Generování vzorů dělení slov v UNICODE

David Antoš, Petr Sojka

Fakulta informatiky, Masarykova univerzita, Brno

Abstrakt: Článek popisuje techniku vzorů jako prostředek pro získávání informace z rozsáhlých dat a zpětné rozpoznávání. Typickou aplikací této techniky je dělení slov. Dosud chybí generátor vzorů dělení pro systém Ω (pro UNICODE) a rozšíření programu `PATGEN`, omezeného osmibitovým ASCII, není únosné. Proto vyvíjíme knihovnu `PATLIB` pro obecnou manipulaci se vzory a na ní postavíme generátor vzorů dělení slov v UNICODE. Popíšeme architekturu připravovaného systému a dále méně známou datovou strukturu dynamic packed trie, kterou lze výhodně použít pro efektivní ukládání konečných jazyků s výstupy. Vzory lze použít i pro rozpoznávání hranic složených slov, proto zmíníme návrhy na rozšíření následníků TeXu o klasifikované dělení s více typy dělících bodů a o automatické potlačování ligatur na švech složených slov.

Klíčová slova: rozpoznávání vzorů, dělení slov, Ω, dělení složených slov

1 Úvod

"Go forth and make masterpieces of hyphenation patterns . . . "
—Yannis Haralambous [4]

Téměř vše lze považovat za symbol a ze symbolů můžeme kombinovat *vzory*. Vzory jsou často zjevením a popisem vyšších zákonitostí. Hofstadter [8] považuje rozpoznávání vzorů za centrální pojem inteligence, a v tomto smyslu jsou zaměřeny i mnohé inteligenční testy.

Technika vzorů je efektivním prostředkem pro extrakci informací a rozpoznávání v datech. Byla použita v TEXu [11] jako elegantní a na jazyku nezávislé řešení pro kvalitní dělení slov; tento nádherný a efektivní algoritmus pak našel cestu využití i v téměř všech následně vznikajících sázecích systémech včetně těch komerčních, a dle našeho mínění si zaslouží ještě mnohem větší pozornost pro své široké možnosti použití.

Generování vzorů pro dělení pomocí programu `PATGEN` [15] nedostačuje plně současným potřebám, a pro jeho širší použití je třeba po dvaceti letech od jeho vzniku provést jeho mnohá zobecnění. Vznikl systém Ω [5] mj. s cílem umožnit přímou práci se slovy všech jazyků v Unicode (universální vzory dělení, kontextové substituce, překladové procesy Ω–ΩTP). Tyto nové výzvy a v dalším textu naznačené analýzy nás dovedly k závěru, že nejlepším řešením bude úplná reimplementace `PATGEN`u. V článku popíšeme základní principy techniky vzorů, jakým způsobem jsem vzory generovány a jak je potom TEX používá. Dále se

Jan Kasprzak, Petr Sojka (editoři): SLT 2001 – sborník semináře o Linuxu a TEXu, str. 23–32, 2001.

budeme věnovat architektuře připravovaného následníka PATGENu a jeho použití jako universální knihovny pro manipulaci se vzory.

2 Vzory

> *Middle English* patron *'something serving as a model', from Old French. The change in sense is from the idea of a patron giving an example to be copied. Metathesis in the second syllable occured in the 16th century. By 1700* patron *ceased to be used on things, and the two forms became differentiated in sense.*
> —Origin of word pattern: [3]

Vzory slouží pro rozpoznávání „zajímavých míst" v datech. Zajímavým místem může být hranice znaků, kde je v daném slovu povoleno dělit, hranice voda/les na leteckém snímku krajiny a podobně. Vzory jsou podslova dané množiny slov, mezi jejichž symboly je vyznačena informace o zajímavých místech.

