# DZ Interset 2.0

Hlavní změny:

* Celé přepsat objektově (use Moose).
* Zveřejnit na CPANu.
* Mělo by to být kompatibilní s Treexem, aby se to v něm dalo využívat. Na druhou stranu ale Interset je a zůstává samostatným projektem, který není součástí Treexu. (Po shlédnutí seznamu cpaních balíčků (<http://www.cpan.org/modules/02packages.details.txt.gz>) se mi jako ideální jeví zařazení do jmenného prostoru Lingua.)

Potřebujeme přinejmenším 2 třídy objektů: FeatureStructure a Tagset. Předpokládaná struktura modulů:

Lingua::Interset::FeatureStructure  
Lingua::Interset::Tagset  
Lingua::Interset::CS::Pdt  
Lingua::Interset::CS::Conll

… tohle všechno už budou implementace sad značek (driverů), které budou dědit od Interset::Tagset, případně i od sebe navzájem. První dvě třídy si zase více méně rozdělí to, co je teď v modulu tagset::common. A možná ještě bude pár dalších pomocných tříd, např. Interset::Converter a Interset::Trie.

Martin navrhoval, že by také třídy konkrétních sad značek mohly být pod ::Tagset::. Technicky to není nezbytné a zpočátku se mi to zdálo jako zbytečné prodlužování názvu modulu, ale možná to nakonec není tak špatný nápad.

V úvahu ještě připadá, že i každý rys bude mít svou vlastní třídu. Potom zřejmě bude existovat společný předek Lingua::Interset::Feature a název pro strukturu by se mohl rozdělit čtyřtečkou na Lingua::Interset::Feature::Structure. Čímž bychom se s každou složkou názvu dostali pod 11 znaků. <http://perldoc.perl.org/perlmodlib.html> to doporučuje kvůli omezením některých souborových systémů. Otázka je, jak moc je tohle v dnešní době ještě důležité, zvlášť když nepředpokládám, že by aplikace, které daný modul využívají, mohly vůbec běžet pod MS DOSem nebo něčím podobným. Na CPANu každopádně je řada modulů, které tohle nedodržují. A například je tam taky modul „Lingua::HPSG::FeatureStructure“, což trochu svádí vyrobit analogicky „Lingua::Interset::FeatureStructure“.

## Třída FeatureStructure

Objekt této třídy je feature structure, česky struktura rysů, nezávislá na konkrétní sadě značek nebo jazyku. Seznam rysů je pevně dán (pokud se změní, je to přísně vzato jiná verze Intersetu). Stejně tak seznam povolených hodnot pro každý rys.

