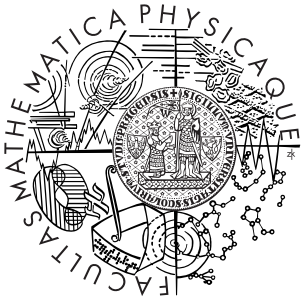
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Ondřej Kazík

Algoritmy dělení slov na slabiky

Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Jan Lánský
Studijní program: Obecná informatika

2007

Chtěl bych zejména poděkovat vedoucímu své bakalářské práce Mgr. Janu Lánskému za četné podněty a za pomoc při řešení některých problémů, které se vyskytly. Dále pak své rodině za podporu a trpělivost.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 23. července 2007 Ondřej Kazík

Obsah

Název práce: Algoritmy dělení slov na slabiky
Autor: Ondřej Kazík
Katedra: Katedra softwarového inženýrství
Vedoucí bakalářské práce: Mgr. Jan Lánský
e-mail vedoucího: lansky@ksi.ms.mff.cuni.cz

Abstrakt: Na kratších a středně dlouhých textech v morfologicky bohatých jazycích se projevila vhodnost komprese na základě slabik. S ohledem na správné dělení je však potřebná relativní variabilita návrhu definic jazyka a algoritmů dělení na slabiky. V této práci představujeme systém SCORP zahrnující jazyk, jeho překladač do podoby pseudokódu a interpret parseru v rámci kompresních algorimů. S jeho pomocí jsme vytvořili český algoritmus dělení na slabiky obsahující znalost jazyka. Jeho výsledky a kompresní poměry slabikových kompresních algoritmů jsou porovnány s odpovídajícími výsledky univerzálních metod dělení.

Klíčová slova: dělení na slabiky, překladače programovacích jazyků, komprese textů

Title: Algorithms of decomposing words into syllables
Author: Ondřej Kazík
Department: Department of Software Engineering
Supervisor: Mgr. Jan Lánský
Supervisor's e-mail address: lansky@ksi.ms.mff.cuni.cz

Abstract: The syllable-based compression gives good results in small or middle-sized text documents written in the richly morphological languages. The problem of decomposition of the words into syllables requires relative variability of the design of the language definitions and division algorithms. We propose SCORP system for this reason. This system includes programming language, compiler and interpret which is incorporated in compression algorithms. In this language we create specific Czech algorithm decomposing words into the syllables. Its result and compression rates are compared with the universal methods of division.

Keywords: syllable-based compression, compilers, syllable division

Kapitola 1

Úvod

V práci [3] byly testovány nové možnosti komprese textu, které se orientovaly na slabiky jako základní element komprese. Ve sledovaných jazycích, to jest anglickém (jazyk s jednoduchou morfologií) a českém (se složitou strukturou tvoření a ohýbání slov), byla provedena měření v porovnání algoritmů LZWL a Huffmanova kódování v jejich písmenné, slovní a slabikové verzi. Výsledky prokazovaly slibnost naznačené cesty na středně rozsáhlých textech a jazycích s bohatým tvaroslovím.

V citované práci se však současně projevila obtížnost problému správného dělení textu na slabiky, jeho vágnost, neurčitost a závislost na konkrétním jazyce. Pro zjednodušení úlohy zde byly proto využity univerzální algoritmy dělení, které byly na něm nezávislé. Jednotlivé algoritmy pak byly porovnány v úspěšnosti, entropii pozorovaných slabik i vlivu na kompresní metody.

Naším úkolem je navrhnout rozhraní, účelový programovací jazyk, ve kterém lze naprogramovat modul definující algoritmy dělení na slova a slabiky, a jeho překladač do binární podoby mezikódu. Ten bude interpretován v rámci těchto slabikových kompresních metod. Dále se pokusíme otestovat obecnost zvoleného řešení vytvořením českého, jazykově správného algoritmu dělení na slabiky. Otestujeme hypotézu, že specifický algoritmus zvyšuje úspěšnost dělení a vlivu této úspěšnosti na kompresi.

Kapitola 2

Slabiky

Slabikou rozumíme ([1]) základní jednotku řeči, je tvořena jádrem (to jest slabikotvorný základ – většinou samohláska) a tak zvanými svahy (skupiny souhlásek před a za jádrem). V promluvě přejímá zejména fonetickou funkci, tvoří rytmus řeči. V morfologicky bohatých jazycích má však dělení slova na předpony, kmen, přípony a koncovku a jejich variabilita silný vliv na slabičný charakter slova a slabika tak zčásti získává i významovou funkci. Například ke slovesu *skočit* lze vytvořit desítky sloves odvozených předponou (*proskočit*, *vyskočit*, *nezaskočit*, ...) a ke každému z nich 24 různých gramatických tvarů. Tedy počet různých tvarů odvoditelných z jazykového kořene *skoč* překračuje 400, jsou však tvořena pouze několika desítkami různých slabik ([3]). Právě nižší počet unikátních slabik v textu je předurčuje jako vhodnou základní jednotku pro kompresní algoritmy.

Na druhou stranu však rozporné místo slabiky ve struktuře jazyka vnáší do dělení slov nemalé problémy. Jde v češtině zejména o tyto případy ([6], [7]):

1. Často užívaná slova ztrácejí svůj původní charakter spojení předpony a kořene a stávají se celým novým kořenem (například *roz-um* se mění na *ro-zum*).

2. Neexistuje úmluva ohledně jednoznačné dělby slov, např. slovo *sestra* lze dělit třemi způsoby (*se-stra*, *ses-tra* nebo *sest-ra*).

3. Dělba kmene se změní po přidání přípony nebo koncovky, např. kmen *hrad* samostatně je nedělitelný, ale při skloňování se rozpadá na dvě slabiky (*hra-du*).

4. Zvláštní skupinu tvoří složená slva (např. *tři-a-třiceti-letý*). V českém jazyce nehrají složená slova tak významnou roli jako v němčině, přesto se zde vyskytují.

5. Dělení slov záleží také na významu slova: *na-rval* a *nar-val*.

Ne všechny zvláštnosti je tedy možné při konstrukci algoritmu dělení slov na slabiky plně zohlednit. Pro účely komprese však úplná korektnost není nutná, postačuje rozdělení slov na často se vyskytující skupiny písmen. V této práci poukážeme na to, že algoritmus, který bude respektovat charakteristiku jazyka, bude mít tyto statistické vlastnosti výhodnější.

Při své další práci užívnáme formalizaci pojmu slabiky z [3]. Zde je text rozdělen na slova pěti druhů. Jde o slova složená z malých písmen, slova složená z velkých písmen, slova s prvním písmenem velkým, slova z číslic a slova ze speciálních znaků. Písmenná slova se dále dělí na slabiky, tedy slova obsahující právě jedno maximální podslovo samohlásek. Jde nám tedy o to, vytvořit co nejuniverzálnější systém, umožňující definici algoritmů dělení pro parser v rámci různých jazyků a metod. V dalších kapitolách si popíšeme systém SCORP, který tento úkol plní, a dále se pokusíme v něm navrhnout základní univerzální algoritmy a algoritmus specifický pro český jazyk.

Kapitola 3

Systém SCORP

V této kapitole popíšeme způsob definice základních množin (velká, malá písmena a číslice), algoritmů (dělení slov, rozpoznání samohlásek, dělení na slabiky) a převodních struktur na velká a malá písmena tak, jak je rozvrhnut v systému SCORP.

