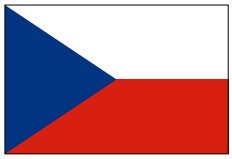


Vysoká škola báňská – Technická univerzita Ostrava

Teoretická informatika

učební text

Petr Jančar
Martin Kot
Zdeněk Sawa

Ostrava 2007

Recenze: Doc. RNDr. Jaroslav Markl

Název: Teoretická informatika – učební text
Autoři: Doc. RNDr Petr Jančar, CSc., Ing. Martin Kot, Ing. Zdeněk Sawa, PhD
Vydání: první, 2007
Počet stran: 269
Náklad: xx
Vydavatel a tisk: Ediční středisko VŠB-TUO

Studijní materiály pro studijní obor Informatika a výpočetní technika fakulty elektrotechniky a informatiky
Jazyková korektura: nebyla provedena

Určeno pro projekt:
Operační program Rozvoj lidských zdrojů
Název: E-learningové prvky pro podporu výuky odborných a technických předmětů
Číslo: CZ.04.01.3/3.2.15.2/0326
Realizace: VŠB – Technická univerzita Ostrava
Projekt je spolufinancován z prostředků ESF a státního rozpočtu ČR

ISBN xxxx

Obsah

Úvod

Poznámka z 2.6.2007: (P. Jančar)

Tato verze textu obsahuje přepracovanou (zásadní) podčást verze z 20.2.2007. (Tam lze nalézt i kapitolu „Základní definice“, která částečně sumarizuje předpokládané matematické pojmy, ale není přímou součástí našeho kursu.)

Tato verze z 2.6. 2007 má sloužit studentům Teoretické informatiky pro přípravu na zkoušku. Zkouška v letním semestru 2006/07 nebude zasahovat do jiných partií než zde uvedených. Tento materiál, spolu s animacemi odkazovanými na web-stránce předmětu, by měl postačovat k přípravě i studentům kombinované formy studia. (Max. 10% obsahu zkoušky se může odkazovat na pochopení referátů, které studenti denního studia prezentovali v rámci cvičení.)

Tato verze se už v průběhu zkouškového období nebude měnit, pokud bude objeven zásadnější nedostatek, bude to sděleno na web-stránce (u odkazu na tento text).

Studentům předmětu Úvod do teoretické informatiky (bak. studium) sdělí ještě případný komentář k tomuto textu vyučující.

Následuje původní nepřepracovaný úvod, který ovšem není v dané chvíli podstatný.

==========

Poznámka autorů určená recenzentům. (z února 2007) Tento učební text byl dopsán a zkompletován před začátkem běhu kursu v letním semestru 2006/2007. Jsme si vědomi, že v něm jsou ještě mnohé nedostatky (byť doufáme, že ne zásadního rázu). V průběhu semestru hodláme celý text postupně revidovat (souběžně s výukou příslušných partií); revidované části budou rovněž zpřístupněny studujícím, bude-li to třeba. Samozřejmě hodláme při revizi

vzít v potaz i kritické připomínky recenzentů.

==========

Předkládaný materiál slouží jako studijní text pro předměty teoretické informatiky, speciálně pro oblasti teorie jazyků a automatů a teorie algoritmů (tj. teorie vyčíslitelnosti a složitosti). Text existuje ve dvou verzích. Základní verze je určena pro kurs „Úvod do teoretické informatiky“, rozšířená verze pak pro kurs „Teoretická informatika“. Rozšířená verze obsahuje veškerý materiál verze základní a navíc má části označené jako pokročilé; tyto části se vyskytují v rámci jednotlivých kapitol či jako celé kapitoly.

Souhrnný název studijního textu by také mohl být *Základy teorie výpočtů* (Theory of Computation). Tato teorie patří k základním (a dnes již klasickým) partiím teoretické informatiky, partiím, jejichž vznik a vývoj byl a je úzce svázán s potřebami praxe při vývoji software, hardware a obecně při modelování systémů. Motivovat teorii výpočtů lze přirozenými otázkami typu:

- jak srovnat kvalitu (rychlost) různých algoritmů řešících tentýž problém (úkol)?
- jak lze porovnávat (klasifikovat) problémy podle jejich (vnitřní) složitosti?
- jak charakterizovat problémy, které jsou a které nejsou algoritmicky řešitelné, tj. které lze a které nelze řešit algoritmy (speciálně „rychlými“, neboli prakticky použitelnými, algoritmy).

Při zpřesňování těchto a podobných otázek, a při hledání odpovědí, nutně potřebujeme (abstraktní) modely počítače (tj. toho, kdo provádí výpočty). Z více důvodů je vhodné při našem zkoumání začít velmi jednoduchým, ale fundamentálním modelem, a sice tzv. *konečnými (tj. konečně stavovými) automaty*. Konečné automaty (pojem byl formalizován ve 40. letech 20. století) lze chápat nejen jako nejzákladnější model v oblasti počítačů, ale ve všech oblastech, kde jde o modelování systémů, procesů, organismů apod., u nichž lze vyčlenit konečně mnoho stavů a popsat způsob, jak se aktuální stav mění prováděním určitých akcí (např. přijímáním vnějších impulsů). (Jako jednoduchý ilustrující příklad nám může posloužit model ovladače dveří v supermarketu znázorněný na obr. 2.1, o němž pojednáme později.)

Velmi běžná „výpočetní“ aplikace, u níž je v pozadí konečný automat, je *hledání vzorků v textu*. Takové hledání asi nejčastěji používáme v textových editorech a při vyhledávání na Internetu; speciální případ také představuje např. *lexikální analýza* v překladačích. Při vyhledávání informací v počítačových systémech jste již jistě narazili na nějakou variantu *regulárních výrazů*, umožňujících specifikovat celé třídy vzorků. Regulárními výrazy a jejich vztahem ke konečným automatům se rovněž budeme zabývat.

Po seznámení se s konečnými automaty a regulárními výrazy budeme pokračovat silnějším modelem – tzv. *zásobníkovými automaty*; o ty se opírají algoritmy *syntaktické analýzy* při překladu (programovacích) jazyků, tedy algoritmy, které např. určí, zda vámi napsaný program v Javě je správně „javovsky“.

Zmínili jsme pojem *jazyk* – obecně budeme mít na mysli tzv. formální jazyk; jazyky přirozené (mluvené) či jazyky programovací jsou speciálními případy. Na naše modely se v prvé řadě budeme dívat jako na *rozpoznávače jazyků*, tj. zařízení, která zpracují vstupní posloupnost písmen (symbolů) a rozhodnou, zda tato posloupnost je (správně utvořenou) větou příslušného jazyka.

S pojmem *jazyk* se nám přirozeně pojí pojem *gramatika*. Speciálně se budeme věnovat tzv. *bezkontextovým gramatikám*, s nimiž jste se již přinejmenším implicitně setkali u definic syntaxe programovacích jazyků (tj. pravidel konstrukce programů); ukážeme mimo jiné, že bezkontextové gramatiky generují právě ty jazyky, jež jsou rozpoznávány zásobníkovými automaty.

Seznámíme se také s univerzálními modely počítačů (algoritmů) – konkrétně s *Turingovými stroji* a *stroji RAM*. Na těchto modelech postavíme vysvětlení pojmů *rozhodnutelnosti* a *nerozhodnutelnosti* problémů a podrobněji se budeme zabývat *výpočetní složitosti algoritmů a problémů*; speciálně pak třídami složitosti PTIME a NPTIME.

Rozšířenou verzi zakončíme úvodem do problematiky *aproximačních, pravděpodobnostních, paralelních a distribuovaných algoritmů*.

Poznamenejme, že účelem kursu není popis konkrétních větších reálných aplikací studovaných (teoretických) pojmů; cílem je základní seznámení se s těmito pojmy a s příslušnými obecnými výsledky a metodami. Jejich zvládnutí je nezbytným základem pro porozumění i oněm reálným aplikacím a pro jejich návrh. Jako pěkný příklad relativně nedávné aplikace může sloužit použití konečných automatů s vahami pro reprezentaci složitých funkcí (na reálném

oboru) a jejich využití při reprezentaci, transformaci a kompresi obrazové informace (viz např. kapitolu v [Gru97]).

Velmi přínosná by samozřejmě byla snaha studujícího prostudovat probírané partie také v některé z doporučených (či jiných) monografií. V češtině či slovenštině vyšly např. [Chy84], [HU78] (což je slovenský překlad angl. originálu z r. 1969), [Kuč83], [MvM87]. Kromě uvedených knih existují jistě i další texty v češtině či slovenštině, které se zabývají podobnou problematikou. Nepoměrně bohatší je ovšem nabídka příslušné literatury v angličtině, což lze snadno zjistit např. „surfováním“ na Webu. Uveďme např. alespoň [Sip97].

Pokyny ke studiu

Jak jsme již zmínili, tento text existuje ve dvou verzích – základní a rozšířené. Základní verze textu je primárně určena pro předmět „Úvod do teoretické informatiky“, zatímco rozšířená pro předmět „Teoretická informatika“ a pro studenty Úvodu do teoretické informatiky s hlubším zájmem o danou problematiku. Rozšířená verze se od základní verze odlišuje v následujících ohledech:

- Obsahuje oproti základní verzi několik dalších kapitol. U názvů těchto kapitol je uvedeno, že patří do pokročilé části.
- Některé kapitoly, které se nacházejí i v základní části jsou rozšířeny o pokročilou část. Začátek této části je označen nadpisem
  Pokročilé partie
- Do textu, který je i v základní verzi, jsou na některých místech přidány rozšiřující poznámky, podrobnější důkazy apod. Tyto (menší) části jsou označeny textem „*Pro pokročilé:*“.

Text je členěn do kapitol podle jednotlivých témat. Kapitola **??** shrnuje základní definice, kterým je třeba rozumět pro studium následujícího textu a které by čtenář již měl znát z dřívějšího studia. Tato kapitola slouží také k upřesnění a shrnutí matematické notace používané v textu.

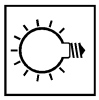
Každá kapitola začíná uvedením cílů dané kapitoly, které stručně shrnují, jaké znalosti by měl čtenář získat po prostudování dané kapitoly. Cíle jsou vždy uvedeny následující ikonkou a nadpisem:

Cíle kapitoly:

- Zde budou uvedeny cíle dané kapitoly.

V případě, že rozšiřující část kapitoly obsahuje témata, která nejsou obsažena v základní části, jsou cíle uvedeny rovněž na začátku rozšiřující části.

Součástí textu jsou řešené příklady, které podrobně ukazují, jak řešit vybrané typy příkladů. Tyto příklady jsou vždy uvedeny následující ikonkou a textem:



Řešený příklad: Zde je uvedeno zadání příkladu.

Řešení: Zde pak následuje ukázkové řešení.

Další součástí textu jsou otázky a cvičení, které by měly čtenáři sloužit k tomu, aby ověřil nabyté znalosti. Rozdíl mezi otázkami a cvičeními je ten, že na otázky by měl být čtenář schopen odpovědět hned či po krátkém zamyšlení bez nutnosti něco řešit. Naproti tomu vyřešení cvičení bude většinou vyžadovat použití tužky a papíru.

Otázky, které jsou roztroušeny v textu a vztahují se přímo k právě diskutované problematice, označujeme jako „kontrolní otázky“ a jsou označeny takto:

Kontrolní otázka: Zde bude uveden text otázky.

Některé otázky jsou shrnuty do bloku otázek na konci příslušné kapitoly či sekce. Tento blok je označen stejnou ikonkou jako kontrolní otázka:

Otázky:

Otázka: Text první otázky.

Otázka: Text další otázky.

Cvičení jsou označena následujícím způsobem (pokud následuje více cvičení za sebou, je ikonka uvedena jen u prvního z nich):

Cvičení: Zde bude uveden text zadání.

Některé kapitoly obsahují samostatnou sekci nazvanou „Cvičení“. Tato sekce obsahuje další příklady k dokonalejšímu procvičení probírané látky. Těžší příklady jsou označeny hvězdičkou (∗), ještě těžší dvěma hvězdičkami (∗∗).

Otázky a cvičení jsou číslovány. Na konci textu jsou pak v Příloze A uvedena řešení většiny z nich.

Kapitola 1

Formální jazyky

Cíle kapitoly:

Po prostudování kapitoly máte plně rozumět pojmům jako (formální) abeceda, slovo, jazyk, operace na slovech a jazycích; máte zvládat práci s těmito pojmy na praktických příkladech.

Klíčová slova: *abeceda, slovo, jazyk, operace na jazycích*

Komentář: Kapitola má dvě výukové části a jednu procvičovací. U každé výukové části jsou uvedeny podrobnější cíle, klíčová slova, orientační čas ke studiu a na závěr shrnutí.

1.1 Formální abeceda a jazyk

Orientační čas ke studiu této části: 1 hod.

Cíle této části:

Po prostudování této části máte rozumět pojmům jako je formální slovo, abeceda, jazyk. Máte je být schopni vysvětlit, uvádět příklady, rozumět popisům jazyků jako množin slov charakterizovaných nějakou podmínkou. Také máte zvládnout elementární pojmy a operace jako je délka slova, prefix, sufix, podslovo, zřetězení slov atd.

Klíčová slova: *abeceda, slovo, znak, jazyk, zřetězení, prefix, sufix, podslovo*

Teoretická informatika poskytuje formální základy a nástroje pro praktické informatické aplikace (jako programování či softwarové inženýrství). Jedním z jejích důležitých úkolů je matematicky popsat různé typy algoritmických problémů a výpočtů. Pro matematický popis vstupů a výstupů problémů (výpočtů) je užitečné nejprve zavést pojmy jako jsou (formální) abeceda, slovo, jazyk.

Použitá symbolická abeceda pro vstupy a výstupy výpočtů závisí na dohodnuté formě zápisu. V počítačové praxi využíváme např. binární abecedu $\{0, 1\}$, hexadecimální abecedu $\{0, 1, \ldots, 9, A, \ldots, F\}$ nebo „textovou“ abecedu, např. v kódování ASCII či nověji UTF-8. Matematicky můžeme za abecedu považovat libovolnou (dohodnutou) konečnou množinu symbolů; převody zápisů mezi různými abecedami jsou přímočaré. (V konkrétním případě obvykle volíme abecedu, která se přirozeně hodí k danému problému.)

Důležitým pojmem je (formální) „slovo“, což znamená libovolný konečný řetězec symbolů nad danou abecedou; pokud je v abecedě mezera, nemá žádný zvláštní význam. (Jakkoli vymezená) množina slov se nazývá (formálním) „jazykem“. Jako příklady slov v abecedě $\{0, 1\}$ můžeme uvést třeba slovo 00110101100 či slovo 10001. Příkladem jazyka s abecedou $\{0, 1\}$ je třeba množina všech slov (v abecedě $\{0, 1\}$), která obsahují sudý počet znaků 0 (správněji řečeno: sudý počet *výskytů* znaku 0); první výše uvedené slovo do tohoto jazyka patří, druhé nikoliv. Všimněme si také, že tento jazyk je nekonečný a nemohli bychom ho tedy zadat výčtem jeho prvků.

Uvedené pojmy nyní přesně nadefinujeme a zároveň zavedeme důležité operace se slovy a jazyky.

Definice 1.1
Abecedou myslíme libovolnou konečnou množinu; často ji označujeme Σ. Prvky abecedy nazýváme *symboly* (či *písmena*, *znaky* apod.).
(Např. abeceda $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ obsahuje dvě písmena.)
Slovem, neboli *řetězcem*, nad abecedou Σ (též říkáme: v abecedě Σ) rozumíme libovolnou konečnou posloupnost prvků množiny Σ. Pro $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ je to například `a,b,b,a,b`; pokud nemůže dojít k nedorozumění, píšeme takovou posloupnost obvykle bez čárek, jako `abbab`.
Prázdné slovo „“ je také slovem a značí se ε.

Důležitá poznámka k značení.

V konkrétních příkladech budeme typicky používat abecedy jako $\{\mathtt{a}, \mathtt{b}\}$, $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$, $\{0, 1\}$ apod. Často ovšem budeme hovořit o obecné abecedě Σ a budeme třeba popisovat nějakou konstrukci, která se má provést pro každé písmeno ze Σ. Řekneme tedy např.:

$$\text{pro každé } a \in \Sigma \text{ provedeme následující } \ldots$$

To neznamená, že fyzický symbol a je prvkem Σ. V této souvislosti a prostě představuje proměnnou, kterou používáme při našem popisu situace. Když tedy např. příkaz

$$\text{postupně pro každé } a \in \Sigma \text{ vypiš } aa$$

aplikujeme na abecedu $\Sigma = \{0, 1\}$, je příslušným výpisem 0011. Ideální by bylo, kdybychom typograficky odlišovali a používali např. '$\mathtt{a}$' jen jako prvek konkrétní abecedy a 'a' jen jako onu proměnnou. Upozorňujeme na to, že náš text to nedodržuje (často používáme abecedu $\Sigma = \{a, b\}$, čili používáme 'a' i pro prvek konkrétní abecedy); na druhé straně by měl být význam konkrétního použití symbolu 'a' vždy jasný z kontextu).

Ve smyslu proměnných budou malá písmena ze začátku anglické abecedy $(a, b, c, \ldots)$ s případnými indexy představovat znaky zkoumané abecedy (která bude v kontextu zřejmá či tiše předpokládaná). Jako proměnné pro slova budeme obvykle používat malá písmena z konce abecedy (u, v, w, x, y, z).

Ilustrujme si toto použití proměnných např. u zavedení následujícího značení.

Značení: Délku slova w, tj. počet písmen ve w, značíme $|w|$; slovo ε má pochopitelně délku 0, tedy $|\varepsilon| = 0$.
Výrazem $|w|_a$ označujeme počet výskytů symbolu a ve slově w.

Symbol w v předchozí úmluvě je tedy proměnná, za niž můžeme dosadit libovolné slovo (v jakékoli zvolené abecedě). Konkrétně z toho plyne např. $|00110| = 5$. Ve výrazu $|w|_a$ se vyskytují dvě proměnné; za w tak můžeme dosadit libovolné slovo ve zvolené abecedě a za a libovolný prvek této abecedy. Z tohoto obecného popisu je nám tak jasné, že v konkrétním případě je např. $|00110|_1 = 2$.

Značení: Výrazem Σ^* značíme *množinu všech slov* nad abecedou Σ; někdy použijeme Σ^+ pro *množinu všech neprázdných slov* v abecedě Σ. (Je tedy $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.)

Množina všech slov nad konečnou abecedou je spočetná; slova v dané abecedě můžeme totiž přirozeně seřadit (uspořádat): nejprve podle délky a v rámci stejné délky podle abecedy, tj. podle zvoleného uspořádání na prvcích abecedy. Tak jsou slova seřazena do jedné posloupnosti, ve které je lze po řadě očíslovat přirozenými čísly.

Kontrolní otázka: Jak byste v tomto pořadí vypisovali (generovali) slova z abecedy $\Sigma = \{0, 1\}$ s abecedním uspořádáním $0 < 1$?

Jistě jste nezapomněli na prázdné slovo, a začali jste tedy posloupnost $\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \ldots$.

Značení: Příslušné uspořádání slov budeme označovat $<_L$ (např. $11 <_L 001$).

Přirozenou operací se slovy je jejich zřetězení, tj. jejich spojení za sebou do jednoho výsledného slova:

Definice 1.2
Zřetězení slov $u = a_1a_2 \ldots a_n$, $v = b_1b_2 \ldots b_m$ označujeme $u \cdot v$, stručněji uv, a definujeme $uv = a_1a_2 \ldots a_nb_1b_2 \ldots b_m$. Výrazem u^n označujeme n-násobné zřetězení slova u; tedy $u^0 = \varepsilon$, $u^1 = u$, $u^2 = uu$, $u^3 = uuu$ atd.

Poznámka: Uvědomme si, že operace zřetězení slov je asociativní (tzn. $(u \cdot v) \cdot w = u \cdot (v \cdot w)$); proto je např. zápis $u \cdot v \cdot w$ (či uvw) jednoznačný i bez uvedení závorek.

Je také přirozené se dohodnout, že

exponent váže silněji (má větší prioritu) než zřetězení.

Pak je jasné, že např. zápisem a^3bc^4a myslíme slovo $aaabcccca$. Chceme-li, aby se zde např. exponent 4 vztahoval ke slovu bc, musíme použít závorky: $a^3(bc)^4a$ znamená slovo $aaabcbcbcbca$.

Někdy potřebujeme mluvit jen o určitých částech slova. Úsek znaků, kterým nějaké slovo začíná, budeme nazývat předponou, neboli odborně prefixem.

Obdobně se úsek znaků, kterým slovo končí, budeme nazývat příponou neboli sufixem. Jakoukoliv část slova budeme nazývat podslovem (nebo podřetězcem).

Definice 1.3

- Slovo u je *prefixem* slova w, pokud lze psát $w = uv$ pro nějaké slovo v.
- Slovo u je *sufixem* slova w, pokud lze psát $w = vu$ pro nějaké slovo v.
- Slovo u je *podslovem* slova w, pokud lze psát $w = v_1uv_2$ pro nějaká slova v_1 a v_2.

Všimněme si, že podslovo u může mít ve w několik *výskytů*; každý výskyt je určen svou pozicí, tj. délkou příslušného v_1 zvětšenou o 1. Dává to smysl i pro podslovo $u = \varepsilon$, byť v tomto případě asi jednotlivé 'výskyty' nikdy nebudeme uvažovat. (Poznamenejme ještě, že konkrétní prefix či sufix u má samozřejmě jen jeden 'výskyt' ve w.)

Příklad: Vezměme si například slovo „abcadbcdc". Pak slovo „abc" je jedním z jeho prefixů, kdežto „bc" prefixem není. Dále „bcdc" je jedním z jeho sufixů. Slovo „bc" je podslovem uvedeného slova, s dvěma výskyty – na pozicích 2 a 6; není ale prefixem ani sufixem.

Prefixů slova w je očividně $|w|+1$; stejně je to s počtem sufixů. Každý prefix i každý sufix daného slova je i jeho podslovem. Prázdné slovo ε je pochopitelně prefixem, sufixem i podslovem každého slova.

Kontrolní otázka: Kolik je podslov slova w ?

To je komplikovanější otázka; počet nezávisí jen na délce slova w, např. slovo aaa má jen jedno podslovo délky 1 (s třemi výskyty), kdežto aba má dvě různá podslova délky 1. Podslov slova w je určitě alespoň $|w| + 1$ a jistě ne více než $|w|^2 + 1$; horní hranici ovšem jistě můžete snížit. Jiná věc je počet *výskytů* daného podslova ve slově; např. slovo aaa má tři výskyty podslova a a dva výskyty podslova aa.

Cvičení 1.1:

a) Vypište všechna slova v abecedě $\{a, b\}$, která mají délku 3.

b) Napište explicitně slovo u (posloupnost písmen), které je určeno výrazem $v^3 \cdot ba \cdot (bba)^2$, kde $v = ab$ (slovo u je tedy výsledkem provedení operací uvedených ve výrazu).

c) Vypište všechna slova délky 2, které jsou podslovy slova 00010 (v abecedě $\{0, 1\}$).

d) Vypište všech pět prefixů slova 0010.

e) Vypište všech pět sufixů slova 0010.

Definice 1.4
Formální jazyk, stručně *jazyk* nad abecedou Σ je libovolná množina slov v abecedě Σ, tedy libovolná podmnožina Σ^*.

Značení: Jazyky obvykle označujeme L (s indexy). Říkáme-li pouze „jazyk“, rozumíme tím, že příslušná abeceda je buď zřejmá z kontextu nebo může být libovolná.

Poznámka: U přirozeného jazyka (jako je čeština) mluvíme o slovech, z nichž se skládají věty. U formálních jazyků ze slov žádné věty netvoříme, naopak samotná slova (řetězce patřící do jazyka) je možné chápat jako ‘věty’ (a někdy se tak i nazývají). Pokud se např. na jazyk ‘čeština’ díváme jako na množinu všech českých gramaticky správných vět, je každá tato věta slovem takto chápaného formálního jazyka ‘čeština’.

Poznámka: Byť v praktických případech jazyků má jejich abeceda např. desítky prvků, v našich příkladech bude abeceda často (jen) dvouprvková (většinou $\{a, b\}$ či $\{0, 1\}$). Uvědomme si, že to není zásadní omezení, jelikož písmena víceprvkové abecedy lze přirozeně zakódovat řetězci dvouprvkové abecedy.

Kontrolní otázka: Jak dlouhé řetězce z abecedy $\{0, 1\}$ byste použili při kódování abecedy, která má 256 prvků?

(Pochopitelně stačí osm bitů, tedy jeden symbol 256-ti prvkové abecedy reprezentujeme řetězcem délky 8 v dvouprvkové abecedě.)

Příklad: Příklady formálních jazyků nad abecedou $\{0, 1\}$ jsou:

- $L_1 = \{\varepsilon, 01, 0011, 1111, 000111\}$
- L_2 je množina všech (konečných) posloupností v abecedě $\{0, 1\}$ obsahujících stejný počet symbolů 0 jako 1, tedy $L_2 = \{w \in \{0, 1\}^* \mid |w|_0 = |w|_1\}$

- $L_3 = \{w \in \{0,1\}^* \mid$ číslo s binárním zápisem w je dělitelné 3$\}$

Jazyk L_1 je zde konečný, kdežto zbylé dva jsou nekonečné. Slovo 101100 patří do jazyka L_2, ale 10100 do L_2 nepatří, neboť obsahuje více nul než jedniček. Slovo 110 binárně vyjadřuje číslo 6, a proto patří do jazyka L_3, kdežto 1000 vyjadřující 8 do L_3 nepatří.

Cvičení 1.2: Vypište prvních deset slov z jazyka $L = \{w \in \{a,b\}^* \mid$ každý výskyt podslova aa je ve w ihned následován znakem $b\}$. (Pochopitelně se odkazujeme k uspořádání $<_L$, kde předpokládáme abecední uspořádání $a < b$.)

Shrnutí: Takže už chápeme, že formální jazyk je něco jiného než přirozený. Je to prostě množina slov neboli konečných řetězců písmen z nějaké konečné abecedy. Malé konečné jazyky lze zadávat výčtem, nekonečné jen vhodnou charakterizací slov jazyka podmínkou, kterou splňují. Prefixy, sufixy, podslova, zřetězení, značení délky, počtu výskytů symbolu ve slově atd. ... to vše není pro nás žádný problém.

1.2 Některé operace s jazyky

Orientační čas ke studiu této části: 1 hod.

Cíle této části:

Po prostudování této části máte rozumět běžným operacím s jazyky, nejen klasickým množinovým, ale i zřetězení, iteraci, zrcadlovému obrazu a (levému) kvocientu jazyka podle slova (a obecně podle jazyka). Máte je být schopni definovat, vysvětlit, uvádět a řešit příklady.

Klíčová slova: *operace s jazyky, sjednocení, průnik, doplněk, rozdíl, zřetězení, iterace, zrcadlový obraz, kvocient*

Někdy je výhodné definovat složitější jazyk prostřednictvím dvou jednodušších a nějaké operace, která je spojí. Protože jsou jazyky definovány jako

množiny, můžeme používat běžné množinové operace (definované v Sekci **??**). Máme tedy:

- Z jazyků L_1, L_2 lze tvořit jazyky $L_1 \cup L_2$ (sjednocení), $L_1 \cap L_2$ (průnik), $L_1 - L_2$ (rozdíl).

Nezmiňovali jsme abecedy Σ_1, Σ_2 jazyků L_1, L_2; pokud nejsou stejné, můžeme výsledný jazyk chápat jako jazyk s abecedou $\Sigma_1 \cup \Sigma_2$. Dále máme

- Pro jazyk L je jazykem i jeho doplněk $\overline{L}$; rozumí se pro příslušnou abecedu Σ, tj. $\overline{L} = \Sigma^* - L$.

Dále můžeme definovat nové operace speciálně pro práci s jazyky. Např. je to zřetězení jazyků (odvozené od zřetězení slov) či iterace (tedy opakované řetězení):

- *Zřetězení jazyků* L_1, L_2 je jazyk $L_1 \cdot L_2 = \{uv \mid u \in L_1, v \in L_2\}$, tj. jazyk všech slov, které lze rozdělit na dvě části, z nichž první je z jazyka L_1 a druhá z jazyka L_2.
- *Iterace jazyka* L, značená L^*, je jazyk všech slov, která lze rozdělit na několik částí, z nichž každá patří do jazyka L; do L^* ovšem vždy zařazujeme ε (chápané jako zřetězení 0 slov). Induktivně můžeme také definovat

  $L^0 = \{\varepsilon\}$, $L^1 = L$, $L^2 = L \cdot L$, ..., $L^{n+1} = L^n \cdot L$, ...

  Iterace L je pak rovna

  $L^* = L^0 \cup L^1 \cup L^2 \cup L^3 \cup \ldots$

Příklad: Uveďme si následující ukázky operací s jazyky nad abecedou $\{0, 1\}$; zkuste vždy uvedenou otázku nejdříve sami zodpovědět.

a) Co je sjednocením jazyka L_0 všech slov obsahujících více 0 než 1 (tedy $L_0 = \{\, w \in \{0,1\}^* \mid |w|_0 > |w|_1 \,\}$) a jazyka L_1 všech slov obsahujících více 1 než 0 (tedy $L_1 = \{\, w \in \{0,1\}^* \mid |w|_0 < |w|_1 \,\}$)?

   Je to jazyk všech slov majících počet 1 různý od počtu 0. (Tedy $L_0 \cup L_1 = \{\, w \in \{0,1\}^* \mid |w|_0 \neq |w|_1 \,\}$.)

b) Jaký jazyk $L_0 \cdot L_1$ vzniklý zřetězením jazyků z předchozí ukázky (a)? Patří sem všechna možná slova?

Všechna slova do tohoto jazyka nepatří, například snadno zjistíme, že např. $10 \notin L_0 \cdot L_1$. Nepatří tam také např. 1111001 a obecně tam jistě nepatří každé slovo, které nemá prefix, v němž je více 0 než 1. Přesné vystižení celého zřetězení není úplně jednoduché. Podle definice tam ale prostě patří všechna ta slova, v nichž existuje prefix mající více 0 než 1, přičemž zbytek slova má naopak více 1 než 0.

c) Je pravda, že $L_0 \cdot L_1 = L_1 \cdot L_0$ v předchozí ukázce?

Není, například, jak už bylo uvedeno, $10 \notin L_0 \cdot L_1$, ale snadno vidíme, že $10 \in L_1 \cdot L_0$.

d) Co vznikne iterací jazyka $L_2 = \{00, 01, 10, 11\}$?

Takto vznikne jazyk L_2^* všech slov sudé délky, včetně prázdného slova. Zdůvodnění je snadné, slova v L_2^* musí mít sudou délku, protože vznikají postupným zřetězením úseků délky 2. Naopak každé slovo sudé délky rozdělíme na úseky délky 2 a každý úsek bude mít zřejmě jeden z tvarů v L_2.

Poznámka: Všimněme si, že jsme např. na výše uvedených jazycích ukázali, že operace zřetězení jazyků není komutativní, tj. obecně neplatí $L_1 \cdot L_2 \neq L_2 \cdot L_1$. (Použili jsme sice pro označení operace zřetězení stejný znak jako užíváme pro násobení (tedy '$\cdot$'), to ale pochopitelně neznamená, že operace zřetězení má stejné vlastnosti jako násobení.)

Poznámka: Všimněme si také, že značení pro iteraci odpovídá našemu značení množiny všech slov Σ^* nad abecedou Σ — na abecedu je možné se dívat jako na množinu všech jednopísmenných slov; a každé (neprázdné) konečné slovo nad abecedou Σ lze rozdělit na části délky 1, z nichž každá pochopitelně patří do Σ.

Definice iterace L^* nám také říká, že prázdné slovo do ní patří vždy (vznikne „zřetězením nula slov z L“).

Tedy mj. platí $\emptyset^* = \{\varepsilon\}$.

Další zajímavou operací definovanou pro jazyky je zrcadlový obraz.

Definice 1.5
Zrcadlový obraz slova $u = a_1 a_2 \dots a_n$ je $u^R = a_n a_{n-1} \dots a_1$, *zrcadlový obraz jazyka* L je $L^R = \{u \mid \exists v \in L : u = v^R\}$, stručněji psáno $L^R = \{u^R \mid u \in L\}$.

Příklad: Zrcadlovým obrazem jazyka $L_1 = \{\varepsilon, a, abb, baaba\}$ je jazyk $L_1^R = \{\varepsilon, a, bba, abaab\}$.

Zrcadlovým obrazem jazyka $L_2 = \{w \mid |w|_a \bmod 2 = 0\}$ je jazyk L_2, neboli $L_2^R = L_2$.

Kontrolní otázka: Platí obecně $(uv)^R = u^R v^R$?

Samozřejmě, že ne (dosaďte např. $u = a$, $v = b$). Jistě snadno nahlédnete, že obecně platí $(uv)^R = v^R u^R$; podobně také $(L_1 L_2)^R = (L_2)^R (L_1)^R$.

Poslední operace, kterou si uvedeme, může na první pohled působit komplikovaně, ale pro výklad v dalších kapitolách je velmi užitečné jí důkladně porozumět (přinejmenším tedy její jednoduché formě). Záměrně začněme obecnou definicí:

Definice 1.6
(Levý) kvocient jazyka L_1 *podle* L_2 je definován takto:

$$L_2 \backslash L_1 = \{\, v \mid \exists u \in L_2 : uv \in L_1 \,\}.$$

Když se setkáme s definicí, které ihned neporozumíme, vždy je užitečné si definici nejdříve ‘osahat’ na konkrétních jednodušších příkladech. Uvažme třeba případ, kdy oba jazyky obsahují jediné slovo, tedy $L_1 = \{v_1\}$, $L_2 = \{v_2\}$. Podle definice $\{v_2\}\backslash\{v_1\} = \{\, v \mid \exists u \in \{v_2\} : uv \in \{v_1\} \,\}$. Tedy libovolné slovo v patří do $\{v_2\}\backslash\{v_1\}$ právě tehdy, když existuje $u \in \{v_2\}$, tedy nutně $u = v_2$, takové, že $uv = v_2 v$ je prvkem $\{v_1\}$, tedy nutně $v_2 v = v_1$. Do jazyka $\{v_2\}\backslash\{v_1\}$ tedy patří vůbec nějaké slovo jen tehdy, když v_2 je prefixem v_1; v tom případě patří do $\{v_2\}\backslash\{v_1\}$ právě to (jediné) slovo, které vznikne z v_1 odtržením (umazáním) prefixu v_2.

Např. $\{ab\}\backslash\{abbab\} = \{bab\}$, kdežto $\{ba\}\backslash\{abbab\} = \emptyset$.

Teď už si snadno odvodíme onu avizovanou jednoduchou formu, kterou je velmi záhodno důkladně pochopit:

(levý) kvocient jazyka podle slova $\{w\}\backslash L$, psaný také zkráceně $w\backslash L$,

je prostě sjednocení jazyků $w\backslash\{v\}$ pro všechna slova $v \in L$. Jinými slovy: jazyk $w\backslash L$ dostaneme tak, že vezmeme všechna slova z L mající prefix w a pak jim ten prefix w umažeme. Ještě jinak řečeno: slovo v patří do jazyka $w\backslash L$ právě tehdy, když po přidání w na začátek patří výsledné slovo wv do L.

Zvlášť důležitý bude pro nás základní případ, kdy w je rovno jedinému písmenu.

Příklad: Pohrajme si trochu s kvocienty; jako vždy, zkuste samozřejmě uvedené otázky nejdříve sami zodpovědět.

a) Jaká slova patří do jazyka $w\backslash L$, kde $w = a$ a $L = \{baaab, aba, aaa, bbb\}$?

   Jsou to slova ba, aa.

b) Jak je to v předchozím příkladu, je-li $w = \varepsilon$?

   Jistě jste si uvědomili, že $\varepsilon\backslash L = L$ pro každý jazyk L, takže správná odpověď v našem konkrétním případě je $baaab, aba, aaa, bbb$.

c) Jak je to v případě $w = aba$? A co v případě $w = bab$?

   V prvním případě se $w\backslash L$ rovná $\{\varepsilon\}$, v druhém případě se $w\backslash L$ rovná $\emptyset$ (žádné slovo z L totiž nemá prefix bab).

d) Chci-li zjistit $ab\backslash L$, mohu s výhodou využít již zjištěný $a\backslash L$?

   Určitě ano, jelikož $ab\backslash L$ je vlastně $b\backslash(a\backslash L)$; obecně platí $uv\backslash L = v\backslash(u\backslash L)$. (Promyslete si důkladně, proč je pořadí u, v prohozeno.)

   V našem konkrétním případě se zajímáme o slova z L, která mají prefix ab (který pak hodláme umazat). Když už ale víme, jak vypadají slova z L začínající a poté, co jim onen prefix a umažeme, tedy $a\backslash L = \{ba, aa\}$, stačí se zde podívat na slova začínající b a ten prefix b jim umazat: Máme tedy $ab\backslash\{baaab, aba, aaa, bbb\} = b\backslash(a\backslash L) = b\backslash\{ba, aa\} = \{a\}$.

e) Samozřejmě se není třeba omezovat na konečné jazyky. Jak byste charakterizovali např. slova z jazyků $0\backslash L$ a $1\backslash L$, kde $L = \{w \in \{0,1\}^* \mid |w|_1$ je liché $\}$?

   Je snadné nahlédnout, že $0\backslash L = L$ a $1\backslash L = \{w \in \{0,1\}^* \mid |w|_1$ je sudé $\}$.

f) Jak byste charakterizovali slova z jazyků $a\backslash L$, $b\backslash L$ kde $L = \{w \in \{a,b\}^* \mid$ každý výskyt podslova aa je ve w ihned následován znakem $b\,\}$?

Určitě rychle vidíme, že $b\backslash L = L$: každé slovo z $b\backslash L$ zajisté musí splňovat, že každý výskyt podslova aa je v něm ihned následován znakem b (tedy $b\backslash L \subseteq L$); ovšem když k libovolnému slovu $u \in L$ přidáme na začátek b, tak výsledné bu jistě patří do L – tedy $L \subseteq b\backslash L$.

Pro a je to jinak: sice i zde platí $a\backslash L \subseteq L$, ale máme např. $a \in L$ a $a \notin (a\backslash L)$. (Proč?) Jazyk $a\backslash L$ můžeme charakterizovat jako

$\{w \in \{a,b\}^* \mid$ každý výskyt podslova aa je ve w ihned následován znakem b a (navíc) pokud w začíná znakem a, pak po něm hned následuje $b\,\}$.

Po pochopení jednoduché varianty $w\backslash L$ není samozřejmě problémem ani obecná definice kvocientu, když si uvědomíme, že

$$L_2\backslash L_1 = \bigcup_{w \in L_2} w\backslash L_1 \,.$$

Ale pro tuto chvíli postačí, že plně rozumíme kvocientu podle slova (či dokonce jen podle písmene).

Shrnutí: Operace s jazyky už pro nás nejsou problémem. Plně rozumíme definicím a umíme je aplikovat. Speciálně jsme si dobře promysleli trochu ‘zapeklitou’ operaci kvocientu.

1.3 Cvičení

Orientační čas ke studiu této části: 2 hod.

Cíle této části:

Tato část obsahuje pouze otázky a příklady. Ty mají přispět k prohloubení vašeho porozumění látce celé této kapitoly.

Otázky:

Otázka 1.3: Můžeme množinu všech přirozených čísel považovat za abecedu v našem smyslu?

Otázka 1.4: Můžeme množinu všech přirozených čísel (alespoň v nějaké reprezentaci) považovat za formální jazyk v našem smyslu?

Otázka 1.5: Lze konečným počtem operací sjednocení a/nebo zřetězení z konečných jazyků vytvořit nekonečný jazyk?

Otázka 1.6: Jaký je rozdíl mezi prázdným jazykem $\emptyset$ a prázdným slovem ε?

Otázka 1.7: Kdy je iterace L^* jazyka L konečným jazykem?

Otázka 1.8*: Můžeme dvojí iterací jazyka dostat více slov než jednou iterací, tj. existuje jazyk, pro nějž $L^* \neq (L^*)^*$?

Cvičení 1.9: Která slova jsou zároveň prefixem i sufixem slova 101110110? (Najdete všechna tři taková?)

Cvičení 1.10: Vypište slova ve zřetězení jazyků $\{110, 0111\} \cdot \{01, 000\}$.

Cvičení 1.11: Uvažujme jazyky
$L_1 = \{w \in \{a, b\} \mid w$ obsahuje sudý počet výskytů symbolu $a\}$,
$L_2 = \{w \in \{a, b\} \mid w$ začíná a končí stejným symbolem $\}$.
Vypište prvních šest slov (rozumí se v uspořádání $<_L$) postupně pro jazyky $L_1 \cup L_2$, $L_1 \cap L_2$, $L_1 - L_2$, $\overline{L_1}$.

Cvičení 1.12: Najděte dva různé jazyky, které komutují v operaci zřetězení, tj. $L_1 \cdot L_2 = L_2 \cdot L_1$.

Cvičení 1.13*: Co vzniká iterací jazyka $\{00, 01, 1\}$? Patří tam všechna slova nad $\{0, 1\}$?

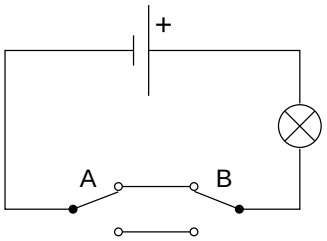
Cvičení 1.14: Uvažujme jazyky nad abecedou $\{0, 1\}$. Nechť L_1 je jazykem všech těch slov obsahujících nejvýše pět (výskytů znaku) 1 a L_2 je jazykem

všech těch slov, která obsahují stejně 0 jako 1. Kolik je slov v průniku $L_1 \cap L_2$?