* FeatureStructure je zhmotněním jádra Intersetu, které zatím existovalo spíše virtuálně a v dokumentaci (pokud pomineme výčtové funkce, které určovaly seznam povolených rysů a hodnot). Jeho základní schopnosti: nastavit hodnotu rysu (a zkontrolovat, že jde o známý rys a hodnotu), říct hodnotu rysu, případně přímo odpovědět na booleovské dotazy typu is\_noun().
  + Poznámka: Mluvíme-li u FeatureStructure o povolené hodnotě rysu, vždy máme na mysli globálně platný repertoár hodnot pro daný rys. U konkrétního Tagsetu mohou být povolené hodnoty dále omezené, ale to nás teď nezajímá.
  + Možná by se hodilo, ale v současném Intersetu to není, aby FeatureStructure měla taky statickou metodu, která ke každému rysu a hodnotě dokáže vypsat vysvětlivky, nápovědu, příklady apod.
* Aniž by bylo potřeba to explicitně říkat, každý rys má vždy také povolenou prázdnou hodnotu. Prázdný řetězec a nedefinovaná hodnota je totéž. Nerozlišujeme, zda je daný rys pro značkované slovo irelevantní, nebo zda by mohl být za jistých okolností relevantní, ale v tomto případě akorát hodnotu neznáme (příklad: v pražských značkách nás netrápí rozdíl mezi znaky „X“ a „-“).
  + Důsledkem je, že některé rysy booleovské povahy (např. poss, reflex, foreign, abbr, hyph) nemají hodnoty *yes* a *no*, popř. 0 a 1 nebo něco podobného. Prázdná hodnota je v tomto případě totéž, co „no“, resp. 0. V takových případech, kdy rys má jedinou neprázdnou hodnotu, pak obvykle tato hodnota je stejná jako název rysu (chci se vyhnout hodnotě *yes*, abych tím nevyvolával dojem, že existuje hodnota *no*).
    - Tenhle přístup ale nepoužívám za všech okolností. Preferuju ho pro spíše zvláštní vlastnosti, které různé druhy slov mohou a nemusí mít. Někdy se mohou kombinovat s jinými takovými vlastnostmi. Jde spíše o slovnědruhové než morfologické kategorie. Naopak negativeness je pro mě vyloženě morfologický rys, který mají některé slovní druhy některých jazyků a má pak vždy buď kladnou, nebo zápornou hodnotu. Pokud má prázdnou hodnotu, znamená to, že pro daný slovní druh tento rys není relevantní nebo že v daném případě anotátor hodnotu rysu neurčil.
    - I zde je hranice mezi oběma přístupy poněkud neostrá a lze uvažovat o tom, zda tento rozdíl nezrušit. (Jako mnoho jiných nejasností v Intersetu, i tahle dvojakost je podmíněná historicky postupným přizpůsobováním Intersetu jednotlivým tagsetům.)
* Je-li to nutné, může mít rys současně několik hodnot (pole, resp. množinu povolených hodnot). Tímto způsobem lze zachytit např. kombinované hodnoty rodů z pražských značek. Rozhraní třídy FeatureStructure musí být schopno od této komplikace odstínit uživatele, který s ní nepočítá a nechce počítat (např. proto, že primárně pracuje s tagsety, kde k ničemu takovému nedochází).
  + Poznámka: Možnost zadat disjunkci hodnot nebyla myšlena pro případ, kdy hodnota rysu ve vstupní anotaci chyběla. Pro takový případ je většinou určena prázdná hodnota. Např. pokud víme, že slovo je zájmeno, ale chybí nám informace o druhu zájmena, nechceme vložit disjunkci všech druhů zájmen, které pro daný jazyk přicházejí v úvahu.
    - Poznámka k poznámce: Právě zájmena jsou dost komplikovaný a ne zcela dořešený jev a právě pro ně tohle momentálně fungovat nebude. Pokud totiž nenastavíme druh zájmena, ztratí se informace o zájmennosti zcela. Tohle ještě vyžaduje další úpravy koncepce. (Dosud jsem to řešil tak, že jsem použil nějaký defaultní druh zájmena, ale nelíbí se mi to, protože tím vnáším informaci, která v původní anotaci nebyla.)
  + Poznámka 2: Hranice mezi tím, kdy zavést novou hodnotu a kdy raději pracovat s disjunkcí existujících hodnot, je poněkud mlhavá a zatím se mi nechce ji násilím zaostřovat. Příklad: Ani na okamžik jsem si nemyslel, že by bylo vhodné mít v Intersetu samostatnou hodnotu odpovídající pražskému rodu T (= mužský životný | mužský neživotný | ženský). Na druhou stranu ale v Intersetu nakonec existuje rod „com“ (common), který v podstatě odpovídá tomu samému. Není to kvůli pražským značkám a při dekódování pražských značek se to nepoužívá. Je to kvůli skandinávským jazykům, které rozlišují rod střední a rod společný (mužsko-ženský). Švédové tomu pěkně říkají „neutrum“ a „utrum“ ☺. Protože je to v tomhle případě záležitost celého jazyka a nikoli jen skupiny slov, u kterých se to někomu nechtělo rozlišovat, nakonec jsem novou hodnotu zavedl. Mým cílem je mj. omezit používání disjunkcí hodnot na případy, které jsou nějak „divné“ než na něco systémového.