Systém SCORP je založen na definici jazyka dle [3]. Skládá se ze dvou základních částí:

1. Překladač ze zdrojového souboru napsaného v jazyce SCORP do binárního souboru, který obsahuje informace o proměnných a konstantách a dále vlastní kód jednotlivých procedur (`GET_WORD`, `LETTER_TYPE`, `GET_SYLLABLE`, inicializační část) pro virtuální stroj.

2. Intepret dělící text na slova a slabiky pomocí běhu virtuálního stroje, který vykonává program z binárního souboru.

Systém tak umožňuje oddělenou definici algoritmů a jazyka bez nutnosti znovu překládat vlastní program, který dělbu na slabiky využívá. V oddílu 3.1 se pokusíme vyložit vlastnosti jazyka a v oddílu 3.2 pak cílový virtuální stroj se svou instrukční sadou.

3.1 Popis jazyka SCORP

Jazyk SCORP (Syllable-oriented Compiler Of Reduced Pascal) vychází koncepčně z jazyka Pascal. Kvůli úloze, pro kterou byl navrhnut, z něj bylo mnoho konstrukcí vypuštěno a několik nových přidáno. Základní struktura programu však zůstává zachována. Program začíná klíčovým slovem

PROGRAM s názvem programu, následuje deklarační část, kde je možné definovat konstanty, globální proměnné a procedury. Program končí inicializačním kódem a znakem tečky. Takže kostra programu by mohla vypadat například takto:

```
PROGRAM nazev_programu;

{Deklarační část}
CONST
  {Deklarace konstant...}

CONST
  {povinné deklarace}
  {malá písmena}
  LOWERS = ['a','b','c','d'...];
  {velká písmena}
  UPPERS = ['A','B','C','D'...];
  {čísla}
  NUMBERS = ['0','1','2','3'...];
  {zobrazení na malá}
  TOLOWER = ('A'=>'a','B'=>'b','C'=>'c'...);
  {zobrazení na velká}
  TOUPPER = ('a'=>'A','b'=>'B','c'=>'C'...);

VAR
  {globální proměnné}

{deklarace procedur}
{deklarace GET_WORD dělící vstupní soubor na jednotlivá slova}
PROCEDURE GET_WORD;
VAR
  {lokální proměnné}
PREDICATES
  {nepovinná predikátová část}
BEGIN
  {vlastní kód procedury GET_WORD}
END;

{deklarace LETTER_TYPE určující typ znaku vzhledem
```

```
 ke kontextu typu token}
PROCEDURE LETTER_TYPE(left, letter, right :CHAR) :TOKEN;
{...}
BEGIN
END;

{deklarace GET_SYLLABLE dělící slovo na jednotlivé slabiky}
PROCEDURE GET_SYLLABLE;
{...}
BEGIN
END;

{inicializační část}
BEGIN
END.
```

V dalším textu se popíšeme některé rysy tohoto jazyka podrobněji.

3.1.1 Typy

Elementární typy byly převzaty z jazyka Pascal. Ty jsou využitelné ve všech procedurch. Jde zejména o typy:

INTEGER je 32-bitové celé číslo (odpovídá typu int), tomuto typu přísluší operátory `+`, `-`, `/` (celočíselné dělení), `*`, `<`, `>`, `<>`, `=`.

CHAR je znak (v této verzi 8-bitový) umožňující pracovat s položkami proudu. S typem `CHAR` pracují operátory `=`, `<>` a spolu s typem `SET` operátor `IN`.

BOOLEAN je výčtový typ s hodnotami `TRUE` a `FALSE` a operátory `AND`, `OR`, `NOT`.

Dále bylo nutné vytvořit nové typy:

TOKEN je výčtový typ umožňující definovat charakter znaku v proudu pomocí procedury `LETTER_TYPE` a dotazovat se na něj v `GET_SYLLABLE`. Zahrnuje hodnoty `CON` (souhláska), `VOW` (samohláska) a `UNKNOWN` (pro ostatní znaky).

SET je množina znaků. Tu lze instanciovat pouze jako pojmenovanou či nepojmenovanou konstantu a dotazovat se na prvky operátorem `IN`. Například definovat množinu obsahující znaky `a`, `b`, `c` je možné takto: `['a','b','c']`. Prostředí obsahuje identifikátor konstantní množiny `ANY_CHAR`, která zastupuje všechny znaky.

typ zobrazení je specifický typ konstanty, umožňující pouze dvě instance: `TOLOWER` definující chování standardní funkce stejného jména, která převádí znaky na malá písmena a `TOUPPER` provádějící analogickou akci pro velká. Implicitně jsou obě zobrazení identitami, tedy výsledkem `TOLOWER('A')` je znovu `'A'`. Pro změnu tohoto chování je třeba vytvořit konstantu tohoto jména:

```
CONST
  TOLOWER = ('A'=>'a','B'=>'b');
```

3.1.2 Proud

Každá procedura má svůj vlastní tzv. proud. Proud v tomto kontextu nazýváme abstrakci lineárního vstupu skládajícího se z posloupnosti znaků a dodatečných informací. Proud vždy začíná aktuálně prvním nezpracovaným znakem. Lze k němu přistupovat buď skrze standardní funkce nebo predikáty a regulární výrazy.

Pro každou proceduru představuje proud jinou strukturu. Pro `GET_WORD` se jedná o vstupní parsovaný soubor, od kterého jsou oddělovány jednotlivá slova. V `LETTER_TYPE` jde o posloupnost tří znaků slova, přičemž druhý je znak, jehož typ (typ `TOKEN`) určujeme, a zbývající dotváří jeho levý a pravý kontext. První či znak nemusí být definovaný a v takové případě je jeho hodnota `0`. Pro `GET_SYLLABLE` je vstupem posloupnost znaků slova opatřená typem z běhu procedury `LETTER_TYPE`.

3.1.3 Regulární výrazy a predikáty

Regulární výraz je zde specifická řídící struktura, umožňující přímý přístup k proudu a jeho parsování. Vyskytují se v hlavičce predikátů nebo jako parametr vestavěné funkce `MATCH(rx)`. Skládá se z těchto operátorů (seřazeny podle priority od nejmenší, prioritu můžeme upravit pomocí závorek):

Posloupnosti (`rx1 rx2 ... rxn`) odpovídá takový proud, jehož počáteční podřetězec odpovídá regulárnímu výrazu rx1, navazující podřetězec

výrazu rx2..., přičemž žádný z výrazů neselže. Jednotlivé podvýrazy jsou vykonávány za sebou.

Variantnímu výrazu (`rx1|rx2|...|rxn`) odpovídá proud, který odpovídá prvnímu úspěšnému podvýrazu `rx1...rxn`. Výraz skončí neúspěchem, pokud žádný z podvýrazů nebyl úspěšný. Upozorněme na tomto místě, že uvnitř varianty nesmí dojít k oddělení slova nebo slabiky. Za variantním výrazem (zde i s počtem variant rovným jedné) se dále může nacházet oddělovač slova (`= WORD`), slabiky (`= SYL`) nebo identifikátor proměnné(`= variable`), do které je uložen index konce odpovídajícího podřetězce. Oddělení příslušného podřetězce se ale provede pouze v případě, že proudu odpovídá celý regulární výraz. V rámci jednoho výrazu je však možné oddělit více slabik, ale pouze na začatku proudu (např. `"po" = SYL "de" = SYL`).