Informace o těchto bodech je v principu dvojího druhu: jedna říká, že dané místo *je* místem zájmu, druhá, že *není*. Typickou reprezentací bývají přirozená čísla, lichá pro ne, sudá pro ano. Tedy vzory máme *pokrývací* a *zabraňující*. Lze také použít další speciální symboly, jako například tečku značící začátek nebo konec slova. Například české dělení obsahuje vzory `i1h`, `i3hl.` a `i2hl.`. Použití vzorů je toto: pro všechna podslova daného slova se najdou všechny odpovídající vzory. Tedy slovu `cihla` odpovídají vzory `i1h`, `i2hl` z naší množiny, takže máme `ci2hla`. Vzory se totiž *přebíjejí*, můžeme říct, že *soutěží*, a výsledkem je maximum z hodnot odpovídajících pozici mezi znaky. Pro pochopení, proč tomu tak je, musíme vědět, jak se vzory generují. Podrobný popis použití vzorů lze nalézt v [11, příloha H]. Laskavého čtenáře se zájmem o podrobnější a formálnější informace o vzorech odkazujeme na články [20,9,1], o nástroje na práci s konečně stavovými automaty pak na [16,10,17,13,2] .

3 Generování vzorů

> *"An important feature of a learning machine is that its teacher will often be very largely ignorant of quite what is going on inside, although he may still be able to some extent to predict his pupil's behaviour."*
> —Alan Turing, [22]

Generování *minimální* množiny soutěživých vzorů pokrývající daný jev plně je NP-úplné. Netrváme-li na minimalitě, iterativní metodou lze dosáhnout překvapivě dobrých výsledků a komprimovat informace ze vstupních dat do množiny vzorů. Popišme, jak takové generování probíhá.

Potřebujeme rozsáhlou množinu vstupních dat, ve které jsou označena místa zájmu, v našem případě se bude jednat o slova některého přirozeného jazyka a v nich označená místa, kde je dovoleno dělit.

Nyní budeme opakovaně procházet procházet vstupní data. Jednotlivé průchody nazvěme *úrovněmi*. V lichých úrovních generujeme pokrývací vzory, v sudých zabraňující.

V každé úrovni zvolíme *pravidlo*, pomocí něhož vybíráme *kandidáty* na vzory. V našem případě pravidlo může mít tvar „k-znakový podřetězec slova obsahující dělící bod“. Vybereme kandidáty na vzory a uložíme je ve vhodné datové struktuře. Ne každý kandidát je ovšem dobrý vzor, proto potřebujeme *pravidlo pro výběr vzorů*. Obvykle je rozumné projít všechny kandidáty na vzory a vyzkoušet jejich funkci (přitom se používají i vzory z předchozích úrovní, je o funkci celku) na slovech ze vstupních dat. Přitom spočteme, kolikrát kandidát pracoval dobře a kolikrát chyboval. Pravidlem pak může být funkce nad těmito hodnotami porovnávaná se zadanou hodnotou.

Kandidáty, které jsme v předchozím procesu označili za dobré, zařadíme mezi vzory. Tyto vzory vykazují ovšem stále určitou chybovost. Takže budeme pokračovat další úrovní, tentokrát sudou, v níž budeme vytvářet zabraňující vzory. Dobrým vzorem určité úrovně je kandidát, který opravuje chyby vzorů nižších úrovní.

Tímto způsobem v principu pracuje program `PATGEN`, s poměrně pevně danými pravidly pro výběr vzorů (lineární prahování). Vzory dělení slov pro TEX existují pro několik desítek jazyků, většina generována `PATGEN`em ze slovníku rozdělených slov. Musíme poznamenat, že se často také postupuje tak, že se vzory také vytváří částečně ručně (buď pro bootstrapping, nebo ručně).

Jaká je úspěšnost této techniky? Ze slovníku velikosti několika MB lze vytvořit vzory velikosti řádově desítek KB pokrývající nad 98 % dělicích bodů a s chybovostí pod 0,1 %. Četné experimenty ukázaly, že se vystačí často se čtyřmi úrovněmi [21]. Pomocí vhodných technik (bootstrapping, stratifikace) a strategií nastavení parametrů pro lineární prahování bylo ukázáno [21,18,19] jak se dají generované vzory optimalizovat. Příklady statistik z generování variant českých vzorů dělení jsou na obrázcích 1, 2 a 3.

