwencjach węzłów. Niedozwolone są wartości niepodzielne. Dzieje się tak, ponieważ operacje na zbiorach odbywają się po identyfikatorze węzła, a nie jego wartości. Efekt zbiorów wartości można uzyskać korzystając z poniższych wyrażeń XPath 2.0; w XPath 1.0 należy skorzystać z rekurencji XSLT. Więcej na ten temat w rozdziale 9.

```
(: Suma zbiorów. :)
distinct-values( ($items1, $items2) )

(: Część wspólna zbiorów. :)
distinct-values( $items1[. = $items2] )

(: Różnica zbiorów. :)
distinct-values( $items1[not(. = $items2)] )
```

Zobacz również

Receptury 9.1 i 9.2 zawierają więcej przykładów operacji na zbiorach.

1.8. Używanie porównań węzłów

Problem

Należy zidentyfikować węzły lub odnieść się do nich według ich pozycji w dokumencie.

Rozwiązanie

XPath 1.0

W poniższych przykładach należy założyć, że `$x` i `$y` zawierają pojedynczy węzeł z tego samego dokumentu. Trzeba też pamiętać, że *kolejność w dokumencie* oznacza kolejność, w jakiej węzły występują w dokumencie.

```
(: Sprawdź, czy $x i $y są tym samym węzłem. :)
generate-id($x) = generate-id($y)

(: Można także skorzystać z tego, że operator | usuwa duplikaty. :)
count($x|$y) = 1
```

```
(: Sprawdź, czy $x poprzedza $y w kolejności w dokumencie (zadziała tylko, jeśli $x i $y nie są atrybutami). :)
count($x/preceding::node( )) < count($y/preceding::node( )) or
$x = $y/ancestor::node( )

(: Sprawdź, czy $x następuje po $y w kolejności w dokumencie (zadziała tylko, jeśli $x i $y nie są atrybutami). :)
count($x/following::node( )) < count($y/following::node( )) or
$y = $x/ancestors::node( )
```

XPath 2.0

```
(: Sprawdź, czy $x i $y są tym samym węzłem. :)
$x is $y

(: Sprawdź, czy $x poprzedza $y w kolejności w dokumencie. :)
$x << $y

(: Sprawdź, czy $x następuje po $y w kolejności w dokumencie. :)
$x >> $y
```

Analiza

Nowe operatory porównania z XPath 2.0 są prawdopodobnie bardziej wydajne i na pewno łatwiejsze do zrozumienia niż ich odpowiedniki z XPath 1.0. Jednakże używając XSLT 2.0, nie napotka się zbyt wielu sytuacji, w których operatory te są wymagane. W wielu sytuacjach zdarza się, że myśli się o użyciu `<<` lub `>>` w miejscu, gdzie preferowany jest element `xsl:for-each-group`.

1.9. Radzenie sobie z rozszerzonym systemem typów XPath 2.0

Problem

Rygorystyczne zasady związane z typami w XPath 2.0 sprawiają, że złorzeczy się na W3C i tęskni za językiem Perl.

Rozwiązanie

Większość niezgodności pomiędzy XPath/XSLT 1.0 i 2.0 wynika z błędów typu. Dzieje się tak bez względu na to, czy schemat jest obecny, czy też nie. Wiele problemów napotkanych przy przenoszeniu dotychczasowych aplikacji pomiędzy XSLT 1.0 i XSLT 2.0 (które różnią się między innymi wersją XPath) można wyeliminować dzięki używaniu trybu zgodności z 1.0.

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

   <!-- ... -->

</xsl:stylesheet>
```

Moim zdaniem każdy w którymś momencie przestaje w końcu używać trybu zgodności. XPath 2.0 posiada kilka udogodnień dla konwersji typów. Po pierwsze, można bezpośrednio zastosować funkcje konwertujące.

```
(: Konwersja pierwszego dziecka X kontekstu na liczbę. :)
number(X[1]) + 17
```

```
(: Konwersja liczby w $n na łańcuch znaków. :)
concat("id-", string($n))
```

XPath 2.0 posiada także konstruktory typów, można więc bezpośrednio kontrolować interpretację łańcucha znaków.

```
(: Stworzenie daty z łańcucha znaków. :)
xs:date("2006-06-01")

(: Stworzenie liczb zmiennoprzecinkowych podwójnej precyzji z łańcucha znaków. :)
xs:double("1.1e8") + xs:double("2300")
```

Wreszcie XPath posiada operatory `castable as`, `cast as` oraz `treat as`. Przez większość czasu korzysta się z dwóch pierwszych z nich.