Cykly (`rx*` nebo `rx+`) umožňují otestovat proud, který odpovídá opakování (v případě `*` i prázdnému) regulárního výrazu `rx`. Ten je testován tak dlouho, dokud mu zbývající část proudu odpovídá. Zde se projevuje determinističnost těchto výrazů. Nikdy například neuspěje výraz `CON* 't'`, protože první část bere maximální podřetězec souhlásek, další znak tedy nebude samohláska a tedy ani `'t'`. Cyklus `rx+` (opakování 1 až n-krát) je pouze zkrácená forma výrazu `rx rx*`.

Řetězci (`"blabla"`) odpovídá takový proud, v kterém je posloupnost znaků (upravená implicitním zobrazením procedury) rovná posloupnosti znaků řetězce.

Zbytek proudu (`~`) bere maximální část proudu od aktuální pozice do konce. Uspěje vždy, když je aktuální začátek platnou pozicí proudu a za symbolem `~` se již žádný výraz nenachází. Toto je použitelné například v případě, pokud chceme oddělit zbytek slova jako slabiku (`(CON VOW+) = SYL ~ = SYL`).

Konec proudu (`$`) končí úspěchem, právě když je aktuální pozice prvním neplatnou pozicí proudu, tedy pokud je rovna délce proudu. Tento obrat umožňuje testovat, zda daným výrazem proud končí, například u koncovek (`"ší" = SYL $`).

Znakový výraz (`at1++at2--at3--at4`) je výraz odpovídající právě jednomu znaku (upravenému implicitním zobrazením procedury) na

akutální pozici. Jednotlivými složkami `at1`,... zde mohou být: *znak*, testovaný na rovnost, *množina*, které přísluší operace náležení nebo *token*, testující zda tento znak je samohláska či souhláska. Tento znakový výraz uspěje právě tehdy, když uspěje každá složka, které předchází operátor `++` nebo je úvodní složkou (v našem případě `at1`, `at2`), a zároveň neuspěje žádná složka, které předchází operátor `--` (zde `at3` a `at4`).

Predikáty procedury nazývám nepovinnou posloupnost pravidel, která je uvedena klíčovým slovem `PREDICATES`, z níž každé obsahuje hlavičku s regulárním výrazem a může obsahovat tělo, v němž se nachází příkaz. Příklad predikátu bez těla:

```
{Pravidlo oddělující ze vstupního textu slovo
s prvním velkým písmenem}
TEXT((UPPERS LOWERS+) = WORD).
```

Predikát s tělem:

```
{Pravidlo oddělující slabiku, přičemž polovina souhlásek
mezi samohláskami připadne této slabice a druhá polovina
slabice následující (algoritmus MR)}
WORD(CON* (VOW*) = beg_block (CON*) = end_block VOW):-
    CUTSTREAM((beg_block+end_block)/2).
```

Vykonávání příslušné predikátové části začíná zavoláním vestavěné funkce `EXECPRED` ve výkonné části procedury. Testovány jsou po sobě hlavičky pravidel, dokud není nalezena taková, která uspěje. V takovém případě jsou přijaty oddělení podřetězců vykonané v příslušném regulárním výrazu, přistoupí se k vykonání příkazu těla pravidla a běh predikátové části končí, funkce `EXECPRED` vrací `TRUE`. Pokud neuspěje žádné pravidlo, vrací tato funkce `FALSE`.

3.1.4 Procedury

Procedurou je podobně jako v Pascalu část zdrojového kódu uvozená klíčovým slovem `PROCEDURE` následovaným identifikátorem a seznamem parametrů a ukončený definicí výkonné části uzavřené v složeném bloku. Na rozdíl od Pascalu však můžeme definovat proceduru vracející typ (funguje jako funkce), a to označením typu za hlavičkou procedury. Vrácení výsledku

procedury se pak provádí buď vestavěným příkazem `RESULT(expr)` nebo přiřazením do proměnné identifikované jménem dané procedury. Dále může každá procedura obsahovat predikátovou část.

Každá procedura vlastní proud a implicitní zobrazení využívané regulárním výrazem (buď `TOLOWER`, `TOUPPER` nebo identita). Uvnitř výkonné části dochází pouze k volání vestavěných příkazů a funkcí. Není tedy umožněna rekurze. Lze definovat pouze tyto tři procedury:

- `GET_WORD` má za úkol rozdělit vstupní text v proudu na samostatná slova. Jejím implicitním zobrazením je identita. Predikáty mají hlavičku začínající identifikátorem `TEXT` a slova se v regulárních výrazech oddělují konstrukcí `= WORD`.

- `LETTER_TYPE` je procedura rozhodující na základě kontextu, zda symbol plní úlohu samohlasky (`VOW`), souhlásky (`CON`) nebo jde o nepísmenný znak (`UNKNOWN`). Dostává tři znakové parametry, a to levý kontext, samotné určované písmeno a pravý kontext. Stejně tak vstupní proud obsahuje pouze tyto tři znaky. Procedura vrací výsledek typu `TOKEN`. Implicitním zobrazením je opět identita a predikáty jsou uvedeny hlavičkou `TEXT(rx)`.

- `GET_SYLLABLE` má za úkol slovo (oddělené procedurou `GET_WORD`) rozdělit na slabiky. Vstupním proudem je tedy celé slovo spolu s otypováním znaků získané procedurou `LETTER_TYPE`. Jako implicitní zobrazení zde slouží `TOLOWER`. To umožňuje dělit slovo bez ohledu na velikost písmen. Predikáty jsou uvozeny `WORD(rx)` a uvnitř regulárních výrazů jsou slabiky odděleny konstrukcí `= SYL`.

3.1.5 Vestavěné funkce a procedury

Vestavěné funkce:

`LENGTH`	Funkce vrací délku zbytku proudu od aktuálního počátku.
`ISVALID(pos)`	Vrací `TRUE`, právě když `pos` je pozice v proudu, tedy je menší než jeho délka.
`NTHCHAR(pos)`	Vrací znak na pozici `pos` od aktuálního počátku. Pozice je číslována od nuly.
`ISTYPE(pos,tok)`	Vrací `TRUE`, právě když je znak na pozici `pos` v proudu otypován jako `tok`.

`MATCH(rx)`	Funkce končí s návratovou hodnotou `TRUE`, když počáteční podřetězec proudu odpovídá regulárnímu výrazu `rx`.
`ODD(ival)`	Funkce vrací `TRUE`, když je `ival` liché číslo.
`TOUPPER(c)`	Vrácí znak převedený na velké písmeno pomocí zobrazení `TOUPPER`.
`TOLOWER(c)`	Vrácí znak převedený na malé písmeno pomocí zobrazení `TOLOWER`.
`EXECPREDS`	Zahájí vykonávání predikátové části procedury a vrací `TRUE`, pokud ta končí úspěchem.

Vestavěné procedury:

`WRITE(val)`	Vypíše na standardní výstup hodnotu `val` (pokud je typu `CHAR`, `INTEGER`, `TOKEN` nebo `BOOLEAN`).
`CUTSTREAM(pos)`	Oddělí od proudu podřetězec délky `pos`. Zároveň posune o tuto hodnotu počátek proudu.
`RESULT(val)`	Nastavuje hodnotu výsledku aktuální procedury na `val`. Tentýž efekt mé přiřazení do proměnné se stejným jménem jako tato procedura.
`INC(var)`	Zvýší hodnotu proměnné `var` o jedna.

