Programátorská dokumentace k projektu Morfo

David Kolovratník a Leoš Přikryl

27. května 2008

Obsah

Kapitola 1

Všeobecné informace

1.1 Doporučení čtenáři

Programátorská část dokumentace byla psána s předpokladem, že čtenář je seznámen s projektem Morfo jako uživatel. Před jejím studiem doporučujeme k přečtení XrefId[?uživatelskou dokumentaci?].

1.2 Terminologie

V dokumentaci se objevují slova a slovní spojení ve specifickém významu - termíny. Pro správné pochopení definujeme následující pro prográmátorskou část (viz též XrefId[?terminologii uživatelské části?]).

struktura pro lemma, LS Struktura pro lemma (LS od Lemma Structure) reprezentuje v programu XrefId[?slovníkové heslo?], tedy veškeré informace k daménu lemmatu, se kterými se v programech pracuje. Více viz XrefId[?struktura slovníku?].

1.3 NaturalDocs

K projektu byla vygenerována automatická dokumentace z komentářů pomocí programu NaturalDocs <`http://www.naturaldocs.org`>. Tuto dokumentaci naleznete na instalačním CD. Prohlédnout si ji můžete otevřením souboru `doc/NaturalDocs/index.html` v libovolném webovém prohlížeči. V dalším textu se na tuto dokumentaci budeme odvolávat pojmem *dokumentace NaturalDocs*.

Kapitola 2

Vstup/Výstup

2.1 Co řeší modul V/V

Modul V/V připravuje informace uložené v souborech ke zpracování do paměti. Ze zvolené perzistentní reprezentace konstruuje reprezentaci vhodnou k výpočtu. Pro data, která se mění, zajišťuje možnost odpovídající úpravy původního zdroje.

Vstupy a výstupy jsou do modulu V/V vyčleněny v případě, že se jedná o operace s daty uživatele. Motivací bylo oddělit zpracování dat od podoby jejich perzistentní reprezentace. Naopak, pomocné souborové operace si provádějí moduly ve vlastní režii. Mají nad nimi plnou kontrolu a mohou si je maximálně přizpůsobit svým potřebám.

Modul V/V obstarává vstup těchto typů zdrojů:

- seznam pozičních značek
- seznam derivačních vzorů
- seznam ohýbacích vzorů
- morfologický slovník
- konfigurační soubor

Modul V/V obstarává aktualizaci těchto typů zdrojů:

- morfologický slovník
- index morfologického slovníku

2.2 Indexace morfologického slovníku

Morfologický slovník je, podle požadavků zadavatele, nehomogenní textový soubor. Z toho a potřeby náhodného přístupu k záznamům vyplývá potřeba zmapování souboru. Textový soubor s morfologickým slovníkem bude dále označován *primární soubor*, jeho mapa *index*.

Index, jako doprovodný soubor primárního souboru, obsahuje redundantí informace - lze jej z primárního souboru odvodit. Explicitní uložení mapy umožní efektivně implementovat zejména tyto operace:

- zjišťovat přítomnost lemmatu v primárním souboru,
- sestavit celkový obraz informací z primárního souboru o daném lemmatu,
- procházet lemmata v primárním souboru v pořadí určeném nějakým (předem zvoleným) lineárním uspořádáním,
- přidávat záznamy (lemmata),
- označovat záznamy (lemmata) jako smazané,

- zneplatňovat záznamy (lemmata) - viz XrefId[?undo?].

2.2.1 Implementace

Index je řešen obecně známou datovou strukturou *B strom* (dále jen strom). Do stromu se ukládají záznamy pevné velikosti. Záznam je dvojice odkazů. Jeden odkaz vede na klíč (lemma), druhý na informace klíči přidružené. Lemmata jsou uložena v prostoru pro řetězce, přidružené informace v prostoru pro čísla. Oba prostory jsou nedílnou součástí indexu.

Ke klíči jsou v prostoru pro čísla přidruženy tyto informace:

- verze lemmatu,
- počet záznamů v primárním souboru, do nichž je lemma rozloženo (jedná-li se o záznam typu XrefId[?undo?], je uloženo opačné číslo),
- pozice záznamů v primárním souboru, do nichž je lemma rozloženo.

2.2.2 Postup

Indexace probíhá ve třech fázích. V první probíhá vlastní mapování primárního souboru. Vytváří se seznam lemmat a staví se strom. Primární soubor se prochází sekvenčně od začátku do konce po jednotlivých XrefId[?entries?]. Lemmata ze seznamu `lslem` jsou zavedena do stromu s pozicí a verzí zpracovávané entry v souboru. Je-li lemma už ve stromě přítomno, přidá se jen entry do seznamu pozic. Seznamy pozic jsou realizovány spojovými seznamy v poli (subalokátor). Index hlavy seznamu pozic v poli je ve stromě uložen v buňce určené pro odkaz na přidružené informace v prostoru pro čísla. Konec seznamu je značen odkazem na -1. buňku.

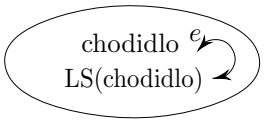
Výsledkem první fáze je struktura, která

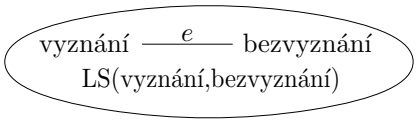
- ví, kde jsou v primárním souboru všechna entries obsahující dané lemma,
- umí v logaritmickém čase lemma vyhledat.

Ve druhé fázi se z entries (bez přístupu k primárnímu souboru) vyberou XrefId[?platné verze?] a spojové seznamy se převedou do úspornější reprezentace polem. Pole se vytvářejí v novém subalokátoru, subalokátor seznamů je uvolněn. Záznamy typu *undo* jsou opět značeny záporným počtem položek.

Ve třetí fázi se ověřuje XrefId[?struktura slovníku?] a zúplňují se seznamy pozic entries potřebných ke stavbě struktury pro lemma. Index se prochází po lemmatech. Cílem zúplnění je zajistit, aby index znal ke každému lemmatu pozice všech záznamů, které jsou třeba pro stavbu LS.

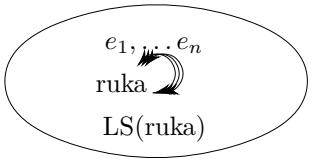
- Většina struktur pro lemma vychází z jediného záznamu. Na takových není co ověřovat, stupeň vrcholu je nula nebo jedna.



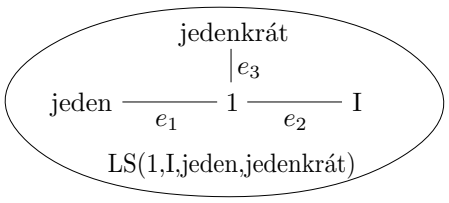


- Patří-li k lemmatu více záznamů, je nutno situaci prozkoumat, načíst z primárního souboru celou strukturu.

a)



b)



  - Je-li ve struktuře nejvýše jeden vrchol (jediné lemma; příklad a)), není co řešit. Všechny záznamy jsou uloženy v indexu na jednom místě.

- Je-li lemmat více (příklad b)), je ověřované lemma kandidátem na ústřední vrchol hvězdy. Stačí ověřit, že pro všechny jeho sousedy (různá lemmat) platí: množina hran vycházejících ze souseda je podmnožinou hran incidentních s ústředním vrcholem. Podle výsledku testu inkluze se rozlišují případy.
  * Jde-li o vlastní podmnožinu, budou sousedovi doplněny chybějící hrany.
  * Jde-li o nevlastní podmnožinu, není třeba nic dělat.
  * Neplatí-li inkluze, ale platí-li opačná inkluze, je za ústřední vrchol považován testovaný soused. Zpracování se přeruší s tím, že se provede, až přijde řada na souseda.
  * Jsou-li množiny inkluzí neporovnatelné, je hlášena chyba. Zpracování pokračuje.

2.3 Parser

Parser slouží ke zpracování zdrojů, pro jejichž reprezentaci byla zvolena forma blízká XML (nejdůležitější odchylky jsou popsány v XrefId[?uživatelské dokumentaci?]).

2.4 Konfigurační soubor

Zpracování konfiguračních souborů z hlediska uživatele je popsáno v XrefId[?uživatelské dokumentaci?]. Zde bude popsáno, jak definovat další parametry.

Zpracování konfiguračního souboru je řešeno modulem profile_read. Požadované chování implementuje funkce `morfo_config_read_file_options_do`. Získané informace se ukládají ve struktuře typu struct TMorfoConfiguration. Parametry se definují makrem CONFIGURATION_OPTION(TYPE, NAME). To ke každému parametru NAME definuje položku *`NAME##_defined`* typu bool_t sloužící k určení, zda je parametr definován.

Přístup k parametrům je doporučeno obalit funkcí, která bere ohled na definovanost parametru, implementuje chování popsané v uživatelské dokumentaci a případně řeší situaci, kdy je hodnota vyžadována, ale není k dispozici (například vyvoláním výjimky). Viz funkce `morfo_config_get_primary_file_path`, `morfo_config_get_primary_file_path_re` a další.

2.4.1 Definice reakce parseru

Lze využít automatické reakce nebo si definovat vlastní obsluhu. Parser ze souboru načte dvojici klíč, hodnota. Nejprve se hledá klíč v poli `morfo_file_options` automaticky zpracovatelných parametrů. Není-li nalezen, provede se manuální zpracování ve funkci `morfo_config_read_file_options`.