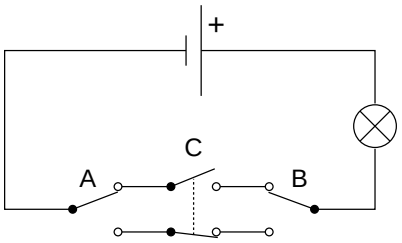
Cvičení 1.15: Uvažujme jazyky nad abecedou $\{a, b\}$. Vypište všechna slova ve zřetězení jazyků $L_1 = \{\varepsilon, abb, bba\}$ a $L_2 = \{a, b, abba\}$.

Cvičení 1.16*: Uvažujme jazyky nad abecedou $\{c, d\}$. Nechť L_0 je jazyk všech těch slov, která obsahují různé počty výskytů symbolu c a výskytů symbolu d. Snažte se co nejjednodušeji popsat, která slova patří do zřetězení $L_0 \cdot L_0$.

Cvičení 1.17: Představme si následující elektrický obvod s dvěma přepínači A a B. (Přepínače jsou provedeny jako aretační tlačítka, takže jejich polohu zvnějšku nevidíme, ale každý stisk je přehodí do druhé polohy.) Na počátku žárovka svítí. Pokusme se schematicky popsat, jaké posloupnosti stisků A, B vedou k opětovnému rozsvícení žárovky.



Cvičení 1.18: Obdobně jako v předchozím příkladě si vezměme následující obvod s přepínači A, B, C a jednou žárovkou. (Přepínač C má dva společně ovládané kontakty, z nichž je spojený vždy právě jeden.) Na počátku žárovka nesvítí. Jaké posloupnosti stisků A, B, C vedou k rozsvícení žárovky?



Cvičení 1.19: Uvažujme jazyky nad abecedou $\{0, 1\}$. Popište (slovně) jazyk vzniklý iterací $\{00, 111\}^*$.

Cvičení 1.20: Uvažujme jazyky nad abecedou $\{0, 1\}$. Nechť L_1 je jazykem všech těch slov obsahujících nejvýše jeden znak 1 a L_2 je jazykem všech těch slov, která se čtou stejně zepředu jako zezadu (tzv. palindromů) – tedy všech slov u, pro něž platí $u = u^R$. Která všechna slova jsou v průniku $L_1 \cap L_2$?

Poznámka: Pozor, průnik obou jazyků je nekonečný.

Cvičení 1.21: Proč obecně neplatí $(L_1 \cap L_2) \cdot L_3 = (L_1 \cdot L_3) \cap (L_2 \cdot L_3)$?

Kapitola 2

Konečné automaty a regulární jazyky

Cíle kapitoly:

Po prostudování této kapitoly máte dobře znát pojmy konečný automat a regulární výraz. Máte umět navrhnout konečný automat rozpoznávající daný (jednoduchý) jazyk a rovněž popsat takový jazyk regulárním výrazem. Máte zvládnout provádění operací s konečnými automaty. Důležitým cílem je rovněž pochopení pojmu nedeterminismus a jeho využití při návrhu automatů. Máte také získat představu o tom, proč některé konkrétní jazyky nemohou být rozpoznávány konečným automatem.

Klíčová slova: *konečné automaty, regulární výrazy*

2.1 Motivační příklad

Orientační čas ke studiu této části: 2 hod.

Cíle této části:

Na konkrétním jednoduchém „programátorském“ příkladu byste měli nejdříve intuitivně pochopit jeden z motivačních zdrojů, který vcelku přirozeně vede k pojmu a návrhu konečného automatu jako rozpoznávače jazyka. Teprve potom (v dalších sekcích) přistoupíme k precizaci takto získané intuice. Tento postup má přispět k pochopení obecného faktu, že teoretické pojmy (v informatice a jinde) „nepadají z nebe“, ale snaží se co nejpreciznějí a nejužitečněji zachytit a objasnit podstatu skutečných praktických problémů a přispět k jejich řešení.

Klíčová slova: *vyhledávání vzorku v textu*

Podívejme se na následující algoritmus, zapsaný jako „pascalský“ program.

```
procedure SEARCH (var F: file)
const length = 6            (* delka hledaneho retezce *)
const P = [ a,b,a,a,b,a ]   (* hledany retezec *)
var A: array [ 1..length ] of char
begin
  for i:=1 to length do
    read( A[i], F ); if EOF (* end of file *) then return
  endfor
  while true do
    if EQUAL(P,A) then ‘‘vypis misto vyskytu’’
    for i:=1 to length-1 do
      A[i]:=A[i+1]
    endfor
    read( A[length], F ); if EOF then return
  endwhile
end
```

Procedura `EQUAL` je naprogramována následovně.

```
function procedure EQUAL
(var S1,S2: array [ 1..length ] of char): boolean
begin
  for i:=1 to length do
    if not( S1[i] = S2[i] ) then return FALSE
```

```
  endfor
  return TRUE
end
```

Programátorsky zběhlý čtenář jistě nemá problémy s pochopením uvedeného (pseudo)kódu, byť sám třeba programuje v jazycích jiného typu.

Kontrolní otázka: Jak byste charakterizovali činnost procedury `SEARCH`, je-li spuštěna na soubor obsahující (dlouhou) sekvenci znaků z množiny $\{\mathtt{a}, \mathtt{b}\}$? (Sekvence je zakončena speciálním znakem, např. $<$ `EOF` $>$.)

Ano, jistě jste pochopili, že procedura vypíše všechny výskyty řetězce (tedy slova) `abaaba` ve vstupním souboru. Pod výpisem si např. představme výpis pozice konce nalezeného řetězce; tato technická otázka teď pro nás není podstatná, i když u kompletního počítačového programu by se samozřejmě musela také dotáhnout.

Z „programátorského hlediska“ si jistě hned všimneme možností zmenšení časové náročnosti uvedeného programu. Např. prováděný posun obsahu pole `A` před přečtením dalšího znaku není jistě nejlepší řešení. (Napadá vás něco elegantnějšího?) Dále si všimneme, že čtení z vnějšího souboru znak po znaku by mohlo být zdrojem velké ztráty času. (Proč?) Měli bychom si být jisti, že tento problém ve skutečnosti tiše řeší knihovní procedury pro čtení apod.; pak se nemusíme tímto problémem dále zabývat.

Vžijme se teď do situace, kdy máme ze sebe vydat maximum a napsat program, který je z hlediska časové náročnosti podstatně lepší než ta uvedená procedura `SEARCH`, byť vylepšená o přímočaré programátorské nápady. To je možné jen tehdy, jde-li úkol realizovat principiálně lepším algoritmem. Existuje takový algoritmus?

Poznámka: Nejde nám pochopitelně prvořadě o hledání speciálního řetězce `abaaba`, ale obecněji o hledání výskytů vzorku v souboru (např. textu). Vzorek `abaaba` nám teď slouží jen jako malý konkrétní příklad.

Podívejme se na jiné řešení procedury `SEARCH`.

```
procedure SEARCH1 (var F: file)
const length = 6
type state = 0 .. length
type alphabet = (a,b)
```

```
const A: array [ state , alphabet ] of state
   = [ [1,0], [1,2], [3,0], [4,2], [1,5], [6,0], [4,2] ]
var q: state
begin
  q:=0
  while true do
    if q=6 then ‘‘vypis misto vyskytu’’
    read( ch, F ); if EOF then return
    q := A[ q, ch ]
  endwhile
end
```

Bez dalšího komentáře, tj. bez pochopení, jak tento program vznikl, není samozřejmě vůbec jasné, že `SEARCH1` realizuje tentýž úkol jako `SEARCH` (tj., že pro stejnou vstupní sekvenci symbolů `a`,`b` vydá stejný výstup). Ihned ale můžeme ověřit, že procedura `SEARCH1` poběží jistě rychleji než `SEARCH`. (Proč?)

Jak můžeme dojít k oné „zázračné tabulce“ (tj. dvourozměrnému poli) `A` zapsané v `SEARCH1` a zároveň k přesvědčení, že je to správně, tedy že `SEARCH1` dělá to, co od ní očekáváme? Nejedná se samozřejmě o zázrak, ale o použití obecně platného postupu, který můžeme naznačit např. takto:

- prvořadé je důkladné porozumění zadání úkolu, jeho přesná specifikace (na správné úrovni abstrakce), promyšlení z různých úhlů, nejprve na jednoduchých případech apod.,
- řešení pak (jakoby samo) vychází z (důkladně promyšlené) podstaty úkolu, stejně jako důkaz jeho správnosti.

Tento ideál se v našem konkrétním příkladu můžeme pokusit realizovat následovně. Specifikujme si náš úkol, označený U_0, např. takto:

> U_0 (specifikace): v dané posloupnosti znaků `a`, `b` (zakončené speciálním znakem), připravené k sekvenčnímu čtení, „ohlaš“ každý výskyt `abaaba`.

Je zřejmé, že budeme muset přečíst první znak posloupnosti. Přečtení speciálního koncového znaku bude v našem případě vždy znamenat ukončení práce, takže tuto možnost nebudeme dále explicitně zmiňovat. Když je přečteným znakem `a`, je očividně naším zbývajícím úkolem

> U_1 („zbytek“ úkolu U_0 po přečtení `a`; specifikace): v dané posloupnosti (což je nepřečtený zbytek původní posloupnosti) ohlaš každý výskyt `abaaba`, ale na začátku také případný výskyt prefixu `baaba` (proč?).

Úkol U_1 je očividně jiný než U_0, proto jsme jej označili jinak (v našem případě dalším dosud nepoužitým indexem).

Promyslíme-li si „zbytek“ úkolu U_0, který máme vykonat v případě, že prvním znakem je `b`, zjistíme, že se zbytkem posloupnosti máme vlastně udělat zase úkol U_0; není tedy teď třeba zavádět nový úkol (U_2), protože jej vyřešíme (rekurzivním) voláním U_0.

Máme tedy:

> U_0 (realizace): přečti další znak;
> když je to `a`, tak (proveď) U_1, když je to `b`, tak (proveď) U_0.

Jak realizujeme výše specifikovaný úkol U_1 ?

Přečteme pochopitelně další znak. Když je to `a`, tak první část specifikace U_1 (ohlaš každý výskyt `abaaba`) nám ukládá, že ve zbytku máme ohlásit každý výskyt `abaaba` a také případný prefix `baaba`, a druhá část specifikace U_1 (případný výskyt prefixu `baaba`) nám už neukládá nic, protože přečtené `a` pohřbilo naděje na prefix `baaba`.

Když je to `b`, tak první část specifikace U_1 (ohlaš každý výskyt `abaaba`) nám ukládá, že ve zbytku máme ohlásit každý výskyt `abaaba` a jinak nic, druhá část specifikace U_1 (případný výskyt prefixu `baaba`) nám ukládá ohlásit případný prefix `aaba`.

Takže máme

> U_1 (realizace): přečti další znak;
> když je to `a`, tak (proveď) U_1, když je to `b`, tak (proveď) U_2

> U_2 (specifikace): v dané („zbývající“) posloupnosti ohlaš každý výskyt `abaaba`, ale na začátku také případný výskyt prefixu `aaba`.

Všimněme si, že naše realizace U_0, U_1 koresponduje s prvními dvěma řádky tabulky v `SEARCH1`.

Cvičení 2.1: Pečlivě dokončete konstrukci vznikajícího „programu“ (s vzájemně se rekurzivně volajícími procedurami $U_0, U_1, U_2, \ldots$). Asi vás napadne, že zachycovat vznikající strukturu můžete zároveň tabulkou i určitým grafem, který vás jistě přirozeně napadne. (Uzly grafu jsou označeny $U_0, U_1, U_2, \ldots$, k orientovaným hranám (tedy „šipkám“) jsou připsány znaky `a`, `b`. (Udělejte to!)

Doufejme, že jste vystačili s „procedurami“ $U_0, U_1, U_2, \ldots, U_6$ a že struktura navržené realizace přesně koresponduje s tabulkou v `SEARCH1`. Speciálně by vám mělo vyjít

> U_6 (specifikace): v dané (zbývající) posloupnosti ohlaš každý výskyt `abaaba`, ale na začátku také případné prefixy ε, `aba`, `baaba`.

V realizaci U_6 dáme pochopitelně před přečtením dalšího znaku povel ‘OHLAŠ’, neboť každá posloupnost má prefix ε.

Takže vznik tabulky v `SEARCH1` už je nám jasný! Navíc bychom jistě byli schopni takovou tabulku sestrojit pro každý zadaný vzorek (řetězec), byť by to u delších řetězců mohla být docela fuška.

Poznámka: Později se k problému vrátíme a uvidíme, že tvorba takových tabulek k zadaným vzorkům se dá zalgoritmizovat (a tedy naprogramovat).

Všimněme si, že na realizaci našeho úkolu U_0 se dá hledět jako na čtení zadané posloupnosti znaků (tedy zadaného slova) zleva doprava, přičemž před přečtením dalšího symbolu vždy blikne „zelené světlo“, jestliže dosud přečtené slovo (tedy dosud přečtený prefix zadané posloupnosti) splňuje podmínku

„mám sufix `abaaba` “,

a blikne „červené světlo“, jestliže dosud přečtený prefix tuto podmínku nesplňuje.

Zkusme teď ještě navrhnout podobnou tabulku pro případ, kdy čteme soubor (tedy slovo) obsahující znaky 0,1 a máme tentokrát (zeleným světlem) ohlásit všechny prefixy, které splňují podmínku

„obsahuji podslovo 010 nebo #1 ve mně je sudý“.

Zde výrazem #1 označujeme počet výskytů znaku 1; „nebo“ myslíme pochopitelně v nevylučovacím smyslu (tedy obě podmínky mohou platit současně).

Specifikovaný úkol si tentokrát označme q_0 a všimněme si, že realizace q_0 bude začínat povelem OHLAŠ (proč?).

Jistě nás již napadlo, že komplikované vyjadřování „úkol, který máme vykonat ve zbytku, když při plnění úkolu q přečteme a“ je vhodné nahradit dohodnutou stručnou notací, např. $\delta(q, a)$.

Co je tedy v našem konkrétním případě $\delta(q_0, 0)$? Jistě snadno přijdeme na to, že

> $\delta(q_0, 0)$ (specifikace): ohlaš (ve zbytku k přečtení) všechny prefixy, které obsahují 010 nebo začínají 10 nebo #1 je v nich sudý.

Tento úkol je očividně jiný než q_0, označíme jej proto q_1; máme tedy $\delta(q_0, 0) = q_1$.

Všimněme si, že každý úkol (který vzniká při našich nynějších úvahách) je typu

> „ohlaš (ve zbylé posloupnosti) všechny prefixy, které splňují jistou podmínku“

Proto se nabízí zjednodušení značení i při specifikaci jednotlivých úkolů. Úkol q zadáme prostě vhodným popisem množiny těch slov (potenciálních prefixů posloupnosti zbývající k přečtení), které splňují onu podmínku. Označme takovou množinu

$$L_q^{toAcc}.$$

Je to tedy jazyk (tj. množina) obsahující právě ta slova, po jejichž přečtení máme zasvítit zeleně, plníme-li úkol q. Přečtení takového slova má vést k „ohlášení“; říkáme také, že slovo je „přijato“, „vede k přijetí“ (anglicky „to Acceptance“) – odtud je použitá zkratka.

V našem příkladu tedy máme

$L_{q_0}^{toAcc} = \{\, w \in \{0,1\}^* \mid w$ obsahuje podslovo 010 nebo $|w|_1$ je sudé $\}$

$L_{q_1}^{toAcc} = \{\, w \in \{0,1\}^* \mid w$ obsahuje podslovo 010 nebo má prefix 10 nebo $|w|_1$ je sudé $\}$

(Připomínáme, že $|w|_1$ označuje počet výskytů znaku 1 ve w.)

Podívejme se teď na úkol $\delta(q_0, 1)$; specifikace úkolu vlastně znamená vhodnou charakterizaci jazyka $L_{\delta(q_0,1)}^{toAcc}$. Jistě rychle zjistíme, že

$L_{\delta(q_0,1)}^{toAcc} = \{\, w \in \{0,1\}^* \mid w$ obsahuje podslovo 010 nebo $|w|_1$ je liché $\}$

což je jistě jiný jazyk (úkol) než $L_{q_0}^{toAcc}$, $L_{q_1}^{toAcc}$ (proč?). Takže zavedeme nový úkol q_2 a definujeme $\delta(q_0, 1) = q_2$ a

$L_{q_2}^{toAcc} = \{\, w \in \{0,1\}^* \mid w$ obsahuje podslovo 010 nebo $|w|_1$ je liché $\}$

Poznámka: Je užitečné si všimnout, že naše činnost se dá charakterizovat jako určitá konstrukce jednoduchých kvocientů jazyků. (Kvocient je ta „složitá“ jazyková operace zmíněná dříve.) Např. popsat $L_{\delta(q_0,1)}^{toAcc}$ vlastně znamená popsat $1 \backslash L_{q_0}^{toAcc}$. Později se k tomu ještě vrátíme.

Celkové vytvářené schéma (funkci δ) pochopitelně můžeme zase zadat tabulkou a grafem. Zatím jsme vytvořili následující fragment tabulky:

	0	1
$\leftrightarrow q_0$	q_1	q_2
$\leftarrow q_1$		
q_2		

Vstupní šipkou $\rightarrow$ jsme označili onen výchozí (počáteční) úkol (říkejme také „stav“), výstupními šipkami $\leftarrow$ označujeme stavy, které začínají „ohlášením“ (zeleným světlem) – říkáme jim také „přijímající stavy“. Jak vidíme, i počáteční stav může být přijímací a přijímajících stavů může být více než jeden.

Kontrolní otázka: Proč je q_1 přijímající a q_2 ne?

Ano, máte pravdu, jistě jste si uvědomili, že q je přijímající právě tehdy, když $\varepsilon \in L_q^{toAcc}$ (tedy když prázdné slovo splňuje příslušnou podmínku).

Cvičení 2.2: Dokončete výše započatou tabulku. Popište přitom pečlivě všechny jazyky $L_{q_i}^{toAcc}$ pro $i = 0, 1, 2, \ldots$. Zkuste přitom předem odhadnout počet stavů (řádků tabulky), které budete potřebovat.

Cvičení 2.3: Vraťte se ještě k úkolu U_0, kde

$$L_{U_0}^{toAcc} = \{\, w \in \{a, b\}^* \mid w \text{ má sufix } abaaba \,\}$$

Definujte přesně jazyky $L_{U_1}^{toAcc}, L_{U_2}^{toAcc}, \ldots, L_{U_6}^{toAcc}$. (Varování: např. $L_{U_1}^{toAcc}$ se nerovná jazyku $\{\, w \in \{a, b\}^* \mid w$ má sufix $abaaba$ nebo prefix $baaba \,\}$; proč?)

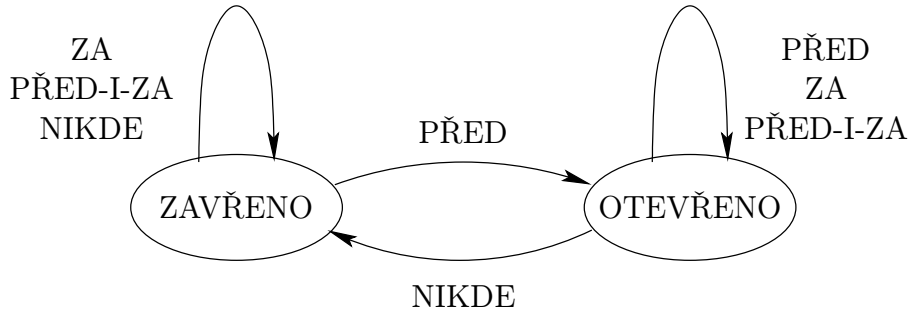
Shrnutí: Tak, a je to! Teď už intuitivně víme, co je to konečný automat rozpoznávající jazyk (no přece ta „tabulka“, která určuje, která slova jsou přijata a která ne). Teď už nás v následující části matematická notace nezastraší; jistě pochopíme, že je to „jen“ formalizace, tedy zpřesnění, toho, o čem už prvotní představu máme.

2.2 Konečné automaty jako rozpoznávače jazyků

Orientační čas ke studiu této části: 2 hod.

Cíle této části:

Cílem je zvládnutí přesné definice a důkladné pochopení pojmu konečného automatu jako „přijímače slov“, dále pak prohloubení schopnosti konstrukce jednoduchých automatů. Začneme ovšem náznakem obecnějšího modelování systémů, který má přispět k pochopení faktu, že pojem konečného automatu (či konečného přechodového systému) má i další motivační zdroje.



Obrázek 2.1:

Klíčová slova: *(deterministický) konečný automat*

Začneme nejdříve trochu obecněji, ať alespoň naznačíme, že pojem konečného automatu (či konečného přechodového systému) se neobjevuje jen v souvislostech přijímání slov nějakého jazyka.

Konečným automatem obecně rozumíme systém (či model systému), který může nabývat konečně mnoho (obvykle ne „příliš mnoho“) stavů. Aktuální stav se mění na základě vnějšího podnětu (možných podnětů také není „příliš mnoho“) s tím, že pro daný stav a daný podnět je jednoznačně určeno, jaký stav bude následující (tj. do jakého stavu systém přejde). Konkrétní konečný automat se často zadává diagramem, který také nazýváme *stavový diagram* či *graf automatu*. Jiná možnost zadání je *tabulkou*, která je sice poněkud suchopárnější, ale např. je vhodnější pro počítačové zpracování a pro složitější automaty může být i přehlednější.

Příklad: Diagram na Obrázku 2.1 je popisem jednoduchého automatu řídícího vstupní dveře do supermarketu. Dveře nejsou posuvné, ale otvírají se dovnitř. Proto se nemohou otvírat ani zavírat, když za nimi někdo stojí.

Automat může být ve dvou stavech (Zavřeno, Otevřeno) a podnětem (např. snímaným v pravidelných krátkých intervalech) je informace, na které z podložek (před dveřmi, za dveřmi) se někdo nachází. Kromě (pro naše oko přehledného) diagramu lze tutéž informaci sdělit tabulkou na Obrázku 2.2.

Cvičení 2.4: (nepovinné) Popište slovně nějaký jednoduchý automat na mince, který je schopen vydat čaj nebo kávu dle volby, a pak jej modelujte stavovým diagramem (grafem automatu).

	PŘED	ZA	PŘED-I-ZA	NIKDE
ZAVŘENO	OTEVŘENO	ZAVŘENO	ZAVŘENO	ZAVŘENO
OTEVŘENO	OTEVŘENO	OTEVŘENO	OTEVŘENO	ZAVŘENO

Obrázek 2.2:

Po tomto elementárním příkladu modelování systému se už vrátíme k chápání konečného automatu především jako *rozpoznávače jazyka*, tedy „zařízení“ k přijímání vybraných slov v dané abecedě. Již víme, že k tomu je také potřeba vymezit počáteční stav a přijímající stavy. Formalizujme nyní naše intuitivní pojmy v jazyce matematiky.

Poznámka: Čtenáři, kterému ještě není jasné, k čemu je formalizace dobrá, snad teď postačí odpověď, že je to dobré přinejmenším jako stručné a jednoznačné značení. (Později snad ocení užitečnost vhodné matematické formalizace hlouběji.)

Definice 2.1
Konečný automat (zkráceně KA) je dán uspořádanou pěticí $A = (Q, \Sigma, \delta, q_0, F)$, kde

- Q je konečná neprázdná množina *stavů*,
- Σ je konečná neprázdná množina zvaná (vstupní) *abeceda*,
- $\delta : Q \times \Sigma \to Q$ je *přechodová funkce*,
- $q_0 \in Q$ je *počáteční* (iniciální) *stav* a
- $F \subseteq Q$ je množina *přijímajících* (koncových) *stavů*.

V zásadě nás nic nepřekvapuje, vyjadřuje to přesně naše dřívější intuitivní porozumění. Význam zápisu $\delta(q, a)$ (kde $q \in Q, a \in \Sigma$) je nám taky jasný; jen jsme si ujasnili, že nám známá tabulka či graf se vlastně matematicky dá chápat jako funkce, která danému stavu („úkolu“) q a znaku a přiřadí následující stav. Definičním oborem funkce δ je tedy množina $\{(q, a) \mid q \in Q, a \in \Sigma\}$, tj. kartézský součin $Q \times \Sigma$; oborem hodnot je Q. Nepřekvapilo nás ani, že ač počáteční stav je jen jeden, přijímajících stavů může být více.

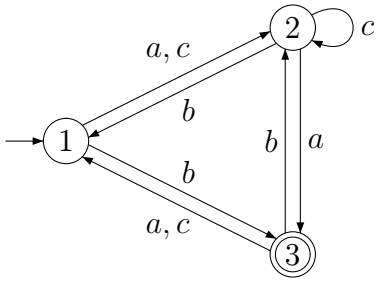
Poznámka: Vidíme, že definice umožňuje i $F = \emptyset$; jak čtenář jistě očekává, takový automat nepřijme žádné slovo a jím rozpoznávaný jazyk bude tedy $\emptyset$.

Grafem automatu (neboli stavovým diagramem) rozumíme orientovaný ohodnocený graf, ve kterém

- vrcholy jsou stavy automatu, tj. prvky množiny Q,
- počáteční stav (q_0) je vyznačen příchozí šipkou a koncové stavy (prvky F) dvojitým kroužkem (či alternativně výchozí šipkou)
- hrana z q do q' je označená výčtem všech písmen abecedy, které stav q převádějí na q', tj. výčtem prvků množiny $\{a \in \Sigma \mid \delta(q,a) = q'\}$.

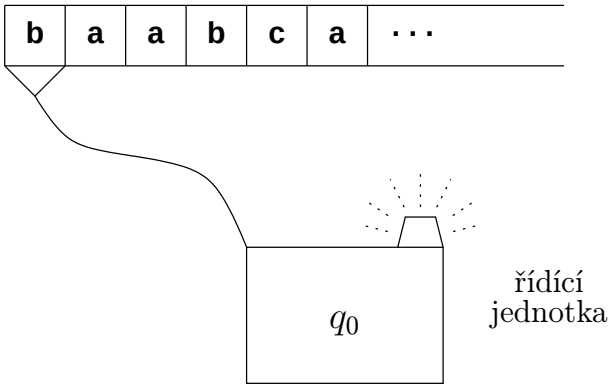
Hrany nekreslíme pro dvojice vrcholů mezi kterými není přechod pro žádné písmeno abecedy. Pokud se z vrcholu q přechází zpět do q, kreslí se smyčka.

Příklad: Podívejme se na ukázku grafu jednoduchého třístavového automatu:



Graf reprezentuje automat $A = (Q, \Sigma, \delta, q_0, F)$, kde $Q = \{1,2,3\}$, $\Sigma = \{a,b,c\}$, $q_0 = 1$, $F = \{3\}$ a pro přechodovou funkci platí například $\delta(1,a) = \delta(1,c) = 2$, $\delta(2,c) = 2$, $\delta(2,b) = 1$, atd. Pokud na vstupu automatu bude slovo „*accaab*“, stane se následující: Automat začne v 1, přejde čtením a do 2, pak čtením c dvakrát zůstává v 2, čtením a přejde do 3 (kde „ohlásí“ přijetí dosud přečteného prefixu „*acca*“), čtením (dalšího) a se vrátí do stavu 1 a pak čtením b přejde do stavu 3, kde ohlásí přijetí dosud přečteného prefixu, což je v našem případě již celé vstupní slovo.

Přechodovou tabulkou automatu rozumíme tabulku s řádky označenými stavy automatu a sloupci označenými symboly abecedy, ve které políčko na řádku q a sloupci a udává stav $\delta(q,a)$. Počáteční stav je značený vstupní šipkou $\rightarrow$ a přijímající stav výstupní šipkou $\leftarrow$ nebo kroužkem kolem čísla stavu.



Obrázek 2.3:

Příklad: Například výše zakreslený automat má přechodovou tabulku:

	a	b	c
$\rightarrow 1$	2	3	2
2	3	1	2
$\leftarrow 3$	1	2	1

Ještě jednou si popišme činnost konečného automatu. Nyní si ale představme, že nevidíme „střeva“ (tedy nevidíme ani graf ani tabulku ani nic jiného popisujícího přechodovou funkci δ), ale jsme v pozici vnějšího pozorovatele. Jsme tedy v situaci znázorněné na obrázku 2.3. Vidíme *řídicí jednotku* (pro nás je to teď černá skříňka např. s několika-bitovou pamětí), která je *čtecí hlavou* spojena se (vstupní) *páskou*, na níž je zapsáno slovo – zleva doprava jsou v jednotlivých buňkách pásky uložena písmena daného slova.

Na začátku je řídicí jednotka v počátečním stavu a hlava je připojena k nejlevějšímu políčku pásky. Činnost automatu, zvaná *výpočet*, pak probíhá v *krocích*: v každém kroku je přečten symbol (hlava se po přečtení posune o jedno políčko doprava) a řídicí jednotka se nastaví do stavu určeného aktuálním stavem a přečteným symbolem (přechodová funkce je implementována v řídicí jednotce).

Je možné si také představit, že na řídicí jednotce je (zelené) „světélko“ signalizující navenek, zda aktuální stav je či není přijímající (čili „zvenku“ rozlišujeme u řídicí jednotky jen dva stavy – přijímá/nepřijímá).

Pojem *přijímání slova* je nám už jistě zcela jasný. A co to je *jazyk přijímaný*, říkáme také *rozpoznávaný*, daným *automatem* A ? Samozřejmě je to množina těch slov, která jsou automatem A přijímána. I když jsou tyto pojmy jasné, vyplatí se zavést pro ně ještě stručné definice, lépe řečeno značení.

Značení: Máme-li dán automat $A = (Q, \Sigma, \delta, q_0, F)$, budeme zápisem

$$q \xrightarrow{w} q'$$

označovat fakt, že automat A ze stavu q přejde přečtením slova w do stavu q'. Pokud by mohlo dojít k nedorozumění, můžeme psát podrobněji $q \xrightarrow{w}_A q'$, aby bylo zřejmé, že se odkazujeme k automatu A.

Zápisem

$$q \xrightarrow{w} Q'$$

pro $Q' \subseteq Q$ (např. $Q' = F$) označujeme fakt, že existuje $q' \in Q'$ takové, že $q \xrightarrow{w} q'$ (tedy že A přejde z q přečtením w do nějakého stavu v Q').

α

Matematická poznámka (možno přeskočit): Zavedené značení $q \xrightarrow{w} q'$, pro automat $A = (Q, \Sigma, \delta, q_0, F)$, je tak názorné, že snad není třeba nic dodávat. Kdybychom chtěli definovat detailně matematicky, mohli bychom podat následující induktivní definici (indukce je vedena podle délky slova):

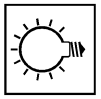
1. $q \xrightarrow{\varepsilon} q$
2. $q \xrightarrow{a} q' \Leftrightarrow \delta(q, a) = q'$
3. když $q \xrightarrow{a} q'$ a $q' \xrightarrow{u} q''$, tak $q \xrightarrow{au} q''$

Připomeňme, že v grafu automatu odpovídá výpočtu $q \xrightarrow{w} q'$ přirozeným způsobem jistý sled (posloupnost hran) z q do q'. Je-li $w = a_1 a_2 \dots a_n$, pak na i-té hraně onoho sledu je písmeno a_i, pro $i = 1, 2, \dots, n$. Tento sled samozřejmě může různě cyklit a opakovat hrany i stavy.

A teď už stručná definice přijímání slov a jazyků:

Definice 2.2
Mějme konečný automat $A = (Q, \Sigma, \delta, q_0, F)$.
Slovo $w \in \Sigma^*$ je *přijímáno* automatem A, jestliže $q_0 \xrightarrow{w} F$.
Jazykem rozpoznávaným (přijímaným) automatem A rozumíme jazyk
$L(A) = \{w \in \Sigma^* \mid$ slovo w je přijímáno $A\} = \{w \in \Sigma^* \mid q_0 \xrightarrow{w} F\}$.

Osvěžme se teď po těch definicích sestrojením malého automatu s jednoznakovou abecedou:



Řešený příklad 2.1: Navrhněme automat rozpoznávající jazyk
$L = \{w \in \{a\}^* \mid w \text{ má sudou délku }\}$.

Řešení: Zavedeme samozřejmě počáteční stav, označme jej třeba q_0; jeho „úkolem“ je L, tedy $L_{q_0}^{toAcc} = L$; to znamená, že právě slova z L mají automat převést z q_0 do některého z přijímajících stavů. Jelikož v L je i prázdné slovo ε, bude q_0 také přijímající.

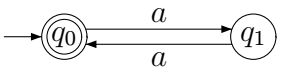
Co je $a\backslash L$, tedy kam má automat přejít z q_0 po přečtení a ? Je zřejmé, že

$$a\backslash L = \{w \in \{a\}^* \mid w \text{ má lichou délku }\},$$

což je jiný jazyk než L. Zavedeme tedy další stav q_1, jehož „úkolem“ $L_{q_1}^{toAcc}$ je

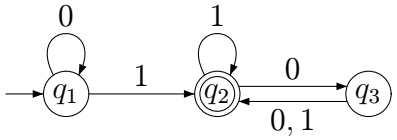
$L_1 = a\backslash L = \{w \in \{a\}^* \mid w \text{ má lichou délku }\}$.

Jelikož $\varepsilon \notin L_1$, tak q_1 není přijímající. A protože vidíme $a\backslash L_1 = L$, dokončíme graf automatu takto:



Říkáte, že počítat paritu délky dosud přečteného slova (tzn. přepínat se mezi stavem „q_0... sudá délka“ a „q_1... lichá délka“) vás napadlo hned, bez přemýšlení o kvocientech? Výborně, pak je alespoň užitečné si uvědomit, že s kvocienty jste stejně nutně pracovali alespoň podvědomě. Je to dobrá „záloha“ pro případ, když si hned nevíme rady či si nejsme jisti.

Cvičení 2.5: Chápejme Obrázek 2.4 jako popis konečného automatu $A = (Q, \Sigma, \delta, q_0, F)$. Vypište *přímým výčtem* hodnoty všech členů pětice A. Pak porovnejte s Obrázkem 2.5.
Vypište si 10 nejkratších slov, která automat přijímá.
Charakterizujte jazyk přijímaný tímto automatem (přesnou) podmínkou, kterou splňují jeho slova.



přijímá:	nepřijímá:
1101,010101,...	0110,0010,...

Obrázek 2.4:

$A = (Q, \Sigma, \delta, q_1, F)$, kde

$$\begin{array}{lcll} Q & = & \{q_1, q_2, q_3\} & \delta(q_1, 0) = q_1, \quad \delta(q_1, 1) = q_2, \\ \Sigma & = & \{0, 1\} & \delta(q_2, 0) = q_3, \quad \delta(q_2, 1) = q_2, \\ F & = & \{q_2\} & \delta(q_3, 0) = q_2, \quad \delta(q_3, 1) = q_2 \end{array}$$

Obrázek 2.5:

Poznámka: Výstižně charakterizovat jazyk přijímaný zadaným automatem může být obecně těžký problém – je to jako snažit se poznat, co dělá zadaný program, který není řádně strukturovaný ani dobře okomentovaný. S automatem na obrázku 2.4 jste ovšem jistě velké problémy neměli.

Na závěr této části ještě sami sestrojte automat pro následující jazyk. Řešení zde není uvedeno, sami se pečlivě přesvědčte, že váš automat plní daný úkol.

Cvičení 2.6: Navrhněte konečný automat přijímající jazyk

$L = \{w \in \{0,1\}^* \mid$ za každým podslovem 11 ve w ihned následuje $0\}$.

(Upozornění: Uvědomte si, že podmínka je automaticky splněna pro každé slovo neobsahující podslovo 11 !)

Pokud tak nebudete postupovat již při konstrukci, tak alespoň dodatečně popište všechny navzájem různé jazyky mezi kvocienty

$\varepsilon\backslash L$, $0\backslash L$, $1\backslash L$, $00\backslash L$, $01\backslash L$, $10\backslash L$, $11\backslash L$, $000\backslash L$, ...

Shrnutí: Takže pojmu konečného automatu jako rozpoznávače jazyka už velmi dobře rozumíme a umíme jej i přesně definovat. Navíc už máme zkušenost s vlastní konstrukcí automatů pro konkrétní (jednoduché) jazyky a

uvědomujeme si (otevřeně či tiše prováděnou) práci s kvocienty jazyků, která je v pozadí těchto konstrukcí.

2.3 Modulární návrh konečných automatů

Orientační čas ke studiu této části: 1 hod.

Cíle této části:

Zde si máte promyslet a zvládnout několik přímočarých způsobů, jak lze složitější automaty (algoritmicky) konstruovat z jednodušších. Přesněji řečeno, máte pochopit, že konstruovat automat pro jazyk, jehož slova jsou charakterizována booleovskou kombinací jednoduchých podmínek, lze tak, že zkonstruujeme automaty pro jazyky dané oněmi jednoduchými podmínkami a pak z nich již mechanicky (algoritmicky) vytvoříme výsledný automat konstrukcemi kopírujícími onu booleovskou kombinaci.

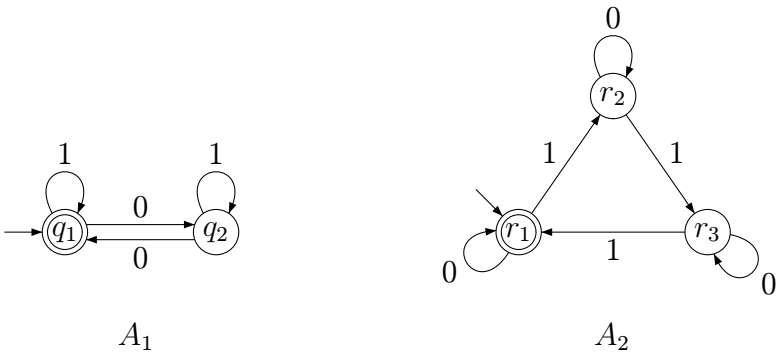
Klíčová slova: *konečné automaty, operace na jazycích, sjednocení a průnik jazyků, doplněk jazyka*

Přestože známe určitý návod k sestrojení konečného automatu pro zadaný jazyk (postupnou charakterizací kvocientů jazyka podle delších a delších slov), jedná se pořád o tvůrčí činnost, kterou není možné úplně algoritmizovat. Uvidíme ale, že existují mnohé algoritmické postupy, které naši práci výrazně usnadní.

Začneme ilustrací toho, jak lze výhodně použít obecný modulární postup; při takovém postupu řešíme složitější problém jeho rozdělením na podproblémy („moduly“), které lze vyřešit jednodušeji a z jejichž řešení lze řešení celkového problému (snadno) složit.

Možná, že čtenáře něco takového již napadlo, když jsme v části 2.1 konstruovali automat pro jazyk

$L = \{ w \in \{0,1\}^* \mid w$ obsahuje podslovo 010 nebo $|w|_1$ je sudé $\}$.



Obrázek 2.6:

Nabízí se zde myšlenka sestrojit automaty pro jazyky $L_1 = \{ w \in \{0,1\}^* \mid w$ obsahuje podslovo 010 $\}$ a $L_2 = \{ w \in \{0,1\}^* \mid |w|_1$ je sudé $\}$ a pak z nich nějak vhodně vytvořit „kombinovaný“ automat pro jazyk $L = L_1 \cup L_2$.

My se nad touto kombinací zamyslíme nejdříve na jiném příkladu. Uvažujme jazyk

$L = \{ w \in \{0,1\}^* \mid$ počet nul ve w je dělitelný dvěma nebo počet jedniček je dělitelný třemi $\}$.

Tedy $L = L_1 \cup L_2$, kde

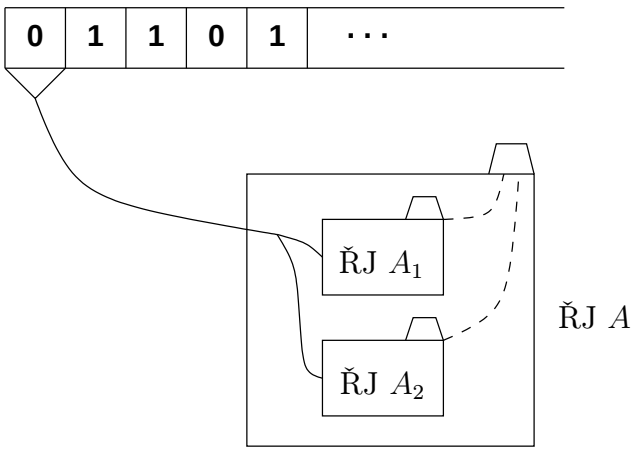
$L_1 = \{w \in \{0,1\}^* \mid |w|_0$ je dělitelné 2$\}$,

$L_2 = \{w \in \{0,1\}^* \mid |w|_1$ je dělitelné 3$\}$.

Na Obrázku 2.6 jsou znázorněny automaty rozpoznávající jazyky L_1 a L_2.

Jak zjistíme, zda dané slovo patří do $L(A_1) \cup L(A_2)$? Prostě je necháme zpracovat (přečíst) oběma automatům A_1, A_2 a podíváme se, zda alespoň jeden z nich skončil v přijímajícím stavu. Ovšem tuto naši činnost může očividně provádět i jistý (větší) konečný automat A, který souběžně realizuje výpočty obou (menších) výchozích automatů. Vnější pohled na onen automat je znázorněn na obrázku 2.7.

Kontrolní otázka: Je vám už jasné, co jsou stavy automatu A, co je jeho počáteční stav, co jsou jeho přijímající stavy, a jak vypadá jeho přechodová funkce? Zkuste se nejdříve sami zamyslet, než se podíváte dále.