4 Proč `PATGEN` nestačí?

“The road to wisdom?
Well it’s plain and simple to express:
Err and err and err again
but less and less and less.”
—Piet Hein [7]

Program `PATGEN` má několik vážných omezení. Je to monolit strukturovaného kódu, který, ačkoli je velmi dobře dokumentován (WEB, tj. dokumentovaný Pascal), není snadné upravovat. Obsahuje radikální optimalizace, které umožnily, že se proces generování vzorů dělení vešel do paměti PDP-11, srozumitelnost a tedy možnosti úprav jsou složité.

Datové struktury `PATGEN`u jsou postaveny na osmibitový ASCII kód, přitom rozšíření na UNICODE prakticky nepřichází v úvahu. Maximální počet úrovní je devět. V průběhu výběru kandidátů na vzory lze současně vybírat pouze kandidáty shodných délek. Data jsou interně ukládána do statických struktur, pokud během generování paměť dojde, je nutno zasáhnout do zdrojového kódu a rekompilovat.

Tabulka 1. Standardní generování českých vzorů s parametry Lianga

úroveň	délka	param	% dobré	% špatné	# vzorů	velikost
1	2–3	1 2 20	96,95	14,97	+ 855	
2	3–4	2 1 8	94,33	0,47	+1706	
3	4–5	1 4 7	98,28	0,56	+1033	
4	5–6	3 2 1	98,22	0,01	+2028	32 kB

Tabulka 2. Standardní generování českých vzorů s optimalizací na velikost vzorů

úroveň	délka	param	% dobré	% špatné	# vzorů	velikost
1	1–3	1 2 20	97,41	23,23	+ 605	
2	2–4	2 1 8	85,98	0,31	+ 904	
3	3–5	1 4 7	98,40	0,78	+1267	
4	4–6	3 2 1	98,26	0,01	+1665	23 kB

Tabulka 3. Standardní generování českých vzorů s optimalizací pokrytí dělicích bodů

úroveň	délka	param	% dobré	% špatné	# vzorů	velikost
1	1–3	1 5 1	95,43	6,84	+2261	
2	1–3	1 5 1	95,84	1,17	+1051	
3	2–5	1 3 1	99,69	1,24	+3255	
4	2–5	1 3 1	99,63	0,09	+1672	40 kB

Samozřejmě lze PATGEN využít pro generování vzorů pro jiné jevy než dělení slov, ovšem pouze tak, že se daný problém na dělení slov převede. To může být značně netriviální a navíc lze takto řešit pouze problémy s dostatečně malou abecedou, přibližně pod 240 symbolů. PATGEN totiž některé ASCII znaky používá jako výstupní symboly.

5 PATLIB

"My library was dukedom large enough."
—Shakespeare, The Temptest (1611), act 1, sc. 2 l. 109

Rozhodli jsme se tedy PATGEN zobecnit, implementovat knihovnu PATLIB (PATtern LIBrary) pro práci se vzory a jako její aplikaci generátor vzorů dělení v UNICODE. O názvu generátoru v době psaní článku ještě nebylo rozhodnuto.

Pro implementaci jsme zvolili méně obvyklou kombinaci v CWEBu psaného C++ z důvodů portability a efektivity a udržování kvalitní dokumentace. Navíc šablony v C++ umožňují odložit přesnou specifikaci typu na co možná nejpozději, což se během analýzy ukázalo jako velká výhoda.

Knihovna PATLIB se skládá z manipulátoru se vzory a překladové služby. Překladová služba pouze zajišťuje bijekci mezi abecedou aplikace a abecedou mani-

pulátoru. Snahou je minimalizovat potřebnou abecedu, proto projdeme nejprve vstupní data a zjistíme, kolik symbolů je skutečně použito. Lze předpokládat, že ze všech možných znaků UNICODE jich v jednom slovníku bude použito nejvýše několik stovek.