Většinu parametrů lze zpracovat automaticky. To znamená, že podle popisu v poli `morfo_file_options` bude vyvolána odpovídající funkce ke zpracování. Položky pole jsou struktury typu struct TAutoFileOption, vyplňují se makrem OPTION(NAME, FIELD, STORE_F).

PARAMETRY MAKRA OPTION

NAME jméno parametru,

FIELD položka struktury struct TMorfoConfiguration, do níž se má uložit hodnota parametru,

STORE_F funkce sloužící k uložení nalezeného páru klíče a hodnoty v konfiguračním souboru do struct TMorfoConfiguration. Jednoduché příklady představují funkce `option_string` a `option_int`. Zpracuje-li funkce bezchybně hodnotu parametru, vrací TRUE, jinak FALSE (na `stderr` je vytištěno varování, zpracování pokračuje).

2.4.2 Shrnutí úkonů spojených se zavedením parametru

- definice položky ve struktuře struct TMorfoConfiguration,
- stručná dokumentace v komentáři,
- definice přístupové funkce,
- zajištění citlivosti parseru (obvykle přidáním záznamu do pole `morfo_file_options` v souboru `io/profile_read.c`),
- využití hodnoty v programu,

2.5 Využití mimo projekt Morfo

S modulem pro vstup a výstup byly vyvinuty podpůrné funkce potřebné pro jeho činnost. Dohromady tvoří jádro projektu, nad nímž je postavena další funkcionalita. Jádro je dostatečné obecné, mohlo by posloužit dalším programům jako základ pro práci se slovníkem.

V této kapitole uvedu základní informace potřebné pro využití jádra, názvy funkcí a konkrétní postupy.

2.5.1 Obecné informace

2.5.1.1 Ukazatele

Objekty vracené prostřednictvým `const` ukazatelů jsou určeny pouze pro čtení, nesmějí být ani uvolňovány. Některé (destruktivní) operace se strukturami, na nichž adresa objektu závisí mohou objekt přemístit a tím ukazatel zneplatnit. Na to si je třeba dávat dobrý pozor (ačkoli my jsme s tím problémy prakticky neměli).

Ve většině případů funkce s parametry typu ukazatel nevyžadují, aby referencovaný objekt existoval po návratu z funkce. Je-li ukazatel ukládán, upozorním na to.

Nebyl-li dobrý důvod pro porušení zásady, plátí, co jádra jest, to nebrat ani nemodifikovat. Tak se chová také jádro vůči uživateli.

2.5.1.2 Subalokátor

Jádro si pamatuje mnoho malých objektů (typicky řetězce, čísla). Kvůli snížení režije se používají subalokátory. Alokuje se větší oblast paměti. Do ní se malé objekty ukládájí. Často jejich platnost závisí na existenci nějakého jiného objektu. Dokud existuje, neuvolňují se. Až zanikne, uvolní se všechny najednou zrušením subalokátoru.

To je důvod možných změn adres vracených objektů. Adresy se nemění samovolně, ale v návaznosti na nějakou (destruktivní) operaci. Budete-li jen číst, můžete si bez obav ukazatele uschovat.

K identifikaci objektu nezávislé na adrese subalokátoru (typ struct TGrowing) se používá přirozené číslo (typ size_t). Toto číslo je platné v rámci svého subalokátoru. Například LS (typ struct TLemma) používá subalokátor pro ukládání řetězců (string repository) a čísel typu int (int repository).

Subalokátor struct TGrowing je naprogramován v souborech `common/data_structures.[ch]`. Funkce jsou prefixovány řetězcem `growing_stack_`.

2.5.1.3 Přidělování paměti

Snažím se vyhýbat nadužívání dynamické alokace paměti. Používá se subalokátor. Používá se automatická paměť (buňky paměti jsou přiděleny na zásobníku). Prakticky se to projeví definicí proměnné přímo požadovaného typu (`struct TLemma lemma` místo ukazatele `struct TLemma *lemma`) a předávání ukazatele získaného referenčním operátorem `&`.

Inicializační funkce a funkce pro úklid jsou dvojího typu. Jedny (init, dispose) pracují pouze s „vnitřkem“ struktury, nestarají se o paměť struktury samotné. Inicializační funkce vyžaduje ukazatel na paměť, kde sídlí struktura, které má být inicializována. Druhé (new=create, free) obhospodařují paměť i pro objekt, se kterým pracují. Funkce create (new) nepotřebují ukazatel na inicializovaný objekt. Paměť samy přidělí.

Obvykle ke každému objektu existuje pouze jeden pár funkcí. Volba byla dána potřebami programu. Překvapivě jsem si převážně vystačil s první variantou. Objekty existují po dobu dobře vymezenou zásobníkem volání funkcí nebo jsou součástí většího objektu, kde mají pevně přidělené místo.

2.5.1.4 Obsluha nestandardních situací

Při běhu programu může dojít k chybám různého charakteru. Mohou selhat systémová volání, může být syntaktická chyba ve vstupních datech, referencován neexistující vzor a pod. Chyba vzniká daleko od místa, kde může být rozhodnuto o dalším osudu procesu. Bylo by úmorné v každé funkci kontrolovat, zda v podprogramu nenastala chyba.

Používáme přijatelné, ačkoli ne dokonalé, řešení (nelze snadno realizovat zotavení) podobné systému výjimek vyšších jazyků, implementované pomocí techniky vzdáleného skoku.

Na začátku běhu programu je inicializován zásobník reakcí (ošetření) výjimek. Do zásobníku se ukládá návěští (`setjmp` (c99)) na kód ošetřující výjimku. V místě vzniku výjimky dochází ke skoku (`longjmp` (c99)) na návěští uložené na vrcholu zásobníku. Systém se umí vyrovnat i s prázdným zásobníkem. Výjimka je ohlášena a program končí.

Vznik výjimky ani následující skok návěští z vrcholu zásobníku nevyzvednou. Důsledkem je, že dojde-li k výjimce v průběhu obsluhy výjimky, je obsloužena již prováděným obslužným blokem. Tam s největší pravděpodobností vznikne tatéž výjimka znovu, dojde k zacyklení.

S výjimkou se šíří informace o jejím typu, funkci, kde vznikla a akci, která selhala. Funkce (raising function) se zadává při vyvolání výjimky jako parametr typu enum TRaisingFunction, neúspěšná akce (failed action) jako parametr typu enum TFailedAction. K akci lze definovat popis v poli `ERR_failed_action`.

Systém výjimek je implementován v souborech `common/err.[ch]` a `common/err_def.[ch]`.

2.5.1.5 Testovací programy

Většina unit (.c .h) obsahuje nějaký testovací prográmek. Někdy demonstruje funkcionalitu unity a může posloužit jako dobrý příklad. Jindy se zvrhnul při hledání chyb ve zmatek a posloužit nemůže.

2.5.2 Podpora slovníku

2.5.2.1 Sdílení prostředků

Unity pro práci se seznamy tagů, ohýbacích a derivačních vzorů obsahují mechanismus pro snadné sdílení otevřených prostředků. Podpora je ve dvou bodech.

- Unita si vede seznam otevřených seznamů podle názvu souboru. Při požadavku na načtení seznamu ze souboru projde seznam otevřených souborů. Najde-li soubor jako otevřený, vrátí již přichystaný seznam.
- Pro každý otevřený seznam se počítá počet referencí. Seznam je uvolněn, klesne-li počet referencí na nulu.

Podpora sdílení se v projektu využívá minimálně a to jen vzájemně mezi seznamy. Typecké použití nekomplikuje a nijak nepřekáží.

2.5.2.2 Podpora různých formátů

Projekt měl být snadno přizpůsobitelný změnám formátů pro perzistentní uložení dat. Flexibilita měla být řešena polymorfizmem. Načítání seznamů tagů, ohýbacích a derivačních vzorů i morfologického slovníku je rozděleno na část přímo závislou na formátu souborů - parser (například soubory `read_pattern.[ch]`) a na část zajišťující další zpracování a rozhraní (například soubory `patterns.[ch]`).

Některé funkce mají deklarovány parametry (jako například `const struct TPatternReaderMT *pattern_mt`), kterými se měla předávat tabulka metod realizujících formátově závislé operace. Mechanismus nakonec využit nebyl, s hodnotou parametru se nikde nepracuje, ale v hlavičkách funkcí zůstal.

2.5.2.3 Seznam tagů

Unita řeší práci s tagy. Tagy jsou vnitřně reprezentovány atomem (přirozené číslo typu int). K dispozici jsou funkce na převod mezi atomem a poziční i krátkou značkou. Funkce existují ve dvou variantách. Jedny signalizují chybu(neexistující atom nebo značku) návratovou hodnotou, druhé (přípona `_try`) výjimkou.

Implementace je v souborech `io/tags.[ch]`. Funkcionalita je velmi jednoduchá. Doporučuji k nahlédnutí testovací program unity.

Očekává se, že seznam tagů je pevně dán a po načtení se nebude měnit. Lze si tedy uschovat ukazatele na řetězcovou reprezentaci značek vracenou funkcemi unity.

2.5.2.4 Ohýbací vzory

Pro práci s ohýbacími vzory slouží unita implementovaná v souborech `io/patterns.[ch]`. Definuje se struktura struct TPattern pro reprezentaci vzoru v paměti.

K dispozici jsou funkce pro nalezení vzoru podle jména (`pattern_by_name`) a pro práci se vzorem (zjištění jména, počtu forem, ...). Doporučuji k nahlédnutí testovací program unity a funkci `pattern_print`.