Obrázek 2.7:

Obrázek 2.8 znázorňuje „střeva“ A (tedy stavový diagram). Měl by vyjadřovat to, co jste již sami pochopili.

Poznámka: Způsob rozmístění stavů A na obrázku 2.8 není náhodný, ale snaží se jádro myšlenky i přehledně „vizualizovat“. Každý řádek odpovídá jednomu stavu automatu A_1 a obsahuje kopii stavové množiny A_2, každý sloupec odpovídá jednomu stavu automatu A_2 a obsahuje kopii stavové množiny A_1.

Důležité je, že uvedený postup lze přímočaře zobecnit pro jakékoli automaty.

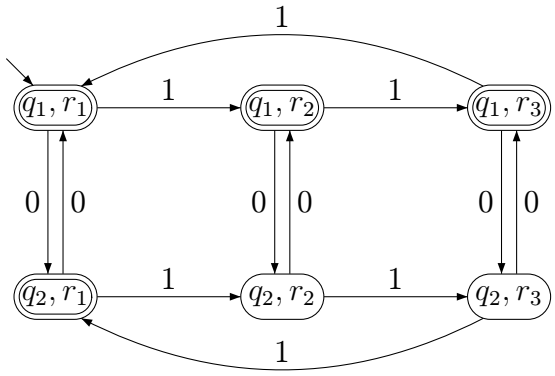
Kontrolní otázka: Jak byste tedy popsali algoritmus, který k libovolným zadaným automatům A_1, A_2 sestrojí automat A tak, že $L(A) = L(A_1) \cup L(A_2)$? Až si to promyslíte, podívejte se dále na to, jak stručně a jasně tento postup sděluje matematická notace v důkazu následující věty.

Věta 2.3
Existuje algoritmus, který k libovolným zadaným automatům A_1, A_2 sestrojí automat A tak, že $L(A) = L(A_1) \cup L(A_2)$.

Důkaz: Nechť $A_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$, $A_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$.

Zkonstruujme automat $A = (Q, \Sigma, \delta, q_0, F)$ tak, že

- $Q = Q_1 \times Q_2$,



Obrázek 2.8:

- $\delta(\,(q_1, q_2), a\,) = (\,\delta_1(q_1, a), \delta_2(q_2, a)\,)$ pro vš. $q_1 \in Q_1$, $q_2 \in Q_2$, $a \in \Sigma$,
- $q_0 = (q_{01}, q_{02})$,
- $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$.

Je očividné (detailně lze ukázat např. indukcí podle délky w), že pro vš. $q_1, q_1' \in Q_1$, $q_2, q_2' \in Q_2$ a $w \in \Sigma^*$ platí

$(q_1, q_2) \xrightarrow{w}_A (q_1', q_2') \iff ((q_1 \xrightarrow{w}_{A_1} q_1') \wedge (q_2 \xrightarrow{w}_{A_2} q_2'))$

Tedy automat A přejde ze stavu (q_1, q_2) přečtením slova w do stavu, jež je dvojicí, která je tvořena stavem, do něhož přejde A_1 ze stavu q_1 přečtením slova w, a stavem, do něhož přejde A_2 ze stavu q_2 přečtením slova w.

Z toho plyne, že $L(A) = \{w \mid (q_{01}, q_{02}) \xrightarrow{w}_A (F_1 \times Q_2) \cup (Q_1 \times F_2)\} = \{w \mid (q_{01} \xrightarrow{w}_{A_1} F_1) \vee (q_{02} \xrightarrow{w}_{A_2} F_2)\}$; máme tedy $L(A) = L(A_1) \cup L(A_2)$. □

Kontrolní otázka: Umíte před dalším čtením zodpovědět, jak byste důkaz věty 2.3 pro sjednocení upravili u analogické věty pro průnik?

Věta 2.4
Existuje algoritmus, který k libovolným zadaným automatům A_1, A_2 sestrojí automat A tak, že $L(A) = L(A_1) \cap L(A_2)$.

(Nápověda: $F = (F_1 \times F_2)$)

Kontrolní otázka: Jak k danému automatu A sestrojíte automat A' přijímající právě ta slova (v abecedě automatu A), která A nepřijímá? (Tedy $L(A')$ je doplněk jazyka $L(A)$.)

Ano, správně, prostě v A prohodíme přijímající a nepřijímající stavy, čímž vznikne A'. (Je-li $A = (Q, \Sigma, \delta, q_0, F)$, pak $A' = (Q, \Sigma, \delta, q_0, Q-F)$.)

Takže můžeme vyvodit:

Věta 2.5
K libovolné booleovské kombinaci $\mathtt{BK}(L_1, L_2, \ldots, L_n)$ *jazyků* $L_1, L_2, \ldots, L_n \subseteq \Sigma^*$ *reprezentovaných automaty* $A_1, A_2, \ldots, A_n$ *(tedy* $L_i = L(A_i)$ *pro* $i = 1, 2, \ldots, n$*) lze (algoritmicky) sestrojit automat* A *tak, že* $L(A) = \mathtt{BK}(L_1, L_2, \ldots, L_n)$.

Kontrolní otázka: Plyne z předchozí věty existence algoritmu, který k automatům A_1, A_2 sestrojí automat A tak, že $L(A) = L(A_1) - L(A_2)$?

Ano, protože $L_1 - L_2 = L_1 \cap \overline{L_2}$. (Jak byste tedy takový A konkrétně sestrojili?)

Cvičení 2.7: Vraťte se k příkladu z části 2.1, kde jsme konstruovali automat pro jazyk

$L = \{ w \in \{0,1\}^* \mid w$ obsahuje podslovo 010 nebo $|w|_1$ je sudé $\}$.

Vytvořte nyní automat modulární konstrukcí a porovnejte s tím, který jste tehdy konstruovali přímo. Mají oba automaty stejný počet stavů? (Pokud ne, zkuste se zamyslet nad tím, proč. Později se k tomu vrátíme.)

Shrnutí: Takže už je nám jasné, že automat pro jazyk, jehož slova jsou charakterizována podmínkou, která je složena z jednodušších podmínek logickými spojkami, nemusíme namáhavě konstruovat přímo, ale můžeme použít modulární přístup. Algoritmy pro „zkombinování“ malých automatů bychom přitom mohli naprogramovat, což by nás zbavilo spousty rutinní práce.

2.4 Dosažitelné stavy, normovaný tvar

Orientační čas ke studiu této části: 1 hod.

Cíle této části:

Máte si uvědomit, že v automatech mohou být (zbytečné) stavy, které nejsou dosažitelné z počátečního stavu. Při návrhu automatů se takové stavy mohou objevit např. rutinním užitím modulárního postupu. Máte zvládnout algoritmus odstranění takových stavů a pochopit souvislost s užitečným pojmem normovaného tvaru automatu.

Klíčová slova: *dosažitelné a nedosažitelné stavy automatu, normovaný tvar automatu*

Není těžké si uvědomit, že jeden a tentýž jazyk může být rozpoznáván více automaty.

Kontrolní otázka: Umíte zdůvodnit, proč ke každému konečnému automatu A existuje nekonečně mnoho konečných automatů rozpoznávajících $L(A)$?

Automaty rozpoznávající tentýž jazyk plní z našeho hlediska tentýž úkol; v tomto smyslu o nich budeme mluvit jako o ekvivalentních:

Definice 2.6
Dva konečné automaty A_1, A_2 jsou (jazykově) *ekvivalentní*, jestliže přijímající shodné jazyky, tedy jestliže platí $L(A_1) = L(A_2)$.

Jedna z triviálních možností, jak k automatu A vyrobit jiný A' ekvivalentní s A je přidat k A nedosažitelný stav.

Definice 2.7
Říkáme, že stav q automatu $A = (Q, \Sigma, \delta, q_0, F)$ je *dosažitelný slovem* w, jestliže $q_0 \xrightarrow{w} q$. Stav q je *dosažitelný*, jestliže je dosažitelný nějakým slovem. (Jedná se tedy o prostou dosažitelnost stavu q ze stavu q_0 v grafu automatu.)

Nedosažitelnými stavy (což jsou pochopitelně ty stavy, které nejsou dosažitelné) nemůže tedy projít žádný výpočet začínající v počátečním stavu. Proto nedosažitelné stavy nemají vliv na přijímaný jazyk (a je samozřejmě lhostejné, zda tyto stavy jsou či nejsou přijímající); přidáváním či odstraňováním takových stavů u automatu A dostáváme automaty ekvivalentní s A.

Z praktického hlediska jsou tedy nedosažitelné stavy zbytečné a je dobré umět se jich zbavovat. Než si ukážeme, jak na to, zamysleme se na chvíli, jak se

vlastně mohou nedosažitelné stavy objevit i v rozumně navrženém automatu. K tomu si zkuste zodpovědět např. následující otázku.

Kontrolní otázka: Uvažujme jazyk

$L = \{w \in \{a,b\}^* \mid w$ začíná znakem a a končí b nebo w začíná znakem b a končí $a\}$.

Zamyslete se nad konstrukcí automatu pro L použitím modulární konstrukce pro sjednocení jazyků. Napadá vás příklad stavu (tedy příslušné dvojice) ve výsledném automatu, který je nedosažitelný?

Poté, co jsme odhalili alespoň jeden zdroj vzniku nedosažitelných stavů v návrhu automatu, podíváme se na možnost mechanického (tedy algoritmického) odstranění takových zbytečných stavů. Zmínka o dosažitelnosti v grafu automatu už možná evokovala v čtenářově mysli přímočarý algoritmus nalézající všechny dosažitelné (a tím pádem i všechny nedosažitelné) stavy v zadaném automatu.

My si ten algoritmus (tedy variantu prohledávání grafu do šířky) připomeneme ve formě, která nejenom zjistí všechny dosažitelné stavy, ale také automat převede do tzv. *normovaného tvaru*. Algoritmus popíšeme a předvedeme na konkrétním příkladu.

Metoda 2.8
Převod konečného automatu do normovaného tvaru

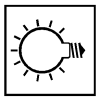
je realizován následujícím algoritmem (který zjistí všechny dosažitelné stavy a ty systematicky seřadí, tedy očísluje).

Nejprve předpokládáme abecedu $\{a,b\}$, pak zobecníme.

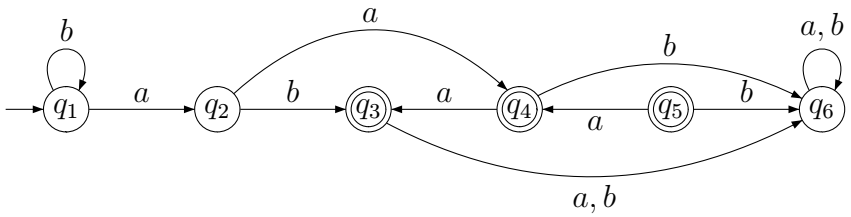
- Na začátku jsou všechny stavy „neoznačené“ a „nezpracované“.
- Počáteční stav označíme 1.
- Dále zjistíme stav q, do něhož automat přejde ze stavu 1 symbolem a; když q není označen, označíme jej 2.
- Pak zjistíme stav q, do něhož automat přejde ze stavu 1 symbolem b; když q není dosud označen, označíme jej nejmenším dosud nepoužitým číslem.
- Takto jsme zpracovali stav 1, pokračujeme teď zpracováním stavu 2 atd..., až budou všechny označené stavy zpracovány.

Můžete se rovnou podívat na následující řešený příklad, a pak se teprve vrátit k následujícímu obecnějšímu popisu. V něm se předpokládá obecná abeceda $\Sigma = \{a_1, a_2, \ldots, a_m\}$, jejíž prvky jsou (úplně) uspořádány; např. $a_1 < a_2 < \ldots < a_m$.

- Na začátku jsou všechny stavy „neoznačené“ a „nezpracované“.
- Počáteční stav označíme číslem 1.
- Dokud nejsou všechny označené stavy zpracované, opakujeme tuto činnost:
  - Vezmeme nejnižším číslem označený nezpracovaný stav q.
  - Postupně pro $j = 1, 2, \ldots, m$ děláme toto:
    jestliže stav q', pro nějž platí $q \xrightarrow{a_j} q'$, není označený,
    označíme ho nejmenším dosud nepoužitým číslem.
  - Stav q prohlásíme za zpracovaný.
- Zbyly-li neoznačené stavy (ty jsou nedosažitelné), odstraníme je. (Stavy odstraníme samozřejmě spolu se „šipkami“ z nich vycházejícími. Proč nemůže ve zbylém grafu automatu zůstat „viset“ nějaká šipka bez cíle?).



Řešený příklad 2.2: Následující automat převeďte do normovaného tvaru.



Řešení: Podle Metody 2.8 označíme stav q_1 číslem 1; stav q_1 je teď jediný označený stav, který je dosud nezpracovaný. Znakem a z q_1 přejdeme do dosud neoznačeného stavu q_2, který tedy označíme číslem 2. Znakem b z q_1 zůstaneme v již označeném stavu q_1. Tím jsme zpracovali q_1.

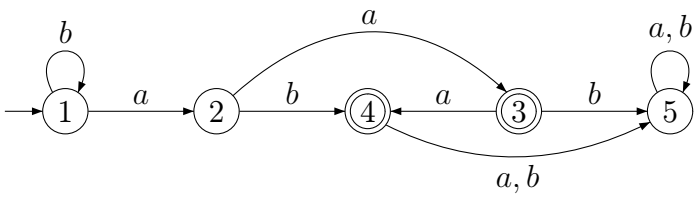
Teď zpracujeme q_2, jelikož je označeným dosud nezpracovaným stavem s nejmenším přiřazeným číslem. Znakem a přejdeme z q_2 do stavu q_4, který

není označen, a kterému tedy přiřadíme nejmenší dosud nepoužité číslo, tj. 3; znakem b přejdeme z q_2 do (neoznačeného) stavu q_3, který označíme číslem 4. Tím jsme zpracovali q_2.

Teď je na řadě ke zpracování stav q_4, jelikož mezi označenými dosud nezpracovanými stavy má nejmenší přiřazené číslo. Z q_4 se přes a dostaneme do již označeného stavu q_3 a přes b do neoznačeného stavu q_6, který tak dostane číslo 5.

Další ke zpracování je q_3 (označený číslem 4); u něj oba přechody vedou do již označeného stavu, takže jeho zpracování nová označení nepřinese. Totéž platí pro stav q_6 (označený číslem 5). Po jeho zpracování tedy již žádný označený nezpracovaný stav nezbývá, takže konstrukce končí.

Stavy, které vůbec nebyly označeny, jsou nedosažitelné; v našem případě se jedná pouze o stav q_5. Výsledný normovaný tvar automatu je tedy tento:



Je jistě jasné, že když mluvíme o normovaném tvaru, nemáme na mysli způsob nakreslení grafu automatu, ale jen očíslování jeho stavů. Když v odpovídající tabulce automatu seřadíme řádky podle vzrůstajících čísel stavů, je její tvar jednoznačný. U našeho příkladu tak dostáváme následující tabulku.

	a	b
$\rightarrow 1$	2	1
2	3	4
$\leftarrow$ 3	4	5
$\leftarrow$ 4	5	5
5	5	5

Shrňme teď naše poznatky.

Tvrzení 2.9
Existuje algoritmus, který v daném konečném automatu A zjistí všechny dosažitelné stavy; odstraněním nedosažitelných stavů vznikne z A jemu ekvivalentní automat A', v němž je již každý stav dosažitelný.

Matematická poznámka: Je-li u automatu $A = (Q, \Sigma, \delta, q_0, F)$ množinou dosažitelných stavů $D \subseteq Q$, pak automat vzniklý z A odstraněním nedosažitelných stavů je $A' = (D, \Sigma, \delta', q_0, F \cap D)$, kde δ' je restrikcí (zúžením) funkce δ na definiční obor $D \times \Sigma$; hodnoty funkce δ' jsou jistě v D (proč?).

Z tvrzení 2.9 mj. snadno nahlédneme následující větu.

Věta 2.10
Existuje algoritmus, který pro zadaný konečný automat A rozhodne, zda $L(A)$ je neprázdný.

Kontrolní otázka: Jak vypadá ten algoritmus?

(Jistě jste přišli na to, že stačí zjistit, zda mezi dosažitelnými stavy je alespoň jeden přijímající stav.)

Ještě si všimněme jedné užitečné věci, kterou nám převod do normovaného tvaru také poskytuje. Převodem do normovaného tvaru totiž rychle zjistíme, zda jsou dva automaty bez nedosažitelných stavů de facto stejné, tj. jeden z druhého se dá dostat přejmenováním stavů. (Když mají stejný normovaný tvar [stejnou tabulku], tak to lze; když ne, tak to nelze. Samozřejmě oba normované tvary konstruujeme pro stejné uspořádání jejich společné abecedy.)

Poznámka: V rozšiřující části se k této otázce vrátíme. Připomeneme si mj. související pojem izomorfismu automatů.

Shrnutí: Takže je nám jasné, jak se rutinně zbavit nedosažitelných stavů u automatu; můžeme samozřejmě naprogramovat algoritmus, který to udělá za nás. Navíc jsme si vědomi, že ten algoritmus můžeme provádět formou převodu do normovaného tvaru, což je výhodné zvláště pro porovnání různých automatů.

2.5 Cvičení

Orientační čas ke studiu této části: 2 hod.

Cíle této části:

Cílem je jistá rekapitulace dosud nabytých znalostí o konečných automatech; k tomu má napomoci promyšlení a vyřešení dalších otázek a příkladů.

Otázky:

Otázka 2.8: Může být počáteční stav automatu zároveň přijímajícím? A co by to znamenalo pro přijímaná slova?

Otázka 2.9: Může se stát, že konečný automat nepřijímá žádné slovo?

Otázka 2.10: Je normovaný tvar automatu určený jednoznačně?

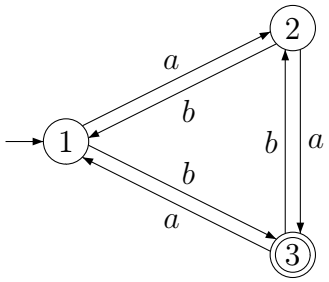
Cvičení 2.11: Existuje konečný automat se dvěma stavy rozpoznávající jazyk všech těch neprázdných slov nad abecedou $\{a, b, c\}$, která obsahují alespoň jeden znak a? Pokud ano, příslušný automat nakreslete.

Cvičení 2.12: Existuje konečný automat se třemi stavy rozpoznávající jazyk všech těch neprázdných slov nad abecedou $\{a, b, c\}$, která neobsahují znak a? Pokud ano, příslušný automat zde nakreslete. (Nezapomeňte, že přijímaná slova mají být neprázdná.)

Cvičení 2.13: Navrhněte konečný automat přijímající všechna ta slova nad abecedou $\{a, b\}$, která obsahují lichý počet výskytů a.

Cvičení 2.14: Navrhněte konečný automat přijímající všechna ta slova nad abecedou $\{a\}$, jejichž délka dává zbytek 2 po dělení 3.

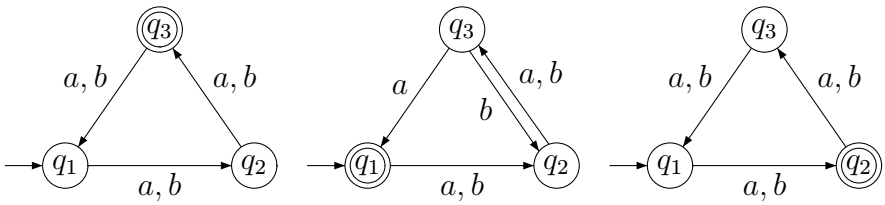
Cvičení 2.15: Jaká všechna slova přijímá automat na Obrázku 2.9?



Obrázek 2.9:

Cvičení 2.16: Jaká všechna slova přijímá automat z Řešeného příkladu 2.2?

Cvičení 2.17: Které z těchto tří automatů nad abecedou $\{a, b\}$ přijímají nějaké slovo délky přesně 100?



Cvičení 2.18: Navrhněte konečný automat přijímající právě všechna slova nad $\{a, b\}$, ve kterých je třetí znak stejný jako první.

Cvičení 2.19: Navrhněte konečný automat přijímající právě všechna slova nad $\{a, b, c\}$, ve kterých se první znak ještě aspoň jednou zopakuje.

Cvičení 2.20: Navrhněte automat rozpoznávající všechna ta slova nad $\{a, b\}$, která začínají znakem a a končí znakem b.

Cvičení 2.21: Umíte navrhnout konečný automat rozpoznávající jazyk všech slov nad abecedou $\{a, b\}$, ve kterých je součin počtů výskytů znaků a a b sudý?

Cvičení 2.22: Umíte navrhnout konečný automat rozpoznávající jazyk všech slov nad abecedou $\{a, b\}$, ve kterých je součet počtů výskytů znaků a a b sudý?

Cvičení 2.23: Umíte navrhnout konečný automat rozpoznávající jazyk všech slov nad abecedou $\{a, b\}$, ve kterých je součet počtů výskytů znaků a a b větší než 100?

Cvičení 2.24: Navrhněte konečný automat přijímající právě ta slova nad abecedou $\{a, b, c, d\}$, která nemají jako první znak a, nemají jako druhý znak b, nemají jako třetí znak c a nemají jako čtvrtý znak d. (Jejich délka může být pochopitelně i < 4.)

Cvičení 2.25: Navrhněte konečný automat přijímající právě ta slova nad abecedou $\{a, b, c, d\}$, která nezačínají a nebo nemají druhý znak b nebo nemají třetí znak c nebo nemají čtvrtý znak d.

Cvičení 2.26: Sestrojte konečný automat přijímající všechna ta slova délky aspoň 4 nad abecedou $\{a, b\}$:

a) ve kterých jsou druhý, třetí a čtvrtý znak stejné,

b) ve kterých jsou třetí a poslední znak stejné.

Cvičení 2.27: Sestrojte konečný automat přijímající všechna ta slova délky aspoň 2 nad abecedou $\{a, b\}$, ve kterých nejsou poslední dva znaky stejné.

Cvičení 2.28: Navrhněte konečný automat rozpoznávající jazyk $L_1 = \{ w \in \{a, b\}^* \mid w$ obsahuje podslovo $aba \}$ a konečný automat rozpoznávající jazyk $L_2 = \{ w \in \{a, b\}^* \mid |w|_b \bmod 2 = 0 \}$ (v L_2 jsou tedy právě slova obsahující sudý počet b-ček). Pak zkonstruujte automat rozpoznávající jazyk $L_1 \cap L_2$.

Cvičení 2.29: Na vybraných zkonstruovaných automatech si procvičte převod do normovaného tvaru.

2.6 Minimalizace konečných automatů

Orientační čas ke studiu této části: 2 hod.

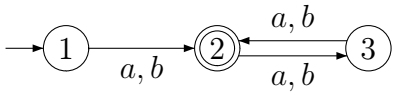
Cíle této části:

Prostudováním této části máte zvládnout algoritmus minimalizace konečného automatu na základě postupu zjišťujícího, které stavy daného automatu jsou ekvivalentní.

Klíčová slova: *redukovaný konečný automat, ekvivalentní stavy, algoritmus minimalizace konečných automatů*

Již jsme si dříve uvědomili, že ke každému konečnému automatu A existuje nekonečně mnoho ekvivalentních automatů (tedy automatů rozpoznávajících $L(A)$). Tyto automaty mohou např. obsahovat spoustu nedosažitelných stavů; takové stavy ovšem umíme algoritmicky odstraňovat. Mohou být ale i některé dosažitelné stavy „nadbytečné“?

Podívejme se například na následující automat.



Když se díváme pozorně, zjistíme brzy, že stavy 1 a 3 plní vlastně stejný úkol, tedy $L_1^{toAcc} = L_3^{toAcc}$. (Ze stavu 1 převedou automat do přijímajícího stavu přesně ta slova, která ho tam převedou ze stavu 3; mimochodem, jak byste charakterizovali ta slova?)

Vzpomeneme-li si na to, že na stavy se lze dívat jako na vzájemně se volající rekurzivní procedury (jak jsme to probírali v části 2.1), znamená to, že máme de facto dva exempláře jedné procedury. Takový „program“ se pochopitelně dá zjednodušit tak, že ponecháme jen jeden exemplář a místo každého volání druhého voláme ten jeden ponechaný.

V řeči grafu našeho automatu: jeden ze stavů 1,3 vypustíme, spolu s šipkami z něj vycházejícími, a šipky do něj vedoucí nasměrujeme do toho ponechaného

stavu. Pokud se rozhodneme vypustit stav, který je počáteční, musíme pochopitelně jako počáteční prohlásit ten ponechaný. (Promyslete si obě úpravy: jak při vypuštění stavu 1, tak při vypuštění stavu 3.)

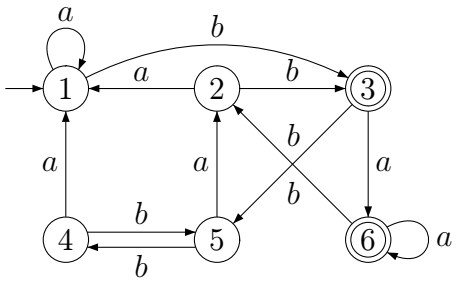
Úvaha provedená na našem elementárním příkladu má ovšem obecnou platnost: Analogickou úpravu pochopitelně můžeme provést v každém automatu pro stavy q, q' takové, že $L_q^{toAcc} = L_{q'}^{toAcc}$; rozpoznávaný jazyk se nezmění! Z toho plyne, že ke každému automatu existuje ekvivalentní automat, který nazveme *redukovaný*:

Definice 2.11
Konečný automat je *redukovaný*, jestliže v něm pro každé dva různé stavy q, q' platí $L_q^{toAcc} \neq L_{q'}^{toAcc}$.

Kontrolní otázka: Proč z předchozích úvah plyne, že ke každému automatu existuje ekvivalentní redukovaný automat?

Doufejme, že prostá existence ekvivalentního redukovaného automatu je jasná, ale z toho neplyne, že takový automat umíme (algoritmicky) sestrojit. Potřebujeme umět u obecného automatu zjišťovat, zda pro dva stavy q, q' platí $L_q^{toAcc} = L_{q'}^{toAcc}$; jak to udělat? Ukážeme si teď na to rychlý systematický algoritmus, který tu otázku zodpoví naráz pro všechny dvojice stavů. Algoritmus popíšeme víceméně slovně a budeme jej hned ilustrovat na konkrétním příkladu, tj. následujícím automatu.

Poznámka: O elegantní matematické formulaci algoritmu je pojednáno v rozšiřující části.



Začneme jednoduchou úvahou.

Kontrolní otázka: Je možné, aby platilo $L_q^{toAcc} = L_{q'}^{toAcc}$, když jeden ze stavů q, q' je přijímající a jeden ne ?

Jistěže to není možné; jeden z jazyků obsahuje ε a druhý ne. Jinými slovy, takové dva stavy rozlišíme již slovem délky 0. Množina stavů, v našem případě $\{1,2,3,4,5,6\}$ se nám tedy rozpadá na dvě (disjunktní) části $\{1,2,4,5\}$ (nepřijímající stavy) a $\{3,6\}$ (přijímající stavy). Vytvořili jsme tedy rozklad na množině $Q = \{1,2,3,4,5,6\}$, konkrétně

$$R_0\text{:}\quad \mathrm{I} = \{1,2,4,5\},\ \ \mathrm{II} = \{3,6\}$$

tak, že dva stavy q, q' jsou v téže třídě rozkladu právě tehdy, když se jazyky $L_q^{toAcc}, L_{q'}^{toAcc}$ shodují na slovech délky 0. (Tedy když buď oba obsahují ε, nebo je neobsahuje ani jeden z nich.) Rozklad má dvě třídy, které jsme označili I, II.

Jak teď zjistíme analogický rozklad R_1 podle slov délky nejvýše 1 ? Zkusme si napsat tabulku automatu, s tím, že místo konkrétních stavů píšeme do políček tabulky jen označení příslušné třídy rozkladu R_0 a řádky tabulky přitom seskupíme podle tříd R_0.

	a	b
1	I	II
2	I	II
4	I	I
5	I	I
3	II	I
6	II	I

Z tabulky je ihned vidět, že např. stavy 2 a 4 (které nelze rozlišit slovem délky 0) lze rozlišit slovem délky 1; tedy L_2^{toAcc}, L_4^{toAcc} se neshodují (již) na slovech délky nejvýš 1 – konkrétně $b \in L_2^{toAcc}$ a $b \notin L_4^{toAcc}$.

Není teď jistě těžké nahlédnout, že jazyky $L_q^{toAcc}, L_{q'}^{toAcc}$ se shodují na slovech délky nejvýše 1 právě tehdy, když q, q' patří do stejné třídy rozkladu R_0 a hodnoty jejich řádků ve výše uvedené tabulce jsou stejné.

Rozklad R_1 (který je zjemněním rozkladu R_0) má tedy následující třídy (třída $\{1,2,3,4\}$ se rozpadla na dvě části):

$$R_1\text{:}\quad \mathrm{I} = \{1,2\},\ \ \mathrm{II} = \{4,5\},\ \ \mathrm{III} = \{3,6\}\,.$$

Rozklad R_2 (odpovídající rozlišení podle slov délky nejvýše 2) dostaneme z R_1 analogicky. Sestrojíme tedy následující tabulku

	a	b
1	I	III
2	I	III
4	I	II
5	I	II
3	III	II
6	III	I

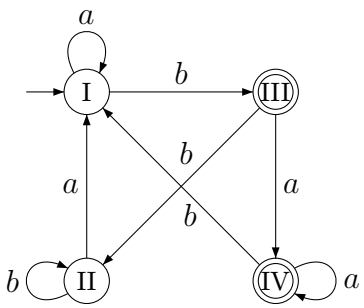
a vidíme, že

$$R_2\colon\ \ \mathrm{I} = \{1,2\},\ \ \mathrm{II} = \{4,5\},\ \ \mathrm{III} = \{3\},\ \ \mathrm{IV} = \{6\}\,.$$

Stavy 3,6 sice nejdou rozlišit slovy délky nejvýše 1, ale znak b je převede do dvojice stavů, které patří do různých tříd rozkladu R_1 a které lze tedy rozlišit nějakým slovem u délky (nejvýš) 1. Stavy 3,6 lze tedy rozlišit slovem bu, které má délku (nejvýš) 2.

Když provedeme analogickou konstrukci pro R_3, zjistíme, že žádná třída se již dále nerozpadne, tedy $R_3 = R_2$. (Ověřte si to!) Pak ovšem $R_2 = R_3 = R_4 = \ldots$; dosáhli jsme „pevného bodu“ naší operace a a naše konstrukce tak vydá jako finální rozklad R_2. Tedy když se v našem automatu jazyky $L_q^{toAcc}, L_{q'}^{toAcc}$ liší, pak se liší již na slovech délky nejvýše dvě!

Z finálního rozkladu snadno zjistíme pro každou dvojici q, q', zda $L_q^{toAcc} = L_{q'}^{toAcc}$. Nemusíme ale teď upravovat (zmenšovat) automat pro každou takovou dvojici postupně (vypuštěním jednoho členu dvojice a ponecháním druhého). Z každé třídy (finálního rozkladu, v našem případě R_2) stačí prostě ponechat jediného zástupce a všechny šipky směrované do této třídy nasměrujeme na onoho zástupce. Počáteční je samozřejmě zástupce třídy obsahující původní počáteční stav. Zástupce třídy je přijímající právě tehdy, když je třída složena z přijímajících stavů. (Uvědomme si, že už v rozkladu R_0 jsou u každé třídy buď všechny prvky přijímající nebo všechny prvky nepřijímající; to pak musí nutně platit u každého jemnějšího rozkladu.) Výsledný automat je na obrázku 2.10. Jelikož na tom, kterého konkrétního zástupce ponecháme, nezáleží, můžeme stavy výsledného automatu označit přímo symboly označující jednotlivé třídy.



Obrázek 2.10: Výsledný automat po provedení redukce

Zrekapitulujme: podali jsme obecný návod (tedy popsali jsme algoritmus), který k zadanému automatu sestrojí ekvivalentní redukovaný automat (zjištěním a sloučením stavů plnících týchž úkol, tedy majících stejný jazyk L_q^{toAcc}). Ukázali jsme tedy následující tvrzení.

Tvrzení 2.12
Existuje algoritmus, nazvěme ho algoritmem redukce*, který k zadanému konečnému automatu A sestrojí ekvivalentní redukovaný automat A'.*

Připomeňme si, že takto máme dvě metody možného zmenšování počtu stavů automatu při zachování rozpoznávaného jazyka:

- odstranění nedosažitelných stavů,
- redukce automatu.

Kontrolní otázka: Je jistě jasné, že když daný automat zredukujeme a pak v něm odstraníme nedosažitelné stavy, tak výsledný automat nebude mít nedosažitelné stavy a bude redukovaný. Proč?

Ano, jistě jste si uvědomili, že odstranění nedosažitelných stavů nemůže samozřejmě pokazit podmínku, že pro dva různé stavy q, q' platí $L_q^{toAcc} \neq L_{q'}^{toAcc}$. Mírně složitější je si uvědomit, že postup lze i obrátit: když nejprve odstraníme nedosažitelné stavy a pak zredukujeme, tak tou redukcí už nemohou nedosažitelné stavy vzniknout.

Existuje ještě nějaký další postup, který by dokázal zmenšit redukovaný automat bez nedosažitelných stavů (při zachování rozpoznávaného jazyka)? Ne,

neexistuje; takový automat je *minimální*, což znamená, že k němu neexistuje ekvivalentní automat s menším počtem stavů. Ale o tom si povíme více v další části.

Teď si ještě alespoň uvědomíme, že porovnání jazyků $L_q^{toAcc}, L_{q'}^{toAcc}$ se samozřejmě nemusí omezovat na stavy q, q' v jednom automatu. Algoritmus redukce ve fázi rozkládání nijak nezávisí na tom, který stav je počáteční. (Ten nám slouží až nakonec k označení příslušné třídy finálního rozkladu jako počáteční.) Proveďte si tedy následující cvičení:

CVIČENÍ 2.30: Zjistěte všechny dvojice stavů q, q' u následujících dvou automatů (tedy $q, q' \in \{0, 1, 2, \ldots, 9\}$), pro něž platí $L_q^{toAcc} = L_{q'}^{toAcc}$. (Nápověda: Označení počátečních stavů ignorujme. Pak slučme obě tabulky do jediné; kdyby v nich stavy nebyly pojmenovány různě, museli bychom nejprve v jedné tabulce stavy přejmenovat. No a pak konstruujme podle výsledné tabulky rozklady $R_0, R_1, R_2, \ldots$.)

	a	b
→0	0	1
←1	1	2
←2	3	1
3	2	4
4	2	3

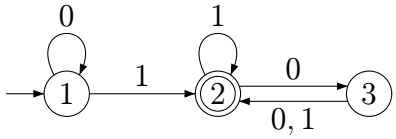
	a	b
→5	5	6
6	7	5
←7	7	9
8	9	8
←9	8	7

Otázky:

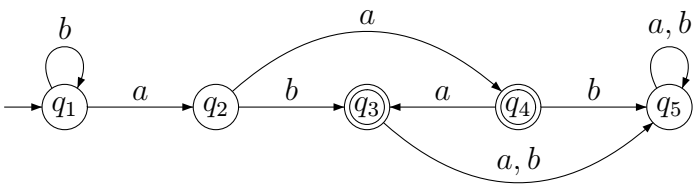
OTÁZKA 2.31: Proč v automatu na Obrázku 2.10 vznikla u stavu II smyčka?

OTÁZKA 2.32: Pokud je zadaný automat již minimální, ukáže se to v postupu minimalizace hned?

CVIČENÍ 2.33: Je tento automat minimální?



Cvičení 2.34: Je tento automat minimální?

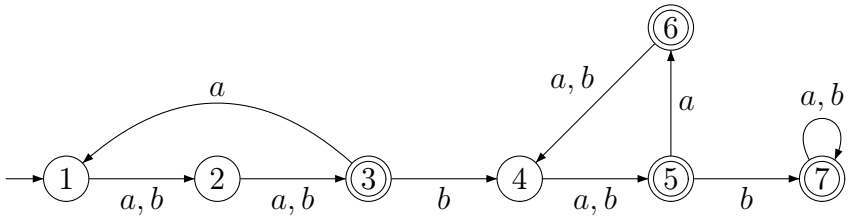


Cvičení 2.35: Najděte minimální automat ekvivalentní s následujícím automatem zadaným tabulkou.

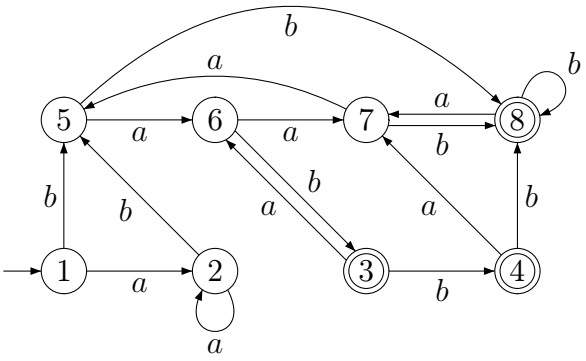
	a	b
→1	2	3
2	2	4
←3	3	5
4	2	7
←5	6	3
←6	6	6
7	7	4
8	2	3
9	9	4

Cvičení 2.36: Nechť L je jazyk všech těch slov nad abecedou $\{a, b\}$, která obsahují lichý počet výskytů znaku a a sudý počet výskytů znaku b. Jaký nejmenší možný počet stavů má konečný deterministický automat rozpoznávající jazyk L?

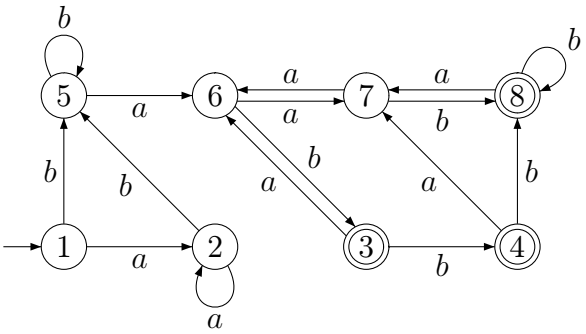
Cvičení 2.37: Zdůvodněte minimalitu tohoto automatu; pro každou dvojici stavů q, q' najděte slovo které patří jen do jednoho z jazyků $L_q^{toAcc}, L_{q'}^{toAcc}$.



Cvičení 2.38: Minimalizujte následující automat:



Cvičení 2.39: Minimalizujte následující automat:



Cvičení 2.40: Nechť L je jazyk všech těch *neprázdných* slov nad abecedou $\{a, b\}$, která obsahují sudý počet výskytů znaku a nebo sudý počet výskytů znaku b. Jaký nejmenší možný počet stavů má konečný deterministický automat rozpoznávající jazyk L a proč?

Cvičení 2.41: Nechť L je jazyk všech těch slov nad abecedou $\{a, b, c\}$, která obsahují alespoň dva výskyty znaku a a méně než dva výskyty znaku b. Jaký nejmenší možný počet stavů má konečný deterministický automat rozpoznávající jazyk L a proč?

Cvičení 2.42: Nechť L je jazyk všech těch slov nad abecedou $\{a, b, c\}$, která obsahují alespoň dva výskyty znaku a nebo alespoň dva výskyty znaku b.

Jaký nejmenší možný počet stavů má konečný deterministický automat rozpoznávající jazyk L a proč?

Cvičení 2.43: Nechť L je jazyk všech těch slov nad abecedou $\{a, b, c\}$, která obsahují alespoň dva výskyty znaku a a alespoň dva výskyty znaku b. Jaký nejmenší možný počet stavů má konečný deterministický automat rozpoznávající jazyk L?

Shrnutí: Jistě už teď máme dobrou intuitivní představu o tom, co je to minimální automat (tuto představu brzy zpřesníme), známe algoritmus minimalizace a díky vyřešeným příkladům to vše máme dobře „zažito“.

2.7 Ekvivalence konečných automatů; minimální automaty

Orientační čas ke studiu této části: 1 hod.

Cíle této části:

V této části si uvědomíte, že již de facto znáte (rychlé) algoritmy, které rozhodují zda zadané konečné automaty jsou (jazykově) ekvivalentní. Rovněž se seznámíte s přesnější definicí pojmu minimálního automatu, který jste již intuitivně pochopili.

Klíčová slova: *jazyková ekvivalence, minimální konečný automat*

Všimněme si, že už jsme vlastně ukázali následující důležitou větu.

Poznámka: V této větě, podobně jako jinde, hovoříme o existenci jistého algoritmu. Tu existenci pochopitelně prokazujeme tak, že příslušný algoritmus prostě popíšeme (a jeho korektnost ukážeme, pokud není očividná). Občas v této souvislosti zmíníme pojem „rychlý algoritmus“. Tento pojem zpřesníme v úvodu do teorie algoritmů, kde je slovo „rychlý algoritmus“ obvykle synonymem pro algoritmus s polynomiální časovou složitostí.

Věta 2.13
Existuje (rychlý) algoritmus, který pro zadané konečné automaty A_1, A_2 rozhodne, zda jsou ekvivalentní (tedy zda $L(A_1) = L(A_2)$).

Důkaz: Stačí si uvědomit, že $L(A_1) = L(A_2)$ právě tehdy, když $L_{q_{01}}^{toAcc} = L_{q_{02}}^{toAcc}$, kde q_{01} je počáteční stav A_1 a q_{02} počáteční stav A_2. Ze studia algoritmu redukce víme, že rozklad na třídy obsahující stavy se „stejnými úkoly" (tedy jazyky L_q^{toAcc}) lze provádět současně na sjednocení množin stavů obou automatů. (Měli bychom říci na „disjunktním sjednocení", čímž se myslí, že ty množiny stavů jsou disjunktní – či jsou jejich prvky přejmenovány tak, aby disjunktní byly.) Stačí tedy ve finálním rozkladu zjistit, zda q_{01} a q_{02} patří do stejné třídy či nikoliv. □

Zamyslíme-li se nad tímto problémem hlouběji, uvědomíme si, že dokázat uvedenou větu ve skutečnosti umíme i bez algoritmu redukce:

Alternativní důkaz předchozí věty. Jelikož $L(A_1) = L(A_2) \iff (L(A_1) - L(A_2)) \cup (L(A_2) - L(A_1)) = \emptyset$, je existence příslušného algoritmu jasná z věty 2.5 a věty 2.10.