Manipulátor pracuje s čísly z důvodu zachování rozumné efektivity. Manipulátor se vzory je v principu konečným automatem s výstupem omezeným na konečný jazyk. Poskytuje základní služby typu „vlož vzor“, „dej výstup vzoru“, „odstraň vzor“ a dále umí vydat po jednotlivých vzorech celý uložený jazyk. Výstupní informací vzoru může být libovolný objekt.

Protože mezi nejčastěji zjišťované charakteristiky kandidátů na vzory patří dvojice čísel udávající, kolikrát kandidát pracuje správně a kolikrát způsobuje chybu, implementujeme i službu zajišťující pohodlnou práci s těmito údaji.

Tím jsme oddělili sémantiku ukládaných informací od jejich reprezentace. Nezáleží tedy na tom, s jakými daty aplikace pracuje. Aplikace používající tuto knihovnu může také implementovat libovolnou strategii výběru kandidátů a ověřování jejich vhodnosti.

Samozřejmě za větší obecnost a flexibilitu platíme snížením výkonu. Zatímco třeba výstupem vzoru `PATGEN`u je odkaz do hashovací tabulky obsahující dvojici <číslo úrovně, pozice>, my musíme mít jako výstupní abecedu objekt s kopírovacím konstruktorem. V této fázi projektu nelze odhadnout, k jaké ztrátě výkonu v poměru k `PATGEN`u dojde.

6 Zhuštěné digitální vyhledávací stromy (packed trie)

Datová struktura trie, kterou používáme pro ukládání vzorů, je poměrně známá. Bohužel se v klasických učebnicích programování zřídkakdy vyskytuje její prakticky použitelná varianta, a proto ji zde popíšeme.

Klasické trie, jak se s ním seznámí posluchači prvních semestrů informatiky a jak je popsáno v [12], vypadá následovně. Trie je m-ární strom, jeho uzly jsou m-prvkové vektory indexované prvky konečné abecedy. Uzel v hloubce l od kořene stromu odpovídá prefixu délky l. Zjištění, zda slovo je v trie, začíná prohledáváním od kořene. Vezme se další symbol hledaného slova, nechť je to k. Pak k-tá složka uzlu, ve kterém se právě nacházíme, ukazuje na uzel nižší vrstvy, který odpovídá dosud nepřečtenému zbytku slova. Není-li hledané slovo v trie, najdeme alespoň nejdelší shodu.

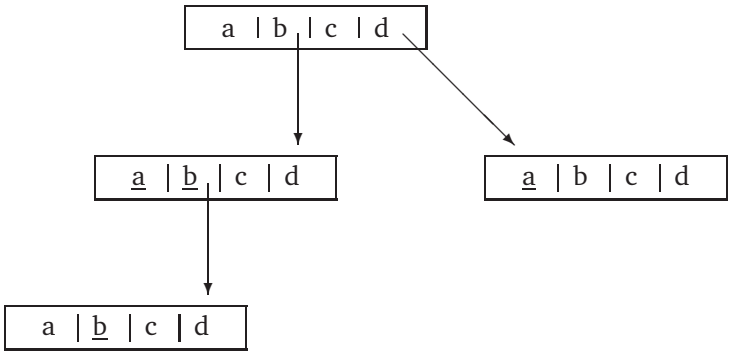
Obrázek 1 ukazuje trie obsahující tato slova ba, bb, bbb, da nad abecedou $\{a, b, c, d\}$. Podtržení označuje konec slova.

Není složité naprogramovat tuto datovou strukturu. Jednotlivé uzly lze za sebe ukládat do pole, ukazatelem je index začátku následujícího uzlu. Je to jen nesmírně paměťově neefektivní, pokud jsou uložená slova delší a pokud nejsou uzly příliš zaplněny. Ani použití dynamické paměti nepomůže.

Výhodou trie je prohledávání a vkládání v čase lineárním *k délce vkládaného slova*, tedy čas nezávisí na množství uložených slov.

Paměťové nároky této struktury lze za cenu minimálních časových ztrát podstatně zredukovat, jak ukázal Liang v [14]. V praktických aplikacích jsou

Obrázek 1. Trie



uzly trie obsazeny poměrně řídce, takže je lze ukládat do lineárního pole *mezi sebe*. Tedy jeden uzel zabírá místa ukazatelů, které další uzel nevyužívá.