```
if ($x castable as xs:date) then $x cast as xs:date else xs:date("1970-01-01")
```

Operator `treat as` nie jest konwersją jako taką, a raczej stwierdzeniem, które obiecuje procesorowi XPath, że w czasie wykonywania programu wartość ta będzie zgodna z danym typem. Jeśli tak się nie dzieje, wystąpi błąd typu. W XPath 2.0 dodano `treat as`, by implementatorzy XPath mogli dokonać statycznego (w czasie kompilacji) sprawdzania typów obok sprawdzania dynamicznego (w czasie wykonywania), pozwalając jednocześnie na selektywne wyłączenie sprawdzania statycznego. Statyczne sprawdzanie typów w implementacjach XSLT 2.0 będzie najprawdopodobniej stosunkowo rzadkie, można więc `treat as` na razie zignorować. O wiele bardziej prawdopodobne jest pojawienie się go w wyższej klasy procesorach XQuery, które dokonują statycznego sprawdzania typów, by ułatwić rozmaite optymalizacje.

Analiza

Praca z procesorem XSLT 2.0 w trybie kompatybilności z wersją 1.0 nie oznacza, że nie można korzystać z żadnych nowych cech 2.0. Załącza on po prostu kilka reguł konwersji z XPath 1.0.

- Pozwala na automatyczną konwersję na liczby typów nieliczbowych użytych w kontekście, w którym oczekiwane są liczby, poprzez atomizację, a następnie użycie funkcji `number()`.

  ```
  (: W trybie zgodności następujący fragment zwraca 18.1, jednak powoduje błąd typu w 2.0. :)
  "1.1" + "17"
  ```

- Pozwala na automatyczną konwersję typów innych niż łańcuch znaków na łańcuch znaków w kontekście, w którym oczekiwane są łańcuchy znaków, poprzez atomizację, a następnie użycie funkcji `string()`.

  ```
  (: W trybie zgodności następujący fragment zwraca wynik 2, jednak powoduje błąd typu w 2.0. :)
  string-length(1 + 2 + 3 + 4 + 5)
  ```

- Automatycznie usuwa elementy z sekwencji o rozmiarze dwa lub większym, jeśli są one użyte w kontekście, w którym oczekiwany jest element pojedynczy. Często się tak dzieje, kiedy wynik wyrażenia ścieżkowego podawany jest do funkcji.

  ```
  <poem>
     <line>There once was a programmer from Nantucket.</line>
     <line>Who liked his bits in a bucket.</line>
     <line>He said with a grin</line>
     <line>and drops of coffee on his chin,</line>
     <line>"If XSLT had a left-shift, I would love it!"</line>
  <poem>

  (: W trybie zgodności oba wyrażenia zwracają wynik 43, jednak to pierwsze powoduje błąd typu w 2.0. :)
  string-length(/poem/line)
  string-length(/poem/line[1])
  ```

1.10. Wykorzystywanie rozszerzonego systemu typów XPath 2.0

Problem

Przy przetwarzaniu XML ściśle przestrzega się zaleceń XML Schema i chce się wykorzystać jego zalety.

Rozwiązanie

Jeśli sprawdza się poprawność dokumentów ze schematem, węzłom wynikowym przypisuje się informację o ich typie. Typy te można następnie sprawdzać w XPath 2.0 (a także w dopasowywaniu szablonów w XSLT 2.0).

```
(: Testowanie, czy wszystkie elementy invoiceDate zostały sprawdzone jako daty. :)
if (order/invoiceDate instance of element(*, xs:date)) then "invoicing complete" else
"invoicing incomplete"
```

`instance of` jest przydatny *tylko* w sprawdzaniu poprawności ze schematem. Dodatkowo nie jest wcale odpowiednikiem `castable as`. Przykładowo, `10 castable as xs:positiveInteger` jest zawsze prawdziwe, podczas gdy `10 instance of xs:positiveInteger` nigdy nie będzie prawdziwe, gdyż literały liczbowe są oznaczone jako `xs:decimal`.

Zaletą sprawdzania poprawności jest jednak nie tylko możliwość sprawdzania typów z użyciem `instance of`, lecz także bezpieczeństwo i wygoda wynikająca ze świadomości, że po przejściu przez sprawdzanie poprawności żadne niespodziewane błędy nie powinny się już pojawić. To może prowadzić do bardziej zwięzłych arkuszy stylów.

```
(: Bez sprawdzania poprawności powinno się kodować następująco. :)
for $order in Order return xs:date($order/invoiceDate) - xs:date($order/createDate)

(: Wiedząc, że wszystkie elementy danych zostały sprawdzone, można opuścić konstruktor xs:date :).
for $order in Order return $order/invoiceDate - $order/createDate
```

Analiza

Osobiście wolę używać XML Schema jako dokumentu ze specyfikacją, a nie narzędzia do sprawdzania poprawności. Dlatego też piszę transformacje XSLT w taki sposób, by były odporne na błędy typu i używam bezpośrednich konwersji tam, gdzie jest to konieczne. Arkusze stylów napisane w ten sposób będą działać bez względu na obecność sprawdzania poprawności.

Jeśli zacznie się pisać arkusze stylów polegające na sprawdzaniu poprawności, ogranicza się do takich implementacji, które muszą przeprowadzać sprawdzanie poprawności. Z drugiej strony jeśli standardy firmowe mówią, że wszystkie dokumenty XML muszą być sprawdzane ze schematem przed ich przetworzeniem, równie dobrze można uprościć swój XSLT, opierając się na pewności, że odpowiednie typy danych pojawią się w odpowiednich sytuacjach.