Očekává se, že seznam vzorů je pevně dán a po načtení se nebude měnit. Lze si tedy uschovat ukazatele na vzory i jejich složky vracené funkcemi unity.

Ohýbací vzory obsahují tagy. Proto funkce `pattern_load_from_file` sloužící k načtení seznamu vzorů ze souboru vyžaduje zadání seznamu tagů. Ukazatel na seznam tagů se ukládá k seznamu vzorů. Musí existovat po celou dobu existence seznamu vzorů.

2.5.2.5 Derivační vzory

Derivační vzory jsou řešeny v témže duchu jako ohýbací vzory v souborech `derivations.[ch]`. Derivační vzory se odkazují na ohýbací vzory. Funkce `derivation_file_create_from_file` pro načtení jejich seznamu vyžaduje ukazatel na seznam ohýbacích vzorů. Ten si ukládá a předpokládá jeho existenci po dobu existence seznamu derivačních vzorů.

Doporučuji k nahlédnutí testovací program unity a funkci `derivation_print`.

2.5.2.6 Sdružená práce se seznamy

Protože seznamy tagů, ohýbacích a derivačních vzorů jsou chápány jako nutné podpůrné zdroje pro práci se slovníkem a očekává se tedy, že budou použity takřka v každém programu, jsou k dispozici funkce pro otevření a uvolnění všech tří zdrojů najednou.

Jedná se o dvě funkce s prefixem `tpd_` (Tags, Patterns, Derivations). Pro otevření zdrojů slouží `tpd_resource_files` pro uvolnění `tpd_resource_files_free`.

2.5.3 Slovník

Se slovníkem lze pracovat na třech rovinách.

- Jednotlivé záznamy - entries (žádný index),
- LS (index),
- LS - více souborů, více indexů.

Roviny se na sebe cibulovitě kupí. K realizaci vyšší roviny jsou využity nižší.

2.5.3.1 Rovina entries

Rovina nabízí první abstrakci nad perzistentními daty. Zavádí se typ struct TIODictionaryWordFull pro reprezentaci entry v paměti.

K dispozici je pouze sekvenční načítání entries. Ke zpracování budou předány všechny načtené entries. Neprovádí se žádné filtrování, přerovnávání ani seskupování. Editor generuje nové verze záznamů a záznamy typu „undo“ a „delete“. Je třeba počítat s entries různých verzí i technickými entries. Technické entries lze snadno XrefId[?poznat?] a odfiltrovat (za cenu ztráty změn - například výmazu položky). Verzováním zneplatněné položky odfiltrovat nelze, protože není jasné, jaká je poslední verze lemmatu (ani zda nebylo smazáno).

Seznamy. S příznaky (stylové, syntaktické, sémantické) a seznamy lemmat, komentářů a derivačních odvození (Nevím, proč je derivační odvození ukládáno jako seznam. Se seznamem jsme nepracovali. Až když jsem psal export do původního formátu, zjistil jsem (pouze přibližně), co jsou jeho prvky.) se pracuje jednotně. Při konstrukci seznamů se využívá toho, že prvky seznamu jsou uloženy v perzistentní podobě společně (prvky seznamu jsou uzavřeny v jednom elementu).

Seznam je řešen pomocí int-repository, string-repository (společné celé struktuře) a dvou čísel. Jedno číslo je index do int-repository, druhé říká, kolik prvků seznam má. Tím je v int-repository vymezen interval. Čísla z vymezeného intervalu jsou indexy do string-repository na nulou ukončené řetězce.

Unita je implementována v souborech `read_word.[ch]`. Doporučuji k nahlédnutí funkci `io_dictionary_word_print` a testovací program unity. Prostředky (seznamy tagů, ohýbacích a derivačních vzorů a struktura struct TIODictionaryWordFull (pro uložení načtené entry)) jsou sdruženy do struktury struct TDictionaryWordReadContext. Struktura struct TIODictionaryWordFull je určena k recyklaci. Očekává se, že bude opakovaně používána k ukládání načtené entry. Před načtením je resetována (není třeba se o to starat).

2.5.3.2 LS, index

Mapování (indexace) slovníku umožňuje realizovat pokročilejší přístup k datům. Zásadní je zavedení LS (soubory `lemma_structure.[ch]`). Lemmata jsou procházena podle uspořádání (používá se porovnávací funkce deklarovaná v `common/utf_sort.h`). Jsou odfiltrovány staré verze lemmat. Lze odfiltrovat technické entries. Je k dispozici průchod podle uspořádání nebo vyhledání a případná konstrukce LS podle lemmatu.

Jedna LS je v indexu tolikrát, kolik různých lemmat obsahuje. Tolikrát také bude při průchodu nabídnuta (je realizován průchod indexem, ne slovníkem). Průchod podle uspořádání je k dispozici ve dvou variantách.

- Jednodušší varianta realizuje průchod s callbackem. Řízení má funkce indexu. Na každý záznam volá uživatelský callback. Viz funkce `btree_for_each_key` unity `common/btree.[ch]`.
- Druhá varianta nechává řízení v rukou uživatele. Pozici v indexu reprezentuje kurzor (struct TBTreeSearchContext). Kurzor může být vytvořen na začátku, na konci, před nebo za daným lemmatem. Pohyb je možný o položku dopředu (`wi_next`) nebo dozadu (`wi_prev`). Konec je signalizován návratovou hodnotou NULL. Průchod lze kdykoli ukončit a kurzor uvolnit.

  Existuje-li nějaký kurzor, nelze index upravovat.

Index je realizován v souborech `io/word_index2.[ch]`. Doporučuji k nahlédnutí funkci `ls_lemma_print` (unita lemma_structure) a testovací program unity lemma_structure.

Otevření slovníku. Slovník se otvírá spolu s indexem. K tomu slouží funkce prefixované `ls_lemma_context_init`. Nejvyspělejší je funkce `ls_lemma_context_init4_brief`. Očekává cesty ke slovníku a indexu. Sama se rozhodne (podle systémováho času souborů a existence indexu), zdaje index třeba stavět (je stavěn do paměti, soubor na disku se nemění) nebo zda lze využít index ze souboru. Zadáním hodnoty NULL jako cesty k indexu lze naznačit, že index v souboru není k dispozici (bude postaven). Ostatní funkce nechávají řízení více v rukou volajícího. Lemma context se zavírá funkcí `ls_lemma_context_dispose`.

2.5.3.3 LS

LS reprezentuje v paměti XrefId[?slovníkové heslo?]. Jde o informace složené z několika entries. Všem entries je společná verze. Jsou zachovány všechny složky entry. Původní entries lze rekonstruovat. K tomu účelu jsou zavedeny záznamy typu struct TPerEntry ukládané v `per_entry` subalokátoru. Z per enries vedou odkazy na blok atributů (seznamy příznaků, lemmat, komentářů a derivačních odkazů), a na `writing` nebo `form` (viz dále). Jedna z položek writing a form je nastavena na `-1`, druhá ukazuje do pole.

Entries se při začleňování do LS třídí do dvou skupin.

- Form - entry s konkrétní formou (tvarem) lemmatu a tagy vymezujícími morfologickou platnost tvaru,
- writing - entry přiřazující lemmatu kofix a derivační vzor.

Pro zařazení do skupiny je rozhodující, zda má entry neprázdný seznam tagů. Prázdný seznam entry zařadí do skupiny writing, neprázdný do form. Seznam tagů se smí objevit jen v doprovodu některých vzorů (viz XrefId[?omezení kladená na slovník?]). Není-li toto pravidlo splněno, volá se metoda `log_lstag_with_in` struktury struct TLemmaContext. Seznam je ignorován, entry je zařazena do skupiny writing.

Pro práci s LS existuje mnoho přístupových funkcí. Typicky vracejí požadovaný objekt (například skupinu atributů), ne zástupnou hodnotu (například index skupiny atributů). Se zástupnými hodnotami lze pracovat přímo. Hodí se zejména funkce `ls_string_rep_index` a `ls_int_rep_index` pro snadnou indexaci ve string-rep a int-rep. Mnoho příkladů lze nalést v `inline` přístupových funkcích implementovaných v hlavičkovém souboru i ve funkcích v .c souboru.

Struktura struct TLemma reprezentující LS musí být před použitím inicializována funkcí `ls_lemma_init` resp. `ls_lemma_new`. Alokované prostředky lze uvolnit voláním `ls_lemma_dispose` resp. `ls_lemma_free`. Vnitřní stav resetuje funkce `ls_lemma_reset`.

2.5.3.4 Konstrukce LS

Pro editor pro použití z prostředí Perlu bylo realicováno prostředí pro konstrukci LS. Počítá se s konstrukcí od začátku, modifikace existující struktury prostředí nepodporuje. Načítaná LS se konstruuje přímo, bez využití tohoto prostředí (vzniklo později).

Konstrukce je provázena kontextem struct TLemmaBuildContext. Zakládá jej funkce `ls_bc_new` (uvolňuje `ls_bc_free`). Sdružuje informace potřebné pro konstrukci, aby nemusely být každé funkci předávány jako argumenty.

K vlastní konstrukci slouží řada funkcí (typicky s prefixem `ls_bc_`). V pořadí volání funkcí je jisté volnost, není však zcela libovolné. Jednoznačné je omezení vyplývající z argumentů vyžadovaných funkcemi (když funkce `a` vyžaduje výsledek funkce `b`, je jasné, že `b` musí být volána před `a`).