3.2 Virtuální stroj a instrukční sada

Po přeložení zdrojového souboru v jazyce SCORP vzniká cílový soubor. Nachází se v něm informace o typech proměnných, hodnoty konstant a vlastní výkonný kód jednotlivých procedur. Tento soubor je nahrán a interpretován. Těsně po načtení programu je spuštěna inicializačná část, nastaven vstupní soubor a buffer s právě děleným slovem `Word` je prázdný. Oddělení slabiky probíhá podle následujícího algoritmu:

```
IF Word prázdné THEN
  Word := slovo oddělené ze vstupního souboru pomocí GET_WORD
  pro každý znak Word
    zjistí typ znaku pomocí LETTER_TYPE
ENDIF
Syllable := slabika oddělená GET_SYLLABLE z Word
RETURN Syllable
```

Běh jednotlivých procedur nebo inicializační části zajišťuje virtuální stroj navržený jako zásobníkový stroj umožňující navíc některé specifické operace s proudem a vykonávání predikátové části. Virtuální stroj obsahuje:

- Vstupní proud, tím může být textový soubor nebo slovo.
- Fronta podřetězců proudu (slov či slabik) vznikající při běhu regulárního výrazu instrukcí `ACPT -1`. Zásobník je vyprázdněn na konci běhu regulárního výrazu a buď jsou podřetězce přijaty (instrukce `PRCT`) nebo v případě neúspěšném zamítnuty (instrukce `UNCT`).
- Implicitní zobrazení procedury (identita, `TOLOWER` nebo `TOUPPER`) využité v regulárních výrazech této procedury.
- Kód, tedy posloupnost instrukcí s operandy.
- Adresa predikátové části.
- Registr `PC` (čítač instrukcí).
- Zásobník hodnot (typu `INTEGER`, `CHAR`, `TOKEN` nebo `BOOLEAN`), s kterými pracují jednotlivé instrukce.
- Registr `SP` (ukazatel vrcholu zásobníku).

Nyní si stručně popíšeme instrukční sadu virtuálního stroje:

`JMP addr`	Nepodmínený skok na adresu `addr`.
`JT addr`	Ze zásobníku je vyvednuta hodnta typu `BOOLEAN` a pokud byla `TRUE`, pak skok na `addr`.
`JF addr`	Pokud je na vrcholu hodnota `FALSE`, skok na adresu `addr`.
`LD var`	Načtení proměnné `var` na zásobník.
`ST var`	Vyzvednutí hodnoty ze zásobníku a uložení do proměnné `var`.
`INC [var]`	Pokud je definovaná hodnota `var`, pak dojde k zvýšení této proměnné. V opačném případě inkrementace hodnoty na vrcholu zásobníku.
`CNST cnst`	Načtení kontanty `cnst` na zásobník.
`INCL cnst`	Vyzvedne hodnotu typu `CHAR` a pokud se nachází v množinové konstantě `cnst`, vrací `TRUE`, jinak `FALSE`.

`INCC cnst` Porovnává znak (upraveny implicitním zobrazením procedury) z proudu na pozici, která je na vrcholu zásobníku (ta zde zde zůstává) s hodnotou konstanty a vrací `TRUE` pokud ji odpovídá. Podle typu konstanty:

- `CHAR` – pokud je rovnost.
- `SET` – pokud znak je prvkem množiny.
- `TOKEN` – pokud má stejný typ (viz `LETTER_TYPE`).

`INCV var` Tatáž akce jako `INCC`, ale s proměnnou `var`.

`ACPT op` Úspěšné ukončení zpracování regulárního podvýrazu. Sejme z vrcholu zásobníku hodnoty začátku a konce podřetězce, ukládá konec a hodnotu `TRUE`. Navíc v případě, že je `op`:

- `-1`, je uložen podřetězec do fronty podřetězců proudu.
- nezáporný, je konec podřetězce uložen do proměnné `op`.

`STRC cnst` Porovnává proud s řetězcem. Ze zásobníku odebírá počátek podřetězce. V případě rovnosti ukládá pozici konce a hodnotu `TRUE`, jinak původní hodnotu a `FALSE`.

`FAIL` Neúspěšné ukončení podvýrazu. Bere hodnotu začátku a konce podřetězce a ukládá začátek spolu s hodnotou `FALSE`.

`PUSH` Uloží na zásobník kopii hodnoty z vrcholu.

`POP` Odebere jednu hodnotu ze zásobníku.

`EOS` Konec streamu (`$`). Vrací `TRUE`, pokud hodnota na zásobníku (která zde zůstává) je rovna délce proudu.

`TAIL` Zbytek proudu (`~`). Ukládá na zásobník hodnotu `-1`, považovanou za symbol konce proudu.

`PRCT` Úspěšné ukončení zpracování proudu regulárním výrazem. Přijmutí fronty podřetězců proudu.

`UNCT` Neúspěšné ukončení regulárního výrazu. Vyprázdnění fronty podřetězců.

`CLPR`	Volání predikátové části (`EXECPREDS`). Uložení adresy návratu na zásobník a skok na adresu predikátové části.
`RET`	Ukončení běhu procedury.
`RTT`	Úspěšný návrat z predikátu. Vyjmutí a skok na adresu návratu, uložení hodnoty `TRUE`.
`RTF`	Neúspěšný návrat z predikátu. Vyjmutí a skok na adresu návratu, uložení hodnoty `FALSE`.
`ADD`	Sečtení dvou hodnot ze zásobníku.
`SUB`	Odečtení dvou hodnot ze zásobníku.
`MUL`	Vynásobení dvou hodnot ze zásobníku.
`DIV`	Celočíselné vydělení dvou hodnot ze zásobníku.
`NEG`	Záporná hodnota.
`ODD`	Vrací `TRUE`, pokud je hodnota na zásobníku liché číslo.
`GRT`	Je větší. Vrací `TRUE`, pokud je první uložená hodnota větší než druhá.
`LSS`	Je menší.
`GEQ`	Je větší nebo roven.
`LEQ`	Je menší nebo roven.
`EQU`	Je rovno. Vrací `TRUE`, pokud jsou dvě hodnoty ze zásobníku rovny.
`NEQ`	Je různý. Vrací `TRUE`, pokud si dvě hodnoty nejsou rovny.
`AND`	Logická konjunkce dvou hodnot typu `BOOLEAN`.
`OR`	Logická disjunkce dvou hodnot typu `BOOLEAN`.
`NOT`	Negace hodnoty typu `BOOLEAN`.
`NTH`	Vrací hodnotu znaku z proudu, který je na pozici odebrané ze zásobníku.
`LEN`	Vrací délku proudu.
`TYPE`	Vyzvednuty hodnoty `tok` a `pos`. Vrací `TRUE`, pokud je znak na pozici `pos` v proudu typu `tok`.
`VLD`	Vrací `TRUE`, pokud je hodnota na zásobníku platnou pozicí proudu (menší než délka a větší nebo rovna nule).
`CUT`	Oddělí z proudu podřetězec, který začíná na pozici `0` a má délku rovnou hodnotě ze zásobníku.
`WRT`	Vypíše hodnotu.
`RES`	Uložení hodnoty výsledku procedury.
`UP`	Provede zobrazení `TOUPPER` na znak nacházející se na zásobníku.

