MASARYKOVA UNIVERZITA V BRNĚ
FAKULTA INFORMATIKY

Korpusové manažery a jejich efektivní implementace

Pavel Rychlý

Disertační práce
Brno, únor 2000

Prohlášení

Prohlašuji, že jsem tuto práci vypracoval samostatně a že jsem uvedl všechny prameny, kterých jsem v práci využil.

V Brně 25. 2. 2000

Pavel Rychlý

Poděkování

Především děkuji svému školiteli doc. Karlovi Palovi za podporu při výběru zvoleného tématu. Také mu jsem vděčný za nasměrování na řadu studijních materiálů i za připomínky a náměty, které jsem od něho v průběhu tvorby práce dostal.

Též děkuji svým kolegům, kteří mě svými poznámkami a požadavky přiměli k hlubšímu rozboru některých oblastí popisovaných v práci.

Zvláštní dík patří mojí ženě Petře za její trvalou podporu a v neposlední řadě za pomoc s elektronickým zpracováním několika obrázků.

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Tato práce se zabývá *korpusovými manažery*, tedy programovými nástroji na zpracování korpusů. V našem kontextu budeme *korpusem* označovat rozsáhlý soubor textů v elektronické podobě (podrobnější rozbor je uveden v první kapitole). Protože obsahem takového souboru jsou *texty* (ať už se jedná o texty přímo psané nebo o přepis mluvené řeči), někdy se též mluví o *textových korpusech*. Většinou slouží jako zásoba příkladů užití jazyka pro studium slov, jejich významů a nejčastějších kontextů. Korpusy nejsou „objevem“ nijak novým, první korpusy vznikaly od 60. let našeho století. Jejich rozšíření a používání však nastává až v posledních letech, kdy rozvoj počítačů umožňuje snadné a levné zpracování rozsáhlých textových dat.

I když v současné době existuje mnoho korpusových manažerů (nebo programů poskytujících alespoň některé funkce korpusového manažeru), v podstatě neexistuje práce, která by korpusový manažer popisovala bez ohledu na konkrétní implementaci. Důvodem je možná jistá „mezioborovost“ tohoto tématu, kdy různé skupiny uživatelů mají různé názory nejen na funkce korpusového manažeru, ale i na vlastní korpus. Příkladem uvedeného může být jedna z debat v poměrně živé elektronické konferenci *Corpora* [Cor], do které přispívají jak lingvisté (jako nejčastější uživatelé korpusů), tak informatici (jako tvůrci nástrojů). Autor korpusového manažeru CUE[Mas96] tam chtěl rozpoutat diskusi o požadavcích jednotlivých uživatelů na funkce korpusového manažeru. Odpovědí přišlo více než dost, žádná ale neodpovídala na daný dotaz. Až po několikátém opětovném vysvětlení, o co tazateli šlo, poslali pouze tři lidé svůj seznam požadavků, většinou bez jakýchkoliv podrobnějších popisů.

Porovnání existujících (v roce 1993) korpusových manažerů bylo provedeno v přípravné fázi projektu DECIDE [SH94], ale pouze z pohledu tohoto projektu, tedy pro určování kolokací ve slovnících a korpusech [Fon94]. Ve většině knížek o korpusové lingvistice najdeme také kapitolky o korpusových manažerech, které se ovšem omezují na jejich výčet se stručným popisem základních vlastností (viz například [Oak98], [Bal97],[Law98]).

Tato práce se tedy snaží především popsat a zdůvodnit, jaké služby by měl všeobecně použitelný korpusový manažer obsahovat. Následně je postupováno podle klasické metodiky vývoje (nejen) informačních systémů z pohledu softwarového inženýrství: analýza – design – implementace. Každému kroku je věnována jedna kapitola z hlavní části práce.

Popisované úvahy jsou doloženy na příkladech několika korpusů, se kterými mám možnost pracovat, a na ukázkovém korpusu, jenž byl vytvořen pouze pro názornou ilustraci k této práci.

Řazení kapitol

První kapitola popisuje objekt korpusového manažeru – korpus. Je v ní vysvětlena základní otázka, co je to korpus, a jsou zde vysvětleny oblasti a způsoby jeho použití. Vysvětleny jsou i některé základní vlastnosti korpusů vzhledem k jejich použití.

Druhou kapitolu tvoří analýza a specifikace požadavků korpusového manažeru. Jsou v ní popsány všechny funkce, které musí korpusový manažer poskytovat uživatelům, a to nejen jejich vlastnosti ale i jejich parametry. Pro jednotlivé operace jsou odhadnuty časové požadavky a rozsahy parametrů. Poslední část kapitoly (2.9 na straně 67) stručně shrnuje požadavky na korpusový manažer.

Třetí kapitola obsahuje návrh implementace. Jednotlivé vlastnosti a funkce korpusového manažeru jsou podrobně rozebrány a uvedeny dosud používané datové struktury a algoritmy pro jejich řešení. Pokud existuje více různých přístupů pro jednu funkci, je zdůvodněn výběr použitého přístupu, v některých méně zřejmých případech podpořený i provedenými praktickými testy. V mnoha případech jsou navrženy optimalizace uvedených struktur či algoritmů pro případ korpusového manažeru nebo vytvořeny zcela nové postupy. U algoritmů jsou uvedeny (a v netriviálních případech i dokázány) jejich paměťové a časové složitosti.

Čtvrtá kapitola přibližuje vlastní implementaci dle předchozího návrhu. Není zde podrobná uživatelská či programová dokumentace k vytvořenému systému, pouze jsou popsány některé základní prvky implementace a řešení některých problémů, které při implementaci vznikly. Výsledný systém také zatím není dokončen do „produkční" verze, je ale funkční a obsahuje všechny základní komponenty. Pro nedokončené části je vždy naznačen způsob budoucí implementace.

Na některých místech textu jsou zmíněny problémy či zajímavosti, jejichž popis zcela nezapadá do vlastního textu. Proto jsou uvedeny v samostatných článcích a v textu je na tyto *textové rámy* uveden pouze odkaz.

Kapitola 1

Korpusy

V této kapitole podrobněji popíšeme, co rozumíme pod pojmem *korpus* z různých pohledů a úrovní. Nejdříve tedy definice. Textový korpus se většinou definuje jako:

> rozsáhlý vnitřně strukturovaný a ucelený soubor textů daného jazyka elektronicky uložený a zpracovávaný.

Korpus tedy obsahuje texty psané jedním přirozeným jazykem, které jsou strojově (počítačově) čitelné. Uvedené definici by mohly odpovídat i existující elektronické knihovny, které dávají k dispozici elektronické verze celých knih. U korpusu je ale kladen důraz na jazykovou stránku, takže elektronické knihovny jsou pouze vydatným zdrojem dat pro korpusy.

Tvorba korpusů vychází z následujících předpokladů [Pal96]:

- jazyková data jsou v korpusu uložena ve své *přirozené* kontextové podobě a užití, proto je lze všestranně a opakovaně zkoumat a vyvozovat z nich příslušné teoretické generalizace,
- *velký rozsah dat* v korpusu minimalizuje nebezpečí, že by mohlo dojít k převaze okrajových jevů nad základními a typickými,
- velký rozsah dat v korpusu je podmínkou dostatečné *reprezentativnosti*.

Jednotlivé korpusy se liší svou velikostí, jazykem, typem textů, zdrojem dat, dodatečnou anotací atd.

1.1 Příklady korpusů

Pro názornost uvedeme několik reálných korpusů. Tyto korpusy byly využity k ověřování hypotéz z této práce i k testování jednotlivých částí korpusového manažeru. Různé úvahy v této práci jsou podloženy příklady právě z těchto korpusů.

Následující seznam uvádí stručnou charakteristiku příkladových korpusů:

Brown je anglický korpus a je již klasikou mezi korpusy. Vznikl v roce 1964 na Brown University [FK79], jako první moderní počítačově čitelný všeobecný korpus, v roce 1979 se objevilo jeho druhé vydání, které již obsahovalo gramatické značkování.

Obsahuje texty psané americkou angličtinou, které byly vytištěny ve Spojených státech v roce 1961. Texty byly vybrány z 15 různých kategorií a celkem mají přes milion slov. Dnes je již tento korpus poněkud malý a zastaralý (co se jazyka týče), ale stále se ještě používá.

Desam je český korpus, který vznikl na Fakultě informatiky Masarykovy University v Brně. Obsahuje články z novin a časopisů. Důležitý je zejména tím, že byl morfologicky označkován a toto značkování bylo ručně zjednoznačněno (odtud i jeho název – DESAMbiguovaný korpus).

CNK25 je rovněž český korpus, velký svým rozsahem. Je to část Českého národního korpusu, odpovídající jeho stavu na začátku roku 1999. Korpus vznikl a neustále je rozšiřován v Ústavu českého národního korpusu v Praze. Jeho největší část tvoří také články z novin a časopisů, obsahuje ale též části knih a jiné formy psaného textu.

KOČKA vznikl pouze pro ilustraci k této práci.[1] Jeho celý text je uveden v samostatném rámu *Kočka na okně* na straně 5. Jedná se o text ke stejnojmenné písni od Jiřího Suchého.

Další vlastnosti těchto korpusů budou uvedeny dále. Na některých místech práce jsou pro porovnání zmiňovány i jiné korpusy, se kterými jsem se ale nemohl osobně seznámit[2], a informace o nich jsou pouze zprostředkované.

Mezi jedny z nejznámějších korpusů patří velké anglické korpusy, které vznikly na významných lingvistických pracovištích ve Velké Británii:

BNC – British National Corpus – vytvořilo konsorcium vedené Oxford University Press, jehož dalšími členy jsou vydavatelé slovníků a akademické instituce. Budování korpusu začalo v roce 1991 a skončilo v roce 1994. Podrobnější informace lze nalézt například na `bnc`. Na

[1] Podle naší definice se vlastně o korpus nejedná, není to soubor textů, ale text jeden, a asi v žádném případě bychom uvedený text nenazvali rozsáhlým.

[2] U korpusu Brown jsem mohl pracovat pouze s jeho necelou čtvrtinou, která se dodává s korpusovým manažerem CQP.

Kočka na okně

Na okně seděla kočka,
byl horký letní den,
na okně seděla kočka
a koukala se ven,
byl horký letní den
a kdekdo chodil bos,
na okně seděla kočka,
venku zpíval kos.

Byl horký letní den
a celý svět se smál
a mně veselý sen
se pod jabloní zdál,
a celý svět se smál,
vidím to jako dnes,
na okně seděla kočka
a venku štěkal pes.

základě tohoto korpusu jsou mimo jiné vytvářeny známé anglické slovníky Oxford English Dictionary, naposledy New Oxford Dictionary of English (1998).

BoE – The Bank of English – vytvořený v COBUILDu (divize HarperCollins Publishers) a na The University of Birgmingham v roce 1991. Korpus je stále rozšiřován a velká snaha je u něj věnována udržení aktuálnosti textů, většina jich vznikla až po roce 1990. Tento korpus používají lexikografové při tvorbě slovníků Collins COBUILD English Dictionary.

1.2 Využití korpusů

Korpusy jsou používány v celé řadě různých výzkumů i aplikací v závislosti na typu korpusu. V této kapitole si uvedeme nejčastější případy.

Korpusová lingvistika

Korpusová lingvistika je odvětvím lingvistiky, kde jsou jazykové fenomény studovány na základě korpusů, tedy příkladů reálných textů. Toto odvětví se rozvinulo jednak díky dostupnosti počítačů, které dokáží zpracovávat rozsáhlé texty, jednak díky dostupnosti textů v elektronické podobě. Právě dostupnost velkého množství textů je klíčovou pro relativně levné budování současných rozsáhlých korpusů (v řádech stovek milionů pozic), texty jsou k volnému stažení zejména na Internetu. Před jeho větším rozšířením tvořil největší část nákladů na vytvoření rozsáhlého korpusu hlavně převod dat do elektronické podoby. Například korpus Brown vznikal ve své prvotní podobě (tedy bez značkování) celé čtyři roky, hlavně díky tomu, že se texty musely do počítače přepisovat. Značkovaná verze byla vydána až po 15 letech v roce 1979.

Většina lingvistických teorií může být nejenom vysvětlena pomocí vykonstruovaných příkladů, ale s pomocí korpusů i dokázána jejich všeobecná platnost, nebo naopak platnost pouze v určitém druhu textů. Použití korpusů tak v řadě případů může odstranit problém introspekce, kdy většina lidí je „z vlastní zkušenosti“ přesvědčena o nějakém jevu (například který z různých významů jednoho slova je nejpoužívanější), skutečnost je však jiná. Korpusy jsou dobrým zdrojem dat pro celou řadu oblastí lingvistiky: od morfologie, přes syntaxi, sémantiku, stylistiku, až po sociolingvistiku.

Mnoho korpusů (například i BNC) bylo vytvořeno zejména pro podporu tvorby slovníků a lexikografové jsou v současné době asi nejčastější uživatelé korpusů.

Korpusem podporovaná výuka jazyků

Nutnou součástí zvládnutí libovolného cizího jazyka je znalost slovíček, frází či celých vět. Některé metody výuky jazyků dokonce téměř nic jiného (například gramatiku) ani neučí. Korpus může být vhodným zdrojem takových frází a vět.

Pokročilí studenti mohou korpus využít jako alternativu či doplnění slovníku. Existují počítačové programy (například *Cobuild on CD-ROM*), které obsahují klasický slovník v elektronické podobě spolu s menším korpusem (pečlivě vybraným z velkého korpusu), ve kterém lze jednotlivá hesla ze slovníku přímo (stiskem klávesy) vyhledávat.

Ostatní obory

Použití korpusů není omezeno jenom na korpusovou lingvistiku. V mnoha aplikacích se používají některé výsledky ze zpracování korpusů: seznamy slov, popř. se svými četnostmi, výběry kolokací atd. Mezi tyto aplikace se řadí zejména systémy na zpracování textů (spellcheckry, značkovače apod.), systémy na syntézu a rozpoznávání řeči atd.

1.3 Tvorba korpusů

Korpusy mohou vznikat pouhým převodem dostupných textů v elektronické podobě, to je velice levné řešení. Pro rozsáhlejší korpusy, které kladou velký důraz na reprezentativnost, existují poměrně rozsáhlá kritéria, co a jak se má do korpusu vložit. Potom je nutné některé texty skenovat a převádět pomocí programů na rozpoznávání písma (OCR), popřípadě ručně přepisovat do počítače. Se získáváním textů pro korpus jsou spojeny nejen technické potíže. Vždy je nutné brát v úvahu autorská práva získaných textů, což mnohdy vyžaduje potvrzování smluv či povolení. Například současná verze BNC nemůže být kvůli autorsko-právním omezením několika textů distribuována mimo Evropskou Unii.

Při získávání textů z elektronických zdrojů se setkáváme s problémem duplicit textů (článků), kdy stejný text je zveřejněn na několika různých místech a v korpusu by pak byl dvakrát či vícekrát. Pro rozsáhlé korpusy samozřejmě nemůžeme k detekci duplicit použít přímou metodu porovnávání všech dvojic textů. Efektivní postup založený na výpočtu tzv. *signatur* (charakteristických kódů) pro každý text popisuje například [RSF99].

Jednou vytvořený korpus je možné stále používat, ale každým rokem se stává zastaralejším, mezi takové korpusy patří i Brown korpus a BNC. Jiné korpusy, jako například BoE, jsou neustále rozšiřovány o nové texty. V korpusu pak je stále aktuální jazyk, ale jen z části, v delším časovém horizontu je korpus stejně zastaralý. Řešením tohoto problému jsou *průběžné korpusy (monitor corpus)*, ve kterých jsou novými přidávanými texty nahrazovány texty starší, které jsou z korpusu odstraňovány do archivu. Korpus je tak stále držen ve stejné velikosti a stále aktuální.

Mluvený a psaný projev

Všechny větší korpusy, které slouží k budování všeobecných slovníků, většinou obsahují kromě čistě psaného textu jak texty, které byly napsány k proslovu (scénáře her, proslovy), tak texty vzniklé přepisem mluvené řeči (běžná konverzace, rádiové/televizní vysílání, rozhovory, diskuse atd.). V jakém poměru má být ve všeobecně použitelném korpusu obsažen jazyk psaný a mluvený není zcela jasné. Některé psycholingvistické studie ukazují, že poměr psaného a mluveného projevu je zhruba 1:2. Pravdou ale je, že mluvená řeč má mnohdy zcela jinou strukturu (syntax) než psaný text, a je tedy otázkou, zda se mají obě formy dávat do jednoho korpusu. Každopádně u zmíněných korpusů bývá poměr psaný:mluvený kolem 9:1. I když mluvená část zabírá pouze 10 %, je její získání nákladnější než pořízení zbytku korpusu, protože záznam řeči (nejčastěji magnetofonové nahrávky) a jeho následný převod (zatím nejčastěji manuální přepis) do textu je velmi nákladný.

Existují i mluvené korpusy obsahující výhradně mluvené slovo, u kterých je primární složkou zvukový záznam promluv a textový přepis (někdy pouze v nějakém fonetickém zápise) je druhotný. Takové korpusy slouží ke studiu výslovnosti nebo jako trénovací či testovací data pro syntézu či analýzu řeči. Protože jde hlavně o zvukový záznam, zpracovávají se na rozdíl od běžných textových korpusů odlišnými nástroji a v dalším textu tedy nebudeme mít tento typ korpusů na mysli. Některé principy platné pro textové korpusy však lze na ty mluvené přímo aplikovat, nebo je možné k textovým korpusům zvukový záznam částečně připojit.

Dokumenty

Při tvorbě korpusu se texty člení do tzv. *dokumentů*. Jsou to části textu, které tvoří nějaký ucelený prvek: jeden článek v novinách/časopise, rubrika v novinách/časopise, kapitola knihy apod. Tyto dokumenty jsou také největší části textu, u kterých jsou uchovávány tzv. metainformace (viz kap. 1.6.1). V některých korpusech jsou všechny dokumenty vytvářeny ve stejné velikosti. Například v korpusu Brown obsahuje každý dokument o málo víc než 2000 slov: končí za první větou, ve které je 2000 slov dosaženo. U většiny ostatních korpusů však dokumenty tvoří logické celky a jejich velikosti se tedy značně liší.

1.4 Pozice, tokenizace

Každý rozsáhlejší systém, který má být snadno použitelný, musí vytvářet nějakou abstraktní úroveň s dostatečně jednoduchými prvky. Použití takových prvků umožňuje snadnou specifikaci problémů a postupů jejich řešení, také usnadňuje komunikaci mezi jednotlivými uživateli.

Jak bylo uvedeno dříve, korpus tvoří texty, a ty se skládají ze slov. Slovo definované jako řetězec alfabetických znaků by tedy mohlo být jedním ze základních prvků. Texty ovšem obsahují i jiné řetězce znaků – zejména čísla a interpunkční znaménka (tečky, čárky, uvozovky, apod.). Jednotlivá slova (a další „objekty") jsou také v textech od sebe navzájem odděleny jiným způsobem – mezerou, novým řádkem, odstavcovou mezerou, nebo také ničím (interpunkční znaménka se většinou píší bezprostředně za předcházející slovo). Způsob oddělení slov je otázka spíše typografická či technická a s jazykem (který je našim předmětem zkoumání) nemá většinou co do činění. Proto budeme v korpusu způsob oddělení jednotlivých slov zaznamenávat pouze sekundárně.

Základním stavebním prvkem korpusu by tedy mělo být něco, co může být slovem, číslem nebo nějakým znaménkem. Tento prvek budeme nadále označovat jako *pozici* (v některých pramenech se používá pojem *L-word*). Korpus tedy tvoří posloupnost pozic, na každé pozici je jedno slovo, číslo, interpunkční znaménko apod. Jakým způsobem byly v původním textu po sobě jdoucí pozice odděleny, nás v tuto chvíli nezajímá. Někdy se místo termínu pozice používá *token* a proces rozdělení textu na pozice je často označován jako *tokenizace (tokenization)*.

Každé zjednodušení ale přináší nějaké problémy. Pomocí uvedené definice můžeme text rozdělit na pozice, v některých případech však nemusí být rozdělení až tak jednoznačné. Má být například *bude-li* jedna pozice, dvě pozice (*bude, -li*) nebo tři pozice (*bude, -, li*)? Má tečka za zkratkami, které se píší vždy s tečkou, být na jedné pozici se zkratkou nebo mají tvořit samostatné pozice? Pokud spojit, jak to bude s takovou zkratkou na konci věty, kde tvoří tečka interpunkční znaménko? Jak se má zacházet s pevnými kolokacemi typu *ad hoc*, kde jednotlivé části nemají samostatně smysl?

Pokud korpus slouží ke zkoumání jazyka z pohledu mluveného projevu, bylo by jistě výhodnější, aby pozice odpovídaly segmentaci v řeči. Tedy například aby předložky, které se vyslovují dohromady s následujícím slovem, tvořily s daným slovem jednu pozici. Tím dostáváme několik různých rozdělení do pozic, která ani nejsou jedno zjemněním druhého (některé pozice prvního rozdělení tvoří v druhém více pozic), ale „kříží se".

Bohužel neexistuje jednoduché řešení vhodné ve všech aplikacích. Obecným postupem je rozdělit do samostatných pozic vše, co je odděleno jakoukoli mezerou nebo je na předělu písmen a jiných znaků, až na ustálená

„nedělitelná“ spojení. Příklady skupin, které tvoří jednu pozici obsahující nějakou mezeru nebo jiný oddělovač, jsou vedeny v tabulce 1.1. Pro uživatele korpusu je nutné uvést úplný popis, jak jsou pozice tvořeny. Z uvedených korpusů existuje takový popis pouze pro BNC, pro který v uživatelské příručce [Bur95, str. 91] zabírá několik seznamů výjimek rovných deset stránek.

Desam	Brown	BNC
i kdyby	had to	a few
a tak	at least	as well as
ve vztahu k	of course	follow up
pro případ	At the same time	for the most part
ad hoc	interior design	up to the minute
U Hybernů	all right	with regard to
SPT Telecom	Miss Ada	ad hoc
Ústí nad Orlicí	United States	ex libris
(02) 2324 718	E. H. Hemenway's	faux pas
198/1990 Sb.	30 minutes	per se

Tabulka 1.1: Příklady „víceslovných“ skupin na jedné pozici

Je nutno poznamenat, že často není rozdělení do pozic provedeno systematicky, tedy na jednom místě v korpusu je zvolená skupina na jedné pozici a na jiném místě tvoří stejná skupina více pozic. Tento stav je nežádoucí a je nutné ho považovat za chybu. Samozřejmě pokud slova skupinu netvoří, musí být rozdělena do jednotlivých pozic, jako například slova *had* a *to* v následujícím úryvku z korpusu Brown: „*... form they have* had to *this day.*“

Z výše uvedeného by mohlo vyplývat, že právě popsaná abstrakce se minula účinkem a zavedení pojmu pozice přináší pouze problémy. Pokud by jedinou funkcí korpusového manažeru bylo vyhledávání v korpusu pouze podle textu, můžeme se bez pozic obejít. Pokud ale do korpusu zavedeme dodatečné značkování (viz níže), nebo pokud provádíme nad korpusy různé statistické výpočty (a to i velice jednoduché jako počet různých slov v korpusu), bez pojmu pozice se neobejdeme.

1.5 Velikosti

Jedna ze základních charakteristik korpusu, jeho velikost, je přirozeně definována jako celkový počet jeho pozic. Velikost korpusů můžeme také

porovnávat podle počtu dokumentů, popřípadě počtu slov (bez čísel, interpunkce atd.).

V této části se zaměříme na otázky: jak velké korpusy existují, zda jsou tyto velikosti dostatečné, zda jsou větší korpusy obecně lepší než menší.

1.5.1 Jak velké korpusy se používají

Velikosti příkladových korpusů jsou uvedeny v tabulce 1.2. Pro porovnání

Jméno korpusu	počet pozic	počet dokumentů
Brown	1 014 312	500
Desam	1 245 537	3 056
CNK25	121 493 290	329 977
KOČKA	77	1

Tabulka 1.2: Velikosti příkladových korpusů

ještě uveďme, že například anglický BNC uvádí přes 100 milionů slov a BoE měl v posledním vydání v polovině roku 1998 přes 329 milionů slov.

Jakékoliv velikosti je nutné vždy brát s rezervou a všímat si spíše řádů než konkrétních čísel. V předešlé podkapitole jsme popsali, že se rozdělení do pozic může lišit. Rozdělení do dokumentů je často i v rámci jednoho korpusu tak rozdílné, že vlastně o ničem nevypovídá.

Některé jednoduché korpusové manažery nedokáží zpracovávat korpusy větší než několik stovek tisíc pozic. I takové programy mají svoje uplatnění. Pokud chceme po jazykové stránce zpracovat pouze jednu knihu nějakého autora, nepotřebujeme více. Když chceme přejít od jednoho díla k veškeré dostupné tvorbě zvoleného autora, můžeme již vytvořit korpus v řádech milionů pozic.

Pro vytvoření korpusu, který bude pokrývat celý jeden jazyk, bychom mohli chtít začlenit veškerý písemný projev ve zvoleném jazyce. Na to nám současné počítače technicky nestačí (ale za několik let to může být jinak), ale hlavně to je prakticky neproveditelné. Takový korpus tedy vytvoříme z dostupných vzorků textů s tím, že se budeme snažit zachovat reprezentativnost zvoleného výběru.

Nyní narážíme na otázku, co je reprezentativní vzorek a co není. Hlavní problém vystihuje pozorování známé jako *Zipfův zákon (Zipf's law)*[Zip49]. Ten tvrdí, že četnost nějakého prvku (jevu) je nepřímo úměrná jeho pořadí (určenému podle četnosti). Tedy, že četnost druhého nejčetnějšího prvku je rovna polovině četnosti nejčetnějšího prvku, četnost třetího nejčetnějšího je rovna třetině četnosti nejčetnějšího prvku atd. Samozřejmě to neplatí absolutně, ale například korpus Desam obsahuje 51,2 % různých slovních tvarů s četností jedna[3] (tzv. *hapax legomena*). Zajímavé grafy zobrazující četnosti slov v korpusu Desam z různých pohledů lze najít v [PRS97].

1.5.2 Reprezentativnost korpusu

Obecně můžeme říct, že korpus je reprezentativní, pokud by v jiném větším korpusu konstruovaném pro stejný účel měly všechny zkoumané parametry stejné hodnoty. Nedá se tedy říct, že třeba milionový korpus (korpus s jedním milionem pozic) není reprezentativní, ale stomilionový korpus už reprezentativní je. Vždy záleží na účelu, na parametrech, pro které v korpusu hledáme hodnoty. Někteří uživatelé hledají číselné hodnoty: pro velikosti slov, počty slov v celém korpusu, počty slov v nějakém konkrétním kontextu apod. Jiní hledají seznamy všech slov, které se vyskytují v blízkosti zvoleného slova, které rozvíjí jistý člen, které se pojí s jistou předložkou apod.

Je samozřejmé, že téměř libovolná zkoumaná charakteristika je závislá na zvoleném jazyku, ale může záviset i na žánru, oboru, stylu atd. V některých případech může být dostatečných pouze několik stovek slov, jindy nestačí několik stovek milionů slov. Pokud stačí méně, nemá cenu používat větší korpus. Při efektivní implementaci, která bude popsána dále, sice pro jednoduché dotazy (na jedno slovo) dostáváme výsledek okamžitě bez ohledu na velikost korpusu, výpočet libovolných statistik ovšem je přímo (pro některé i kvadraticky) úměrný velikosti korpusu, a tedy trvá pro větší korpusy zbytečně déle.

Příkladem úspěšného použití malých korpusů je projekt na automatické určování jazyka neznámého textu (`www.let.rug.nl/~vannoord/TextCat/`). Pro každý jazyk (popřípadě v různých kódováních, viz textový rám *Kódování znaků pro češtinu*) je uložen trénovací text velikosti necelých 5 KB. Pro každý z těchto textů je spočítána statistika n-gramů písmen (pro $1 \leq n \leq 5$), pro neznámý text je spočtena stejná statistika a jako jazyk je označen ten s „nejbližším" modelem. (Podrobnější popis je uveden v [CT94].)

Celý systém je přes svoji jednoduchost a celkové velké množství použitých jazyků (v listopadu 1999 to bylo 69 jazyků, z toho 6 ve více než jednom kódování) až překvapivě úspěšný. Autoři na své stránce vyzývají ostatní k soutěži a ze známých produktů řešících tuto úlohu rozpoznává uvedený program rozhodně největší počet jazyků. O rozpoznání jazyka za použití statistických metod je podrobněji psáno například v [Dun94].

[3] Tyto slovní tvary ale pokrývají pouze 5,1 % textu.

Kódování znaků pro češtinu

Pro češtinu se používají v současné době dvě hlavní kódování: *ISO-Latin-2* podle mezinárodní normy ISO [ISO99] a *Windows 1250* prosazované společností Microsoft v operačních systémech Windows. V prostředí Internetu se bohužel stále ještě velice často setkáváme s „kódováním" ASCII, tedy psaní „cestiny" bez háčků a čárek, což ovšem není správný český text. Zatím neexistuje žádný volně dostupný program pro „počeštění" textů bez háčků a čárek, ale experimentální pokusy ukazují, že i při jednoduché metodě, kdy se z různých možností vybere slovo s nejvyšší četností v korpusu, je chybovost pouze kolem 3,5 % z celého textu. V systémech MS DOS se nejčastěji používají „oficiální" kódování *852* (opět od firmy Microsoft) a kódování *bratří Kamenických*, které bylo navrženo ještě před řešením Microsoftu a má mnoho výhod.
Převod mezi jednotlivými kódováními je již víceméně snadno zvládnutelná procedura[*)], například programem `cstocs` [Kas], ale neustálý převod mezi různými kódováními je přinejmenším obtěžující. Situace se snad bude zlepšovat, protože programů pro DOS neustále ubývá a i firma Microsoft přechází v novějších verzích systému Windows na kódování standardu ISO.

[*)] někdy mohou dělat problémy převody typografických znaků, kdy pro klasickou uvozovku (") existují dva ekvivalenty: levá („) a pravá (")

Porovnání úspěšnosti je ovšem velice subjektivní. Málokterý člověk má k dispozici dostatek různých textů pro všechny rozpoznávané jazyky, aby mohl provést dostatečně věrohodné posouzení úspěšnosti.

Test samozřejmě selže na textech, které jsou stejné v různých jazycích. Například i mezi angličtinou a češtinou, které jsou od sebe dost vzdálené co do pravopisu, existuje velké množství slov, která se relativně často vyskytují v obou jazycích s naprosto rozdílným významem, příklad takových slov je uveden v tabulce 1.3[4] i s jejich četnostmi v relativně malých korpusech Brown a Desam.

Slovo	Brown	Desam
had	1135	3
jet	2	21
let	56	968
list	14	79
most	154	7
my	208	105
on	1083	95
past	51	8
pasty	2	6
plot	9	2
pole	1	74
proud	12	14
rod	1	10
step	18	2
to	4062	4653

Tabulka 1.3: Stejná slova v češtině a angličtině s různým významem

Potenciální funkčnost systému můžeme podpořit faktem, že leckterý člověk dokáže poznat, o jaký jazyk se jedná, i když ho vůbec neovládá. Použitý princip předpokládá, že uvedená statistika bude u různých textů jednoho jazyka stále stejná (s jistou tolerancí) a že pro různé jazyky je uvedená statistika výrazně rozdílná.

Pro alespoň částečné ověření správnosti použité myšlenky ovšem vystačíme pouze s jedním jazykem, v našem případě s češtinou. Jazykové modely, které se porovnávají s testovaným textem, se týkají písmen. Můžeme tedy ověřit, jestli jsou dané statistiky shodné na různých částech korpusu.

[4]Že mohou existovat i celé texty (smysluplné v obou jazycích) složené z takových slov, se dalo vidět na konferenci Sofsem'97, kde bylo prezentováno hned několik takových děl.

Pokud tyto statistiky nejsou dostatečně/alespoň relativně stabilní v celém korpusu, použitý systém nemůže být celkově úspěšný. V tabulce 1.4 jsou uvedeny statistiky četností písmen a jejich n-gramů pro různé části textů z korpusu Desam: nejdříve pro 10 000 pozic a potom pro sedm různých částí velikosti 500 pozic (v tabulce je pouze prvních 35 údajů). Jak je vidět, statistiky jsou poměrně stabilní. Na druhou stranu bude tento systém těžko rozpoznávat kódování češtiny *ISO Latin 2* a *1250*, protože první písmeno (*ž*), ve kterém se oba systémy liší, je až na 44. místě a další (*š*) dokonce až na 70. místě.

1.5.3 Vztahy type/token

Oblíbenou charakteristikou korpusů (i textů obecně) je poměr *type/token*, resp. *token/type*. *Token* zde znamená počet pozic v textu, *type* počet všech různých slov v daném textu. Tento poměr samozřejmě závisí na tom, co všechno považujeme za slova (jestli počítáme i čísla a interpunkční znaménka apod.) a jak je rozdělujeme do pozic. Podle použitých pravidel dostáváme různé číselné hodnoty, které se ale v zásadě příliš neliší. Nic nám také nebrání, abychom tuto charakteristiku počítali pouze pro slova, lze ji vyčíslit například i pro základní tvary.

