Dokumentace úlohy JSN: JSON2XML v Perl do IPP 2011/2012
Jméno a příjmení: Jan Pacner
Login: xpacne00

Pro řešení uvedeného zadání jsem zvolil využití pouze 3 stěžejních modulů (*Data::Types, JSON, XML::RegExp*). Vzhledem k omezením na zpracování parametrů nebylo možné využít existujících modulů pro jednoduché zpracování parametrů, a tak velkou část skriptu tvoří právě zpracování argumentů. Konkrétně povolené moduly nenabízely možnost výhradní existence daného argumentu (nesmí se opakovat). Ruční zpracování řeším běžným způsobem - v cyklu procházím celé pole a po jednom kontroluji, zdali argument je platný (tzn. zdali se již neobjevil předtím a zdali je hodnota přípustná). Nakonec zkontroluji výlučnost některých argumentů (např. --*start*). Pro zjednodušení práce s vstupními a výstupními soubory jsem si uložil do proměnných i souborové deskriptory a nadále používal pro vstupně-výstupní operace pouze je.

Pro vytvoření stromu ze vstupního JSON souboru jsem zvolil výše uvedený modul „JSON“. Tento umožňuje tvorbu stromu jak inkrementálně, tak „vše naráz“. Inkrementální metoda má jisté nezanedbatelné nevýhody a proto je implementována metoda „vše naráz“. Tato vyžaduje kompletní JSON řetězec v celku. Načetl jsem proto po řádcích celý vstup do proměnné. Čtení po řádcích je nejrychlejší metoda čtení textového vstupu dostupná v jazyce Perl. Proto byla také zvolena (lze nalézt poměrně překvapivé benchmarky). Protože prvotní implementace skriptu nevyhovovala zadání v případě zacházení s čísly a řetězci, bylo nutné obejít limitaci JSON parsovacího modulu, která vychází z podstaty typování v Perlu (proměnná se implicitně přetypovává podle kontextu a potřeb). Řešení napsat si vlastní parser jsem zavrhnul, protože bych se zbavil kvalitního, optimalizovaného a odzkoušeného modulu JSON a strávil s tím více času. Proto si všechny relevantní řetězcové dvojice nejprve označím vlastním prefixem, následně využiji finálně JSON modul a později prefix ve vhodném okamžiku odstraním. Toto řešení velice degraduje celkový výkon výsledného skriptu, ale rychlost nebyla předmětem zadání. Protože module JSON implicitně pracuje s utf-8, nebylo se čeho obávat z hlediska chyb s kódováním spojených. Nakonec modul JSON okamžitě zastaví výpočet s chybou, pokud nalezne nekorektní vstupní řetězec.

Ústřední činností skriptu je procházení vytvořeného stromu a generování příslušných XML entit. XML vyžaduje pro prázdnou entitu zápis <*entita*/> namísto delšího <*entita*><*/entita*>. Tato na první pohled příjemná vlastnost však znamená, že se nějakým způsobem musí „dopředu“ zjistit, zdali entita neobsahuje nějaké položky. V případě JSON stromu je naštěstí toto poměrně jednoduchá záležitost a za tímto účelem vznikly dvě funkce – pro JSON array (*array_empty()*) a JSON object (*hash_empty()*).