Kontrolní otázka: Je vám jasné proč? Pokud ne, měli byste si příslušné věty znovu důkladně promyslet.

Otázka ekvivalence automatů rovněž úzce souvisí s pojmem minimálního automatu. Tohoto pojmu jsme se již dotkli, stejně jako následující věty.

Definice 2.14
Konečný automat A je *minimální*, jestliže neexistuje automat A', který je ekvivalentní s A (pro nějž je tedy $L(A) = L(A')$) a který má méně stavů než A.

Věta 2.15
Je-li automat A redukovaný a nemá nedosažitelné stavy, pak je minimální.

Kontrolní otázka: Platí i obrácená implikace?

(Samozřejmě. Když je automat minimální, nemůže jej algoritmus redukce ani algoritmus odstranění nedosažitelných stavů zmenšit.)

Uvedené poznatky doplňuje ještě následující věta.

Věta 2.16
Dva minimální automaty jsou ekvivalentní právě tehdy, když jsou „stejné až na pojmenování stavů", což znamená, že mají stejný normovaný tvar.

Poznámka: Už jsme se zmiňovali o tom, že převod do normovaného tvaru je rychlou metodou zjištění izomorfismu automatů (tj. oné stejnosti až na pojmenování stavů). K tomu se vrátíme podrobněji v rozšiřující části.

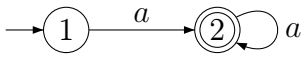
Teď se spokojíme jen s intuitivním pochopením uvedených vět. Vrátíme se k nim ještě později a precizně je dokážeme až v rozšiřující části. Teď si ještě všimneme souvislosti s ekvivalencí automatů.

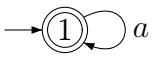
Další důkaz věty 2.13. Oba automaty lze zminimalizovat algoritmem odstranění nedosažitelných stavů a algoritmem redukce a převést je do normovaného tvaru. Když jsou oba výsledky (obě tabulky) shodné, platí $L(A_1) = L(A_2)$, v opačném případě $L(A_1) \neq L(A_2)$.

Otázky:

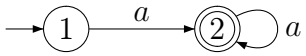
Otázka 2.44: Mohou k danému jazyku L existovat dva neizomorfní minimální automaty, tedy dva minimální automaty, jejichž normované tvary jsou různé, které přijímají jazyk L ?

Cvičení 2.45: Jsou tyto dva automaty nad abecedou $\{a\}$ ekvivalentní?





Cvičení 2.46: Jsou tyto dva automaty nad abecedou $\{a\}$ ekvivalentní?



Shrnutí: Máme tedy dobře promyšleno několik způsobů, jak lze rychle algoritmicky rozhodovat ekvivalenci konečných automatů. Zároveň jsme si upřesnili intuitivní pochopení pojmu minimálního automatu.

2.8 Regulární a neregulární jazyky

Orientační čas ke studiu této části: 2 hod.

Cíle této části:

Měli byste pochopit charakterizaci regulárních jazyků, tj. jazyků rozpoznatelných konečnými automaty, alespoň do té míry, která umožňuje rychlé rozhodování, zda zadaný jazyk je či není regulární.

Klíčová slova: *regulární a neregulární jazyky, dokazování neregularity jazyků*

Připomeňme si, že zobecnění naší konstrukce konečných automatů v části 2.1 se dá abstraktně vyjádřit následovně:

Máme-li konstruovat konečný automat rozpoznávající jazyk L nad abecedou $\Sigma = \{a, b\}$, de facto zkoumáme postupně kvocienty $\varepsilon\backslash L$, $a\backslash L$, $b\backslash L$, $a\backslash(a\backslash L) = aa\backslash L$, $b\backslash(a\backslash L) = ab\backslash L$, $a\backslash(b\backslash L) = ba\backslash L$, $b\backslash(b\backslash L) = bb\backslash L$, $aaa\backslash L$, $aab\backslash L$, $aba\backslash L$, ...

Začínáme tedy jazykem $\varepsilon\backslash L = L$; označíme jej L_0 a přiřadíme mu počáteční stav, označený např. q_0. (Jazyk L_0 má být ve výsledném automatu roven $L_{q_0}^{toAcc}$). Jazyk L_0 (a stav q_0) je teď jediný jazyk (stav), který je označený, ale nezpracovaný. Zpracujeme ho tak, že prozkoumáme kvocienty $a\backslash L_0$ a $b\backslash L_0$.

Pokud $a\backslash L_0 \neq L_0$, označíme tento jazyk L_1, přiřadíme mu nový stav q_1 a položíme $\delta(q_0, a) = q_1$ (tedy uděláme šipku $q_0 \xrightarrow{a} q_1$); pokud $a\backslash L_0 = L_0$, uděláme smyčku $\delta(q_0, a) = q_0$ ($q_0 \xrightarrow{a} q_0$). Pokud se nyní $b\backslash L_0$ nerovná některému již označenému jazyku, označíme jej L_i pro nejmenší zatím nepoužité i, přiřadíme mu nový stav q_i a uděláme šipku $q_0 \xrightarrow{b} q_i$; pokud se $b\backslash L_0$ rovná jazyku již dříve označenému L_j, uděláme šipku $q_0 \xrightarrow{b} q_j$. Tím jsme zpracovali jazyk L_0 (a stav q_0). Pokud teď máme jazyky, které jsou označené ale nezpracované, vezmeme takový L_i s nejnižším pořadovým číslem i a zpracujeme ho podobně jako L_0. (Zkoumáme tedy nejprve, zda $a\backslash L_i$ se rovná některému již označenému jazyku, atd.)

Takto pokračujeme, dokud nejsou všechny označené jazyky (stavy) zpracovány. Počátečním stavem je tedy q_0 a jako přijímající prohlásíme každý stav

q_i, pro nějž $\varepsilon \in L_i$. Tato konstrukce očividně zaručuje, že ve výsledném automatu pro každý stav q_i platí $L_{q_i}^{toAcc} = L_i$.

Kontrolní otázka: Jak zobecníte tuto abstraktní konstrukci pro libovolnou abecedu Σ ?

Jistě jste si vzpomněli na postup při převodu do normovaného tvaru (při němž zároveň zjišťujeme všechny dosažitelné stavy). Při zpracování stavu prostě postupně probereme všechny prvky abecedy Σ v pevně daném pořadí.

Poznámka: Znovu si uvědomme, že uvedená konstrukce je skutečně (jen) abstraktní, nemůžeme ji realizovat algoritmem. Charakterizování kvocientů a hlavně zjišťování, zda jsme aktuálně popsali jazyk, který se nerovná dosud vytvořeným (označeným) jazykům, se algoritmickým postupům vymyká – vyžaduje to náš „lidský“ kreativní přístup.

Poznámka: Přinejmenším intuitivně cítíme, že popsaná abstraktní konstrukce nutně vede k minimálnímu automatu pro daný jazyk (pokud vůbec skončí). K tomu se ale vrátíme až v rozšiřující části.

Při promýšlení výše uvedeného postupu nás jistě napadne, že tento proces nemusí pro nějaký konkrétní jazyk L nikdy skončit. Ano, očividně neskončí právě tehdy, když je množina jazyků

$$\{w\backslash L \mid w \in \Sigma^*\}$$

nekonečná. Může se to stát? Jistě může, vezměme si například relativně jednoduchý jazyk

$$L = \{a^n b^n \mid n \geq 0\}$$

(kde každé slovo začíná úsekem a-ček, za nímž následuje stejně dlouhý úsek b-ček). Ihned vytušíme, že např. kvocienty $a\backslash L$, $aa\backslash L$, $aaa\backslash L$, ... nemohou být stejné. Exaktně je to ukázáno např. tím, že pro jakékoli $i \neq j$ máme $b^i \in a^i\backslash L$ a $b^i \notin a^j\backslash L$; tedy pro $i \neq j$ jsou jazyky $a^i\backslash L$ a $a^j\backslash L$ jistě různé.

Jistě teď už intuitivně chápeme, že jazyk $\{a^n b^n \mid n \geq 0\}$ nelze rozpoznat žádným konečným automatem. A nijak nás teď nepřekvapí následující obecná věta, charakterizující jazyky rozpoznatelné konečnými automaty; říkáme jim regulární jazyky.

Definice 2.17
Jazyk $L \subseteq \Sigma^*$ nazveme *regulární*, jestliže jej lze rozpoznat konečným automatem (s abecedou Σ), tj. existuje-li konečný automat A takový, že $L = L(A)$.

Poznámka: Nepleťme si zatím regulární jazyky s regulárními výrazy, které známe u počítačů, třeba v příkazu `grep`. Později si ale ukážeme, že regulární výrazy popisují právě regulární jazyky.

Věta 2.18
Jazyk $L \subseteq \Sigma^*$ *je regulární právě tehdy, když je množina kvocientů* $\{w\backslash L \mid w \in \Sigma^*\}$ *konečná.*

Podrobněji se k uvedené větě a jejímu důkazu vrátíme v rozšiřující části. Teď se spokojíme s jejím intuitivním pochopením a uvědomíme si, že je to mocný nástroj, umožňující nám rozlišovat regulární jazyky od neregulárních.

Věta 2.18 vlastně potvrzuje intuici, kterou jsme již jistě o konečných automatech nabyli. Každý konečný automat má omezenou paměť (tedy omezený počet stavů), a proto si po přečtení (dlouhého) prefixu vstupního slova může pamatovat jen omezenou informaci. Je-li onen prefix např. delší než je počet stavů automatu, tak si jej automat už prostě nemůže celý pamatovat.

Kontrolní otázka: Má-li si automat s n stavy zapamatovat vždy celý prefix délky k, jak velké to k může maximálně být?

(Ta maximální délka k není samozřejmě ani n ani $n-1$, ale méně než $\log_2 n$; prefixů délky k je 2^k.)

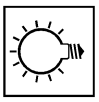
V tomto smyslu může někdo argumentovat: To je přece hned jasné, že jazyk $L = \{a^n b^n \mid n \geq 0\}$ není regulární; konečný automat si přece nemůže pamatovat (libovolně) velký počet znaků a v počátečním úseku!

Ano, je jasné, že si automat tento počet nemůže pamatovat, ale to samo o sobě k prokázání neregularity jazyka L nestačí. Musíme zároveň nějak dokázat, že bez onoho pamatování se v tomto případě automat neobejde.

Naše (povrchní) intuice by nás mohla klamat; může se třeba stát, že prostě jen nevidíme způsob, jak se bez počítání a-ček můžeme obejít. Kdyby nám např. někdo tvrdil, že jazyk $\{\ a^m \mid m$ je dělitelné třemi $\}$ není regulární, protože počet a-ček (který by pak dělil třemi) si konečný automat nemůže pamatovat, ukázali bychom mu, že v tomto případě se bez pamatování si počtu přečtených a-ček lze obejít.

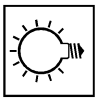
Kontrolní otázka: Jakou informaci z přečteného prefixu si pamatuje vámi navržený automat pro jazyk $\{ a^m \mid m$ je dělitelné třemi $\}$?

K spolehlivému přesvědčení se o tom, že nějaký jazyk je opravdu neregulární, může sloužit právě věta 2.18. Promysleme si ji ještě na dvou příkladech:



Řešený příklad 2.3: Zjistěte, zda jazyk $L = \{w \in \{a,b\}^* \mid |w|_a = |w|_b\}$ je regulární.

Řešení: Není. I zde jsou např. všechny kvocienty $a\backslash L$, $aa\backslash L$, $aaa\backslash L$, ..., navzájem různé. Můžeme použít úplně stejný argument jako u jazyka $L = \{a^n b^n \mid n \geq 0\}$.



Řešený příklad 2.4: Zjistěte, zda jazyk $L = \{w \in \{a,b\}^* \mid$ počet výskytů ab ve w je stejný jako počet výskytů $ba\}$ je regulární.

Řešení: Je. Nenecháme se zmást možným povrchním dojmem, že bychom přece museli počítat a porovnávat počty výskytů ab a ba. Kdybychom začali konstruovat kvocienty, brzy by nám jistě došlo, pokud to nevidíme hned, že v každém slově v abecedě $\{a,b\}$ se počty výskytů ab a ba mohou lišit nejvýš o jedničku. (Umíte jistě rychle sestrojit konečný automat přijímající uvedený jazyk.)

Na základní úrovni bychom si alespoň měli vybudovat intuici k rozlišování regulárních a neregulárních jazyků, tedy schopnost poznat, kdy bychom k zadanému jazyku uměli zkonstruovat konečný automat (byť to fyzicky nebudeme dělat) a kdy by byly takové pokusy beznadějné (byť nepodáme detailní důkaz např. popisem nekonečné množiny různých kvocientů). Zkusme si tedy zodpovědět alespoň následující otázky.

Otázky:

Otázka 2.47: Je jazyk $\{w \in \{a,b\}^* \mid |w|_a \bmod 2 = 0\}$ regulární?

Otázka 2.48: Je jazyk $\{w \in \{a,b\}^* \mid w$ začíná nebo končí dvojicí stejných písmen $\}$ regulární?

Otázka 2.49: Je jazyk $L_1 = \{w \in \{a,b\}^* \mid |w|_a < |w|_b\}$ regulární?

Otázka 2.50: Je jazyk $\{w \in \{a,b,c\}^* \mid$ jestliže w neobsahuje podřetězec abc, pak končí $bca\}$ regulární?

Otázka 2.51: Je jazyk $\{w \in \{a,b\}^* \mid |w|_a > |w|_b$ nebo w končí $baa\}$ regulární?

Otázka 2.52: Je jazyk $\{w \in \{a,b\}^* \mid |w|_a > |w|_b$ nebo $|w|_b \geq 2\}$ regulární?

Otázka 2.53: Je jazyk $\{u \mid$ ex. $w \in \{a,b\}^*$ tak, že $u = ww^R\}$, stručněji také psáno $\{ww^R \mid w \in \{a,b\}^*\}$, regulární?

Otázka 2.54: Je jazyk $\{w \in \{a,b\}^* \mid w = w^R\}$ regulární?

Otázka 2.55: Je jazyk $\{w \in \{a\}^* \mid w = w^R\}$ regulární?

Otázka 2.56: Je jazyk $\{ww \mid w \in \{a,b\}^*\}$ regulární?

Otázka 2.57: Je jazyk $\{ww \mid w \in \{a\}^*\}$ regulární?

Otázka 2.58: Je jazyk $\{w \in \{a,b\}^* \mid$ rozdíl počtů znaků a a znaků b ve w je větší než 100$\}$ regulární?

Otázka 2.59: Je jazyk $\{w \in \{a,b\}^* \mid$ součin $|w|_a$ a $|w|_b$ je větší nebo roven 100$\}$ regulární?

Otázka 2.60*: Je jazyk $\{w \in \{a\}^* \mid |w|$ je prvočíslo $\}$ regulární?

Shrnutí: Tak už máme dobrou představu nejen o tom, jak konstruovat automaty přijímající různé regulární jazyky, ale i o tom, jak poznat (či alespoň „spolehlivě vytušit“), zda daný jazyk je či není regulární. V pozadí všeho jsou zase kvocienty jazyka podle jednotlivých slov v jeho abecedě.

2.9 Nedeterministické konečné automaty

Orientační čas ke studiu této části: 4 hod.

Cíle této části:

Cílem je pochopení pojmu nedeterministického výpočtu a role nedeterminismu v usnadnění návrhu konkrétních automatů. Dále pak zvládnutí algoritmu převodu nedeterministického automatu na deterministický.

Klíčová slova: *nedeterminismus, nedeterministický konečný automat, zobecněný nedeterministický konečný automat, převod nedeterministického konečného automatu na deterministický*

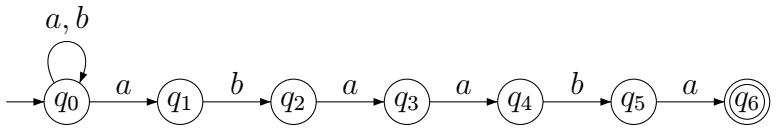
Připomeňme si, že jsme se v části 2.1 poměrně namáhali, než jsme sestrojili automat pro vyhledávání vzorku *abaaba* (tedy automat rozpoznávající jazyk $\{w \in \{a,b\}^* \mid w$ má sufix $abaaba\}$). Měli-li bychom toto činit ručně pro hodně vzorků, jistě bychom přemýšleli, zda tuto práci nelze algoritmizovat (a pak naprogramovat a svěřit počítači). To jistě lze, i když vidíme, že úplně triviálně ten úkol návrhu příslušného algoritmu nevypadá. Za chvíli si ovšem ukážeme, že existuje nástroj, s jehož pomocí se úkol triviálním stane.

Pro jiný motivační zdroj zmíněného „nástroje“ se na chvíli vraťme k modulárnímu návrhu automatů, který jsme probírali v části 2.3. Co když bychom měli automaty pro jazyky L_1, L_2 a chtěli bychom je využít pro (algoritmické) sestrojení automatu přijímajícího zřetězení těch jazyků, tedy jazyk $L_1 \cdot L_2$? Asi nás napadne tento přístup: necháme na první část slova běžet první automat, v přijímajícím stavu pak předáme řízení druhému automatu a ten dočte slovo do konce. Problém je, že obecně nepostačí prostě předat řízení při *prvním* příchodu do přijímajícího stavu, ale je nutno nechat to jako *možnost při jakémkoli* příchodu do přijímajícího stavu.

Kontrolní otázka: Proč nestačí předat řízení při prvním příchodu do přijímajícího stavu? Umíte zkonstruovat jednoduchý příklad, kdy to selže?

Jak to tedy udělat? Odpověď zase není triviální, pokud nepoužijeme již avizovaný „nástroj“. Co je tedy tím „tajuplným nástrojem“, měnícím (některé) technicky komplikované otázky v téměř triviální? Je to *nedeterminismus*. Abychom přiblížili, co to je, podívejme se nejprve na následující graf.

Vypadá to na první pohled jako graf automatu, ale co to? V některých stavech chybí vycházející šipka a, v některých b, a ze stavu q_6 nevychází šipka vůbec! Což o to, chybějící vycházející šipky se ještě dají rozumně pochopit tak, že by asi vedly do nějakého „chybového“ (nepřijímajícího) stavu, tak je ani nekreslíme. Ale u stavu q_0 jsou dvě vycházející šipky označeny a ! Do



Obrázek 2.11:

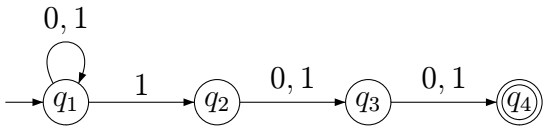
jakého stavu se automat přesune, když je ve stavu q_0 a čte a ? Odpověď je překvapivá: automat se „nedeterministicky rozhodne“ a přesune se buď do stavu q_0 nebo do q_1, podle „momentální nálady“. K čemu je takový „nesmysl“ dobrý? Vždyť pro jedno a totéž slovo w má takový automat najednou více možných výpočtů. Tedy pro jedno w teď můžeme mít více stavů q takových, že $q_0 \xrightarrow{w} q$; slovu w prostě odpovídá více sledů v grafu začínajících v q_0.

Kontrolní otázka: Umíte ve výše uvedeném grafu např. zjistit všechny prvky množiny $Q' = \{q_i \mid q_0 \xrightarrow{aba} q_i\}$ a množiny $Q'' = \{q_i \mid q_0 \xrightarrow{babaaba} q_i\}$? (Provedete-li si to pečlivě sami, a přijdete-li na systematický způsob, jak to dělat, výrazně vám to usnadní pochopení dalšího textu.)

Všimněme si, že v našem grafu je očividné, že množina $\{w \mid q_0 \xrightarrow{w} q_6\}$ (tedy množina slov, pro něž existuje výpočet [sled v grafu], vycházející z počátečního stavu a končící v přijímajícím stavu) je jazykem $L = \{\, w \in \{a, b\}^* \mid w$ má sufix $abaaba\,\}$, tedy přesně tím jazykem, pro nějž jsme (relativně pracně) konstruovali automat v části 2.1. Takže náš graf vlastně uvedeným přirozeným způsobem reprezentuje jazyk L.

Podívejme se teď na následující graf.

Kontrolní otázka: Jaký jazyk je reprezentován (analogicky jako výše) tímto grafem? (Zodpovězte si sami, než se podíváte dále.)



Obrázek 2.12:

Jistě jste si odvodili, že všechna slova, která dávají možnost přejít z počátečního stavu do přijímajícího stavu (tedy všechna slova $w = a_1 a_2 \dots a_n$, pro něž existuje příslušný sled o n hranách, kde a_i je obsaženo v ohodnocení i-té hrany), jsou právě ta slova v abecedě $\{0, 1\}$, v nichž je třetím symbolem od konce znak 1.

Poznámka: Jako vždy u takových tvrzení, je potřeba promyslet obě inkluze (implikace); tedy jak to, že pro každé slovo w mající 1 jako třetí znak od konce existuje v grafu (alespoň jeden) požadovaný sled, tak také to, že pokud pro nějaké slovo w požadovaný sled existuje, tak v tom w je nutně třetí znak od konce 1 (což mj. samozřejmě znamená, že w má délku alespoň tři).

Jistě se shodneme, že návrh a ověření takového grafu reprezentujícího jazyk $\{ w \in \{0,1\}^* \mid |w| \geq 3$ a třetí znak od konce w je $1 \}$ je jednodušší než navrhnout pro tento jazyk (deterministický) konečný automat z definice 2.1.

Cvičení 2.61: Přesto takový automat sestrojte. Později ho můžete porovnat s tím, který vytvoří algoritmus převodu nedeterministického automatu na deterministický.

Pochopili jsme tedy, že výše uvedené grafy, které jsou obecnější než grafy konečných automatů, o nichž jsme dosud pojednávali, také odpovídají jistým jazykům. Je ale rozumně možné dívat se na takový graf jako reprezentaci automatu? Už jsme vytušili, že ano, pokud připustíme, že výpočet není zadaným vstupním slovem determinován (jednoznačně určen). To nás vede k následující definici.

Definice 2.19
Nedeterministický konečný automat, zkráceně NKA, je dán uspořádanou pěticí $A = (Q, \Sigma, \delta, I, F)$, kde

- Q je konečná neprázdná množina *stavů*,
- Σ je konečná neprázdná *abeceda*,
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ je *přechodová funkce*,
- $I \subseteq Q$ je množina *počátečních (iniciálních) stavů*
- $F \subseteq Q$ je množina *přijímajících (koncových) stavů*.

Rozdíl proti definici 2.1 je především v tom, že $\delta(q, a)$ teď nepředstavuje jeden stav, ale množinu stavů (z nichž může být jakýkoli vybrán jako následující); tato množina může být i prázdná (v tom případě výpočet nemůže pokračovat). Další zobecnění je v tom, že počátečních stavů může být více.

Přímočaře lze opět nadefinovat *graf* (též nazývaný stavový diagram) NKA. Příklad takového grafu jsme již viděli např. na Obrázku 2.12. Tímto grafem je tedy reprezentován NKA $A = (Q, \Sigma, \delta, I, F)$, kde $Q = \{q_1, q_2, q_3, q_4\}$, $\Sigma = \{0, 1\}$, $I = \{q_1\}$, $F = \{q_4\}$ a pro δ máme například $\delta(q_1, 1) = \{q_1, q_2\}$, $\delta(q_2, 0) = \{q_3\}$, $\delta(q_4, 1) = \emptyset$ apod.

Samozřejmě můžeme i NKA zadávat i *tabulkou*. V políčkách pak nebudou stavy, ale množiny stavů. Pro stručnost pak obvykle vynecháváme množinové závorky a píšeme pomlčku místo prázdné množiny. Automat reprezentovaný grafem na obrázku 2.12 zadáme tedy tabulkou takto:

	0	1
$\rightarrow q_1$	q_1	q_1, q_2
q_2	q_3	q_3
q_3	q_4	q_4
$\leftarrow q_4$	-	-

Pojem výpočtu a přijímání slova už intuitivně chápeme i u tohoto typu automatu. A uvědomujeme si, že

> *nedeterministický automat může mít pro zadané slovo w hodně výpočtů (začínajících v některém z počátečních stavů); říkáme, že automat slovo w přijímá, jestliže alespoň jeden z těchto výpočtů končí v přijímajícím stavu.*

Podobně jako u standardního automatu, i zde pojmy přijímání slova a jazyka ještě definujeme exaktněji. Nejprve si uvědomíme, že můžeme nadále užívat zavedenou notaci

$$q \xrightarrow{w} q' ,$$

kterou teď ovšem čteme takto: ze stavu q se (nedeterministický) automat *může* přečtením slova w dostat do stavu q'. Podobně chápeme značení $q \xrightarrow{w} Q'$ (speciálně $q \xrightarrow{w} F$) apod.

α

Matematická poznámka: V poznámce před definicí 2.2 jsme značení $q \xrightarrow{w} q'$ zaváděli induktivně. Pro nedeterministický automat $A = (Q, \Sigma, \delta, I, F)$ je v

příslušné definici jediná změna; v bodě 2/ je $\delta(q,a) = q'$ nahrazeno $\delta(q,a) \ni q'$:

1. $q \xrightarrow{\varepsilon} q$
2. $q \xrightarrow{a} q' \Leftrightarrow \delta(q,a) \ni q'$
3. když $q \xrightarrow{a} q'$ a $q' \xrightarrow{u} q''$, tak $q \xrightarrow{au} q''$

Definice 2.20
Mějme NKA $A = (Q, \Sigma, \delta, I, F)$.
Slovo $w \in \Sigma^*$ je *přijímáno* automatem A, jestliže alespoň pro jeden stav $q_0 \in I$ platí $q_0 \xrightarrow{w} F$.
Jazykem rozpoznávaným (přijímaným) automatem A rozumíme jazyk
$L(A) = \{w \in \Sigma^* \mid$ slovo w je přijímáno $A\} = \{w \in \Sigma^* \mid \exists q_0 \in I : q_0 \xrightarrow{w} F\}$.

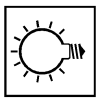
Dříve definovanému konečnému automatu budeme také někdy říkat *deterministický* automat. Je dobré si uvědomit, že

> *deterministický konečný automat je speciálním případem nedeterministického.*

Kontrolní otázka: Máme-li (deterministický) konečný automat $A = (Q, \Sigma, \delta, q_0, F)$, co musíme formálně udělat, aby splňoval definici nedeterministického konečného automatu?

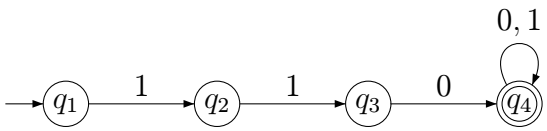
(V našem znázornění grafem a tabulkou nic. Jde jen o tu formalitu, že počáteční stav q_0 „zaměníme“ jednoprvkovou množinou $\{q_0\}$ a hodnotu $\delta(q,a)$ začneme chápat jako jednoprvkovou množinu; bylo-li $\delta(q,a) = q'$, je nově $\delta(q,a) = \{q'\}$.)

Navrhněme teď společně jednoduchý NKA.



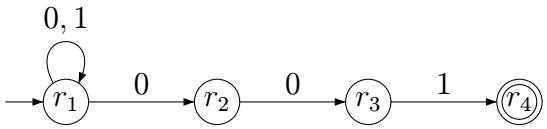
Řešený příklad 2.5: Sestrojte (jednoduchý) *nedeterministický* konečný automat, který přijímá právě ta slova v abecedě $\{0,1\}$, jež začínají 110 nebo končí 001.

Řešení: Zkusme nejdříve NKA A_1 pro slova začínající 110: takový automat přečte na začátku 110 (u jiného začátku „havaruje“, výpočet se prostě „zasekne“), a pak už přijme cokoliv. Takže nás jistě hned napadne graf



Všimněme si, že automat A_1 je vlastně deterministický, jen mu prostě chybí některé přechody (které by vedly do chybového stavu).

Pro slova končící 001 je to trochu jinak: automat čte jakýkoli prefix, až se v nějakém okamžiku (nedeterministicky) rozhodne, že teď bude následovat závěrečný sufix 001; to ověří a přijme (či zhavaruje, pokud „hádal špatně“). Takže graf příslušného automatu A_2 zachycuje následující obrázek.



Tady už jsme nedeterminismus skutečně využili. Je jasné, že každé slovo w končící 001 automat přijme, tedy existuje možnost, že A_2 přejde z počátečního stavu r_1 přečtením slova w do přijímajícího stavu r_4. Naopak je to ještě zřejmější: každé slovo, které automat přijme, nutně končí 001.

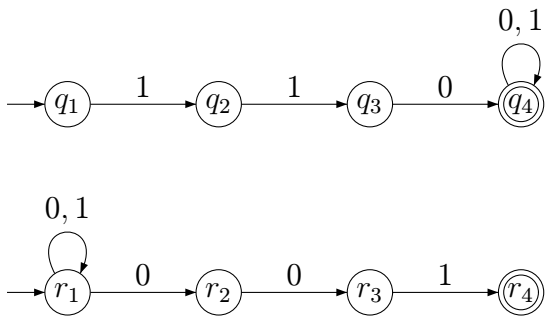
A jak teď dostaneme automat přijímající jazyk $\{w \in \{0,1\}^* \mid w$ začíná 110 nebo končí 001 $\}$?

Podle definice lze mít počátečních stavů více, tak výše uvedené grafy prostě položíme vedle sebe (odborně řečeno: provedeme disjunktní sjednocení automatů A_1, A_2), a je to! (Výsledný automat má tedy 8 stavů.)

Převod na deterministické automaty

Teď už sice víme, jak lze definovat přijímání slov a jazyků nedeterministickými automaty (či jejich grafy), ale jak můžeme např. pro konkrétní NKA A a slovo w (algoritmicky) rozhodnout, zda $w \in L(A)$? Už jste byli dříve vyzváni, ať si to sami vyzkoušíte, teď se na to podíváme společně.

Je jasné, že obecně musíme nějak systematicky sledovat všechny možné výpočty, ať můžeme rozhodnout, zda alespoň jeden je úspěšný (přijímající). Pro konkrétnost vezměme např. dříve zkonstruovaný NKA reprezentovaný následujícím grafem:



Můžeme si představit, že slovo w nám někdo bude diktovat znak po znaku a my máme vždy ohlásit, pokud je dosud nadiktovaný prefix u slova w (které dopředu neznáme) přijímán (pokud tedy v grafu existuje alespoň jeden sled z nějakého počátečního do nějakého přijímajícího stavu, jehož ohodnocení je v souladu s prefixem u).

Asi přijdeme na postup, který lze popsat následovně, v termínech jakési knoflíkové hry na grafu.

Na začátku položíme knoflíky právě na počáteční stavy (respektive vrcholy grafu jim odpovídající); v našem případě je tedy počáteční „knoflíkovou množinou“ množina $K_0 = \{q_1, r_1\}$.

Leží-li nyní jeden knoflík na přijímajícím stavu, hlásíme přijetí (prefixu ε); v našem případě tomu tak není.

Je-li nám nyní nahlášen znak 0, prozkoumáme, kam se mohou posunout knoflíky z aktuální knoflíkové množiny po šipkách označených 0, a příslušné cílové vrcholy vytvoří novou aktuální knoflíkovou množinu. V našem případě se knoflík z q_1 nemůže posunout nikam, takže mizí bez náhrady. Z r_1 se po 0-šipce lze přesunout do r_1 nebo r_2. Takže nová aktuální knoflíková množina je $K_1 = \{r_1, r_2\}$.

Kdyby nám v situaci K_0 byl nahlášen znak 1, sestrojíme obdobně množinu $K_2 = \{q_2, r_1\}$.

Pokud máme situaci (knoflíkové rozestavení) K_1 a je nám nahlášena 0, přejdeme do $K_3 = \{r_1, r_2, r_3\}$ (počet knoflíků se pochopitelně může zvyšovat i snižovat).

Konstrukce, které takto popisujeme slovně, je samozřejmě lepší zachycovat stručnějším a přehlednějším způsobem, např. takto:

		0	1
$\rightarrow$	$K_0 = \{q_1, r_1\}$	K_1	K_2
	$K_1 = \{r_1, r_2\}$	K_3	
	$K_2 = \{q_2, r_1\}$		
	$K_3 = \{r_1, r_2, r_3\}$		

Tohle je nám samozřejmě nějaké povědomé; no ano, vždyť my vlastně konstruujeme jistý deterministický konečný automat (DKA) A'. Stavem tohoto DKA A' je ovšem množina stavů původního NKA A; počátečním stavem A' je množina počátečních stavů A.

Kontrolní otázka: Co budou přijímací stavy A' ?

(Jistě jste přišli na to, že stav automatu A' je přijímající právě tehdy, když příslušná množina stavů A obsahuje alespoň jeden přijímající stav.)

Cvičení 2.62: Dokončete pečlivě konstrukci výše uvedeného automatu se stavy $K_0, K_1, K_2, \ldots$.

Teď nás už ovšem nepřekvapí následující věta a neměl by nás překvapit ani její důkaz.

Věta 2.21
Existuje algoritmus, který ke každému nedeterministickému konečnému automatu A sestrojí ekvivalentní (deterministický) konečný automat A' (tedy $L(A) = L(A')$).

Důkaz: K NKA $A = (Q, \Sigma, \delta, I, F)$ sestrojíme DKA $A' = (Q', \Sigma, \delta', I, F')$, kde

- $Q' = \mathcal{P}(Q)$ je množina všech podmnožin stavů Q; množina $I \in Q'$ je počátečním stavem,

- $F' = \{K \in Q' \mid K \cap F \neq \emptyset\}$ (F' tedy obsahuje všechny podmnožiny množiny Q, které obsahují některý stav z F),
- Přechodová funkce $\delta' : Q' \times \Sigma \to Q'$ každé podmnožině původních stavů $K \subseteq Q$ a písmenu $a \in \Sigma$ přiřadí podmnožinu $\{q \in Q \mid \exists q' \in K : q' \xrightarrow{a} q\}$.

Není těžké se přesvědčit, že nový DKA A' přijímá stejná slova jako původní NKA A – aktuální stav K deterministického A' po přečtení libovolného slova u představuje množinu těch stavů nedeterministického A, které lze dosáhnout (nedeterministickými) výpočty A přečtením u. □

Opět můžeme (snad všichni) ocenit, jak stručně a přesně vystihuje matematická notace myšlenku předešlé konstrukce. Všimněme si teď ještě několika věcí.

Důkaz věty 2.21 ukazuje pro NKA s n stavy konstrukci DKA s 2^n stavů! Proto takový algoritmus nemůže být nazván rychlý (jak o tom budeme mluvit v teorii algoritmů).

Kontrolní otázka: Ve výše uvedeném cvičení, kdy jste dokončovali konstrukci množin $K_0, K_1, K_2, \ldots$, vám ovšem jistě vyšlo (podstatně) méně než $2^8 = 256$ stavů; čím to je?

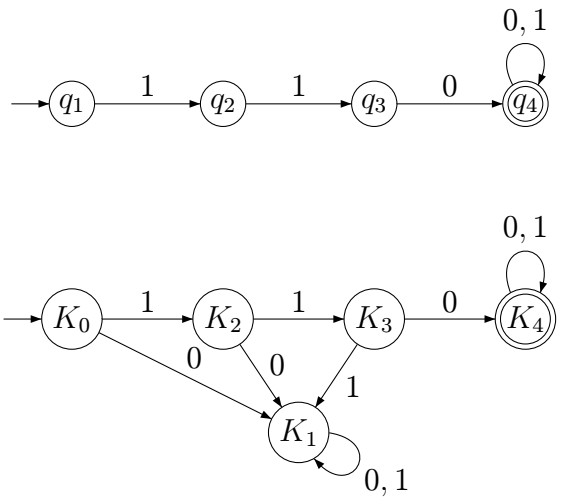
Jistě jste si uvědomili, že navržená konstrukce automaticky sestrojí jen *dosažitelné* stavy deterministického A', což je v mnoha případech podstatně menší množina než množina všech podmnožin stavové množiny automatu A.

Poznámka: Víme, že výsledný deterministický automat může být i pak možné zmenšit redukcí (což jste možná u zmíněného cvičení alespoň částečně využili). Bohužel existují i nedeterministické automaty s n stavy, u nichž má i ten nejmenší ekvivalentní deterministický automat oněch 2^n stavů. Podrobněji se k tomu vrátíme v rozšiřující části.

Ještě stojí za zvláštní zmínku speciální stav DKA A' odpovídající prázdné množině. Takový stav je dosažitelný např. v deterministickém automatu odpovídajícím výše zkoumanému automatu s následujícím grafem.

Převod na DKA (pro nějž musí být podle definice definován přechod pro každý stav a každé písmeno) vydá tento výsledek:

(kde $K_0 = \{q_1\}$, $K_1 = \emptyset$, $K_2 = \{q_2\}$, $K_3 = \{q_3\}$, $K_4 = \{q_4\}$).



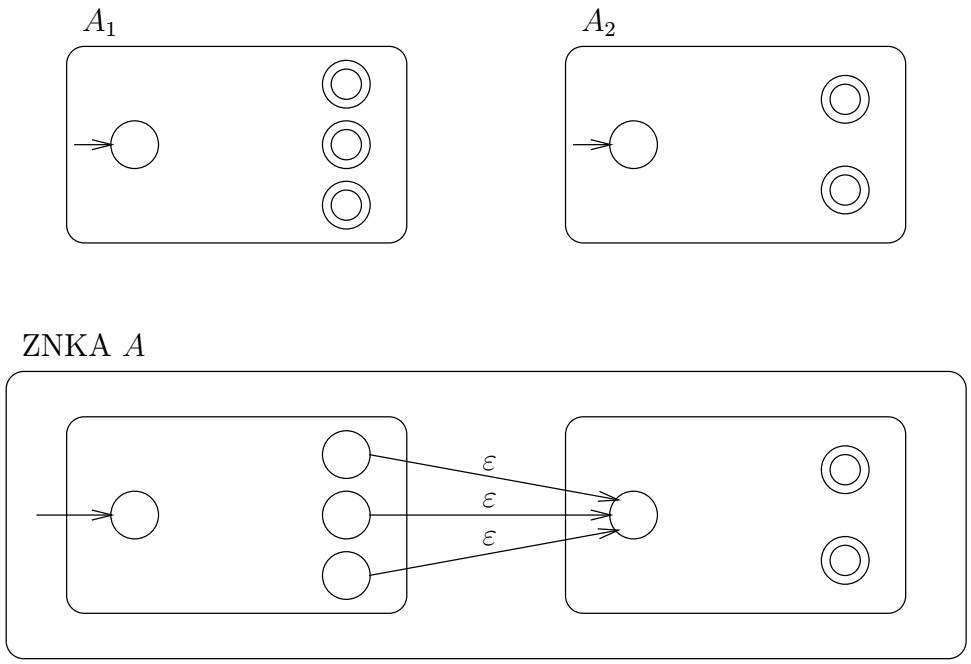
Stav odpovídající prázdné množině tedy přirozeně odpovídá „chybovému stavu“ (v něm už výpočet natrvalo zůstává). Někdy se u deterministických automatů takový stav ani nekreslí; musí ale být z kontextu jasné, že všechny „chybějící šipky“ chybí záměrně, neboť by vedly do takového chybového stavu.

Cvičení 2.63: Zkonstruujte ekvivalentní deterministický automat k automatu z obrázku 2.11.

Vzpomeňme si, že na začátku této části jsme hovořili o nedeterminismu jako prostředku umožňující „beznámahovou“ konstrukci DKA pro vyhledávání vzorku *abaaba* apod. To už je nám jasné: dokonce je nám jasný algoritmus, který k zadanému vzorku navrhne nedeterministický automat, no a ten lze pak algoritmicky převést na deterministický.

Ale co vyřešení problému s konstrukcí automatu pro jazyk $L(A_1) \cdot L(A_2)$? K tomu se ještě více hodí mírně obecnější forma nedeterminismu. Přidáme možnost tzv. ε-přechodů, tedy přidáme automatu možnost změnit stav bez čtení vstupního symbolu (a to třeba vícekrát za sebou). Vše by mělo být intuitivně jasné ze schematického obrázku 2.13.

Automaty A_1, A_2 prostě položíme vedle sebe, uděláme tedy jejich disjunktní sjednocení, a z přijímajících stavů A_1 vedeme ε-šipky do počátečního stavu A_2. Pak ponecháme jako počáteční stav jen počáteční stav A_1 a jako při-



Obrázek 2.13: $L(A) = L(A_1) \cdot L(A_2)$

jímající ponecháme jen přijímající stavy A_2. Ač jsme ještě nepodali přesné definice, už by mělo být intuitivně jasné, že vniklý nedeterministický automat, kterému říkáme „zobecněný“ díky přidané možnosti ε-šipek, skutečně přijímá právě slova z $L(A_1) \cdot L(A_2)$.

Zobecněný nedeterministický konečný automat

Formálně definujeme uvedený automat takto:

Definice 2.22
Zobecněný nedeterministický konečný automat (ZNKA) je dán pěticí $A = (Q, \Sigma, \delta, I, F)$, kde

- Q je konečná neprázdná množina *stavů*,
- Σ je konečná neprázdná *abeceda*,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ je *přechodová funkce*,

- $I \subseteq Q$ je množina *počátečních (iniciálních) stavů*
- $F \subseteq Q$ je neprázdná množina *přijímajících (koncových) stavů*.