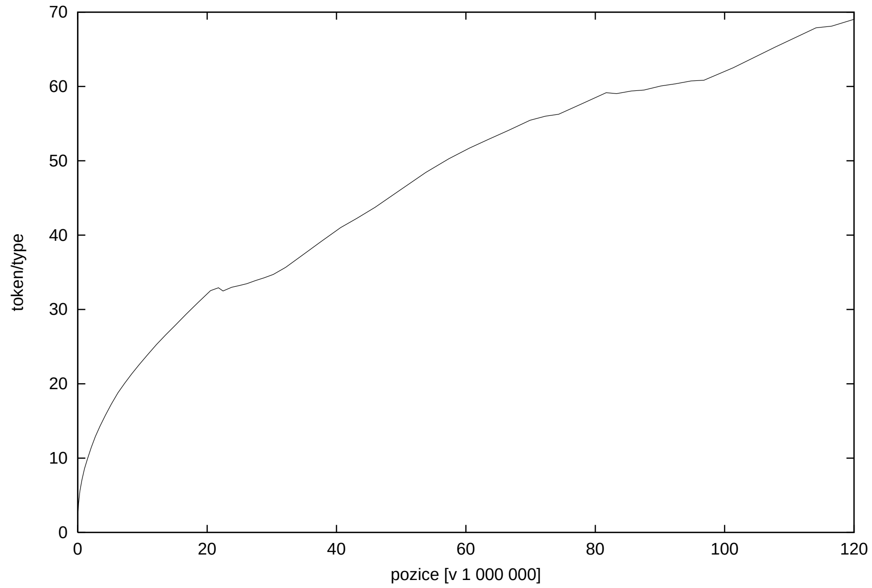
Vztah type/token udává průměrný počet opakování jednoho slova v textu, což může ukazovat na „bohatost" zkoumaného jazyka, a to v různých pohledech. Můžeme tak porovnávat velikost použité slovní zásoby jednotlivých autorů či stylů, nebo můžeme porovnávat rozdílné jazyky. Například jazyky, kde se tvoří nové významy kombinací několika samostatných slov budou mít zřejmě vyšší poměr token/type než jazyky, které tvoří slova předponami a příponami.

Jakékoliv pozorování založené na vztahu type/token má ovšem jedno úskalí, na které se někdy zapomíná. Vždy totiž musíme tuto charakteristiku počítat na textech o stejných velikostech, protože s velikostí textu se poměr token/type velice mění (roste). Tato změna je dobře viditelná na obrázku 1.1, kde je zobrazen poměr token/type v závislosti na velikosti textu v korpusu CNK25. Hodnota poměru pro celý korpus je 69,3, pro menší velikosti byl výpočet proveden pro odpovídající část ze začátku korpusu.

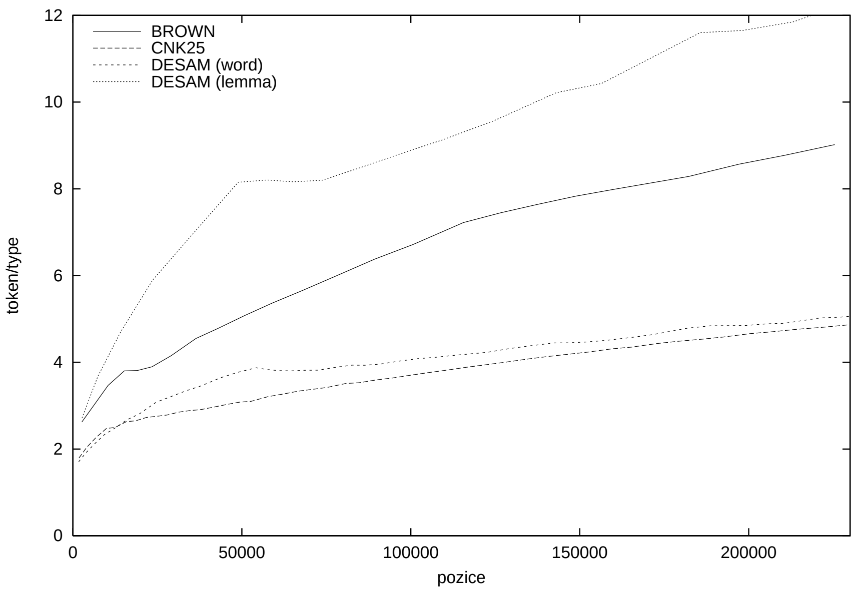
Další obrázek (1.2) ukazuje stejnou závislost v korpusech Brown a Desam (zde navíc i pro základní tvary slov) a je z něho patrný rozdíl mezi angličtinou a češtinou i základním tvarem a vlastním tvarem slova v češtině. Také je vidět blízkost hodnot pro různé texty stejného jazyka.

n-gram	Relativní četnost n-gramu v promile							
_	338,0	331,6	330,3	341,7	342,6	359,9	333,0	345,3
e	62,0	61,3	61,6	58,9	62,1	58,2	57,4	60,5
o	61,0	58,7	69,0	67,2	61,7	62,2	62,7	65,0
a	53,0	57,4	39,9	52,0	42,1	51,6	55,1	45,0
n	49,0	55,4	53,8	50,3	50,5	51,6	55,1	48,5
t	41,0	37,5	39,3	41,3	44,1	38,2	37,4	43,7
i	37,0	29,8	26,1	34,4	31,9	36,4	32,7	27,5
s	35,0	23,5	29,8	32,7	33,9	28,0	35,7	29,2
l	34,0	32,5	25,4	34,1	23,4	30,2	26,0	24,1
v	33,0	31,2	25,7	33,1	35,3	32,4	30,7	29,9
r	32,0	28,5	27,1	27,2	28,8	31,3	34,0	35,8
d	29,0	27,9	31,5	30,0	33,9	28,0	28,4	22,0
k	28,0	24,9	27,7	24,8	30,5	29,4	20,4	26,8
m	25,0	21,9	15,6	21,7	23,4	24,4	31,0	23,0
í	24,0	27,2	33,2	24,8	24,8	21,8	32,0	24,8
u	23,0	17,2	25,7	26,5	21,4	24,0	22,4	21,0
p	22,0	26,2	30,5	23,1	25,8	22,2	29,7	24,8
c	20,0	18,6	16,9	19,3	20,0	20,0	19,0	21,0
h	18,0	18,9	18,6	18,9	17,3	17,8	15,7	16,2
á	17,0	18,9	19,6	12,7	23,4	13,4	18,0	12,0
a_	16,0	17,6	13,2	12,7	11,2	12,4	10,7	10,7
e_	16,0	17,6	16,2	19,3	18,7	14,2	19,7	17,9
_p	15,0	14,6	15,9	13,8	19,7	10,9	17,7	16,9
j	15,0	16,6	15,2	15,8	12,9	18,2	12,7	12,4
z	15,0	20,6	22,0	23,4	15,6	18,2	15,0	18,9
y	14,0	12,9	11,5	16,5	15,3	12,7	12,0	11,0
_s	12,0	10,6	11,5	10,7	11,5	5,8	13,0	9,6
_v	12,0	9,6	9,1	10,0	11,9	14,2	11,7	10,3
b	12,0	11,6	12,5	12,4	13,6	13,8	11,3	11,3
o_	12,0	11,9	9,1	15,8	12,2	12,4	8,0	12,4
ě	12,0	12,9	7,4	12,4	11,2	13,4	13,7	8,3
í_	12,0	13,3	22,3	13,4	10,9	10,2	17,0	11,3
,	11,0	11,3	12,9	7,9	9,5	14,2	11,3	10,0
,_	11,0	11,3	12,9	7,9	9,5	14,2	11,3	10,0
_,	11,0	11,3	12,9	7,9	9,5	14,2	11,3	10,0

Tabulka 1.4: Relativní četnosti (v promile) n-gramů písmen pro různé části korpusu Desam.



Obrázek 1.1: Poměr token/type v závislosti na velikosti textu v korpusu CNK25.



Obrázek 1.2: Poměr token/type v závislosti na velikosti textu v korpusech Brown, Desam a CNK25.

1.6 Značkování (anotace)

Značkovaný (anotovaný) korpus obsahuje kromě vlastního textu i další dodatečné informace, které jsou do korpusu přidávány buď automatickými nástroji, nebo ručním značkováním. Programům pro značkování (zejména gramatické) se často říká *taggery (značkovače)* z anglického *taggers*. Značkování můžeme rozdělit do čtyř druhů:

- Informace o jednotlivých článcích či větších částech textu jsou tzv. *metainformace*. Ty udávají zejména bibliografické informace (kdo je autorem daného článku, kdy byl text publikován apod.) a případné informace o způsobu značkování.
- Informace o struktuře textu. Na této úrovni zaznamenáváme mimo jiné logickou strukturu textu (kapitoly, odstavce, nadpisy apod.) a typografický vzhled (řádkový a stránkový zlom, zvýraznění pomocí různých typů písma apod.)
- Jazykové jevy na úrovni jednotlivých slov, například slovní druh, základní tvar slova, popis významu.
- Jazykové jevy na úrovni vět, například vyznačení vztahů či závislostí mezi jednotlivými slovy.

Každá z úrovní je dále podrobněji rozebrána.

1.6.1 Metainformace

Metainformace jsou data, která nejsou přímo v textu, ale jsou o textu známa a v některých kontextech mohou být důležitá nebo přinejmenším zajímavá. Slouží také k výběru specifických textů podle zvolených kritérií. Celkový seznam možných atributů se liší korpus od korpusu (nebo pracoviště od pracoviště), jako příklad uvádím seznam metainformací používaný na Fakultě informatiky Masarykovy univerzity (FI MU) v Brně (podrobný popis s příklady i formální definicí formátu zápisu lze nalézt v [Fil00]):

- název dokumentu (například titulek článku nebo kapitoly knihy),
- šiřitelnost – možná omezení při zveřejnění (některé texty jsou například do korpusu vkládány pod podmínkou, že části textů nebudou nikdy zveřejněny jinak než formou krátkých citací)
- popis zdroje (nakladatelství, adresa na Internetu, . . .),
- typ dokumentu, jak byl uveden ve zdroji (typicky rubrika),

- typ obsahu (přírodní vědy, humanitní vědy, sport, dění ve společnosti, literární díla, ekonomie/státní správa),
- upřesnění typu obsahu (asi 50 subkategorií),
- stylová forma (učebnice, slovník, populární styl, reklamní text, . . .),
- médium (kniha, časopis, noviny, scénář, Internet, . . .),
- jméno autora,
- pohlaví autora,
- jméno překladatele,
- pohlaví překladatele,
- jazyk,
- jazyk, ze kterého bylo překládáno,
- datum první publikace,
- jméno autora transformace (různé převody mezi kódováním nebo formátem textu) či anotace (ruční nebo automatické značkování),
- datum transformace/anotace,
- popis transformace/anotace.

Ne všechny atributy jsou povinné, tedy musejí mít určenou hodnotu u všech textů. (Pro případy, kdy nejsme schopni specifikovat hodnotu u povinného parametru, existují univerzální hodnoty typu *nezjištěné* či *nezjistitelné*.) Poslední tři atributy (informace o transformaci či anotaci) se mohou v popise opakovat, podle toho, jak byl text upravován či doplňován o značkování. Velmi podobná skupina atributů se používá i v případě ČNK nebo BNC [Bur95] a víceméně odpovídá doporučení EAGLES[5] [SB99].

1.6.2 Značkování struktury textu

Také konkrétní značkování struktury textu se různí. Podle značkované skutečnosti se v zásadě dá rozdělit na několik typů:

- hierarchické rozdělení na části: kapitoly, odstavce, věty;

[5]Expert Advisory Group on Language Engineering Standards, viz `http://www.ilc.pi.cnr.it/EAGLES/home.html`

- vyznačení částí textu, které nejsou řádnými větami: nadpisy, podpisy, tabulky;
- vyznačení mimoslovních prvků ve větách: značky, čísla, slova cizího jazyka;
- typografické informace: změna typu písma, mezery, zlom.

Značkování je realizováno pomocí *strukturních značek* (někdy též jen krátce *struktur*). Jako konkrétní příklad uvedeme seznam strukturních značek opět ze systému na FI MU [Fil00]:

caption – průběžný titulek, popisky (nepatří do textu);

doc – hranice dokumentů;

head – nadpis;

list – seznam;

item – odrážka v seznamu;

note – poznámky (pod čarou);

p – odstavec;

q – uvození (uvozovky, apostrofy), zvýraznění (typ písma);

s – věta, souvětí;

sign – podpis;

lang – jazyk jiný než čeština;

pre – nahrazuje větší vložené netextové objekty typu obrázek, tabulka (číselná), graf, zdrojový kód;

code – krátké části kódu, počítačové adresy apod.;

corr – korekce v původním textu (zjevná, nechtěná chyba v textu je opravena a původní slova jsou označena touto značkou);

table – tabulka – strukturované textové údaje, které netvoří věty;

g (glue) – pozice kolem této značky nejsou odděleny mezerou.

U některých značek má smysl zaznamenávat kromě jejich umístění v textu i další informace. Například u změny jazyka (značka **lang**) je dobré vědět, v jakém jazyce jsou vyznačená slova. U čísel se zase může hodit znát číselnou hodnotu nezávislou na konkrétním zápise. Těmto informacím říkáme *atributy struktur*.

Základní tvar slov (lemma) v češtině

O lemmatu má smysl mluvit pouze pro ohebné slovní druhy. Pro substantiva volíme první pád (nominativ) jednotného čísla. Pokud takový tvar neexistuje, například pro slova pomnožná (*dveře*), vybereme nominativ plurálu. U adjektiv ze všech nominativů v singuláru volíme mužský rod a první stupeň. Problém mohou tvořit přivlastňovací adjektiva (*otcovy*), pro která můžeme volit příslušné substantivum (*otec*) místo zájmena (*otcův*).

Stejně volíme lemma i pro zájmena a číslovky, tedy nominativ singuláru mužského (životného) rodu.

Lemma sloves je infinitiv. V češtině ovšem běžně máme pro jedno sloveso více infinitivů: pravidelně končí na *-t* nebo *-ti*; některá slova jako *sejmout, snít* mají čtyři i více infinitivů. Musíme tedy jasně definovat, který tvar budeme považovat za základní.

Podobně jako přivlastňovací zájmena jsou odvozena od substantiv, odvozujeme od sloves (*dělat*) adjektiva (*dělaný, dělající*) a substantiva (*dělání*); od adjektiv (*rychlý*) adverbia (*rychle*). Ve všech těchto případech můžeme volit lemma stejného slovního druhu, nebo původní slovo, od kterého byl tvar odvozen. Další komplikací, kde základní tvar nemusí být zřejmý, jsou dublety (*myslit/myslet, gymnázium/gymnasium*).

Celkově tedy musí každý korpus z vyznačenými lemmaty či lemmatizátor (program, který slovním tvarům určuje lemmata) doprovázet příručka popisující všechny možné varianty. V různých aplikacích můžeme pro vybrané situace volit jiná lemmata.

1.6.3 Značkování jazykových jevů na úrovni slov

Vzhledem k tomu, že korpusy slouží zejména ke zkoumání jazyka, patří značkování jazykových jevů k nejdůležitějším. Základní (bylo použito již u prvního korpusu – Brown [FK79]) je přiřazení gramatické značky každému slovu. Značka určuje především slovní druh, někdy i další gramatické kategorie (rod, číslo, pád, osobu atd.). Zejména pro flektivní jazyky (například češtinu) je neméně důležité vyznačení základního tvaru (lemmatu) u každého slova, například prvního pádu jednotného čísla u jmen, infinitivu u sloves. Základní tvar slova je intuitivně zcela jasný a jeho definice (třeba právě uvedená) se může zdát jednoznačná. Bohužel to pro češtinu tak jednoduché není – více viz textový rám *Základní tvar slov v češtině*.

Gramatické značky i základní tvary se váží na jednotlivá slova, v korpusu je tedy můžeme přiřadit jednotlivým pozicím. Zavedeme tedy pojem *poziční atribut*. Korpus tak obsahuje několik pozičních atributů a na každé pozici existuje pro každý poziční atribut nějaká hodnota. Toto zobecnění vynucuje definovat například základní tvar i pro neslovní pozice, ale v takových případech můžeme použít nějakou speciální či „prázdnou“ hodnotu. Pro čísla lze jako základní tvar uvádět hodnotu čísla v nějakém normálním tvaru.

Systémy gramatických značek

Konkrétní gramatické značky se v různých korpusech velice liší. Je to dáno jednak použitím daného korpusu, ale hlavně jazykem. Například pro flektivní jazyky (češtinu) je používáno více různých značek než třeba pro angličtinu. Systémy značek se v čase vyvíjejí, a tak například BNC používá dva různé systémy (starší a novější), přičemž oba vznikly na stejném pracovišti.

Formálně značku tvoří řetězec písmen a znaků, který má nějakou vnitřní strukturu, ze které lze dekomponovat hodnoty všech požadovaných kategorií.

Zcela základní gramatickou kategorií je *slovní druh (part of speech, POS)*. Ten se vyskytuje ve všech systémech a i hodnoty slovního druhu jsou většinou u jednotlivých jazyků stejné[6]. Pro jednotlivé slovní druhy potom existují skupiny dalších kategorií. Například pro podstatná jména určujeme pád, číslo a rod; pro přídavná jména ještě navíc stupeň; pro slovesa osobu, rod, číslo, čas, vid atd.

Jako názorný příklad je na obrázku 1.3 zobrazena označkovaná část korpusu KOČKA ve dvou systémech značek: „Pražském“ (3. sloupec) a „Brněnském“ (4. sloupec). Pražský je založen na Hajičově lemmatizátoru, brněnský na Ševečkově lemmatizátoru. I když oba systémy používají 11 stejných základních atributů, jednotlivé hodnoty se někdy mírně liší a bohužel nejsou mezi sebou automaticky převoditelné bez rozsáhlého slovníku výjimek. Jejich stručný popis je uveden v přílohách A a B. Z technického pohledu je pražský systém založen na pozičním systému, kdy každá značka je vždy dlouhá 15 znaků, kterým odpovídají různé atributy. Brněnský naopak používá pro každý atribut dva znaky, první (malé písmeno) reprezentuje určitý atribut, druhý (velké písmeno nebo číslice) určitou hodnotu. Pražský systém je v některých směrech podrobnější, a tak obsahuje více různých značek (asi 1800) než brněnský (asi 1250). Dále v práci uváděné příklady používají brněnský systém značek (příloha B).

[6] Až na detaily, jako že v češtině nemáme žádné členy.

Na	na	RR--6----------	k7c6
okně	okno	NNNS6-----A----	k1gNnSc6
seděla	sedět	VpQW---XR-AA---	k5eApFnStMmPaI
kočka	kočka	NNFS1-----A----	k1gFnSc1
,		Z:-------------	
byl	být	VpYS---XR-AA---	k5eApInStMmPaI
horký	horký	AAIS1----1A----	k2eAgInSc1d1
letní	letní	AAIS1----1A----	k2eAgInSc1d1
den	den	NNIS1-----A----	k1gInSc1
,		Z:-------------	
na	na	RR--6----------	k7c6
okně	okno	NNNS6-----A----	k1gNnSc6
seděla	sedět	VpQW---XR-AA---	k5eApFnStMmPaI
kočka	kočka	NNFS1-----A----	k1gFnSc1
a	a	J/-------------	k8xC
koukala	koukat	VpQW---XR-AA---	k5eApFnStMmPaI
se	sebe	P7-X4----------	k3xXnSc4
ven	ven	Db-------------	k6xLeAd1
,		Z:-------------	
byl	být	VpYS---XR-AA---	k5eApInStMmPaI
horký	horký	AAIS1----1A----	k2eAgInSc1d1
letní	letní	AAIS1----1A----	k2eAgInSc1d1
den	den	NNIS1-----A----	k1gInSc1
a	a	J/-------------	k8xC
kdekdo	kdekdo	PZM-1----------	k3xUgPnSc1
chodil	chodit	VpYS---XR-AA---	k5eApMnStMmPaI
bos	bos	AAIS4----1A----	k2eAgMnSc1d1
,		Z:-------------	
na	na	RR--6----------	k7c6
okně	okno	NNNS6-----A----	k1gNnSc6
seděla	sedět	VpQW---XR-AA---	k5eApFnStMmPaI
kočka	kočka	NNFS1-----A----	k1gFnSc1
,		Z:-------------	
venku	venku	Db-------------	k6xLeAd1
zpíval	zpívat	VpYS---XR-AA---	k5eApMnStMmPaI
kos	kos	NNMS1-----A----	k1gMnSc1
.		Z:-------------	

Obrázek 1.3: Příklad textu s gramatickými značkami.

Pro angličtinu existuje celá řada systémů gramatických značek, každý z uváděných anglických korpusů používá trochu jiný systém a mnoho dalších jich existuje na jiných pracovištích. Není proto divu, že se objevila snaha provést jistou standardizaci systémů značek, aby bylo alespoň dosaženo jednoznačné převoditelnosti mezi různými systémy. Tato snaha byla realizována v rámci sdružení EAGLES a výsledkem je doporučení [LW96]. Tam jsou popsány jednotlivé kategorie spolu s příslušnými hodnotami, které jsou rozděleny na povinné, doporučené a speciální. Také je definován jeden konkrétní formát, do/z kterého by každý systém značek měl být převoditelný. Tím by byla zajištěna převoditelnost mezi všemi systémy, které odpovídají tomuto doporučení.

Negramatické značky

Dalšími značkami na úrovni slov bývají syntaktické a sémantické značky. I když značení syntaxe vyjadřuje strukturu věty (viz další podkapitolu), někdy se pro jednoduchost alespoň u některých slov udává jejich syntaktická role. Určení takových značek může být mnohem jednodušší než zaznamenání celé syntaxe věty, a navíc není potřeba pro jejich uložení používat složitější aparáty pro strukturní značky.

Sémantickou značkou může být jednak určení konkrétního významu daného slova (například podle slovníku) – tedy jisté doplnění základního tvaru o pořadové číslo nebo slovní vysvětlení významu, nebo může jít o zaznamenání jistých sémantických atributů nebo kategorií (viz např. [Pal99]).

1.6.4 Značkování jazykových jevů na úrovni vět

Na úrovni vět jde o značkování zejména syntaktické struktury, tedy zaznamenání jakýchsi stromů (někdy dokonce obecných acyklických grafů) nad pozicemi. Rozeznáváme dva odlišné typy syntaktických stromů:

Závislostní (dependency structure), který zaznamenává závislost jednotlivých slov mezi sebou. Z každého slova ukazuje hrana stromu na nadřazené slovo (nebo interpunkci). Hrany většinou ještě nesou informaci o typu podřízenosti daného slova.

Složkové (phrase structure), které slova sdružují do skupin. Slova jsou tedy pouze v listech stromu a další uzly tvoří jakési virtuální uzly označující stále větší skupiny až celou větu. Složkové stromy většinou vznikají jako derivační stromy při analýze věty pomocí bezkontextové gramatiky, vnitřní uzly stromu potom odpovídají neterminálům v gramatice.

Oba typy jsou mezi sebou převoditelné, i když takový převod není vůbec triviální.

Často bývá určení jednoznačné (a správné) syntaktické struktury věty příliš obtížným úkolem. Proto se někdy v korpusu nevyznačuje celá syntaxe věty, ale označí se pouze některé slovní skupiny (jmenné, slovesné), které je možné bez větších problémů analyzovat. Pak se mluví o *částečné* syntaktické analýze.

1.6.5 Víceznačnost

Značkování není jednoduchá procedura. Pokud je prováděno ručně, je velice nákladné a nevyhneme se jistému procentu chyb, které i při sebevětší pečlivosti lidští anotátoři udělají. Větší korpusy (desítky až stovky milionů pozic) prakticky nelze značkovat ručně, je nutno použít některou z automatických metod.

V závislosti na zvoleném jevu jsou automatické značkovače různě úspěšné, většinou ovšem nedokáží rozhodnou všechny případy zcela správně a jednoznačně. Pokud rozhodují vždy jednoznačně, někdy ale chybně, a přitom nejsme schopni tyto chyby detekovat, nezbývá nám než počítat s jistou chybovostí (naměřenou například na ručně označkované části korpusu). Zajímavější je případ, kdy značkovací program prakticky nedělá chyby, zato ale v některých situacích nerozhodne úplně, ale dá na výběr několik možností. Vzniká tedy *víceznačnost* v korpusu.

Víceznačnost se může projevovat ve všech dříve popsaných typech značkování. Asi nejčastějším značkováním vůbec, je gramatické značkování na úrovni slov, na němž staví mnoho jiných automatických značkovačů (syntaxe, sémantika atd.). V angličtině mají tyto značkovače úspěšnost 97 i více procent, tedy pro maximálně 3 % slov je určena špatná značka. V češtině je situace silně závislá na použitém systému značek (viz např. [PRS98]). Pro systémy s menším počtem značek přirozeně dosahujeme lepších výsledků, ale vhodnou volbou speciálních značek mohou značkovače i pro rozsáhlé systémy pracovat úspěšně.

Víceznačnost lze odstranit několika způsoby. Nejjednodušší je „definování“, že se o víceznačnost nejedná. Například některé systémy gramatických značek pro angličtinu obsahují několik málo speciálních značek, které reprezentují kombinaci dvou jiných. V češtině zase pražský systém značek (příloha A) udává například pro slovo *seděla* jedinou (a tedy jednoznačnou) značku `VpQW---XR-AA---`, ve které speciální hodnoty *Q* a *W* pro rod a číslo znamenají, že jde o singulár ženského rodu nebo plurál středního.

Další možností je ruční *zjednoznačnění (desambiguace)*, tím fakticky můžeme zvětšit množství textu, které jsou schopni anotátoři zpracovat ručně. (Takto například vznikal korpus Desam.) Poslední možností je použití nějaké automatické metody na zjednoznačňování. Tím vlastně vytvoříme kombinovanou automatickou metodu značkování, která opět končí buď jistou chybovostí, nebo víceznačností.

Automatické metody desambiguace většinou potřebují nějakou trénovací množinu již označkovaných textů (korpus), na které se použitý systém „naučí“ správně rozhodovat. Oblíbeným typem automatických metod je statistická (pravděpodobnostní) desambiguace, v níž se zpracovávají četnosti jednotlivých značek v kombinaci z četnostmi jejich bigramů a trigramů (pro češtinu například popsáno v [PRS97]. Modifikace pro flektivní jazyky, ve které se zpracovávají samostatně jednotlivé atributy místo celých značek je uvedena v [HH98]. Další typy metod zahrnují téměř všechny metody strojového učení.

1.7 Paralelní korpusy

Paralelní korpus obsahuje stejný text v několika (alespoň dvou) různých jazycích, přičemž sobě odpovídající si části (odstavce, věty, slovní skupiny apod.) jsou nějakým způsobem „zarovnány“, pro každou takovou část je jednoznačně určeno, ke které části textu jiného jazyka patří. Texty pro paralelní korpus vznikají nejčastěji překladem z jednoho jazyka do druhého, v některých případech však může jít o dva různé překlady z jiného jazyka, který není v korpusu uveden. Klasickým příkladem jsou paralelní korpusy obsahující části Bible.

Na paralelní korpusy, a tím i na korpusové manažery pro paralelní korpusy, se můžeme dívat ze dvou úhlů:

1. Paralelní korpusy jsou od běžných korpusů zásadně odlišné (zpracovává se najednou dva nebo více různých jazyků), uživatelé je používají k jiným účelům a pracují pomocí jiných metod a prostředků. Funkce používané v nástrojích pro jednojazyčné korpusy jsou zde sice také uplatňovány, ale až sekundárně. Jsou to tedy speciality, pro které je potřeba specializované nástroje, které se od korpusových manažerů mohou podstatně lišit.

2. Každý z jazyků paralelního korpusu můžeme brát zvlášť jako samostatný korpus. Navíc musíme v korpusech zaznamenat zarovnání, které musí být transparentně zařazeno i do dotazovacího jazyka a všech metod vizualizace výsledku.

Současný stav nástrojů pro paralelní korpusy odpovídá spíše prvnímu úhlu pohledu: pouze malé množství korpusových manažerů nějakým způsobem podporuje paralelní korpusy a většinou velice omezeně. Přitom jistým zobecněním zarovnání lze získat aparát použitelný i v jednojazyčných korpusech.

My zde popíšeme druhý přístup, tedy rozšíření korpusového manažeru o možnosti zpracování paralelních korpusů. Pokud paralelní korpus obsahuje texty více než dvou jazyků, uvažujeme je vždy po dvojicích. Dále tedy předpokládáme zarovnání právě dvou jazykových verzí.

Typy zarovnání

V principu rozlišujeme dva typy zarovnání:

- Zarovnání *souvislých částí textu*: odstavců, vět, souvislých slovních skupin apod.
- Zarovnání *jednotlivých slov (pozic)* – jde tedy o jistou formu *ukazatelů*, které k sobě spojují vzájemné překlady.

Zarovnání slov lze realizovat i pomocí prvního typu, stačí vyznačit příslušná slova jako skupiny.

Při zarovnávání dvou textů se snažíme o nalezení vztahů *1:1*, kdy například jedné větě odpovídá zase jedna věta. V praxi ovšem existují případy (zvláště při zpracování velice odlišných jazyků), kdy jedné větě z prvního jazyka odpovídá dvě nebo více vět z druhého jazyka. Obecně tedy máme i vztahy *1:n* či *n:1*.

1.8 Formy uložení korpusu

Technicky může mít uložení korpusu v počítači různou podobu. „Nejvolnější“ formou je pouhý *archiv (kolekce) textů* v různých formátech a kódováních podle toho, z jakého zdroje text pochází. Organizovanější jsou *textové banky*, ve kterých jsou texty uloženy v jednotném formátu a je na nich provedeno základní značkování – rozdělení textu na jednotlivé články (dokumenty), určení typu zdroje pro každý článek apod. Konečnou formou uložení je použití nějakého *korpusového manažeru*, který texty zakóduje do určité databáze a uživateli umožňuje prohlížení korpusu z různých úhlů.

Pro zpracování textů mimo vlastní korpusový manažer se nejčastěji používají dva formáty: tzv. *vertikální text* a SGML. Oběma formátům je dále věnována samostatná kapitolka. Další kapitolka je věnována způsobům kódování jednotlivých znaků, což je společný problém obou formátů.

1.8.1 Vertikální text

Vertikální text je běžný textový soubor, ve kterém je již vyznačeno rozdělení do pozic tím, že se každá pozice nachází na samostatném řádku. Slova jsou zde tedy uvedena pod sebou – vertikálně, odtud pochází název. Pokud má text obsahovat více pozičních atributů (například základní tvar a gramatickou značku, viz dále) jsou uvedeny všechny atributy na jednom řádku a od sebe odděleny znakem tabulátoru. Strukturní značky jsou vyznačeny ve stylu SGML, ale každá značka je na samostatném řádku.

Výhodou tohoto formátu je jeho jednoduchost. Je čitelný ve všech textových editorech a prohlížečích, snad ve všech programovacích jazycích je velice snadné formát zpracovávat (stačí číst vstup po řádcích). Mnoho základních operací s takovým textem i výpočet různých statistik lze provádět za použití standardních programů v UNIXu (`grep`, `cut`, `sort`, `uniq`, atd.)

Mnoho korpusových manažerů umožňuje (někdy i vyžaduje) v tomto formátu zadávat text k zakódování (zpracování) do interní podoby. Jedná se tedy o jistý *import* dat. Některé manažery poskytují i opačnou funkci – *export* dat. Dokáží celý korpus nebo jeho část zapsat do souboru ve formě vertikálního textu. Část korpusu je možné zadat například intervalem pozic nebo identifikací dokumentů v korpusu. Formát vertikálního textu, zejména použití více atributů než pouhé slovo nebo použití strukturních značek, není oficiálně nijak standardizován. Důvodem je hlavně rozdílnost ve schopnostech manažerů tyto informace zpracovávat.

1.8.2 SGML

SGML znamená *Standard Generalised Markup Language*, tedy standardní zobecněný jazyk pro značkování. Jedná se o mezinárodní standard ISO 8879 :1986(E) [ISO86], který mimo jiné popisuje metody, jak specifikovat aplikačně nezávislou strukturu dokumentu. Struktura se definuje pomocí tzv. *elementů (elements)*, u nichž je vždy definováno, jaké jiné elementy může daný element obsahovat. Také lze určit pořadí jednotlivých elementů, jejich volitelnost či opakovatelnost.

Elementy jsou v textu většinou vyznačeny počáteční a koncovou *značkou*. Tu tvoří jméno elementu uzavřené do úhlových závorek (< a >), v případě koncové značky je před jméno elementu ještě vložen znak lomítka (`/`). Například nadpis značený elementem `head` bude začínat značkou `<head>` a končit značkou `</head>`. Každý element může mít definovány tzv. *atributy (attributes)*, kterým se potom v textu přiřazují hodnoty například pro kategorizování jednotlivých elementů. Pokud chceme u elementu zapsat hodnotu některého z atributů, vložíme dvojici [atribut, hodnota] do počáteční značky zvoleného elementu, například `<head type=sub>`.