Když v takovém stromě prohledáváme, musíme mít způsob, jak rozhodnout, ke kterému uzlu trie dané ukazatele patří. To lze následujícím trikem. Ke každému ukazateli přidáme informaci o tom, který symbol abecedy je s ním spojen. Ukazatel patří určitému uzlu, právě když tento symbol odpovídá pozici od začátku uzlu. Navíc dva uzly nesmí začínat na stejné pozici v poli. Přidáme tedy informaci o tom, zda daná pozice je *bázová* a při vkládání nikdy neuložíme dva uzly na stejnou bázovou pozici.

Na obrázku 2 je uložen jazyk z předchozího příkladu. Strom začíná na pozici 1, tato pozice je bázová. Kořen stromu má z implementačních důvodů poněkud výsadní postavení, je vždy v poli uložen celý, i když neexistuje slovo začínající příslušným znakem. Pouze ukazatel je v takovém případě nastaven na prázdnou hodnotu.

Jak tedy poznáme položky náležející danému uzlu? Prvkům abecedy jsme přiřadili hodnoty $a = 1$, $b = 2$, $c = 3$, $d = 4$. Nechť uzel začíná na pozici z. Projdeme pozice $z + a, \ldots, z + d$ a kontrolujeme, na kterých z nich platí, že znak na pozici $z + i$ je i. Pro kořen je to splněno vždy. Z kořene existuje ukazatel pod znakem b (na pozici 3), ukazuje na uzel začínající na pozici 5. Dále kořen říká, že máme slovo začínající znakem d. Projděme nyní pozice příslušející uzlu 5, tedy odpovídající prefixu b. Jsou to

- pozice 6 příslušející znaku a, znak souhlasí, ukazatel není nastaven, je nastaven příznak konce slova, tedy slovo *ba* patří do jazyka a žádné delší slovo začínající *ba* nepatří
- pozice 7 příslušející znaku b, znak je b, tedy pozice patří tomuto uzlu, pozice je označena jako koncová, tedy slovo *bb* patří do jazyka a další slovo začínající *bb* pokračuje uzlem na bázi 6
- pozice 8 a 9 příslušející znakům c a d, znaky nesouhlasí

Čtenář si nyní již lehce ověří, že tato tabulka obsahuje týž jazyk jako obrázek 1. K jednoduchému uložení tohoto jazyka bychom potřebovali šestnáct položek, pro zahuštěné uložení postačuje devět. To samozřejmě není reprezentativní poměr, neboť je závislý na uloženém jazyce.

Obrázek 2. Zahuštěné trie

Index	1	2	3	4	5	6	7	8	9
Znak		a	b	c	d	a	b	b	a
Ukazatel			5		8		6		
Bázová?	A				A	A		A	
Koncová?						A	A	A	A

Uzly trie lze zahušťovat first-fit algoritmem. To značí, že při ukládání uzlu najdeme první pozici, na kterou se vejde, aby se nepřekrýval s existujícím uzlem a nepoužil stejnou bázovou pozici. Můžeme použít následující heuristiku. Je-li uzel, který chceme vložit, zaplněn méně, než předem daná konstanta, použijeme first-fit. Je-li zaplněn více, nebudeme se zdržovat a vložíme ho na začátek dosud nepoužité oblasti. Praktické zkušenosti ukazují, že většinu nepoužitých prvků pole se podaří zaplnit.

7 Aplikace techniky vzorů u následníků TEXu

“But at least I can point out a minor weakness of TEX’s algorithm: all possible hyphenations have the same penalty. This might be ok for english, but for languages like German that have a lot of composite words there should be the ability to assign lower penalties between parts of a composite i.e. Um-brechen should be favored against Umbre-chen.”
—Florian Hars [6]

7.1 Dělení slov

Připomeňme si, kdy TEX slova dělí. V následujícím popisu zanedbáváme detaily, které nejsou podstatné pro následující úvahy. Pro přesný popis odkazujeme na [11], nebo články [18,19].