  + Poznámka 3: V Intersetu nelze zatím nijak podchytit, že dva různé rysy mají nějak omezenou kombinaci svých hodnot. Příklad: rod Q a číslo W v pražských značkách. Můžeme říct, že rod je (ženský | střední) a číslo je (jednotné | množné). Nemůžeme ale dodat, že povolené kombinace jsou (ženský-jednotné | střední-množné) a nikoli ty druhé dvě. Jediná možnost je přiřadit jednomu slovu disjunkci dvou nebo více struktur rysů (objektů třídy FeatureStructure). Až dokončíme převod stávajícího Intersetu do objektové verze, mohli bychom doimplementovat i nějakou částečnou podporu kombinací hodnot. Uživatel ale bude muset vědět, co dělá, aby ji mohl plně využít. Např. pokud pro výše uvedený příklad zavolá nejdřív is\_feminine() a potom is\_plural(), dostane asi v obou případech kladnou odpověď, přestože kdyby existovala metoda is\_feminine\_plural(), vrátila by 0.
* Původně jsem se snažil, aby tatáž informace nebyla ve struktuře rysů zachycena na více než jednom místě, tj. aby hodnota nějakého rysu nevyplývala z hodnoty jiného rysu. To se ukázalo jako obtížně udržitelné, protože tatáž hodnota rysu může mít odlišné chování a důsledky v různých jazycích (např. původně jsem čas *perfect* řešil kombinací dokonavého vidu a minulého času; později se ale ukázalo, že bulharština má dokonavá slovesa, která mohou být v čase zvaném *imperfekt*, a musel jsem zavést i odpovídající hodnotu pro čas. Jiný příklad je indefinite typ zájmena vs. hodnota atributu definiteness v arabštině, bulharštině aj.) Nyní se spíš přikláním k tomu, že některé rysy mají hierarchickou podobu: např. čas je v zásadě minulý, přítomný a budoucí, ale v některých jazycích jsou některé tyto hodnoty ještě dále jemněji rozčleněny. Je to potřeba učesat, ale můj pohled se ještě může změnit, jak přibývají další jazyky.
* Některé rysy (zejména pos, tj. slovní druh) by si zasloužily zvláštní zacházení a přídavný komfort v API. Jde o to, že v zásadě pracujeme s informací, která byla v původním korpusu, ale ta je v různých sadách strukturovaná různě. U nás jsou zájmena i adjektivní, u Bulharů dokonce i adverbiální, u Angličanů zase adjektivním zájmenům říkají determiners, někde jsou řadové číslovky prostě adjektiva, příčestí jsou někde řazena pod slovesa, někde pod přídavná jména, někde pod obojí, někde zcela zvlášť, obdobné to může být s podstatnými jmény slovesnými… Takže mnohdy by se hodila např. funkce is\_pronoun(), ale bude potřeba rozmyslet, co přesně má říkat, a zda není náhodou závislá na konkrétní sadě značek (potom by nepatřila sem, ale spíš do třídy Tagset).
* Existují dva zvláštní rysy, tagset a other. Rys other může mít libovolnou hodnotu. Na rozdíl od ostatních hodnot to může být dokonce odkaz na hash, popř. na pole nebo složitější strukturu. Jeho cílem je uchovat informaci, kterou nelze uchovat v ostatních rysech (a z nějakého důvodu si nemyslíme, že je vhodné kvůli právě této informaci rozšiřovat Interset o nové rysy). Díky tomu je možné načíst značku z konkrétní fyzické sady, později ji zase uložit a nic přitom neztratit. Hodnota rysu other má tedy smysl pouze ve vztahu ke konkrétnímu fyzickému tagsetu. Proto musíme mít v rysu tagset uložený identifikátor tohoto tagsetu (kód jazyka, čtyřtečka, identifikátor sady, to vše malými písmeny). Objekt FeatureStructure se tedy v zásadě chová nezávisle na konkrétní fyzické sadě značek, ale v případě, že víme, z jaké sady hodnoty pocházejí, a umíme tuto informaci využít, tak ji pro nás dokáže uchovat.
* FeatureStructure má serializační metodu, která vrátí řetězec s lidsky čitelnými hodnotami rysů: [pos=”noun”, gender=”masc”, number=”sing”, case=”gen”, other={“x” => “strom”, “y” => “jehlicnan”}, tagset=”cs::mujtagset”]. Řetězec obsahuje všechny rysy, které mají neprázdné hodnoty, v jednotném předem stanoveném pořadí. V podstatě jsme tak vytvořili nový fyzický tagset, do kterého se ovšem vejde vše, co se vejde do Intersetu.
* FeatureStructure umí změnit hodnoty svých rysů na základě omezujících požadavků. Omezující požadavek je podmnožina možných hodnot daného rysu, do které se musíme vejít. Jestliže aktuální hodnota rysu není v této podmnožině, FeatureStructure ji změní na jednu z povolených hodnot podle přednastavených heuristik. (Obvykle se nejméně škody napáchá tím, že aktuální hodnotu nahradíme prázdnou hodnotou. Někdy ale existuje intuitivně vhodnější řešení, např. duál můžeme nahradit plurálem.)
  + Omezující požadavek také může být, že u libovolného rysu chceme nejvýše jednu hodnotu, nikoli disjunkci hodnot.