Texty ve formátu SGML mohou obsahovat pouze omezený rozsah znaků: jsou dovolena pouze velká a malá písmena anglické abecedy, číslice a některé interpunkční znaky. Všechny ostatní znaky se zapisují pomocí tzv. *entit (entity references)*, které začínají znakem ampersand (`&`), následuje mnemonická zkratka daného znaku a jsou ukončeny středníkem (`;`). Například české *é* se zapíše jako `é`.

Každý dokument v SGML musí odpovídat jisté definici, která je většinou uvedena samostatně ve formě tzv. *DTD (Document Type Definition)*, ve kterém jsou uvedeny všechny přípustné elementy i se svými atributy (popř. s omezením hodnot jednotlivých atributů) a jejich struktura a všechny povolené entity. Mnoho DTD je standardizováno, například jazyk HTML (Hypertext Markup Language), používaný pro zápis webových stránek na Internetu, je aplikací SGML a každá z jeho standardizovaných verzí má svoje DTD [W3C98a].

SGML je poměrně rozsáhlý formát (podrobnější výklad SGML lze nalézt například v [GS95], [vH95] nebo přímo ve specifikaci standardu [ISO86]) a poskytuje značnou volnost pro zápis textů vyhovujícím zvolenému DTD. Pro jeho zpracování je tedy nutné používat tzv. *SGML parsery*, tedy programy, které čtou text v SGML, kontrolují, zda svojí strukturou odpovídá zadanému DTD a na výstup dávají jednotlivé části textu: počáteční/koncové značky, atributy značek s hodnotami, vlastní text. Pro tento výstup již existují knihovny do všech nejpoužívanějších programovacích jazyků, které poskytují programátorovi relativně příjemné rozhraní.

Nevýhodou volnosti zápisu v SGML je, že i pro poměrně jednoduché věci je potřeba psát samostatné programy, které využívají nějaký SGML parser. Rychlost běhu těchto programů je samozřejmě limitována parserem a často bývá nízká. Výhodou této volnosti je naopak snadnost převodu textů z jiných zdrojů do SGML.

Pro formát SGML existuje celá řada standardních typů značkování, pro která jsou dostupná již hotová DTD. Pro značkování textů přirozeného jazyka je nejvýznamnějším standardem TEI (Text Encoding Iniciative) Guidelines [SMB94] vytvořený několika mezinárodními asociacemi. Zvláště pro potřebu kódování korpusů byla několika organizacemi vybrána podmnožina TEI s jistými úpravami a vznikl Corpus Encoding Standard (CES) [CES96].

Někdy je zvoleno kompromisní řešení, kdy je text uložen ve formátu SGML, ale navíc musí splňovat další podmínky. Tento postup byl zvolen například v Hajičově skupině a převzat v ÚČNK. Podle jejich specifikace musí být texty v SGML a zároveň musí být každá pozice na samostatném řádku.

XML

Jistou novinkou v oblasti SGML je nový značkovací jazyk XML (Extensible Markup Language). Vznikl jako reakce na velkou složitost SGML (a naopak přílišnou jednoduchost masově rozšířeného HTML, který již přestal pro některé aplikace na Internetu dostačovat [7]). XML je podmnožinou SGML, která by měla postačovat k značkování všech možných textů i k vytváření textových databází. Textové databáze jsou běžné textové (ASCII, viz dále) soubory, které ovšem neobsahují plynulý text, ale jistým způsobem strukturovaná data. Hlavní snahou při návrhu standardu XML [W3C98b] bylo zvýšení praktické použitelnosti. Při použití SGML nebyla mnohdy jednoduchá pouhá korektní instalace a konfigurace systémů pro použití nějakého SGML parseru. Při použití XML dokonce v nejjednodušších případech ani nepotřebujeme DTD, XML parser tedy stačí pouze spustit (bez jakékoliv předchozí konfigurace). Jako jistý důkaz větší jednoduchosti XML proti SGML může být mnohem menší počet stránek, na kterých je jazyk oficiálně definován (32 stránek pro XML, 155 stránek pro SGML).

Přes své nedlouhé trvání (standard byl zveřejněn teprve v roce 1998) se formát velice rychle začíná používat a stále více aplikací, které dříve používaly SGML, je převáděno na použití XML, mimo jiné bude brzy uvedena první specifikace XHTML, což je náhrada HTML používající XML. Výjimkou není ani TEI a CES, pro něž bylo ohlášeno, že se pracuje na převedení příslušných DTD podle XML. Pro XML existuje mnohem více parserů a jsou přirozeně díky jednoduchosti XML menší a rychlejší.

Standard XML je také mnohem otevřenější: je spravován organizací W3C (World Wide Web Consortium [W3C94]), což je mezinárodní volné sdružení (kdokoliv se může stát členem po zaplacení poplatků) organizací, firem a institucí vzniklé v roce 1994, které se snaží definovat standardy v oblasti WWW a všemi možnými způsoby přispívat k rozvoji WWW. Na svých webovských stránkách prezentuje informace pro vývojáře i uživatele WWW, vytváří prototypy programů a aplikací jako demonstrace nových technologií. Na těchto stránkách je mimo jiné uveden i text definice XML, naproti tomu za specifikaci SGML je nutné zaplatit přes 200 švýcarských franků.

Celkově v tomto směru vidíme budoucnost právě v XML a zpracování (import) textů ve formátu XML jistě zvyšuje použitelnost každého korpusového manažeru.

[7] Aplikace XML na internetu byly určitě hlavním hnacím motorem rychlého vývoje XML.

1.8.3 Kódování znaků

Samostatnou oblastí je kódování jednotlivých znaků v korpusu či vertikálním textu. Nejčastějším kódováním používaným v současných počítačích a počítačových sítích je ASCII (American Standard Code for Information Exchange) z roku 1968 definující osmibitový kód. Pro každé číslo z rozsahu 0–127 je definován příslušný znak (písmeno, číslo, symbol, kontrolní znak), kódy 128–256 jsou vyhrazeny pro lokální použití. Definovány jsou kódy pro všechna písmena anglické abecedy a pro jazyky, které potřebují více znaků, existují rozšíření ASCII využívající druhé poloviny tabulky. Bohužel těchto kódování je pro každý jazyk několik, jak postupně vznikala v jednotlivých systémech různých výrobců. Pro češtinu je situace popsána v textovém rámu na straně 13.

Pokud se v textech jednoho jazyka vyskytují slova obsahující „cizí" znaky, musíme použít kódování, které všechny potřebné znaky obsahuje. Příkladem takových kódování jsou znakové sady podle norem ISO 8859-1 až 8859-16 [ISO99], které definují šestnáct tabulek znaků – každou pro jinou geografickou oblast. Například 8859-1 (označovaná Latin-1) pokrývá jazyky západní Evropy, 8859-2 (Latin-2) pokrývá jazyky střední a východní Evropy. V korpusu ovšem můžeme mít slova (nejčastěji jména osob), která obsahují znaky ze zcela odlišných regionů. Pak musíme použít jiný aparát podobný entitám ze SGML. To ale jakékoliv zpracování značně komplikuje.

Samozřejmě pro jazyky, které mají více než zhruba 180 znaků, nelze ASCII použít a pro některé východní jazyky, které mají znaků i několik tisíc, nelze použít žádné osmibitové kódování. Použití SGML entit je v takových případech zcela nepraktické, protože téměř každý znak je tvořen entitou, která zabírá až deset bytů.

Unicode

Řešením je použití vícebytového kódování, které by obsahovalo všechny myslitelné znaky. Existují dva standardy pro takové univerzální kódování, které se naštěstí v roce 1991 spojily. *Unicode* (`www.unicode.org`) spravuje Unicode Technical Committee, ISO/IEC 10646-1 (známé jako Universal Character Set, UCS) spravuje International Organization for Standardization. Oba standardy definují šestnáctibitový kód (s více než 65 000 různými znaky), který je možné ještě rozšířit na třicetidvoubitový s více než milionem znaků. Unicode navíc definuje speciální kódování UTF-8, které má proměnný počet bytů na jeden znak a je kompatibilní s ASCII, tedy každý text obsahující znaky pouze z první poloviny tabulky ASCII je správným zápisem kódování Unicode ve formátu UTF-8.

Standard Unicode byl převzat i pro XML a novější operační systémy a aplikace ho též podporují. Pro korpus pokládáme Unicode za vhodné řešení.

Kapitola 2

Korpusový manažer

V této kapitole je uvedena analýza a specifikace požadavků korpusového manažeru. Poslední podkapitola na straně 67 stručně (bodově) všechny požadavky shrnuje.

2.1 Logická struktura korpusu

Korpus budeme pro účely korpusového manažeru tedy chápat jako *posloupnost pozic*. Počet pozic by neměl být nijak omezen: malé korpusy obsahující pouze jeden text mohou mít pouze několik tisíc pozic, velké korpusy, které se snaží reprezentovat celý určitý jazyk, naopak obsahují minimálně několik stovek milionů pozic, ale v blízké budoucnosti (několika let) mohou obsahovat i desítky miliard pozic. Uživatel musí mít možnost určit podle svých potřeb vlastní rozdělení do pozic, i když většinou vystačí s jednoduchým rozdělením na slova podle mezer a interpunkce.

Pro každý korpus definujeme množinu *pozičních atributů*, pro každý z nich obsahuje korpus na každé pozici nějakou hodnotu. Korpus musí mít alespoň jeden atribut s vlastními slovy ze zdrojového textu korpusu, dále podle druhu a hloubky značkování obsahuje i další poziční atributy. Počet pozičních atributů v jednom korpusu by také neměl být nijak omezen, s některými korpusy často pracuje větší počet uživatelů, kteří mají rozdílné nároky na další poziční atributy.

Korpus může obsahovat značkování ve formě *struktur*. Struktury netvoří pozice, jsou jaksi nad nimi. Pro každou strukturu v korpusu jsou definovány intervaly pozic, které struktura „pokrývá". Některé struktury navíc mohou obsahovat množinu *atributů struktur*, které jsou analogií pozičních atributů. Každý interval potom musí obsahovat hodnotu pro každý takový atribut. Ani počet různých struktur, resp. atributů struktur, by neměl být nijak limitován.

Paralelní korpusy jsou tvořeny samostatnými korpusy pro každý z jazyků. Jednotlivé korpusy navíc obsahují pro všechny ostatní jazyky *zarovnání* určující sobě si odpovídající části textu. Nejčastěji jsou zarovnány texty/překlady pouze dvou jazyků, ale v době, kdy se Evropská unie rozšiřuje o stále další státy se svými vlastními národními jazyky, nás nemusí překvapit paralelní korpusy s desítkami různých překladů.

2.2 Typy atributů a struktur

Věnujme nyní pozornost různým typům pozičních atributů a struktur.

2.2.1 Atributy

Povinným atributem je atribut uchovávající vlastní slova textu. Jedná se tedy o řetězec znaků bez jakékoliv vnitřní struktury. Jedinou podmínkou je potřeba reprezentovat všechny možné znaky, které se v textu vyskytly (viz kap. 1.8.3).

Podobně pro zachycení základního tvaru nemusíme chtít nic víc než znakový řetězec. Pokud jsou ale v lemmatu zakódovány například i významy, můžeme alespoň některé hodnoty tohoto atributu rozdělit na dvě části: základní tvar a jeho bližší určení. Například u víceznačného slova *jeřáb* použijeme `jeřáb-1` nebo `jeřáb-stroj` v jednom významu a `jeřáb-3` nebo `jeřáb-strom` v jiném.

Mnohem složitější strukturu tvoří některé systémy gramatických značek (viz kap. 1.6.3) a korpusový manažer prakticky nemůže počítat se všemi potenciálními možnostmi. Pro pohodlnou práci s různými značkami potřebuje uživatel dvě operace:

- Zobrazení značky v rozšířené/textové podobě. Například místo značky `PNP` (z anglických korpusů) zobrazovat *personal pronoun*, nebo *osobní zájmeno*.

- Zadávání značky v dotazu. Pokud uživatel (zvláště začátečník) tvoří dotaz pomocí grafického rozhraní (viz kap. 2.7.4), potřebuje si vybírat z plných popisů, a ne z kryptických kódů.

Dynamické atributy

Zobrazení jiné podoby značek můžeme realizovat pomocí *dynamických atributů*. To jsou atributy, které nejsou přímo uloženy v korpusu, ale jejich hodnoty se dynamicky vytvářejí z jiných atributů na stejné pozici. Technicky je můžeme popsat jako funkce zobrazující hodnoty daných atributů na hodnoty nějakých „virtuálních" atributů. Pro vytvoření textového popisu značky použijeme funkci, která převede (například podle připravené tabulky) krátký kód značky na delší popis.

Je jasné, že dynamické atributy (jejich převodní funkce) nemohou být v korpusovém manažeru předem připraveny, ale uživatel (nebo alespoň správce korpusů) si je definuje v závislosti na použitých systémech značek.

Dynamické atributy můžeme použít i k jiným účelům. Například můžeme definovat atribut, jehož hodnoty jsou délky slov či celkové četnosti slov v korpusu. Ty potom použijeme při třídění nebo počítání statistik. Funkce tedy nemusí vracet pouze znakový řetězec, ale i číslo (pravděpodobně se můžeme vždy omezit pouze na čísla celá).

Dalším možným použitím je „vytažení " jedné atomické hodnoty z nějaké komplexní značky. Například gramatická značka pro jména většinou obsahuje číslo (jednotné nebo množné). Pokud nás z celé značky zajímá pouze tato jeho část, použijeme dynamický atribut, který ji ze značky vybere. Takto definované atributy mohou uživatelé chtít použít i v dotazech, na což je potřeba vytvořit dodatečné datové struktury popsané v kapitole 3.4.

Významným rozšířením možností dotazovacího jazyka je použití dynamických atributů k propojení korpusu s externími jazykovými zdroji, jako jsou slovníky či thesaury. Například v dokumentaci manažeru CQP je popisováno napojení korpusu na anglickou lexikální databázi WORDNET [Mil93], která mimo jiné obsahuje hyperonymické vztahy (nadpojem/podpojem). Uživatel potom může klást dotazy na slova, která jsou podpojmem slova *human* apod.

Typově tedy můžeme „běžné" poziční atributy omezit pouze na řetězec znaků (přičemž vnitřní struktura značek se neuvažuje) a případné složitější formy ošetřit pomocí dynamických atributů.

Interpretace hodnot atributů

Při tvorbě dotazů ovšem nevystačíme pouze s dynamickými atributy. Uživatel by měl být veden možnostmi použitého systému značek, který například po zvolení slovního druhu nabídne k zadání další kategorie pouze jemu příslušející. V krajním případě korpusový manažer alespoň upozorní na nekorektně vytvořenou značku, aniž by prováděl vyhodnocování dotazu.

Korpusový manažer (resp. ta jeho část pro asistenci při tvorbě dotazu) může obsahovat deklarativní popis značek pro daný poziční atribut. Takto je problém vyřešen v manažeru CQM. Vzhledem k různorodosti a rozsáhlosti některých systémů značek ovšem uživatelé nemohou nikdy spoléhat pouze na vlastní korpusový manažer a měli by mít k dispozici podrobnou dokumentaci, která vysvětluje použité značky i s příklady použití.

Multihodnoty

Jak bylo uvedeno v kapitole 1.6.5, ne vždy dosáhneme jednoznačných značek. Zvláště pokud použijeme dynamických atributů (například na získání základního tvaru slova z lemmatizátoru pro neoznačkovaný korpus), prakticky nemůžeme provést jakoukoliv desambiguaci, a tak potřebujeme více hodnot jednoho atributu na jedné pozici – *multihodnoty*. Problém lze řešit prostým spojením všech hodnot za sebe a jejich oddělením třeba čárkou. Jakékoli dotazy potom musíme upravit (použitím regulárních výrazů), abychom postihli všechny možné tvary hodnot atributů, což je minimálně nepříjemné. Výpočet statistik nad takovým tvarem hodnot již ale nejsme schopni nijak korigovat a výsledek bude vždy zkreslený.

Korpusový manažer by tedy měl umožnit ukládat a zpracovávat multihodnoty (jinak řečeno množiny hodnot) u pozičních atributů.

2.2.2 Struktury

Struktury můžeme rozdělit na následující typy (značky uvedené v příkladech byly popsány v kap. 1.6.2):

plochá struktura
: Struktury nejsou vnořovány, ani se nijak nepřekrývají. Každá struktura je dána svým začátkem a koncem. Mohou korpus rozdělovat na části, tedy kde jedna struktura končí, tam další začíná – příkladem je **doc**. Nebo se jednotlivé intervaly vyskytují samostatně v textu – příkladem je **head**, **lang**.

stromová struktura
: Struktury jsou také dány začátkem a koncem, ale navíc lze stejné značky vnořovat do sebe. Například **list**, **q**.

prázdná struktura
: Značky tvoří pouhé „oddělovače“. Jsou dány pouze jediným údajem: mezi kterými pozicemi se nachází. Příkladem jsou značky **pre** a **g**.

Pomocí těchto typů dokážeme pokrýt všechny možnosti značkování struktury textu, které obsahuje jazyk SGML.

Teoreticky je prázdná struktura pouze speciálním případem ploché struktury (začátek je roven konci) a plochá struktura je speciálním případem stromové (jednotlivé intervaly se nepřekrývají). Pro korpusový manažer by

tedy stačilo implementovat pouze stromovou strukturu. Ta je ale ve srovnání s předchozími mnohem složitější na implementaci (zvláště s ohledem na rychlost vyhodnocování dotazů), a tak současnými korpusovými manažery není většinou podporována. Implementace prázdné struktury má proti ploché výhodu polovičních paměťových nároků.

S ohledem na efektivitu je tedy vhodné implementovat všechny tři typy struktur.

Atributy struktur

Atributy struktur k jednotlivým intervalům struktur přidávají další informace. Můžeme je brát stejně jako poziční atributy, tedy hodnotami jsou řetězce znaků, resp. množiny řetězců znaků, lze pro ně definovat dynamické atributy. Žádným způsobem nebudeme omezovat počet různých atributů k jedné struktuře. Každá struktura má samozřejmě svoji vlastní skupinu atributů, jejichž jména se případně mohou shodovat.

2.3 Konkordance

Konkordance (nebo někdy též *konkordanční seznam*, anglicky *concordance*) je seznam všech výskytů jistého jevu ve zvoleném korpusu. Jevem může být v nejjednodušším případě určité slovo nebo slovní skupina, ale může jít například i o slovní skupinu ve zvolené syntaktické pozici, které v paralelním korpusu (viz kap. 1.7) odpovídá jiná zvolená slovní skupina.

Jednotlivé výskyty (*occurence, hit*) v konkordanci se nejčastěji zobrazují (na monitoru počítače i vytištěné na papíře) s jistým kontextem, který je bezprostředně předchází resp. následuje. Nejčastějším typem takového zobrazení je formát *KWIC* (Key Word In Context), kdy každému výskytu odpovídá jeden řádek s hledaných jevem zvýrazněným uprostřed. Příklad části konkordance s výskyty slova „korpus“ v korpusu CNK25 ukazuje obrázek 2.1.

Aby uživatel mohl prostudovat jednotlivé výskyty jevu, musí být schopen zobrazit pro daný jev dostatečný kontext. Korpusový manažer by měl umožňovat operativně měnit velikost zobrazeného kontextu, a to v různých jednotkách: v počtu písmen, slov/pozic, struktur. U některých jevů je vhodné zobrazovat větší text pouze na jedné straně nalezeného jevu, má smysl tedy specifikovat samostatně levý a pravý kontext. V některých konkrétních výskytech nám ani zobrazený kontext nemusí stačit, a proto většina korpusových manažerů umožňuje v interaktivním režimu zobrazit i *detail* či *rozšířený kontext* pouze pro zvolený výskyt.

```
jeho idea pouhou šlehačkou na korpus , který se musí nejdřív upéci
      či jiným vědátorům jako korpus delikti proti ničitelům životního
a kopcích okolo města . První korpus bosenské armády , bránící Sarajevo
acovaných esejů , esejistický korpus hodný Montaigne . Stejně jako jeho
       dortu mrazíren , jehož korpus tvoří výroba polotovarů z brambor a
paměti . Jedná se o úctyhodný korpus čítající přes 4 tis . stran
z nich , když u nich prodával korpus Krista pocházející - jak se později
   budovat textový počítačový korpus . Je to soubor počítačově zapsaných
reprezentativní český národní korpus s alespoň 100 milióny běžných slov
    slovníky podávají . Slovo korpus v posledních letech získalo několik
```

Obrázek 2.1: Některé výskyty slova „korpus“ z korpusu CNK25 ve formátu KWIC

Součástí každého výskytu v konkordanci je jeho *reference*, tedy nějaký identifikátor udávající, kde se v korpusu daný výskyt nachází. Nejpřímější formou reference je číslo pozice v korpusu. Nebo to může být pořadové číslo některé struktury (věty, odstavce). Na nejvyšší úrovni může být reference identifikátorem dokumentu (zdroje textu). Někdy je též vhodné kombinovat několik různých způsobů.

Tvorba konkordance: dotaz

Konkordance může vzniknout jistou kombinací jiných konkordancí nebo nějakou úpravou jiné konkordance. Prvotní konkordance ale téměř vždy vznikají jako výsledek dotazu na korpus. Proto je potřeba, aby uživatelé měli k dispozici dostatečně mocný dotazovací jazyk, kterým by byli schopni popsat a následně nalézt všechny pro ně myslitelné jevy. O dotazovacím jazyku se podrobně zmíníme dále.

2.3.1 Úpravy konkordančního seznamu

Ani při použití silného dotazovacího jazyka se nám vždy nepodaří v jednom dotazu zadat všechny své požadavky na hledaný jev. Proto musíme konkordanční seznam dodatečně upravit – odstranit z něj některé výskyty, které daný jev buď neobsahují, nebo jsou jeho nevhodným příkladem. V této podkapitole si ukážeme, jak nám v tom může korpusový manažer pomoci.

Redukce počtu výskytů

Nejjednodušším způsobem odstranění některých výskytů z konkordance je *ruční mazání*, kdy uživatel přímo označí, které řádky se mají z konkordance zrušit, nebo které mají v konkordanci zůstat (a smazat všechny ostatní).

Tímto způsobem například vznikl obrázek 2.1. Takto může uživatel postupovat, pokud je celkový počet výskytů relativně malý – maximálně stovky či tisíce, nebo pokud mu nezáleží na reprezentativnosti výsledku a chce pouze vybrat několik zajímavých ukázek (tak tomu bylo v uvedeném příkladě).

Pro větší množství je potřeba použít nějakou automatickou metodu. Poměrně jednoduché na implementaci jsou automatické redukce podle zadaného počtu výskytů nebo procent (výsledný počet je potom velikost konkordance vynásobená danými procenty). Pro výběr výskytů je možné použít několik metod:

první/poslední/střední – z konkordance se vybere žádaný počet řádků ze začátku/konce/středu konkordance (zde nemusí být zajištěna reprezentativnost výsledku),

rovnoměrně – vybere se první a pak každý n-tý, kde n je podíl původního a žádaného počtu řádků,

náhodně – výběr se provede dle generátoru náhodných čísel (tedy při opakovaném použití dostáváme různé výsledky).

Filtry

Pokud uživatel zjistí, že některé výskyty systematicky obsahují určitou chybu, může být schopen „odfiltrovat“ nežádoucí výskyty popř. „vyfiltrovat“ ty žádoucí pomocí dotazu. Dotaz vytvoří jako běžný dotaz na korpus pro novou konkordanci, navíc ale určí interval vzhledem k dosavadním výskytům, v jakém se má prohledávat. Například můžeme klást podmínku na (ne)existenci nějakého slova ve větě s vyhledaným klíčovým slovem.

Existují dva druhy filtrů: *P-filtr* (pozitivní) – v konkordanci zůstanou pouze řádky, které vyhovují filtru (v zadaném intervalu obsahují nějaké hledané pozice); *N-filtr* (negativní) – v konkordanci zůstanou řádky, které nevyhovují danému filtru.

Další nežádoucí jev, který vzniká při dotazech na posloupnosti pozic, je stav, kdy jeden výskyt je částí jiného výskytu. Například pokud hledáme jmenné skupiny skládající se z posloupnosti přídavných jmen zakončených podstatným jménem, vyhovuje dotazu kromě posloupnosti se dvěma přídavnými jmény i posloupnost o jedno slovo kratší. Uživatel může chtít vybrat z takové skupiny výskytů, kdy jeden je částí druhého, pouze jeden, který je nejdelší, nejkratší, popřípadě zleva nebo zprava.

Podobně lze nakládat s výskyty, které se překrývají: je možné je nechat, jak jsou, nebo spojit do jednoho výskytu většího.

Množinové operace

Na konkordance můžeme nahlížet jako na *množinu* výskytů a můžeme tedy chtít provádět s konkordancemi množinové operace: průnik, rozdíl, sjednocení. Pokud nemáme k dispozici P/N-filtry, lze některé jejich funkce nahradit aplikací průniku, resp. rozdílu konkordancí. Sjednocením lze naopak realizovat spojení několika dotazů do jediné konkordance.

Množinové operace mají celkem zřejmou interpretaci, opět je třeba dát pozor na situace, kdy je jeden výskyt podmnožinou jiného nebo se výskyty překrývají.

Expanze

Jiným druhem úprav konkordance je rozšíření nalezeného jevu o nejbližší pozice z jeho pravého či levého kontextu. Toto rozšíření navíc může být žádoucí pouze u vybraných výskytů. Rozšířit můžeme nejenom o pozice, ale například po nejbližší zvolenou strukturu, například do konce věty.

Ukládání mezivýsledků

Pokud uživatel zkoumá nějaký jazykový jev podrobněji, pracuje se stále stejným konkordančním seznamem. Aplikuje na něj různé filtry, třídění či statistiky. Proto je výhodné si konkordanční seznam, který mohl vzniknout vyhodnocením složitého dotazu, uložit a neopakovat vyhodnocování téhož dotazu. Když pracuje několik uživatelů na stejném tématu, chtějí uložené konkordanční seznamy sdílet i mezi sebou navzájem.

2.3.2 Vizualizace

Základním typem zobrazení je již zmíněný formát KWIC, který ukazuje pouze vlastní text. V korpusu jsou ale většinou uloženy i další informace: další poziční atributy (lemma, značka), struktury (hranice vět), hodnoty struktur. Hodnoty pozičních atributů se většinou vypisují za sebou, navzájem oddělené lomítkem (/). Pro zobrazení struktur i s případnými hodnotami se zachovává formát SGML. Například třetí řádek z obrázku 2.1 je v rozšířeném formátu zobrazen na obrázku 2.2.

Každou z vyjmenovaných informací potřebují uživatelé zobrazovat v jiných případech. Přitom ale kompletní zobrazení všech informací je velice nepřehledné, a to i při použití různých barev či typů písma pro lepší odlišení jednotlivých typů značkování. Uživatel tedy musí mít možnost si jednoduše zapínat zobrazení jednotlivých pozičních atributů a struktur. Ještě výhodnější je *grafické zobrazení* nebo *podmíněné zobrazení*.


```
<doc file="S/NWS/1994/lnd94039" id=081>
   kopcích/kopec/1 okolo/okolo/7 města/město/1 ./././</s><s>První/první/4
            korpus/korpus/1
                       bosenské/bosenský/2 armády/armáda/1 ,/,/ bránící/bráněný/2
```

Obrázek 2.2: Formát KWIC se zobrazenými pozičními atributy a strukturami.

Při grafickém zobrazení není například vybraná gramatická značka zobrazena přímo, ale jednotlivé hodnoty značky jsou ve výpise odlišeny graficky – změnou barvy, typu či velikostí písma daného slova. Vlastní hodnoty nezabírají na monitoru zbytečně místo a přitom jsou na první pohled rozlišitelné. Tuto metodu lze samozřejmě aplikovat pouze v případech, kdy má zvolená charakteristika omezený počet různých hodnot (zhruba do pěti). Větší počet různých grafických zvýraznění je sice technicky možný (současné počítače dokáží běžně zobrazovat tisíce různých barev), ale ubírá na požadované přehlednosti.

Podmíněné zobrazení slouží k zobrazení pouze vybraných hodnot z vybraných atributů. Hodnoty, které uživatele nezajímají, opět na obrazovce nezabírají místo. Obě metody (grafické a podmíněné zobrazení) můžeme kombinovat, tedy podmíněně graficky zvýrazňujeme pouze některé hodnoty. Metody jsou svojí podstatou náročné na zadání, a jsou tak vlastně vyhrazeny pouze pro pokročilé uživatele.

Pro některé konkrétní řádky z konkordančního seznamu nemusí zobrazený kontext stačit k posouzení zkoumaného jevu. Je sice možné zvětšit kontext celého seznamu, ale tím se výpis stává nepřehlednějším, nebo je třeba provádět dodatečné rolování obrazovky. Proto většina korpusových manažerů umožňuje jeden zvolený řádek zobrazit ve specielním zobrazení s mnohem větším kontextem, některé konkordanční programy pracující s omezenou velikostí korpusu zobrazují přímo celý text korpusu a pouze přesunou „kurzor“ na požadovaný řádek.

Třídění

Jakákoliv data se jednodušeji a snadněji kontrolují, jsou-li seřazena podle abecedy (nebo velikosti), nejinak je tomu u konkordance. *Třídění* je tedy další způsob vizualizace konkordance. Jednotlivé hodnoty, ať už to jsou slova, gramatické značky, nebo hodnoty atributů struktur, jsou přehledně uspořádané a lze je snadno porovnávat. Korpusový manažer tedy musí umožňovat třídit nejen podle slov, ale i podle všech pozičních atributů (i dynamických, například třídění podle délky slov) a atributů struktur, na klíčové pozici i v levém či pravém kontextu.

Některé manažery dovolují třídit i vícestupňově. Například nejdříve podle lemmatu, potom podle vlastního slova na klíčové pozici, nebo nejdříve podle značky v bezprostředním levém kontextu, potom podle značky v pravém kontextu. Specielním typem vícestupňového třídění je *cik-cak (zig-zag) třídění* [AEO90], při kterém se porovnávají postupně první slova z levého kontextu, první z pravého, druhá z levého, druhá z pravého, třetí z levého, třetí z pravého atd., dokud není nalezen rozdíl.

Pro získání jistého „náhledu" na konkordanční seznam je vhodné zobrazení vždy pouze jednoho řádku ze všech řádků se stejnou hodnotou třídění, popř. jednoho řádku z každého dokumentu. Tím sice ztrácíme vypovídací schopnost pro pozdější použití statistik, získáme ale přehled o různých typech řádků ve výsledku dotazu.

Samostatnou kapitolou je vlastní způsob uspořádání „hesel" ve výsledku třídění. Korpusový manažer by měl třídit vždy podle použitého jazyka v tříděných hodnotách atributů, tedy například slova českého korpusu podle „českého" třídění, slova anglického korpusu podle anglického třídění. Zde narážíme na zásadní problém, jak takové třídění vůbec nadefinovat. Například o uvedeném českém třídění (podle národní normy) je známo, že není vůbec algoritmizovatelné, závisí totiž v některých případech na interpretaci (významu) tříděných slov. Nezbývá nám než použít nějaké přiblížení, které by alespoň v principu bylo použitelné a dovolovalo definovat různá národní třídění.

Programátoři pro tento účel často používají formu třídicí tabulky, ve které je každému znaku přiřazena číselná hodnota reprezentující prioritu při třídění. Tento přístup byl použit například v manažeru GCQP, ale obecně je nedostatečný, například nedokáže zachytit správné třídění českého *ch*. Na systémech typu UNIX (odpovídající specifikaci POSIX) je možné pro porovnávání znakových řetězců v různých jazycích využít funkce ze subsystému `locale` [loc], ve kterém lze definovat i poměrně složité algoritmy třídění. Bohužel donedávna nebyla implementace `locale` na všech UNIXových systémech běžná (proto také GCQP používá jednodušší, ale přenositelný přístup).

Samozřejmě neslovní atributy (značky) není vhodné třídit podle národního třídění, ale pouze podle ASCII tabulky; Číselné hodnoty, jako délku slov, naopak musíme třídit podle jejich velikosti. Každý poziční atribut korpusu by tedy měl mít ve své konfiguraci (explicitně nebo implicitně) definován způsob třídění. Pokud je používán systém `locale`, lze způsob definovat pouhým kódem zvoleného jazyka (popř. s regionem jako *kanadská francouzština*), který je zpravidla pouze dva znaky dlouhý.

U slovních atributů může uživatel ještě volit speciální volby pro třídění:

- ignorování velikosti písmen,
- retrográdní třídění – slova se třídí „odzadu", nejdříve podle posledního písmene, pak podle předposledního atd.

Při retrográdním třídění s použitím národních pravidel může dojít ke špatnému zatřídění některých slov. Například v češtině, slova obsahující *ch* budou na daném místě zatříděna pod *h*, naopak slova obsahující *hc* (například *vlhce*) budou zatříděna pod *ch*, což je jistě matoucí. Pro korektní české retrográdní třídění bychom museli definovat nové uspořádání zpracovávající *hc* místo *ch*.

Anotace

V souvislosti s ukládáním a sdílením konkordančních seznamů je důležitá možnost tvorby vlastních poznámek k jednotlivým řádkům nebo skupinám řádků. I samotná tvorba skupin řádků je pro dlouhodobější práci s konkordančním seznamem důležitá a v současných korpusových manažerech neobvyklá. Uživateli může zjednodušit práci automatická tvorba skupin řádků, například na základě výsledků třídění.