`LO`	Provede zobrazení `TOLOWER` na znak nacházející se na zásobníku.
`NOP`	Žádná akce

Kapitola 4

Algoritmy dělení na slabiky

Algoritmy dělení slov na slabiky můžeme rozdělit na dva typy. Buď se jedná o algoritmus univerzální, který umožňuje rozdělit text na slabičné bloky ve všech jazycích, nebo jde o specifický algoritmus, který je závislý na vstupním jazyku, a tedy zpravidla využívá vlastnosti konkrétního jazyka, v kterém je vstupní text napsán. Systém je navrhnut tak, aby v něm byl umožněn popis obou tříd algoritmů. Nejdříve ukážeme definice jazyka, které jsou všem algoritmům společné.

V každém programu je třeba definovat v části `CONST` základní množiny, a to malá písmena, velká písmena a číslice. Definice malých písmen (písmeno `ß` je zde pouze malé):

```
LOWERS=['a', ... ];
```

Velká písmena:

```
UPPERS=['A', ... ];
```

Číslice:

```
NUMBERS=['0','1','2','3','4','5','6','7','8','9'];
```

Dále pak zobrazení na malá a velká písmena. Tato nejsou povinná, ale automaticky jsou nastaveny na identitu:

```
TOUPPER=('a'=>'A', ... );
TOLOWER=('A'=>'a', ... );
```

Dále následují definice jednotlivých pomocných konstant, umožňující práci dělící a značkovací procedury. Malé samohlásky:

```
s_vow=['a',... ];
```

Malé souhlásky (vynecháváme zde `l` a `r`, slabikotvorné souhlásky, které v kontextu souhlásek mohou plnit úlohu samohlásek):

```
s_con=['b','c',... ];
```

Obdobně se definují velké samohlásky (`c_vow`) a velké souhlásky (`c_con`).

V dalším oddílu se přistupuje k procedurám. V následujících implementacích jsou, jak procedura oddělující slova (`GET_WORD`), tak značkovací procedura (`LETTER_TYPE`), společné oběma třídám algoritmů. Procedura `GET_WORD`:

```
PROCEDURE GET_WORD;
VAR
  end_block:INTEGER;
  b:BOOLEAN;
PREDICATES
  {Slova typu Abcd}
  TEXT((UPPERS LOWERS+) = WORD ).
  {Slova typu ABCD}
  TEXT((UPPERS+) = WORD ).
  {Slova typu abcd}
  TEXT((LOWERS+) = WORD ).
  {Čísla 1234}
  TEXT((NUMBERS+) = end_block):-
    IF end_block>4 THEN
      CUTSTREAM(4)
    ELSE
      CUTSTREAM(end_block).
  {mezery a jiné znaky}
  TEXT((ANY_CHAR--NUMBERS--LOWERS--UPPERS*) = WORD).

BEGIN
  b:= EXECPREDS;
END;
```

Vykonávání procedury začíná zavoláním predikátové části pomocí funkce `EXECPREDS`. Zde první pravidlo odděluje slova s prvním velkým písmenem následovaným alespoň jedním malým písmenem. Další pravidla oddělují

slova složená z velkých písmen a malých písmen. Čtvrté pravidlo je příklad pravidla s tělem. Předpokládá se totiž, že číslo není delší než čtyři znaky. Tedy regulární výraz vyhledává maximální podřetězec číslic a jeho konec uloží do proměnné `end_block`. V těle pravidla je pak oddělena pouze taková část tohoto podřetězce, která nepřesahuje délku čtyři znaky. Poslední pravidlo odděluje maximální podřetězec nealfanumerických znaků.

Procedura `LETTER_TYPE` určuje na základě kontextu, zda daný znak plní úlohu samohlásky, souhlásky, nebo se vůbec nejedná o písmeno:

```
PROCEDURE LETTER_TYPE(left,letter,right:CHAR):TOKEN;
PREDICATES
  TEXT(ANY_CHAR s_vow++c_vow):-
    RESULT(VOW).
  TEXT(ANY_CHAR s_con++c_con):-
    RESULT(CON).
  TEXT(s_con++c_con ['l','r','L','R'] ANY_CHAR--s_vow--c_vow):-
    RESULT(VOW).
  TEXT(ANY_CHAR ['l','r','L','R']):-
    RESULT(CON).
  TEXT(~):-
    RESULT(UNKNOWN).
VAR
  b:BOOLEAN;
BEGIN
  b:=EXECPREDS;
END;
```

Znovu je valná část rozhodovacího algoritmu přesunuta do predikátové části. Levý kontext, písmeno a pravý kontext se nacházejí jak v parametrech procedury, přístupných přímo v kódu, tak v proudu, který porovnávají regulární výrazy. První pravidlo vrací samohlásku, pokud se jedná o samohláskový znak nezávisle na kontextu (zde je brán v úvahu pouze levý kontext, kterým může být i nulový znak, tedy se jedná o začátek slova). Povšimněme si, že návratová hodnota je nastavena vestavěnou funkcí `RESULT`, druhou možností by bylo přiřazení do identifikátoru procedury (`LETTER_TYPE := ...`). Druhé pravidlo obdobně obsluhuje souhlásku (kromě `l` a `r`). Další dvě pravidla určují typ znaků `l` a `r`. V prvním případě jde o slabikotvornou hlásku, pokud je za souhláskou a po něm nenásleduje samohláska (může se nacházet také na konci slova). V opačném případě jde o souhlásku. Poslední pravidlo

určuje typ nepísmenného znaku.

Ukázali jsme implementaci dělení na slova a určení samohlásek a souhlásek. Tyto jsou společné všem algoritmům a podstatně se měnit nebudou. Dále popíšeme implementaci algoritmů dělících na slabiky.

4.1 Implementace univerzálních algoritmů

Univerzální algoritmy dělící slova na slabiky jsou algoritmy, které neobsahují znalost o konkrétním jazyku a vycházejí z dělení písmenných znaků na samohlásky a souhlásky. Tyto algoritmy navazují na algoritmy P_U, jak jsou popsány v [3]. Založeny jsou na maximálních blocích samohlásek (posloupnost samohlásek, jež nemůže být prodloužena), které tvoří základ slabik. Souhlásky, které jsou před prvním blokem samohlásek ve slově, jsou přidány k tomuto bloku, a tedy přísluší k první slabice. Obdobně souhlásky za posledním blokem jsou k tomuto bloku přidány. Jednotlivé algoritmy se pak liší způsoby, jakými jsou souhlásky mezi dvěma bloky samohlásek rozděleny.

Každý z algoritmů je implementován v predikátové části. Nejdříve musí dojít k oddělení maximálního bloku nepísmenných znaků, aby další dělení neovlivňoval. Proto je prvním pravidlem vždy:

```
WORD((UNKNOWN+)=SYL).
```

Přistupme k vlastnímu popisu P_U.

Algoritmus P_{UL} přiřazuje všechny souhlásky mezi samohláskovými bloky k levému bloku. Toto je provedeno pravidlem:

```
WORD((CON* VOW* CON*)=SYL).
```

Toto pravidlo tedy přiděluje k první slabice počáteční blok souhlásek, stejně jako maximální blok souhlásek zprava. Při každém dalším volání tak zbývající slovo vždy začíná samohláskou a toto pravidlo tak přiděluje k levé slabice všechny souhlásky. Pokud je samohláska na konci slova, tvoří sama slabiku, což není vždy vhodné.