Explicitně je nutno formulovat omezení na zadávání příznaků. Vyžaduje se, aby zadávání příznaků jednoho typu nebylo proloženo zadáváním příznaků jiného typu. Nelze tedy (například) zadat dva syntaktické příznaky, pak derivační odkaz a pak další syntaktické příznaky. Toho je nutné dbát při použití funkcí se sufixem `_add` (například `ls_bc_lemma_add`). Variantou je použití funkcí s prefixem `ls_bc_set_` (například `ls_bc_set_lemmas`). Ty nastavují všechny příznaky daného typu najednou. Ukázaly se však jako nevhodné pro využití z Perlu.

MOŽNÝ SCÉNÁŘ KONSTRUKCE LS

- Vytvoření kontextu konstrukce funkcí `ls_bc_new`.
- OPAKOVÁNÍ ČINNOSTÍ
  - Nastavení jednoduchých položek `ls_lemma_set_version`, `ls_lemma_set_lemma`, `ls_set_global_status` nebo
  - Konstrukce writingu nebo formy
    1. Zadání atributů,
    2. ukončení zadávání atributů funkcí `ls_bc_commit_attributes`,
    3. zadání informací specifických pro form (funkce `ls_bc_form_push`) nebo writing (funkce `ls_bc_writing_push`),
    4. registrace zadaných informací funkcí `ls_bc_entry_register`.
- Výpočet modusu atributů funkcí `ls_compute_attributes_modus`,
- uvolnění kontextu konstrukce funkcí `ls_bc_free`.

Po dobu existence kontextu konstrukce lze volat funkci `ls_bc_get_lemma`.

2.5.3.5 LS, další funkce

Ekvivalence LS. Je k dispozici sada funkcí (v čele `ls_is_equivalent`) pro testování ekvivalence morfologické hodnoty dvou LS. Při testování se neberou v úvahu organizační údaje (původní soubor, autor ani časová razítka (primární ani změn), verze) ani pořadí položek v seznamech (není relevantní).

Kontrola konzistence. LS je složitý objekt. Ke kontrole interních vazeb lze využít funkci `ls_validity_check`. Vrací kód prvního nalezeného selhání (problému). Funkce neprověřuje naprosto všechno, ale co se (rozumně) dalo.

2.5.3.6 Složený index

Složený index umožňuje sestavit jednotný pohled na slovník rozdělený do několika souborů. Dotazy na složený slovník budou vyřízeny, jako by se jednalo o jeden soubor. Objevila se ovšem otázka, jak budou řešeny konflikty.

Složený index (WIC - *Word Index Compound*) je realizován jako poklička nad prostředky (jednoduchého) indexu (WI - *Word Index*) v souborech `io/wi_comp.[ch]`. Sám si neotvírá slovníky ani indexy. Ty se do něj jen zastrčí případně vyndají.

Prázdný WIC se založí voláním `wic_new`. Funkce vrací dynamicky alokovanou strukturu struct TWIC (uvolnění provede volání `wic_free`). Jednotlivé slovníky lze přidávat resp. vyndávat funkcí `wic_add_source` resp. `wic_pop`. Seznam slovníků je realizován zásobníkem. Tím by se měl omezit zmatek ve změnách indexů, který by vznikal vytahováním libovolných slovníků.

Unita WIC obsahuje mnoho jednoduchých funkcí. Typicky jde o přístupové funkce. Často se objevují podobné funkce lišící v nějakém detailu. Skutečná funkcionalita se točí okolo procházení složeného slovníku a vyhledávání v něm.

Průchod slovníkem. Průchod slovníkem je (podobně, jako u WI) řešen pomocí kurzoru. Kurzor reprezentuje pozici ve slovníku. Lze jej vytvořit na začátku, na konci, před nebo za lemmatem funkcemi s prefixem `wic_cursor_create_` (k uvolnění slouží funkce `wic_cursor_free`). Kurzor v sobě obsahuje kurzory indexů. Operace next/prev vybírá index, ze kterého bude vydána odpověď. Protože WIC si neeviduje existující kurzory, nemohou reagovat na přidání ani odebrání slovníku. Slovník přidaný po vytvoření kurzoru bude ignorován. Chování kurzorů po odebrání slovníku není definováno (manipulace s kurzoren velmi pravděpodobně způsobí výjimku nebo pád systému).

Míchání indexů. Míchání indexů řídí funkce `wic_cursor_move_` (další funkce spojující indexy: `wic_record_by_lemma_` a `wic_get_max_abs_version`). Z ní jsou parametrizací odvozeny další funkce, například `wic_cursor_next`

a `wic_cursor_prev`. Například při dopředném procházení se z dostupných indexů vybírá (v abecedním uspořádání) nejmenší lemma s nejvyšší verzí. Nižší verze téhož lemmatu se na výstup nikdy nedostanou.

Konflikty. Při míchání nezávisle indexovaných slovníků nastává konflikt v případě, že více indexů nabízí strukturu pro totéž lemma v téže verzi. Jsou-li záznamy v různých verzích, je chování jasně definováno - přednost má nejnovější verze. Řešit konflikt spojením více struktur do jedné by moc šťastné nebylo. Struktura v každém slovníku existovala samostatně. Sloučením dvou samostatných struktur (reálně možná v různých verzích, znamenajících něco jiného (formální verzování mohlo být provedeno v každém slovníku nezávisle)) by vedlo ke zmatku. Jiné řešení by bylo hodit si kostkou, jednu strukturu zahodit (stopit) a druhou vrátit. To evidentně také není ono. Postupně budou vráceny obě struktury.

Filtrování. Míchání indexů odstraňuje pouze struktury starších verzí, než nejnovější. Všechno ostatní předává ke zpracování. Podle využití je žádoucí odfiltrovat výsledky podle různých kritétií:

- pozice v primárním souboru
- typu záznamu (undo, delete, present).

Pro potřeby projektu jsou vypracování dva filtrační scénáře. Základní (na výstup jsou pouze záznamy typu present) je realizován funkcemi obsahujícími v názvu `_visible_`, revizní scénář funkcemi obsahujícími `_changes_`.

Režim revize. Režim revize byl vytvořen na míru použití v editoru. Zavádí se dodatečné podmínky na seznam slovníků, interpretují se různě (dosud na pořadí ani počtu slovníků nezáleženo, se všemi se zacházelo stejně). Zavádí se revizní scénář filtrování (`_changes_`).

Očekávají se tři slovníky (v uvedeném pořadí):

1. „velký“ editovaný slovník,
2. soubor změn,
3. pokračování souboru změn.

Rozdělení souboru změn na dvě části má technické důvody vyplývající z realizace parseru slovníku.

V revizním režimu míchací funkce nultý slovník nebere v úvahu (nerevidujeme jej). Prakticky vše z obou částí souboru změn se dostane do výstupu. Podle typu nalezeného záznamu se pozná akce smazání. Přidání nového záznamu se od editace odlišuje nahlédnutím do nultého indexu. Záznamy typu undo jsou odfiltrovány.

Přepínání režimů. Pro základní i revizní režim existují samostatné sady funkcí. Pro snadnější implementaci režimů editoru existuje třetí sada funkcí, jejichž chování lze přepínat. Funkce jsou označeny řetězcem `_dispatch` ve svých názvech.

Příklad 2.5.1 Využití systému výjimek v programu

```
1  /* otevření souboru - může dojít k~chybě */
2  int open_re(const char *pathname, int flags)
3  {
4    int fd = open(pathname, flags);
5    if (fd == -1)
6      raise_with_errno_and_string(rf_open_re, fa_open, strdup(pathname));
7    return fd;
8  }
9
10 char *read_file_re(const char *filename)
11 {
12   int fd = open_re(filename, O_RDONLY);
13   char *buff = NULL;
14   try(read)
15     off_t filesize = get_file_size_re(fd);
16     buff = malloc(filesize);
17     read_re(fd, buff, filesize);
18     close_re(fd);
19   otherwise
20     free(buff); free může být zavoláno na NULL
21   close(fd); close_re by mohlo způsobit výjimku a~zacyklení
22     reraise;
23   end;
24   close_re(fd);
25 }
26
27 int main(int argc, char *argv[])
28 {
29   ERR_exception_stack_init();
30   /* program využívající systém výjimek */
31   ...
32   ERR_exception_stack_dispose();
33 }
```

Místo chyby (řádek 6). Z místa vzniku výjimky lze šířit informace o okolnostech chyby. Informace jsou dostupné ve struktuře ERR_error typu struct TError. Řetězce jsou alokovány dynamicky. Vždy je definován typ výjimky (výčet enum TErrorType). Podle něj se pozná, jaké další informace jsou k dispozici. Typ výjimky se určuje výběrem makra pro její vyvolání (není závazné).

Začátek bloku ošetřených příkazů (řádek 14). K ošetření výjimek se používá trojice maker `try`, `otherwise` a `end`. Makro `try` označuje začátek bloku příkazů, jejichž selhání má být ošetřeno. Parametr makra se použije ke konstrukci identifikátoru návěští.

Blok obsluhy (řádek 19). Makro `otherwise` ukončuje blok ošetřených příkazů a zároveň otevítá blok obsluhy.