## Třída Tagset

Reprezentuje někým definovanou sadu fyzických značek, tj. řetězců použitých v nějakém korpusu (obvykle jde o nějaký jednoduchý poziční řetězec N znaků, ale může to být třeba i kus kódu XML nebo nějaký jiný text, který obsahuje potřebné informace). Tagset má svůj identifikátor, který je vždy psán malými písmeny anglické abecedy (přestože odpovídající moduly a package boudou v souladu s CPANí kulturou využívat velká písmena): kód jazyka :: kód sady nebo korpusu, tj. např. „cs::pdt“ nebo „cs::conll“. Tento identifikátor se hodí mj. pro výše popsaný rys tagset.

Třída poskytuje tři základní metody:

* decode() – převede fyzický tag, tedy řetězec, na strukturu rysů, tedy objekt třídy FeatureStructure
* encode() – převede strukturu rysů, resp. objekt třídy FeatureStructure, na fyzický tag, tedy řetězec
* list() – dodá seznam fyzických tagů (řetězců), které daný Tagset obsahuje / umožňuje

Další služby, které zatím v Intersetu nejsou, ale asi by se časem hodily:

* Umět nějak popsat specifika tagsetu, např. říct, že nemáme determinátory, protože jsou zahrnuty pod zájmena. A pokud možno nejen to říct, ale nějak to i využít při přechroustání informace do tvaru, se kterým je uživatel zvyklý pracovat.
  + Modelový příkaz: metoda is\_pronoun(). Podle současné verze Intersetu by FeatureStructure::is\_pronoun() vracel 1, jestliže get\_prontype() ne ‘’. Pokud tuto funkci zavolá uživatel Intersetu, který chce abstrahovat od původní sady fyzických značek, jak jen to jde, je tohle v pořádku (resp. možná budeme chtít dotyčnou metodu pojmenovat malinko odlišně, třeba is\_pronominal(), ale to je detail). V jiné situaci je ale třída Tagset a její implementace metody encode(). Jestliže daná sada značek rozlišuje (substantivní) zájmena a (adjektivní) determinátory, budeme chtít, aby is\_pronoun() vracel 1 pouze pro ty první. Tedy Tagset::is\_pronoun($x) bude 1, jestliže $x->is\_pronoun/pronominal() && $x->get\_pos() eq ‘noun’.
  + Obdobně, pokud známe konkrétní jazyk a sadu značek a pokud si to uživatel přeje, můžeme mu některé hodnoty převést na disjunkce možných hodnot. Např. víme, že v rumunštině lze rozlišit N druhů zájmen (podmnožina známých hodnot rysu prontype), a zároveň víme, že daná sada značek druh zájmena vůbec nezaznamenává, můžeme tedy uživateli alespoň u každého zájmena vyjmenovat druhy, které přicházejí v úvahu. Tedy uměle vyrobit přesně to, co jsem výše napsal, že nechceme reprezentovat disjunkcemi přímo v objektu FeatureStructure.

Poznámky:

* Ne vždy máme seznam povolených tagů k dispozici. Ve skutečnosti jsem ho měl snad jen u pražských značek, u všech ostatních sad jsem prostě pracoval se seznamem značek, které jsem našel v korpusu. To je samozřejmě nedostatek, ale jeho praktické dopady nejsou zásadní, protože list() se používá na pomocné operace a základní služby Intersetu mohou být poskytovány i bez něj.
  + I kdybychom si byli jisti, že máme seznam povolených značek úplný, dekodér by měl být tolerantní a schopen načíst i nepovolené značky, které se přečíst dají (např. máme na vstupu poziční značku, na každé pozici je povolený znak (hodnota rysu), akorát daná kombinace hodnot povolená není).
* Seznam povolených značek se používá při testování (ladění) Tagsetu.
  + Základní test: Pro každou povolenou značku $x (je na seznamu vráceném metodou list()) by mělo platit, že encode(decode($x)) eq $x.
  + Pro každou povolenou značku $x by mělo platit, že když po jejím dekódování vyprázdníme hodnotu rysu other a výsledek opět zakódujeme, dostaneme povolenou značku.
* Seznam povolených značek se také používá pro usměrnění funkce encode() při takzvaném *striktním* kódování. Např. pražská sada značek nám principiálně umožňuje zapsat, že slovo je citoslovce v 7. pádě. Výsledný řetězec není povolená značka podle seznamu pražských značek, ale přesto tuto informaci dokážeme přečíst a porozumět jí. A třeba existuje jazyk, ve kterém citoslovce rozlišují pády. *Laxní* kódování zapíše do značky vše, na co má daná sada aparát, abychom ztratili co nejméně informace uložené ve struktuře rysů (FeatureStructure). Tohle jsem se např. snažil dělat při zobrazování pražských značek pro cizí jazyky v HamleDTu. Naproti tomu *striktní* kódování se pokusí část informace zahodit, aby výsledek byl platná pražská značka (např. proto, aby nepadaly nástroje, které na novotvary nejsou připravené). V tomto případě to nejspíš proběhne tak, že jako první vezme informaci o slovním druhu, pak zjistí, že pražská citoslovce nemají pád ani další rysy, a zbytek hodnot tudíž zahodí. Striktnímu kódování se tedy budeme chtít raději vyhnout, jestliže víme, že náš seznam povolených značek má povážlivé mezery.
  + Kvůli striktnímu kódování umí každý Tagset vyrobit ze svého seznamu povolených značek strukturu, která popisuje povolené kombinace hodnot rysů. Je to něco jako trie na hodnotách rysů: z hodnot rysů 0 až i vyplývají omezení na hodnoty rysů i+1 až n. Z toho plyne, že pro tento účel existuje uspořádání rysů Intersetu (ne nutně shodné s pořadím, ve kterém chceme rysy zobrazovat uživateli), které definuje jejich prioritu. Interset zatím funguje tak, že dotyčné uspořádání je jednotné a definované centrálně pro všechny sady značek (tj. mohla by to být statická metoda FeatureStructure). Předpokládal jsem, že pro některé sady budu chtít toto výchozí pořadí nějak upravit, ale zatím jsem to nevyužil.
  + Ke konci současného modulu tagset::common mám taky pár ladících funkcí, které umí něco říct o povolených kombinacích hodnot rysů, ukázat příklad fyzické značky, která k nim vedla apod. Z těch se zřejmě také stanou metody třídy Tagset.
* Jedním z povinných testů pro nový Tagset je, že překódujeme všechny povolené značky ze všech ostatních sad do něj (a také všechny jeho povolené značky do všech ostatních sad) a při použití striktního kódování musíme vždy zase dostat povolené značky v daných sadách.
  + Poznámka: Tohle není úplně triviální úloha v případech, kdy povolenost některých kombinací hodnot má vazbu na hodnotu rysu other.
* Pro masivní konverze značek ve velkých korpusech se hodí cachování. Typická sada obsahuje řádově stovky, nanejvýš tisíce různých značek. A mělo by platit, že pro tutéž fyzickou značku nám decode() vrátí vždy stejnou FeatureStructure se stejnými hodnotami rysů. Takže Tagset si může pro každou dekódovanou značku nahashovat odkaz na výslednou FeatureStructure a příště ho rovnou vrátit. Má to ale háček: musel by vrátit kopii nahashovaného objektu, ne pouhý odkaz na něj. Protože mu nikdo nezaručí, že volající hodnoty objektu, který dostane, nějak nezmění.
  + Poznámka: S ohledem na nutnost duplikování objektů je možná jednodušší nechat cachování na uživateli. Pokud uživatel opravdu používá Interset ke konverzi mezi dvěma fyzickými sadami značek, může si do hashe ukládat rovnou cílovou fyzickou značku, čímž ušetří mnohem víc. Případně bychom na to mohli vyrobit novou třídu Converter, která bude vědět o dvou Tagsetech a bude schopná konvertovat (včetně cachování) mezi nimi.

## Další funkce

Obecná varianta funkcí decode(), encode() a list(), která dostane jako první parametr identifikátor tagsetu. Zdroják, kde tohle použiju, tedy neobsahuje use na konkrétní modul s Tagsetem, to se udělá až nějakým evalem uvnitř. Např.

$tag = decode(‘cs::pdt’, $znacka);

by mělo dělat totéž jako

use Interset::CS::Pdt;  
my $tagset = new Interset::CS::Pdt;  
$tag = $tagset->decode($znacka);

Zatím nemám zcela jasno, ve kterém modulu by tyhle obecné funkce měly být a zda by to případně měly být metody nějaké třídy. Možná by to mohly být metody FeatureStructure:

$tag->decode(‘cs::pdt’, $znacka);

Kromě toho mám (hlavně pro ladící účely) obecnou funkci find\_drivers(), která vyjmenuje všechny známé Tagsety. Dělá to tak, že projde cesty v @INC a hledá v nich složky tagset. Teď by tedy hledala složky Interset (a nejspíš by se jmenovala find\_tagsets()).