Algoritmus zlomu odstavce má nejvýše tři průchody. V prvním průchodu se slova nedělí a hledá se řešení, při kterém mají všechny řádky hodnotu badness menší nebo rovnu `\pretolerance`. Pokud takové řešení neexistuje, nastupuje druhý průchod.

Ve druhém průchodu se do slov odstavce přidají symboly pro možné dělicí body. Pak se vyhodnocují všechny zlomy, pro něž platí, že všechny řádky mají badness nejvýše `\tolerance`. Pokud neexistuje přijatelné řešení, ve třetím průchodu se navíc povolí roztažitelné mezislovní mezery.

Tento algoritmus má však nepříjemné omezení pro jazyky, v nich jsou častá složená slova. Švy složených slov jsou typografy považována za místa pro dělení vhodnější, ostatní místa jsou vhodná méně. Bohužel TEX nedovoluje klasifikovat dělicí body, pro libovolné místo vybrané jako možný dělicí bod ve druhém průchodu algoritmu zlomu odstavce má pouze jedinou možnou `\hyphenpenalty`.

Možným řešením by bylo vytvoření dvojice vzorů, jedna sada vzorů pro švy složených slov, druhá pro všechny dělící body. Navrhujeme zavedení penalty za dělení slova na švu, `\compoundhyphenpenalty`. Ta by byla menší než `\hyphenpenalty` a tím by bylo preferováno dělení na švech slov. To samozřejmě vyžaduje zásah do algoritmu zlomu odstavce, tedy se může týkat pouze některého z následníků TEXu.

Rozšíření o tuto klasifikaci dělení přináší otázku, kdy a jak je během zlomu odstavce provádět. Jednou možností je nahradit současný průchod, kdy se dělení provádí, průchodem, ve kterém se zjistí dělicí místa na švech slov a nastaví se jim `\compoundhyphenpenalty`. Dále se zjistí ostatní vhodné dělicí body a těm, které ještě nemají nastavenou penaltu (tj. nejsou na švu slova), se nastaví hodnota `\hyphenpenalty`.

Jinou možností je místo současného průchodu s dělením slov implementovat průchody dva. V prvním by se provedlo pouze dělení na švech slov a pokud by zlom odstavce byl dostatečně kvalitní, ponechal by se. Pokud ne, provedlo by se i dělení v dalších možných místech (s `\hyphenpenalty`) a odstavec by se lámal znovu.

Navíc znalost švů složených slov je výhodná i z dalšího důvodu. Typografická pravidla požadují, aby se na švech nevyskytovaly ligatury. Tedy například slovo šéflékař má být správně vysázeno šéflékař. Bohužel to je nutno TEXu sdělit ručně, proto jsem poslední slovo předchozí věty musel psát `šéf\hskip0ptlékař`.

Bylo by vhodné, aby se všechna slova ze vstupního proudu otestovala na švy složených slov a vstupní proud se příslušně upravil. To samozřejmě přináší další otázky, jako například, zda nevypustit první průchod zlomu odstavce a nezačít rovnou s povoleným dělením na švech slov.

7.2 Překladové procesy Ω

Systém Ω umožňuje téměř v jakémkoliv okamžiku zpracování textu ve svém zažívacím traktu aplikovat *překladový proces* (ΩTP, Omega Translation Process). Ten je dosud implementován pomocí, zjednodušeně řečeno, substitucí regulárních výrazů. Tato realizace dostačuje u jednoduchých zobrazení, pro složitější (například dělení slov) není dostatečně efektivní. Proto se pro složitá mapování (dělení slov, aplikace ligatur v daném jazyce, spelling či dokonce grammar checker, kontextové zpracování znaků v arabštině) nabízí pro generování a uložení těchto informací použití techniky vzorů. Knihovna `PATLIB` by pak hrála v efektivní implementaci těchto nástrojů klíčovou roli.