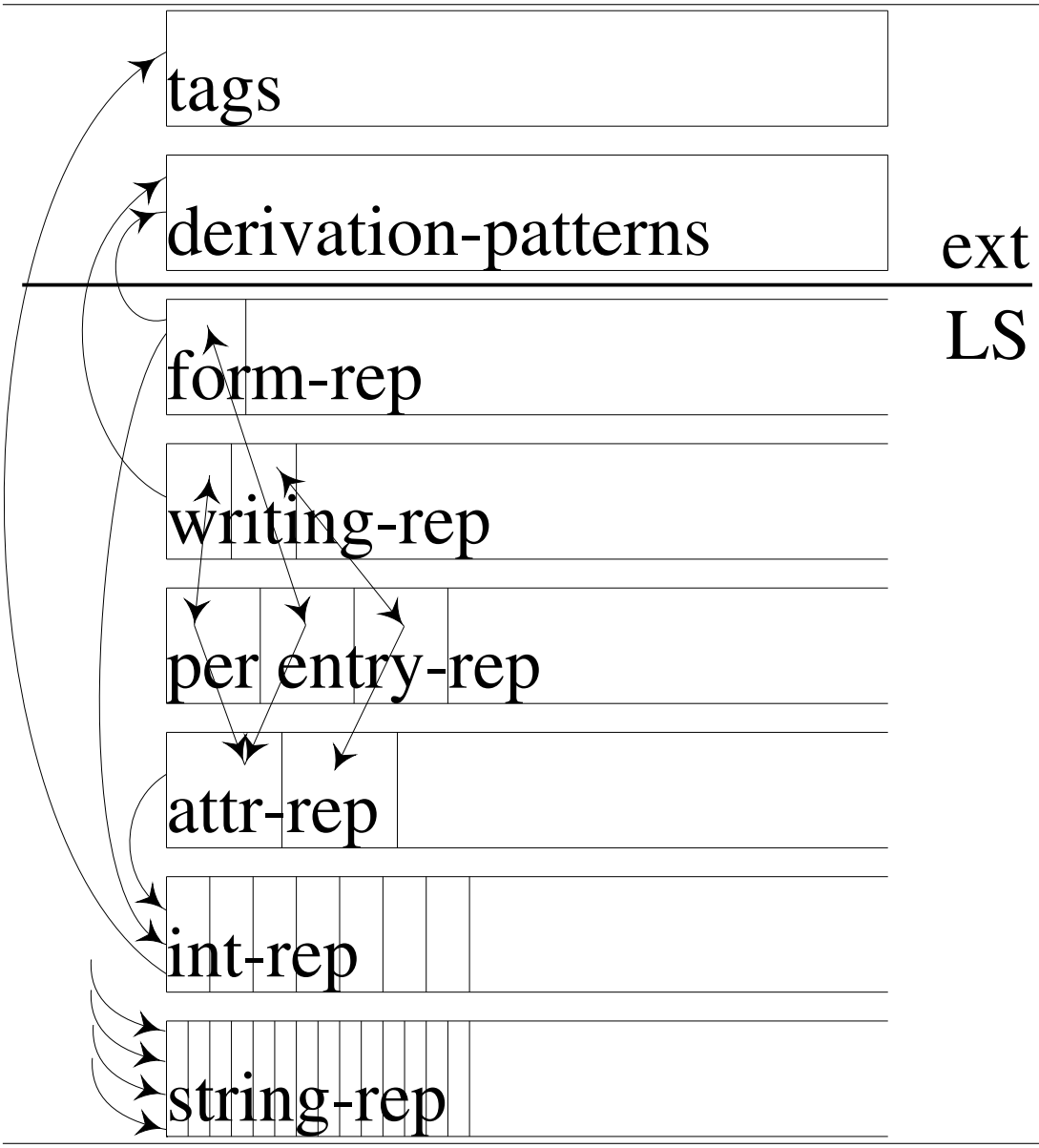
Propagace výjimky (řádek 22). Makrem `reraise` lze znovu vyvolat šíření výjimky na vrcholu zásobníku. Makro lze použít, neprojde-li běh programu makrem `end`. Příklad ukazuje využití zachycení výjimky k uvolnění alokovaných prostředků. Protože zde ještě není jasné, jak se zachovat (jak s chybou naložit), je výjimka šířena ke zpracování výš.

Konec bloku obsluhy (řádek 23). Makro `end` ukončuje blok obsluhy výjimky. Z vrcholu zásobníku je odstraněno návěští. Končí také jeho lexikální platnost. Běh programu pokračuje za makrem.

Inicializace systému výjimek (řádek 29). Před použitím musí být systém výjimek inicializovám voláním funkce `ERR_exception_stack_init`. Chování neincializovaného systému není definováno.

Uvolnění systému výjimek (řádek 32). Nebude-li systém výjimek dále využíván, mohou být jeho prostředky uvolněny voláním funkce `ERR_exception_stack_dispose`.

Příklad 2.5.2 Představa LS



Kapitola 3

Morfologická analýza

3.1 Řešená úloha

Morfologická analýza (dále jen m. a.) je aplikace morfologie, která si klade za cíl k zadanému slovnímu tvaru nalézt a vypsat všechna lemmata, jejichž tvarem je, včetně morfologické platnosti, kterou tvar zastává. Úloha je řešena s pomocí slovníku. Úplnost a správnost odpovědi je tedy třeba uvažovat vůči němu. V popisovaném pojetí se nepracuje se slovníkem přímo, ale s reprezentací z něj odvozenou. Odvozená reprezentace se specializuje na řešenou úlohu, je kompaktnější a neobsahuje zdrojová data v textové podobě.

Ve slovníku, tedy v prvotní podobě vstupních dat, je cesta od struktury pro lemma ke slovním tvarům rozdělena na derivační expanzi (slovotvorba) a ohýbací expanzi (tvarosloví).

Volně napsáno, chceme k mnoha možným slovním tvarům hledat tvar základní, lemma.

Schema vstupu. Pro pochopení následujícího textu doporučuji seznámit se se schematickým popisem struktury vstupních dat (viz Schema vstupu).

3.2 Myšlenka zvoleného řešení

Řešení se přirozeně rozpadá na dva základní celky. Jedním je vytvoření datové struktury, která umožní efektivně získat k zadanému lemmatu žádaný výstup, druhým pak užití struktury k zodpovídání dotazů. Druhá část je z velké části určena tou první, v níž jsou přijata zásadní rozhodnutí.

Protože hledáme inverzní funkci k expanzi tvaroslovného lemmatu (viz schema vstupu), je pro volbu řešení klíčovou otázkou uvědomit si vztah mezi tvaroslovným lemmatem, které uživatel očekává na výstupu, a tvary, které k němu přísluší. Dobrá analýza musí vzít v úvahu i slovotvorná schemata a hledat jejich efektivní začlenění mezi slovník a tvaroslovná schemata.

Zdroje slovních tvarů. Myšleným t. lemmatem odpovídáme pro všechny tvary, které s ním slovník spojuje. Tvary jsou zadány buď přímo příkazem `AddForm` nebo přes slovotvorné a následně tvaroslovné schema, tedy příkazem `ApplyRule`. O prvním případu nelze předpokládat nic. O to zajímavější (i ve srovnání s ohýbáním) je případ druhý. Oba příkazy se mohou objevit násobně a to v různých strukturách pro lemma (ty tvoří třídu ekvivalence na slovotvorném, nikoli tvaroslovném lemmatu)

Pozorným prozkoumáním funkcí `ApplyRule` a `ApplyInflRow` zjistíme, že slovní tvar zadaný příkazem `ApplyRule` vždy začíná (nezkráceným) t. základem (kofixem) a ten zase s. základem. Formulováno jinak, tvar vzniká prodloužením s. základu na t. základ a jeho dalším prodloužením na tvar samotný. Poskládání slovního tvaru ilustruje příklad Vznik slovního tvaru.

Způsob poskládání tvaru je zcela zásadní, umožňuje přímočaré jednoduché a přitom velice efektivní řešení (srovnej s ohýbáním). Druhé důležité pozorování se týká počtu existujících kombinací tvaroslovných zakončení se slovotvorným prodloužením, na nějž t. zakončení navazuje. To umožňuje sloučit s. a t. schemeta do jednoho a zkrátit tak dvoukrokovou cestu na krok jediný. Podrobněji to popisuji dále.

Příklad 3.2.1 Vznik slovního tvaru

form = stem + d-rule.stem-end + i-row.ending
„docezenej“ = „doce“ + „zen“ + „ej“

lemma = CutChars(lemma-source, d-rule.lemma-cut) + d-rule.lemma-end

Pro srovnání je doplněno skládání t. lemmatu. Znak plus naznačuje spojení řetězců.

Sloučení schemat. Jak je naznačeno výše, je možné součit s. a t. schemata. Tím se zjednodušší vyhledávání slovního tvaru.

Sloučení vede k vytvoření nových schemat. Z každého s. schematu se odvozuje jedno nové. K prodloužení kofixu v každém pravidle se přiřetězuje t. konec. Tak vznikají prodloužení, která vedou přímo od kofixu k tvarům. Prodloužení se vkládají do hashovací tabulky (jako klíč). Hodnotou je dvojice seznamu m. značek a instrukce pro konstrukci lemmatu.

Uložení kofixů. Přirozenou volbou pro vyhledávání začátků se stává „trie“. Začátek vložený do trie musí posloužit k rozhodnutí, zda může být některým složeným schematem dotvořen na analyzovaný tvar. K začátku (jako klíči) se vede jako hodnota seznam dvojic odkazu na složené schema a instrukce pro odvození s. lemmatu z kořene. S. lemma může být východiskem pro konstrukci t. lemmatu.

3.3 Vyhledávání

Jasnější představu o fungování navrženého řešení jako celku dá vysvětlení, jak se používá pro vyhledávání. V další podkapitole je pak detailněji popsána realizace.