Jediný rozdíl mezi ZNKA a NKA je tedy v definici přechodové funkce, která zde připouští i ε-kroky.

Rozšíření významu naší notace

$$q \xrightarrow{w} q' ,$$

i na případ ZNKA by mělo být zřejmé: ze stavu q se (zobecněný nedeterministický) automat *může* přečtením slova w, s případnými ε-mezikroky, dostat do stavu q'. Podobně chápeme značení $q \xrightarrow{w} Q'$ (speciálně $q \xrightarrow{w} F$) apod.

α

Matematická poznámka: Pro ZNKA $A = (Q, \Sigma, \delta, I, F)$ lze podat tuto induktivní definici relace $q \xrightarrow{w} q'$

1. $q \xrightarrow{\varepsilon} q$
2. $\delta(q, a) \ni q' \Rightarrow q \xrightarrow{a} q'$ pro $a \in \Sigma \cup \{\varepsilon\}$
3. když $q \xrightarrow{u} q'$ a $q' \xrightarrow{v} q''$, tak $q \xrightarrow{uv} q''$

Definice přijetí slova i jazyka pak pro ZNKA vypadá úplně stejně jako pro NKA (viz definice 2.20).

Knoflíková hra, podávající algoritmický návod k rozhodnutí, zda dané w je přijato daným ZNKA, obsahuje další aspekt: aktuální oknoflíkovanou množinu stavů musíme vždy rozšířit o ty stavy, do nichž se z oknoflíkovaných lze dostat pomocí ε-šipek. Podívejme se na konkrétní ZNKA znázorněný následující tabulkou (ta teď samozřejmě obsahuje i sloupeček pro ε).

	a	b	c	ε
→1	2	-	-	3
2	1	-	-	-
3	-	4	-	5
4	-	3	-	-
←5	-	-	6	-
6	-	-	5	-

Co je teď počáteční stav K_0 ? Obsahuje samozřejmě původní počáteční stav 1, díky tomu ale musí obsahovat i stav 3, protože do něj lze ze stavu 1 přejít ε-krokem; pak ovšem obsahuje i 5. Čili $K_0 = \{1, 3, 5\}$ je onou počáteční knoflíkovou množinou, obsahující všechny počáteční stavy a ty stavy, které jsou z počátečních dosažitelné jen pomocí ε-šipek. K_0 je tedy i přijímající, protože obsahuje původní přijímající stav 5.

Když máme ve stavu $K_0 = \{1, 3, 5\}$ zpracovat znak a, vygeneruje knoflík na stavu 1 knoflík na stavu 2 a knoflíky na 3,5 nevygenerují nic. Základem následující knoflíkové množiny je tedy množiny $\{2\}$; z jejich prvků se už ovšem ε-šipkami dál nedostaneme, takže máme $K_0 \xrightarrow{a} K_1$, kde $K_1 = \{2\}$ a není přijímající.

Cvičení 2.64: Dokončete započatou konstrukci deterministického automatu se stavy $K_0, K_1, \ldots$.

Aplikovali jsme tedy následující metodu (platnou samozřejmě i pro nedeterministický automat bez ε-šipek).

Metoda 2.23
Převod ZNKA A na ekvivalentní DKA A'.

- Počátečním stavem K_0 automatu A' bude stav reprezentující množinu všech počátečních stavů A doplněnou o všechny stavy dosažitelné v A libovolnými sekvencemi ε-přechodů.
- Dokud máme v sestrojovaném automatu A' nějaký stav K_i, který nemá definovány přechody pro všechna písmena, opakujeme následující:
  - vybereme si jeden takový stav K_i (např. ten s nejmenším pořadovým číslem) a písmeno a, pro nějž přechod z K_i není definován. Pro všechny stavy q v množině reprezentované stavem K_i najdeme všechny možnosti přechodu znakem a v A a příslušné cílové stavy shrneme v nové množině stavů K, do níž ještě přidáme všechny stavy dosažitelné v A libovolně dlouhou sekvencí ε-přechodů ze stavů již se v K nacházejících.
  - Pokud se K rovná nějakému již vytvořenému K_j, vedeme v A' šipku $K_i \xrightarrow{a} K_j$; jinak použijeme nejnižší dosud nepoužitý index j, definujeme $K_j = K$ a vedeme šipku $K_i \xrightarrow{a} K_j$.

- Každý stav K_i, který reprezentuje množinu obsahující alespoň jeden přijímající stav z A, označíme v A' jako přijímající.

Všimněme si, že v uvedeném postupu se odvoláváme k algoritmu dosažitelnosti (jako k již definované proceduře).

Kontrolní otázka: Jak upravíte náš známý algoritmus zjišťující dosažitelnost stavů z počátečního stavu pro případ zjišťování stavů dosažitelných jen sekvencemi ε-přechodů ze stavů dané množiny K ?

De facto jsme tedy dokázali následující větu.

Věta 2.24
Existuje algoritmus, který ke každému zobecněnému nedeterministickému konečnému automatu A sestrojí ekvivalentní (deterministický) konečný automat A' (tedy $L(A) = L(A')$).

Ukázali jsme tedy, že ačkoli (zobecněné) nedeterministické konečné automaty mají více možností, jak reprezentovat jazyky, neumějí v tomto smyslu víc než standardní konečné automaty:

Věta 2.25
Zobecněné nedeterministické automaty rozpoznávají právě regulární jazyky (stejně jako standardní deterministické konečné automaty).

(Ke každému ZNKA A existuje DKA A' takový, že $L(A') = L(A)$.)

Otázky:

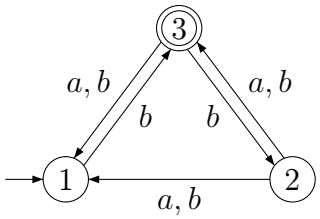
Otázka 2.65: Je výpočet ZNKA vždy konečný?

Otázka 2.66: Co se stane, pokud Metodu 2.23 aplikujeme na deterministický automat?

Otázka 2.67: Kdy při konstrukci podle Metody 2.23 vznikne stav reprezentovaný prázdnou množinou $\emptyset$?

Otázka 2.68: Může být stav $\emptyset$ (dle předchozí otázky) přijímajícím?

Cvičení 2.69: Kdy následující automat přijímá neprázdné slovo složené ze samých písmen a?

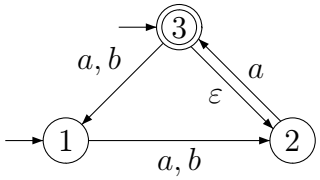


Cvičení 2.70: Sestrojte ekvivalentní deterministický automat k nedeterministickému automatu ze Cvičení 2.69:

Cvičení 2.71: Která slova z $\{b\}^*$ přijímá automat ze cvičení 2.69 ?

Cvičení 2.72: Navrhněte nedeterministický automat přijímající jazyk všech těch slov nad $\{a, b\}$, které končí sufixem „abb“ nebo sufixem „aa“.

Cvičení 2.73:Následující zobecněný nedeterministický konečný automat převeďte na deterministický bez nedosažitelných stavů.



Shrnutí: Seznámili jsme se podrobně s nedeterministickými konečnými automaty. Uvědomili jsme si přitom, že jejich užitečnost nespočívá v tom, že by uměly reprezentovat více jazyků než deterministické konečné automaty, ale v tom, že umějí regulární jazyky reprezentovat mnohdy daleko stručněji a přehledněji. Významným faktorem z hlediska praktické užitečnosti pak je existence algoritmů převádějících nedeterministické automaty na ekvivalentní deterministické.

2.10 Uzávěrové vlastnosti třídy regulárních jazyků.

Orientační čas ke studiu této části: 1 hod.

Cíle této části:

Máte zvládnout některé další modulární konstrukce (využívající nedeterminismus) pro tvorbu automatů pro složitější jazyky. Dále máte pochopit a zrekapitulovat uzavřenost třídy regulárních jazyků vůči různým jazykovým operacím.

Klíčová slova: *třída regulárních jazyků, zřetězení a iterace jazyků, uzavřenost třídy jazyků vůči různým operacím*

Již máme dobrou představu o tom, že existují regulární i neregulární jazyky (každý má svou příslušnou abecedu). Zkusme se trochu z nadhledu podívat na množinu všech regulárních jazyků; říkáme jí také

třída regulárních jazyků a budeme ji označovat REG.

Známe tedy mnohé příklady jazyků $L \in \mathsf{REG}$ i jazyků $L \notin \mathsf{REG}$. Připomeneme-li si naše znalosti z modulárních konstrukcí automatů apod., uvědomíme si např., že pro jakékoli jazyky $L_1, L_2 \in \mathsf{REG}$ platí, že také $L_1 \cup L_2 \in \mathsf{REG}$. Říkáme, že třída REG je *uzavřena vůči operaci sjednocení.*

Poznámka: Jedná se o speciální případ uzavřenosti množiny vzhledem k operaci, jak je to definováno v Sekci **??**.

Takže v prvé řadě si uvědomíme, že jsme si již dokázali následující větu.

Věta 2.26
Třída REG *je uzavřena vůči sjednocení, průniku, doplňku. (Je-li tedy* $L_1, L_2 \in$ REG*, pak také* $L_1 \cup L_2$*,* $L_1 \cap L_2$*,* $\overline{L_1}$ *jsou v* REG*.)*

Kontrolní otázka: Plyne uzavřenost REG vůči některé z operací z uzavřenosti vůči dalším dvěma ?

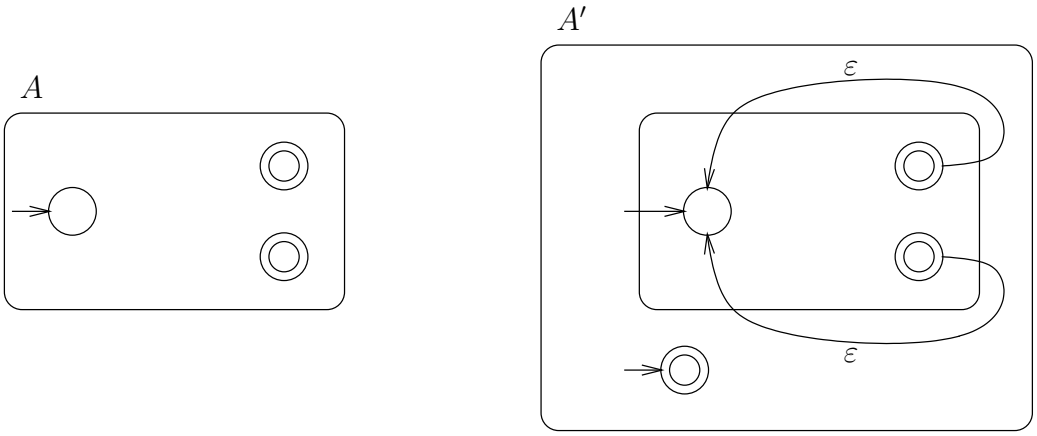
(Ano, např. uzavřenost vůči průniku plyne z uzavřenosti vůči sjednocení a doplňku – díly de Morganovým pravidlům.)

Podobně je třída REG uzavřena i na jazykové operace zřetězení a iterace:

Věta 2.27
Třída REG *je uzavřena vůči zřetězení a iteraci. (Je-li tedy* $L_1, L_2 \in$ REG*, pak také* $L_1 \cdot L_2$*, a* $(L_1)^*$ *jsou v* REG*.)*

Důkaz: Spokojíme se zde s názornými obrázky. Pro zřetězení se odvoláme na již diskutovaný obrázek 2.13.

Pro iteraci postačí následující obrázek.



Obrázek 2.14: $L(A') = L(A)^*$

Ten ukazuje, jak k automatu A zkonstruovat ZNKA A' rozpoznávající $L(A)^*$. Prostě ze všech přijímajících stavů vedeme ε-šipky do počátečního stavu a přidáme nový (izolovaný) počáteční stav, který je zároveň přijímajícím.

Kontrolní otázka: Proč?

(Víme, že podle definice každý L^* musí obsahovat ε [dokonce i $\emptyset^*$].)

Je snadné se přesvědčit, že ke každému slovu z $L(A)^*$ (to je tvaru $u_1u_2 \ldots u_n$, kde $u_i \in L(A)$ pro $i = 1, 2, \ldots, n$) existuje přijímající výpočet v ZNKA A' a naopak existence přijímajícího výpočtu automatu A' pro slovo w nutně znamená, že $w \in L(A)^*$.

□

Třída REG je uzavřena vůči mnoha dalším operacím (o některých pojednáme v rozšiřující části); zde se spokojíme ještě s jednou operací:

Věta 2.28
Třída REG *je uzavřena vůči operaci zrcadlového obrazu. (Je-li tedy* $L \in \mathsf{REG}$*, pak také* $L^R \in \mathsf{REG}$*.)*

Důkaz: Není těžké přijít na to, že u automatu pro L stačí zaměnit počáteční stavy s přijímajícími (stav, který byl počáteční, je po té změně přijímající, stav, který byl přijímající, je po té změně počáteční) a obrátíme všechny šipky. Tím vznikne (obvykle nedeterministický!) automat očividně přijímající L^R. □

Všimněme si, že všechny naše důkazy uzavřenosti REG vůči operacím jsou *konstruktivní*, tedy obsahují vždy návod, jak lze k výchozím automatům pro jazyky L_1, L_2 algoritmicky zkonstruovat automat pro jazyk vzniklý aplikací příslušné operace na jazyky L_1, L_2. (U unárních operací jako je doplněk, iterace, zrcadlový obraz je samozřejmě výchozím jen jeden automat.)

Když tedy složitější jazyky popíšeme pomocí aplikací příslušných operací na jednoduché jazyky, pro něž umíme sestrojit konečné automaty, víme, že ony složitější jazyky jsou také regulární a sestrojení konečných automatů pro ně můžeme vlastně svěřit počítači (naprogramováním příslušných algoritmů).

Zvláštní roli mezi operacemi, na které je uzavřena třída regulárních jazyků, hrají tzv. regulární operace:

Definice 2.29
Regulárními operacemi s jazyky nazýváme operace

- *sjednocení* $L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$,
- *zřetězení* $L_1 \cdot L_2 = \{uv \mid u \in L_1, v \in L_2\}$
- a *iterace* $L^* = \{w \mid w$ lze psát $w = u_1 u_2 \dots u_n$ pro nějaké $n \geq 0$ a $u_i \in L, i = 1, 2, \dots, n\}$ (zřetězení $n = 0$ slov chápeme jako ε, které je tedy vždy prvkem L^*).

Poznámka: Pro tuto chvíli jen poznamenáme, že regulárními operacemi lze vytvořit všechny regulární jazyky z tzv. elementárních (jednoprvkových)

jazyků; o tyto operace se také opírají (teoretické) regulární výrazy, o kterých budeme hovořit dále.

Shrnutí: Rozšířili jsme naše obzory ohledně konstrukcí automatů pro složitější jazyky (vyjádřitelné aplikacemi jazykových operací na jednodušší jazyky). Všimli jsme si, jak nám při tom opět pomáhá nedeterminismus. Je nám také jasné, že kromě konstrukce konkrétních automatů je užitečné podívat se na naše snažení trochu „seshora“ a formulovat naše poznatky ve formě vlastností třídy REG, která obsahuje všechny regulární jazyky.

2.11 Cvičení

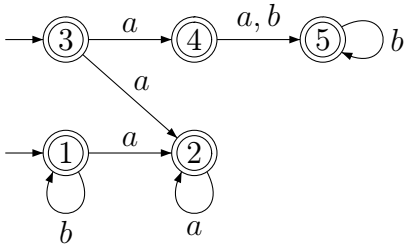
Orientační čas ke studiu této části: 1 hod.

Cíle této části:

Cílem je opět určité zopakování a procvičení nabytých znalostí. Tomu má napomoci promyšlení a vyřešení dalších otázek a příkladů.

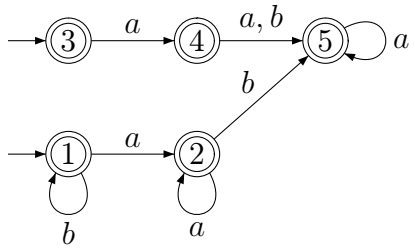
Cvičení 2.74: Sestrojme nedeterministický automat (ZNKA) rozpoznávající jazyk všech těch slov nad abecedou $\{a, b, c\}$, která neobsahují žádný znak a, nebo počet výskytů znaku b je sudý nebo počet výskytů znaku c dává při dělení třemi zbytek 2.

Cvičení 2.75: Existuje slovo nad abecedou $\{a, b\}$, které *nepatří* do jazyka přijímaného následujícím nedeterministickým automatem se dvěma počátečními stavy?

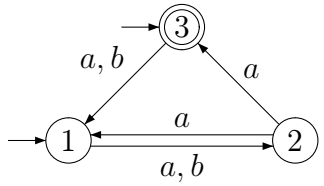


Poznámka: Pozor, přestože všechny stavy jsou přijímající, odpověď není tak triviální.

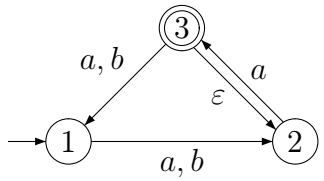
Cvičení 2.76: Najděte libovolné slovo nad abecedou $\{a, b\}$, které *nepatří* do jazyka přijímaného tímto nedeterministickým automatem se dvěma počátečními stavy:



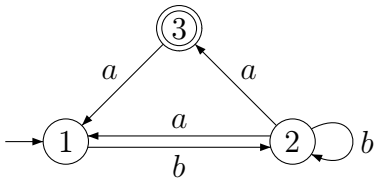
Cvičení 2.77: Následující nedeterministický konečný automat převeďte na deterministický (bez nedosažitelných stavů).



Cvičení 2.78: Následující zobecněný nedeterministický konečný automat převeďte na deterministický bez nedosažitelných stavů.



Cvičení 2.79*: Slovně popište jazyk přijímaný následujícím nedeterministickým automatem.



2.12 Regulární výrazy

Orientační čas ke studiu této části: 2 hod.

Cíle této části:

Cílem je seznámení se s regulárními výrazy, jež jsou dalším prostředkem popisu regulárních jazyků a které jsou mj. základem pro popis vzorků vyhledávaných v (textových) souborech. Máte zvládnout algoritmus převodu regulárních výrazů na ekvivalentní konečné automaty a také si procvičit návrh regulárních výrazů pro konkrétní jazyky.

Klíčová slova: *regulární výraz, konstrukce konečného automatu k danému regulárnímu výrazu*

Všichni se téměř každodenně potkáváme s úlohou vyhledávání vzorků (slov) v textech (např. na Internetu). Na vyhledávání existují standardní softwarové nástroje, které jsou obvykle přímo zabudovány do systému nebo do textových editorů. Jako informatici bychom jistě měli mít alespoň ponětí o tom, na čem jsou tyto nástroje založeny.

Již v části 2.1 jsme pochopili, že v pozadí stojí něco, čemu se říká konečný automat, a v dalších částech jsme si tento prostředek docela podrobně „ohmatali“.

Máme-li tedy vyhledávat v textu konkrétní slovo (např. ‘`abaaba`’, ‘`PES`’, ‘`informatika`’ apod.), víme, jak bychom to sami algoritmicky řešili; sestrojili (a implementovali) bychom automat rozpoznávající jazyk sestávající ze všech slov v příslušné abecedě, jež mají sufix rovnající se hledanému vzorku.

Často ovšem potřebujeme vyhledávat obecnější vzorky než konkrétní slova. Vzorek může být např. specifikován (booleovskou) kombinací jednoduchých podmínek. Např. si lze představit, že výrazem

„(česk∗ & sloven∗) ∨ (česk∗ & němec∗)“

zadáváme (v nějakém systému) přání nalézt všechny dokumenty, které zároveň obsahují slovo začínající na „česk“ a slovo začínající na „sloven“ nebo zároveň obsahují slovo začínající na „česk“ a slovo začínající na „němec“.

Na výrazy podobné uvedenému lze pohlížet jako na popis (reprezentaci) určitého jazyka – reprezentovány jsou ty posloupnosti písmen (v „reálu“ např. dokumenty, v našich pojmech jim říkáme prostě slova), které danému výrazu vyhovují; všimněme si, že takto reprezentovaný jazyk je pak obvykle nekonečný.

My se teď nesoustředíme na konkrétní (počítačový) systém, ale uvedeme *regulární výrazy*, které jsou určitým způsobem základní a na které se obvykle omezujeme v teorii.

Poznámka: Tyto (teoretické) regulární výrazy samozřejmě mají úzký vztah k různým „praktickým“ regulárním výrazům, s nimiž se setkáváme v různých softwarových systémech. Slouží nám jako určitý (nejjednodušší) prototyp; pochopíme-li tento prototyp a jeho vztah ke konečným automatům, nebudeme mít problémy s pochopením oněch praktických systémů.

Musíme si tedy především domluvit *syntaxi* (způsob utvoření) zápisu, který budeme nazývat regulárním výrazem. Dále musíme popsat *sémantiku* (význam), která ke každému regulárnímu výrazu α přiřazuje jazyk (množinu slov), kterou tento výraz reprezentuje; jazyk reprezentovaný výrazem α zde označujeme $[\alpha]$. K zadání přesné syntaxe a sémantiky slouží následující (induktivní) definice. (Máte-li s nimi při prvním čtení problémy, podívejte se nejprve na příklady za nimi.)

Definice 2.30
Regulárními výrazy nad abecedou Σ rozumíme množinu $RV(\Sigma)$ slov v abecedě $\Sigma \cup \{\, \emptyset, \varepsilon, +, \cdot, ^*, (,) \,\}$ (kde předpokládáme, že $\emptyset, \varepsilon, +, \cdot, ^*, (,) \notin \Sigma$), která splňuje tyto podmínky:

- Symboly $\emptyset, \varepsilon$ a symbol a pro každé písmeno $a \in \Sigma$ jsou prvky $RV(\Sigma)$; tyto symboly také nazýváme *elementárními regulárními výrazy*.

- Jestliže $\alpha, \beta \in RV(\Sigma)$, pak také $(\alpha + \beta) \in RV(\Sigma)$, $(\alpha \cdot \beta) \in RV(\Sigma)$ a $(\alpha^*) \in RV(\Sigma)$.
- $RV(\Sigma)$ neobsahuje žádné další řetězce, tedy do $RV(\Sigma)$ patří právě ty řetězce (výrazy), které jsou konstruovány z $\emptyset, \varepsilon$ a písmen abecedy Σ výše uvedenými pravidly.

Definice 2.31
Regulární výraz α *reprezentuje jazyk*, který označujeme $[\alpha]$; ten je dán následujícími pravidly:

- $[\emptyset] = \emptyset$, $[\varepsilon] = \{\varepsilon\}$, $[a] = \{a\}$
- a dále $[(\alpha + \beta)] = [\alpha] \cup [\beta]$, $[(\alpha \cdot \beta)] = [\alpha] \cdot [\beta]$, $[(\alpha^*)] = [\alpha]^*$.

Pro abecedu $\Sigma = \{0, 1\}$ je regulárním výrazem např. řetězec $(((0 \cdot 1) + ((1 \cdot 0) \cdot 0))^*)$.

Kontrolní otázka: Jakou posloupností syntaktických pravidel výraz vznikl?

Jistě snadno odvodíme, že uvedený výraz reprezentuje jazyk $\{01, 100\}^*$.

Definice předepisuje důsledné používání závorek, které odrážejí posloupnost použitých pravidel, pro praktické použití se ovšem hodí možnost „zbytečné“ závorky vynechávat; jde nám totiž v prvé řadě o sémantiku, tedy o reprezentovaný jazyk. Také je užitečná možnost vynechávání symbolu $\cdot$ (jehož skrytá přítomnost je dána kontextem). Intuitivně jistě hned pochopíme, že pokud uvedený výraz zjednodušíme na zápis $(01 + 100)^*$, zachováme všechnu informaci o reprezentovaném jazyku. Využíváme tedy následující možnosti zjednodušení značení:

Značení: Při zápisu regulárních výrazů vynecháváme zbytečné závorky (asociativita operací, vnější pár závorek) a tečky pro zřetězení; další závorky lze vynechat díky dohodnuté prioritě operací: * váže silněji než $\cdot$, která váže silněji než $+$.
Např. místo $((((0 \cdot 1)^* \cdot 1) \cdot (1 \cdot 1)) + ((0 \cdot 0) + 1)^*)$ napíšeme $(01)^*111 + (00 + 1)^*$.

Komentář: Jistě se shodneme, že regulární výrazy celkem „průhledně“ popisují jazyky vzniklé z elementárních jazyků ($\emptyset$, $\{\varepsilon\}$ a $\{a\}$ pro $a \in \Sigma$) *regulárními operacemi* (připomeňme si definici 2.29). Snad jediné drobné „překvapení“ je, že operace sjednocení se zda zapisuje symbolem $+$. (Mohli

bychom samozřejmě používat znak $\cup$, ale zůstaneme u v této oblasti zaběhnutého $+$.)

Všimněme si speciálně, že v našich regulárních výrazech není symbol pro operaci průniku ani doplňku! (To jsou příklady operací, které se často v počítačové praxi regulárních výrazů objevují, my se zde ale bez nich obejdeme.)

Jaké jazyky lze vlastně popsat regulárními výrazy? Nikoho jistě nepřekvapí, že to jsou regulární jazyky, tedy třídou jazyků reprezentovatelných regulárními výrazy je třída REG. V jednom směru nám to vlastně ihned plyne z dřívějších poznatků:

Věta 2.32
Ke každému regulárnímu výrazu α lze sestrojit konečný automat přijímající jeho jazyk $[\alpha]$.

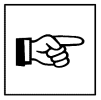
Důkaz: Stačí samozřejmě ukázat, že k zadanému regulárnímu výrazu α lze sestrojit zobecněný nedeterministický konečný automat A_α takový, že jazyk $[\alpha]$ je roven $L(A_\alpha)$.

Pro jazyky $\emptyset$, $\{\varepsilon\}$, $\{a\}$ lze triviálně zkonstruovat příslušný (ZN)KA; pro další technickou jednoduchost můžeme konstruovat automaty s jediným počátečním stavem, do něhož nevchází žádná šipka a s jediným přijímajícím stavem, z něhož nevychází žádná šipka. (Navrhněte takové automaty pro elementární regulární výrazy!)

K výrazům $(\alpha + \beta)$, $(\alpha \cdot \beta)$ a (α^*) umíme konstruovat automaty z automatů A_α, A_β podle vět 2.3 a 2.27.

Obrázek 2.15 ovšem navíc názorně ukazuje, že lze používat konstrukce zachovávající u konstruovaných automatů, že mají právě jeden počáteční stav bez vstupních šipek a právě jeden přijímající stav bez výstupních šipek.

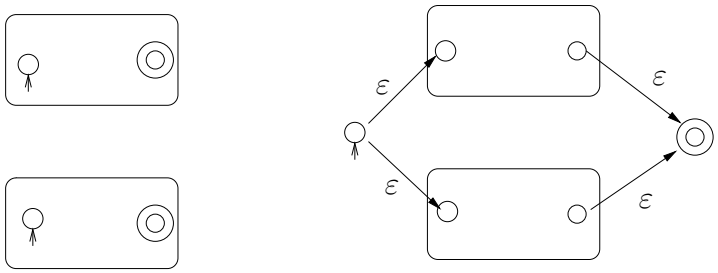
□



Cvičení 2.80: Aplikujte algoritmus naznačený na obrázku 2.15 na regulární výraz $((01^*0 + 101)^*100 + (11)^*0)^*01$. (Můžete samozřejmě průběžně vypouštět ty ε-šipky, které jsou očividně nadbytečné.)

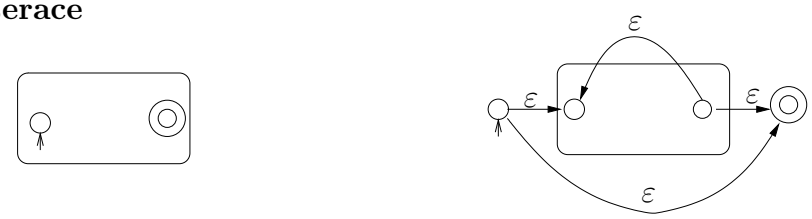
Komentář: Všimněme si, že konstrukce automatu pro sjednocení na obrázku 2.15 ukazuje alternativní důkaz věty 2.3. Výsledný automat je sice i pro deterministické vstupní automaty (zobecněný) nedeterministický, ale

Sjednocení



Zřetězení

Iterace



Obrázek 2.15: Konstrukce ZNKA k regulárnímu výrazu

jeho počet stavů je jen o 2 větší než je součet stavů vstupních automatů (jeho stavovou množinou není kartézský součin stavů vstupních automatů). Obrázek 2.15 tak navíc názorně ukazuje, že počet stavů ZNKA A_α zkonstruovaného k regulárnímu výrazu α je úměrný délce α. (To je aspekt, který by už neplatil, kdybychom v regulárních výrazech používali symboly pro průnik či doplněk.)

K odvození avízované věty

Věta 2.33
Regulárními výrazy lze reprezentovat právě regulární jazyky.

bychom ještě potřebovali ukázat, že ke každému konečnému automatu A existuje (lze algoritmicky) sestrojit regulární výraz α_A tak, že jazyk $[\alpha_A]$ je

roven jazyku $L(A)$. Zde se spokojíme jen s poznámkou, že důkaz lze nalézt v rozšiřující části. Pro nás je teď důležité, že víme, jak k regulárnímu výrazu zkonstruovat ekvivalentní konečný automat. (A víme to i pro regulární výrazy rozšířené o průnik, doplněk apod.)

Otázky:

Otázka 2.81: Je zápis jazyka regulárním výrazem jednoznačný, nebo jinak, lze jeden jazyk zapsat různými výrazy?

Cvičení 2.82: Jak zapíšete regulárním výrazem jazyk všech slov, kde za počátečním úsekem znaků a se může jednou (ale nemusí vůbec) objevit znak c a pak následuje úsek znaků b?

Cvičení 2.83: A co když v předchozí úloze vyžadujeme, že úsek a i úsek b musí být neprázdný?

Cvičení 2.84: A co když v předchozí úloze ještě povolíme, že znak c se mezi a a b může vyskytnout 0-, 1- nebo 2-krát?

Cvičení 2.85: Zjistěte, zda jsou jazyky

$[(011 + (10)^*1 + 0)^*]$ a $[011(011 + (10)^*1 + 0)^*]$

stejné.

Cvičení 2.86: Zjistěte, zda jsou jazyky

$[((1 + 0)^*100(1 + 0)^*)^*]$ a $[((1 + 0)100(1 + 0)^*100)^*]$

stejné.

Cvičení 2.87*: Zadejte regulárním výrazem jazyk
$L = \{ w \in \{0, 1\}^* \mid$ ve w je sudý počet nul a každá jednička je bezprostředně následována nulou $\}$

Cvičení 2.88: Procvičte si regulární výrazy i tím, že jimi popíšete některé další regulární jazyky, s nimiž se v našem textu setkáváte.

Cvičení 2.89: Sestrojte konečný automat (třeba nedeterministický) přijímající jazyk zapsaný výrazem $(0 + 11)^*01$.

Cvičení 2.90: Upravte předchozí automat, aby přijímal jazyk zapsaný $(0 + 11)^*00^*1$.

Cvičení 2.91*: Zapište regulárním výrazem jazyk všech slov nad abecedou $\{0, 1\}$, která neobsahují tři stejné znaky za sebou.

Cvičení 2.92: Zapište regulárním výrazem jazyk všech slov nad abecedou $\{a, b, c\}$, ve kterých se nikde nevyskytují znaky a, b hned za sebou (ani ab, ani ba).

Cvičení 2.93: Zapište regulárním výrazem jazyk všech slov nad abecedou $\{a, b, c\}$, ve kterých se nikde nevyskytují dva znaky a hned za sebou.

Cvičení 2.94: Zapište regulárním výrazem jazyk všech slov nad abecedou $\{a, b, c\}$, ve kterých je po a vždy b a po b vždy a.

Cvičení 2.95: Zapište regulárním výrazem jazyk všech slov nad abecedou $\{a, b, c\}$, ve kterých je po a vždy b a po b nikdy není c.

Cvičení 2.96*: Zapište regulárním výrazem jazyk všech slov nad abecedou $\{a, b, c\}$, ve kterých je podslovo aa a není podslovo cc.

Cvičení 2.97*: Mějme dva regulární jazyky K a L popsané regulárními výrazy

$$K = [\, 0^*1^*0^*1^*0^* \,], \qquad L = [\, (01 + 10)^* \,].$$

a) Jaké je nejkratší a nejdelší slovo v průniku $L \cap K$?

b) Proč žádný z těchto jazyků K a L není podmnožinou toho druhého?

c) Jaké je nejkratší slovo, které nepatří do sjednocení $K \cup L$? Je to jednoznačné?

Všechny vaše odpovědi dobře zdůvodněte!

Shrnutí: Víme tedy, co to jsou základní (teoretické) regulární výrazy a umíme jimi popisovat (jednoduché) jazyky. Jsme si vědomi, že regulární výrazy umějí popsat právě regulární jazyky a máme dobře promyšlen převod regulárního výrazu na (nedeterministický) konečný automat. Je nám jasné, že tento převod se dá rozšířit i na regulární výrazy rozšířené např. o operace průniku a doplňku.

Kapitola 3

Bezkontextové jazyky

Cíle kapitoly:

Po prostudování této kapitoly máte dobře znát pojmy bezkontextová gramatika a zásobníkový automat. Máte umět navrhovat a analyzovat jednoduché bezkontextové gramatiky a zásobníkové automaty, být si vědomi jejich vzájemného vztahu, souvislosti s programovacími jazyky apod.

Klíčová slova: *bezkontextové gramatiky a jazyky, zásobníkové automaty*

3.1 Motivační příklad

Orientační čas ke studiu této části: 1 hod.

Cíle této části:

V této části máte získat představu o jazycích, jejichž slova mají (syntaktickou) strukturu definovanou pravidly tzv. bezkontextové gramatiky. Rovněž máte získat první představu o tom, že při rozboru (a překladu) slov takových jazyků se uplatní tzv. zásobníkový automat.

Ze studia regulárních jazyků víme, že každý takový jazyk lze popsat regulárním výrazem a k regulárnímu výrazu umíme zkonstruovat příslušný konečný automat, který pak může sloužit jako nástroj vyhledávání; např. lze zjišťovat, které dokumenty (slova) vyhovují zadanému regulárnímu výrazu (tedy patří do jazyka popsaného oním regulárním výrazem).

Připomeňme si například, že umíme rychle mechanicky zkonstruovat (nedeterministický) konečný automat, který rozpoznává jazyk popsaný regulárním výrazem

$$R = (ab)^*bbb + (aa + b)^* + ba^*b\,.$$

Nemáme s tím problém, protože ihned vidíme, že výraz R se dá chápat jako „součet“ (reprezentující sjednocení) dvou jednodušších výrazů, konkrétně

$$R = R_1 + R_2, \text{ kde } R_1 = (ab)^*bbb,\ R_2 = (aa + b)^* + ba^*b\,,$$

a stačí tedy zkonstruovat automaty pro R_1, R_2 a ty pak známou konstrukcí pro sjednocení spojit do výsledného automatu.

Úkol konstrukce automatu pro R_1 si pochopitelně také můžeme rozložit na menší úkoly; vidíme např., že R_1 je vlastně zřetězením

$$R_1 = R_{11} \cdot R_{12}, \text{ kde } R_{11} = (ab)^*,\ R_{12} = bbb\,,$$

a automat pro R_1 lze tedy získat konstrukcí automatů pro R_{11}, R_{12} a jejich následným spojením známou konstrukcí odpovídající zřetězení. Pochopitelně automat pro R_{11} lze získat z automatu pro (pod)výraz ab aplikací známé konstrukce pro iteraci, atd.

Je očividné, že takto provádíme de facto mechanickou (algoritmickou) činnost. Zamysleme se teď nad tím, jak lze příslušný algoritmus popsat tak, aby šel již přímočaře naprogramovat a my jsme tedy mohli svěřit konstrukci automatu k regulárnímu výrazu počítači.

Takový algoritmus musí být mimo jiné schopen zjistit, zda zadaná posloupnost symbolů v abecedě $\Delta = \{\, a, b, \varepsilon, \emptyset, +, \cdot, ^*, (,)\,\}$ je skutečně (správně utvořeným) regulárním výrazem. (Omezujeme se zde na popis jazyků v abecedě $\Sigma = \{a, b\}$.) V případě, že zadaná posloupnost regulárním výrazem není (např. ‘$bb*+\cdot(ba))+a$’), běh algoritmu by měl skončit ohlášením chyby (např. vstupem do „chybového“ stavu či podobně).

Již z toho je jasné, že náš algoritmus musí být postaven na něčem obecnějším než je konečný automat. Jazyk $RV(\{a,b\})$ sestávající z těch slov v abecedě Δ, která jsou správně utvořenými regulárními výrazy, totiž regulární není.

Kontrolní otázka: Proč (pro žádnou abecedu Σ) není jazyk $RV(\Sigma)$ regulární?

Jistě jste si při zodpovídání otázky uvědomili, že je to díky používání závorek. Stačí si uvědomit, že kvocienty $RV(\Sigma)$ podle $($, $(($, $((($, ... jsou jistě navzájem různé. (Pro $i \neq j$ levý kvocient jazyka $RV(\{a,b\})$ podle slova $(^i$ obsahuje např. slovo $a)^i$, ale levý kvocient jazyka $RV(\{a,b\})$ podle slova $(^j$ slovo $a)^i$ neobsahuje.)

Poznamenejme, že uplatňujeme naši dřívější dohodu o možném vynechávání závorek, které nezpůsobí nejednoznačnost díky asociativitě sjednocení a zřetězení a díky dohodnuté prioritě operátorů ('*' váže silněji než '$\cdot$' a '$\cdot$' váže silněji než '$+$'). Rovněž umožňujeme vynechávání symbolu '$\cdot$'. (Tyto dohody jsme již tiše uplatnili u uvedení a diskuse regulárního výrazu $R = (ab)^*bbb + (aa + b)^* + ba^*b$ výše.)

Úmluva. Do $RV(\{a,b\})$ zde zařazujeme i výrazy zjednodušené v souladu s našimi dohodami.

Poznámka: Naše pozorování o neregularitě jazyka $RV(\{a,b\})$ samozřejmě platí i v případě, že bychom se omezili jen na regulární výrazy bez nadbytečných závorek. Výše uvedenou úvahu ukazující, že kvocientů je nekonečně mnoho, bychom museli jen trochu upravit.

Napadne nás přesto (tj. přes uvedenou neregularitu) způsob, jak zadané slovo z Δ^* přečíst zleva doprava a rozhodnout, zda patří do $RV(\{a,b\})$? Po chvilce přemýšlení jistě dojdeme na to, že při čtení (potenciálního) regulárního výrazu zleva doprava budeme muset kontrolovat, zda se neobjeví nesmyslná dvojice sousedních symbolů apod. (viz následující Cvičení) – ale také něco dalšího.

Cvičení 3.1: Vyjmenujte všechna slova délky 2 (v abecedě Δ), která se nemohou v regulárním výrazu vyskytovat. (Např. '$+$)'.) Kterými symboly nemůže regulární výraz začínat? Kterými nemůže končit? Snažte se pak výstižně charakterizovat, kdy slovo v abecedě Δ neobsahující žádnou 'zakázanou' dvojici ani 'zakázaný' první či poslední symbol přesto není regulárním výrazem (tedy není prvkem $RV(\{a,b\})$).

Vyřešením cvičení jste si jistě uvědomili, že kromě kontroly drobných 'regulárních podmínek' (což může provádět malý konečný automat), si stačí průběžně pamatovat počet otevřených závorek, tedy rozdíl mezi počty dosud přečtených symbolů '(' a ')'. Tento rozdíl musí být samozřejmě stále nezáporný a po přečtení celého vstupního řetězce musí být nulový. Potřebujeme tedy jakýsi čítač (proměnnou ukládající nezáporná celá čísla); jeho hodnoty ovšem nemůžeme nijak omezit (proto také si hodnoty takového čítače nemůže pamatovat konečný automat svým stavem).

Teď je nám už tedy jasné, jak může algoritmus přečtením zleva doprava a použitím datové struktury „neomezený čítač" ověřit, zda zadané slovo $w \in \Delta^*$ je regulárním výrazem (prvkem $RV(\{a, b\})$).

Pozastavme se ještě u té datové struktury 'neomezený čítač' a všimněme si, že nám postačují tři elementární operace: přičtení jedničky, odečtení jedničky – v případě, že je aktuální hodnota kladná (jinak pokus o odečtení způsobí chybu, výpočet se 'zasekne') a test, zda je hodnota nulová (což v našem případě použijeme po přečtení celého slova k potvrzení uzavřenosti všech závorek). Zkusme na chvíli uvažovat jisté zobecnění regulárních výrazů, kdy umožníme kromě závorek '$($', '$)$' používat i závorky '$[$', '$]$' (např. pro větší přehlednost u složitějších výrazů); příkladem je $[(ab)^*bbb + ((aa)^* + b)]^* + [bbabb]^*$. Při používání dvou (či více) typů závorek nám už obecně pouhý čítač nestačí.

Kontrolní otázka: Proč datová struktura čítač v tomto případě nepostačuje?

Jistě přijdeme na to, že si v případě dvou typů závorek musíme pamatovat i jejich pořadí. K tomu se hodí datová struktura, které budeme říkat *zásobník* (anglicky *stack* či *pushdown*). Hodnota uložená v zásobníku není číslo, ale řetězec symbolů z nějaké konečné množiny (tedy slovo v jisté abecedě). Elementárními operacemi je přidání symbolu či ubrání symbolu – ale *obojí se děje jen na jednom konci.*

Poznámka: Jedná se tedy o typ LIFO (Last In First Out).