2.4 Statistiky

Další důležitou vlastností korpusových manažerů je schopnost přesně spočítat výskyty jednotlivých jevů. Počet výskytů je vlastně základní statistika – četnost, kterou musí umět počítat každý korpusový manažer. Pokročilejší by měly zvládat i jiné (složitější) statistiky.

Většinu statistik je možné počítat ve dvou různých úrovních:

globálně (z celého korpusu), kdy se nijak neomezujeme;

lokálně (z konkordance), kdy jsme omezeni pouze na pozice v nalezených jevech, popřípadě na nějakým způsobem zadaný kontext jevů.

Například relativní četnost slov v celém korpusu bude ohraničena velikostí korpusu, kdežto relativní četnost slov v konkordanci bude omezena počtem výskytů v konkordanci.

2.4.1 Četnosti

Jak již bylo uvedeno, základní statistickou charakteristikou je četnost. Pro některé aplikace (například určení, která slova se mají zařadit do slovníku) ale běžná četnost nemusí být nejlepším vodítkem. Problémy tvoří jevy, které se v korpusu vyskytují mnohokrát, ale pouze lokálně, například pouze v jediném dokumentu[1]. Může se zdát, že uvedený problém nastává u malých nebo nevyvážených korpusů, bohužel ale ani velmi rozsáhlé korpusy nejsou pro některé zkoumané jevy dostatečně reprezentativní a pravděpodobně nikdy ani nebudou, protože bychom pro jejich charakteristiku potřebovali korpusy obsahující prakticky veškeré psané texty. Korpusový manažer by tedy měl nabízet i jistou modifikaci uvedené veličiny.

V dalším budeme mluvit o četnosti slov, ale samozřejmě můžeme stejnou definici použít na výskyt jakéhokoliv jiného jevu: četnost lemmatu, značky, slovní skupiny atd.

Logaritmická četnost

Logaritmická četnost byla navržena v [Ryc98] a předpokládá, že korpus je rozdělen do malých částí (dokumentů) a četnost je počítána ve vztahu k těmto dokumentům.

V následující definici označuje $f(x)$ absolutní četnost slova x v celém korpusu, *docs* množinu všech dokumentů (celý korpus) a $f_D(x)$ četnost slova x v dokumentu D. Logaritmická četnost l_a je potom definována následovně:

$$l_a(x) = \sum_{D \in docs} \begin{cases} \log_a(f_D(x)) + 1, & x \in D \\ 0, & x \notin D \end{cases}$$

Parametr a je základ logaritmu, musí tedy být větší než 1.

Označme $d(x)$ počet dokumentů, které obsahují alespoň jeden výskyt slova x a uveďme si nyní některé vlastnosti definované veličiny:

- Pro každé x platí: $d(x) \leq l_a(x) \leq f(x)$, protože $f(x)$ a $f_D(x)$ jsou ve vztahu $f(x) = \sum_{D \in docs} f_D(x)$.
- Případ $d(x) = l_a(x) = f(x)$ nastává právě tehdy, když se slovo x vyskytuje nejvýše jednou v každém dokumentu. To také pokrývá situaci $f(x) = 1$, což implikuje $d(x) = l_a(x) = 1$.
- Při a jdoucím k 1 se $l_a(x)$ blíží k $f(x)$.
- Při a jdoucím k nekonečnu se $l_a(x)$ blíží k $d(x)$.

V článku [Ryc98] je kromě logaritmické četnosti popsána i analogická modifikace MI-score (viz dále).

[1]Stejný problém je řešen v aplikacích Information Retrieval, kde jsou za tímto účelem definovány různé veličiny *relevance* slova ve vztahu k dokumentům.

Redukovaná četnost

Redukovaná četnost [HR99] byla definována jako reakce na zásadní nevýhodu logaritmické četnosti – vazbu na dokumenty. Dokumenty totiž nemusí mít stejné velikosti i u odpovídajících textů (zdrojů). Například v CNK25 jsou novinové texty z jedněch novin rozděleny do dokumentů podle článků (co článek to dokument) a z jiných novin je vytvořen pouze jeden velký dokument se všemi články daného vydání.

Při návrhu redukované četnosti byla snaha splnit následující dvě podmínky:

1. Slova, která se vyskytují v korpusu pouze v omezeném (malém) intervalu pozic, by měla mít nižší redukovanou četnost než slova se stejnou absolutní četností vyskytující se v korpusu rovnoměrně.
2. Pokud se slovo vyskytuje v korpusu zcela rovnoměrně (mezi dvěma po sobě jdoucími výskyty je stejný počet pozic), měla by se redukovaná četnost rovnat absolutní.

Stejně jako v předchozí definici $f(x)$ označuje absolutní četnost slova x v korpusu. Četnost se počítá ve vztahu k pozicím, na kterých se vyskytují slova x, pozice počítáme od 0 do $N-1$, kde N je velikost korpusu. Pro každé slovo x rozdělíme korpus do $f(x)$ intervalů:

$$I_i^x = \left\langle \left\lfloor \frac{(i-1)N}{f(x)} \right\rfloor, \left\lfloor \frac{iN}{f(x)} - 1 \right\rfloor \right\rangle \quad \forall i = 1, \ldots, f(x)$$

Závorky $\lfloor a \rfloor$ znamenají největší menší celou část z a. Redukovanou četnost potom definujeme jako:

$$r(x) = \sum_{i=1}^{f(x)} \begin{cases} 1, & x \in I_i^x \\ 0, & x \notin I_i^x \end{cases}$$

Snadno se dá ukázat, že uvedená definice splňuje výše uvedené požadavky. Příklady slov s vypočtenými redukovanými četnostmi lze nalézt v [HR99].

Porovnání

Obě definované veličiny se dají použít pro eliminaci nerovnoměrného rozdělení slov korpusu. Výpočet obou četností pro všechna slova v korpusu lze provést jedním sekvenčním průchodem přes reverzní index (viz kap .3.2.2) pro slova. Redukovaná četnost má výhodu nezávislosti na rozdělení korpusu do dokumentů, není tedy nijak zkreslena při velkých rozdílech velikostí dokumentů.

Jedinou nevýhodou redukované četnosti je její globalita. K jejímu výpočtu je nutné mít celý korpus a při rozšíření korpusu o nové texty nebo při spojování korpusů do větších (viz kap. 2.6) je potřeba redukovanou četnost pro všechna slova počítat znovu (a to i v případech kdy přidaná část neobsahuje žádný výskyt počítaného slova). Naopak logaritmická četnost při spojování korpusů je rovna součtu logaritmických četností v jednotlivých částech.

2.4.2 Kolokace

Za *kolokaci* většinou považujeme výraz skládající se ze dvou či více slov, který se často používá pro označení něčeho. Kolokace tvoří jmenné skupiny (*osobní automobil*), slovesné skupiny (*promrznout na kost*, v angličtině například frázová slovesa), nebo celé ustálené fráze (*zabít dvě mouchy jednou ranou*). O kolokacích a jejich identifikaci bylo napsáno mnoho publikací a vztah ke korpusovým manažerům není bezprostřední. Některé manažery ale pro identifikaci kolokací v celém korpusu nebo pouze v konkordanci nabízejí některé ze statistických metod.

Nejčastěji se používají *MI-score* a *T-score*, které vycházejí z obecně používaných statistických metod. Veličiny se počítají vždy pro dvojici slov (bigram) a klíčovým parametrem je vždy počet výskytů bigramu v korpusu. Podle potřeby můžeme za výskyt bigramu považovat pouze ty výskyty, kde za prvním slovem bezprostředně následuje druhé slovo nebo se druhé slovo nachází v nějakém *okně* – například pět slov před i za prvním slovem. Různou volbou okna získáme různé výsledky: pro okno velikosti jedna dostáváme pevné kolokace, pro velká okna (nad deset slov) můžeme dostat spíše slova blízká (*nemocnice/doktor*).

Protože ale absolutní číselná hodnota u většiny veličin sama o sobě nic neříká (vždy je závislá na velikosti a druhu korpusu a dalších parametrech veličin), počítají se hodnoty pro všechny možné dvojice slov, nebo se jedno slovo bere pevně a další volitelně. Zajímavé jsou pak ty dvojice slov, které mají největší vypočtené hodnoty.

MI-score

MI-score neboli *vzájemná informace (mutual information)* vychází z teorie informace, kde je definována pro dva jevy x a y jako:

$$I(x,y) = \log_2 \frac{P(xy)}{P(x)P(y)} = \log_2 \frac{P(x|y)}{P(x)},$$

kde $P(x)$ znamená pravděpodobnost jevu x, $P(xy)$ pak pravděpodobnost současně nastalých jevů x a y. $I(x,y)$ vyjadřuje množství informace poskytované výskytem y o výskytu jevu x. V korpusech jednotlivými jevy rozumíme výskyty různých slov (viz [CH90]). Souvýskyt jevů x a y je pro nás výskyt daných slov za sebou, tedy výskyt potenciální kolokace. Pravděpodobnosti odhadujeme z korpusu jako počet výskytů dělený velikostí korpusu. Vzorec MI-score v korpusech pro slova x a y má potom tvar:

$$MI(x,y) = \log_2 \frac{\frac{q}{N}}{\frac{f_x}{N}\frac{f_y}{N}} = \log_2 \frac{qN}{f_x f_y},$$

kde f_x, resp. f_y je četnost slova x, resp. y v korpusu, q je četnost bigramu v korpusu a N je velikost korpusu (jeho celkový počet pozic).

MI-score je bohužel velice citlivé na četnost jednotlivých slov, nejvyšších hodnot dosahují dvojice slov, která jsou v korpusu málo častá. Proto se při výpočtech nastavují spodní hranice četnosti a slova, která jsou pod touto hranicí, se vůbec neuvažují. Ani to ovšem neřeší uspokojivě daný problém, protože většinou nejvyšší hodnotu MI-score vypočítáme pro slova s četnostmi na zvolené hranici.

Vzorec pro výpočet MI byl tedy modifikován tak, aby více zvýhodňoval častější jevy. V [Oak98] je popsána studie, ve které se testovalo MI^a pro a od 2 do 10 a nejlépe vyhovovala veličina MI^3:

$$MI^3(x,y) = \log_2 \frac{q^3 N}{f_x f_y}$$

Tento vzorec již není založen na teoretických základech, ale jde o jistou heuristiku, jak dané veličiny vylepšit na základě empirických pokusů.

Na závěr ještě uveďme variantu pro výpočet MI-score v konkordanci. Jako první slovo chápeme nalezená klíčová slova, druhé slovo je libovolné slovo nacházející se v zadaném kontextu (okně). f_x je potom rovno velikosti konkordance (počet nalezených řádků), q je počet výskytů druhého slova v zadaném kontextu, f_y a N jsou stejná jako v globálním případě.

T-score

T-score vychází ze statistické metody testování hypotéz pomocí tzv. *t testu* (podrobně popsáno snad v libovolné učebnici statistiky, např. [And93]). V případě kolokací testujeme, zhruba řečeno, jestli zjištěné počty výskytů jednotlivých slov a jejich bigramu odpovídají náhodnému rozložení slov v korpusu. Čím vyšší je vypočtená hodnota, tím méně je pravděpodobné náhodné rozložení slov, a slova tedy odpovídají nějakému pevnému vzorci – kolokaci.

Statistika definuje náhodnou veličinu:

$$T = \frac{\bar{x} - \mu}{s}\sqrt{N},$$

kde x je náhodná veličina, $\bar{x}$ průměr měření, μ očekávaná hodnota, s směrodatná odchylka a N počet měření. Směrodatná odchylka je dána vztahem:

$$s = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2 - \bar{x}^2},$$

kde x_i jsou jednotlivá měření. Pokud ale náhodná proměnná x nabývá pouze hodnot $0/1$ (což je náš případ, protože zvolené slovo se na dané pozici buď vyskytuje, nebo nevyskytuje), můžeme vztah zjednodušit:

$$s = \sqrt{\bar{x} - \bar{x}^2} = \sqrt{\bar{x}(1 - \bar{x})}$$

Pokud se navíc $\bar{x}$ blíží nule (velikost korpusu je řádově větší než četnost bigramu), druhý činitel se blíží jedné a můžeme ho zanedbat. Dostáváme tedy pouhé $s = \sqrt{\bar{x}}$.

Pro korpus máme (v dřívějším označení): očekávaná míra výskytu bigramu $\mu = f_1/N \cdot f_2/N$, skutečná míra výskytu bigramu $\bar{x} = q/N$. Po dosazení dostáváme:

$$T = \frac{\frac{q}{N} - \frac{f_1 f_2}{N^2}}{\sqrt{\frac{q}{N}}}\sqrt{N} = (q - \frac{f_1 f_2}{N})/\sqrt{q}$$

Pro výpočet v konkordanci děláme stejné substituce jako v případě MI-score.

Vztah MI-score a T-score

I když jsou obě veličiny navrženy ke stejnému účelu – identifikace kolokací – dávají většinou dost rozdílné výsledky. Jak píše Patrick Hanks:

> T-score nám často říká, co už známe. MI nám s větší pravděpodobností řekne něco neobvyklého a zajímavého (ale ne nutně užitečného).
>
> ...
>
> MI-score nás upozorňuje na neobvyklé, čeho bychom si nemuseli všimnout. T-score má naopak tendenci vyzdvihovat běžné syntaktické vzorce. Editoři slovníků anglických i jiných mají rádi T-score, protože jim pomáhá sestavit popis typické, normální frazeologie a syntaxe.

Pro T-score je nutné používat tzv. *stop list*, tedy seznam nejčastějších pomocných (nevýznamových) slov, která se při výpočtech nemají brát v úvahu. Bez jeho použití by bigramy s nejvyšším T-score obsahovaly právě pomocná slova (předložky, zájmena atd.).

2.5 Způsob použití

Vzhledem k širokému využití korpusů není překvapující, že ve způsobech použití se jednotliví uživatelé značně liší. Můžeme rozlišit dva základní přístupy: interaktivní a dávkový, které se ale mohou navzájem prolínat.

2.5.1 Interaktivní

Většina začínajících uživatelů, kteří se seznamují s korpusovým manažerem nebo nějakým korpusem, využívá nejdříve *interaktivní* způsob práce, kdy na každý krok vidí okamžitou odezvu. Často to též bývá pomocí *grafického uživatelského rozhraní (GUI)*, které vždy uživateli ukazuje, v jakém stavu se nachází a kterým směrem se může dále vydat.

Pokročilí uživatelé využívají interaktivní režim k ladění dotazů, kdy zkouší různé modifikace dotazů a filtrů tak, aby získaná konkordance co nejlépe vystihovala hledaný jev. V tomto režimu práce by tedy korpusový manažer měl podporovat snadné opakování a drobné modifikace všech prováděných operací. Základem je zřejmě historie operací zadaných uživatelem, jejíž prvky lze před opětovným použitím modifikovat (editovat). Také je vhodné, aby měl uživatel po ruce zásobu nejčastěji používaných operací (dotazů, filtrů atd.), které může snadno (nejlépe stiskem zvolené kombinace kláves) vyvolat.

Pro práci v interaktivním režimu je důležité, aby uživatel nemusel dlouho čekat na výsledek některé operace (vyhodnocení dotazu, setřídění konkordance apod.). Pokud to je možné, měl by manažer provádět časově nákladné operace na pozadí, tedy aby uživatel mohl při provádění operace normálně pracovat. Například při vyhodnocování složitého dotazu uživatel nemusí čekat na jeho úplné dokončení (které může obsahovat třeba desítky tisíc řádků), ale již při vyhodnocení několika řádků výsledku si je může prohlížet. Často se stane, že uživatel „na první pohled“ pozná, že něco přehlédl, nebo se v dotazu spletl, a může náročný výpočet zastavit před jeho dokončením.

Zvláště pokročilejší uživatelé nalézají ve své práci posloupnosti operací, které často opakují. Například pro jistý typ dotazu je na výsledek vždy aplikován určitý filtr, výsledná konkordance setříděna a zobrazena se speciálně nastaveným kontextem. Jedná se tedy o jistý typ *dávky*. Korpusový manažer by měl uživateli umožnit maximálně zautomatizovat takovéto opakující se činnosti.

Grafické uživatelské rozhraní

Grafické (nebo alespoň semigrafické) rozhraní (narozdíl od řádkového/ /textového) umožňuje jednak lépe prezentovat výsledky, třeba i použitím grafů, jednak navádí uživatele na jednotlivé operace. Ten si nemusí pamatovat přesné názvy a formáty parametrů jednotlivých příkazů, pouze je vybírá z menu či vyplňuje přehledné formuláře. Pro prezentaci je vhodné, aby šlo snadno měnit různé úhly pohledu na data: měnit kontext, zobrazované atributy a struktury. Zajímavé je také současné zobrazení několika různých pohledů, například v různých oknech.

Protože jsou korpusy velice rozmanité (zvláště v použitých značkách), univerzální manažer nemůže zcela vyhovovat všem. Proto by mělo být GUI manažeru plně konfigurovatelné nebo ještě lépe rozšiřitelné. Aby si uživatel mohl doplňovat sloje vlastní funkce přímo do GUI tak, aby se nijak nelišily od vestavěných funkcí manažeru. Toto je obecný trend, který se v současné době nejvíce rozbíhá v kancelářských aplikacích (textové procesory, tabulkové kalkulátory atd.). Například nové verze balíku Office firmy Microsoft jsou prezentovány ne jako skupina spolupracujících aplikací, ale jako obrovská množina drobných objektů, které je možné snadno kombinovat například do podoby známých aplikací. Uživatel tedy potřebuje na toto propojování nějaký programovací jazyk, kterým lze nejenom provádět všechny operace korpusového manažeru, ale i výsledná data dále prezentovat či zpracovávat.

GUI je typicky doménou osobních počítačů a zde se dostáváme do jistého sporu. I přes značný výkon současných osobních počítačů nelze velké korpusy (v řádech stovek milionů pozic) přímo zpracovávat na každém počítači. Hlavním problémem je sdílení obrovských datových souborů (v řádech GB). Řešením tohoto problému je použití architektury *klient/server*, kdy na jednom výkonném počítači (serveru) jsou uložena data a probíhají tam všechny datově náročné operace a na ostatních osobních počítačích (klientech) probíhá zobrazování a ostatní méně náročné operace.

Rozhraní WWW

Populární WWW (Word Wide Web) je jedním z extrémních přístupů klient/server, označovaný též jako *tenký klient*, kdy server zpracovává *všechny* operace a klient (WWW prohlížeč) pouze zobrazuje připravené (naformátované) výsledky. Jeho velkou výhodou je prakticky absolutní dostupnost klienta. Pomocí WWW lze jen těžko implementovat všechny vymoženosti GUI, ale přinejmenším na ukázku či pro úvodní seznámení to je ideální řešení.

Pomocí WWW lze také realizovat některé speciální aplikace korpusů, které nepotřebují bohatost funkcí korpusového manažeru. Uživatel (či spíše programátor) by měl tedy mít možnost takové aplikace vytvářet – nejlépe ve výše uvedeném programovacím jazyce pro rozšiřování GUI.

2.5.2 Dávkové zpracování

Dávkové zpracování se používá při osvědčených postupech, které je potřeba vyhodnotit pro velké množství různých dat (slov). Například výpočet některé ze statistik (redukovaná četnost, MI-score apod.) pro všechny slova (resp. bigramy) v korpusu je časově náročná operace, kvůli které nemusí mít uživatel v GUI otevřené okno s korpusovým manažerem. Dalšími typicky dávkovými aplikacemi jsou všechny druhy značkování či desambiguace.

Mezi dávkové zpracování můžeme zařadit i aplikace, které používají korpus pouze jako zdroj informací. Například program převádějící český text v ASCII kódování (bez háčků a čárek) na správné tvary s diakritikou vybírá z korpusu četnosti jednotlivých slov a bigramů pro rozhodnutí víceznačností. Takové programy již nepotřebují celý korpusový manažer, ale pouze nějakou knihovnu, která by zpřístupnila klíčová data v korpusu uložená. Pro různé účely jsou používány různé programovací jazyky, knihovna by tedy měla být dostupná (nebo alespoň vytvořitelná) pro současné nejpoužívanější jazyky.

K dispozici by měl také být příkazový jazyk (shell), který by umožňoval jednoduše a rychle aplikovat vyzkoušené postupy a jehož výstup by byl snadno zpracovatelný pomocí standardních nástrojů operačních systémů nebo speciálních aplikací.

2.5.3 Požadavky na rychlost

Z předchozích myšlenek můžeme vybrat následující body týkající se rychlostí jednotlivých operací manažeru:

- Preferujeme předávání alespoň částečných výsledků co nejdříve před celkovou rychlostí. U interaktivního způsobu práce to „opticky“ urychluje operace, u dávkového to umožňuje plynulejší následné zpracování.

- Jakékoliv prezentační operace (vypsání výsledku) musí probíhat okamžitě (do několika sekund).

- Počáteční vytváření korpusu (datových struktur, zejména indexů) může trvat hodiny až desítky hodin. Rozšiřování či upravování korpusu může být realizováno znovuvytvořením celého korpusu.

2.6 Správa korpusů

Textové korpusy mají většinou statický charakter: texty se jednou do korpusu vloží, převedou se do korpusového manažeru a v něm se již data pouze prohledávají. Nikde se neprování žádná změna v textech nebo jejich značkování. Výjimku tvoří korpusy, které se neustále rozrůstají, popř. průběžné korpusy (viz kap. 1.3). Změna ale nikdy nenastává průběžně, ale po skocích, takže vždy můžeme z daných zdrojů „zakódovaný" korpus vytvořit znovu. Druhou výjimkou jsou korpusy, které procházejí stádiem *ručních* úprav, čištění nebo značkování. Ty jsou většinou malého rozsahu (protože jenom přečíst celý BNC by trvalo přes čtyři roky při osmi hodinách denně a 365 dny v roce) a nevyžadují se všechny operace korpusového manažeru popsané v předešlé kapitole. Naopak jsou potřeba jiné funkce, které běžný uživatel korpusového manažeru vůbec nepotřebuje. Je proto lepší k ručním úpravám a opravám korpusů používat specializované programové systémy. Příkladem může být korpusový editor CED [Veb99] vyvinutý na FI MU. Pro uživatele, kteří potřebují i v korpusovém editoru využívat některé pokročilejší prvky korpusového manažeru, bylo realizováno propojení programů CED a GCQP, které zajišťuje předávání v manažeru vyhledaných řádků do korpusového editoru.

Virtuální korpusy

Všeobecné korpusy záměrně obsahují texty z různých zdrojů – mají být reprezentativním výběrem z *celého* jazyka. V mnoha případech se ale uživatelé chtějí omezit jen na určitou oblast. Například jen na novinové texty, texty od jednoho autora apod. Někteří se ovšem na zvolenou oblast specializují, tedy veškeré své práce s korpusem chtějí daným směrem omezit. Může jít dokonce o celou skupinu uživatelů, kteří by chtěli jistý výběr z korpusu sdílet.

Některé korpusové manažery umožňují za tímto účelem vytvářet *virtuální korpusy*, které samy o sobě nejsou nikde fyzicky uloženy. Jedná se pouze o jisté odkazy na existující „fyzické" korpusy. Rozlišujeme následující typy virtuálních korpusů:

- Korpusy, které jsou **výběrem** jistých částí (textů) z fyzického korpusu.
- **Sloučení** několika fyzických korpusů.
- **Kombinace** obou předchozích metod, tedy sloučení výběrů z různých fyzických korpusů.

Pro uživatele může korpusový manažer zcela zakrýt rozdíly mezi fyzickými a virtuálními korpusy a umožnit tak tvořit virtuální korpusy z jiných

virtuální korpusů (ať už výběrem, nebo sloučením). V důsledku lze ale výsledný korpus vždy převést na jednu ze tří vyjmenovaných možností.

Práce s virtuálními korpusy může být o něco pomalejší než s fyzickými, protože některé základní údaje (zejména četnost slova v korpusu) využívané jak při vyhodnocování dotazů, tak při výpočtu statistik jsou u fyzických korpusů přímo uloženy (a dostupné okamžitě, s konstantní složitostí), zatímco u virtuálních je nutné je počítat. Pokud chtějí někteří uživatelé pracovat s určitým virtuálním korpusem intenzivně, má smysl kritické hodnoty předpočítat a uložit na disk. Tato pomocná data budou samozřejmě menší, než kdyby se vytvořil nový fyzický korpus překopírováním zvolených textů.

Vytváření nových virtuálních korpusů by mělo být pro uživatele dostatečně jednoduché, aby byl schopen si „namíchat" texty podle svých aktuálních potřeb. *Sloučení* by mělo být realizováno výběrem ze všech dostupných korpusů. Pokud nechceme do virtuálního korpusu vložit nějaký korpus celý, je možné u něho stanovit procento, podle kterého je automaticky (náhodně) vybrána pouze příslušná část textů.

Výběr textů budou uživatelé nejčastěji určovat podle atributů dokumentů (autor, rok vydání atd.), z technického pohledu tedy podle atributů struktur. Jinou možností je „přetvoření" konkordance na virtuální korpus vložením klíčových slov konkordance spolu s jistým kontextem. Dostáváme tak korpus obsahující například věty, ve kterých se vždy vyskytuje požadovaný jev.

2.7 Dotazovací jazyk

Jak bylo uvedeno dříve, základním způsobem, kterým vnikají nové konkordance, je *dotaz* na korpus. Přestože nejčastějším typem dotazu je vyhledání jednoho konkrétního slova, uživatel by měl mít obecně možnost využít při dotazu všechny informace, které jsou k danému korpusu uloženy (tedy všechny druhy různých značek). Pro korpusové manažery s bohatými možnostmi značkování přirozeně dostáváme silně strukturované a bohaté dotazovací jazyky.

V této kapitole popíšeme různé konstrukce dotazovacích jazyků vzhledem k dříve popsaným možnostem uložených informací. Vycházíme z dotazovacího jazyka CQP, který přehledně a čistě umožňuje zápis rozličných vztahů mezi pozičními atributy v rámci jedné pozice i vztahy mezi atributy na různých pozicích a z dostupných korpusových manažerů jde o jazyk zřejmě nejsilnější (vše, co dokážeme vyjádřit v dotazovacím jazyce jiného manažeru, lze zapsat i v CQP).

Z formálního pohledu ještě musíme dodat, že každý dotaz v jazyce CQP musí být ukončen středníkem (;), který v následujících příkladech bude chybět.

2.7.1 Dotazovací jazyk CQP

Konstrukce pro jednu pozici

Nejjednodušší forma dotazu (slovo na jedné pozici) je i nejjednodušší konstrukcí v dotazovacím jazyce CQP – požadované slovo se pouze uzavře do uvozovek. Pokud tedy chceme získat všechny výskyty slova *jazyk*, zadáme:

```
"jazyk"
```

a dostáváme:

```
 svobodný lid ; b ) tradice , jazyk , duch , historie . . . Já
 doslovně vzal řeč - básnický jazyk jeho mateřštiny , " připomněl
bezprostřední přístup a jejíž jazyk by jako jediná ovládala .
      si mysleli , že jejich jazyk obsahuje jakousi speciální
    " je jazyk . Dovodil , že jazyk , který nejdříve z orgánů
  náhrobním kamenem . Pouze " jazyk " byl chován zvlášť na oltáři
   do manuálu , zjistíme , že jazyk Delphi je Object Pascal .
```

Pokud chceme klást dotaz na jiný poziční atribut, než je vlastní slovo, například základní tvar, použijeme složitější konstrukci. Následující dotaz vybere všechna slova se základním tvarem *jazyk*:

```
[lemma="jazyk"]
```

a dostáváme:

```
dosahují vrcholu své činnosti .  Jazyky jsou plus Naše zkušenosti
      , kterým neznalost cizích  jazyků brání plně využít své
   svobodný lid ; b ) tradice ,  jazyk  , duch , historie . . .
 pověstného vtipu brousí i cizí  jazyky , ve městě se zpívá a
itikové , kteří často mluví jen jazykem svých vlastních zájmů ?
velšské písemnictví v keltských jazycích - tedy literaturu , jež
      - li však volací konvenci  jazyka C , budou se parametry do
```

Poziční atribut pro vlastní slovo se jmenuje `word`. První, výše uvedený dotaz (na slovo *jazyk*) můžeme tedy zapsat:

```
[word="jazyk"]
```

Pro kombinaci hodnot různých pozičních atributů použijeme operátorů logického součtu (*nebo* – „|“), součinu (*a zároveň* – „&“) a negace („!“). Následující dotaz vybere slova, pro která platí, že mají základní tvar *jazyk* **a zároveň** jsou substantiva mužského neživotného rodu v lokativu v singuláru (značka *k1gInSc6*) **nebo** plurálu (značka *k1gInPc6*).


```
[lemma="jazyk" & (tag="k1gInSc6" | tag="k1gInPc6")]
```

```
    písemnictví v keltských jazycích - tedy literaturu , jež na
  možná uvítáte prográmek v jazyce   REXX , který snadno a rychle
 knihovny ( DLL ) napsané v jazyku   C , které obsahují funkce
poslední z velkých novinek v jazyce   Delphi - na výjimky . Než
platformě a na programovacím jazyku   a bude mít otevřenou archi
    , ale především v českém jazyce   . Ovládá půltucet světových
 neumím konverzovat v cizích jazycích , neoslním jeho zahraniční
```

Jednotlivé atributy nemusíme porovnávat s jednoznačnými hodnotami, ale můžeme použít *regulárních výrazů*, ve kterých použití některých znaků (tzv. *metaznaků*) má speciální význam. Nejpoužívanější metaznaky jsou se svým popisem uvedeny v tabulce 2.1. Protože se každý řetězec vyhodno-

Znak	Jeho význam
.	libovolný znak
*	neomezené opakování předchozího znaku
[	začátek množiny znaků
|	alternativní podvýrazy
{	začátek počtu opakování
\	změna významu následujícího znaku

Tabulka 2.1: Nejpoužívanější metaznaky regulárních výrazů

cuje jako regulární výraz, musíme ošetřit vyhledání samotných metaznaků. Například dotaz `"."` nalezne všechny výskyty jednopísmenných slov, samotnou tečku musíme zapsat `"\."`.

Použitý formát regulárních výrazů je totožný UNIXovému příkazu `egrep` [egr]. Jeho implementace též umožňuje zadávat *třídy znaků*, které zjednodušují konstrukci složitých výrazů. Každá třída zastupuje ve výrazu jistou množinu znaků, několik hlavních tříd popisuje tabulka 2.2. Třídy

Třída	Popis
[:alnum:]	písmeno nebo číslice
[:alpha:]	písmeno
[:digit:]	číslice
[:lower:]	malé písmeno
[:upper:]	velké písmeno

Tabulka 2.2: Třídy znaků v regulárních výrazech

znaků jsou vhodné zejména pro národní abecedy, kde by vypisování všech možných písmen bylo velice zdlouhavé a nepřehledné. Bohužel v CQP nelze zvolit jiné národní prostředí (již dříve zmíněný `locale`) než implicitní (angličtina). Pro plné využití této funkce je tedy důležité, aby korpusový manažer pracoval s národním prostředím definovaným pro zvolený atribut.

Konstrukce pro posloupnosti pozic

Hledání posloupnosti pozic se zadá zřetězením dotazů pro jednotlivé pozice. Například dotaz na předložku (značka začínající *k7*) následovanou slovem s lemmatem *jazyk* vypadá takto:

```
[tag="k7.*"] [lemma="jazyk"]
```

```
Navíc ve všech programech v jazyce    REXX musí být na prvním řádku
 knihovny ( DLL ) napsané v jazyku    C , které obsahují funkce
    státu Alabama znamená v jazyce    Čerokézů " zde odpočíváme
esionalita byla patrna už z jazyku    jejich projevů . Vyvrcholení
    říci to , co mu slina na jazyk    přinese - chrání jej poslanecká
 poněkud jinak . V Ústavu pro jazyk   český ČSAV byly dokonce zpracovány
  připomíná při zamyšlení nad jazykem vůní a pachů . Třeba že japonské
```

Silnou variantou dotazů CQP jsou regulární výrazy nad pozicemi. Stejně jako v případě jednotlivých znaků v hodnotě pozičního atributu můžeme specifikovat alternativy (i různě dlouhých) posloupností pozic, nebo jestli (popř. kolikrát) se má některá pozice v dotazu opakovat. Následující dotaz vybere posloupnosti pozic, které začínají předložkou, případně následovanou zájmenem (značka začínající *k3*), pokračující libovolným (i žádným) opakováním adjektiva a končící slovem se základním tvarem *jazyk*.

```
[tag="k7.*"] [tag="k3.*"]? [tag="k2.*"]* [lemma="jazyk"]
```

```
  a velšské písemnictví v keltských jazycích - tedy literaturu , jež na
 struktury známé z ostatních procedurálních jazyků ( cykly , podmínky ,
 programy píší i v jiných programovacích jazycích , a vyšli ostatnímu
    . Viry programované ve vyšších jazycích obsahují množství nadbytečných
kolik ? Kdy se začíná   s dalším jazykem ? Kolik hodin matematiky děti mají
        stylu . Ostatně v našem jazyce ( stejně jako v jiných - což ovšem
připomíná při zamyšlení nad jazykem vůní a pachů . Třeba že japonské
```

Pokud na některou pozici neklademe žádné požadavky, zapíšeme ji jako „prázdné" hranaté závorky: `[]`.