Od tvaru k lemmatu. Zadaný slovní tvar se vyhledává v trie kofixů. Postupujse se induktivně podle délky tvaru. V každém kroku (hloubce v trie) se zkoumá, zda je (dosud přečtený) prefix rozpoznán, tedy zda je k dispozici seznam možných pokračování. To odpovídá hledání různých možných rozdělení tvaru na kofix a konec. Je-li prefix rozpoznán, následuje procházení uloženého seznamu. Potom se pokračuje sestupem v trie.

Postupně se zpracuje každý prvek seznamu. Ten může popisovat buď přímé vložení tvaru do slovníku (`AddForm`), nebo použití schematu. Postup v prvním případě je jednoduchý. Ověří se, zda byl přečten celý tvar (prefix se rovná celému tvaru, už nezbývají další znaky). Je-li tomu tak, znamená to, že byla nalezena analýza a přidá se do výstupního zásobníku. Případ schematu popisuje další odstavec.

Schema v seznamu. V případě, že položka seznamu odpovídá schematu, je v seznamu uložen odkaz na příslušné schema. To je připraveno ve tvaru hashovací tabulky. Klíči jsou prodloužení, které schema podporuje. Stačí tedy v h. tabulce vyhledat zbytek analyzovaného tvaru (, který nebyl využit k sestupu v trie), aby se zjistilo, zda podporuje tvar. Je-li klíč nalezen, vede nás přímo k seznamu m. značek. Lemma je třeba zkonstruovat. K tomu se použije instrukce v h. tabukly a případně také z uzlu trie.

3.4 Datové struktury, realizace

Potřebné informace jsou rozloženy do mnoha provázaných struktur. Výše napsané poskytuje pohled shora. Nezabíhá se do všech detailů, aby nezastínily celek. Nyní popíši právě detaily.

Uložiště. Používá se několik uložišť – subalokátorů. Podrobněji se o nich píše u ohýbače. Zde tedy jen stručně. Opět je použito uložiště řetězců, čísel, „cut-back“, subalokátor pro seznam, dále pak index skupin značek a uložiště pro data svázaná s kofixem se schematem vloženým do trie.

3.4.1 Morfologické značky, skupiny

Zacházení se značkami nese rysy podobné s ohýbačem. Ten byl napsán později a tam použité řešení považuji za vyspělejší. Dosud nebylo převzato do analyzátoru. Ten tak používá vlastní.

Značky jsou zkopírovány v uložišti řetězců a indexovány v uložišti čísel. Počítá se s tím, že indexace je provedena jako první operace s uložištěm čísel a tak index začíná od nuly.

Se značkou se pracuje pod jejím číslem. Řetězcová reprezentace se použije jen na výstupu, případně ke zjištění konkrétních vlastností značky. Tím se řešení v analyzátoru liší od ohýbače, kde je stupeň a negace kódována v číselném identifikátoru značky.

Skupiny. Na několika místech si je třeba pamatovat skupinu značek. Skupiny se často opakují. Proto je pro skupiny značek zavedena zvláštní podpora, která zamezuje násobnému ukládání totožných skupin a umožňuje nahradit seznam jediným číslem, jeho identifikátorem.

Skupiny značek se ukládání do uložiště čísel. Skupinu tvoří všechny značky od indexu jejích začátku až po zvláštní značku konce. Index začátku se ukládá do zvláštního uložiště čísel s cílem získat husté číslování a snahou vejít se s identifikátorem do 16ti bitů.

Skupiny se vkládájí také do h. tabulky, aby bylo snadné rozhodnout, zda je skupina už zavedena a případně zjistit její identifikátor. H. tabulka se neukládá, protože k analýze není potřeba. Používá se pouze ke stavbě.

Od čísla skupiny k řetězci. Identifikátor skupiny se použije k indexaci ve zvláštním uložišti čísel. Z něj se zjistí index první značky skupiny v uložišti čísel. Skupina je ohraničena značkou konce. Čísla ve skupině identifikují značku. Vedou opět do uložiště řetězců, kde indexují začátek řetězcové reprezentace v uložišti řetězců. Schema indexace ilustruje pseudoprogram 3.4.1

Příklad 3.4.1 Proindexování od skupiny k řetězci značky

```
začátek = skupiny[číslo skupiny]
první značka skupiny = čísla[začátek]
začátek řetězce = čísla[první značka skupiny]
řetězec = řetězce[začátek řetězce]
```

Pseudoprogram: ilustrace rozložení skupiny značek. Hranaté závorky naznačují indexaci v uložišti.

3.4.2 Složené schema

Složené schema v sobě slučuje schema slovotvorné a tvaroslovné. Sloučení je umožněno tím, že t. schema není často sdíleno mnoha pravidly s. schematu a tak sloučení nepředstavuje kombinatorickou expanzi. Podmínkou je také to, že obě schemata na sebe navazují pouhým sřetězením a sloučení tedy nic nebrání.

Organizace složeného schematu. Ke každému s. schematu se vytvoří právě jedno složené. To je popsáno v samostatné h. tabulce. Za každé pravidlo s. schematu a za každý k němu příslušející konec t. schemetu se h. tabulky vkládá jako klíč prodloužení, které vzniknou sřetězením prosloužení obou schemat. Klíč je uložen v uložišti řetězců. Hodnotou přidruženou ke klíči jsou dvě čísla. Jedno identifikuje skupinu značek, druhé „cut-back“ instrukci. Obě čísla jsou uloženy přímo v h. tabulce.

Pole pro h. tabulky bylo předem dynamicky alokováno. Protože každé s. schema je převedeno na jedno složené, používá se pro jejich číslování původní číslování s. schemat.

Může se stát, že v rámci jednoho schematu vznikne různými sřetězeními vícekrát totéž prodloužení. H. tabulky jsou postaveny tak, že mechanizmus řešení kolizí dokáží využít k uložení více hodnot k jednomu klíči. Toho se zde využívá. Při procházení je třeba využít všech nashromážděných hodnot. Tento případ nastává zřídka.

Kapitola 4

Ohýbání

4.1 Řešená úloha

Ohýbání je aplikace morfologie, která si klade za cíl k zadanému lemmatu nalézt a vypsat všechny jeho (tvary včetně jejich morfologické platnosti), které popisuje slovník. V popisovaném pojetí se nepracuje se slovníkem přímo, ale s reprezentací z něj odvozenou. Odvozená reprezentace se specializuje na řešenou úlohu, je kompaktnější a neobsahuje zdrojová data v textové podobě.

Ve slovníku, tedy v prvotní podobě vstupních dat je cesta od struktury pro lemma ke slovním tvarům rozdělena na derivační expanzi (slovotvorba) a ohýbací expanzi (tvarosloví). Cestu stručně popíšu.

Schema vstupu. Na vstupu je množina dvou typů příkazů — vlož slovní tvar s morfologickou platností a proveď derivování dle předepsaného vzoru. Funkce by mohly vypadat takto.

- `AddForm`(*`lemma`*, *`word-form`*, *`list-of-tags`*)
- `AddDerivation`(*`lemma`*, *`stem`*, *`derivational-scheme`*)

Příkaz `AddForm` pracuje přímočaře, příkaz `AddDerivation` vyžaduje definici derivačního schematu. Pak provádí operace schematem předepsané.

Derivační schema. Derivační schema se sestává ze sady pravidel. Každé pravidlo zobrazuje vstupní dvojici *`(lemma, stem)`* na trojici *`(lemma', stem', inflectional-scheme)`*. Formalizace pravidla omezuje zobrazení lemmatu a základu na odtržení konstantního počtu znaků od konce a přiřetězení konstanty. Postup naznačuje schematická implementace funkce `ApplyRule` (viz 4.1.1).

Příklad 4.1.1 Schematická implementace funkce `ApplyRule`

```
ApplyRule(lemma, stem, d-rule) = let
  lemma-source=case d-rule.lemma-source of
         useStem  -> stem
         useLemma -> lemma
  lemma'=Concat(CutChars(lemma-source, d-rule.lemma-cut), d-rule.lemma-end)
  stem' =Concat(CutChars(stem, d-rule.stem-cut)         , d-rule.stem-end)
  infl-scheme=d-rule.inflectional-scheme
in (lemma', stem', infl-scheme)
```

Výsledkem aplikace pravidla (na slovotvorné lemma a slovotvorný začátek) je (tvaroslovné) lemma konkrétního slova a řetězec společný začátkům všech slovních tvarů, tzv. kofix či tvaroslovný začátek. S kofixem dále pracuje ohýbací schema.

Ohýbací schema. Ohýbací schema či vzor je tabulka konců a morfologických značek popisujících m. platnost tvarů, které vzniknou konkatenací kofixu s koncem. Použití řádku tabulky dokumentuje schematická implementace funkce `ApplyInflRow` (viz 4.1.2).

Příklad 4.1.2 Schematická implementace funkce `ApplyInflRow`

```
ApplyInflRow(stem, i-row) = let
  form=Concat(stem, i-row.ending)
in (form, i-row.tag-list)
```

4.2 Myšlenka zvoleného řešení

Řešení se přirozeně rozpadá na dva základní celky. Jedním je vytvoření datové struktury, která umožní efektivně získat k zadanému lemmatu žádaný výstup, druhým pak užití struktury k zodpovídání dotazů. Druhá část je z velké části určena tou první, v níž jsou přijata zásadní rozhodnutí.