V našem konkrétním případě stačí ukládat symboly '$($', '$[$'. Např. při čtení řetězce $[(ab)^*bbb + ((aa)^* + b)]^* + [bbabb]^*$ by hodnota zásobníku (u něhož přidáváme a ubíráme na pravém konci) postupně byla

ε , [, [(, [, [(, [((, [(, [, ε , [, ε .

Brzy zavedeme pojem *zásobníkový automat*; bude to de facto konečný automat obohacený o datovou strukturu zásobník. Ukáže se, že tyto zásobníkové automaty rozpoznávají třídu tzv. *bezkontextových jazyků*, což je nadtřída třídy regulárních jazyků.

Poznámka: Později objasníme, proč budou v základní verzi zásobníkové automaty definovány jako nedeterministické.

Když se vrátíme k našemu motivačnímu úkolu, uvědomíme si, že pro algoritmickou konstrukci konečného automatu k zadanému regulárnímu výrazu potřebujeme podstatně více než jen algoritmus zjištění, zda zadaný řetězec je opravdu (správně utvořeným) regulárním výrazem. Potřebujeme totiž rovněž algoritmický rozbor *syntaktické struktury zadaného regulárního výrazu*, o niž se pak konstrukce kýženého automatu může opřít. Když se např. zjistí, že zadaný výraz R je součtem $R_1 + R_2$, pak lze konstrukci vést tak, že se zkonstruují automaty pro R_1, R_2 a ty se pak spojí konstrukcí pro sjednocení; atp.

Syntaktická struktura regulárního výrazu je ovšem přirozeně dána pravidly, která jsme použili v definici pojmu regulární výraz; tuto definici si teď připomeneme.

Regulárními výrazy nad abecedou Σ rozumíme množinu $RV(\Sigma)$ slov v abecedě $\Sigma \cup \{\, \emptyset, \varepsilon, +, \cdot, ^*, (,) \,\}$ (kde předpokládáme, že $\emptyset, \varepsilon, +, \cdot, ^*, (,) \notin \Sigma$), která splňuje tyto podmínky:

- Symboly $\emptyset, \varepsilon$ a symbol a pro každé písmeno $a \in \Sigma$ jsou prvky $RV(\Sigma)$; tyto symboly také nazýváme *elementárními regulárními výrazy*.
- Jestliže $\alpha, \beta \in RV(\Sigma)$, pak také $(\alpha + \beta) \in RV(\Sigma)$, $(\alpha \cdot \beta) \in RV(\Sigma)$ a $(\alpha^*) \in RV(\Sigma)$.
- $RV(\Sigma)$ neobsahuje žádné další řetězce, tedy do $RV(\Sigma)$ patří právě ty řetězce (výrazy), které jsou konstruovány z $\emptyset, \varepsilon$ a písmen abecedy Σ výše uvedenými pravidly.

Tato pravidla se dají přirozeně zachytit následovně; omezíme se přitom na abecedu $\Sigma = \{a, b\}$.

$$R \longrightarrow \emptyset \mid \varepsilon \mid a \mid b \mid (R + R) \mid (R \cdot R) \mid (R^*)$$

Tento souhrn pravidel je příkladem *bezkontextové gramatiky*; takové gramatiky jsou základním prostředkem popisu výše zmíněných bezkontextových jazyků.
V dalších sekcích uvedené pojmy nadefinujeme přesně, důkladněji je prozkoumáme; v rozšiřující části se pak vrátíme k úplnému dořešení našeho motivačního problému (tedy k dokončení algoritmu, který zkonstruuje konečný automat k regulárnímu výrazu).

Shrnutí: Uvědomili jsme si, že se např. v programátorské praxi běžně setkáme s jazyky, které nejsou regulární, ale na jejichž rozpoznávání (tj. ke zjišťování, zda do takového jazyka patří zadaný řetězec) postačí konečný automat obohacený o (jeden) zásobník. Zároveň jsme si uvědomili, že pro „překlad“ (např. regulárního výrazu na ekvivalentní konečný automat) se potřebujeme podrobněji podívat na syntaktickou strukturu slov zdrojového jazyka, kterou lze často zadat pravidly tzv. bezkontextové gramatiky.

3.2 Bezkontextové gramatiky a jazyky

Orientační čas ke studiu této části: 3 hod.

Cíle této části:

Po prostudování této části máte být schopni definovat a ilustrovat pojmy jako je bezkontextová gramatika, odvození podle dané gramatiky, derivační strom apod. Rovněž máte být schopni porozumět zadaným (jednoduchým) gramatikám a také je sami navrhovat.

Klíčová slova: *bezkontextová gramatika, odvození (derivace), derivační strom, bezkontextový jazyk*

V předešlé sekci jsme si mj. uvědomili, že jazyk regulárních výrazů (nad danou abecedou) není regulární, ale dá se definovat syntaktickými pravidly, kterým dohromady říkáme bezkontextová gramatika. Než si uvedeme formální definice, podíváme se ještě na jiný podobný jazyk (de facto fragment programovacího jazyka).

Uvažujme jazyk aritmetických výrazů vytvořených z prvků abecedy $\Sigma = \{\, 0, 1, \ldots, 9, +, \times, (,) \,\}$ Příklady takových výrazů jsou $805 + 23 \times 91$ nebo $(805 + 23) \times 91$. Po dřívějších úvahách (o závorkách) je nám hned jasné, že se nejedná o regulární jazyk. Způsobem běžným v programátorské praxi (v manuálech programovacích jazyků apod.) bychom uvedený jazyk mohli popsat např. těmito (přepisovacími) pravidly:

$$\begin{aligned}
&\langle EXPR\rangle \longrightarrow \langle EXPR\rangle + \langle EXPR\rangle\\
&\langle EXPR\rangle \longrightarrow \langle EXPR\rangle \times \langle EXPR\rangle\\
&\langle EXPR\rangle \longrightarrow (\ \langle EXPR\rangle\ )\\
&\langle EXPR\rangle \longrightarrow \langle NUMB\rangle\\
&\langle NUMB\rangle \longrightarrow \langle DIGIT\rangle\\
&\langle NUMB\rangle \longrightarrow \langle DIGIT\rangle\langle NUMB\rangle\\
&\langle DIGIT\rangle \longrightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}$$

Výrazy $\langle EXPR\rangle, \langle NUMB\rangle, \langle DIGIT\rangle$ jsou zde chápány jako (tři) speciální symboly, které nejsou prvky uvedené abecedy Σ. Slouží nám jako pomocné (syntaktické) proměnné, kterým se také říká *neterminální symboly* či zkráceně *neterminály*; odlišují se tak od tzv. *terminálních symbolů* neboli *terminálů*, tj. prvků abecedy jazyka, který takto definujeme (v našem případě tedy od prvků Σ). Na konkrétní volbě neterminálních symbolů nezáleží, jen se musejí odlišovat od terminálů, což můžeme vyjádřit tak, že terminální abeceda a neterminální abeceda musí být disjunktní (jejich průnik je prázdný).

Použijeme-li místo symbolů $\langle EXPR\rangle, \langle NUMB\rangle, \langle DIGIT\rangle$ neterminály E, N, D, vypadá výše uvedená soustava pravidel takto:

$$\begin{aligned}
&E \longrightarrow E + E \mid E \times E \mid (E) \mid N\\
&N \longrightarrow D \mid DN\\
&D \longrightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}$$

Zde jsme sloučili všechna pravidla s levou stranou E a všechna pravidla s levou stranou N vždy do jednoho řádku, přičemž jednotlivé pravé strany jsme oddělili speciálním znakem „|“; to je běžná praxe, kterou jsme již výše využili pro pravidla s levou stranou $\langle DIGIT\rangle$. (Znak „|“, podobně jako „$\longrightarrow$“, pak pochopitelně nepoužíváme jako terminál či neterminál.)

Uveďme si příklad *odvození*, neboli *derivace*, slova $(81+2)\times 35$; zápis $\alpha \Rightarrow \beta$ můžeme číst „z řetězce α lze v jednom kroku odvodit β“:

$E \Rightarrow E \times E \Rightarrow (E) \times E \Rightarrow (E + E) \times E \Rightarrow (N + E) \times E \Rightarrow (N + N) \times E \Rightarrow (N + N) \times N \Rightarrow (DN + N) \times N \Rightarrow \ldots \Rightarrow (81 + 2) \times 35.$

Kontrolní otázka: Umíte uvedené odvození zkompletovat ?

Všimněme si teď, že množina všech terminálních slov (tj. řetězců terminálních symbolů), které lze odvodit z neterminálu N, je regulárním jazykem (obsahujícím neprázdné posloupnosti číslic). My si teď další úvahy zjednodušíme tím, že se zaměříme na gramatiku (tj. soustavu pravidel)

$$E \longrightarrow E + E \mid E \times E \mid (E) \mid a$$

kde symbol a je chápán jako terminál.

Poznámka: V překladačích programovacích jazyků nejdříve probíhá tzv. *lexikální analýza* založená de facto na konečném automatu, při níž by při zpracování výrazů typu $(801 + 23) \times 35$ byly řetězce $801, 23$ apod. rozpoznány (a zpracovány) jako lexikální jednotky („atomy“). Pro jednoduchost si teď představme, že všechny atomy tohoto typu (tedy řetězce číslic) byly nahrazeny symbolem a; výsledné řetězce, které vstupují do následné *syntaktické analýzy* jsou tedy typu $(a + a) \times a$ apod.

Jedno možné odvození (neboli derivace) slova $a + a \times a$ (podle posledně uvedené gramatiky) je toto:

$E \Rightarrow E + E \Rightarrow a + E \Rightarrow a + E \times E \Rightarrow a + a \times E \Rightarrow a + a \times a\,.$

Jedná se o příklad tzv. *levé derivace*, kdy jsme v každém kroku přepisovali nejlevější neterminál (či přesněji řečeno: nejlevější výskyt neterminálního symbolu). Uveďme příklad *pravé derivace* pro totéž slovo:

$E \Rightarrow E + E \Rightarrow E + E \times E \Rightarrow E + E \times a \Rightarrow E + a \times a \Rightarrow a + a \times a$

A ještě příklad derivace, která není ani levá ani pravá:

$E \Rightarrow E + E \Rightarrow E + E \times E \Rightarrow E + a \times E \Rightarrow a + a \times E \Rightarrow a + a \times a$

Je zřejmé, že se vlastně ve všech třech případech jedná o jedno a totéž odvození – jen pořadí přepisování neterminálů je různé. Strukturu onoho odvození nezávislou na konkrétním pořadí přepisování neterminálů zachycuje tzv. *strom odvození*, neboli *derivační strom*. V našem případě je derivační strom odpovídající všem třem uvedeným derivacím znázorněn na Obrázku 3.1 vlevo.

Obrázek 3.1: Derivační stromy

Slovo $a + a \times a$ má ovšem i jinou *levou* derivaci než tu uvedenou výše, a sice:

$E \Rightarrow E \times E \Rightarrow E + E \times E \Rightarrow a + E \times E \Rightarrow a + a \times E \Rightarrow a + a \times a$

Této derivaci odpovídá *jiný* derivační strom – je zachycen na Obrázku 3.1 vpravo.

Existence dvou různých derivačních stromů (neboli dvou různých levých derivací) pro jedno slovo jazyka, je nežádoucí vlastnost – příslušná gramatika (tj. soubor přepisovacích pravidel) je *nejednoznačná*. K tomuto problému se vrátíme později.
Teď je načase zavést přesné definice; neměly by být pro nás žádným překvapením, protože zachycují pojmy, které jsme na intuitivní úrovni již pochopili.

Definice 3.1
Bezkontextová gramatika je definována jako uspořádaná čtveřice $G = (\Pi, \Sigma, S, P)$, kde

- Π je konečná množina *neterminálních symbolů* (neterminálů)
- Σ je konečná množina *terminálních symbolů* (terminálů), přičemž $\Pi \cap \Sigma = \emptyset$
- $S \in \Pi$ je *počáteční* (startovací) *neterminál*
- P je konečná množina *pravidel* typu $A \rightarrow \beta$, kde
  - A je neterminál, tedy $A \in \Pi$
  - β je řetězec složený z terminálů a neterminálů, tedy $\beta \in (\Pi \cup \Sigma)^*$.

Poznámka: Slovo „bezkontextová“ v názvu gramatiky znamená, že na levé straně každého pravidla stojí jeden neterminál bez sousedních symbolů; ten můžeme při jakémkoli odvození přepsat podle uvedených pravidel nezávisle na jeho okolí, tedy „bez ohledu na kontext“.

Značení: Běžná konvence je, že jako neterminály používáme velká písmena $(A, B, C, \ldots)$; terminály jsou typicky malá písmena $(a, b, c, \ldots)$ či další symboly (jako $+, (,)$ apod.).

Definice 3.2
Mějme gramatiku $G = (\Pi, \Sigma, S, P)$ a uvažujme řetězce $\gamma, \delta \in (\Pi \cup \Sigma)^*$. Řekneme, že γ *lze přímo přepsat na* (či *přímo odvodí*) δ (podle pravidel gramatiky G), značíme $\gamma \Rightarrow_G \delta$ nebo jen $\gamma \Rightarrow \delta$ když G zřejmá z kontextu, jestliže existují slova μ_1, μ_2 a pravidlo $A \to \beta$ v P tak, že $\gamma = \mu_1 A \mu_2$, $\delta = \mu_1 \beta \mu_2$.

Řekneme, že γ *lze přepsat na* (*odvodí*) δ, značíme $\gamma \Rightarrow^* \delta$, jestliže existuje posloupnost $\mu_0, \mu_1, \ldots, \mu_n$ slov z $(\Pi \cup \Sigma)^*$ (pro nějaké $n \geq 0$) tž. $\gamma = \mu_0 \Rightarrow \mu_1 \Rightarrow \ldots \Rightarrow \mu_n = \delta$. Zmíněnou posloupnost pak nazveme *odvozením* (*derivací*) délky n slova δ ze slova γ.

Definice 3.3
Jazyk generovaný gramatikou G označujeme $L(G)$; je to množina $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$.

Jazyk L je *bezkontextový*, jestliže existuje bezkontextová gramatika G taková, že $L(G) = L$.

Uvedené definice tedy formalizují pojmy, které jsme si již neformálně ilustrovali dříve. Doplníme ještě několik poznámek.

- Zdůrazněme, že do jazyka generovaného gramatikou patří pouze *terminální slova* odvozená z S.
- Relace $\Rightarrow$ zachycuje pojem „odvození v jednom kroku“ a relace $\Rightarrow^*$ pojem „odvození v konečně mnoha krocích“. Všimněme si, že pro každý řetězec α platí $\alpha \Rightarrow^* \alpha$ (jedná se o odvození v 0 krocích).
- Zápis $\gamma \Rightarrow^* \delta$ čteme také „δ dostaneme z γ“, „γ generuje δ“ apod.

- V řeči algebry jsou $\Rightarrow$ a $\Rightarrow^*$ (binární) relace na množině $(\Pi \cup \Sigma)^*$ (tedy na množině všech slov tvořených terminálními i neterminálními symboly). Relace $\Rightarrow^*$ je *reflexivním a tranzitivním uzávěrem* relace $\Rightarrow$.

Pokračujeme formalizací pojmů levá derivace a derivační strom.

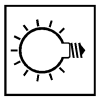
Definice 3.4
Mějme bezkontextovou gramatiku $G = (\Pi, \Sigma, S, P)$. Řekneme, že α lze přepsat na β *levým přepsáním*, jestliže $\alpha = uX\delta$, $\beta = u\gamma\delta$ pro nějaké $u \in \Sigma^*$, $\delta \in (\Pi \cup \Sigma)^*$ a pravidlo $X \to \gamma$ v P. Odvození $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$ je levým odvozením (levou derivací), jestliže pro vš. $i = 0, 1, \ldots, n-1$ lze α_i přepsat na α_{i+1} levým přepsáním. (Pravé odvození lze definovat obdobně.)

Poznámka: Lze snadno ukázat, že platí-li $X \Rightarrow_G^* w$, pak w lze z X odvodit (nějakým) levým odvozením i (nějakým) pravým odvozením.

Definice 3.5
Derivační strom, vztahující se k bezkontextové gramatice $G = (\Pi, \Sigma, S, P)$, je (konečný) uspořádaný kořenový strom (tj. souvislý graf bez cyklů, s vyznačeným vrcholem - kořenem, následníci každého vrcholu jsou uspořádáni „zleva doprava“), jehož vrcholy jsou ohodnoceny symboly z $\Pi \cup \Sigma \cup \{\varepsilon\}$. Přitom kořen je ohodnocen symbolem S a dále platí, že každý vrchol ohodnocený neterminálem $X \in \Pi$ musí mít následníky, kteří odpovídají pravé straně nějakého pravidla $X \to Y_1 Y_2 \ldots Y_n$ v P; jsou tedy (zleva doprava) ohodnoceni $Y_1, Y_2, \ldots, Y_n$ ($Y_i \in \Pi \cup \Sigma$). V případě pravidla $X \to \varepsilon$ se jedná o jednoho následníka ohodnoceného ε.
Vrcholy ohodnocené terminály (či slovem ε) jsou listy (nemají tedy následníky). Řekneme, že se jedná o *derivační strom pro slovo* w, jestliže w je zřetězením ohodnocení listů (v uspořádání zleva doprava).

Poznámka: Všimněme si, že každému odvození slova w v gramatice G odpovídá (přirozeným způsobem) právě jeden derivační strom pro w; derivačnímu stromu pro w odpovídá obecně více odvození slova w, ovšem např. právě jedno levé odvození.

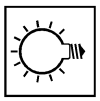
Je čas na procvičení zavedených pojmů. Nejtěžší jsou samozřejmě opět úkoly vyžadující jistou míru kreativity, tedy návrhy bezkontextových gramatik pro konkrétní jazyky.



Řešený příklad 3.1: Navrhněme gramatiku generující množinu booleovských formulí s proměnnými typu $x0$, $x1$, $x2$, ..., se spojkami $\wedge, \vee, \neg$ (a se závorkami „(“, „)“).

Řešení: To je typ úkolu z „programátorského manuálu“. Uvědomíme si, že každá formule F je buď proměnná, nebo negace nějaké (menší) formule, nebo konjunkce dvou formulí, nebo disjunkce dvou formulí, anebo vznikne uzávorkováním (menší) formule. Tento rozbor vede přímočaře k následující gramatice (kde F je počáteční neterminál).

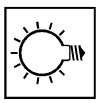
$$\begin{aligned} F &\longrightarrow V \mid \neg F \mid F \wedge F \mid F \vee F \mid (F) \\ V &\longrightarrow xN \\ N &\longrightarrow D \mid DN \\ D &\longrightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \end{aligned}$$



Řešený příklad 3.2: Navrhněme gramatiku generující jazyk palindromů $L_1 = \{w \in \{a,b\}^* \mid w = w^R\}$ (kde w^R značí zrcadlový obraz slova w).

Řešení: Jako vždy, nejdříve si úkol musíme důkladně ujasnit. V prvé řadě se tedy podíváme na slova z L_1. Do L_1 jistě patří ε a každé jednoznakové slovo, tedy a i b. Ze slov délky dvě tam patří jen aa a bb, u slov délky tři se jedná o aaa, aba, bab, bbb. atd. Jistě si takto uvědomíme, že delší palindrom začíná a končí stejným písmenem a mezi nimi je kratší palindrom. Tato úvaha vede přímočaře k následující gramatice s jediným neterminálem:

$$S \longrightarrow \varepsilon \mid a \mid b \mid aSa \mid bSb\,.$$



Řešený příklad 3.3: Co nejvýstižněji popište jazyk generovaný gramatikou

$$S \longrightarrow SbS \mid a\,.$$

Řešení: Úkol pochopení zadané gramatiky může být taky náročný, pokud je gramatika „neprůhledná“. (Možná vám to připomene náročnost pochopení cizího neokomentovaného programu.) Pokud nám řešení není hned jasné, můžeme nejdříve systematicky prozkoumat nejkratší derivace terminálních slov

podle dané gramatiky a doufat, že takto získáme dostatečný vhled, který nám pak už umožní charakterizovat všechna generovaná slova. V našem případě tak zjistíme, že gramatika generuje slova a, aba, $ababa$ atd., a postupně nás jistě napadne hypotéza, že všechna generovaná slova jsou všechna slova ve tvaru „$abab \ldots aba$“. Když si např. uvědomíme, že každou derivaci terminálního slova můžeme přeskupit tak, že v první fázi je používáno jen pravidlo $S \longrightarrow SbS$ a v druhé fázi jen pravidlo $S \longrightarrow a$, je to už definitivně jasné: po první fázi dostaneme slovo „$SbSb \ldots SbS$“ a v druhé fázi se všechny výskyty S nahradí a.

Poznámka: Všimněme si, že bezkontextová gramatika v předchozím příkladu generuje regulární jazyk. Již z dřívějšího výkladu plyne, že každý regulární jazyk je bezkontextový, ale ne naopak. V rozšiřující části se zmíníme o *regulárních gramatikách*, což jsou speciální bezkontextové gramatiky generující právě regulární jazyky.

Otázky:

Otázka 3.2: Může v bezkontextové gramatice mít jedno slovo nekonečně mnoho odvození?

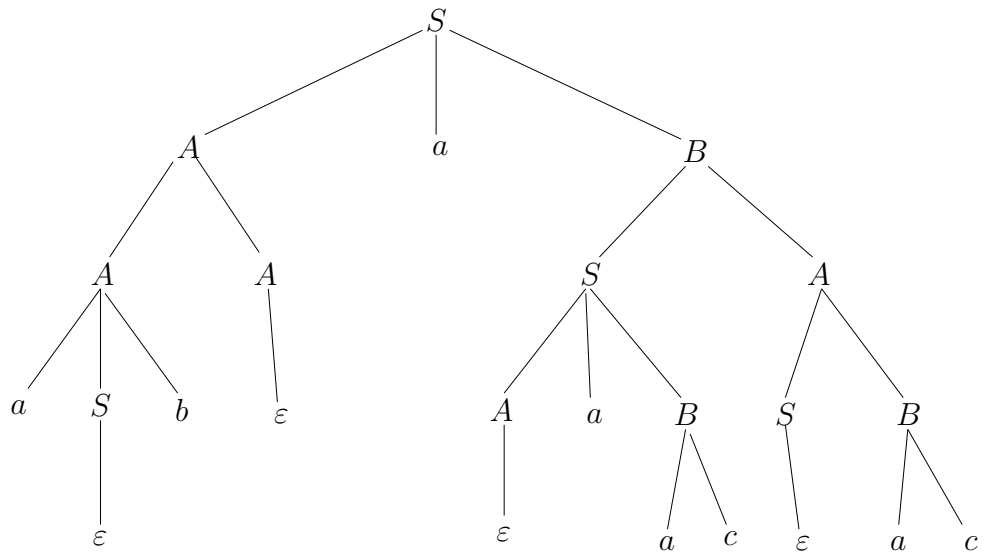
Otázka 3.3: Jaký jazyk generuje gramatika $SaS \mid Sb$?

Cvičení 3.4: Na Obrázku 3.2 je derivační strom popisující odvození slova $w = abaaacac$ podle jisté bezkontextové gramatiky G.

- Vypište pravidla gramatiky G, jejichž existenci můžeme vyvodit z uvedeného derivačního stromu.
- Napište levé odvození slova w podle gramatiky G.
- Napište pravé odvození slova w podle gramatiky G.

Cvičení 3.5: Upravte gramatiku z Příkladu 3.2 tak, aby generovala jen palindromy sudé délky.

Cvičení 3.6: Zjistěte, jaká (terminální) slova generuje gramatika



Obrázek 3.2: Derivační strom pro slovo $w = abaaacac$

$$S \longrightarrow aBC \mid aCa \mid bBCa$$
$$B \longrightarrow bBa \mid bab \mid SS$$
$$C \longrightarrow BS \mid aCaa \mid bSSc$$

Cvičení 3.7: Generuje gramatika

$$S \longrightarrow abSa \mid \varepsilon$$

stejný jazyk jako gramatika

$$S \longrightarrow aSa \mid bS \mid \varepsilon \ \ ?$$

Cvičení 3.8: Navrhněte bezkontextové gramatiky generující následující jazyky:

- $L_1 = \{\, w \in \{a, b\}^* \mid w$ obsahuje podslovo $baab \,\}$
- $L_2 = \{\, w \in \{a, b\}^* \mid |w|_b \bmod 3 = 0 \,\}$

- $L_3 = \{\, ww^R \mid w \in \{a, b\}^* \,\}$
- $L_4 = \{\, 0^n 1^m 0^n \mid m, n \geq 0 \,\}$
- $L_5 = \{\, 0^n 1^m \mid 1 \leq n \leq m \leq 2n \,\}$

Shrnutí: Pojmu bezkontextová gramatika a souvisejícím pojmům již plně rozumíme a nedělá nám problémy rozbor a návrh gramatik generujících (jednoduché) bezkontextové jazyky.

3.3 Jednoznačné gramatiky

Orientační čas ke studiu této části: 1 hod.

Cíle této části:

Máte porozumět pojmu jednoznačná gramatika, získat schopnost poznat (ne)jednoznačnost u konkrétních gramatik. Rovněž máte pochopit postup, jak se odstraňuje nejednoznačnost u jednoduchých příkladů z programátorské praxe.

Klíčová slova: *jednoznačná gramatika, víceznačná gramatika, jednoznačný jazyk*

Připomeňme si obrázek 3.1, který ukazuje dva různé derivační stromy pro slovo $a + a \times a$ podle gramatiky

$$G_1 : \quad E \longrightarrow E + E \mid E \times E \mid (E) \mid a\,.$$

Zmínili jsme se, že je to nežádoucí vlastnost; naznačíme proč. Na derivační strom se můžeme dívat jako na výsledek syntaktické analýzy, na jehož základě se budou provádět další operace; v našem případě vyhodnocení aritmetického výrazu.

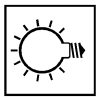
Poznámka: Podrobněji o takovém vyhodnocení, či generování kódu v nižším programovacím jazyku (assembleru), který ono vyhodnocení realizuje, pojednáme v rozšiřující části.

S pravidlem $E \longrightarrow E + E$ aplikovaným v uzlu derivačního stromu přirozeně pojíme akci „vyhodnoť následníky a příslušné výsledky sečti“, zatímco s pravidlem $E \longrightarrow E \times E$ pojíme akci „vyhodnoť následníky a příslušné výsledky vynásob“. V tomto smyslu jistě nahlížíme, že stromy na obrázku 3.1 dávají různé návody k vyhodnocení.
Tyto úvahy nás vedou k následující definici.

Definice 3.6
Řekneme, že *bezkontextová gramatika* G je *jednoznačná*, jestliže každé slovo z $L(G)$ má právě jedno levé odvození (tj. právě jeden derivační strom). V opačném případě je G *nejednoznačná* (či *víceznačná*).

Výše uvedená gramatika G_1 tedy není jednoznačná. My ovšem příslušná slova umíme jednoznačně syntakticky rozebrat; zápis $a + a \times a$ chápeme jako ekvivalentní zápisu $a + (a \times a)$, protože tiše předpokládáme běžně dohodnutou prioritu násobení před sčítáním. Zamysleme se nad otázkou, zda takovou prioritu lze modelovat pravidly bezkontextové gramatiky; v kladném případě sestrojíme jednoznačnou gramatiku G_2, která je ekvivalentní G_1 v tom smyslu, že generuje tentýž jazyk. Provedeme si to ve formě řešeného příkladu.

Definice 3.7
Dvě bezkontextové *gramatiky* G_1, G_2 nazveme *ekvivalentní*, jestliže $L(G_1) = L(G_2)$.



Řešený příklad 3.4: Navrhněte ke gramatice

$$G_1: \quad E \longrightarrow E + E \mid E \times E \mid (E) \mid a$$

ekvivalentní jednoznačnou gramatiku.

Řešení: Všimněme si, že zápis v každého aritmetického výrazu (námi uvažovaného typu) buď je tvaru $v_1 + v_2$, kde v_1, v_2 jsou dva (jednodušší) výrazy, nebo v tomto tvaru není. V prvním případě řekneme, že v je *součtem*, a navíc volíme v_1 tak, že tento součtem není (tedy vezmeme nejlevější $+$ ve v, pro nějž na levé i pravé straně jsou správně utvořené výrazy); v druhém případě nazveme v *termem*.

Např. výraz $a + a \times a$ je součtem termů a a $a \times a$, protože ho chápeme jako $a + (a \times a)$; výraz $(a + a) \times a$ ovšem součtem není, celý je jedním termem. Ani výraz $(a + (a \times a))$ není chápán jako součet, kvůli vnějším závorkám. Dále např. $a \times a + a + a$ lze sice vyjádřit dvěma způsoby jako součet, ale díky naší úmluvě ho chápeme jako součet termu $a \times a$ a výrazu $a + a$. (Zde tedy „asociujeme doprava“, neboli $v_1 + v_2 + v_3$ chápeme jako $v_1 + (v_2 + v_3)$.)

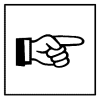
Podobně každý term t buď je nebo není *součinem* $t_1 \times t_2$; v kladném případě zase bereme nejlevější výskyt $\times$, pro nějž jsou na levé i pravé straně správně utvořené výrazy. Výraz, který není ani součtem ani součinem nazveme *faktorem*.

Tyto úvahy nás přirozeně vedou k následujícím pravidlům

$$E \longrightarrow T + E \mid T$$
$$T \longrightarrow F \times T \mid F$$
$$F \longrightarrow (E) \mid a$$

a zároveň jsme těmito úvahami vlastně již prokázali, že vzniklá gramatika G_2 je ekvivalentní původní gramatice ($L(G_2) = L(G_1)$) a je navíc jednoznačná. (Např. slovo $a+a\times a$ má v G_2 jen jednu levou derivaci (a tedy jediný derivační strom): $E \Rightarrow T+E \Rightarrow T+T \Rightarrow F+T \Rightarrow a+T \Rightarrow a+F\times T \Rightarrow a+F\times F \Rightarrow a+a\times F \Rightarrow a+a\times a$.)

Poznámka: V rozšiřující části ukážeme, že ne každou nejednoznačnou gramatiku lze převést na ekvivalentní jednoznačnou. Existují tedy tzv. *(vnitřně) nejednoznačné bezkontextové jazyky*. Např. jazyk $L = \{a^i b^j c^k \mid i = j$ nebo $j = k\}$ negeneruje žádná jednoznačná gramatika.



Cvičení 3.9: Ukažte, že jazyk $L = \{a^i b^j c^k \mid i = j$ nebo $j = k\}$ diskutovaný v předchozí poznámce je bezkontextový.

Cvičení 3.10: Lze v úkolu 3.4 z dostupné informace zjistit něco ohledně jednoznačnosti příslušné gramatiky?

Shrnutí: Máme alespoň představu o tom, proč je (např. pro překlad programovacích jazyků) důležitá jednoznačnost gramatik. Umíme nadefinovat pojem jednoznačné bezkontextové gramatiky a rozpoznáme (ne)jednoznačnost

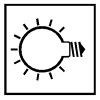
jednoduchých gramatik (z programátorské praxe). Umíme ilustrovat převod nejednoznačných gramatik v programátorské praxi na jednoznačné gramatiky. Zaznamenali jsme také fakt, že některé bezkontextové jazyky nelze generovat jednoznačnou gramatikou.

3.4 Cvičení

Orientační čas ke studiu této části: 1 hod.

Cíle této části:

Cílem je opět určité zopakování a procvičení nabytých znalostí. Tomu má napomoci promyšlení a vyřešení dalších otázek a příkladů.



Řešený příklad 3.5: Sestrojte bezkontextovou gramatiku generující všechna slova nad abecedou $\{a, b\}$ mající stejně výskytů symbolů a jako b, tedy gramatiku generující jazyk $L = \{w \in \{a, b\}^* \mid |w|_a = |w|_b\}$.

Řešení: Možná by čtenáře mohlo napadnout používat pravidla jako $S \longrightarrow abS \mid baS$ nebo i složitější pravidla podobného typu, která zajistí stejný počet a jako b; problémem ovšem je zajistit *úplnost*, tedy vygenerování *všech* požadovaných slov (tedy i těch s dlouhými úseky $aa \ldots a$ apod.).

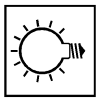
Jedna možnost řešení je „nacpat S všude“, tedy použít pravidla

$$S \longrightarrow SaSbS \mid SbSaS \mid \varepsilon$$

která také zaručeně odvozují jen slova se stejným počtem výskytů a a b, a pak dokazovat úplnost např. indukcí podle délky slova.

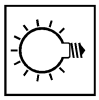
Jako vždy, lepší možnost je ovšem o problému více zapřemýšlet, napsat si pár slov z daného jazyka, a pak si třeba uvědomit, že slovo w z jazyka L je buď prázdné nebo je zřetězením dvou kratších slov z L, a pokud není zřetězením dvou slov z L, pak jeho první a poslední symbol jsou různé a mezi nimi je slovo z L. Tyto úvahy vedou k následující gramatice.

$$S \longrightarrow \varepsilon \mid SS \mid aSb \mid bSa$$



Řešený příklad 3.6: Zjistěme, zda některá z gramatik z předcházejícího příkladu je jednoznačná.

Řešení: Důkladnější zamyšlení by mělo napovědět, že tomu tak nebude. V obou případech jistě nalezneme dvě různé levé derivace např. pro slovo *abab*. V případě první gramatiky jde např. o derivace $S \Rightarrow SaSbS \Rightarrow aSbS \Rightarrow abS \Rightarrow abSaSbS \Rightarrow abaSbS \Rightarrow ababS \Rightarrow abab$ a $S \Rightarrow SaSbS \Rightarrow aSbS \Rightarrow aSbSaSbS \Rightarrow abSaSbS \Rightarrow abaSbS \Rightarrow ababS \Rightarrow abab$; u druhé gramatiky jde např. o derivace $S \Rightarrow SS \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSb \Rightarrow abab$ a $S \Rightarrow aSb \Rightarrow abSab \Rightarrow abab$.



Řešený příklad 3.7: Ověřme, zda následující gramatiky generují tentýž jazyk.

$$G1: \quad S \longrightarrow aaSbb \mid ab \mid aabb$$
$$G2: \quad S \longrightarrow aSb \mid ab$$

Řešení: Gramatika G_2 zřejmě generuje jazyk $\{a^i b^i \mid i \geq 1\}$. Každá derivace podle G_1 nejdříve užívá několikrát (včetně možnosti 0-krát) pravidlo $S \longrightarrow aaSbb$, čímž se odvodí $a^{2k}Sb^{2k}$ pro nějaké $k \geq 0$, a pak skončí buď pravidlem $S \longrightarrow ab$ nebo pravidlem $S \longrightarrow aabb$, tedy odvozením terminálního slova $a^{2k+1}b^{2k+1}$ nebo $a^{2k+2}b^{2k+2}$. Gramatika G_1 tedy rovněž vyvodí všechna slova typu $a^i b^i$, kde $i \geq 1$. Obě gramatiky tedy generují stejný jazyk (jsou ekvivalentní).

Cvičení 3.11: Navrhněte gramatiku generující jazyk $\{\, a^i b^j \mid i < j \,\}$.

Cvičení 3.12: Navrhněte bezkontextovou gramatiku generující jazyk všech palindromů v abecedě $\{a, b\}$, jejichž délka je násobkem čtyř.

Cvičení 3.13*: Navrhněte bezkontextovou gramatiku generující jazyk všech palindromů v abecedě $\{a, b\}$, jejichž délka je násobkem tří.

Cvičení 3.14: Generují obě následující gramatiky tentýž jazyk?

$$G_1: \quad S \longrightarrow aaSbb \mid ab \mid \varepsilon$$
$$G_2: \quad S \longrightarrow aSb \mid ab$$

Cvičení 3.15: Generují obě následující gramatiky tentýž jazyk?

$$G_1: \quad S \longrightarrow aaSb \mid ab \mid \varepsilon$$
$$G_2: \quad S \longrightarrow aSb \mid aab \mid \varepsilon$$

Cvičení 3.16: Zjistěte, které z následujících gramatik generují regulární jazyk (tedy jazyk přijímaný také konečným automatem).

a) $S \longrightarrow aSb \mid bSa \mid \varepsilon$

b) $S \longrightarrow abS \mid baS \mid \varepsilon$

c) $S \longrightarrow ASa \mid \varepsilon$; $A \longrightarrow b$

d) $S \longrightarrow BSa \mid \varepsilon$; $B \longrightarrow a$

e) $S \longrightarrow aSb \mid bSa \mid bbS$

f) $S \longrightarrow ab \mid ba \mid bbS$

Cvičení 3.17*: Navrhněte gramatiku generující jazyk všech těch slov nad abecedou $\{a, b, c\}$, ve kterých za každým úsekem znaků a bezprostředně následuje dvakrát delší úsek znaků b.

3.5 Zásobníkové automaty

Orientační čas ke studiu této části: 3 hod.

Cíle této části:

Po prostudování této části máte být schopni definovat a ilustrovat pojmy jako je zásobníkový automat, jeho výpočet (posloupnost konfigurací), jazyk rozpoznávaný zásobníkovým automatem. Rovněž máte být schopni analyzovat zadané (jednoduché) zásobníkové automaty a sami je navrhovat. Máte si uvědomit vztah bezkontextových gramatik a zásobníkových automatů a pochopit konstrukci zásobníkového automatu ke gramatice, na níž je založena syntaktická analýza (např. při překladech programovacích jazyků).

Klíčová slova: *zásobníkový automat, výpočet, přijímání slova a jazyka*

V sekci 3.1 jsme si uvědomili, že např. jazyk všech regulárních výrazů nad danou abecedou není regulární, tedy není rozpoznatelný konečným automatem, ale lze jej rozpoznávat pomocí přidané datové struktury zvané zásobník.

Podobně je tomu např. u (jednoduššího) jazyka

$$\begin{array}{ll}\{ & \langle begin\rangle\langle end\rangle,\\ & \langle begin\rangle\langle begin\rangle\langle end\rangle\langle end\rangle,\\ & \langle begin\rangle\langle begin\rangle\langle begin\rangle\langle end\rangle\langle end\rangle\langle end\rangle,\\ & \ldots \quad \}\end{array}$$

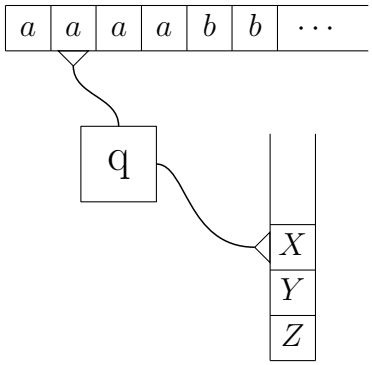
či jazyka $L = \{a^n b^n \mid n \geq 1\}$.

Kontrolní otázka: Jakým způsobem lze rozpoznávat slova z L pomocí zásobníku?

Asi vás napadl přímočarý způsob: přečtené symboly a se ukládají do zásobníku a při čtení symbolů b se pak tyto zásobníkové symboly odebírají. Tímto způsobem jsme schopni počet a-ček a b-ček porovnat.

„Vnější pohled“ na *zásobníkový automat* je ilustrován Obrázkem 3.3.

Máme již jistě dobrou představu o tom, jak takové „zařízení“, skládající se z (konečně-stavové) *řídicí jednotky*, *zásobníku* a *vstupní pásky* čtené zleva doprava, pracuje a jakým způsobem reprezentuje (rozpoznává) jazyk. Tuto představu je ovšem opět potřebné zpřesnit, ať se odstraní jakékoli neurčitosti či nejednoznačnosti. To činí následující definice; zdůrazněme hned, že obecným termínem „zásobníkový automat“ se obvykle myslí *nedeterministický* zásobníkový automat. (Důvody vysvitnou později.)



Obrázek 3.3: Vnější pohled na zásobníkový automat

Definice 3.8
Zásobníkový automat (zkráceně ZA) M je definován jako šestice $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$, kde

- Q je konečná neprázdná množina *stavů*,
- Σ je konečná neprázdná množina *vstupních symbolů* (vstupní abeceda),
- Γ je konečná neprázdná množina *zásobníkových symbolů* (zásobníková abeceda),
- $q_0 \in Q$ je *počáteční stav*,
- $Z_0 \in \Gamma$ je *počáteční zásobníkový symbol* a
- δ je zobrazení množiny $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ do množiny všech konečných podmnožin množiny $Q \times \Gamma^*$.

Než se pustíme do podrobnějšího vysvětlení, definice přijímání jazyka apod., uvedeme si příklad zásobníkového automatu, který bude přijímat jazyk $L = \{a^n b^n \mid n \geq 1\}$. Podívejme se na následující *instrukce*:

$(r_1, a, Z) \to (r_1, A)$
$(r_1, a, A) \to (r_1, AA)$

$(r_1, b, A) \to (r_2, \varepsilon)$
$(r_2, b, A) \to (r_2, \varepsilon)$

Vzpomeňme si, že bezkontextovou gramatiku jsme normálně zadávali jen pravidly; co je množinou terminálů, množinou neterminálů a který neterminál je počáteční se dalo odvodit podle našich dohod o používaných symbolech (a dohody, že počáteční je neterminál na levé straně prvního pravidla). Většinou jsme tedy explicitně nevyjmenovávali všechny součásti gramatiky $G = (\Pi, \Sigma, S, P)$, ale jen zapsali pravidla z P.
Podobně je tomu u zásobníkového automatu.