Struktury v dotazech

I když v korpusovém manažeru CQP lze uchovávat hodnotu pro struktury, nelze je používat v dotazech. Systém se omezuje pouze na existenci začátku nebo konce určité struktury. Používá se zápisu ve tvaru SGML, tedy jméno

struktury uzavřené v úhlových závorkách, například `<s>` pro začátek věty a `</s>` pro konec.

Druhou možností začlenění struktury do dotazu je použití konstrukce `within`, která zajistí, aby celá posloupnost hledaných pozic byla v jedné struktuře, tedy aby nenastala situace s hledanou posloupností pozic začínající v jedné struktuře a nekončící v následující struktuře.

Omezení na atributy struktur bychom mohli psát analogicky booleovským výrazům na jedné pozici pouze s tím rozdílem, že výraz bude uveden v úhlových závorkách a bude začínat jménem struktury.

Globální omezení

Dotaz může obsahovat několik typů *globálních omezení*, která se nevztahují přímo k některé pozici, ale dávají nějakým způsobem do vztahu několik pozic. V prvé řadě to jsou *reference*, pomocí nichž se můžeme z jedné pozice odkazovat na hodnoty atributů na jiné pozici. Například zajištění shody čísla a pádu u přídavného a podstatného jména může vypadat takto (předpokládáme existenci atributů *number* a *case* v korpusu):

```
a:[tag="k2.*"] [tag="k1.*" & number=a.number & case=a.case]
```

Další formou globálních omezení jsou testy na hodnoty *funkcí*, které jsou podobné dynamickým atributům. Podrobnější výklad k referencím a funkcím lze nalézt v [SC96].

Operátory `meet` a `union`

Dotazovací jazyk CQP vlastně tvoří dva systémy. Jeden založený na regulárních výrazech nad pozicemi (popsaný výše), druhý založený na operátorech `meet` a `union`. Oba jazyky nelze navzájem míchat, jejich rozlišení je dáno prefixem `MU` pro druhý jazyk.

Ve výsledku libovolného dotazu typu MU jsou vždy řádky s jediným klíčovým slovem. I když můžeme v dotazu dávat do souvislosti několik různých pozic (posloupnost), výsledkem je vždy pouze jedna.

Operátor `meet` má dva povinné a dva nepovinné parametry. Každý z prvních dvou parametrů obsahuje dotaz na jednu pozici, výsledkem je společný výskyt dvou zadaných slov. Druhé dva parametry obsahují rozsah pozic (okno), ve kterých se musí druhé slovo objevit vzhledem k prvnímu. Implicitní hodnoty jsou 1, tedy slova musí být bezprostředně za sebou v daném pořadí. Například výše uvedený příklad na předložku následovanou slovem se základním tvarem *jazyk* můžeme pomocí operátoru `meet` zapsat takto:

```
MU (meet [tag="k7.*"] [lemma="jazyk"])
```

Klíčová slova tvoří pouze předložky:

```
    Navíc ve všech programech v jazyce REXX musí být na prvním
     knihovny ( DLL ) napsané v jazyku C , které obsahují
        státu Alabama znamená v jazyce Čerokézů ” zde odpočívá
 profesionalita byla patrna už z jazyku jejich projevů .
poslanec říci to , co mu slina na jazyk přinese – chrání jej
     poněkud jinak . V Ústavu pro jazyk český ČSAV byly dokonce
   se připomíná při zamyšlení nad jazykem vůní a pachů . Třeba
```

Zajímavější je použití okna. V následujícím příkladě musí být lemma *jazyk* v rozmezí od jednoho slova před a do tří slov za předložkou.

```
MU (meet [tag=”k7.*”] [lemma=”jazyk”] -1 3)
```

Ve výsledku jsou mimo jiné řádky:

```
   stojí za zmínku programy pro výuku jazyků , pro tvorbu
   nezávislá na platformě a na programovacím jazyku a bude
  je věhlasný gurmán takřka s chlupatým jazykem , jen s
 vaše matka vyučovala jazyky v závodním klubu Spojených
       . ” Jednacími jazyky na kongresu MOV jsou angličtina
stejným významem zachytíme i v jiných slovanských jazycích
  tento jev poněkud jinak . V Ústavu pro jazyk český ČSAV
```

Operátor `union` provádí sjednocení dvou dotazů, které jsou jeho parametry. Pokud tedy jsou jako parametry jednoduché dotazy, výsledek je ekvivalentní konjunkci.

Největší výhodou operátorů `meet` a `union` je ovšem jejich kombinovatelnost: parametrem obou operátorů může být opět jeden z nich. Následující dotaz tedy vybere slova *jazyku*, *jazyce*, nebo *jazycích*, před kterými je ve vzdálenosti maximálně čtyř pozic slovo *v* nebo *ve*.

```
MU (meet (union (union ”jazyku” ”jazyce”) ”jazycích”)
         (union ”v” ”ve”) -4 -1)
```

```
  písemnictví v keltských jazycích – tedy literaturu , jež na
    ve všech programech v  jazyce  REXX musí být na prvním řádku
 části lze psát i v jiném  jazyce  pomocí protokolu DDE nebo
i v jiných programovacích jazycích , a vyšli ostatnímu světu
  programované ve vyšších jazycích obsahují množství nadbytečných
   i v jiných slovanských jazycích . Také pomístní jména
  stylu . Ostatně v našem  jazyce  ( stejně jako v jiných –
```

Existují pravděpodobně dva hlavní důvody, proč byl do CQP tento druhý dotazovací jazyk zaveden. 1) Pomocí operátorů `meet` lze vyhledávat nespojité skupiny slov, u kterých dokonce nemusí záležet na pořadí jednotlivých složek. Velkou nevýhodou ale zůstává, že ve výsledku je vždy

vyznačena (klíčové slovo) pouze jedna z komponent, další nejsou na první pohled patrné. Řešením by mohlo být například automatické vytvoření kolokace, které by vše hledané ve výsledku zvýraznilo, jak bylo ukázáno v posledním příkladě. 2) Manažer vyhodnocuje dotazy typu MU odlišným algoritmem, který je většinou mnohem rychlejší, než vyhodnocení ekvivalentního dotazu ve tvaru regulárního výrazu.

2.7.2 Rozšíření jazyka o ukazatele

Korpusový manažer CQP nedokáže v žádné podobě zpracovávat ukazatele, jeho dotazovací jazyk tedy pro ukazatele nenabízí žádné konstrukce. Tato podkapitola popisuje nová rozšíření dotazovacího jazyka, která umožňují jednoduchým způsobem zadávat v dotazech ukazatele.

Existují dva rozdílné způsoby použití ukazatelů:

- omezení hodnoty atributu na odkazované pozici nebo struktuře,
- porovnání ukazatele s některou pozicí v dotazu.

První variantu realizujeme analogicky referencím:

```
[tag=”k7.*” & depend.tag=”k1.*”] within
                    <doc source=”LN” & date=”1991”>
```

V druhé variantě porovnáváme hodnoty ukazatelů přímo s referencemi v dotazu:

```
a:[...][]{0,5} [... & uk=a]
```

2.7.3 COFE

Popsaný dotazovací jazyk se může zejména pro začátečníky zdát příliš složitý. Již dotaz na jedno slovo musí obsahovat uvozovky, jen trošičku složitější dotazy se neobejdou bez různých závorek (například hranaté závorky se ani nevyskytují na standardní české klávesnici), rovnítek a identifikátorů. Pro tyto uživatele jsme navrhli další dotazovací jazyk, který byl inspirován přístupem „query by example“, tedy dotaz pomocí příkladu. Příkladem v našem případě bude napodobení výstupu dotazu ve formátu KWIC. Jazyk byl pojmenován *COFE podle COncordance From Example*.

Pokud se dotazujeme pouze na jedno slovo, je celým dotazem přímo toto slovo. Pro posloupnost za sebou jdoucích slov je dotazem opět daná posloupnost. Pokud ovšem je na jedné pozici několik slov (oddělených mezerou), jak bylo např. popsáno v kap. 1.4 musíme daný výraz zapsat do uvozovek. Stejně tak do uvozovek píšeme různé oddělovače, které mají jinak speciální význam.

Další atributy (např. základní tvar a značka) se většinou zobrazují u každého slova za lomítkem (/), stejně je tomu i při dotazu. Následující příklady postupně znamenají: slovní tvar *žena* s lemmatem *hnát*; lemma *hrad* se značkou *k1gInSc6*; libovolné substantivum (značka začínající *k1*).

```
žena/hnát
/hrad/k1gInSc6
//k1.*
```

Pokud neznáme závazné pořadí atributů (v uvedených příkladech bylo *word/lemma/tag*), můžeme za lomítko zapsat i jméno atributu s dvojtečkou. Stejné příklady by potom vypadaly následovně:

```
žena/lemma:hnát
lemma:hrad/tag:k1gInSc6
tag:k1.*
```

Zápis s lomítky samozřejmě odpovídá konjunkci – logické spojce AND (& v CQP).

Každou z hodnot atributů můžeme negovat pomocí vykřičníku (!) na začátku hodnoty. Následující dva dotazy tedy shodně vybírají slova s lemmatem *jazyk* v jiném než sedmém pádě jednotného čísla (v prvním případě značka je různá od *k1gInSc7*, v druhém slovní tvar různý od *jazykem*)

```
lemma:jazyk/tag:!k1gInSc7
lemma:jazyk/word:!jazykem
```

Při uvedeném zápise netvoří zadávané hodnoty regulární výrazy, znaky `.[]*+{}` tedy nemají žádný speciální význam. Pokud chceme použít regulárních výrazů, jako první znak hodnoty zapíšeme vlnku (˜):

```
lemma:˜On[ao]?
```

Pro negaci regulárního výrazu používáme `!˜`. Následující dotaz tedy hledá slovní tvary s lemmatem *jazyk* mimo 2. a 7. pádu libovolného čísla:

```
lemma:jazyk/tag:!˜k1gIn.c[27]
```

Pokud v některém atributu potřebujeme zadat vykřičník nebo vlnku, musíme umístit hodnotu uvozovek:

```
"!"
nic/tag:!"!"
```

Disjunkci (OR, | v CQP) zapisujeme na úrovni pozic stejně jako v CQP (jednotlivé pozice ovšem musí být odděleny mezerou), opakování a strukturní značky také:


```
( ... | ....).
(....){1,5}+*?.
<s>, <s ....>, </s ....>
```

Pro strukturní značky je dostupná ještě speciální konstrukce na začátek a konec značky. Následující dotazy jsou tedy ekvivalentní:

```
<s .../>
<s ...> ()* </s>
```

Prázdné závorky znamenají libovolnou pozici.

Globální omezení se zapisují na konec dotazu a oddělují se znakem and (&):

```
lemma:%1/tag:k7 tag:k1 & freq(%1) > 10
```

Ukazatele a zarovnání jsou realizovány analogicky navrženému rozšíření jazyka CQP:

```
/číslo/ang:number
```

Tento dotaz tedy vyhledá slova s lemmatem *číslo* s anglickým ekvivalentem *number* v paralelním korpusu (atribut *ang*).

2.7.4 Grafická rozhraní pro tvorbu dotazu

Zejména začínající uživatelé často naráží na přílišnou složitost některých konstrukcí dotazovacího jazyka. Ta se projevuje na dvou úrovních:

- složitost různých typů dotazů (regulární/booleovské výrazy),
- složitost použitého systému značek.

Korpusové manažery někdy nabízejí možnost asistence při tvorbě dotazu formou grafického uživatelského rozhraní (GUI). Dotaz je uživateli prezentován ve tvaru nějakého grafu či tabulky, ze které potom manažer automaticky vytvoří textový příkaz, u kterého je zajištěna syntaktická (formální) správnost (nikde nechybí uzavřené závorky, ve jménech atributů nejsou překlepy atd.). U gramatických značek manažer například zajišťuje, aby uživatel pro substantiva netestoval hodnotu času apod.

Pravděpodobně žádné GUI nemá tak jednoduché a přímočaré ovládání, aby v něm dokázal zkušený uživatel tvořit složité dotazy rychleji, než přímým zápisem textu dotazu, dobře navržené GUI však může být mnohem pohodlnější. Korpusové manažery obsahují následující typy GUI:

Menu
: Výběr z menu je asi nejjednodušší možností. Uživateli jsou předkládány jednotlivé typy dotazů, ve kterých pouze doplňuje žádané hodnoty. Většinou nelze tímto způsobem vyjádřit všechny možnosti či kombinace dotazovacího jazyka.

Grafy
: Dotaz je reprezentován formou stromu, v jehož vnitřních uzlech jsou operace spojující jednoduché/základní testy atributů. Uživatel tak má i u složitých dotazů celkový přehled o jejich struktuře.

Diagramy
: V diagramech zastupují základní dotazy okna či tabulky, které se dále do sebe vnořují. Graficky je též možné vyjádřit různé vztahy mezi nesouvislými částmi dotazu.

Dobrý přehled všech možných uživatelských rozhraní lze nalézt v [Hea99]. Popsány jsou GUI k aplikacím *Information Retrieval*, ve kterých jde o vyhledávání informací (významů), což je problematika blízká korpusům.

2.8 Dosavadní korpusové manažery

2.8.1 CQP (IMS Corpus Workbench)

CQP znamená *Corpus Query Processor* a je to jeden ze skupiny programů *IMS Corpus Workbench*. Slouží k vyhodnocování dotazů, práci s konkordancemi, subkorpusy (typ virtuálních korpusů vzniklých výběrem) a jednoduchému vypisování konkordancí ve formě textových souborů. Celý systém byl vyvinut na Solarisu (UNIX od firmy Sun) a nyní je dostupný ještě i na Linuxu. Kromě CQP obsahuje IMS Corpus Workbench samostatné programy pro zakódování vstupních (vertikálních) textů do binární podoby, se kterou potom všechny další programy pracují, programy pro kompresi některých částí binárních dat, pomocné programy pro vypisování slovníku všech slov z korpusu a další. Systém též obsahuje GUI XKWIC [Chr95] v X Window System a knihovnu v jazyce C pro přístup k binárním datům (i komprimovaným) z externích programů.

CQP i celý IMS Corpus Workbench má dvě zásadní výhody, které systém výrazně odlišují od ostatních:

- silný dotazovací jazyk,
- modulární návrh ve stylu UNIXu.

Dotazovací jazyk byl popsán v předchozí kapitole, zmiňme se tedy o *modulárním návrhu*. Všechny programy systému (mimo XKWIC) lze použít v dávkovém zpracování, akceptují standardní vstup a zapisují na standardní výstup, snadno se tak propojují do větších celků pomocí *roury (pipe)*. Většinu jednoduchých statistik či výpočtů nad korpusy je možné provést pouhým zadáním několika příkazů, bez nutnosti jakéhokoliv programování.

Hlavní nevýhoda je dána jednou z výhod – mocným dotazovacím jazykem. V CQP jsou kvůli potenciálním složitým konstrukcím dotazovacího jazyka implementovány komplexní metody vyhodnocování dotazů, při nichž bohužel nejsou aplikovány prakticky žádné optimalizace, a tak nastávají případy, kdy i relativně jednoduchý dotaz (například posloupnost dvou konkrétních slov) je vyhodnocován velice dlouho. Přitom většinou existuje způsob, jak dotaz přeformulovat nebo převést na sérii několika dotazů, jejichž vyhodnocení může být prakticky okamžité. Pro optimální výběr vhodného tvaru dotazu musí uživatel znát nejen způsob výpočtu různých typů dotazů, ale i odhady počtů výskytů jednotlivých atomických částí (hodnot pozičních atributů).

Pro některé uživatele jsou podstatným omezením podporované platformy (Solaris, Linux) zvláště pro GUI. Toto omezení se snaží na Stuttgartské Universitě, kdy celý produkt vznikal, řešit přepsáním (reimplementací) systému v jazyce Java. Tím by bylo dosaženo téměř absolutní přenositelnosti na všechny běžné operační systémy. Zatím ale není žádný výsledek této snahy k dispozici.

2.8.2 CQM

CQM je nadstavbou nad korpusový manažer CQP. Vyhodnocování dotazů se provádí přímým voláním CQP, prezentace výsledků je zajišťována pomocí knihovny v jazyce C. Primárním důvodem vzniku CQM bylo vytvoření GUI v semigrafickém (textovém) prostředí. Uživatelé tak mohou používat GUI i ze vzdálených počítačů pomocí služby *telnet*.

Program též obsahuje zdařilou podporu tvorby dotazu. Uživatelé z menu vybírají různé typy dotazů, složitější dotazy jsou prezentovány ve formě stromu. Podporována je též tvorba gramatických značek na základě rozsáhlého konfiguračního souboru.

2.8.3 GCQP

GCQP je také nadstavbou manažeru CQP. Systém staví na architektuře klient/server a ve skutečnosti se jedná o dva programy: `cqsd` jako server pracuje na počítači s CQP, které využívá pro vyhodnocování dotazů, a GCQP jako klient běžící v grafickém prostření na různých platformách (Windows

95/98/NT/2000, Apple Macintosh, Unix & X Window System). Komunikace mezi klientem a serverem je založena na TCP/IP a je optimalizována na množství dat, takže práce s korpusem může probíhat i po pomalých (modemových) linkách.

Systém byl oproti CQP rozšířen o výpočet víceúrovňových statistik, kolokací na základě MI-score a T-score, vyhledávání kolokací a použití filtrů při tvorbě konkordance. GCQP obsahuje celou řadu možností podporujících tvorbu dotazu:

- seznamy nejčastějších dotazů,
- vytváření a používání šablon dotazů, které umožňují často se opakující podobné dotazy nahradit identifikátorem s parametry,
- grafická tvorba dotazu ve formě stromu.

Server umožňuje řízení přístupu jednotlivých uživatelů (omezení adres počítačů, maximální počet zobrazených řádků, seznam dostupných korpusů pro každého uživatele). GCQP je široce konfigurovatelné: existuje ve dvou jazykových mutacích (a další je snadné přidat), vzhled i obsah všech grafických prvků lze upravovat, v omezené míře uživatel může doplňovat další ovládací prvky.

2.8.4 CUE, XCUE, QWICK

CUE je knihovna funkcí pro manipulaci s binárními daty korpusu, XCUE je potom korpusový manažer vybudovaný nad touto knihovnou. Původně byly oba produkty naprogramovány v jazyce C++ na UNIXu, novější verze knihovny je již implementována v jazyce Java a nad ní je též v jazyce Java vytvořen nový manažer QWICK. Knihovna klade velký důraz na optimální uložení binárních dat, proto používá moderní techniky komprese.

Dotazovací jazyk je v porovnáni s CQP mnohem jednodušší. Umožňuje klást dotazy na jednotlivá slova (místo plných regulárních výrazů je možné používat pouze jeden metaznak `"*"` zastupující libovolnou posloupnost znaků), posloupnosti slov, alternace nebo slovo v okně jiného slova (analogicky operátoru `meet`). Na druhé straně poskytuje QWICK více možností pro práci se strukturami, zejména možnost uložení a dotazování atributů struktur. QWICK dokáže spočítat velké množství statistik v konkordanci: kromě MI-score a T-score zde nalezneme i MI^2 a MI^3, z-score a další.

Implementace v jazyce Java přináší sice platformovou nezávislost, má ale za následek velkou náročnost aplikace na výkon počítače, takže na jen několik let starých počítačích prakticky nelze QWICK provozovat.

2.8.5 SARA

Program SARA byl vyvinut jako primární korpusový manažer BNC[2]. Jde o klienta běžícího na systémech Windows, který se musí připojit pomocí TCP/IP na (UNIXový) server s vlastními daty BNC. Celý systém (klient SARA, server `sarad` a několik pomocných programů) je dostupný na internetu (`http://info.ox.ac.uk/bnc/sara/`) včetně rozsáhlé dokumentace.

SARA umožňuje vyhledávat jednotlivá slova či sekvence slov, konkrétní slovo s určitou gramatickou značkou, omezovat slova na jisté SGML struktury. Naopak nedokáže najít posloupnosti značek, používat informace o četnostech v dotazech, vyhledávat kolokace. Program je těsně vázán na BNC (systémy SGML značek a gramatických značek), i když teoreticky je možné jeho použití i pro jiné korpusy.

2.8.6 WordSmith

Systém WordSmith obsahuje šest navzájem spolupracujících nástrojů pro Windows. Program *Wordlist* generuje seznamy slov s jejich četnostmi (s několika možnostmi uspořádání) a několik globálních statistik (počet všech slov, počet různých slov, průměrná délka slova a věty atd.). Program *Concord* vytváří konkordance a vyhledává a zobrazuje kolokace. Nástroj *Keywords* slouží k identifikaci klíčových slov v daném textu pomocí jednoduchých statistik. Další nástroje se používají k přípravě textů pro korpus.

Celý systém pracuje s běžnými textovými soubory (popř. s SGML značkami) a nevytváří si žádné pomocné soubory (indexy) pro vyhledávání, které je tedy vždy úměrné velikosti zkoumaného textu. Proto se příliš nehodí pro zpracování rozsáhlých korpusů.

2.8.7 TACT

TACT (Text Analysis Computing Tools) je soubor 16 programů pro MS-DOS. Všechny programy mají konfigurovatelný výstup, který lze snadno zpracovávat jinými nástroji ze systému nebo uživatelem vytvořenými speciálními programy. Tři z programů slouží ke značkování textů na úrovni slov – pro každé slovo je uveden základní tvar, slovní druh (part-of-speach, POS) případně další značka. Uživatel si musí před značkováním nejdříve vytvořit slovník se značkami, není tak omezen na vestavěné značky pro an-

[2]Část BNC (obsahující po jednom milionu pozic mluveného a psaného textu) se prodává také na jednom CD, na kterém jsou přiloženy různé korpusové manažery. V prvé řadě to je samozřejmě klient SARA, dále CQP, QWICK a WordSmith (popsaný dále).

gličtinu. Malý slovník obsahující nejčastější (více než 300) slova angličtiny je přímo v distribuci.

Žádný program systému není omezen na jeden jazyk, vždy je možné pomocí konfiguračních souborů upravit chování nástrojů pro speciality (kódování, značky atd.) zvoleného jazyka. I když si systém vytváří pomocné indexy pro vyhledávání, autoři ho doporučují pouze pro malé a středně velké texty typu Shakespearovy hry apod.

2.8.8 Konkordanční programy

V této podkapitole zmíníme několik konkordančních programů, které nemůžeme považovat za korpusové manažery, protože nedokáží zpracovávat opravdu rozsáhlé texty. Neumožňují ukládat značkovaný text (ať už na úrovni pozičních atributů nebo struktur) a jejich vyhledávací možnosti se také většinou omezují na nalezení všech výskytů jednoho slova v textu. S programy je často dodáván editor, ve kterém si uživatel může připravit text pro korpus.

Většinou jsou dostupné pouze pro platformy DOS, resp. Windows nebo Macintosh. Nové konkordanční programy stále vznikají a jiné přestávají být podporovány. V následujícím seznamu jsou vyjmenovány programy existující koncem roku 1999.

OCP (Oxford Concordance Program) je jedním z nejstarších konkordančních programů. Starší (původní) verze v jazyce FORTRAN byla určena pro systémy VMS, UNIX a VM/CMS, nyní je dostupná verze pro MS-DOS nazvaná *Micro-OCP*.

Program se ovládá z příkazové řádky a lze jím vytvářet konkordance a seznamy slov v jakémkoliv jazyce.

Concordance vytváří konkordance a jednoduché statistiky z textů. Specialitou je *Web Concordance*, což je soubor vygenerovaných HMTL stránek obsahující všechny výskyty všech slov v textu navzájem propojených odkazy. Web Concordance lze přímo umístit na WWW, ale svojí podstatou se hodí pouze pro nevelké texty.

MonoConc Pro je nová verze programu *MonoConc* určená pro systém Windows 95. Umožňuje vyhledávat slova i pomocí regulárních výrazů a kontextu. Jednoduchým způsobem (pomocí četností) nalézá kolokace.

2.8.9 `sgrep`

Program `sgrep` není korpusovým manažerem. Slouží k vyhledávání řetězců v textových souborech (stejně jako příkaz UNIXu `grep`). Jeho dotazovacím jazykem nejsou regulární výrazy, ale poměrně mocný systém operátorů nad částmi textu (posloupnostmi znaků). Obsahuje dvacet různých operátorů, které lze navzájem libovolně kombinovat, nástroje pro tvorbu maker pro zjednodušení často zadávaných částí dotazů. Protože zpracovává přímo soubory (nebo i standardní vstup), nevytváří si žádné pomocné vyhledávací struktury a jeho použití je omezeno na nevelké soubory. Může však sloužit jako inspirace pro komplexní dotazovací jazyky.

2.9 Shrnutí požadavků na korpusový manažer

V následujícím seznamu jsou uvedeny všechny požadavky na dobrý korpusový manažer tak, jak byly v této kapitole podrobně popsány. Mnohé z nich zatím nebyly v žádném existujícím manažeru implementovány, nebo pouze v omezené míře. Naše původní vlastnosti jsou v seznamu vyznačeny tučným typem písma klíčových slov. Některé z vyjmenovaných vlastností není jednoduché implementovat a zejména na ně je zaměřena následující kapitola.

Vlastnosti korpusového manažeru:

1. Uložené informace
   (a) více pozičních atributů (ne pouhé slovo) na jedné pozici
   (b) **multihodnoty** pozičních atributů
   (c) struktury (strukturní značky)
      i. stromová struktura – vnořené intervaly
      ii. prázdné struktury (pouze „oddělovače“)
      iii. atributy struktur stejně jako poziční atributy
   (d) dynamické atributy
      i. uživatelsky definovatelné
      ii. použitelné **v dotazech**
   (e) základní statistiky
      i. četnost slov (a hodnot jiných pozičních atributů)
      ii. **logaritmická/redukovaná** četnost
      iii. bigramy bez omezení kombinace atributů a velikosti okna
      iv. „**vícegramy**“

   (f) **ukazatele na pozice**
   (g) ukazatele na struktury

2. Dotazovací jazyk
   (a) atribut/hodnota
      i. regulární výrazy i s **třídami** ([:upper:])
      ii. **volba jazyka** (kódování) pro každý atribut
      iii. podpora interaktivní tvorby dotazu ve formě popisu struktury hodnot u každého atributu
      iv. booleovské operátory (and, or, not) mezi atributy na jedné pozici
   (b) regulární výrazy nad pozicemi
   (c) struktury
      i. dotazy i na atributy
      ii. specifikace začátku, konce i **celého obsahu** struktury
   (d) všechny prvky uložené v korpusu: dynamické atributy, ukazatele, četnosti, bigramy, ...
   (e) uživatelsky definovatelná makra pro transformaci dotazu
   (f) volba vztahu výsledných řádků (podřetězec zprava/zleva)

3. Práce s konkordancemi
   (a) třídění **podle jazyka** (kódování)
   (b) strukturovaná (víceúrovňová) frekvenční distribuce a v libovolných intervalech
   (c) výpočet kolokací na základě statistik (T-score, MI-score, ...)
   (d) třídění, frekvenční distribuce i kolokace na základě struktur, atributů struktur i dynamických atributů
   (e) ukládání mezivýsledků (konkordancí) a jejich sdílení mezi různými uživateli
   (f) ruční/automatické **sdružování řádků do skupin** a jejich **anotace**

4. Uživatelské rozhraní
   (a) knihovna funkcí a její **začlenění do skriptovacích jazyků**
   (b) příkazový řádek pro dávkové zpracování
   (c) grafické uživatelské rozhraní (GUI)
   (d) interaktivní podpora tvorby dotazu v GUI

   (e) klient/server architektura
   (f) správa přístupových práv
      i. seznam dostupných korpusů pro každého uživatele
      ii. možnost omezení počtu řádek ve výsledku dotazu
      iii. možnost omezení zobrazeného kontextu

Kapitola 3

Návrh efektivní implementace

Tato kapitola popisuje struktury a algoritmy, na kterých korpusový manažer stojí. Pro jednotlivé algoritmy jsou vypočteny složitosti operací nebo alespoň jejich reálné odhady a limity.

Po úvodní poznámce o efektivitě v první podkapitole následuje povídání o různých způsobech vyhodnocování dotazů. V dalších podkapitolách jsou pak rozebrány jednotlivé části (moduly) korpusového manažeru a jeho celková struktura. Závěrečné kapitoly jsou věnovány výpočetně nejnáročnějším částem manažeru: vyhodnocování dotazů a prvotní vytváření datový struktur (indexace).

3.1 Úvodní poznámky o efektivitě

Efektivní implementací budeme rozumět takovou implementaci, která dokáže zpracovávat *velká* data dostatečně *rychle* (viz kap. 2.5.3). Rychle zpracovávat znamená rychle procházet zvolené datové struktury, zejména při vyhodnocování dotazů. Tyto snahy nám přinášejí dva konfliktní body:

1. Více dat většinou samo o sobě znamená menší rychlost zpracování.

2. Při použití kompresních metod pro zmenšení objemu dat se obvykle používají složitější datové struktury, které mají opět za následek snížení rychlosti.

Pokud existuje více algoritmů nebo datových struktur, některé přinášejí vyšší rychlost na úkor požadované velikosti paměti, jiné zase menší rozsah paměti na úkor rychlosti. Pokud se nám ale podaří pochopit podstatu daného problému, rozpoznat jeho strukturu, můžeme navrhnout algoritmy a datové struktury, které jsou rychlejší, resp. menší.

Jak ukážeme v této kapitole, v některých případech se nám podařilo navrhnout datové struktury, které jsou menší než obecné přímočaré řešení a přitom umožňují podstatně rychlejší zpracovávání.

V kap. 2.6 jsme vysvětlili statickou povahu korpusů, v principu tedy můžeme počítat se statickými strukturami korpusového manažeru a můžeme využít datových struktur, které jsou přizpůsobeny spíše rychlému vyhledávání, popř. malé paměťové náročnosti než změnám či přidávání nových dat. V dalším textu tedy nebudeme popisovat modifikační operace jednotlivých datových struktur a v samostatné kapitole bude popsána jejich konstrukce.

3.2 Vyhledávání v korpusu

Přes různorodost dotazů a velice rozsáhlé možnosti dotazovacího jazyka, musí být vyhodnocení dotazů dostatečně rychlé i pro rozsáhlé korpusy (v řádech stovek milionů pozic). Pro vyhodnocení dotazu existují dva základní přístupy: sekvenční prohledání celého textu a přímý přístup pomocí některé z forem indexů.

3.2.1 Sekvenční prohledání

Při tomto přístupu se postupně prohlíží každá pozice v korpusu a testuje se na splnění všech podmínek dotazu.

Vzhledem k požadavku na možnost práce s korpusy o velikostech až stovek milionů pozic, je sekvenční průchod *celého* korpusu nereálný. Velikost datových souborů je u takto rozsáhlých korpusů v řádech gigabytů a i při použití efektivních kompresních technik (viz dále), které mohou zmenšit množství dat až na jednu pětinu, je velikost několik stovek megabytů. Na současných počítačích ovšem trvá i pouhé kopírování 100MB souboru několik (desítek) vteřin (25 vteřin na počítači s procesorem Intel Pentium III 500MHz s rychlými SCSI disky).[1] Prakticky je tedy tento způsob nepoužitelný.

Signatury

Jistým vylepšením této metody je použití tzv. *signatur*. Celý text se nejdříve rozdělí do menších částí a pro každou část je vypočítána signatura, což je

[1]Je nutno ovšem poznamenat, že pokud máme dostatek paměti, a několik set MB je v současné době běžné i na pracovních stanicích, využije operační systém vyrovnávací paměť a opakované kopírování proběhne za asi 5 vteřin. Pokud navíc nebudeme zapisovat na disk a budeme data pouze číst, sníží se čas na uvedeném počítači na pouhé 0,4 vteřiny.

nějaký kód velikostí řádově menší než původní část textu, který obsahuje jistým způsobem zkombinované informace o všech slovech v dané části. Nejčastěji se pro každé slovo volí nějaký bitový kód (bitový vektor) o stejné délce jako celá signatura a kombinací těchto kódů je bitový logický součet (or).

Pro názornost uveďme příklady bitových vektorů pro několik slov z korpusu KOČKA (tabulka 3.1) a odpovídající signatury pro několik řádků korpusu (tabulka 3.2). V příkladu zanedbáváme velikosti písmen a vynecháváme interpunkci.

Slovo	Bitový vektor
a	1000 0100 0000
bos	0000 1000 0100
byl	0001 0100 0000
chodil	0000 0010 1000
den	0100 0000 0010
horký	0001 0001 0000
kdekdo	0100 0000 0001
kos	1001 0000 0000
koukala	0000 1000 0001
kočka	0000 1100 0000
letní	0010 0001 0000
na	0010 0000 0100
okně	0000 0010 0010
se	0000 0001 0100
seděla	0000 0010 0010
ven	1010 0000 0000
venku	0000 0001 0001
zpíval	0100 0000 1000

Tabulka 3.1: Bitové vektory pro část slovníku korpusu KOČKA.

Část textu	Signatura
Kočka na okně	0010 1110 0110
Na okně seděla kočka, byl horký letní den,	0111 1111 0110
na okně seděla kočka a koukala se ven,	1010 1111 0111

Tabulka 3.2: Signatury pro několik částí korpusu KOČKA.