Zdroje slovních tvarů. Klíčovou otázkou pro volbu řešení je uvědomit si, odkud a kolik může vzejít příspěvků do množiny slovních tvarů zadaného lemmatu a také to, které lemma uživatel zadává. Protože lemma, které uvádí slovník a vstupuje do funkce `ApplyRule` představuje spíš základ slovotvorby než tvarosloví, pracuje se až s lemmatem, které vytvoří derivační pravidlo (je tedy výstupem funce `ApplyRule` a týká se tvarosloví, ne slovotvorby). Toto zvolené lemma označuji dále jako tvaroslovné, lemma ze slovníku jako slovotvorné.

Slovní tvary příslušící k tvaroslovnému lemmatu mohou být doplňovány z různých struktur pro lemma (ta tvoří třídu ekvivalence na slovotvorném, nikoli tvaroslovném lemmatu). V rámci každé struktury pro lemma může být příspěvek formou příkazů `AddForm` i `AddDerivation`. Příkaz `AddForm` pracuje přímočaře. Přes příkaz `AddDerivation` se k myšlenému tvaroslovnému lemmatu můžeme dostat jeho různým dělením na výchozí prefix a přiřetězený konec (viz funkce `ApplyRule`).

Cílem návrhu je kompaktní a efektivní datová struktura. Proto se vyhýbám expanzi příkazu `AddDerivation` na jednotlivá tvaroslovná lemmata a uvažuji jiný postup. Zvlášť pracuji s výchozími prefixy tvaroslovných lemmat a zvlášť s derivačními schematy. Různá možná rozdělení vyžadují postupné zkoumání uživatelem zadaného lemmatu po znacích od začátku a dotaz do derivačních schemat náležících k prefixu, zda připouštějí (umějí zpracovat) zbývající konec. Myšlenku různých konstrukcí lemmatu ilustruje příklad 4.2.1 na situaci, kde podle slovníku k začátku „chod“ existuje derivační vzor s pravidlem přiřetězujícím konec (d-rule.lemma-end) „ívat“ a obdobně k začátku „chodíva“ existuje konec „t“.

Příklad 4.2.1 Představa dělení lemmatu

chodívat = chod | ívat, chodíva | t

Přirozenou volbou pro vyhledávání začátků se stává „trie“. Začátek vložený do trie musí posloužit k rozhodnutí, zda může být některým derivačním schematem (některým jeho pravidlem) dotvořen na žádané lemma. Pravidla se liší v tom, zda navazují na slovotvorné lemma nebo slovotvorný začátek (to rozhoduje příznak d-rule.lemma-source) a v tom, kolik znaků z něj (od konce) při aplikaci odmazávají. Podle toho je třeba do trie vložit správný řetězec a možnosti jeho pokračování testovat jen v odpovídající podmnožině pravidel der. schematu.

Ekvivalence na pravidlech. Problém různých faktických vstupů pravidel popsaný v předchozím odstavci řeším rozdělením pravidel (každého der. schematu zvlášť) do skupin, tříd ekvivalencí, podle počtu odmazávaných znaků a volby výchozího řetězce. Ke každé skupině vytvářím hashovací tabulku (k ní více později) pro podporované konce. Skupiny se liší začátkem, na který navazují, a tak pro každou skupinu do trie vkládám zvláštní začátek. Na jeden začátek může navazovat více skupin. Ty se nacházejí při prohledávání slovníku neuspořádaně (ne jako celistvý blok), proto jsou řazeny k začátku do spojového seznamu.

Odvození začátku. Slovní tvary ve slovníku zadané příkazem `AddDerivation` jsou v konečném důsledku odvozeny ze slovotvorného začátku (a to přes tvaroslovný začátek). Při prohledávání stromu je k dispozici pouze vstup uživatele a tím je tvaroslovné lemma.

Slovotvorné lemma a s. začátek v obecnosti nemusí mít nic společného. Jejich vztah není popsán pravidlem, ale „tabulkou“ ve slovíku. K hashovací tabulce možných pokračování (ve třídě ekvivalence na derivačních pravidlech) se k vkládanému prefixu s. lemmatu ukládá také s. začátek, přesněji způsob jeho odvození. Zatímco hashovací tabulka je na slovníkové položce nezáviská (závisí pouze na d. schematu

a zpracovávané třídě), způsob odvození s. začátku lze určit až se znalostí konkrétní slovníkové položky. Proto vyžaduje explicitní uložení k prefixu, nelze jej řešit v rámci schematu nebo jiných struktur.

Pokračování, hashovací tabulka. S každým (jednotlivým) vložením prefixu do trie se ukládá i třída pravidel d. schematu, která si vložení vynutila. Třída je zastoupena h. tabulkou konců, které třída zná (jsou klíčem). Tabulka je konci přiřazuje dvojici — ohýbací schema a popis odvození kofixu z s. začátku.

4.3 Realizace

4.3.1 Uložiště

V programu se často používají pole čísel a pole řetězců. Pole mají obvykle dobu života určenou buď na celou dobu běhu programu, tedy stavby datové struktury, nebo se používají jen uvnitř funkce a doba jejich života odpovídá pulzování zásobníku či technice známé jako „*mark and sweep*“. Pro čísla i řetězce (a několik struktur) jsou zavedena uložiště. Uložiště je realizováno dynamicky alokovaným polem a strukturou s řídícími údaji.

S uložišti se pracuje jako se zásobníkem. V případě, že uložené pole má dlouhodobý charakter, zůstává v zásobníku. V případě, že funce potřebuje pole jen na dobu svého běhu, uloží si jeho velikost (vrchol), pracuje a na konci vrchol nastaví na uloženou (původní) hodnotu. Oba přístupy lze kombinovat jedině v pořadí, kdy jsou déle platná data ukládána před dočasnými. Opačné pořadí využít nelze, v programu není potřeba nebo se potřeba obchází.

Uložiště si samo sleduje velikost pole a v případě nutnosti jej zvětší. Uložiště řetězců je navíc doplněno hashovací tabulkou. Tak lze řetězce převádět na atomy, zástupná čísla, s bijektivní vlastností. Důsledkem je také možnost řetězce sdílet.

Uložiště „cut-back“. Uložiště „*cut-back*“ slouží k ukládání a sdílení instrukcí ve tvaru trojice — jednobitový příznak, přirozené číslo z intervalu [0..8191] a řetězec. Vkládaná instrukce je nejprve vyhledávána v h. tabulce. Není-li nalezena, je jí přiděleno volné číslo. Jinak se využije již vložená instrukce.

Instrukce se ukládá do řetězcového uložiště. Nově je zavedena pouze h. tabulka a index uložiště „cut-back“. Index se používá proto, aby k reprezentaci instrukce postačovalo 16-ti bitové číslo. To se přes index odkazuje do uložiště řetězců. H. tabulka se neukládá do souboru, používá se pouze při stavbě.

Uložiště „cut-back“ se používá pro reprezentaci příkazů typu odmazání nějakého počtu znaků z jednoho ze dvou možných vstupů a připojení řetězce.

4.3.2 Sada morfologických značek

Součástí datového souboru vytvořeného pro program na ohýbání slovních tvarů je sada m. značek. Značky popisují morfologickou platnost tvarů na výstupu.

Značka nese hodnoty různých morfologických kategorií včetně stupně (adjektiva, adverbia) a negovanosti. Pro uživatele se jeví jako řetězec. V programu je reprezentována přirozeným číslem. Na stupeň a negovanost se hledí jako na proměnné dosazované do šablony značky.

V přípravné fázi je sada značek zmapována s tím, že na varianty šablony lišící se ve stupni a negaci se hledí právě jako na jednu šablonu. Značky (či šablony) jsou abecedně utřízeny, aby uspořádání na zaváděném očíslování odpovídalo uspořádání na značkách. Po té se v určeném pořadí vkládají šablony (základní tvary řetězcové podoby značek) do uložiště řetězců. Tím jsou značkám přidělována nová čísla — přímo indexy do uložiště. Indexy se navíc zapisují do tabulky pro převod z atomu (původního hustého očíslování) na nové „přímé“ číslování. Převodní tabulka je nakonec doplněna o varianty šablon ve stupni a v negaci. Tabulku není třeba ukládat, používá se pouze při stavbě struktury.

Stupeň a negace se v novém číslování označuje příznaky. Příznaky se ukládají na nejvýznamnější bity indexu do uložiště řetězců. Číslo je 32-bitové a tak z něj lze bez obav 5 bitů (3 pro stupeň a 2 pro negaci) vyhradit. Před použitím identifikátoru značky k indexaci do uložiště řetězců je třeba nejvýznamnější bity vynulovat.

Skupiny značek. Morfologická platnost slovního tvaru je často popsána skupinou značek. Skupiny se opakují, proto je výhodné zvlášť je identifikovat.

Skupina značek je identifikována přirozeným číslem. Protože je žádoucí, aby stačilo 16-ti bitové číslo a skupiny mohou přibývat během celého procesu tvorby datové struktury, odkazuje se identifikátor skupiny do samostatného uložiště. Indexovaná buňka pak ukazuje do uložiště čísel. Do skupiny patří všechny značky v uložišti čísel od první odkazované až po zvláštní hodnotu, která je příznakem konce seznamu značek skupiny.

K vyhledání skupiny mezi již zavedenými slouží hashovací tabulka. Ta se používá pouze ke stavbě, proto se do souboru neukládá.