Samostatné zpracování stromu je již od pohledu problém vyžadující rekurzi. Tato je zajištěna funkcí *traverse_json()*. V ní je postupně kontrolováno, zdali se jedná o pole, object (objekt), či „finální“ položku v objektu, která je skalárního charakteru. V případě pole proběhne nejprve kontrola na prázdnost (viz. výše) a poté postupný průchod polem a rekurzívní volání *traverse_json()* na jednotlivé prvky. Nutno podotknout, že v rámci tohoto volání se musí předat i informace o indexu daného prvku v poli – tato je předána vždy a až uvnitř zanořeného volání *traverse_json()* se podle argumentu -*t* tato hodnota použije, či nikoliv. Pokud je zadán argument -a, tak se ještě před průchodem pole spočítá jeho velikost a hodnota se vytiskne spolu s hlavičkou daného pole. Pokud se nejedná o pole, je testována přítomnost hash tabulky, která reprezentuje JSON object. Zde je taktéž proveden test na prázdnost a dále průchod přes všechny klíče. Pokud je hodnotou klíče hash nebo pole, vytiskne se příslušná hlavička a rekurzivně se zavolá *traverse_json()*. V opačném případě se zavolá fce *handle_scalar()* se speciálním parametrem indikujícím volání z hash stavu (vysvětlení tohoto šeredného řešení viz. níže). Tato má na starosti obsluhu výstupu pro konkrétní dvojici jméno-hodnota. Pozornost je též nutno věnovat stavu, kdy se hash (potažmo JSON object) nachází v poli. Tento případ jsem původně řešil tak, že se objekt rozgeneroval jako mnoho položek pole, které však úmyslně nebyly indexovány. Až posléze jsem zjistil, že v tomto specifickém případě je nutno generovat „prázdnou“ entitu, indikující samotný objekt a uzavírající jeho položky. Tímto končí problém hash a zbývá poslední možnost – skalární položka v poli. Řešení opět využívá volání fce *handle_scalar()* s parametry vyhovujícími položce pole (což znamená zahrnutí případného indexu pokud byl zadán argument -*t*, nic víc).

Výše zmiňovaná funkce *handle_scalar()* pracuje se třemi argumenty – položkou, hodnotou a speciálním indikátorem. Pokud je hodnota booleovského typu nebo null, je zavolána malá pomocná fuknce *handle_l()*, která zajišťuje konzistentní výstup těchto tří speciálních entit v závislosti na argumentu -*l*, jak již název napovídá. Pokud se jedná o jiné hodnoty, musíme se rozhodnout, zdali jde o číslo, ale to pouze v případě, že byla tato fce *handle_scalar()* volána z kontextu hash. To proto, že ze zadání plyne, že není nutné uvažovat přítomnost řetězce v poli – proto může být řetězec přítomen pouze v hash. Tento předpoklad je využit již při předzpracování vstupního řetězce přidáním prefixu právě těmto řetězcovým dvojicím. V případě, že je to řetězec, odřízneme přidaný prefix z hodnoty, nahradíme pomocí fce normalize() všechny výskyty nevalidních XML znaků za mínus a zkontrolujeme její validitu za pomoci modulu *XML::RegExp*. Pokud byl zadán argument -*c*, nahradíme XML-nevalidní znaky v hodnotě (atributu) za jejich XML varianty. V opačném případě zkontrolujeme validitu atributu a při neshodě zastavíme výpočet s chybou. Nakonec je výsledná položka s hodnotou vypsána v závislosti na argumentu -*s* buď v podobě <*položka*>*hodnota*<*/položka*> nebo <*položka value*=“*hodnota*“/>. Pokud se však jednalo o číslo a ne řetězec, bylo toto převedeno na celé funkcí *to_int()*,

která jako jedna z mála respektuje „matematické“ zaokrouhlování (pět a výše) a nebere v úvahu pravděpodobnost zaokrouhlování nahoru a dolů. Přesněji řečeno tato funkce vlastně provádí následující kód

my $tmp = int(abs($float) + 0.5); my $res = ($float < 0) ? -$tmp : $tmp;

Výsledek je vypsán obdobně jako v případě řetězce výše, avšak ne v závislosti na parametru *-s*, nýbrž *-i*.

Ve výsledném skriptu není explicitně použito žádné uzavírání souborových deskriptorů, protože to interpret Perlu dělá automaticky při ukončení. Protože v zadání nebylo specifikováno, zdali může docházet k „přímému“ generování výstupu inkrementálně a nebo pouze „v celku“ (tzn. až po úspěšném ukončení všech operací), zvolil jsem samostatně, a sice inkrementální přístup, protože je to více v „unixovském“ duchu. Proto také skript v případě chyby na starndartní výstup často něco „stihne“ vypsat.