Vyhledávání potom probíhá tak, že se testuje bitový vektor hledaného slova vzhledem ke všem signaturám. Signatura, která obsahuje daný vektor (logický součin signatury a vektoru je roven vektoru), určuje část textu, ve které se slovo *může vyskytovat*. Například bitový vektor 0001 0001 0000 slova *horký* obsahuje pouze signatura druhého řádku a tam je také hledané slovo. Naopak vektor 0010 0001 0000 slova *letní* je obsažen ve všech třech signaturách, přestože je hledané slovo pouze v druhém řádku. Určené části textu je tedy potřeba ještě projít klasickým sekvenčním vyhledáváním.

Signatury můžeme přirovnat k obsahu knihy. Kniha je rozdělena do kapitol a každá kapitola je v obsahu charakterizována svým názvem. Pokud potřebujeme v knize najít nějakou pasáž, projdeme obsah a vytipujeme kapitoly, ve kterých by se hledaná pasáž mohla vyskytovat. Potom vybrané kapitoly postupně projdeme. Obsah zabírá proti celé knize pouze několik stránek.

Určením délky signatur a bitových vektorů a počtu nastavených bitů (jedniček) ve vektorech můžeme volit mezi velikostí dat a počtem dodatečných vyhledávání. Jedním extrémem je situace, kdy každému slovu odpovídá jeden bit signatury, délka signatury je tedy rovna počtu různých slov. Při vyhledávání potom nedochází k mylným určením části textů a sekvenční průchod již není potřeba (pokud se spokojíme pouze s určením části textu). Tato specielní forma signatur se někdy označuje termínem *bitmapy*.

V opačném extrému se délka signatury rovná dvojkovému logaritmu počtu různých slov, každé slovo je identifikováno binárním zápisem svého pořadového čísla. V takovém případě již pro malé části textů dostáváme signatury se všemi bity nastavenými (samé jedničky) a sekvenčně procházíme téměř celý text.

Signatury (a zvláště bitmapy) lze velice dobře komprimovat. Výsledkem může být pouze zlomek paměťových nároků při nepatrném zvýšení celkového času vyhledávání (podrobně rozebráno například v [WMB99]). Použití indexů (viz dále) však většinou vede k rychlejším algoritmům s menšími datovými strukturami. Pro korpusový manažer to platí ještě výrazněji, protože potřebujeme vyhledávat až na jednotlivé pozice (nejenom na části textů) a nemůžeme použít optimalizační techniky stop-listů apod.

3.2.2 Použití indexů

Tato technika využívá dodatečných datových struktur pro přímý přístup k pozicím s danými slovy. Dále popíšeme dvě datové struktury: *reverzní indexy* a *PAT arrays*. Jiné používané struktury jsou vždy jistou modifikací jedné z uvedených. Tyto pomocné struktury jsou co do množství informace ekvivalentní vlastnímu korpusu – veškerá informace je v nich uložena, pouze v jiném uspořádání. Tedy nejenom že z korpusu lze vytvořit například jeho reverzní index, ale i naopak z reverzního indexu lze zpětně vytvořit vlastní korpus.

Indexy můžeme přirovnat k rejstříku knihy (ne náhodou je *rejstřík* anglicky *index*). Hledané slovo nebo pojem si podle abecedy najdeme v rejstříku a okamžitě máme čísla stránek s výskyty daného slova. Pro korpus ovšem indexujeme všechna slova a odkazy jsou místo stránek přímo na pozice v textu.

Reverzní indexy

Reverzní index, často též označovaný *invertovaný soubor (inverted file)*, obsahuje pro každé slovo (type) z korpusu samostatně seznam pozic, na kterých se dané slovo vyskytuje. Opět si pro názornost uveďme část reverzního indexu pro korpus KOČKA (v tomto příkladě nezanedbáváme velikosti písmen ani interpunkci).

Slovo	Seznam pozic
Kočka	1
na	2 14 32 70
okně	3 5 15 33 71
Na	4
seděla	6 16 34 72
kočka	7 17 35 73
,	8 13 22 31 36 58 64 69
byl	9 23
horký	10 24 42
letní	11 25 43
den	12 26 44

Tabulka 3.3: Část reverzního indexu korpusu KOČKA

Použitím reverzního indexu je vyhodnocení jednoduchého dotazu s jedním slovem triviální – výsledkem je seznam pozic přímo uložený v reverzním indexu, jedinou náročnější operací zůstává vyhledání slova ve slovníku. Pokud dotaz obsahuje regulární výraz (pro jeden poziční atribut), musíme prohledat slovník (mnohdy sekvenčně – záleží na typu regulárního výrazu) a sloučit (množinově sjednotit) příslušné seznamy pozic všech vyhovujících slov. Posloupnost pozic v dotazu se řeší spojením (po „posunutí“ množinově průnik) příslušných seznamů.

Přestože indexy obsahují více informací než signatury, pomocí pokročilých technik komprese je můžeme uložit v mnohem menším prostoru. Podrobnosti budou popsány dále.

PAT arrays

PAT arrays (PAT-pole), někdy též nazývaná *sufixová pole (suffix arrays)*, jsou efektivní implementací *PAT-stromů (PAT trees)*. PAT-stromy jsou speciálním případem *stromů Patricia* (viz například[Knu73]), což jsou vyhledávací stromy založené na porovnávání potenciálně nekonečných řetězců. Podrobnější popis lze nalézt například v [FBY92, kapitola 5].

Výhodou PAT-stromů je možnost snadno a efektivně vyhledávat sekvence slov libovolné délky a různé speciální dotazy (nejdelší opakující se řetězec, přibližné vyhledávání posloupností slov atd.) těžko realizovatelné pomocí jiných datových struktur.

Naopak nevýhodou je pomalé vyhodnocování „víceúrovňových“ dotazů, které obsahují podmínky pro několik různých pozičních atributů. Celkově se tedy PAT-stromy spíše hodí pro aplikace plnotextového vyhledávání než pro korpusový manažer.

Shrnutí

Jako hlavní pomocná struktura pro vyhledávání v korpusovém manažeru je nejvýhodnější reverzní index, proto byl zvolen pro implementaci. Drtivá většina dotazů se vyhodnocuje pomocí reverzních indexů.

Některé dotazy jsou ovšem svou podstatou natolik složité, že k jejich vyhodnocení nám reverzní index nijak nepomůže, dokonce použití sekvenčního prohledání korpusu může být rychlejší. Takovéto dotazy nemusejí mít nijak složitou strukturu. Příkladem může být dotaz, který se snaží najít v korpusu zkratky psané velkými písmeny. Dotaz by tedy mohl vypadat takto:

```
"[[:lower:]].*" "[[:upper:]]{2,}" "[[:lower:]].*"
```

Jsou to tedy slova složená ze dvou či více velkých písmen, která jsou zprava i zleva obklopena slovy začínajícími malými písmeny. Při použití indexů by musel manažer spojit až několik milionů seznamů (pro každé slovo začínající malým písmenem jeden seznam), což by v mezivýsledku (před spojením se slovy s velkými písmeny) dávalo téměř celý korpus. Celková složitost by se tedy blížila vytvoření korpusu z reverzního indexu.

3.3 Poziční atributy

Z technického pohledu jsou poziční atributy posloupností slov (nebo jiných znakových řetězců). Celkově je poziční atribut reprezentován čtyřmi do jisté míry samostatnými moduly. Jak uvidíme dále, některé moduly bude možné beze změny použít i v jiných částech korpusového manažeru.

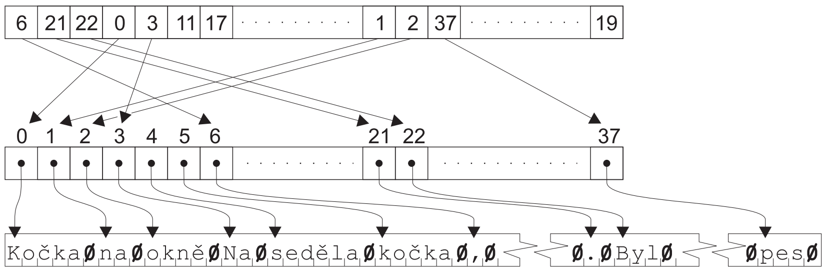
Všechna slova z daného pozičního atributu (například slovo, nebo základní tvar) jsou uložena ve *slovníku*. Slovník zajišťuje „očíslování“ slov, tedy převod z řetězce na jednoznačné číslo a zpět. Ve vlastním *korpusu* jsou pak uvedena pouze čísla. Pro rychlejší vyhledávání i pro urychlení některých výpočtů se využívají *reverzní indexy*, ve kterých je pro každé slovo (zadané svým číslem) uveden seznam všech jeho výskytů v korpusu (pozic). Tento seznam je setříděný a umožňuje rychlé (logaritmické) vyhledávání. Dále je pro každé slovo ve slovníku přístupný počet výskytů daného slova v korpusu.

3.3.1 Slovník

Slovník zajišťuje jednoznačné zobrazení znakových řetězců na přirozená čísla a zpět. Zaměřme se nejdříve na převod z řetězce na číslo. Ten se používá hlavně při dotazování všeho druhu, protože v dotazu zadáváme přirozeně přímo slovo a nikoliv jeho číslo. Většina dotazů obsahuje pouze malý počet slov, takže převod *slovo* $\rightarrow$ *číslo* v tomto případě není nijak kritický. Druhou oblastí, kde je potřeba převod ze slova na číslo, je počáteční kódování korpusu, kdy je nutné každé slovo ze vstupního textu převést na číslo. Tady může být pomalý převod úzkým hrdlem celého kódování, na druhé straně se kódování provádí pouze jednou, dávkově.

Druhý směr se používá při každé prezentaci libovolných výsledků, například při zobrazování výsledků dotazu s kontexty je tato operace klíčová. Musíme tedy zajistit prakticky okamžitý (s konstantní složitostí) převod z čísla na slovo. Nejjednodušší struktura s konstantním přístupem je homogenní pole, kde každý prvek je stejného typu (a stejné velikosti). Bohužel nemůžeme omezit velikost slova ve slovníku nějakou rozumně malou konstantou (třeba 15 znaků na slovo), protože se téměř v každém textu vyskytují některé výjimky přesahující tento limit. Musíme tedy ukládat řetězce bez omezení velikosti.

Přímočará struktura vyhovující předchozím požadavkům se skládá ze dvou částí: vlastních slov a pole ukazatelů. Oblast pro slova obsahuje všechna slova uložená přímo za sebou, oddělená nějakým oddělovačem (speciálním znakem, nejčastěji binární nulou), nebo s uloženou velikostí slova. Na každé slovo vede odkaz z homogenního pole ukazatelů, jehož indexy reprezentují jednoznačná čísla jednotlivých slov. Schematicky je příklad takového slovníku zobrazen na obrázku 3.1. Převod *číslo* → *slovo* je okamžitý: z pole na pozici *číslo* vybereme ukazatel, který odkazuje přímo na požadované slovo. Opačný převod bychom museli realizovat sekvenčním prohledáním. Musíme tedy vytvořit pomocnou setříděnou strukturu, nebo musí být čísla slovům přiřazena podle abecedy, tedy 0 bude mít první slovo podle abecedy, 1 druhé, 2 třetí atd. V obou případech potom můžeme vyhledávat slovo binárním vyhledáváním, tedy v logaritmickém čase vzhledem k velikosti slovníku.



Obrázek 3.1: Datová struktura slovníku pozičního atributu.

Na první pohled je výhodnější použití vhodného očíslování slov, protože nepotřebujeme žádné pomocné struktury. V některých případech ovšem budeme potřebovat, aby slovník byl uspořádaný podle jiných veličin než podle abecedy, například podle četnosti slova v korpusu. Potom samozřejmě musíme použít druhou variantu. První varianta má ještě jednu výhodu: vlastní slova lze poměrně jednoduchým způsobem komprimovat. Využívá se při tom vlastnosti, že za sebou jdoucí slova mají jistou část (prefix) společnou. Ve slovníku potom uložíme vždy jenom rozdíl oproti předchozímu slovu. Na celkovou rychlost zpracování nemá tento přístup velký vliv a přitom lze dosáhnout až 20% úsporu místa. Podrobnější diskuse je uvedena v [WMB99, kapitola 4.1].

Jinou možností, jak efektivně vyhledávat ve slovníku je struktura *trie*, která umožňuje tzv. *digitální vyhledávání*. Většinou je svým rozsahem větší než výše uvedené setříděné pole čísel, a tak, i když probíhá vyhledávání rychleji (časová složitost je úměrná délce hledaného slova), lze se spokojit s mnohem jednodušší metodou pole. Datová struktura trie je popsána například v [Knu73], výrazné optimalizace velikosti jsou podrobně rozebrány v [Sed99].

3.3.2 Text korpusu

V *textu korpusu* je uložena posloupnost slov reprezentující korpus. Nejsou v něm přímo slova, ale pouze identifikátory (čísla) slov. Jedinou operací, kterou na textu korpusu vyžadujeme, je určení čísla slova na zadané pozici.

Nejjednodušší struktura pro text korpusu je opět homogenní pole čísel slov, a pro menší korpusy je naprosto vyhovující. Pro větší (stovky milionů pozic) bychom dostali datové sobory v řádech GB, což už přestává být únosné.[2] Musíme tedy použít některou kompresní techniku tak, aby častější slova (jejich čísla) byla zakódována na menší počet bitů než méně častá slova.

Téměř optimální (minimální) velikost dosáhneme použitím *Huffmanova kódování* (původně prezentováno v [Huf52]), jehož efektivní konstrukce z četností jednotlivých slov je diskutována v [WMB99]. I když existují modifikace Huffmanova kódování umožňující průběžné rozšiřování slovníku, nejvyšší komprese a nejjednodušší algoritmy získáme při *statickém* kódování, kdy jednotlivé kódy slov jsou určeny na základě celkových statistik, tedy až po zpracování celého korpusu. Počáteční vytváření korpusu v takovém případě musíme dělat nadvakrát: v prvním průchodu zdrojových textů (vertikálních textů, XML) vytváříme slovník a počítáme četnosti jednotlivých slov; v druhém průchodu potom zapisujeme text korpusu v Huffmanově kódování. Alternativou je použití nějakého *dynamického* kódu.

Texty v přirozených jazycích mají jednu přirozenou vlastnost, že nejčetnější slova se poprvé v textu objevují na jeho začátku. Pokud tedy budeme přidělovat čísla slovům podle jejich prvních výskytů (jako například na obrázku 3.1), bude zhruba platit, že nižší čísla slov znamenají vyšší četnost slova v textu. Pro kompresi textu korpusu tak můžeme využít kódy pro zápis přirozených čísel, které zapisují menší čísla na menší počet bitů. Příklady takovýchto kódů jsou uvedeny v tabulce 3.4. Binární kód je samozřejmě omezený pouze na čísla v určitém intervalu ($< 0, 2^n - 1 >$, kde n je počet bitů), ostatní kódy umožňují zapsat libovolné přirozené číslo. Unární kód je svojí podstatou (délka v bitech je přímo úměrná velikosti čísla) nepoužitelný pro zápisy velkých čísel, ale v některých situacích je jeho použití oprávněné. Konstrukce unárního kódu je zřejmá, γ-kód a δ-kód se vždy skládají ze dvou částí (v tabulce oddělených malou mezerou): první udává velikost druhé části v bitech, druhá potom obsahuje vlastní číslo v binárním kódování. První část γ-kódu je unární kód, první část δ-kódu je γ-kód.

Až překvapivé výsledky dává použití δ-kódu pro kódování textu korpusu při očíslování slovníku slov podle jejich prvního výskytu. Pro menší korpusy (do desítek milionů pozic), ve kterých se ještě velikost slovníku

[2] Většina současných operačních systémů používá 32-bitovou adresaci souborů, díky níž jsou schopny zpracovávat soubory do velikosti pouze 2 GB.

číslo	unární	binární	γ-kód	δ-kód
0	1	0000	1	1
1	01	0001	01 0	010 0
2	001	0010	01 1	010 1
3	0001	0011	001 00	011 00
4	00001	0100	001 01	011 01
5	000001	0101	001 10	011 10
6	0000001	0110	001 11	011 11
7	00000001	0111	0001 000	00100 000
8	000000001	1000	0001 001	00100 001
9	0000000001	1001	0001 010	00100 010

Tabulka 3.4: Příklady bitových zápisů kódování přirozených čísel.

podstatnou měrou odráží v celkové velikosti dat korpusu, je výraznou úsporou místa absence dalších pomocných struktur (pro Huffmanovo kódování potřebujeme pro každé slovo uchovávat jeho kód). Celkově například text korpusu Desam zaznamenaný v δ-kódu má téměř stejnou velikost (o 3 % větší) jako zakódovaný Huffmanovým kódem. Jistou modifikací δ-kódu dosáhneme dokonce o více než 8 % *menší* velikost.

3.3.3 Reverzní index

Reverzní index obsahuje pro každé slovo ze slovníku seznam pozic s jeho výskyty v korpusu. Pokud jednotlivé seznamy uspořádáme podle velikosti (a to je výhodné i z hlediska vyhledávání), můžeme ukládat pouze rozdíly mezi po sobě jdoucími prvky seznamu. Ukládáme celkově menší čísla a opět můžeme využít kódy popsané v předešlé kapitole. V tabulce 3.5 jsou uvedeny „rozdílové“ seznamy pozic odpovídající příkladu z tabulky 3.3.

V [WMB99] autoři podrobně diskutují použití různých bitových kódů v reverzním indexu (kromě γ- a δ-kódů popisují i několik dalších) a například δ-kód je vhodnou variantou. Použití všech uvedených metod má jednu nevýhodu: zmíněné kódy zabírají pro různá čísla různý počet bitů (tím se dosahuje komprese), a ze své podstaty tak vynucují sekvenční přístup. Pro slova s menší četností to nijak nevadí, ale pro seznamy s více než stovkami tisíc odkazů[3] potřebujeme pro efektivní vyhodnocování složitějších dotazů rychlý přístup do seznamu pozic. Musíme tedy prostý seznam

[3]Aplikace Information Retrieval většinou taková slova vůbec neobsahují. Odkazy jsou vedeny pouze po dokumentech a slova vyskytující se ve většině dokumentů jsou obsahově nevýznamná, tudíž se ani neindexují.

Slovo	Seznam rozdílů pozic
Kočka	1
na	2 12 18 38
okně	3 2 10 18 38
Na	4
seděla	6 10 18 38
kočka	7 10 18 38
,	8 5 9 9 5 22 6 5
byl	9 14
horký	10 14 18
letní	11 14 18
den	12 14 18

Tabulka 3.5: Část reverzního „rozdílového“ indexu korpusu KOČKA

pozic v reverzním indexu obohatit o další informace (kontrolní body), které by umožnily efektivní vyhledání určité hodnoty.

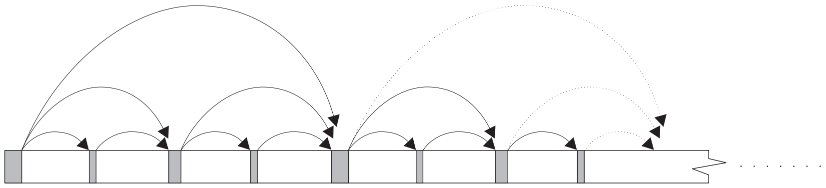
V principu se nabízí dvě možnosti řešení daného problému:

- Do posloupností vložíme vždy po pevném počtu pozic (v řádech desítek až stovek) odkaz na místo s dalším odkazem. Při vyhledávání potom můžeme daný počet pozic „přeskočit“. Pro opravdu velké seznamy, musíme vytvořit odkazy několikaúrovňově, tedy přidáme ještě další odkazy přeskakující vždy několik základních odkazů. Schematicky je situace zachycena na obrázku 3.2.

- Místo v paměti pro uložení celé posloupnosti pozic rozdělíme na bloky stejných velikostí a na začátek každého bloku umístíme kontrolní informace o pozici, kterou blok začíná. Při vyhledávaní potom podle potřeby přeskakujeme celé bloky nebo skupiny bloků. Samozřejmě může nastat situace, kdy na konci jednoho bloku zůstane „volných“ (nevyužitých) několik bitů, protože další pozice již potřebuje bitů více. V takových případech je nutné použít synchronizační kódy, které volné místo detekují.

3.4 Dynamické atributy

Získání hodnoty dynamického atributu znamená vyhodnocení určité funkce. Protože se tato operace může provádět často (například pro každou pozici v kontextu při vypisování výsledků dotazu), je nutné ji vyhodnotit co



Obrázek 3.2: Rozšíření reverzního indexu o odkazy.

nejrychleji, na co nejnižší úrovni. Nejlepší je volání funkce přímo v jazyce C. Naopak musíme uživatelům dovolit definovat si svoje vlastní funkce (dynamické atributy) bez zásahu do korpusového manažeru (úpravy zdrojových kódů a kompilace). Moderní operační systémy umožňují přidávat externí funkce k běžícímu programu pomocí *sdílených (dynamicky linkovaných) knihoven*, což je jistě ideální řešení dynamických atributů.

Formálně je tedy dynamický atribut definován deklarací externí funkce a určením cesty ke sdílené knihovně s touto funkcí. Pokud uživatel potřebuje vytvářet funkce programovacím jazyce, který neumožňuje vytvoření sdílené knihovny, lze předdefinovat funkce poskytující jiné aplikační rozhraní.

Pro dynamické atributy, které mohou uživatelé používat v dotazech, je nutné vytvořit dodatečné reverzní indexy pro vyhledávání. Teoreticky bychom mohli vytvořit klasický reverzní index jako pro poziční atribut, ale ztratili bychom tak hlavní výhodu dynamických atributů, že nejsou nijak paměťově náročné. Index tedy nebude obsahovat pro každou hodnotu dynamického atributu seznam pozic, ale pouze seznam hodnot pozičního atributu, od kterého je odvozen. Například dynamický atribut reprezentující gramatický rod je závislý na pozičním atributu gramatické značky. Pro hodnotu ženského rodu potom index obsahuje všechny gramatické značky (resp. jejich čísla ze slovníku), které obsahují ženský rod.

Fyzické uložení takovýchto indexů bude stejné jako pro reverzní indexy pozičních atributů. Velikost dat je potom úměrná velikosti slovníku příslušného pozičního atributu a ve srovnání se zbytkem korpusu zanedbatelná. Při vyhodnocování dotazů musí manažer provést sloučení seznamů pozic příslušejících všem slovům pro danou hodnotu dynamického atributu.

3.5 Struktury

Pro obecné struktury musíme být schopni uložit a efektivně vyhledat každý interval, který je dán svým začátkem a koncem (v pozicích). I v tomto případě se nabízí použití homogenního pole, ve kterém každý prvek obsahuje čísla dvou pozic – začátku a konce. I když můžeme aplikovat stejné metody komprese jako v případě reverzních indexů nebo textu korpusu, většinou s touto strukturou vystačíme, protože zabírá pouze zlomek paměti zabrané celým korpusem. Výjimku mohou tvořit struktury, které jsou v korpusu použity ve velkém množství.

Plochá struktura (nepřekrývající se, nevnořující se) nepotřebuje pro vyhledávání žádné dodatečné prostředky. Intervaly uložíme v poli setříděně (podle začátku i konce) a můžeme vyhledávat podle obou hranic binárním vyhledáváním v logaritmickém čase.

Vnořovaná struktura s neomezenou hloubkou vnoření již potřebuje k efektivnímu vyhledávání podle konce intervalu (předpokládáme, že pole je setříděno podle začátku intervalů) pomocný index. Ten může být realizován stejným způsobem jako pro vyhledávání ve slovníku pozičních atributů, tedy pole obsahující indexy do základního pole intervalů. Popsaná pole intervalů s indexem dokonce můžeme použít pro zcela obecnou strukturu, která dovoluje nejen libovolné vnořování jednotlivých intervalů, ale i jejich překrývání.

Pro prázdné struktury samozřejmě ukládáme místo dvou hodnot intervalu (začátku a konce) pouze jednu hodnotu.

Atributy struktur

Atributy struktur mohou být uloženy úplně stejným způsobem jako poziční atributy. Pouze interpretace „pozic“ je odlišná – reprezentují jednotlivé intervaly.

Zarovnání a ukazatele

Zarovnání dvou korpusů je realizováno jako propojení dvou struktur, každé v jednom korpusu. Technicky může být jeden směr uložen jako index struktury: pole, jehož prvky odkazují na pořadová čísla struktur v druhém korpusu. Pokud potřebujeme zaznamenat obousměrné zarovnání, uložíme každý směr zvlášť.

Ukazatele z jedné pozice na jinou pozici si můžeme představit jako intervaly: začátek intervalu reprezentuje pozici, odkud vede ukazatel, konec naopak, kam vede ukazatel. Můžeme tedy použít stejné metody, jako pro uložení obecných struktur. Drobnou odlišností ukazatelů je, že mohou ukazovat „zpět“, tedy odpovídající interval by měl konec před začátkem. I tuto podmínku ovšem výše popsaná metoda splňuje.

3.6 Modulární struktura manažeru

Protože jednotliví uživatelé potřebují přistupovat ke korpusům z dost rozdílných úhlů, nemůže korpusový manažer tvořit jedna monolitická aplikace. Nejnižší rozhraní, které budou využívat zejména programátoři aplikací využívající korpus, se skládá z knihovny funkcí v jazyce C. Knihovna zprostředkovává přímý přístup k datům: pozičním i dynamickým atributům, strukturám; pomocí dotazů umožňuje vytvářet konkordance a zpracovávat je atd. Takové rozhraní lze snadno (mnohdy i automaticky) přizpůsobit vyšším programovacím jazykům (jako Perl, Python, Tcl atd.).

Další rozhraní by mělo být ve formě příkazového jazyka (shellu), který by poskytoval komplexní funkce typu: vytvoření konkordance dotazem a její výpis ve formátu KWIC, výpočet kolokací z celého korpusu apod. Toto rozhraní umožní uživatelům zadávat jednoduché dávky, aniž by museli znát nějaký programovací jazyk. Nejvyšší úrovní je potom GUI, které uživatele vede při všech jeho činnostech.

3.6.1 Rozdělení na klient a server

Využití architektury klient/server je přirozené pro všechny aplikace pracující s velkým množstvím dat, což korpusy jsou. Otázkou je, které funkce manažeru by měl zajišťovat server, a které klient. Základní filosofií této architektury je, že zpracování datově náročných operací na serveru je výhodnější (rychlejší), než pouhé předávání dat klientovy, který si provádí operaci sám. Snažíme se tedy optimalizovat (minimalizovat) celkový tok dat mezi serverem a klientem.

Komunikační protokol

Z výše uvedeného vyplývá, že není vhodné realizovat rozhraní mezi klientem a serverem na nejnižší úrovni přímého přístupu k datům.[4] Dobrou volbou může být úroveň příkazového jazyka. To také odpovídá mnohým osvědčeným Internetovým protokolům (FTP, HTTP atd.), ve kterých je komunikace vedena výhradně textově. Jednotlivé programy (server i různé druhy klientů) potom mohou programátoři vyvíjet samostatně s minimálními nároky na ladící nástroje.

V tomto duchu je navržen protokol k manažeru SARA i komunikace mezi `cqsd` a GCQP. I jako ilustrace příkazového jazyka je posledně jmenovaný protokol popsán v příloze C.

[4] Tento přístup byl použit v korpusovém manažeru CQP, ale pro naprostou nepoužitelnost byl z něho nakonec odstraněn.

3.6.2 Konfigurační soubor korpusu

Jednotlivé korpusy se od sebe navzájem liší, pro každý může být výhodnější jiná datová struktura. Manažer tedy musí být dostatečně konfigurovatelný, a to nejen celkově, ale i pro jednotlivé korpusy. Pro každý korpus uživatel (správce) musí v konfiguračním souboru specifikovat všechny poziční a dynamické atributy, struktury atd. Navíc může zvolit jinou než standardní metodu implementace některé části korpusu. Například pro strukturu, o které ví, že se v korpusu bude vyskytovat často, zvolí komprimované uložení. Dále může uživatel zadat různá kódování pro poziční atributy, rozhodnout, jestli může atribut obsahovat multihodnoty. Také lze přidat popisy a vysvětlení atributů, popř. hodnot atributů, které se použijí při prezentaci výsledků nebo v GUI.

V případě vzdáleného přístupu ke korpusu, může správce zadat omezení pro daný korpus, popř. pouze pro zadané uživatele. Omezení mohou být několika typů:

- Maximální počet řádků ve výsledku dotazu (tzv. *HardCut*). Tímto omezením můžeme zkrátit vyhodnocování mnoha náročných dotazů, a tak například kontrolovat zátěž systému. Samozřejmě tím omezujeme použitelnost korpusu, takže to nelze použít pro všechny uživatele.
- Určení kontextu při vypisování výsledků. Tak můžeme zabránit zobrazení větších částí textu (například celých dokumentů), pokud v tom brání licenční ujednání.
- Zamezení přístupu k některým (zvláště dynamickým) atributům může opět snížit celkové zatížení systému.

3.7 Vyhodnocování dotazů

Vyhodnocování dotazů je založeno hlavně na reverzních indexech. Dotazy obsahující struktury nebo ukazatele jsou samozřejmě zpracovávány i s použitím odpovídajících datových struktur. Klíčovou roli v každém vyhodnocení dotazu hraje třída `FastStream`, od které jsou odvozeny všechny konkrétní prvky, které se v dotazech používají.

Třída `FastStream`

Instance třídy `FastStream` reprezentují proud čísel (většinou pozic), který musí splňovat následující podmínky:

- čísla jsou v proudu setříděna vzestupně,
- průchod proudem (jeho čtení) je realizováno pouze sekvenčně (opakovaným voláním metody `next`), není tedy možné se vracet k dříve přečteným číslům,
- konec proudu je označen zarážkou (volání metody `next` vrací hodnotu zarážky), což je číslo větší než libovolné číslo obsažené v proudu,
- metoda `peek` „nahlíží“ do proudu – vrací hodnotu, kterou vrátí další volání metody `next`,
- metoda `find` efektivně (logaritmicky vzhledem k délce proudu) hledá zadané číslo v proudu.

3.7.1 Operace AND – spojení proudů pozic

Operace AND je jednou ze zcela základních, jak dále popíšeme, velké množství prvků dotazovacího jazyka bude převedeno právě na operaci AND. Vstupem do binární operace AND jsou dva proudy pozic, výstupem je proud obsahující pozice společné oběma vstupním proudům.

Tady můžeme dobře využít jednu definiční podmínku třídy `FastStream`, že jsou v něm čísla setříděna. Celá operace tedy může proběhnout pouze s minimálními nároky na paměť: u každého proudu si pamatujeme pouze aktuální číslo – jakýsi ukazatel do proudu. Výsledek je přirozeně také setříděn, takže by mohl splňovat podmínky kladené na `FastStream`.

Nejjednodušší implementace prochází sekvenčně oba proudy a porovnává jejich hodnoty. V každém kroku se posune na další číslo v tom proudu, ve kterém je aktuální číslo s menší hodnotou. Téměř všechny metody `FastStream` jsou zde triviální, jedinou výjimku tvoří metoda `next`. Tu můžeme schematicky implementovat takto:

```
AND1::next ():
    while src1.peek() != src2.peek():
        if src1.peek() < src2.peek():
            src1.next()
        else:
            src2.next()
    return src1.next()
```

Uvedený kód lze samozřejmě optimalizovat, aby se zbytečně nevolala metoda `peek()`, která dává v po sobě následujících voláních vždy stejnou hodnotu. Podobné optimalizace lze udělat i u většiny dalších ukázek, ale

protože se jedná spíše o technický detail (a kvalitní kompilátor by teoreticky mohl provést navrhovanou optimalizaci automaticky), zůstaneme raději u přehlednějších forem zápisu.

V algoritmu jsme využili další pěknou vlastnost struktury `FastStream`: konec proudu je značen zarážkou. V cyklu tedy nikde explicitně netestujeme konec libovolného z proudů, metoda v nejhorším případě skončí právě zarážkou a každé další volání metody `next` vrátí stejnou hodnotu. To ale platí pouze za předpokladu, že oba proudy končí stejnou zarážkou, což můžeme celkem snadno zajistit.

Uvedený algoritmus prochází sekvenčně každý z proudů, a tak zpracování výsledku operace AND zabere zřejmě $O(m+n)$ času, kde m a n jsou velikosti vstupních proudů. Když nepočítáme uložení vstupních proudů, tak paměťové nároky vlastní operace AND jsou konstantní (a v podstatě minimální). Konstantní paměť a lineární čas, který vlastně odpovídá pouhému přečtení vstupu, může navozovat dojem, že máme řešení optimální. Není tomu ovšem tak. Ještě můžeme lépe využít setřídění čísel v proudu a s pomocí metody `find` podstatně snížit časové nároky se zachováním konstantní paměti.