Kontrolní otázka: Říká vám intuice již teď, jak lze (podle naši budoucí dohody) vyvodit součásti zásobníkového automatu $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ zadaného výše uvedenými instrukcemi?

Jistě jste uhodli, že množina stavů je $Q = \{r_1, r_2\}$ a asi jste usoudili, že r_1 (tedy první uvedený stav v instrukcích) je počáteční stav (tedy $q_0 = r_1$); máte také jistě správnou představu, že

řídicí jednotka je na začátku výpočtu v počátečním stavu.

Vstupní abeceda Σ je samozřejmě množinou $\{a, b\}$.
Dále jste jistě vyvodili, že množina zásobníkových symbolů je zde $\Gamma = \{Z, A\}$; přitom jste asi usoudili, že symbol Z uvedený na levé straně první instrukce je počáteční zásobníkový symbol (tedy $Z_0 = Z$). Možná jste také již pochopili, že podle našich definic výpočet nebude začínat s prázdným zásobníkem;

zásobník na začátku výpočtu obsahuje právě (jeden) počáteční zásobníkový symbol.

Jistě jste také usoudili, že uvedená sada instrukcí reprezentuje přechodovou funkci δ; zápis $(q, a, X) \to (q', \alpha)$ znamená, že $\delta(q, a, X) \ni (q', \alpha)$.

Podívejme se teď na

význam instrukce $(q, a, X) \to (q', \alpha)$ (kde $a \in \Sigma$):

- Tato instrukce je aplikovatelná jen v situaci (neboli konfiguraci), kdy řídicí jednotka je ve stavu q, čtecí hlava na vstupní pásce čte symbol a a na vrcholu zásobníku je symbol X.

- Pokud je instrukce skutečně aplikována, vykoná se následující:
  - řídicí jednotka přejde do stavu q',
  - čtecí hlava na vstupní pásce se posune o jedno políčko doprava,
  - vrchní symbol v zásobníku se odebere (vymaže),
  - na vrchol zásobníku se přidá řetězec α tak, že jeho nejlevější symbol je aktuálním vrcholem zásobníku.

V rozšiřující části definujeme výpočet zásobníkového automatu exaktně pomocí relace odvození na konfiguracích. Zde se spokojíme s intuitivním pochopením, založeném na následujícím příkladu. Výpočet výše uvedeného zásobníkového automatu na vstupním slově *aaabbb* lze zapsat takto:

$(r_1, aaabbb, Z) \vdash (r_1, aabbb, A) \vdash (r_1, abbb, AA) \vdash (r_1, bbb, AAA) \vdash (r_2, bb, AA) \vdash (r_2, b, A) \vdash (r_2, \varepsilon, \varepsilon)$

Výpočet je tedy posloupnost konfigurací, kde každá jednotlivá *konfigurace* zachycuje

- (aktuální) stav řídicí jednotky,
- obsah vstupní pásky, který zbývá přečíst,
- obsah zásobníku (zapsaný jako řetězec symbolů; nejlevější symbol odpovídá vrcholu zásobníku).

Jak jste jistě vyrozuměli, zápis $K_1 \vdash K_2$ znamená, že z konfigurace K_1 lze přejít do K_2 provedením jedné (aplikovatelné) instrukce.

Později využijeme i tzv. *ε-kroky*, tedy instrukce typu

$$(q, \varepsilon, X) \to (q', \alpha)\,.$$

Význam je obdobný jako u instrukce $(q, a, X) \to (q', \alpha)$, kde $a \in \Sigma$, ale zde se ze vstupu nic nečte (změna stavu a vrcholu zásobníku tedy na vstupním symbolu nezávisí) a čtecí hlava se neposunuje.

Možná jste si již dříve všimli, že v definici 3.8 nejsou žádné přijímající stavy. Úspěšné (tj. přijímající) výpočty a potažmo jazyk rozpoznávaný zásobníkovým automatem jsou definovány následovně.

Definice 3.9
Výpočet zásobníkového automatu $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ *je přijímajícím*, jestliže skončí v konfiguraci $(q, \varepsilon, \varepsilon)$ $(q \in Q)$, tedy když se přečte celé vstupní slovo a vyprázdní se zásobník (přičemž na dosaženém stavu nezáleží).

Slovo $w \in \Sigma^*$ *je přijímáno* automatem M, jestliže existuje přijímající výpočet začínající v konfiguraci (q_0, w, Z_0).

Jazyk $L(M)$, tedy *jazyk rozpoznávaný* (též říkáme *přijímaný*) *zásobníkovým automatem* M, je definován takto:
$L(M) = \{w \in \Sigma^* \mid w$ je přijímáno automatem $M\}$.

Poznámka: Některé varianty zásobníkových automatů v literatuře předpokládají i přijímající stavy; výpočet je pak přijímající, jestliže po přečtení vstupního slova řídicí jednotka vstoupí do přijímajícího stavu, přičemž na (ne)prázdnosti zásobníku nezáleží. Tato definice je ekvivalentní té naší, jak si ukážeme v rozšiřující části; zde se ovšem přidržíme přijímání prázdným zásobníkem.

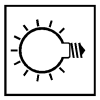
Poznámka: Jak jsme již zmínili, definice zahrnuje nedeterminismus – k danému vstupnímu slovu může existovat více výpočtů. U našeho automatu pro jazyk $L = \{a^n b^n \mid n \geq 1\}$ to nenastává (automat je deterministický), ale brzy se k užití nedeterminismu dostaneme.

Kontrolní otázka: Když už teď máme přesné definice, umíte ověřit, že zásobníkový automat zadaný dříve uvedenými instrukcemi skutečně rozpoznává jazyk $L = \{a^n b^n \mid n \geq 1\}$?

Není problémem se přesvědčit, že pro každé slovo $a^n b^n$, $n \geq 1$, existuje přijímající výpočet. Co se děje, když vstupní slovo není z jazyka L ? Všimněme si, že pokud je např. řídicí jednotka ve stavu r_2 a čtecí hlava čte na vstupu a, není aplikovatelná žádná instrukce; jinými slovy $\delta(r_2, a, X) = \emptyset$ pro každé $X \in \Gamma$. Výpočet prostě dál nemůže pokračovat; proto např. slovo *aababb* není přijato. Podobně $\delta(r_1, b, Z) = \emptyset$; začíná-li tedy vstupní slovo symbolem b, nelze provést žádnou instrukci. Všimněme si, že ani ε není přijato, zásobník totiž na začátku není prázdný. Je-li vstupní slovo tvaru $a^m b^n$ pro $m > n$, výpočet skončí rovněž s neprázdným zásobníkem, tedy neúspěšně. Pro slovo $a^m b^n$, kde $1 \leq m < n$, skončí výpočet předčasným vyprázdněním zásobníku – předtím, než se dočte vstupní slovo. Zdůrazněme, že podle naší definice platí, že

je-li zásobník prázdný, není aplikovatelná žádná instrukce.

Zatím je možná nejasné, proč jsme v definici 3.8 uváděli typ přechodové funkce δ jako zobrazení množiny $Q\times(\Sigma\cup\{\varepsilon\})\times\Gamma$ do množiny všech konečných podmnožin množiny $Q \times \Gamma^*$, a proč jsme nepoužili např. typ $Q \times \Sigma \times \Gamma \to Q\times\Gamma^*$. Podívejme se v této souvislosti nejdříve na následující řešené příklady.



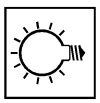
Řešený příklad 3.8: Navrhněte zásobníkový automat rozpoznávající jazyk $\{a^ib^jc^k \mid i,j,k \geq 1 \text{ a } i = j\}$.

Řešení: Není těžké přijít na níže uvedenou modifikaci instrukcí použitých pro $\{a^nb^n \mid n \geq 1\}$:

$(r_1, a, Z) \to (r_1, AZ)$
$(r_1, a, A) \to (r_1, AA)$
$(r_1, b, A) \to (r_2, \varepsilon)$
$(r_2, b, A) \to (r_2, \varepsilon)$
$(r_2, c, Z) \to (r_3, Z)$
$(r_3, c, Z) \to (r_3, Z)$
$(r_3, c, Z) \to (r_3, \varepsilon)$

Počáteční symbol Z jsme teď ponechali na dně zásobníku, aby výpočet neskončil předčasně jeho vyprázdněním. Tento symbol Z zároveň zajišťuje, že při čtení prvního vstupního c může výpočet pokračovat jen tehdy, když před ním předcházelo a^nb^n pro nějaké $n \geq 1$. Poprvé jsme také použili *nedeterminismu*: v konfiguraci se stavem r_3, čteným symbolem c a vrcholem zásobníku Z jsou aplikovatelné dvě různé instrukce; musíme totiž počítat jak s možností, že čtené c je posledním znakem vstupního slova, tak s možností, že nějaká c ještě následují. Pro přechodovou funkci δ zde tedy platí $\delta(r_3, c, Z) = \{(r_3, Z), (r_3, \varepsilon)\}$.

Poznámka: Někdy je užitečné si představit, či to takto přímo definovat, že vstupní slovo je zakončeno speciálním symbolem označujícím konec slova. Pak by uvedené využití nedeterminismu nebylo nutné; brzy se ale dostaneme k případům, kdy je využití nedeterminismu zásadní.

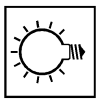


Řešený příklad 3.9: Navrhněte zásobníkový automat rozpoznávající jazyk $\{a^ib^jc^k \mid i,j,k \geq 1 \text{ a } j = k\}$.

Řešení: Tady už zkušeně navrhneme např.

$(q_1, a, Z) \to (q_2, Z)$
$(q_2, a, Z) \to (q_2, Z)$
$(q_2, b, Z) \to (q_2, B)$
$(q_2, b, B) \to (q_2, BB)$
$(q_2, c, B) \to (q_3, \varepsilon)$
$(q_3, c, B) \to (q_3, \varepsilon)$

Zdůrazněme, že první dvě instrukce není možné nahradit jedinou; podle definice jazyka musí slovo začínat alespoň jedním znakem a.



ŘEŠENÝ PŘÍKLAD 3.10: Navrhněte zásobníkový automat rozpoznávající jazyk $\{a^i b^j c^k \mid i, j, k \geq 1 \text{ a } i = j \text{ nebo } j = k\}$.

Řešení: Teď se v plné míře uplatní nedeterminismus (i kdyby vstupní slova byla zakončena speciálním symbolem). (V rozšiřující části se ještě zmíníme, že deterministickým zásobníkovým automatem uvedený jazyk rozpoznávat nelze. Můžeme si rovněž připomenout, že tento jazyk negeneruje žádná jednoznačná gramatika.)

Samozřejmě se nabízí využití automatů zkonstruovaných v předchozích příkladech. Můžeme elegantně využít dosud nepoužitou možnost *ε-kroků*. Dejme instrukce z předchozích dvou příkladů prostě dohromady a přidejme nový stav q_0, definovaný jako počáteční, a instrukce

$(q_0, \varepsilon, Z) \to (r_1, Z)$
$(q_0, \varepsilon, Z) \to (q_1, Z)$

Důležité je, že stavy ze dvou předchozích množin instrukcí se nepomíchají, neboť jsme je volili v obou příkladech různé; jinak bychom je prostě v jednom případě dodatečně přejmenovali.

Výsledný automat se tedy na začátku „nedeterministicky rozhodne“, zda bude porovnávat počty výskytů znaku a a znaku b nebo počty výskytů znaku b a znaku c. Víme, že existence neúspěšných (tj. nepřijímajících) výpočtů na správném slově (tedy slově z jazyka) nevadí, podstatné je, že alespoň jeden výpočet přijímající je.

Je načase uvést, že zásobníkové automaty tvoří „automatový protějšek“ k bezkontextovým gramatikám; to vyjadřuje následující věta.

Věta 3.10
Zásobníkové automaty rozpoznávají právě bezkontextové jazyky (a jsou takto ekvivalentní bezkontextovým gramatikám).

Zde je podstatné, že zásobníkové automaty obecně chápeme jako nedeterministické. Jazyky rozpoznatelné deterministickými zásobníkovými automaty tvoří totiž vlastní podtřídu třídy bezkontextových jazyků; na rozdíl od konečných automatů má tedy u zásobníkových automatů nedeterministická verze větší rozpoznávací sílu než deterministická. (K této problematice se vrátíme v rozšiřující části.) Právě kýžená ekvivalence s bezkontextovými gramatikami je důvodem, proč u zásobníkových automatů bereme nedeterministickou verzi jako základní, a proč jsme příslušně definovali i typ přechodové funkce v Definici 3.8.

Větu 3.10 dokážeme až v rozšiřující části, zde jen uvedeme konstrukci zásobníkového automatu k zadané gramatice, která mimo jiné ilustruje základní ideu syntaktické analýzy v překladačích (konkrétně pro metodu „shora dolů“ („top-down“)).

Konstrukce zásobníkového automatu ke gramatice

Vstup: bezkontextová gramatika G.

Výstup: zásobníkový automat M (s jedním stavem) takový, že $L(M) = L(G)$.

Postup: Nechť $G = (\Pi, \Sigma, S, P)$.
Definujme $M = (\{q_0\}, \Sigma, \Pi \cup \Sigma, \delta, q_0, S)$, kde

- pro každé $X \in \Pi$ je $\delta(q_0, \varepsilon, X) = \{(q_0, \alpha) \mid (X \to \alpha) \in P\}$,
- pro každé $a \in \Sigma$ je $\delta(q_0, a, a) = \{(q_0, \varepsilon)\}$;
- jiným argumentům přiřazuje δ prázdnou množinu.

Např. pro gramatiku (s očíslovanými pravidly)

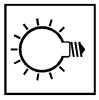
$$
\begin{aligned}
&1/\ A \longrightarrow A + B\\
&2/\ A \longrightarrow B\\
&3/\ B \longrightarrow B \times C\\
&4/\ B \longrightarrow C\\
&5/\ C \longrightarrow (A)\\
&6/\ C \longrightarrow a
\end{aligned}
$$

je příslušný zásobníkový automat zadán instrukcemi

$$
\begin{array}{l}
(q_0, \varepsilon, A) \to (q_0, A + B) \\
(q_0, \varepsilon, A) \to (q_0, B) \\
(q_0, \varepsilon, B) \to (q_0, B \times C) \\
(q_0, \varepsilon, B) \to (q_0, C) \\
(q_0, \varepsilon, C) \to (q_0, (A)\,) \\
(q_0, \varepsilon, C) \to (q_0, a) \\
(q_0, a, a) \to (q_0, \varepsilon) \\
(q_0, +, +) \to (q_0, \varepsilon) \\
(q_0, \times, \times) \to (q_0, \varepsilon) \\
(q_0, (, (\,) \to (q_0, \varepsilon) \\
(q_0,),)\,) \to (q_0, \varepsilon)
\end{array}
$$

Poznámka: Jak vidíme, takto zkonstruovaný zásobníkový automat k zadané gramatice (mohutně) využívá nedeterminismu. V praktických algoritmech založených na uvedené konstrukci se ovšem používají různé modifikace vedoucí k deterministickému automatu. (To ale není možné u všech gramatik, jak jsme již zmínili dříve.)

Uveďme teď ještě další příklady konstrukce zásobníkového automatu, v nichž rovněž použijeme přirozené schéma, umožňující stručnější zadávání skupiny instrukcí.



Řešený příklad 3.11: Navrhněte zásobníkový automat rozpoznávající jazyk $L = \{\, wc(w)^R \mid w \in \{a, b\}^* \,\}$.

Řešení: Jistě nás napadne ukládat úsek slova do (prvního) výskytu c do zásobníku, a pak porovnávat symbol po symbolu s následujícím úsekem. Nesmíme také zapomenout, že automat musí přijmout i slovo c.

$$
\begin{array}{l}
(r_1, a, Z) \to (r_1, A) \\
(r_1, b, Z) \to (r_1, B) \\
(r_1, c, Z) \to (r_1, \varepsilon) \\
(r_1, a, A) \to (r_1, AA) \\
(r_1, b, A) \to (r_1, BA) \\
(r_1, c, A) \to (r_2, A) \\
(r_1, a, B) \to (r_1, AB) \\
(r_1, b, B) \to (r_1, BB) \\
(r_1, c, B) \to (r_2, B)
\end{array}
$$

$(r_2, a, A) \to (r_2, \varepsilon)$
$(r_2, b, B) \to (r_2, \varepsilon)$

Dá se snadno ověřit, že automat má přijímající výpočet pro každé slovo z L; naopak pro jakékoli slovo, které není z L, se výpočet nutně „zasekne“ před dočtením vstupního slova, nebo skončí s neprázdným zásobníkem.

Připomeňme ještě, že zásobníkové symboly nemusí být jiné než vstupní symboly. (Vstupní abeceda Σ a zásobníková abeceda Γ v Definici 3.8 nemusí být disjunktní, na rozdíl od množin terminálů a neterminálů u gramatik.) My jsme zde použili symboly A, B jen pro určité zvýraznění, že jde o zásobníkové symboly.
Všimněme si dále, že některé skupiny instrukcí jsou podobné a lze je stručněji zachytit *schémata instrukcí*. Např. místo dvou instrukcí

$(r_1, a, A) \to (r_1, AA)$
$(r_1, a, B) \to (r_1, AB)$

lze stručněji psát

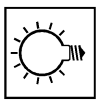
$(r_1, a, X) \to (r_1, AX)$, pro $X \in \{A, B\}$.

Pokud použijeme a, b i jako zásobníkové symboly, lze zásobníkový automat pro náš jazyk zapsat následovně:

$(r_1, x, Z) \to (r_1, x)$, pro $x \in \{a, b\}$
$(r_1, c, Z) \to (r_1, \varepsilon)$

$(r_1, x, y) \to (r_1, xy)$, pro $x, y \in \{a, b\}$
$(r_1, c, x) \to (r_2, x)$, pro $x \in \{a, b\}$

$(r_2, x, x) \to (r_2, \varepsilon)$, pro $x \in \{a, b\}$



Řešený příklad 3.12: Navrhněte zásobníkový automat rozpoznávající jazyk $L = \{ w(w)^R \mid w \in \{a, b\}^* \}$.

Řešení: Pochopitelně nás hned napadne, že bychom mohli nějak využít automat z předchozího příkladu. Teď ale nemáme střed slova označený c, jak to

tedy vyřešit? S použitím nedeterminismu snadno: v instrukcích z předchozího příkladu prostě místo c napíšeme ε.

Všimněme si, že posledně zmíněný nedeterminismus se zde neprojevuje v tom, že bychom měli dvě různé instrukce se stejnou levou stranou $(q, a, X) \to \ldots$, ale v tom, že máme instrukci typu $(q, a, X) \to \ldots$ i instrukci typu $(q, \varepsilon, X) \to \ldots$, kde a je vstupní symbol. (V našem případě máme např. instrukci $(r_1, b, a) \to (r_1, ba)$ i instrukci $(r_1, \varepsilon, a) \to (r_2, a)$.) I tento jev totiž vede k tomu, že existuje více možných výpočtů pro jedno vstupní slovo.

Tento příklad také ilustruje dříve zmíněnou větší rozpoznávací sílu nedeterministických (tedy našich základních) zásobníkových automatů oproti automatům deterministickým. Dá se totiž dokázat, že jazyk $\{ w(w)^R \mid w \in \{a, b\}^* \}$ deterministickým zásobníkovým automatem (v němž by tedy ke každému vstupnímu slovu existoval jen jeden výpočet) rozpoznávat nelze.

Otázky:

Otázka 3.18: Jakým způsobem může zásobníkový automat rozpoznat prázdné slovo?

Otázka 3.19: Jak můžeme zásobníkovým automatem simulovat konečný automat?

Cvičení 3.20: Demonstrujte přijímající výpočet (nedeterministického) zásobníkového automatu, kterým jsme ilustrovali konstrukci BG→ZA, pro vstupní slovo $a \times (a + a)$.

Cvičení 3.21: Sestrojte zásobníkový automat rozpoznávající jazyk $L = \{ u \in \{a, b, c\}^* \mid$ po vynechání všech výskytů symbolu c z u dostaneme slovo ve tvaru $w(w)^R \}$.

Cvičení 3.22: Charakterizujte co nejjednodušeji jazyk, který je přijímán zásobníkovým automatem $M = (Q, \Sigma, \Gamma, \delta, p, Z)$, kde $Q = \{p, q\}, \Sigma = \{a, b, c\}, \Gamma = \{A, B, Z\}$ a δ je zadána následujícím výčtem. (Zde jsme nepoužili přepis na instrukce jako dříve, což je samozřejmě nepodstatné.)
$\delta(p, a, Z) = \{(p, AZ)\}$
$\delta(p, b, Z) = \{(p, BZ)\}$

$\delta(p, c, Z) = \{(p, Z)\}$
$\delta(p, a, A) = \{(p, AA)\}$
$\delta(p, b, A) = \{(p, BA)\}$
$\delta(p, c, A) = \{(p, A)\}$
$\delta(p, a, B) = \{(p, AB)\}$
$\delta(p, b, B) = \{(p, BB)\}$
$\delta(p, c, B) = \{(p, B)\}$
$\delta(p, \varepsilon, Z) = \{(q, \varepsilon)\}$
$\delta(p, \varepsilon, A) = \{(q, A)\}$
$\delta(p, \varepsilon, B) = \{(q, B)\}$
$\delta(q, a, A) = \{(q, \varepsilon)\}$
$\delta(q, c, A) = \{(q, A)\}$
$\delta(q, b, B) = \{(q, \varepsilon)\}$
$\delta(q, c, B) = \{(q, B)\}$
$\delta(q, c, Z) = \{(q, Z)\}$
$\delta(q, \varepsilon, Z) = \{(q, \varepsilon)\}$

Cvičení 3.23: Charakterizujte co nejjednodušeji jazyk, který je přijímán zásobníkovým automatem $M = (Q, \Sigma, \Gamma, \delta, q, S)$, kde $Q = \{q\}, \Sigma = \{a, b, c\}, \Gamma = \{A, B, C, S\}$ a δ je zadána následujícím výčtem.

$\delta(q, \varepsilon, S) = \{(q, ASA), (q, BSB), (q, CS), (q, SC), (q, \varepsilon)\}$
$\delta(q, a, A) = \{(q, \varepsilon)\}$
$\delta(q, b, B) = \{(q, \varepsilon)\}$
$\delta(q, c, C) = \{(q, \varepsilon)\}$

Cvičení 3.24: Sestrojte zásobníkový automat přijímající jazyk generovaný následující gramatikou.

$$S \longrightarrow \varepsilon \mid SS \mid aSb \mid bSa$$

Cvičení 3.25*: Charakterizujte co nejjednodušeji jazyk L generovaný gramatikou z předchozího příkladu (která by vám měla být povědomá z dřívějška), a pak zkuste pro jazyk $L \cdot \{\#\}$ (obsahující slova z L zakončená speciálním znakem #) navrhnout deterministický zásobníkový automat, tj. takový, v němž ke každému vstupnímu slovu existuje jen jeden výpočet.

Shrnutí: Pojmu zásobníkový automat a souvisejícím pojmům již plně rozumíme a nedělá nám problémy rozbor a návrh zásobníkových automatů rozpoznávajících (jednoduché) bezkontextové jazyky. Také jsme si vědomi vztahu zásobníkových automatů a bezkontextových gramatik; speciálně máme promyšlenu konstrukci automatu ke gramatice.

Kapitola 4

Bezkontextové jazyky – rozšiřující část

Cíle kapitoly:

V této kapitole se máte seznámit s některými speciálními formami bezkontextových gramatik a máte zvládnout některé související algoritmy. Dále se máte seznámit s některými variantami zásobníkových automatů. Speciálně si máte uvědomit, že třída jazyků rozpoznávaných deterministickými zásobníkovými automaty, označovaná DCFL, je vlastní podtřídou třídy bezkontextových jazyků, označované CFL, což se projevuje i v jiných uzávěrových vlastnostech těchto tříd. Dále máte pochopit pumping lemma pro bezkontextové jazyky a vybudovat si porozumění tomu, které rysy činí konkrétní jazyky nebezkontextovými.

4.1 Speciální bezkontextové gramatiky

Orientační čas ke studiu této části: 3 hod.

Cíle této části:

V této části máte zvládnout algoritmus redukce gramatiky (odstranění zbytečných neterminálů) a také převod na tzv. nevypouštějící grama-

tiku (která neobsahuje ε-pravidla). Dále se máte alespoň seznámit s pojmy Chomského normální forma a Greibachové normální forma.

Klíčová slova: *redukované gramatiky, nevypouštějící gramatiky, Chomského normální forma, Greibachové normální forma*

Redukované gramatiky

Vzpomeňme si na odstraňování nedosažitelných stavů u konečných automatů – takové stavy jsou „zbytečné“. Teď se podíváme na odstraňování „zbytečných“ neterminálů u bezkontextových gramatik. Neterminál chápeme jako zbytečný, jestliže se nemůže objevit v žádném odvození terminálního slova z počátečního neterminálu. Neterminál je tedy zbytečný, jestliže z něj nelze odvodit žádné terminální slovo nebo je nedosažitelný (z počátečního neterminálu), což znamená, že se nemůže objevit při jakémkoli přepisování začínajícím z počátečního neterminálu. Redukovaná gramatika je gramatikou bez takových zbytečných neterminálů:

Definice 4.1
Bezkontextová gramatika $G = (\Pi, \Sigma, S, P)$ se nazývá *redukovaná*, jestliže pro každý neterminál $X \in \Pi$ jsou splněny tyto dvě podmínky:

1. existuje alespoň jedno $w \in \Sigma^*$ tž. $X \Rightarrow^* w$,
2. existují slova $\alpha, \beta \in (\Pi \cup \Sigma)^*$ tž. $S \Rightarrow^* \alpha X \beta$.

Uvažujme nejprve, jak pro danou gramatiku $G = (\Pi, \Sigma, S, P)$ zjistit neterminály splňující podmínku 1.
Chceme tedy zkonstruovat množinu $\mathcal{T} = \{X \in \Pi \mid \exists w \in \Sigma^* : X \Rightarrow^* w\}$.
Např. u gramatiky

$$
\begin{aligned}
S &\longrightarrow aSb \,|\, aAbb \,|\, \varepsilon \\
A &\longrightarrow aAB \,|\, bB \\
B &\longrightarrow aAb \,|\, BDB \\
C &\longrightarrow CC \,|\, DcS \\
D &\longrightarrow aba \,|\, cS
\end{aligned}
$$

vidíme, že v prvé řadě patří do $\mathcal{T}$ neterminál S, neboť na pravé straně jednoho S-pravidla (tj. pravidla s S na levé straně) je terminální slovo (ε je pochopitelně terminálním slovem); ze stejného důvodu vidíme, že také $D \in \mathcal{T}$, ale pro jiný neterminál už tuto úvahu neuplatníme. Vidíme ale dále, že na pravé straně jednoho z C-pravidel (tedy z pravidel s C na levé straně) je řetězec složený jen z terminálů a těch neterminálů, o nichž již víme, že patří do $\mathcal{T}$; tedy C také jistě odvodí terminální slovo (byť ne v jednom kroku), a tudíž také patří do $\mathcal{T}$. Pro A ani pro B takovou pravou stranu nenajdeme, ani když už vezmeme v potaz, že $S, C, D \in \mathcal{T}$, takže A ani B do $\mathcal{T}$ nepatří.

Zformulujme pořádně postup, který jsme právě prováděli.

Konstrukce množiny $\mathcal{T} = \{X \in \Pi \mid \exists w \in \Sigma^* : X \Rightarrow^* w\}$:

Konstruujeme postupně množiny $\mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_3 \ldots$, kde

- $\mathcal{T}_1 = \{X \in \Pi \mid \exists w \in \Sigma^* : (X \to w) \in P\}$,
- $\mathcal{T}_{i+1} = \mathcal{T}_i \cup \{X \in \Pi \mid \exists \alpha \in (\Sigma \cup \mathcal{T}_i)^* : (X \to \alpha) \in P\}$

až k případu $\mathcal{T}_n = \mathcal{T}_{n+1}$. Na takový případ nutně narazíme pro $n \leq |\Pi|$ a očividně platí $\mathcal{T}_n = \mathcal{T}$.

Neterminály, splňující podmínku 2., tedy dosažitelné neterminály, neboli prvky množiny $\mathcal{D} = \{X \in \Pi \mid \exists \alpha, \beta : S \Rightarrow_G^* \alpha X \beta\}$ lze také zjistit přímočarým postupem; ilustrujme ho opět na gramatice

$$\begin{aligned} S &\longrightarrow aSb \,|\, aAbb \,|\, \varepsilon \\ A &\longrightarrow aAB \,|\, bB \\ B &\longrightarrow aAb \,|\, BDB \\ C &\longrightarrow CC \,|\, DcS \\ D &\longrightarrow aba \,|\, cS \end{aligned}$$

Počáteční neterminál S je samozřejmě dosažitelný, tedy $S \in \mathcal{D}$.

Kontrolní otázka: Tedy existují α, β takové, že $S \Rightarrow_G^* \alpha S \beta$. Umíte rychle říci příklad takových řetězců α, β ?

(Samozřejmě u počátečního neterminálu vždy vyhovují $\alpha = \beta = \varepsilon$. V našem konkrétním případě vyhovují také třeba $\alpha = aa$, $\beta = bb$, ale to není podstatné.)

Navíc je jistě jasné, že pro každý dosažitelný neterminál X platí, že všechny neterminály na pravé straně X-pravidel jsou také dosažitelné.

Kontrolní otázka: Umíte toto zdůvodnit podle definice množiny $\mathcal{D}$?

(Když $S \Rightarrow^* \alpha X \beta$ a Y je na pravé straně nějakého X-pravidla, tedy máme pravidlo $X \longrightarrow \gamma Y \delta$, tak $S \Rightarrow^* \alpha\gamma Y \delta\beta$}.)

V našem případě tedy i A patří do $\mathcal{D}$. Pak ovšem do $\mathcal{D}$ patří také B, a pak také D. Tím se naše konstrukce uzavře, neterminál C „zůstal mimo“. Zformulujme konstrukci zase přesně.

Konstrukce množiny $\mathcal{D} = \{X \in \Pi \mid \exists \alpha, \beta : S \Rightarrow_G^* \alpha X \beta\}$ *:*

Konstruujeme postupně množiny $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3 \ldots$, kde

- $\mathcal{D}_1 = \{S\}$,
- $\mathcal{D}_{i+1} = \mathcal{D}_i \cup \{X \in \Pi \mid$ ex. $Y \in \mathcal{D}_i$ a α obsahující X tž. $(Y \to \alpha) \in P\}$

až k případu $\mathcal{D}_n = \mathcal{D}_{n+1}$.

Kontrolní otázka: Existuje redukovaná gramatika $G = (\Pi, \Sigma, S, P)$, pro niž platí $L(G) = \emptyset$?

(Jistě jste si uvědomili, že v takovém případě by počáteční neterminál S nesplňoval podmínku 1. z definice 4.1; takže redukovaná gramatika generující prázdný jazyk neexistuje.)

Nyní využitím výše uvedených konstrukcí snadno vyvodíme následující větu.

Věta 4.2
Existuje algoritmus, který k zadané bezkontextové gramatice G, pro niž platí $L(G) \neq \emptyset$, sestrojí ekvivalentní redukovanou gramatiku.

Důkaz: Mějme $G = (\Pi, \Sigma, S, P)$. Nejdříve zkonstruujeme množinu neterminálů splňujících podmínku 1. (z definice redukované gramatiky).

Pak v G vynecháme všechny neterminály nesplňující 1. a všechna pravidla, která takové neterminály obsahují. (I pro X splňující 1. se tak může stát, že některá X-pravidla budou odstraněna.) Dostaneme tak jistou gramatiku G' a je očividné, že $L(G) = L(G')$.

Pro gramatiku G' nyní zkonstruujeme množinu neterminálů splňujících podmínku 2. a dále vynecháme všechny neterminály nesplňující 2. a všechna pravidla, která takové neterminály obsahují. (Z definice $\mathcal{D}$ je zřejmé, že pro každé X budou teď buď zachována všechna X-pravidla, nebo budou všechna odstraněna.) Dostaneme tak jistou gramatiku G'' a je opět očividné, že $L(G) = L(G') = L(G'')$.

Důležité je, že G'' je redukovanou gramatikou, tedy poslední úpravou nemohlo dojít k tomu, že by některé neterminály v G'' nesplňovaly podmínku 1. V jakékoli derivaci začínající dosažitelným neterminálem se totiž mohou vyskytovat jen dosažitelné neterminály. □

Aplikujme teď *algoritmus redukce gramatiky* obsažený v předchozím důkazu na gramatiku

$$\begin{aligned} S &\longrightarrow aSb \,|\, aAbb \,|\, \varepsilon \\ A &\longrightarrow aAB \,|\, bB \\ B &\longrightarrow aAb \,|\, BDB \\ C &\longrightarrow CC \,|\, DcS \\ D &\longrightarrow aba \,|\, cS \end{aligned}$$

Jelikož $\mathcal{T} = \{S, C, D\}$, bude gramatika po první úpravě vypadat následovně.

$$\begin{aligned} S &\longrightarrow aSb \,|\, \varepsilon \\ C &\longrightarrow CC \,|\, DcS \\ D &\longrightarrow aba \,|\, cS \end{aligned}$$

Po druhé úpravě pak dostaneme (již redukovanou) gramatiku

$$S \longrightarrow aSb \,|\, \varepsilon$$

Kontrolní otázka: Co myslíte, vede prohození úprav v důkazu Věty 4.2 také vždy k redukované gramatice?

Pokud jste se hlouběji zamysleli, asi jste přišli na to, že tomu tak nemusí být. Ilustruje to už naše výše uvedená gramatika. Kdybychom v první fázi odstranili pravidla s nedosažitelnými neterminály, dostali bychom gramatiku

$$\begin{aligned} S &\longrightarrow aSb \,|\, aAbb \,|\, \varepsilon \\ A &\longrightarrow aAB \,|\, bB \end{aligned}$$

$$B \longrightarrow aAb \,|\, BDB$$
$$D \longrightarrow aba \,|\, cS$$

Když bychom teď odstranili pravidla obsahující neterminály, z kterých nelze odvodit žádné terminální slovo, dostali bychom

$$S \longrightarrow aSb \,|\, \varepsilon$$
$$D \longrightarrow aba \,|\, cS$$

tedy gramatiku, která není redukovaná!
(Neterminál D byl dosažitelný jen díky neterminálům neodvozujícím terminální slovo a po jejich odstranění byl tedy „odříznut“, stal se nedosažitelným.)

Poznámka: Všimněme si, že metoda redukce gramatiky nijak nezajišťuje, že výsledná gramatika je nejmenší možná pro daný jazyk, či něco takového. Je to zde jinak než u konečných automatů; neexistuje totiž algoritmus, který by k dané bezkontextové gramatice zkonstruoval nejmenší s ní ekvivalentní. Dá se to ukázat metodami teorie vyčíslitelnosti, z nichž rovněž plyne, že neexistuje algoritmus, který by rozhodoval, zda dvě zadané gramatiky jsou ekvivalentní. (Vrátíme se k tomu při pojednání o rozhodnutelnosti a nerozhodnutelnosti problémů.)
Ale platí alespoň následující věta.

Věta 4.3
Existuje algoritmus, který pro libovolnou bezkontextovou gramatiku G rozhodne, zda $L(G)$ je či není prázdný.

Důkaz: Stačí ověřit, zda S patří do množiny neterminálů splňujících podmínku 1. (tedy do množiny $\mathcal{T}$). □

Nevypouštějící gramatiky

Z technických důvodů mohou být někdy nepříjemná tzv. ε-pravidla, tj. pravidla typu $X \longrightarrow \varepsilon$. Pak se hodí postup, který umožňuje se jich zbavit, aniž změníme generovaný jazyk. Jde to modifikací konstrukce množiny $\mathcal{T}$ diskutované u redukce gramatik a následnou specifickou úpravou pravidel; je tu ovšem jeden drobný technický problémek: bez ε-pravidel nevygenerujeme prázdné slovo.

Definice 4.4
Bezkontextová gramatika se nazývá *nevypouštějící*, jestliže neobsahuje žádné ε-pravidlo, tedy pravidlo typu $X \longrightarrow \varepsilon$.

Věta 4.5
Existuje algoritmus, který k zadané bezkontextové gramatice G sestrojí nevypouštějící gramatiku G' tak, že $L(G') = L(G) - \{\varepsilon\}$.

Důkaz: Konstrukce využívá obdoby výše uvedené konstrukce pro neterminály splňující podmínku 1. z definice redukované gramatiky.

Ke gramatice $G = (\Pi, \Sigma, S, P)$ totiž nejprve sestrojíme množinu $\mathcal{E} = \{X \in \Pi \mid X \Rightarrow^* \varepsilon\}$, tedy množinu těch neterminálů, z nichž lze odvodit prázdné slovo (jedním či více kroky).

Konstruujeme postupně množiny $\mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \ldots$, kde

- $\mathcal{E}_1 = \{X \in \Pi \mid (X \to \varepsilon) \in P\}$,
- $\mathcal{E}_{i+1} = \mathcal{E}_i \cup \{X \in \Pi \mid \exists \alpha \in \mathcal{E}_i^* : (X \to \alpha) \in P\}$

a skončíme při dosažení „pevného bodu“, tedy v případě $\mathcal{E}_n = \mathcal{E}_{n+1}$ – je zřejmé, že pak $\mathcal{E}_n = \mathcal{E}$.

Na základě $\mathcal{E}$ teď sestrojíme množinu pravidel P' takto: pro každé pravidlo $(X \to \alpha) \in P$ zařadíme do P' všechna možná pravidla $X \to \beta$, kde β vznikne z α vypuštěním některých (třeba žádných) výskytů symbolů z $\mathcal{E}$; přitom ovšem vynecháme (nezařazujeme) případnou možnost $X \to \varepsilon$.

Položíme $G' = (\Pi, \Sigma, S, P')$; lze snadno ověřit, že skutečně $L(G') = L(G) - \{\varepsilon\}$ (formálně lze postupovat např. indukcí podle délky odvození). □

Aplikujme uvedený postup např. na gramatiku

$$\begin{aligned} S &\longrightarrow AB \mid \varepsilon \\ A &\longrightarrow aAAb \mid BS \mid CA \\ B &\longrightarrow BbA \mid CaC \mid \varepsilon \\ C &\longrightarrow aBB \mid bS \end{aligned}$$

Snadno zjistíme, že $\mathcal{E}_1 = \{S, B\}$, $\mathcal{E}_2 = \{S, B, A\}$, $\mathcal{E}_3 = \mathcal{E}_2 = \mathcal{E}$.
Pravidlo $S \longrightarrow AB$ teď podle návodu nahradíme pravidly $S \longrightarrow AB$, $S \longrightarrow B$ a $S \longrightarrow A$ (oba neterminály A, B najednou vynechat nemůžeme, protože

bychom dostali ε). Pravidlo $S \longrightarrow \varepsilon$ pochopitelně odstraníme. K pravidlu $A \longrightarrow aAAb$ přibude $A \longrightarrow aAb$ a $A \longrightarrow ab$, atd. Celkovým výsledkem bude následující gramatika.

$$
\begin{aligned}
&S \longrightarrow AB \mid B \mid A \\
&A \longrightarrow aAAb \mid aAb \mid ab \mid BS \mid S \mid B \mid CA \mid C \\
&B \longrightarrow BbA \mid bA \mid Bb \mid b \mid CaC \\
&C \longrightarrow aBB \mid aB \mid a \mid bS \mid b
\end{aligned}
$$

Jak jsme již zmínili, bez ε-pravidel prázdné slovo nevygenerujeme. Pokud chceme mít gramatiku (v zásadě) nevypouštějící, ale chceme zachovat generování prázdného slova, můžeme použít konstrukci z následující věty.

Věta 4.6
Ke každé bezkontextové gramatice $G = (\Pi, \Sigma, S, P)$ existuje ekvivalentní bezkontextová gramatika $G_1 = (\Pi_1, \Sigma, S_1, P_1)$, kde ε může být pravou stranou pouze u pravidla $S_1 \to \varepsilon$ a v takovém případě se pak S_1 nevyskytuje na pravé straně žádného z pravidel v P_1.

Důkaz: Ke G lze sestrojit nevypouštějící gramatiku $G' = (\Pi, \Sigma, S, P')$ podle Věty 4.5. Platí-li $\varepsilon \notin L(G)$ (tj. S nepatří do zkonstruované množiny $\mathcal{E}$), položíme $G_1 = G'$. Je-li $\varepsilon \in L(G)$, vznikne G_1 z G' přidáním nového neterminálu S_1, který bude počátečním, a dodáním pravidel $S_1 \to \varepsilon$, $S_1 \to S$. □

Chomského normální forma a Greibachové normální forma

Z technických důvodů je užitečné, že bezkontextové gramatiky lze transformovat do různých *normálních forem*, u nichž jsou kladena další syntaktická omezení na povolená pravidla.

Zde alespoň nadefinujeme dvě užívané normální formy a uvedeme bez důkazu tvrzení o příslušných převodech obecných gramatik do oněch normálních forem.

Definice 4.7
Bezkontextová gramatika je v *Chomského normální formě*, zkráceně v CNF, jestliže každé její pravidlo je tvaru $X \to YZ$ nebo $X \to a$, kde X, Y, Z jsou neterminální symboly a a terminální symbol.

Věta 4.8
Ke každé bezkontextové gramatice G lze sestrojit gramatiku G' v CNF tž. $L(G') = L(G) - \{\varepsilon\}$.

Definice 4.9
Bezkontextová gramatika je v *Greibachové normální formě*, zkráceně v GNF, jestliže každé její pravidlo je v tvaru $X \to aY_1Y_2 \dots Y_n$ ($n \geq 0$, a je terminál, Y_i jsou neterminály).