Další algoritmus, P_{UR}, přiřazuje souhlásky mezi bloky samohlásek k pravému (následujícímu) bloku. Pak jsou pravidla implementována takto:

```
WORD((CON* VOW*)=SYL CON* VOW).
WORD((CON* VOW* CON*)=SYL).
```

První pravidlo odděluje slabiku, za kterou následuje alespoň jeden blok samohlásek, tedy nekoncovou slabiku. Souhlásky nalevo od bloku přiřazuje k slabice a napravo nechává na další zpracování. Další je pak záchytné pravidlo pro koncovou slabiku (za kterou již nenásleduje žádný samohláskový blok), které k ní přidává i koncovou posloupnost souhlásek.

Další dva algoritmy, P_{UMR} a P_{UML}, dělí souhláskovou posloupnost mezi dvěma bloky o sudé délce ($2n$) na stejně dlouhé části, tedy n k levému a n k pravému bloku. Liší se ale v rozdělování lichých posloupností. P_{UMR} dělí souhláskovou posloupnost délky $2n + 1$ tak, že k levému bloku přidává n souhlásek a k pravému $n + 1$ souhlásek.

```
WORD(CON* VOW*=beg_block CON*=end_block VOW):-
  CUTSTREAM((beg_block+end_block)/2).
WORD((CON* VOW* CON*)=SYL).
```

První pravidlo ukládá do proměnné `beg_block` začátek a do `end_block` konec posloupnosti souhlásek za samohláskovým blokem. V těle je pak proveden `CUTSTREAM` na střední hodnotu mezi nimi. Celočíselné dělení zaokrouhluje dolů, tedy levý blok je kratší než pravý. Poslední pravidlo odděluje koncovou slabiku.

Algoritmus P_{UML} pak z $2n+1$ souhlásek přiřazuje $n+1$ k levé a n k pravé části. Výjimka nastane, když mezi dvěma bloky je pouze jedna souhláska. Potom je přiřazena k pravému bloku. Tím se zabrání tomu, aby na konci slov zůstaly samostatné bloky samohlásek.

```
WORD((CON* VOW*)=SYL CON VOW).
WORD(CON* VOW*=beg_block CON*=end_block VOW):-
  CUTSTREAM((beg_block+end_block+1)/2).
WORD((CON* VOW* CON*)=SYL).
```

Prvním pravidlem se oddělí blok s předcházejícími souhláskami, když je mezi ním a následujícím blokem jedna souhláska. Další pak vykonává rozdělení souhláskové posloupnosti mezi dvěma bloky, pokud je delší než jeden znak. Třetí pravidlo opět zajišťuje zpracování poslední slabiky.

4.2 Implementace specifického algoritmu pro český jazyk

Pro ověření funkčnosti základního konceptu systému SCORP bylo přistoupeno k vytvoření jednoduchého specifického algoritmu dělení v

českém jazyce. Tato úloha je nejčastěji řešena v rámci typografických systémů k dělení slov na konci řádky ([7], [6]) nebo úloh syntézy a rozpoznání řeči založené na slabikových segmentech ([2]). Avšak zatímco v prvním případě jde o nalezení všech možných slabikových švů a vybrání nejvhodnějšího vzhledem k estetické stránce a v druhém je přihlíženo spíše k zvukové funkci, vzrůstala složitost dělení, ať už byla za algoritmus zvolena slovníková metoda, množina vzorů nebo morfologický analyzátor. Vzhledem k těmto obtížím, pravděpodobné nižší citlivosti kompresních metod na správnost a potřebě jednoznačnosti dělení byl vybrán algoritmus [5].

Tento algoritmus je popsán ve formě pseudokódu a dostává na vstupu slovo jako celek. Výstupem je pak posloupnost bodů dělení uvnitř tohoto slova. V první řadě algoritmus označkuje slovo jako posloupnost samohlásek a souhlásek (včetně dvojhlásek *au*, *ou* a *ch*) s výjimkou hlásek *l* a *r*. Ty jsou následně považovány za samohlásku či souhlásku obdobně našemu přístupu v univerzálních algoritmech. Dále se přistupuje k samotnému dělení. Nejdříve je v několika fázích testováno slovo na známé předpony (*nej*, *beze*, *zpoza*, *ne*, *ně*, *před*, *ode*, *přes*, *pře*...) a pokud nějaké z nich odpovídá, je oddělena a pokračuje se v průchodu slovem. Po každé jsou brány v potaz některé výjimky, kdy toto slovo začíná danou skupinou, ale není jeho předponou (například skupina končící souhláskou by měla být následována další souhláskou, ty které končí na samohlásku nejsou následovány dvojicí souhlásek, z níž druhá je *c*, *n* nebo *š*). Po provedení této části se v hlavním cyklu dělí slovo očištěné od těchto předpon. Vždy je vyhledána první samohláska a dále samohláska, která ji následuje. Ty tvoří jádra prvních dvou slabik. Pokud takové neexistují, je zbytek slova považován za jednu slabiku a dělení končí. Uvnitř cyklu je řízení rozděleno podle počtu souhlásek mezi těmito dvěma jádry. Pokud jde o jednu nebo žádnou souhlásku (vzor VV, VCV, kde V je samohláska a C souhláska), je dělící bod nastaven hned za první samohlásku (např: *cti-utrhání*, *člo-věk*). V případě, že jsou takové souhlásky dvě (vzor VCCV), je situace složitější. Implicitně se tato posloupnost dělí mezi oběma souhláskami (*zpát-ky*). V úvahu se berou některé předpony, které nemohli být obslouženy dříve (*na-*, *po-*), a dále některé kombinace souhlásek, které odkazují pravděpodobně na nějaký kořen nebo příponu (například *kl* v *dů-kladně*, *vš* v *a-však*). Pokud jsou mezi jádry tři a více souhlásek (VCCCV), implicitně se odděluje hned po první (*zpravodaj-ství*, *tak-hle*). Výjimky opět tvoří předpony (*na-*, *po-*), některé komplexy souhlásek (*stř*: *zpro-středkování*), pravé svahy kořenů (*st*: *rost-lina*) popřípadě příponové hlásky (*n*: *prázd-né*).

Tento algoritmus by se dal implementovat ve SCORP v plné podobě, ale

vytvářelo by to některé potíže. Jde zejména o oddělování předpon v první fázi. Následnou analýzou bylo zjištěno, že nevznikají větší odchylky, když k testování předpon dochází při každém dělení přednostně. Umožňuje to implementovat celý algoritmus v predikátové části procedury `GET_SYLLABLE`. Vzhledem k tomu, že je algoritmus v původní verzi po rozdělení na jednotlivé vzory většinou uspořádán jako posloupnost, v které výjimka předchází obecnějšímu pravidlu, kopírují pravidla programu toto rozdělení. Pokud dochází k větvení algoritmu, je stejného efektu dosaženo buď propagací nepřípustného případu směrem před pravidlo a vytvoření výjimky (např. vzor CVCCV, který ale nezačíná *na-* nebo *po-*, řádka 70), nebo se řeší vytvořením těla obecnějšího pravidla (pravidlo vzoru VCCC+V podle koncové posloupnosti). Dalším úskalím bylo to, že algoritmus považuje dvojhlásky za jednu hlásku, na rozdíl od SCORP, který je spíše znakově orientován. Program to obchází tak, že hned třetí a čtvrté pravidlo odděluje dvě samohlásky, které následují po sobě, pokud není jednou z dvojhlásek *au*, *ou* nebo *eu* (přesněji všechny, které nezačínají *a*, *o*, *e* a končí *u*, a pak takové, které na *u* nekončí – takové řešení negace je určeno povahou pravidel a jejich výrazů). Při dalším průchodu, se pak na místě jader, kde se má nacházet samohláska, vyskytuje výraz `VOW+`, popřípadě `VOW*`, tedy jde o jednu samohlásku nebo jednu z dvojhlásek. Souhlásky tam, kde je to nutné, jsou zapsány jako variantní výraz `"ch"|CON`, což dává parseru možnost na aktuálním místě řetězce odebrat podřetězec *ch* ještě před testem na souhlásku.