Uvedený postup lze naprogramovat i pomocí vložených cyklů:

```
AND2::next ():
    while src1.peek() != src2.peek():
        if src1.peek() < src2.peek():
            src1.next()
        else:
            while src1.peek() > src2.peek():
                src2.next()
    return src1.next()
```

Celková složitost se zde nezměnila, ale vnitřní cyklus nyní můžeme nahradit voláním metody `find`:

```
AND3::next ():
    while src1.peek() != src2.peek():
        if src1.peek() < src2.peek():
            src1.next()
        else:
            src2.find(src1.peek())
    return src1.next()
```

Nyní procházíme sekvenčně pouze první proud, ve druhém rychle (logaritmicky) dohledáváme. Vyjádření Časové složitosti ale není tak jednoduché. Z podmínek, které musí splňovat metoda `find` vyplývá, že v celkovém průchodu se budou vždy střídat volání `src1.next()` (uvnitř podmínky nebo při návratu z metody) a `src2.find()`. Metoda `src1.next()` se provede m-krát a metoda `src1.next()` tudíž $O(m)$-krát (přesněji $(m \pm 1)$-krát). Každá z operací `next` zabere konstantní čas, operace `find` logaritmický čas vzhledem k „přeskočené" části proudu. Formálně to bude

$$\sum_{i=1}^{m} \log a_i \quad \text{, přičemž} \quad a_i > 0, \sum_{i=1}^{m} a_i = n$$

Takovýto výsledek nás nemůže uspokojit, protože je velice těžko porovnatelný s původní složitostí. Sumu ale můžeme odhadnout shora na základě následující věty.

Věta 1 Pro libovolnou posloupnost čísel $a_1, a_2, \ldots, a_n$ takových, že $\forall i : a_i > 0$, platí

$$\sum_{i=1}^{n} \log a_i \leq n \log \frac{\sum_{i=1}^{n} a_i}{n}. \tag{3.1}$$

Přitom rovnost nastává právě tehdy, když $a_1 = a_2 = \ldots = a_n$.

Důkaz: pro $n = 1$ máme přímo identitu, pro $n = 2$ provedeme přímo, pro $n > 2$ provedeme sporem. Pokud jsou si všechny členy posloupnosti rovny, je rovnost zřejmá.

$$\begin{aligned}
\log a + \log b &\leq 2 \log \frac{a+b}{2} \\
\log(ab) &\leq \log \frac{(a+b)^2}{4} \\
ab &\leq \frac{(a+b)^2}{4} \\
4ab &\leq a^2 + 2ab + b^2 \\
0 &\leq a^2 - 2ab + b^2 \\
0 &\leq (a-b)^2
\end{aligned}$$

Nyní mějme $n > 2$ a předpokládejme, že existuje posloupnost $a_1, a_2, \ldots, a_n$, pro kterou neplatí (3.1). Označme $s = \sum_{i=1}^{n} a_i$ a ze všech posloupností, které nesplňují (3.1) a mají součet s, vyberme tu s maximálním $\sum_{i=1}^{n} \log a_i$. Máme tedy posloupnost $b_1, b_2, \ldots, b_n$, pro kterou platí:

$$\sum_{i=1}^{n} \log b_i > n \log \frac{s}{n}$$

$$\forall a_1, a_2, \ldots, a_n; \quad \sum_{i=1}^{n} a_i = s : \quad \sum_{i=1}^{n} \log a_i \leq \sum_{i=1}^{n} \log b_i$$

V této posloupnosti zřejmě musí existovat minimálně dva členy, které si nejsou rovny. Bez újmy na obecnosti předpokládejme, že $b_1 \neq b_2$. Nyní platí:

$$\sum_{i=1}^{n} b_i = b_1 + b_2 + \sum_{i=3}^{n} b_i < 2 \log \frac{b_1 + b_2}{2} + \sum_{i=3}^{n} b_i$$

S posloupností $\frac{b_1+b_2}{2}, \frac{b_1+b_2}{2}, b_3, b_4, \ldots, b_n$ se tak dostáváme do sporu s původní maximalitou součtu $\sum_{i=1}^{n} \log b_i$ a předpoklad neplatnosti (3.1) není splněn.

Použitím dokázané věty můžeme časovou složitost operace AND nad proudy shora odhadnout a výsledkem je $O(m + m \log \frac{n}{m})$. Přitom nejhorší případ nastává, pokud jsou jednotlivá čísla v obou proudech rozložena rovnoměrně. Získali jsme tak podstatné urychlení operace AND (samozřejmě předpokládáme $m < n$, tedy sekvenčně procházíme ten proud, který má méně prvků).

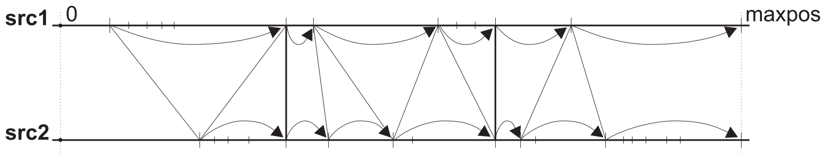
Dalšího zrychlení můžeme docílit stejným trikem (tedy „logaritmickým přeskakováním" většího počtu prvků v proudu) použitým na první proud (s menším počtem prvků). Algoritmus by tedy mohl vypadat takto:

```
AND4::next ():
    while src1.peek() != src2.peek():
        if src1.peek() < src2.peek():
            src1.find(src2.peek())
        else:
            src2.find(src1.peek())
    return src1.next()
```

V tomto případě se v cyklu `while` stále střídají obě větve podmínky `if`, dokud není nalezena stejná hodnota v obou proudech, která je vrácena jako výsledek. Každé volání metody `find` má hodnotu parametru větší, než je aktuální hodnota (vrácená předchozím voláním metody `peek`) volaného proudu, a voláním se tedy přeskočí minimálně na následující prvek. V každém volání `AND4::next` se tedy tělo cyklu provede maximálně tolikrát, kolikrát by se provedlo při stejném volání tělo cyklu v `AND4::next`. Celkově můžeme časovou složitost vyjádřit vztahem

$$O(s + s \log \frac{mn}{s^2}), \tag{3.2}$$

kde s je počet skupin, které vzniknou rozdělením posloupností čísel vstupních proudů do skupin, pro něž neexistují v opačném proudu hodnoty, které by ležely mezi dvěma hodnotami jedné skupiny. Pro názornost je na obrázku 3.3 graficky zobrazen průchod metodou `AND4::next`. Čárky u každého z proudů značí jednotlivé hodnoty v proudu. Oblouk z jedné hodnoty do další reprezentuje volání metody `find` pro příslušný proud, rovná čára spojující hodnoty obou proudů reprezentuje porovnání hodnot v podmínce `if`. Pokud je tato svislá čára tučně, nastává shoda a dané číslo je vráceno jako návratová hodnota.



Obrázek 3.3: Grafické znázornění průchodu metodou `AND4::next`

Obecně už nelze složitost (3.2) více upřesnit bez znalosti vstupu. Z praktického hlediska by se hodila alespoň průměrná nebo očekávaná složitost, to ale silně závisí na zadaných vstupech. I když se omezíme pouze na vyhodnocování korpusových dotazů, typy dotazů od jednotlivých uživatelů se natolik různí, že nelze rozumný odhad časové složitosti provést. Na závěr poznamenejme, že i u posledního případu (`AND4::next`) stále dosahujeme konstantní paměťové náročnosti operace.

3.7.2 Další jednoduché operace nad pozicemi

V této podkapitole popíšeme operace, které zpracovávají proudy pozic – tedy ne intervalů.

Operace OR

Vstupem do binární operace OR jsou dva proudy a výstupem je proud, který obsahuje sjednocení prvků obou proudů. Výsledek tedy obsahuje minimálně tolik prvků, jako větší ze vstupů. To nám dává dolní odhad časové složitosti celkového průchodu výstupním proudem: musí být minimálně rovný délce výstupu, tedy $O(\min(m, n))$, kde m, resp. n jsou počty prvků v jednotlivých vstupních proudech.

Sjednocení provedeme paralelním sekvenčním čtením obou vstupních proudů, přičemž na výstup dáme vždy menší číslo. Jediným drobným problémem, na který musíme dát pozor, je možnost stejných hodnot v obou proudech. V takovém případě dáváme hodnotu na výstup pouze jednou.

Rychlé vyhledávání provedeme voláním metody `find` v obou proudech a porovnáním nalezených hodnot.

Operace NOT

Operace NOT vyžaduje pouze jeden vstupní proud a na výstup dává čísla, která nejsou v původním proudu až po zarážku. Realizujeme ji opět přímočaře sekvenčním průchodem vstupního proudu a voláním `find` pro vyhledávání.

3.7.3 Jednoduché operace nad intervaly

Operace nad intervaly jsou analogické operacím nad jednotlivými pozicemi. Protože ale každý prvek (interval) obsahuje dvě hodnoty, dostáváme mnohem více variant jedné operace.

Podle operace AND můžeme pouhou změnou testovacích podmínek realizovat následující operace nad intervaly:

AND porovnání na přesnou shodu intervalů,

IN interval druhého proudu je uvnitř intervalu prvního proudu,

CONTAIN interval druhého proudu je obsažen v intervalu prvního proudu,

INTERSECT průnik dvou intervalů.

Drobnou modifikací předchozích můžeme též vytvořit třídy pro operace s různými hraničními podmínkami. Například pro operaci IN můžeme vyžadovat shodu začátků intervalů apod.

3.7.4 Převod dotazu na jednoduché operace

Dotaz se obecně skládá až z regulárních výrazů nad pozicemi, což neodpovídá přímo výše uvedeným operacím.

Booleovské výrazy

V rámci jedné pozice může dotaz obsahovat kombinace operátorů AND, OR nebo NOT. Ty můžeme přímo převést na odpovídající proudy. V některých speciálních případech lze vyhodnocení dotazu urychlit převodem booleovského výrazu do normální formy, popř. eliminací tautologií. Obecně ale uvedené optimalizace nemusejí být přínosem, dokonce existují případy, kdy provedení optimalizace výpočet naopak zpomalí.

Pokud dotaz obsahuje regulární výraz v hodnotě pozičního atributu, musíme nejdříve prohledat celý slovník daného atributu a všechny odpovídající slova propojit do jednoho proudu operátory OR.

Regulární výrazy nad pozicemi

Dotazy obsahující více než jednu pozici rozložíme na jednotlivé pozice, které potom dále spojujeme. Drtivá většina regulárních výrazů nad pozicemi je tvořena pouhou posloupností pozic. Ty můžeme vyhodnocovat pomocí operátoru AND s tím, že hodnoty každé další pozice musíme zvětšit o jedničku. Stále tedy pracujeme pouze s proudy pozic, i když výsledkem dotazu s posloupností jsou již intervaly (ale s konstantní velikostí).

Alternace posloupností pozic převedeme na spojení intervalů operátorem OR. Pro složitější regulární výrazy již nevystačíme s operátory nad proudy pozic či intervalů. Musíme použít klasickou metodu na zpracování regulárních výrazů – konečné automaty. Jednotlivé hrany konečného automatu reprezentují jednotlivé pozice v dotazu.

Struktury, zarovnání a ukazatele

Struktury, zarovnání i ukazatele se do vyhodnocování zapojují stejným způsobem. Podle typu dotazu je s ostatními pozicemi spojujeme operátorem AND omezeným na začátek nebo konec intervalu, popř. použijeme některý z operátorů IN, CONTAIN atd.

Atributy struktur se vyhodnocují stejně jako poziční atributy a výsledkem jsou potom omezeny vlastními proudy intervalů struktur.

3.8 Vytváření datových struktur

Jak bylo zdůvodněno v kapitole 2.6, korpusy můžeme považovat za statická data, a tak jsme v předcházejících kapitolách neuváděli žádné algoritmy na modifikaci popsaných struktur. Datové struktury ale musíme na počátku vytvořit. V této podkapitole si popíšeme převod vstupního formátu (vertikálního textu nebo XML – viz kap. 1.8) do binární podoby popsané v podkapitolách předcházejících.

Kódování korpusu probíhá v několika krocích.

1. Prvním krokem je průchod vertikálního textu, při kterém se vytvářejí slovníky pro všechny poziční atributy a atributy struktur, zapisují se intervaly struktur, vlastní korpus se ve všech pozičních atributech ukládá ve formě δ-kódu. Zároveň s budováním slovníků se počítají četnosti jednotlivých hesel (slov, základních tvarů, značek apod.).

2. Z četností hesel ve slovnících se vytvoří Huffmanovy kódy pro poziční atributy.

3. Postupně nebo paralelně se prochází korpus pro každý poziční atribut, vytváří se korpus v Huffmanově kódování a zároveň se vytváří reverzní indexy. Paralelně se také mohou vytvářet případné dotatečné indexy struktur nebo ukazatelů.

4. Pro dynamické atributy, které mohou být použity v dotazech, se prochází slovníky příslušných pozičních atributů a vytváří se index pro všechny hodnoty.

Kapitola 4

Vlastní implementace

Tato kapitola podrobně popisuje některé technické detaily programového řešení korpusového manažeru. Některé zde uvedené úvahy či postupy se neváží přímo na korpusový manažer, ale mají obecnou platnost.

4.1 Volba programovacích jazyků

V předešlé kapitole jsem vybrali jako jedno z rozhraní korpusového manažeru knihovnu v jazyce C. Samozřejmě nemusíme použít přímo jazyk C, můžeme zvolit libovolný programovací jazyk, který umožňuje vytvářet standardní moduly s volacími konvencemi jazyka C. Protože se manažer skládá z poměrně složitých datových struktur v různých variantách, zvolili jsme objektový přístup s pomocí jazyka C++. V něm jsou napsány všechny kritické operace.

Další části manažeru jsou napsány v jazyce Python, který byl zvolen opět díky výborné podpoře objektů a také díky celkově rychlému vývoji programů. (Nezávislé studie dokazují několikanásobně kratší dobu vývoje systémů v jazyce Python oproti jazykům C/C++, při zachování všech implementovaných funkcí.) Nezanedbatelnou výhodou Pythonu je jeho dobrá přenositelnost na všechny klíčové platformy: UNIX (bez ohledu na konkrétního výrobce), Windows, Macintosh.

Díky otevřenému protokolu, aplikace GUI nejsou žádným způsobem vázány na konkrétní implementaci serveru a mohou být tedy řešeny odlišnými prostředky. Pro programování grafických rozhraní je stále oblíbenější knihovna *Tk* začleněná do různých skriptovacích jazyků. Nejlepší integraci poskytuje přirozeně jazyk Tcl [Ous94], ze kterého byla vyčleněna. Kombinace Tcl/Tk se osvědčila již při tvorbě manažeru GCQP, a tak ji považujeme za ideální řešení pro tvorbu GUI.

4.2 Rozdělení na moduly

Nejnižší vrstvu, jádro celého systému, které zajišťuje fyzický přístup na disk, tvoří knihovna *fin*lib. Je napsána v programovacím jazyce C++ a její základní rysy jsou popsány v následující kapitole. Celá knihovna není vázána na korpusový manažer, lze ji dobře použít i v jiných aplikacích.

Knihovna *fin*lib je zastřešena několika třídami a funkcemi, které již reprezentují prvky korpusového manažeru (poziční atributy, struktury atd.). Tato vrstva je integrována do programovacích jazyků Python, Perl a Tcl. Vlastní manažer, obsahující zpracování dotazů, výpočet statistik, základní prezentace výsledů, je realizován v jazyce Python nad touto vrstvou. Touto formou lze realizovat další rozhraní manažeru, například WWW stránky pro jednoduché dotazování.

GUI bude vytvořeno jako zcela samostatný program (stejným způsobem, jako GCQP), který se k manažeru připojuje internetovým protokolem.

4.3 Nejnižší vrstva: knihovna *fin*lib

4.3.1 Struktura knihovny

Jádro knihovny je napsáno v jazyce C++ za použití šablon. Proto nelze některé její funkce přímo používat v jiných programovacích jazycích. Nejčastěji používané instance šablon jsou vytvořeny a uloženy ve sdílené knihovně `libfin.so`, kterou lze běžně požívat i mimo C++.

Pro vyšší skriptovací jazyky je vytvořeno rozhraní pomocí systému SWIG. Z jazyků Python, Perl a Tcl je tedy možné po zavedení modulu `finlib` přímo používat většinu tříd a funkcí.

4.3.2 Třídy pro čtení/zápis bitových proudů

Pro čtení různě kódovaných čísel z proudu bitů slouží třída `read_bits`, pro zápis třída `write_bits`. Pomocí těchto tříd je možné do jednoho bitového proudu ukládat čísla v různých kódech, aplikace pak musí zajistit, že ke čtení používá posloupnost stejných kódů, které byly použity při zápise. Obě třídy jsou řešeny jako šablony, u kterých je potřeba specifikovat typ bitového proudu ze/do kterého se čte/zapisuje a typ základního prvku (implicitně `unsigned char`). Deklarace tedy mají tvar:

```
template <class Iterator, class AtomType = unsigned char>
class write_bits;

template <class Iterator, class AtomType = unsigned char>
class read_bits;
```


a jsou přístupné v hlavičkovém souboru `bitio.hh`.

Volba iterátoru

Typem bitového proudu (iterátoru) určujeme, kde jsou vlastní data uložena. Může to být operační paměť (pak jako iterátor lze použít přímo ukazatel – například „`unsigned char *`“) soubor na disku apod. Třída musí mít implementovány operátory dereference (`*`) pro získání hodnoty a operátor inkrementace (prefixový `++`) pro posun na další hodnotu. Tomu vyhovují například všechny iterátory odvozené od standardní třídy `forward_iterator<T,Distance>`.

V hlavičkovém souboru `bititer.hh` je definováno několik iterátorů pro použití s třídami `read_bits` a `write_bits`. Pro zápis do paměti je možné používat přímo ukazatel. Čtení ale probíhá destruktivně (zejména kvůli efektivitě), nelze tedy číst opakovaně stejná data přímo z paměti (pomocí ukazatele jako iterátoru). Můžeme ale použít třídu `SafeBytes`, která ukazatel do paměti „obalí“ a vytvoří iterátor, který při čtení paměť nemění. Třída `SafeBytes` používá jako základní typ byte (`unsigned char`), pokud potřebujeme pracovat s jiným základním typem, můžeme použít přímo šablonu `SafeInBits`. Použití:

```
unsigned char *pc;
...
SafeBytes i(pc);
read_bits rb(i);

unsigned int *pi;
SafeInBits<unsigned int> sib (pi);
read_bits j(sib);
```

Pro přístup k bitovým proudům v souborech slouží třídy `InFileBits` pro čtení a `OutFileBits` pro zápis. Třídy vytvoří příslušné iterátory ze souboru. Soubor může být zadán buď svým jménem, v tom případě se soubor v konstruktoru otevře, čtení se provádí od začátku soboru a v destruktoru se zavře. Pozici, kde se právě čte, nelze změnit, ani zjistit. Druhou možností je vytvořit iterátor z již otevřeného souboru (konstruktor s parametrem typu `FILE *`). Pak se provádí čtení od právě aktuální pozice v souboru. Použití:

```
OutFileBits ob ("vystup.data");
write_bits wb (ob);

f = fopen ("vstup.data");
fseek (f, ...);
```

```
InFileBits ib (f);
read_bits rb (ib);
...
fseek (f, ...);
rb.new_block()
....
```

Třída `read_bits`

Třída slouží ke čtení čísel z bitového proudu. Je potřeba dávat pozor, jaká čísla jednotlivé funkce vracejí. Vždy to jsou celá čísla nezáporná, u některých funkcí to jsou celá kladná (tedy mimo nulu). Třída obsahuje tyto metody:

read_bits(Iterator memory, int skipbits=0) konstruktor. `memory` je iterátor, ze kterého se bude číst; `skipbits` je počet počátečních bitů, které se mají před prvním čtením ze vstupního proudu přeskočit.

int get_bit () přečte jeden bit – vrací 1 nebo 0.

int unary () přečte kladné číslo v unárním kódování

unsigned int binary_fix (int len) přečte nezáporné číslo v binárním kódování s délkou bitů `len`

unsigned int binary (unsigned int b) přečte kladné číslo v binárním kódování s limitem `b`, číslo tedy zabírá $\lfloor \log_2 b \rfloor$ nebo $\lceil \log_2 b \rceil$ bitů – vrací číslo z intervalu $< 1, b >$

unsigned int gamma () přečte kladné číslo v γ-kódování

unsigned int delta () přečte kladné číslo v δ-kódování

unsigned int golomb (unsigned int b) přečte kladné číslo v Golombově kódování s parametrem b

void new_block () přeskočí další nepřečtené bity až na další celý základní prvek (parametr šablony `AtomType`), tedy většinou na další byte

Třída `write_bits`

Třída slouží k zápisu čísel do bitového proudu. Většina metod je přímou analogií metod třídy `read_bits`, pouze mají vždy o jeden parametr víc a nevracejí žádnou hodnotu. Pro tyto metody je uveden pouze prototyp:

write_bits (Iterator memory)

int bitslen () vrátí počet zatím zapsaných bitů v aktuálním základním prvku. Například po provedení metody `new_block` vrátí `bitslen` vždy nulu.

void load (Iterator src, int srcbits) do bitového proudu zapíše (překopíruje) `srcbits` bitů ze `src`

void set_bit (int x)

void unary (int val)

void binary_fix (unsigned int x, int len)

void binary (unsigned int x, unsigned int b)

void gamma (unsigned int x)

void delta (unsigned int x)

void golomb (unsigned int x, unsigned int b)

void new_block ()

Třídy `read_huffman` a `write_huffman`

Tyto třídy slouží ke čtení resp. zapisování Huffmanova kódu. Obě využívají instance třídy `huffman_data`, kde jsou uloženy vlastní Huffmanovy kódy pro všechna čísla, která lze ukládat. Jde tedy o pomocnou třídu, u které se neočekává přímý přístup k jejím datovým prvkům, pouze má sloužit ke zprostředkování informací instancím tříd `read_huffman` a `write_huffman`. Deklarace všech tříd pro práci s Huffmanovy kódy jsou ve hlavičkovém souboru `huffman.hh`.

Třída `huffman_data` obsahuje tyto metody:

huffman_data (const char *filename, int read_write) konstruktor, který rychle načte popis Huffmanova kódu ze souboru `filename`. Příznak `read_write` musí obsahovat jednu z hodnot `huffman_data::rw_READ`, `huffman_data::rw_WRITE` nebo jejich binární kombinaci podle toho, jestli chceme číst a/nebo zapisovat. Pro čtení a zápis se budují rozdílné datové struktury.

huffman_data (int num, int *freqs) konstruktor, který vytvoří popis Huffmanova kódu ze zadaných četností jednotlivých čísel. Četnosti jsou předány v poli `freqs`, jejich počet je `num`. Instance vytvořené tímto konstruktorem mohou být použity ke čtení i zápisu.

bool store (char *filename) Metoda uloží datové struktury do souboru `filname`. Při úspěšném uložení vrací `true`, při neúspěchu `false`.

Stejně jako třídy `read_bits` a `write_bits` jsou třídy `read_huffman` a `write_huffman` deklarovány jako šablony:

```
template <class Iterator, class AtomType = unsigned char>
class read_huffman
    read_huffman (huffman_data *d,
                  Iterator memory, int skipbits=0);
    int next ();
;

template <class Iterator, class AtomType = unsigned char>
class write_huffman
    write_huffman (huffman_data *d, Iterator memory);
    void next (int x);
    int bitslen ();
;
```

Na místě iterátoru lze použít stejné třídy jako u `read_bits` a `write_bits`. Parametry konstruktorů si také odpovídají, pouze je v obou případech nutné zadávat ukazatel na instance pomocné třídy `huffman_data`. Pro vlastní čtení/zápis čísel slouží metody `next`.

4.3.3 Přístup ke korpusům

Korpusy (resp. jejich poziční atributy) jsou posloupností slov. Z důvodů efektivity je ale vše uloženo poněkud jinak. Všechna slova z daného pozičního atributu (například slovo, nebo základní tvar) jsou uložena ve slovníku. Slovník zajišťuje „očíslování“ slov, tedy převod z řetězce na číslo a zpět. Ve vlastním korpusu jsou pak uvedena pouze čísla. Pro rychlejší vyhledávání i pro urychlení některých výpočtů se využívají tzv. „reverzní indexy“, ve kterých je pro každé slovo (zadané jeho číslem) uveden seznam všech jeho výskytů (pozic). Tento seznam je setříděný a umožňuje rychlé (logaritmické) vyhledávání. Dále je pro každé slovo ve slovníku přístupný počet výskytů daného slova v korpusu. Ke korpusům je možné vypočítat a uložit bigramy, tedy počty dvojic slov, která se vyskytnou v korpusu přímo za sebou. Použité datové struktury nejsou samozřejmě omezeny pouze na bigramy – obecně je můžeme použít na jedné celočíselné hodnoty pro (potenciálně) každou dvojici čísel ze slovníku.

Všechny třídy mají definován konstruktor s parametrem `filename`, který udává cestu k datovým souborům. Protože jednotlivé instance nemusí být tvořeny pouze jedním souborem, `filename` není přímo celé jméno (cesta) souboru, ale pouze společný prefix.

Slovník – třída `lexicon`

Třída `lexicon` poskytuje obousměrný převod mezi znakovým řetězcem (slovem) a číslem (jednoznačným číselným identifikátorem slova). Čísla jsou v rozsahu 0 až $lexsize - 1$, kde $lexsize$ je velikost slovníku (počet slov). Třída `lexicon` je abstraktní třídou, která definuje rozhraní a některé obecné metody, vlastní implementace je ve třídách odvozených.

`lexicon` obsahuje tyto metody:

int size () – abstraktní metoda, která má vracet velikost slovníku

const char* id2str (int id) – abstraktní metoda, která převádí číslo na řetězec

int str2id(const char *str) – abstraktní metoda, která převádí řetězec na číslo

ID_list *pref2ids(const char *str) – vrací seznam čísel, která odpovídají řetězcům začínajícím znaky v `str`

Třídou, která uvedené metody implementuje, je `map_lexicon`. Data tato třída ukládá v binárních souborech, které se „mapují“ do paměti. Konstruktor má pouze jeden parametr – prefix datových souborů.

Korpus – třída `corpus`

hlavičkový soubor `corpus.hh`

corpus (const string &filename)

at (int pos) – iterátor pro sekvenční čtení čísel od pozice `pos`

int pos2id(int pos) – číslo na pozici `pos`

Reverzní index – třída `revidx`

hlavičkový soubor `revidx.hh`

FastStream *id2poss (int id) vrátí proud pozic pro dané `id`

int count (int id) vrátí počet pozic v proudu pro dané `id`

Bigramy

hlavičkový soubor `bigram.hh`

int count (int id) – počet bigramů s prvním prvkem `id`

int value (int id1, int id2) – hodnota bigramu `<id1,id2>`, pokud hodnota není definována, vrací 0

bgrpair* ids_to (int id) – počáteční iterátor pro sekvenční procházení bigramů s prvním prvkem `id`

bgrpair* ids_from (int id) – zarážka iterátoru pro sekvenční procházení bigramů s prvním prvkem `id`

4.3.4 Rychlé číselné seznamy – `FastStream`

Třída `FastStream` zajišťuje efektivní procházení (vzestupně) uspořádaných seznamů celých čísel. Je to abstraktní třída definovaná v hlavičkovém souboru `fstream.hh`.

Instanci potomka této třídy (seznam čísel) lze zpracovávat buď sekvenčně (pomocí metody `next()`), nebo „náhodně" (pomocí metody `find (Position pos)`), nebo libovolnou kombinací předešlého. Celý seznam lze projít pouze jednou, v seznamu se není možno „vracet" k dříve přečteným hodnotám. Seznam je ukončen zarážkou, po průchodu celým seznamem vracejí všechny metody pro získání prvku seznamu stejnou hodnotu (zarážku), která do vlastního seznamu nepatří. Jednotlivé metody:

Position peek () – vrátí hodnotu, kterou by vrátilo příští volání metody `next()`

Position next () – přečte další (první dosud nepřečtený) prvek ze seznamu a vrátí jeho hodnotu

Position find (Position pos) – pokusí se v seznamu nalézt prvek s hodnotou `pos`. Pokud ho nalezne vrátí `pos`, pokud ne, vrátí hodnotu prvku nejbližšího následujícího. Celkově se chová jako `peek()`, tedy vrácená hodnota je stejná, jakou by vrátilo příští volání metody `next()`

NumOfPos rest_min () – vrací dolní odhad počtu dosud nepřečtených prvků v seznamu

NumOfPos rest_max () – vrací horní odhad počtu dosud nepřečtených prvků v seznamu

Position final () – vrací hodnotu zarážky

4.3.5 Operace nad instancemi `FastStream`

Operace jsou deklarovány v hlavičkovém souboru `fsop.hh`. Všechny dále popsané operace berou jako parametry instance potomků třídy `FastStream` a „vracejí " opět `FastStream`. Implementovány jsou jako třídy, jejichž konstruktory mají za parametry příslušné proudy. V programu tedy „provedení " operace odpovídá volání konstruktoru příslušné třídy.

V následujícím výčtu definovaných operací je vždy uveden prototyp konstruktoru příslušné třídy:

QNotNode(FastStream *source)
negace – výsledný seznam obsahuje všechny hodnoty od 0 po zarážku, které nejsou v původním seznamu

QAndNode (FastStream *source1, FastStream *source2)
logické AND – ve výsledku jsou všechny společné hodnoty operandů

QOrNode (FastStream *source1, FastStream *source2)
logické OR – ve výsledku je sjednocení prvků z operandů

QMoveNode (FastStream *source, int delta)
posunutí – ve výsledku jsou hodnoty původního seznamu zvětšené o `delta` (může být i záporné), pokud dávají smysl (Například hodnota 1 zvětšená o -3 nedává smysl – je záporná.)

4.3.6 Programy na vytváření popsaných struktur

Popsané třídy slouží vždy pouze ke čtení již vytvořených struktur. Pro vlastní vytváření je nutno použít samostatné programy popsané v této části.

4.3.7 Pomocné šablony a třídy

V této části jsou popsány některé pomocné struktury a algoritmy, které jsou používány v jiných částech knihovny *fin*lib, ale je možné je použít i samostatně.

Binární soubory jako pole

V hlavičkovém souboru `binfile.hh` jsou definovány třídy, které umožňují binární soubory s homogenní strukturou (pole stejných prvků) používat přímo jako pole. Všechny třídy jsou deklarovány jako šablony s parametrem `AtomType` (typ jednotlivých prvků) a definují následující metody:

třída (const string &filename) – konstruktor, `filename` je jméno souboru; třída `string` má definovánu automatickou konverzi z `char *`, můžeme tedy přímo používat řetězcové konstanty.

int size () – vrátí velikost pole

AtomType operator[] (int pos) – vrátí hodnotu prvku pole s indexem `pos`

iterator at (int pos) – vrátí iterátor pro sekvenční čtení jednotlivých prvků od indexu `pos`

4.4 GUI klient

Při implementaci GCQP bylo použito několik netradičních prvků, které významně přispěly k funkčnosti celé aplikace. Předpokládáme, že stejné postupy budeme používat i GUI nového manažeru. Nejdůležitější prvky implementace jsou:

- Texty uživatelského rozhraní i různé konfigurační parametry (například jméno implicitního korpusu, velikost kontextu apod.) jsou uchovávány odděleně od vlastního kódu aplikace. Přizpůsobení novému jazyku či novému systémovému prostředí tak nemusí provádět přímo programátor.
- Uživatelé mají na výběr několik vytvořených prostředí, které si mohou vybrat při spuštění nebo dokonce za běhu aplikace.
- Pokročilí uživatelé mohou manažer v omezené míře rozšiřovat o vlastní funkce.

Kapitola 5

Závěr

V této kapitole je uveden stručný souhrn hlavních výsledků popsaných v práci a také jsou uvedeny některé nezpracované problémy a výhledy do budoucna.

5.1 Souhrn hlavních výsledků

Hlavním výsledkem práce je celková analýza korpusového manažeru. Práce postupně popisuje vlastnosti korpusů, požadavky na korpusový manažer, návrh algoritmů a datových struktur pro implementaci všech požadavků.

Většina navrhovaných prvků manažeru byla implementována: pro přístup klient/server (nutný pro rozsáhlé korpusy) byl v rámci korpusového manažeru GCQP realizován protokol nad TCP/IP; grafické uživatelské rozhraní spolu s několika původními metodami tvorby dotazů je též součástí GCQP. Efektivní uložení klíčových datových struktur (text korpusu a reverzní index) a rychlé vyhodnocování dotazů je implementováno v knihovně *fin*lib. Zpracování dotazů v novém dotazovacím jazyce COFE zjednodušující zadávání dotazů realizuje nadstavba nad knihovnu *fin*lib. Výpočty nově navržených statistik byly provedeny pomocí samostatných programů.

Na základě analýzy byly pro vytváření a ukládání datových struktur korpusu navrženy modifikace známých postupů, které umožňují korpusovému manažeru efektivnější zpracování dat.

5.2 Budoucí práce

Popsaná analýza si jistě zaslouží *plnou* implementaci korpusového manažeru. Proto hodlám na základech knihovny *fin*lib vybudovat nový korpusový manažer, který by měl všechny uvedené funkce, včetně pohodlného grafického uživatelského rozhraní.

Příloha A

„Pražské“ gramatické značky

Popis morfologických značek – poziční systém

Struktura značky

Každá značka je řetězcem 15 znaků. Značka je konstruována tak, aby každá pozice odpovídala jedné morfologické kategorii podle víceméně tradičního lingvistického pojetí. Každé hodnotě v dané kategorii odpovídá jeden znak, převážně písmeno velké abecedy (např. ’P’ pro plurál, neboli množné číslo), výjimečně i jiný znak (např. ’f’ pro infinitiv, nebo ’^’ pro souřadící spojky). Hodnota, která nedává smysl (např. pád u sloves), je reprezentována znakem ’-’ (pomlčka).