4.3.3 Ohýbací schemata

Protože datový soubor pro program na ohýbání slovních tvarů má být použitelný samostatně, je třeba do něj nakopírovat ohýbací vzory. Při té příležitosti se využívá uložišť čísel a řetězců dostupných v programu.

Vzor je tvořen příznakem podpory negace a množinou produkčních pravidel. Produkční pravidlo je tvořeno příznakem podpory stupňování, jedním koncem (ten se přiřetězí k t. základu) a množinou značek (morfologická platnost).

V nakopírované reprezentaci je vzor reprezentován přirozeným číslem, které po sečtení se základem slouží k indexaci uložiště čísel. Vztažení indexu k základu bylo zvoleno proto, aby k reprezentaci vzoru postačovalo 16-ti bitové číslo. Číslo na indexu v uložišti odkazuje na první produkční pravidlo v uložišti pravidel (ta jsou v samostatném uložišti). Ke vzoru patří všechna pravidla v uložišti od odkazovaného až po pravidlo z příznakem, že je poslední.

Produkční pravidlo je uloženo jako trojice

- odkazu na přiřetězovaný konec do uložiště řetězců,
- čísla skupiny značek (skupiny značek viz výše)
- a příznaků podpory stupňování, negace a poslednosti. Příznak podpory negace je pro všechna pravidla daného vzoru totožný (patří ke vzoru).

Příznaky stupňování a negace a žolíky ve značce. Podpora stupňování či negace je značena jak příznaky ohýbacího schematu či pravidla tak v každé konkrétní m. značce. Příznakem možnosti stupňovat či negovat je žolík na příslušné pozici ve značce. Při kopírování ohýbacích schemat se kontroluje konzistence obou značení, neschoda je hlášena. Neschoda toho typu, že schema nějakou operaci podporuje, ale ve značce to není vyznačeno žolíkem, se řeší doplněním žolíku do značky. Opačný případ se neřeší.

4.3.4 Derivační schemata

Jak už bylo napsáno výše, pravidlo d. schematu odvozuje ze slovníkové položky (tedy slovotvorného lemmatu nebo základu) tvaroslovný základ a lemma. T. lemma se odvozuje z s. lemmatu nebo s. základu umazáním daného počtu znaků od konce a přiřetězením konce. Do trie se vkládá prefix, který po přiřetězení dává t. lemma.

Z derivačního schematu se předpočítávají třídy ekvivalence podle způsobu odvození t. lemmatu, tedy zdroje (s. lemma či s. základ) a počtu odmazávaných znaků od konce. Tak vzniknou skupiny pravidel s totožnými požadavky na prefix vkládaný do trie. Dalším krokem je sestavení hashovací tabulky pro každou třídu. Klíčem bude přiřetězovaný konec konkrétního pravidla (konec lemmatu), hodnotou dvojice — číslo ohýbacího schematu a konec k získání t. základu (kofixu). Konec je kódován jako „cut-back“.

Hashovací tabulky se ukládají do uložiště hashovacích tabulek (jejich počet závisí na členitosti d. schematu). Zároveň s každou h. tabulkou se do jiného uložiště vkládá struktura popisující třídu. Popis jsou instrukce, co dělat se slovníkovým heslem, tedy z čeho a jak odvozovat začátek t. lemmatu. Popis a h. tabulka se ukládájí zvlášť proto, že popisy není třeba ukládat do souboru, jsou potřeba jen pro stavbu.

Derivační pravidlo je reprezentováno přirozeným číslem od nuly. Po přičtení základu ukazuje do uložiště čísel (vztažení kvůli základu bylo zvoleno proto, aby číslo odpovídalo průvodnímu atomu d. schematu

a aby k jeho reprezentaci postačovalo 16-ti bitové číslo). Odkazované číslo dále ukazuje na první h. tabulku a popisnou strukturu. K derivačnímu schematu patří všechny třídy od první odkazované až po tu, která nemá v popisné struktuře nastaven příznak pokračování.

4.4 Vyhledávání ve struktuře

Zde vysvětlím, jak popsané části zapadají do sebe a jak použít postavenou struktu k vyhledávání. Vynechám při tom technické detaily rozložení toho kterého schematu do uložišť.

Vyhledávání má charakter „backtracku". To není na škodu, protože procházený prostor je malý.

Procházení stromu. V první úrovni se vyhledává lemma v trie. Vyhledává se postupně od začátku po jednotlivých písmenech a v každém navštíveném vrcholu se testuje, zda je k němu připojen spojový seznam. Je-li seznam zadán, znamená to, že strom zná lemma s prefixem odpovídajícím zkoumanému vrcholu a následuje procházení seznamu. Pro to je důležité rozdělení lemmatu na část již vyhledanou a tu zbývající. Po zpracování seznamu se pokračuje hledáním delšího prefixu lemmatu ve stromě.

Procházení seznamu. V seznamu jsou zachyceny možnosti, které prefix nabízí. Seznam se prochází od začátku celý a využívají se ty jeho buňky, které vyhovují hledanému lemmatu.

V nejvyšších bitech odkazu na seznam ze stromu i v odkazech na jednotlivé články seznamu je kódován typ buňky. Rozlišují se dva typy — forma a pravidlo (přesněji třída ekvivalence na pravidlech d. schematu). Podle typu se rozlišuje další zpracovágní buňky.

Zpracování formy. Forma k lemmatu patří tehdy, je-li prefix shodný s hledaným lemmatem, tedy se rovnají. Buňka formy nese číslo „cut-back" instrukce, která se má použít ke konstrukci slovního tvaru a číslo skupiny m. značek k němu náležících. Základem pro odvození slovního tvaru je vyhledávané lemma.

Zpracování pravidla. Buňka pravidla nese číslo „cut-back" instrukce a identifikátor pravida (třídy ekvivalence). V h. tabulce pravidla se hledá zbývající část lemmatu, jeho konec. Najde-li se, pravidlo k lemmatu patří. Jinak se pravidlo ignoruje.

„Cut-back" instrukce se využije k odvození slovotvorného základu z prefixu lemmatu. Z h. tabulky jsme se dozvěděli číslo ohýbacího schematu a číslo „cut-back" instrukce. Podle instrukce je odvozen ze s. základu kofix. Slovní tvary se značkami produkuje ohýbací schema. Ty jsou přidány do výstupu.

4.5 Konstrukce výstupu

Z jednotlivých příspěvků do množiny slovních tvarů získaných z vyhledávací struktury je třeba sestavit a vytisknout ucelený výstup pro uživatele. V přípravné fázi to znamená vypořádat se se žolíky pro stupeň a negaci, filtrovat m. značky podle uživatelem zadaného regulárního výrazu a setřídit řádky výstupu podle značky.

Sběr příspěvků. Příspěvek do množiny slovních tvarů vychází buď přímo z tvaru zadaného do slovníku nebo z aplikace pravidla ohýbacího schematu. Oba typy příspěvku jsou předávány k dalšímu zpracování v jednotné reprezentaci trojicí čísel — odkaz do lokálního uložiště řetězců na slovní tvar, číslo skupiny m. značek a číslo ohýbacího schematu. Sběr příspěvků je ukončen vyčerpáním všech možností procházení vyhledávací struktury.

Expanze skupin značek. Skupina značek je výhodná pro reprezentaci v programu. V přípravě výstupu pro uživatele je však třeba ji rozbít na jednotlivé značky. Nejzávažnějším argumentem pro tento krok je potřeba řádky výstupu třídit podle řetězcové reprezentace m. značky.

Trojice popisující příspěvek se rozbíjí na jednu či více trojic, v nichž je skupina nahrazena značkou samotnou. Protože se jedná opět o dvojici přirozených čísel, ukládají se dvojice do téhož uložiště.

Při rozbíjení skupiny se značky se žolíky pro stupeň a negaci nahrazují odpovídajícími konkretizovanými protějšky, žádal-li uživatel expanzi. Tak se docílí toto, že každý řádek výstupu má svou explicitní reprezentaci v paměti a umožní se tak třídění. Příznak expanze žolíka (superlativ či negativ) se

uloží do nejvýznamějších bitů čísla odkazu. Podle nich se pak před slovní tvar připojí předpony „nej“ a „ne“.

Třídění. Třídění řádek výstupu probíhá podle číselné reprezentace m. značky. Značky byly očíslovány tak, aby jejich uspořádání odpovídalo uspořádání řetězcových šablon (značek s pomlčkou na pozici stupně a negace). To umožňuje porovnávat namísto řetězců čísla. Problém spočívá v substituci hodnot stupně a negace do šablony, protože se stupeň a negace vyznačují na nejvýznamnější bity šablony. To by vedlo k tomu, že by po setřídění značky tvořili souvislé skupiny podle stupně a negace — tyto vlastnosti by měly největší váhu.

Značky se tedy porovnávají nejprve podle čísla šablony a až následně podle bitů stupně a negace. To vede naopak k nejnižší váze těchto vlastností, což sice také neodpovídá třídění na řetězcové reprezentaci, ale je mu mnohem blíž.

Tisk. Tisk je ve formě dvojic m. značek a slovních tvarů doplněních CSTS značkováním. Tisk je ovlivněn parametry zadanými z příkazové řadky — expanze žolíků, filtrování m. značky regulárním výrazem a tisk jména ohýbacího schematu. Parametrizace má vliv na vnitřní smyčky v programu. S cílem omezit v těchto smyčkách větvení jsou připraveny různé verze tiskových funkcí. Podle parametrů z příkazové řádky se zvolí konkrétní sada a odkazy na odpovídající funkce se nachystají do funkčních proměnných, z nichž se pak volají.