V podstatě tedy jsou pravidla tohoto algoritmu seřazeny v tomto pořadí:

1. Speciální přednostní pravidla: nepísmenných znaků, poslední slabiky a oddělení samohlásek.
2. Pravidla oddělení předpon: *ne*, *nej*, *beze*, *bio*, *bez*, *při*, *vze*...
3. Pravidlo vzoru VCV (`WORD((CON* VOW+)=SYL "ch"|CON VOW).`)
4. Pravidla vzoru VCCV.
5. Pravidla s třemi a více souhláskami mezi oběma jádry.

Takto vytvořená pravidla odpovídající původnímu algoritmu podléhala následnému testování na menším souboru textů. Celková struktura obstála, ale některé výjimky v dělení byly v reálných případech sporné. Pokud připouštělo dané slovo takovou možnost, bylo původní dělení zachováno, i když nevyznělo ideálně. Přesto došlo k některým změnám:

- Z předpon byla vypuštěna předpona *re*, naopak přidána předpona *vý*.
- Oddělení předpon končících na samohlásku původně proběhlo pokud dvojice souhlásek za touto skupinou neobsahovala *c*, *n* nebo *š*. Takové omezení se zdá příliš silné, přidány přípustné varianty *zn*, *chc*, *vn*, *hn*, *sch*, *pn* a *tn* (např. *vy-značit*).
- Zavedeno oddělení předpon *nad-*, *na-* a *po-* u vzoru VCCV
- U stejného vzoru na řádce 89 odebrány dvojice hlásek *vt* (*stav-te*), *dm* (*sed-mi*). Naopak přidány *šp*, *šl*, *šv*, *sp*, *sv*, *tř*, *vs*.
- U vzoru VCCCV na řádce 63 je odebrána dvojice *kt* (*elek-tron*)
- U stejného vzoru se na řádce 66 spíše než první samohlásku hodí oddělovat poslední dvě a obdobně na řádce 68 poslední příponová souhláska.

Kapitola 5

Výsledky experimentů

V této kapitole přistoupíme k ověření funkčnosti systému SCORP na úloze komprese textových souborů na základě slabik. Budeme sledovat závislost jednotlivých ukazatelů (úspěšnost, počet unikátních slabik, entropie a kompresní poměr) u všech univerzálních metod dělení a specifického algoritmu na reálných textech.

5.1 Testovací množina dokumentů

Pro zjištění těchto údajů je potřeba vybudovat reprezentativní vzorek textů českého jazyka. Za tím účelem byla vytvořena množina dokumentů T_{CZ}, obsahující 69 dokumentů o celkové velikosti 15 MB a kódované v CP1250. Jedná se z velké části o beletrii (například část tvorby Karla Čapka), dále texty básnické a historické. Texty pocházejí z volně dostupných internetových zdrojů (server eKnihy).

Celá množina T_{CZ} byla použita k extrakci statistických údajů, četností znaků a charakteristických slabik. Tyto údaje byly následně užity pro srovnání kompresních algoritmů běžících na všech textech z množiny T_{CZ}. Empiricky bylo zjištěno, že je takový postup přípustný a že není nutné zavádět dvě disjunktní množiny dokumentů.

5.2 Správnost dělení a entropie

Nejdříve porovnáme jednotlivé algoritmy v parametrech týkajících se kvality dělení na slabičné elementy. Jedná se o entropii, správnost dělení a počet

unikátních slabik v závislosti na jejich celkovém počtu. Správnost nemohla být sledována na celé množině dokumentů, proto je odhadnuta na základě menšího úseku textu, náhodně vybraného z množiny T_{CZ}. V případě, že dané slovo připouštělo více možností dělení, byly brány do úvahy všechny. Vzhledem k tomu, že univerzální algoritmy jsou založeny na maximálních blocích samohlásek, je počet slabik u všech stejný. Na druhou stranu specifický algoritmus odděluje slabičná jádra, která obsahují pouze jednu samohlásku nebo dvojhlásku. Tímto postupem vzniká více slabik (například slovo *samo-ú-čel-ný* obsahuje pět slabik, ale pouze čtyři samohláskové bloky).

Charakteristika	P_{UL}	P_{UR}	P_{UML}	P_{UMR}	P_{S87M}
Správnost	49 %	88 %	93 %	95 %	99 %
Slabik celkem	7319409	7319409	7319409	7319409	7340629
Unikátních	38526	27420	26327	25474	24629
Slabiková entropie	8,0296	7,9268	7,9503	7,9720	7,8713

Tabulka 5.1: Charakteristiky dělících algoritmů

Jak vidíme v tabulce 5.1, nejhorších výsledků dosáhl algoritmus P_{UL}. Především přiřazuje veškeré souhlásky k levé slabice a tedy tvoří mnoho slabik začínajících samohláskou (*nen-ávr-atn-ě*). Proto i jeho nízká správnost. Vytváří také nejvyšší počet unikátních slabik a má nejhorší entropii. Zajímavých výsledků dosáhl P_{UR}. Ze všech univerzálních algoritmů měl nejnižší entropii, avšak správnost dosahovala pouze 88 % a i počet unikátních slabik je vyšší. Je to způsobeno tím, že český jazyk upřednostňuje levé svahy[1], a také tím, že rozdělením algoritmy třídy P_{UM} může vzniknout příliš mnoho mutací jinak častých slabik (například *výk-lad*). Oba dva, P_{UML} i P_{UMR}, tedy dosahovaly nejvyšší správnosti, podobný počet unikátních slabik i entropie. Vidíme také to, že ve všech těchto charakteristikách měl nejlepší výsledky specifický algoritmus. Produkuje sice vyšší počet slabik než univerzální, avšak nižší počet unikátních a s nižší entropií.

Tedy specifický algoritmus se více přibližuje ideálu rozdělení textu na slabiky. Ukazuje se nám zde, že správné dělení skutečně má vhodnější statistické vlastnosti pro kompresi než dělení univerzální. Problém opět dělají citoslovce, slova zvukového charakteru, která se vymykají pravidelnému chování. Naopak jednoslabičná slova a slabiky speciálního charakteru se započítávaly také do statistik, přestože jejich dělení je u všech algoritmů stejné, a tedy zvyšuje jejich úspěšnost.