Tradiční lingvistické detailní rozdělení není ovšem vždy respektováno (z nejrůznějších důvodů). Například tvary minulého příčestí sloves (aktivního i pasívního) nejsou rozlišeny z hlediska rodu (ve spojení s gramatickým číslem) pro tvary končící na -l, -ly ani -la. Podobně zkratky a nesklonná substantiva nedávají na výstupu morfologické analýzy 14 značek, jak by bylo možno očekávat, ale jsou anotovány (v technickém smyslu) jednoznačně značkou, kde je pro číslo a pád uveden znak ’X’, používaný převážně pro tento typ nejednoznačnosti (či spíše neurčitosti).

Popis jednotlivých pozic značky

Pozice jsou číslovány od 1 do 15. V nadpisech jsou na konci v závorce uvedeny zkratky pro jednotlivé pozice, používané v jiných programech (jen pro informaci).

Pozice 1 – Slovní druh (POS)

Označuje hlavní slovní druh, víceméně podle obvyklého schématu známého z českých gramatik včetně školních. Přiřazení i těchto hlavních slovních druhů je však řízeno především potřebami konzistentnosti další analýzy přirozeného jazyka. Proto je možné, že v některých případech (zejména tehdy, kdy se gramatiky a slovníky v určení slovního druhu neshodují nebo uvádějí jiné rozdělení na významy slova nebo tam, kde ve slovníku najdeme slovnědruhové perly typu ″zájmenné příslovce″) nemusí být zařazení zcela ″tradiční″.

`A` – adjektivum (přídavné jméno)
`C` – numerál (číslovka, nebo číselný výraz s číslicemi)
`D` – adverbium (příslovce)
`I` – interjekce (citoslovce)
`J` – konjunkce (spojka)
`N` – substantivum (podstatné jméno)
`P` – pronomen (zájmeno)
`R` – prepozice (předložka)
`T` – partikule (částice)
`V` – verbum (sloveso)
`X` – neznámý, neurčený, neurčitelný slovní druh
`Z` – interpunkce, hranice věty

Pozice 2 – Detailní určení slovního druhu (SUBPOS)

Detailní slovní druh slouží především k určení dalších relevantních morfologických kategorií, které jsou uvedeny na dalších pozicích (ne vždy však jednoznačně). Ze znaku použitého pro detailní určení slovního druhu je možné jednoznačně vyvodit hlavní slovní druh (pozice 1).

`!` – zkratka jako adverbium
`#` – hranice věty (jen u ″virtuálního″ slova ″###″)
`*` – slovo ″krát″ (slovní druh: spojka)
`,` – spojka podřadící (vč. ″aby″ a ″kdyby″ ve všech tvarech)
`.` – zkratka jako adjektivum
`0` – předložka s připojeným ″-ň″ (něj), proň, naň, atd. (slovní druh: zájmeno)
`1` – vztažné přivlastňovací zájmeno ″jehož″, ″jejíž″, ...
`2` – pomlčka (vždy je samostatně)
`3` – zkratka jako číslovka
`4` – vztažné zájmeno s adjektivním skloňováním (obou typů: jaký, který, čí, ...)
`5` – zájmeno ″on″ ve tvarech po předložce (tj. ″n-″: něj, něho, ...)
`6` – reflexívní zájmeno ″se″ v dlouhých tvarech (sebe, sobě, sebou)

`7` – reflexívní zájmeno ″se″, ″si″ pouze v těchto tvarech, a dále ″ses″, ″sis″
`8` – přivlastňovací zájmeno ″svůj″
`9` – vztažné zájmeno ″jenž″, ″již″, ... po předložce (″n-″: ″něhož″, ″níž″, ...)
`:` – interpunkce všeobecně (ne však ″virtuální″ slovo ### jako hranice věty)
`;` – zkratka jako substantivum
`=` – číslo psané číslicemi (slovní druh: ′C′)
`?` – číslovka ″kolik″
`@` – slovní tvar, který nebyl morfologickou analýzou rozpoznán (sl. druh: ′X′)
`A` – adjektivum obyčejné
`B` – sloveso, tvar přítomného nebo budoucího času
`C` – adjektivum, jmenný tvar
`D` – zájmeno ukazovací (ten, onen, ...)
`E` – vztažné zájmeno ″což″
`F` – součást předložky, která nikdy nestojí samostatně (nehledě, vzhledem, ...)
`G` – přídavné jméno odvozené od slovesného tvaru přítomného přechodníku
`H` – krátké tvary osobních zájmen (mě, mi, ti, mu, ...)
`I` – citoslovce (slovní druh: ′I′)
`J` – vztažné zájmeno ″jenž″ (″již″, ...), bez předložky
`K` – zájmeno tázací nebo vztažné ″kdo″, vč. tvarů s ″-ž″ a ″-s″
`L` – zájmeno neurčité ″všechnen″, ″sám″
`M` – přídavné jméno odvozené od slovesného tvaru minulého přechodníku
`N` – substantivum, obyčejné
`O` – samostatně stojící zájmena ″svůj″, ″nesvůj″, ″tentam″
`P` – osobní zájmena (vč. tvaru ″tys″)
`Q` – zájmneo tázací/vztažné ″co″, ″copak″, ″cožpak″
`R` – předložka, obyčejná
`S` – zájmeno přivlastňovací ″můj″, ″tvůj″, ″jeho″ (vč. plurálu)
`T` – částice (slovní druh ′T′)
`U` – adjektivum přivlastňovací (na ″-ův″ i ″-in″)
`V` – předložka vokalizovaná (″ve″, ″pode″, ″ku″, ...)
`W` – zájmena záporná (″nic″, ″nikdo″, ″nijaký″, ″žádný″, ...)
`X` – slovní tvar, který byl rozpoznán, ale značka (ve slovníku) chybí
`Y` – zájmeno ″co″ spojené s předložkou (″oč″, ″nač″, ″zač″)
`Z` – zájmeno neurčité (″nějaký″, ″některý″, ″číkoli″, ″cosi″, ...)
`^` – spojka souřadící
`a` – číslovka neurčitá (″mnoho″, ″málo″, ″tolik″, ″několik″, ″kdovíkolik″, ...)
`b` – příslovce (bez určení stupně a negace; ″pozadu″, ″naplocho″, ...)

`c` – kondicionál slovesa být (”by”, ”bych”, ”bys”, ”bychom”, ”byste”)
`d` – číslovka druhová, adjektivní skloňování (”jedny”, ”dvojí”, ”desaterý”, ...)
`e` – slovesný tvar přechodníku přítomného (”-e”, ”-íc”, ”-íce”)
`f` – slovesný tvar: infinitiv
`g` – příslovce (s určením stupně a negace; ”velký”, ”zajímavý”, ...)
`h` – číslovky druhové ”jedny” a ”nejedny”
`i` – slovesný tvar rozkazovacího způsobu
`j` – číslovka druhová ¿= 4, substantivní postavení (”čtvero”, ”desatero”, ...)
`k` – číslovka druhová ¿= 4, adjektivní postavení, krátký tvar (”čtvery”, ...)
`l` – číslovky základní 1-4, ”půl”, ...; sto a tisíc v nesubstantivním skloňování
`m` – slovesný tvar přechodníku minulého, příp. (zastarale) přech. přít. dokonavý
`n` – číslovky základní ¿= 5
`o` – číslovky násobné neurčité (”-krát”: ”mnohokrát”, ”tolikrát”, ...)
`p` – slovesné tvary minulého aktivního příčestí (vč. přidaného ”-s”)
`q` – archaické slovesné tvary minulého aktivního příčestí (zakončení ”-ť”)
`r` – číslovky řadové
`s` – slovesné tvary minulého pasívního příčestí (vč. přidaného ”-s”)
`t` – archaické slovesné tvary přitomného a budoucího času (zakončení ”-ť”)
`u` – číslovka tázací násobná ”kolikrát”
`v` – číslovky násobné (”-krát”: ”pětkrát”, ...)
`w` – číslovky neurčité s adjektivním skloňováním (”nejeden”, ”tolikátý”, ...)
`x` – zkratka, slovní druh neurčen/neznámý
`y` – zlomky zakončené na ”-ina” (slovní druh: ’C’)
`z` – číslovka tázací řadová ”kolikátý”
`}` – číslovka psaná římskými číslicemi
`~` – zkratka jako sloveso

Pozice 3 – Jmenný rod (GENDER)

`-` – neurčuje se
`F` – femininum (ženský rod)
`H` – femininum nebo neutrum (tedy nikoli maskulinum)
`I` – maskulinum inanimatum (rod mužský neživotný)
`M` – maskulinum animatum (rod mužský životný)
`N` – neutrum (střední rod)
`Q` – femininum singuláru nebo neutrum plurálu (pouze u příčestí a jmenných adj.)
`T` – masculinum inanimatum nebo femininum (jen plurál u příčestí a jm. adj.)
`X` – libovolný rod (F/M/I/N)
`Y` – masculinum (animatum nebo inanimatum)
`Z` – ’nikoli femininum’ (tj. M/I/N; především u příslovcí)

Pozice 4 – Číslo (NUMBER)

`-` – neurčuje se
`D` – duál (pouze 7. pád feminin)
`P` – plurál (množné číslo)
`S` – singulár (jednotné číslo)
`W` – pouze v kombinaci s jmenným rodem ’Q’ (singulár pro fem., pl. pro neutra)
`X` – libovolné číslo (P/S/D)

Pozice 5 – Pád (CASE)

`-` – neurčuje se
`1` – nominativ (1. pád)
`2` – genitiv (2. pád)
`3` – dativ (3. pád)
`4` – akuzativ (4. pád)
`5` – vokativ (5. pád)
`6` – lokativ (6. pád)
`7` – instrumentál (7. pád)
`X` – libovolný pád (1/2/3/4/5/6/7)

Pozice 6 – Přivlastňovací rod (POSSGENDER)

Rody mužský neživotný a střední se nikdy nevyskytují samostatně. ’M’ se může vyskytnout jen u přivlastňovacích adjektiv (me u příslovcí).
`-` – neurčuje se
`F` – femininum (ženský rod)
`M` – maskulinum animatum (rod mužský životný)
`X` – libovolný rod (F/M/I/N)
`Z` – ’nikoli femininum’ (tj. M/I/N; u přivlastňovacích příslovcí)

Pozice 7 – Přivlastňovací číslo (POSSNUMBER)

`-` – neurčuje se
`P` – plurál (množné číslo)
`S` – singulár (jednotné číslo)

Pozice 8 – Osoba (PERSON)

`-` – neurčuje se
`1` – 1. osoba
`2` – 2. osoba
`3` – 3. osoba
`X` – libovolná osoba (1/2/3)

Pozice 9 – Čas (TENSE)

`-` – neurčuje se
`F` – futurum (budoucí čas)
`H` – minulost nebo přítomnost (P/R)
`P` – prézens (přítomný čas)
`R` – minulý čas
`X` – libovolný čas (F/R/P)

Pozice 10 – Stupeň (GRADE)

`-` – neurčuje se
`1` – 1. stupeň
`2` – 2. stupeň
`3` – 3. stupeň

Pozice 11 – Negace (NEGATION)

`-` – neurčuje se
`A` – afirmativ (bez negativní předpony "ne-")
`N` – negace (tvar s negativní předponou "ne-")

Pozice 12 – Aktívum/pasívum (VOICE)

`-` – neurčuje se
`A` – aktívum
`P` – pasívum

Pozice 13 – Nepoužito (RESERVE1)

`-` – neurčuje se

Pozice 14 – Nepoužito (RESERVE2)

`-` – neurčuje se

Pozice 15 – Varianta, stylový příznak apod. (VAR)

`-` – neurčuje se ("základní" tvar pro kategorie v pozicích 1–14)
`1` – varianta, víceméně rovnocenná ("méně častá")
`2` – řídká, archaická nebo knižní varianta
`3` – velmi archaický tvar, též hovorový
`4` – velmi archaický nebo knižní tvar, pouze spisovný (ve své době)
`5` – hovorový tvar, ale v zásadě tolerováno ve veřejných projevech
`6` – hovorový tvar (koncovka standardní obecné češtiny)
`7` – hovorový tvar (koncovka standardní obecné češtiny), varianta k '6'
`8` – zkratky
`9` – speciální použití (tvary zájmen po předložkách apod.)

Příloha B

„Brněnské“ gramatické značky

Přehled symbolů programu LEMMA:

Slovní druh:

Substantiva	k1	subs
Adjektiva	k2	adje
Zájmena	k3	pron
Číslovky	k4	numb
Slovesa	k5	verb
Příslovce	k6	advb
Předložky	k7	prep
Spojky	k8	conj
Částice	k9	part
Citoslovce	k0	intr
Possessiva	k2	poss
Zkratky	kX	abbr

Rod:

Mužský životný	gM	Man
Mužský neživotný	gI	Min
Ženský	gF	Fem
Střední	gN	Neu
Libovolný	gX	Any
Muž.než.+střední	gY	I+N
Mužský +střední	gU	AIN
Životný (kdo)	gP	Prs
Neživotný (co)	gT	Npr

Číslo:

Jednotné	nS	sg
Množné	nP	pl

Pád:

Pád č. ?	c?	?

Osoba:

1.	p1	1.p
2.	p2	2.p
3.	p3	3.p

Čas:

Minulý	tM	pret
Přítomný	tP	pres
Budoucí	tF	fut

Modus:

Infinitiv	mF	infi
Indikativ	mI	indi
Imperativ	mR	impe
Participium	mP	part
Transgresiv	mT	trsg
Condicional	mC	cond
Konjunktiv	mK	konj

Vid:

Perfectum	aP	perf
Imperfectum	aI	impf

Stupňování:

Nominativ	d1	nomi
Comparativ	d2	comp
Superlativ	d3	supr

Negace:

Afirmace	eA	
Negace	eN	~

Přirozený rod:

Mužský	hM	Mas
Ženský	hF	Fem

Druhy zájmen:

Osobní	xP	P
Přivlastňovací	xO	O
Ukazovací	xD	D
Tázací	xQ	Q
Vztažná	xR	R
Neurčitá	xU	U
Záporná	xN	N
Reflexivní	xX	X

Druhy číslovek:

Základní	xC	C
Řadové	xO	O
Druhové	xR	R
Názvy jmen	xN	N
Názvy zlomků	xD	D
Názvy n-tic	xT	T

Druhy příslovcí:

Místa	xL	loc
Času	xT	tem
Způsobu	xM	man
Modální	xD	mod

Druhy spojek:

Souřadné	xC	cor
Podřadné	xS	sub

Příloha C

Komunikační protokol mezi `cqsd` a GCQP

Program `cqsd` čte příkazy ze standardního vstupu a vypisuje výsledky těchto příkazů na standardní výstup. Síťová komunikace je realizována konfigurací sítě na počítači (soubory `/etc/services` a `/etc/inetd.conf`), ověřování přístupu je realizováno zvláštním skriptem, který po úspěšné autentizaci volá vlastní program `cqsd`.

Syntaxe vstupu i výstupu je přizpůsobena jazyku Tcl [Ous94]. Jedná se zejména o zvláštní význam znaků: “`\\`”, “`"`” a “`{`”, které se interpretují stejným způsobem jako v Tcl.

Vstupní příkaz je tvořen posloupností slov (slov ve smyslu Tcl, například řetězec ve složených závorkách může obsahovat několik řádků a je považován za jedno slovo), ukončenou středníkem (’;’), nebo koncem řádku. Výstup je tvořen posloupností řádků, které mohou být dále strukturovány (seznamy slov, seznamy seznamů atd.), odpověď na každý příkaz je ukončena znakem “`\v`” (vertikální tabulátor, 0xB, `^K`) a koncem řádku.

V dalších podkapitolách jsou pospány jednotlivé příkazy.

C.1 Systémové příkazy

echo word . . . opíše argumenty na výstup – využívá se pro ladící účely

help seznam příkazů, které jsou k dispozici

help command nápověda k příkazu `<command>`

exit ukončení běhu serveru po dokončení všech výstupních operací

exec arg . . . je-li nastavena proměnná prostředí CQSD_EXEC, zavolá externí příkaz definovaný obsahem proměnné s parametry <arg ...>. Na výstup předá standardní výstup externího příkazu.

stop number zruší výstup příkazu určeného číslem <number> od konce. 0 (default) znamená poslední zadaný příkaz.

encoding zobrazí aktuální výstupní kódování

encoding enc nastavuje výstupní kódování <enc> (zatím jsou podporována ascii il1 il2 kam 1250)

C.2 Korpusové příkazy

lscorp seznam všech korpusů, které jsou k dispozici

lsqpos corpus seznam všech pozičních atributů korpusu <corpus>, které lze použít v dotazech

lsdpos corpus seznam všech pozičních atributů korpusu <corpus>, které lze zobrazovat

lsqstr corpus seznam všech strukturálních atributů korpusu <corpus>, které lze použít v dotazech

lsdstr corpus seznam všech strukturálních atributů korpusu <corpus>, které lze zobrazovat

C.3 Konkordanční příkazy

set conclist corpus query cut wnum wunit uplatní dotaz <query> na korpus <corpus>, výsledkem je seznam konkordancí pojmenovaný <conclist>. <query> je v formátu cqp, <cut> je maximální počet konkordancí, <wnum> odpovídá klausuli 'within #' a <wunit> je příslušný strukturální atribut.

coloc conclist num lpos rpos query offset wnum wunit nastaví kolokaci číslo <num> v seznamu konkordancí <conclist> na výsledek dotazu <query>. <lpos> a <rpos> jsou pozice určující intervaly, ve kterých se má hledat. <query>, <wnum> a <wunit> jsou stejné jako u set. <offset> určuje kolikátý výskyt v určeném intervalu se má použít, záporná čísla se berou od pravého konce.

Pozice <pos> má formát: num[s-attr|#[p-attr]][:[<|>]coloc], kde jednotlivé prvky mají následující význam:

num – počet jednotek doprava (záporná čísla pro levý směr)

s-attr – jednotka je určena strukturálním atributem <s-attr> nebo je-li uvedeno #, jedná se o minimální počet znaků pozičního atributu <p-attr> (u příkazu get je implicitní právě získávaný poziční atribut) nebo se jedná o počet pozic, pokud není uvedeno nic

<> od levého/pravého konce konkordance/kolokace implicitně je < pro <lpos> a > pro <rpos>

coloc – vzhledem ke konkordanci (0 – default) nebo kolokaci číslo (1,. . .)

get conclist beg end colocs pos-attrs stru-attrs vrátí hodnoty pozičních a strukturálních atributů podle šablon <pos-attrs> a <stru-attrs> konkordancí <beg> až <end> ze seznamu <conclist> pro kolokace ze seznamu <colocs>. <beg> a <end> jsou čísla udávající interval konkordancí v seznamu. Pokud je <end> < <beg> – řádky se vypisují v sestupném pořadí.

<pos-attrs> má formát: {{attr lpos rpos} ...} – seznam slov, která ve své podstruktuře určují jméno pozičního atributu <attr> a začátek a konec intervalu pozic, pro které se má hodnota atributu vrátit (kontext).

<stru-attrs> má formát: {[+|-]attr ...} – seznam strukturálních atributů, +/- znamená, že se mají/nemají vypisovat hodnoty (default je -).

Pro každou konkordanci je na výstup vypsán řádek ve tvaru <colocs> <pos-attrs> <stru-attrs>, jednotlivé formáty jsou: <colocs>: {c1 l1 r1 c2 l2 r2 . . . }, kde c# je číslo konkordance (0) nebo kolokace (1,. . .), l# je začátek a r# je konec kolokace v pozicích pro každý prvek z <colocs> v příkazu get <pos-attrs>: {attr pos word . . . } – jméno atributu, počáteční pozice a seznam hodnot atributů <stru-attrs>: {attr lpos rpos value [lpos rpos value . . .]} - jméno atributu, počáteční a koncová pozice a hodnota strukturálního atributu. Jednotlivé položky odpovídají příslušným šablonám v příkazu get, všechny pozice jsou relativní ke stejnému počátku (začátek konkordance).

sort conclist criteria uniq setřídí seznam konkordancí <conclist> podle kritérií <criteria>. Je-li číslo <uniq> != 0, odstraňují se redundance.

<criteria> má formát: {{[<|>][+|-][?|!]attr lpos rpos} ...} – seznam slov, která ve své podstruktuře určují jméno pozičního atributu <attr>, podle kterého se bude třídit a začátky a

konce intervalů, které se budou lexikograficky porovnávat (nebo antilexikograficky pokud `<rpos>` předchází `<lpos>`). `</>` znamená normální/reversní třídění (default je <) +/- znamená normální/retrográdní třídění (default je +) ?/! znamená neignorovat/ignorovat velikost písmen (default je ?).

uniq conclist criteria vyřadí redundance ze seznamu konkordancí `<conclist>`. Rovnost konkordancí se posuzuje podle kritérií `<criteria>`.

fdist conclist criteria limit sort vypočítá počty výskytů konkordancí ze seznamu `<conclist>`, které jsou shodné podle kritérií `<crits>` a vypíše seznam výskytů jejichž počet je větší než `<limit>`. Je-li číslo `<sort>` != 0, setřídí se seznam podle počtu výskytů.

Pro každý prvek seznamu je na výstup vypsán řádek ve tvaru

```
<count> <pos-attrs>
```

`<count>` – počet výskytů, `<pos-attrs>`: {`attr word ...`} – jméno attributu a seznam hodnot. Jednotlivé položky odpovídají kritériím `<crits>`.

expand conclist expands vyexpanduje hranice konkordancí nebo kolokací v seznamu konkordancí `<conclist>`.

`<expands>` má formát {{`coloc lpos rpos`} ...} – seznam slov, která ve své podstruktuře určují konkordanci (0) nebo kolokaci (1,...), která se má rozšířit a novou levou a pravou mez konkordance/kolokace.

del conclist {**range ...**} vymaže intervaly konkordancí ze seznamu `<conclist>`.

`<range>` má formát: {`beg end [pred]`} – interval konkordancí (zleva uzavřený, zprava otevřený), které se mají smazat, `<pred>` má formát: {`col ...`} – čísla kolokací které jsou-li > 0 musí být nastaveny nebo jsou-li < 0 nesmí být nastaveny nebo: num – číslo konkordance, která se má smazat

copy conclist1 conclist2 {**range ...**} zkopíruje intervaly konkordancí ze seznamu `<conclist1>` do `<conclist2>`.

move conclist1 conclist2 {**range ...**} přesune intervaly konkordancí ze seznamu `<conclist1>` do `<conclist2>`.

reduce conclist factor[%] proporcionálně vymaže některé konkordance ze seznamu `<conclist>` tak, aby jeho nová délka byla `<factor>` nebo `<factor>`/100 násobkem původní délky, pokud byl uveden znak %.

len conclist délka a velikost seznamu konkordancí `<conclist>` (velikost nemusí být známa, pokud zpracování dotazu nebylo dosud ukončeno, v takovém případě se délkou rozumí aktuální počet získaných konkordancí).

err conclist vrátí a vymaže chybové výstupy cqp spojené se seznamem konkordancí `<conclist>`. Je to zatím jediná možnost jak odhalit, že dotaz skončil chybou.

sync conclist počká na dokončení všech rozběhnutých operací na seznamu konkordancí `<conclist>` a opíše jeho jméno na výstup.

pend conclist vypíše na výstup počet rozběhnutých operací na seznamu konkordancí `<conclist>`.

list seznam všech seznamů konkordancí, které jsou k dispozici

lsdcpos conclist seznam všech pozičních atributů seznamu konkordancí `<conclist>`, které lze zobrazovat.

lsdcstr conclist seznam všech strukturálních atributů seznamu konkordancí `<conclist>`, které lze zobrazovat

save conclist uloží seznam konkordancí `<conclist>` tak, aby zůstal uchován i pro další běhy serveru.

rename conclist1 conclist2 přejmenuje seznam konkordancí `<conclist1>` na `<conclist2>`

erase conclist smaže seznam konkordancí `<conclist>`

clone conclist1 conclist2 vytvoří kopii seznamu konkordancí `<conclist1>`, pojmenovanou `<conclist2>`

status conclist vypíše paměťový status seznamu konkordancí:
MEM – pouze v paměti,
MOD – i na disku ale v paměti je modifikovaný,
SAV – i na disku a jsou totožné.

saveall uloží všechny modifikované seznamy konkordancí

C.4 Čítače

count count corpus query cut wnum wunit uplatní dotaz <query> na korpus <corpus>, výsledkem je čítač konkordancí pojmenovaný <count>. <query> je v formátu cqp, <cut> je maximální počet konkordancí, <wnum> odpovídá klausuli 'within #' a <wunit> je příslušný strukturální atribut.

freq count corpus pos-attr value spočítá počet výskytů hodnoty atributu <pos-attr> <value> v korpusu <corpus> do čítače <count>.

cntget count počká na dokončení čítače <count> a vypíše jeho stav.

cnterr count vrátí a vymaže chybové výstupy cqp spojené s čítačem <count>. Je to zatím jediná možnost jak odhalit, že dotaz skončil chybou.

cntstate count stav a nejvyšší předpokládaný konečný stav čítače <count> (druhá položka nemusí být známa, pokud zpracování dotazu nebylo dosud ukončeno, v takovém případě se stavem rozumí aktuální počet získaných konkordancí). Třetí položka ve výstupu indikuje, zda čítání ještě probíhá (!= 0) nebo zda již bylo dokončeno (0).

cntlist seznam všech čítačů, které jsou k dispozici

cnterase count smaže čítač <count>.

Literatura

[AEO90] B. Altenberg and M. Eeg-Olofsson. Phraseology in spoken english: Presentation of a project. In J. Aarts and W. Meijs, editors, *Theory and Practice in Corpus Linguistics*. Rodopi, Amsterdam and Atlanta, 1990.

[And93] Jiří Anděl. *Statistické metody*. MATFYZPRESS, Praha, 1993.

[Bal97] Catherine N. Ball. Concordances and corpora. `http://www.georgetown.edu/cball/corpora/tutorial.html`, 1997. online tutorial.

[Bur95] Lou Burnard, editor. *Users Reference Guide for the British National Corpus*. Oxford University Computing Service, May 1995.

[CES96] Corpus encoding standard. `http://www.cs.vassar.edu/CES/`, 1996.

[CH90] Kenneth Ward Church and Patrick Hanks. Word association norms, mutual information, and lexicography. *Computational Linguistics*, 16(1):22–29, March 1990.

[Chr95] O. Christ. *The XKWIC User Manual*, 1995.

[Cor] Corpora. `http://www.hd.uib.no/corpora/`. International email discussion list.

[CT94] W. B. Cavnar and J. M. Trenkle. N-gram-based text categorization. In *Proceedings of Third Annual Symposium on Document Analysis and Information Retrieval*, pages 161–175, Las Vegas, NV, April 1994.

[Dun94] Ted Dunning. Statistical identification of language. Technical Report MCCS-94-273, 1994.

[egr] Grep(1) - print lines matching a pattern. `man egrep`.

[FBY92] William B. Frakes and Ricardo Baeza-Yates, editors. *Information Retrieval: Data Structures & Algorithms*. Prentice Hall, 1992.

[Fil00] Pavel Filipenský. Systém pro správu a kontrolu korpusových dat. Master's thesis, Fakulta informatiky Masarykovy university, 2000.

[FK79] W. Nelson Francis and Henry Kučera. *Brown Corpus Manual*. Brown University, Providence, Rhode Island, revised and amplified edition, 1979.

[Fon94] Th. Fontenelle. What are collocations? – the DECIDE approach. DECIDE Deliverable D-2a, DECIDE, November 1994.

[GS95] Michel Goossens and Janne Saarela. A practical introduction to SGML. *TUGboat*, 16(2):103–145, June 1995.

[Hea99] Marti A. Hearst. *Modern Information Retrieval*, chapter User Interfaces and Visualization, pages 257–323. Addison-Wesley, 1999.

[HH98] Jan Hajič and Barbora Hladká. Tagging inflective languages: Prediction of morphological categories for a rich, structured tagset. In *Proceedings of the Conference COLING - ACL '98*, Montreal, Canada, 1998.

[HR99] Jaroslava Hlaváčová and Pavel Rychlý. Dispersion of words in a language corpus. In Matoušek et al. [MMkS99], pages 321–324.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. In *Inst. Radio Engineers*, pages 1098–1101, September 1952.

[ISO86] ISO 8879:1986(E). Information processing – Text and Office Systems – Standard Generalized Markup Lnaguage (SGML). Geneva, 1986.

[ISO99] ISO/DIS 8859-2 Information processing – 8-bit single byte coded graphic character sets – Part 2: Latin alphabet No. 2. Geneva, 1999.

[Kas] Jan Kasprzak. cstocs – charset encoding convertor for the czech and slovak languages. `man cstocs`.

[Knu73] Donald E. Knuth. *The Art of Computer Programming, Sorting and Searching*. Addison-Wesley, Reading, MA, 1973.

[Law98] John M. Lawler, editor. *Using Computers in Linguistics: A Practical Guide*. Routledge, 1998.

[loc] locale — description of multi-language support. `man 7 locale`. POSIX.1.

[LW96] G. Leech and A. Wilson. Recommendations for the morphosyntactic annotation of corpora. Expert Advisory Group on Language Engineering Standards document EAG-TCWG-MAC/R, EAGLES, March 1996.

[Mas96] Oliver Mason. Corpus access software: The CUE system. *TEXT Technology*, Vol 6(No 4):p257–266, 1996.

[Mil93] G. A. Miller. Five paper on wordnet. Technical report, 1993.

[MMkS99] Václav Matoušek, Pavel Mautner, Jana Ocelíková, and Petr Sojka, editors. *Text, Speech, Dialogue — Second International Workshop, TSD'99*, Berlin, Sep 1999. Springer-Verlag.

[Oak98] Michael. P. Oakes. *Statistics for Corpus Linguistics*. Edinburgh University Press, Edinburgh, 1998.

[Ous94] John K. Ousterhout. *Tcl and Tk Toolkit*. Addison-Wesley, 1994.

[Pal96] Karel Pala. Informační technologie a korpusová ligvistika. *Zpravodaj ÚVT MU*, 6(3 a 4), 1996.

[Pal99] Karel Pala. Semantic annotation of (czech) corpus texts. In Matoušek et al. [MMkS99], pages 56–61.

[PRS97] Karel Pala, Pavel Rychlý, and Pavel Smrž. Desam – approaches to desambiguation. Technical Report FI-MU-97-09, Faculty of Informatics, Masaryk University, Brno, 1997.

[PRS98] Karel Pala, Pavel Rychlý, and Pavel Smrž. Corpus annotation in inflectional languages: Czech. In A Min Tjoa and Roland R. Wagner, editors, *Ninth International Workshop on Database and Expert Systems Applications*, pages 149–153. IEEE Computer Society, 1998.

[RSF99] Pavel Rychlý, Pavel Smrž, and Pavel Filipenský. Document multiplicity elimination and corpora management. In *Proceedings of SCI/ISAS'99*, pages 231–235, Orlando, Florida, 1999.

[Ryc98] Pavel Rychlý. The Improvement of Common Statistical Measure. In Petr Sojka, Václav Matoušek, Karel Pala, and Ivan Kopeček, editors, *Text, Speech, Dialog*, pages 109–112, Brno, Czech Republic, Sep 1998. Masaryk University Press.

[SB99] J. H. Sinclair and J. Ball. Peliminary recommendation on text typology. EAGLES Document EAG-TCWG-TTYP/P, EAGLES, June 199.

[SC96] B. M. Schulze and O. Christ. *The CQP User's Manual*, 1996.

[Sed99] Radek Sedláček. Morfologický analyzátor češtiny. Master's thesis, Fakulta informatiky Masarykovy university, Brno, 1999.

[SH94] Bruno Maximilian Schulze and Ulrich Heid. State-of-the-art survey of corpus tools. DECIDE Deliverable D-1b II, DECIDE, November 1994.

[SMB94] C. M. Sperberg-McQueen and Lou Burnard. Guidelines for electronic text encoding and interchange. Document No. TEI P3, Text Encoding Initiative, Chicago, Oxford, April 1994.

[Veb99] Marek Veber. Ced – program for corpora editing. Technical Report FIMU-RS-99-04, Faculty of Informatics, Masaryk University, September 1999.

[vH95] Eric van Herwijnen. *Practical SGML*. Kluwer Academic Publishers, second edition, 1995.

[W3C94] The World Wide Web Consortium. `http://www.w3.org/`, October 1994.

[W3C98a] HTML 4.0 specification. `http://www.w3.org/TR/REC-html40`, April 1998.

[W3C98b] Extensible Markup Language (XML) 1.0. `http://www.w3.org/TR/REC-xml`, February 1998.

[WMB99] I. H. Witten, A. Moffat, and T. C. Bell. *Managing gigabytes : compressing and indexing documents and images*. Morgan Kaufmann Publishers, Inc., San Francisco, CA, second edition, 1999.

[Zip49] G. K. Zipf. *Human Behaviour and the Principle of Least Effort*. Addison-Wesley, Reading, MA, 1949.