8 Shrnutí

'I když se neprosadím, chtěl bych věřit, že bude někdo pokračovat v tom, co jsem započal. Ne bezprostředně, ale člověk není sám ve víře v opatrnost.'
—Vincent van Gogh

V článku jsme popsali techniku vzorů, možnosti jejího využití a návrh knihovny pro práci se vzory.

Zdrojové kódy knihovny `PATLIB` v její aktuální vývojové verzi naleznete na adrese `http://www.fi.muni.cz/~xantos/PATLIB`. Po dokončení knihovny doufáme v její využití v širokém spektru aplikací, od implementace `PATGEN`u, přes implementaci překladových procesů sázecího systému Ω, po aplikace v oblastech zpracování přirozeného jazyka či grafiky.

Reference

1. Cezar Câmpeanu, Nicolae Sânteau, and Sheng Yu. Minimal cover-automata for finite languages. In Champarnaud et al. [2], pages 43–56.
2. Jean-Marc Champarnaud, Denis Maurel, and Djelloul Ziadi, editors. *Automata Implementation, Third International Workshop on Implementing Automata, WIA'98*, Berlin, Heidelberg, 1999. Springer-Verlag.
3. Patrick Hanks, editor. *The New Oxford Dictionary of English*. Oxford University Press, Oxford, 1998.
4. Yannis Haralambous. A Small Tutorial on the Multilingual Features of PATGEN2. in electronic form, available from CTAN as `info/patgen2.tutorial`, January 1994.
5. Yannis Haralambous and John Plaice. Methods for Processing Languages with Omega. In *Proceedings of the Second International Symposium on Multilingual Information Processing, Tsukuba, Japan*, 1997. available as `http://genepi.louis-jean.com/omega/tsukuba-methods97.pdf`.
6. Florian Hars. Typo-l email discussion list, 4 January 1999.
7. Piet Hein. *Grooks*. MIT Press, Cambridge, Massachusetts, 1966.
8. Douglas R. Hofstadter. *Godel, Escher, Bach: An Eternal Golden Braid*. Basic Books, 1979.
9. Tao Jiang, Arto Salomaa, Kai Salomaa, and Sheng Yu. Decision problems for patterns. *Journal of Computer and Systems Sciences*, 50(1):53–63, 1995.
10. Lauri Karttunen, Tamás Gaál, and André Kempe. Xerox finite-state tool. Technical report, Xerox research Centre Europe, Grenoble, June 1997. `http://www.xrce.xerox.com/research/mltt/fssoft/docs/fst-97/xfst97.html`.
11. Donald E. Knuth. *The TEXbook*, volume A of *Computers and Typesetting*. Addison-Wesley, Reading, MA, USA, 1986.
12. Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, third edition, 1998.
13. András Kornai. *Extended Finite State Models of Language*. Cambridge University Press, 1999.
14. Franklin M. Liang. *Word Hy-phen-a-tion by Com-put-er*. Ph.D. Thesis, Department of Computer Science, Stanford University, August 1983.
15. Franklin M. Liang and Peter Breitenlohner. `PAT`tern `GEN`eration program for the TEX82 hyphenator. Electronic documentation of `PATGEN` program version 2.3 from web2c distribution on CTAN, 1999.

16. Mehryar Mohri, Fernando C.N. Pereira, and Michael D. Riley. FSM Library – General-purpose finite-state machine software tools, 1998. `http://www.research.att.com/sw/tools/fsm/`.
17. Emmanuel Roche and Yves Schabes. *Finite-State Language Processing*. MIT Press, 1997.
18. Petr Sojka. Notes on Compound Word Hyphenation in TEX. *TUGboat*, 16(3):290–297, 1995.
19. Petr Sojka. Hyphenation on Demand. *TUGboat*, 20(3):241–247, 1999.
20. Petr Sojka. Competing Patterns for Language Engineering. LNAI 1902, pages 157–162, Brno, Czech Republic, Sep 2000. Springer-Verlag.
21. Petr Sojka and Pavel Ševeček. Hyphenation in TEX – Quo Vadis? *TUGboat*, 16(3):280–289, 1995.
22. Alan Turing. Computing machinery and intelligence. *Mind*, (59):433–460, 1950.