[1] dle [2] většina slabik neobsahuje pravé svahy

5.3 Kompresní poměry

Nyní porovnáme naprogramované algoritmy dělení v úspěšnosti při kompresi textů v závislosti na velikosti dokumentů. V těchto porovnáních užíváme slabikových kompresních algoritmů LZWL a HuffSyllable popsané v [3]. Nejdříve byly pro každý dělící algoritmus vytvořeny slovníky slabik z celého T_{CZ} [2], následně pak byly komprimovány jednotlivé dokumenty.

V tabulce 5.2 jsou kompresní poměry LZWL v kombinaci s jednotlivými dělícími algoritmy v závislosti na velikosti dokumentů, v tabulce 5.3 stejné údaje pro HuffSyllable. Kompresní poměr je vyjádřen jako střední počet bitů potřebný k zakódování jednoho znaku (bpc).

Metoda	5 - 50 kB	50 - 100 kB	100 - 500 kB	500 - 2000 kB
LZWL+P_{UL}	4,11	3,79	3,58	3,33
LZWL+P_{UR}	4,04	3,73	3,55	3,31
LZWL+P_{UML}	4,04	3,74	3,55	3,31
LZWL+P_{UMR}	4,04	3,74	3,55	3,31
LZWL+P_{S87M}	4,02	3,72	3,53	3,29

Tabulka 5.2: Výsledky algoritmu LZWL

Metoda	5 - 50 kB	50 - 100 kB	100 - 500 kB	500 - 2000 kB
HS+P_{UL}	3,94	3,86	3,88	3,80
HS+P_{UR}	3,84	3,75	3,79	3,74
HS+P_{UML}	3,84	3,76	3,79	3,74
HS+P_{UMR}	3,85	3,76	3,80	3,75
HS+P_{S87M}	3,83	3,74	3,77	3,72

Tabulka 5.3: Výsledky algoritmu HuffSyllable

Vidíme zlepšení kompresních poměrů u obou metod se zvýšením jazykové správnosti u specifického algoritmu dělení. U univerzálních algoritmů dosahoval nejlepších výsledků s pomocí dělení P_{UR}, absolutně pak ve všech případech měl převahu algorimus specifický. Tyto výsledky vyplývají z entropií metod pozorovaných v předchozím oddílu.

[2] V obou případech pracujeme s inicializačními slovníky slabik A35, tedy obsahující slabiky, které se vyskytují alespoň v 35 % dokumentů.

Kapitola 6

Závěr

V práci [3] bylo poukázáno na možnost komprese textů na základě slabiky jako elementu komprese. Byly zde navrhnuty modifikace statistických a slovníkových metod komprese po slabikách. Jde zejména o metody LZWL a HuffSyllable. Následně byly tyto metody testovány na reprezentativním vzorku českého a anglického jazyka. Daný přístup se ukázal jako vhodnější než metody založené na slovech zejména pro středně velké dokumenty napsané v jazycích s bohatým tvaroslovím.

V této bakalářské práci jsme navázali návrhem rozhraní SCORP, umožňující definovat jazyk a metodu dělení, které budou užity v rámci parseru kompresních metod. Systém se skládá z programovacího jazyka SCORP, jeho překladače do podoby pseudokódu a interpretu virtuálního stroje, parsujícího vstupní text.

Programovací jazyk SCORP principiálně vychází z jazyka Pascal, který je účelově omezen na deklaraci základních množin velkých písmen, malých písmen, číslic, zobrazení na velká a malá písmena a procedur označující souhlásky, samohlásky a jiné znaky (`LETTER_TYPE`), oddělující slova (`GET_WORD`) a slabiky (`GET_SYLLABLE`). Dále je obohacen o predikátovou část a výrazy řídící oddělování základních jednotek (slov a slabik) ze vstupního proudu (vstupní text či slovo).

Překladač pak vytváří kód pro virtuální stroj, zásobníkový stroj rozšířený o instrukce usnadňující práce s proudem. Ten je integrován do parseru kompresních metod a řídí dělení textu na slabiky. Samotné parsování probíhá tak, že se ze vstupního souboru odděluje slovo, zjistí se typ jednotlivých jeho písmen a oddělují se z něho jednotlivé slabiky od začátku, dokud není vstupní slovo prázdné.

Pro demonstraci možností systému SCORP byly v tomto jazyce vytvořeny moduly implementující algoritmy dělení slabik. Šlo o univerzální algoritmy P_{UL}, P_{UR}, P_{UML} a P_{UMR}, jež byly součástí původní práce, na kterou se odkazuje v [3]. Tato sada metod byla navíc doplněna o specifický algoritmus P_{S87M}, respektující charakteristiky českého jazyka. Ten odděluje množinu obvyklých předpon slov a obsahuje informace o častých charakteristických kombinacích souhlásek kořenů a přípon.

Další měření prokázala, že specifický algoritmus, který dělí slabiky na základě znalostí o konkrétním jazyce, má lepší statistické vlastnosti. Protože souhlásky nesou výraznou část informace slova, je správné dělení klíčové. Především vytváří menší počet unikátních slabik s nižší entropií. To se promítlo i ve výsledcích obou testovaných kompresních algoritmů, LZWL a HuffSyllable, kde kompresní poměry specifického algoritmu byly ve všech případech vzhledem k ostatním metodám dělení lepší.

Na závěr naznačme další možnosti vývoje. I přesto, že programovací jazyk SCORP prokázal možnosti implementace algoritmů slabičného dělení, při návrhu narážel na některá omezení. Jedná se především o znakovou orientaci výrazů řídících oddělování slabik. Proto je myslitelná definice dvojhlásek, s kterými lze pracovat jako s jednou hláskou. Další cestou vývoje je užší provázanost predikátové části a hlavního bloku programu. Predikátová část se pak vyznačuje značnou jednoduchostí, ale i neefektivitou. To, že se pro každé zamítnuté pravidlo prochází řetězec, nás přivádí k možnostem částečného převodu pravidel na konečný stavový automat a jeho optimalizaci. V souvislosti s rozvojem vyjadřovacích schopností programovacího jazyka lze uvažovat i o vybudování sofistikovanějšího specifického algoritmu.

Literatura

[1] Kolektiv: *Malá československá encyklopedie*, Academia, Praha, 1987.

[2] Kopeček I.: *Syllable Segments in Czech*, Proceedings of the XXVII. Mezhvuzovskoy naucznoy konferencii, Vypusk 10, St. Petersburg, 1998. 60–64

[3] Lánský J., Žemlička M.: *Text Compression: Syllables*, V: Richta K., Snášel V., Pokorný J. (Eds.): DATESO 2005. ČVUT, Praha, 2005. 32–45

[4] Lánský J., Žemlička M.: *Compression of a Dictionary*, V: Snášel V., Richta K., Pokorný J. (Eds.): Proceedings of the Dateso 2006 Annual International Workshop on DAtabases, TExts, Specifications and Objects. CEUR-WS, Vol. 176. 11–20

[5] Smržová J.: *Slabičný algoritmus dělení českých slov*, http://nlp.fi.muni.cz/projekty/deleni-slov/.

[6] Sojka P., Ševeček P.: *Hyphenation in TEX—Quo Vadis?*, TUGboat: The Communications of the TEX Users Group, San Francisco: TEX Users Group **16(3)** (2004). 280–289

[7] Sojka P.: *Slovenské vzory dělení: čas pro změnu?*, V: Olšák P. (Ed.): SLT 2004 Proceedings of the 4th Seminar on Linux and TEX: SLT 2004, KONVOJ, Brno, 2004. 67–72