Věta 4.10
Ke každé bezkontextové gramatice G lze sestrojit gramatiku G' v GNF tž. $L(G') = L(G) - \{\varepsilon\}$.

Cvičení 4.1: Zredukujte následující bezkontextovou gramatiku

$$\begin{array}{lcl} S & \longrightarrow & aSb \mid aAbb \mid aDaS \mid \varepsilon \\ A & \longrightarrow & aAB \mid bB \\ B & \longrightarrow & aAb \mid BB \\ C & \longrightarrow & CC \mid cS \\ D & \longrightarrow & aSb \mid cD \mid aEE \\ E & \longrightarrow & EB \mid bD \end{array}$$

Cvičení 4.2: Zredukujte následující bezkontextovou gramatiku:

$$\begin{array}{l} S \longrightarrow aA \mid bB \mid aSa \mid bSb \mid \varepsilon \\ A \longrightarrow bCD \mid Dba \\ B \longrightarrow Bb \mid AC \\ C \longrightarrow aA \mid c \\ D \longrightarrow DE \\ E \longrightarrow \varepsilon \end{array}$$

Cvičení 4.3: Zjistěte, zda pro následující gramatiku G je $L(G) \neq \emptyset$

$$\begin{array}{l} S \longrightarrow aS \mid AB \mid CD \\ A \longrightarrow aDb \mid AD \mid BC \end{array}$$

$B \longrightarrow bSb \,|\, BB$
$C \longrightarrow BA \,|\, ASb$
$D \longrightarrow ABCD \,|\, \varepsilon$

Cvičení 4.4: Následující gramatiku převeďte na ekvivalentní nevypouštějící gramatiku, v níž připouštíme, že počáteční neterminál se může přepsat na ε, ale v tom případě se nesmí vyskytovat na pravé straně žádného pravidla.

$$\begin{array}{lcl} S & \longrightarrow & A \,|\, 0SA \,|\, \varepsilon \\ A & \longrightarrow & 1A \,|\, 1 \,|\, B1 \\ B & \longrightarrow & 0B \,|\, 0 \,|\, \varepsilon \end{array}$$

Shrnutí: Takže umíme precizně definovat, co jsou zbytečné neterminály u bezkontextových gramatik, a máme zvládnutý algoritmus, který je odstraní. Umíme se rovněž zbavit tzv. ε-pravidel a převést tak gramatiku na tzv. nevypouštějící. Zaznamenali jsme také, že bezkontextové gramatiky se dají převést do tzv. Chomského normální formy a Greibachové normální formy.

4.2 Varianty zásobníkových automatů, deterministické bezkontextové jazyky, uzávěrové vlastnosti tříd CFL a DCFL

Orientační čas ke studiu této části: 2 hod.

Cíle této části:

V této části máte zvládnout definice dalších variant zásobníkových automatů, speciálně deterministických. Dále máte pochopit definice tříd CFL a DCFL a jejich uzávěrové vlastnosti.

Klíčová slova: *zásobníkový automat s přijímajícími stavy, deterministický zásobníkový automat, třída bezkontextových jazyků (CFL), třída deterministických bezkontextových jazyků (DCFL), uzávěrové vlastnosti tříd CFL a DCFL*

Zásobníkové automaty s přijímajícími stavy

V základních definicích 3.8 a 3.9 jsme uvedli, že slovo $w \in \Sigma^*$ je chápáno jako přijímané zásobníkovým automatem $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$, tedy $w \in L(M)$, právě tehdy, když existuje výpočet začínající v počáteční konfiguraci (q_0, w, Z_0), při němž se přečte celé vstupní vstupní slovo w a skončí se s prázdným zásobníkem (výpočet tedy dospěje do konfigurace $(q, \varepsilon, \varepsilon)$ pro nějaký stav $q \in Q$).

Jak jsme také již uvedli, jiná běžná varianta definice zásobníkových automatů počítá s přijímajícími stavy, podobně jako to známe z oblasti (nedeterministických) konečných automatů. Zde uvedeme příslušné definice a alespoň zaznamenáme fakt, že přijímání prázdným zásobníkem je možné simulovat přijímáním přijímajícími (též říkáme koncovými) stavy, a naopak.

Definice 4.11
Zásobníkový automat (*s přijímajícími stavy*) lze definovat jako sedmici $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, která se liší od šestice v definici 3.8 jen přidáním $F \subseteq Q$, což je množina *koncových (přijímajících)* stavů.

Pro automat M definujeme *jazyk rozpoznávaný koncovým stavem*

$$L_{KS}(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_M^* (q, \varepsilon, \alpha) \text{ pro nějaké } q \in F \text{ a } \alpha \in \Gamma^*\}$$

a *jazyk rozpoznávaný prázdným zásobníkem*

$$L_{PZ}(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_M^* (q, \varepsilon, \varepsilon) \text{ pro nějaké } q \in Q\}.$$

Kontrolní otázka: Pochopili jste smysl zápisů jako $(q_0, w, Z_0) \vdash_M^* (q, \varepsilon, \alpha)$ apod.?

Jistě jste si uvědomili, že se jedná o reflexivní a tranzitivní uzávěr relace $\vdash_M$ na množině konfigurací. Zápis $K_1 \vdash_M^* K_2$ tedy znamená, že automat M se může konečným počtem kroků (tj. postupným provedením aplikovatelných instrukcí) dostat z konfigurace K_1 do K_2.
Tedy vidíme, že slovo $w \in \Sigma^*$ patří do jazyka $L_{KS}(M)$ právě tehdy, když existuje výpočet automatu M, který začíná v počáteční konfiguraci (q_0, w, Z_0) a při němž se přečte celé vstupní vstupní slovo w a dosáhne se přitom přijímajícího stavu (přičemž na (ne)prázdnosti zásobníku nezáleží).

Tvrzení 4.12
K libovolnému ZA M_1 lze zkonstruovat ZA M_2 tž. $L_{KS}(M_1) = L_{PZ}(M_2)$ a také ZA M_2' tž. $L_{PZ}(M_1) = L_{KS}(M_2')$.

Deterministické zásobníkové automaty

Připomeňme si, že u zásobníkových automatů jsme jako základní vzali *nedeterministickou* verzi. Takto totiž ZA odpovídají bezkontextovým gramatikám, tj. rozpoznávají právě bezkontextové jazyky.

Již jsme také zmínili, že praktické algoritmy postavené na zásobníkových automatech (např. pro syntaktickou analýzu programovacích jazyků) využívají deterministickou verzi. Neformálně jsme již hovořili o tom, že zásobníkový automat je deterministický, jestliže má pro každé vstupní slovo jediný možný výpočet; to je zaručeno, když nemá dvě různé instrukce se stejnou levou stranou $(q, a, X) \to \ldots$ (pro $a \in \Sigma \cup \{\varepsilon\}$) či s levými stranami $(q, a, X) \to \ldots$ a $(q, \varepsilon, X) \to \ldots$ (pro $a \in \Sigma$). Tato podmínka je stručně a přesně zachycena následující definicí.

Definice 4.13
Deterministický zásobníkový automat (DZA) je zásobníkovým automatem $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, pro nějž platí následující dvě podmínky:

1. $\delta(q, a, X)$ je nejvýše jednoprvková množina
   pro každé $q \in Q, a \in \Sigma \cup \{\varepsilon\}, X \in \Gamma$;

2. je-li $\delta(q, \varepsilon, X) \neq \emptyset$, pak $\delta(q, a, X) = \emptyset$
   pro každé $q \in Q, a \in \Sigma, X \in \Gamma$.

Jazyk L je *deterministický bezkontextový jazyk*, jestliže $L = L_{KS}(M)$ pro nějaký DZA M.

Kontrolní otázka: Je vám jasné, že u deterministického zásobníkového automatu existuje pro každé vstupní slovo jediný výpočet DZA M ? Je tento výpočet nutně konečný?

(Jistě jste si uvědomili, že vždy je aplikovatelná nejvýš jedna instrukce, ale není vyloučeno, že automat bude provádět nekonečně mnoho ε-kroků.)

Poznámka: Takový nekonečný výpočet je samozřejmě obvykle nežádoucí, ale definice jej nevylučuje. Podobně vám váš oblíbený programovací jazyk nezabraňuje naprogramovat např. nežádoucí nekonečný cyklus.

Ještě si všimněme, že v případě přijímání prázdným zásobníkem by byl jazyk rozpoznávaný deterministickým zásobníkovým automatem nutně *bezprefixový* – nebylo by možné, aby bylo přijato slovo w a také nějaký jeho vlastní prefix u. (Např. jazyk $\{a, aa\}$ by tímto způsobem nemohl být přijímán.) Proto jsou deterministické bezkontextové jazyky, tvořící třídu DCFL, definovány pomocí přijímání koncovým stavem.

Již jsme zmínili, že třída jazyků rozpoznatelných deterministickými zásobníkovými automaty, označovaná DCFL (deterministic context-free languages), je vlastní podtřídou třídy bezkontextových jazyků, označované CFL (context-free languages). Bez důkazu zde alespoň uvedeme následující příklady jazyků, které jsou v CFL, ale nejsou v DCFL:

- $\{a^i b^j c^k \mid (i = j) \vee (j = k)\}$
- $\{a^i b^j c^k \mid (i \neq j) \vee (j \neq k)\}$
- $\{ww^R \mid w \in \{a, b\}^*\}$

Poznámka: Vzpomeňme si, že existuje (rychlý) algoritmus, který o dvou zadaných konečných automatech rozhodne, zda jsou ekvivalentní (tj. zda přijímají tentýž jazyk). Později si ukážeme, že podobný algoritmus pro (nedeterministické) zásobníkové automaty neexistuje. Od 60. let dvacátého století ale byla otevřena otázka, zda existuje algoritmus rozhodující ekvivalenci deterministických zásobníkových automatů. Pozitivní řešení prezentoval v r. 1997 G. Sénizergues, později důkaz zjednodušil C. Stirling. (Uvedení důkazu v našem kursu však pro jeho náročnost nepřipadá v úvahu.)

Uzávěrové vlastnosti třídy CFL a třídy DCFL

V tomto oddíle uvedeme jen *velmi stručný přehled* některých uzávěrových vlastností třídy bezkontextových jazyků, označené CFL (contex-free languages), a třídy deterministických bezkontextových jazyků, označené DCFL (deterministic contex-free languages).

CFL není uzavřena na všechny operace, na které je uzavřena třída regulárních jazyků. Nejdříve si ukážeme případy operací, vůči nimž CFL uzavřena je. Důkazy jsou samozřejmě opět konstruktivní – ukazují algoritmy, které k zadané reprezentaci jazyků (operandů) zkonstruují reprezentaci jazyka, jenž je výsledkem operace.

Poznámka: Jako příslušnou reprezentaci bezkontextových jazyků lze samozřejmě volit bezkontextové gramatiky či zásobníkové automaty. V následující větě jsou výhodné gramatiky.

Věta 4.14
CFL je uzavřena vůči sjednocení, zřetězení, iteraci, zrcadlovému obrazu.

Důkaz: K libovolným bezkontextovým gramatikám $G_1 = (\Pi_1, \Sigma, S_1, P_1)$, $G_2 = (\Pi_2, \Sigma, S_2, P_2)$ lze zkonstruovat gramatiku $G = (\Pi, \Sigma, S, P)$ tž. $L(G) = L(G_1) \cup L(G_2)$ takto: Předpokládáme, že $\Pi_1 \cap \Pi_2 = \emptyset$ (docílíme toho případným přejmenováním neterminálů). Položíme $\Pi = \Pi_1 \cup \Pi_2 \cup \{S\}$, kde $S \notin \Pi_1 \cup \Pi_2$, a $P = P_1 \cup P_2 \cup \{S \longrightarrow S_1, S \longrightarrow S_2\}$.

Rovněž velmi přímočará je konstrukce gramatik generujících jazyky $L(G_1) \cdot L(G_2)$, $L(G_1)^*$, $L(G_1)^R$.

□

Kontrolní otázka: Umíte podrobně dořešit případ $L(G_1) \cdot L(G_2)$, $L(G_1)^*$, $L(G_1)^R$?

(Nápověda: U zřetězení je klíčovým pravidlem $S \longrightarrow S_1 S_2$, u iterace $S \longrightarrow \varepsilon \mid S_1 S$, a v případě zrcadlového obrazu nahradíme pravé strany *všech* pravidel jejich zrcadlovými obrazy.)

Neuzavřenost CFL vůči některým operacím se nejpříměji dokáže konstrukcí (jednoduchých) protipříkladů; je samozřejmě možné užít i další úvahy. Před uvedením příslušné věty se nejprve na chvíli zamyslete nad následující otázkou.

Kontrolní otázka: Co vám říká vaše intuice o jazyku $\{a^n b^n c^n \mid n \geq 0\}$? Je bezkontextový ?

Při promýšlení toho, jak by mohl vypadat např. zásobníkový automat rozpoznávající uvedený jazyk, jste jistě dospěli k závěru, že to asi nejde. Takový automat umí porovnat např. počet a-ček a b-ček tak, že a-čka při čtení dává do zásobníku a při čtení b-ček je odebírá, ale je pak „bezradný“, když má

s tímto („zapomenutým“) počtem porovnat ještě počet c-ček. Tento jazyk opravdu není bezkontextový, ale pořádně to ukážeme až později. Teď jen jeho nebezkontextovosti využijeme v následující větě.

Věta 4.15
CFL není uzavřena vůči průniku a doplňku.

Důkaz: Jazyky $L_1 = \{a^i b^j c^k \mid i = j\}$, $L_2 = \{a^i b^j c^k \mid j = k\}$ jsou zřejmě bezkontextové. Přitom $L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$ bezkontextový není.

Z de Morganových pravidel plyne, že kdyby byla CFL uzavřena vůči doplňku, tak by díky uzavřenosti vůči sjednocení byla uzavřena i vůči průniku (to ale není, takže není uzavřena ani vůči doplňku). □

Již jsme diskutovali, že na rozdíl od konečných automatů je deterministická verze zásobníkových automatů skutečně slabší než nedeterministická, tedy DCFL je *vlastní* podtřídou CFL. Plyne to i z toho, že DCFL má jiné uzávěrové vlastnosti než CFL. Objasňuje to následující věta, kterou zde uvedeme bez důkazu.

Věta 4.16
Třída DCFL je uzavřena vůči doplňku. Není ale uzavřena vůči průniku ani vůči sjednocení.

Kontrolní otázka: Jak se tedy liší DCFL od CFL v uzavřenosti vzhledem k základním booleovským operacím?

(Jistě jste si připomněli, že CFL je uzavřena vůči sjednocení, není ale uzavřena vůči průniku ani vůči doplňku.)

Shrnutí: Máme tedy jasno, že varianty (nedeterministických) zásobníkových automatů přijímajících prázdným zásobníkem či koncovými (neboli přijímajícími) stavy jsou ekvivalentní; obě rozpoznávají právě bezkontextové jazyky, tedy prvky třídy CFL. Deterministické bezkontextové jazyky, tvořící třídu DCFL, jsou definovány pomocí deterministických zásobníkových automatů přijímajících koncovými stavy. Umíme snadno ukázat uzavřenost třídy CFL vůči sjednocení, zřetězení, iteraci a zrcadlovému obrazu. Víme, že CFL není uzavřena vůči průniku ani doplňku. Rovněž alespoň víme, že DCFL je uzavřena vůči doplňku, ale ne vůči průniku ani vůči sjednocení.

4.3 Nebezkontextové jazyky

Orientační čas ke studiu této části: 3 hod.

Cíle této části:

V této části máte pochopit tzv. pumping lemma pro bezkontextové jazyky a vybudovat si porozumění dostatečné k tomu, abyste poznali, které jazyky nejsou bezkontextové.

Klíčová slova: *pumping lemma pro bezkontextové jazyky, důkazy nebezkontextovosti konkrétních jazyků*

Připomeňme si, že jsme ukázali, že jazyk $\{a^n b^n \mid n \geq 0\}$ není regulární (např. kvocienty jazyka podle slov $a, a^2, a^3, \ldots$ jsou různé). Je nám také jasné, že uvedený jazyk je bezkontextový; přijímá jej jednoduchý zásobníkový automat.

Již jsme ale intuitivně dospěli k názoru, že např. jazyk

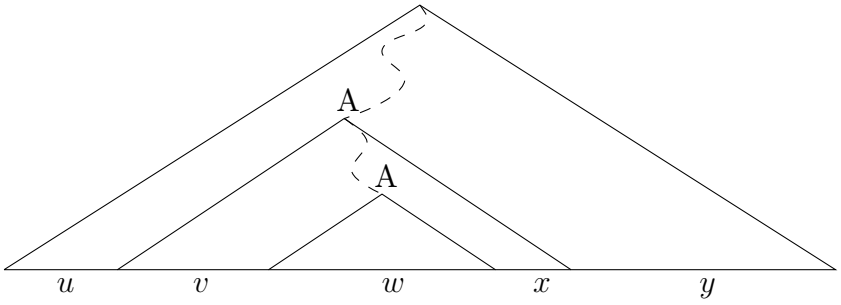
$$L = \{a^i b^i c^i \mid i \geq 0\}$$

bezkontextový není (tedy že pro něj není možné sestrojit ani bezkontextovou gramatiku ani zásobníkový automat). Nyní tuto úvahu rozvineme a zformalizujeme.

Jak tedy můžeme jasně dokázat, že jazyk $L = \{a^i b^i c^i \mid i \geq 0\}$ není bezkontextový? Úvahy o kvocientech tady příliš nepomohou, bezkontextový jazyk samozřejmě může mít nekonečně mnoho kvocientů podle jednotlivých slov, jak jsme ukázali u jazyka $\{a^n b^n \mid n \geq 0\}$.
Zkusme předpokládat, že L je bezkontextový; vyvodíme-li z tohoto předpokladu logický spor, dokážeme tak, že L bezkontextový není. Když tedy předpokládáme, že L je bezkontextový, musí existovat bezkontextová gramatika G, která ho generuje, tedy $L(G) = L$.

Poznámka: Uvažovat gramatiku generující L se ukáže pro naše účely vhodnější než uvažovat zásobníkový automat rozpoznávající L.

Pro každé n tedy existuje derivační strom (podle gramatiky G) pro slovo $a^n b^n c^n$. Vezmeme-li slovo „dostatečně dlouhé“ (tj. n „dostatečně velké“),



Obrázek 4.1: Schéma derivačního stromu pro $uvwxy$.

v příslušném derivačním stromu nutně dochází k opakování nějakého neterminálu na nějaké větvi (tj. cestě od kořene k listu) – viz Obrázek 4.1. Přesněji řečeno: derivačních stromů, ve kterých se takové opakování nevyskytuje, je konečně mnoho. Výraz „n je dostatečně velké“ lze zpřesnit tak, že $3n$ (tj. délka slova $a^n b^n c^n$) je větší než délka nejdelšího slova odvoditelného derivačním stromem bez takového opakování.

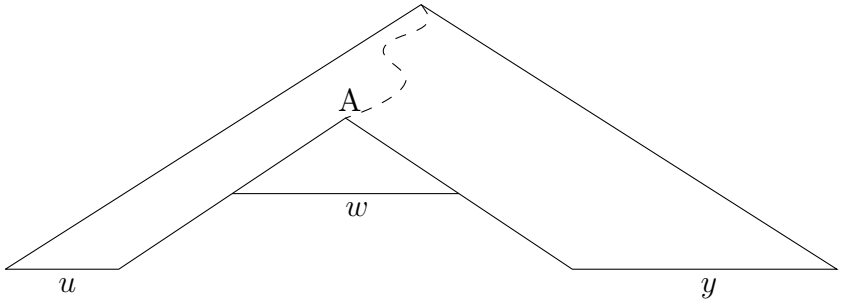
Kontrolní otázka: Proč je jen konečně mnoho derivačních stromů, ve kterých se neopakuje nějaký neterminál na nějaké větvi?

Vezměme nyní tedy ono velmi dlouhé slovo $a^n b^n c^n$ a pro něj *nejmenší* možný derivační strom (i ten má opakování jako na Obrázku 4.1). Slovo $a^n b^n c^n$ se dá psát ve tvaru $uvwxy$ (jak znázorněno na obrázku), kde navíc alespoň jedno ze slov v, x je neprázdné (jinak bychom mohli oba uzly označené neterminálem A ztotožnit a získali bychom pro $a^n b^n c^n$ menší derivační strom). Je jasné, že derivační stromy existují i pro uwy (viz Obrázek 4.2), a také uv^2wx^2y (Obrázek 4.3), uv^3wx^3y, ... Tato slova tudíž také patří do L.

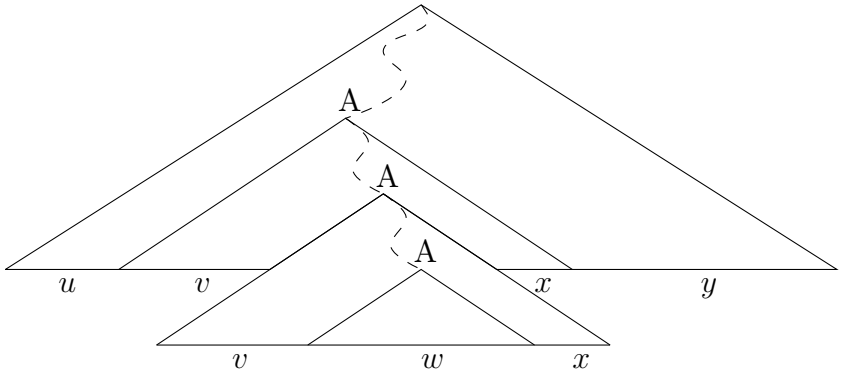
Snadno se ale můžeme přesvědčit, že ať rozdělíme slovo $a^n b^n c^n$ na 5 částí $uvwxy$ *jakkoliv*, přičemž alespoň jedno ze slov v, x je neprázdné, pak slovo $uvvwxxy$ zaručeně nepatří do L. Dosáhli jsme tedy kýženého logického sporu.

Kontrolní otázka: Proč slovo $uvvwxxy$ zaručeně nepatří do L ?

(Když v obsahuje dva různé symboly (např. a i b), tak $uvvwxxy$ není ani ve tvaru $a^{k_1} b^{k_2} c^{k_3}$; podobně je to, když x obsahuje dva různé symboly. Když ale slova v, x neobsahují alespoň jeden ze symbolů a, b, c (a samozřejmě alespoň



Obrázek 4.2: Schéma derivačního stromu pro uwy



Obrázek 4.3: Schéma derivačního stromu pro uv^2wx^2y

jeden obsahují), tak ve slově $uvvwxxy$ nemůže být stejný počet symbolů a, b i c.)

Odvodili jsme tak určité „pumping lemma“ (pumpovací lemma) platné obecně pro bezkontextové jazyky (nikoli jen pro náš L) a demonstrovali jsme jeho použití pro důkaz, že L není bezkontextový. Zmíněné lemma následuje.

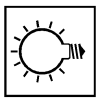
Věta 4.17
(Pumping lemma pro bezkontextové jazyky, neboli $uvwxy$-teorém.)
Nechť L je bezkontextový jazyk. Pak existuje přirozené číslo n tž. každé slovo $z \in L$ s délkou $|z| \geq n$ lze psát ve tvaru $z = uvwxy$, kde platí

- $|vx| \geq 1$, *tj.* $v \neq \varepsilon$ *nebo* $x \neq \varepsilon$,
- $|vwx| \leq n$,
- *pro všechna* $i \geq 0$ *platí, že* $uv^iwx^iy \in L$.

Uvedené číslo n k zadanému bezkontextovému jazyku L konkretizuje naše intuitivní vyjádření o „dostatečně dlouhých slovech z L“; takové n se dá konkrétně vypočítat z bezkontextové gramatiky G generující jazyk L, ale to teď není pro nás podstatné. Postačuje nám myšlenka, že n je tak velké, že zaručuje, že v každém derivačním stromu podle G pro slovo dlouhé alespoň n nutně dochází k zopakování neterminálu na některé větvi. Někdy se hodí, že číslo n zároveň může sloužit k omezení velikosti úseku obsahujícího „pumpovací úseky“ (to vyjadřuje podmínka $|vwx| \leq n$). To plyne z toho, že v derivačním stromu s opakováním nějakého neterminálu na nějaké větvi nutně existuje podstrom, v němž je kořen ohodnocen stejně jako nějaký vnitřní vrchol (jak je to znázorněno na obrázku 4.1, kde se jedná o ohodnocení neterminálem A), ale v němž už k žádnému jinému zopakování neterminálu na jedné větvi nedochází; takový podstrom má tedy omezenou velikost.

Kontrolní otázka: Umíte zdůvodnit, proč takový podstrom musí nutně existovat?

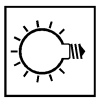
Pumping lemma, a hlavně úvahy, které jsme provedli, nám mají především posloužit k vybudování porozumění tomu, které jazyky bezkontextové nejsou. Provedeme to formou několika řešených příkladů; začneme případem, který už známe.



Řešený příklad 4.1: Ukažte, že jazyk $L_1 = \{a^ib^ic^i \mid i \geq 0\}$ není bezkontextový.

Řešení: Kdyby byl, existovalo by příslušné n podle pumping lemmatu. Pak např. slovo $z = a^n b^n c^n$ (pro nějž platí $z \in L_1$ a $|z| \geq n$) by se dalo psát ve tvaru $z = uvwxy$, kde mj. platí $v \neq \varepsilon$ nebo $x \neq \varepsilon$, $|vwx| \leq n$ a $uwy \in L_1$. Jelikož $|vwx| \leq n$, slova v, x neobsahují alespoň jeden ze symbolů a, b, c (a samozřejmě alespoň jeden obsahují). Proto ve slově uwy nemůže být stejný počet symbolů a, b i c a slovo tedy nepatří do L_1, což je spor (s předpokladem, že L_1 je bezkontextový).

Poznámka: Z úvodní analýzy (před Větou 4.17) víme, že sporu dosáhneme i při nevyužití podmínky $|vwx| \leq n$. Protože jsme tam tuto podmínku nevyužili, museli jsme se pustit do mírně složitějších úvah a vyvodit spor pro slovo $uvvwxxy$.
Ovšem např. v následujícím příkladu je už využití podmínky $|vwx| \leq n$ k vyvození sporu podstatné.



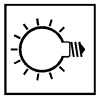
ŘEŠENÝ PŘÍKLAD 4.2: Ukažte, že jazyk $L_2 = \{ww \mid w \in \{0,1\}^*\}$ není bezkontextový.

Řešení: Kdyby byl, existovalo by příslušné n podle pumping lemmatu. Tedy např. slovo $z = 0^n 1^n 0^n 1^n$ (pro nějž platí $z \in L_2$ a $|z| \geq n$) by se dalo psát ve tvaru $z = uvwxy$, kde mj. platí $v \neq \varepsilon$ nebo $x \neq \varepsilon$, $|vwx| \leq n$ a $uwy \in L_2$. Jelikož $|vwx| \leq n$, slova v, x mohou zasahovat nejvýš do jednoho úseku nul a nejvýš do jednoho úseku jedniček v $z = 0^n 1^n 0^n 1^n$ (přičemž alespoň do jednoho úseku zasahují, tj. mají s ním neprázdný průnik). Tedy lze psát $uwy = 0^{k_1} 1^{k_2} 0^{\ell_1} 1^{\ell_2}$, kde určitě $k_1 \neq \ell_1$ nebo $k_2 \neq \ell_2$. To znamená, že uwy nepatří do L_2, což je spor (s předpokladem, že L_2 je bezkontextový).

Poznámka: Uvědomme si, že důkaz nebezkontextovosti není nijak mechanický. Musíme si nejprve důkladně promyslet, v čem je podstata toho, že daný jazyk bezkontextový není, a podle toho volit konkrétní slovo z k vyvození sporu. Kdybychom např. v předchozím příkladu zvolili slovo $z = 01^n 01^n$, spor bychom nevyvodili, protože tohle slovo se dá psát $z = uvwxy$ tak, že příslušné podmínky jsou splněny.

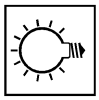
Kontrolní otázka: Jak konkrétně?

(Jistě jste přišli na to, že lze volit $u = 01^{n-1}$, $v = 1$, $w = 0$, $x = 1$, $y = 1^{n-1}$.)



ŘEŠENÝ PŘÍKLAD 4.3: Ukažte, že jazyk $L_3 = \{\, 0^m 1^n 0^m \mid 0 \leq n \leq m \,\}$ není bezkontextový.

Řešení: Kdyby byl, existovalo by příslušné n podle pumping lemmatu. Tedy např. slovo $z = 0^n1^n0^n$ (pro nějž platí $z \in L_3$ a $|z| \geq n$) by se dalo psát ve tvaru $z = uvwxy$, kde mj. platí $v \neq \varepsilon$ nebo $x \neq \varepsilon$, $|vwx| \leq n$ a $uvvwxxy \in L_3$ (zde je k vyvození sporu více užitečnější skutečné pumpování v, x než „nulové pumpování“, tj. odebírání v, x). Jelikož $|vwx| \leq n$, tak slova v, x mohou zasahovat nejvýš do jednoho úseku nul. Když do některého zasahují (mají s ním neprázdný průnik), tak slovo $uvvwxxy$ buď není ve tvaru $0^{k_1}1^{k_2}0^{k_3}$ nebo v něm je, ale pak $k_1 \neq k_3$; slovo $uvvwxxy$ tudíž nepatří do L_3 (spor). Když obě v, x jsou celé v úseku jedniček, tak slovo $uvvwxxy$ je tvaru $0^n1^{n+k}0^n$, kde $k \geq 1$, a i tak tedy nepatří do L_3 (spor).



Řešený příklad 4.4: Ukažte, že jazyk $L_4 = \{\, a^k \mid k = n^2$ pro nějaké $n \geq 1 \,\}$ není bezkontextový.

Řešení: Základní myšlenka je v tom, že mezery mezi druhými mocninami přirozených čísel neomezeně rostou. Precizovat to můžeme zase následovně. Kdyby L_4 byl bezkontextový, existovalo by příslušné n podle pumping lemmatu. Tedy např. slovo $z = a^{(n+1)^2} = a^{n^2}a^{2n}a$ (pro nějž platí $z \in L_4$ a $|z| \geq n$) by se dalo psát ve tvaru $z = uvwxy$, kde mj. platí $v \neq \varepsilon$ nebo $x \neq \varepsilon$, $|vwx| \leq n$ a $uwy \in L_4$. Ovšem uwy je očividně rovno $a^{n^2}a^{2n-k}a$, kde $1 \leq k \leq n$; tedy $uwy = a^j$, kde j není druhou mocninou přirozeného čísla, a tudíž uwy nepatří do L_4 (spor).

Na závěr si své vybudované porozumění regulárním a bezkontextovým jazykům prověřte na následujícím cvičení.

Cvičení 4.5: Zjistěte, které z daných jazyků

jsou regulární:

jsou bezkontextové, ale ne regulární:

nejsou bezkontextové:

$L_1 = \{w \in \{a, b\}^* \mid |w|_a = |w|_b\}$
$L_2 = \{w \in \{a, b\}^* \mid |w|_a$ je sudé $\}$
$L_3 = \{w \in \{a, b\}^* \mid w$ obsahuje podslovo $abba\}$
$L_4 = \{w \in \{a, b, c\}^* \mid |w|_a = |w|_b = |w|_c\}$
$L_5 = \{w \in \{a, b\}^* \mid |w|_a$ je prvočíslo$\}$

$L_6 = \{\, 0^m 1^n \mid m \leq 2n \,\}$
$L_7 = \{\, 0^m 1^n 0^m \mid m = 2n \,\}$

Shrnutí: Máme tedy dobrou představu o tom, jak idea velkých derivačních stromů, v nichž se nutně opakují neterminály na nějaké větvi, umožňuje přesně prokázat (odvozením logického sporu), že některé konkrétní jazyky nejsou bezkontextové; umíme to shrnout ve formě pumping lemmatu. Máme vybudováno porozumění, díky němuž jsme schopni klasifikovat zadané jazyky jako regulární či bezkontextové neregulární či nebezkontextové.

Kapitola 5

Úvod do teorie vyčíslitelnosti

Cíle kapitoly:

V této kapitole si máte ujasnit, co to je (výpočetní) „problém“ a co to znamená, že „algoritmus A řeší problém P“, včetně souvislostí s kódováním vstupů a výstupů. Máte zvládnout základní programování Turingových strojů a počítačů s libovolným přístupem („RAMů“). Máte pochopit vzájemnou simulaci mezi výpočetními modely a důvody, proč přijímáme Churchovu-Turingovu tezi, jež ztotožňuje pojem „algoritmus“ s pojmem „Turingův stroj“. Máte se seznámit s některými algoritmicky nerozhodnutelnými problémy a udělat si představu o způsobu důkazu nerozhodnutelnosti. Speciálně máte pochopit pojem algoritmické převeditelnosti mezi problémy.

Začínáme se věnovat základům teorie algoritmů. V této kapitole jde o (algoritmickou) vyčíslitelnost, v další kapitole pak o (výpočetní) složitost.

5.1 Problémy a algoritmy k jejich řešení

Orientační čas ke studiu této části: 2 hod.

Cíle této části:

V této části máte pochopit specifický význam pojmu „výpočetní problém“, stručněji „problém“, v teorii algoritmů; speciálně pak pojem

„rozhodovací (neboli ANO/NE) problém“ a také „optimalizační problém“ a jeho „ANO/NE verze“. Máte umět precizovat, co to znamená, že „algoritmus A řeší problém P“, či „algoritmus A rozhoduje problém P“ v případě ANO/NE problémů. Máte si zatím alespoň uvědomit, že ne všechny (výpočetní) problémy jsou algoritmicky řešitelné.

Klíčová slova: *(výpočetní) problém, kódování vstupů a výstupů, algoritmicky řešitelné (a neřešitelné) problémy, ANO/NE problém, rozhodnutelné (a nerozhodnutelné) problémy, optimalizační problém, ANO/NE verze optimalizačního problému*

Zkusme začít otázkou: Co je to vlastně *algoritmus*?

V historii lidstva se tento pojem postupně utvářel. Konkrétní algoritmy známe již ze starověku (vzpomeňme si třeba na Euklidův algoritmus pro nalezení největšího společného dělitele dvou čísel), ale teprve ve dvacátém století nazrála nutnost zabývat se pojmem *algoritmus* jako takovým. Všimněme si, že pokud chceme tento pojem nějak podrobněji popsat, užíváme slova a slovní spojení jako „postup“, „návod“, „posloupnost elementárních kroků“ apod. a požadujeme, aby takový postup bylo možné provádět „mechanicky“ (bez potřeby nějaké tvůrčí invence apod.), aby jeho jednotlivé kroky byly konečné (finitní), atp.
Uvědomme si však, že toto nejsou přesné definice (v „matematickém smyslu“). Pojem *algoritmus* je totiž podobně jako pojem *množina* základním pojmem, který není definován pomocí jednodušších pojmů, je jen objasňován svými vlastnostmi na konkrétních příkladech.
Za chvíli se dostaneme k matematickým *modelům algoritmu* (jako jsou Turingovy stroje či stroje RAM); teď se však nejprve podíváme na to, k čemu vlastně algoritmy slouží. Obecně se dá říci, že algoritmy slouží k řešení problémů.

Definice pojmu „problém“

Slovo *problém* má v přirozeném jazyce hodně významů; my se zaměříme jen na jeden specifický význam, který nyní přiblížíme. Příklady problémů, kterými se budeme zabývat jsou například problémy typu sečíst dvě čísla, nalézt nejkratší cestu v grafu, zjistit, zda je dané číslo prvočíslem, vynásobit dvě matice, setřídit danou databázi podle zvoleného klíče apod., tj. typicky

problémy, které se dají přesně formulovat pomocí matematických pojmů, a u kterých má rozumný smysl uvažovat o tom, že k jejich řešení použijeme počítač. Tyto problémy budeme většinou zadávat následujícím schématem:

NÁZEV: *XY*

VSTUP: Zde je popsáno, co je přípustným vstupem (zadáním, instancí) našeho problému.

VÝSTUP: Zde je popsáno, jaký výstup (výsledek) je očekáván pro zadaný vstup (je přiřazen zadanému vstupu).

Problém sečíst dvě přirozená čísla zadaný podle schématu vypadá např. takto:

NÁZEV: *Součet dvou čísel*

VSTUP: Dvojice přirozených čísel x a y.

VÝSTUP: Přirozené číslo z takové, že $z = x + y$.

Problém nalezení nejkratší cesty v grafu lze zadat např. takto:

NÁZEV: *Nejkratší cesta v grafu*

VSTUP: Orientovaný graf $G = (V, E)$ a dvojice vrcholů $u, v \in V$.

VÝSTUP: Nejkratší cesta z u do v, tj. nejkratší sekvence $v_0, v_1, \ldots, v_k$ prvků V taková, že $v_0 = u$, $v_k = v$ a $(v_{i-1}, v_i) \in E$ pro $i \in \{1, 2, \ldots, k\}$, případně odpověď „neexistuje“, pokud žádná cesta z u do v v G není.

Pokud chceme napsat formální definici pojmu problém, mohla by vypadat takto:

Definice 5.1
Problém je určen trojicí (IN, OUT, p), kde IN je množina (přípustných) *vstupů*, OUT je množina *výstupů* a $p : IN \to OUT$ je funkce přiřazující každému vstupu odpovídající výstup.

Takto definovaný problém se někdy též nazývá *výpočetní problém* (computational problem). Nás budou u takových problémů především zajímat algoritmy, které je řeší. Abychom ovšem mohli říci, co to znamená, že algoritmus A řeší problém P, musíme si něco říci o kódování vstupů a výstupů.

Kódování vstupů a výstupů

Ve výše uvedené definici pojmu „problém“ jsme neřekli, jaké mohou být prvky množin IN a OUT. Je zřejmé, že pro účely algoritmického („počítačového“) řešení musí být vstupy a výstupy nějak rozumně reprezentovány konečným způsobem.

Pro konkrétnost si jako příklady toho, co mohou být prvky množin IN a OUT, uveďme

- slova v nějaké dané abecedě Σ,
- slova v abecedě $\{0, 1\}$, tj. sekvence bitů,
- sekvence zápisů celých čísel,
- zápisy přirozených čísel.

Ve skutečnosti lze tyto různé typy chápat jako ekvivalentní; např.:

- Slova libovolné abecedy Σ je možné reprezentovat sekvencemi přirozených čísel:

  stačí totiž očíslovat symboly abecedy (tedy přiřadit jim číselné kódy, jak to známe např. u ASCII).

  Např. pro $\Sigma = \{a, b, c, d\}$ můžeme znaku a přiřadit číslo 0, znaku b číslo 1, znaku c číslo 2 a znaku d číslo 3. Slovo *bddaba* pak bude reprezentováno jako posloupnost $1, 3, 3, 0, 1, 0$.

- Slova libovolné abecedy Σ je možné reprezentovat jako slova v abecedě $\{0, 1\}$:

  slova převedeme na sekvence čísel jako v předchozím případě, přičemž čísla zapíšeme binárně jako k-bitová čísla, kde k musí být zvoleno dostatečně velké, aby bylo možné reprezentovat všechny symboly abecedy (např. 7 bitů u ASCII kóduje 128-prvkovou abecedu).

Pro $\Sigma = \{a, b, c, d\}$ stačí tedy použít dva bity, kde znak a je chápán jako 00, znak b jako 01, znak c jako 10 a znak d jako 11. Slovo *bddaba* pak zapíšeme jako 011111000100.

- Přirozená čísla je možno zapisovat jako slova v abecedě $\{0, 1\}$:

  stačí použít binární zápis čísla.

  Například číslo 22 je možné zapsat jako `10110`.

- Sekvence celých čísel je možné reprezentovat jako slova ve vhodně zvolené abecedě Σ:

  například můžeme čísla zapisovat binárně, pro označení záporných čísel používat znak `-` a pro oddělení jednotlivých čísel používat znak `#`. V tomto případě je tedy $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{-}, \texttt{\#}\}$, a například sekvence

  $$4, -6, 3, 0, 13, -5$$

  je pak reprezentována slovem

  `100#-110#11#0#1101#-101`

- Slova v abecedě $\{0, 1\}$ je možné reprezentovat přirozenými čísly:

  na začátek slova přidáme symbol 1 a výsledné slovo chápeme jako zápis čísla ve dvojkové soustavě. Například slovu 0110 přiřadíme číslo 22 (tj. 10110 binárně).

  Pozn.: Přidání symbolu 1 na začátek je třeba, abychom byli schopni rozlišit slova, která se liší jen počtem nul na začátku, např. slova 1, 01, 001 atd.

Podotkněme, že výše popsané transformace samozřejmě nejsou jediné možné.

Cvičení 5.1: Pro každý z výše uvedených převodů jedné reprezentace na druhou uveďte nějaký alternativní způsob, jak by bylo možné onen převod provést.

Další typy objektů pak můžeme reprezentovat pomocí výše popsaných.
Pokud je například vstupem matice čísel, je možné ji reprezentovat jako slovo v nějaké abecedě, přičemž jednotlivé řádky budou zapsány za sebou,

oddělený nějakým speciálním oddělovacím znakem, a každý jednotlivý řádek bude reprezentován podobným způsobem, jaký jsme použili pro reprezentaci sekvence čísel.
Grafy můžeme reprezentovat např. jako sekvence tvořené seznamem vrcholů a seznamem hran, případně pomocí incidenční matice.
Logické formule můžeme reprezentovat jako slova v nějaké vhodně zvolené abecedě, apod.

Algoritmicky řešitelné problémy

Co to vlastně znamená, že algoritmus A řeší problém P ? Můžeme to zpřesnit např. takto:

> Algoritmus A *řeší problém* P zadaný trojicí (IN, OUT, p) spolu s dohodnutým (většinou samozřejmým a tedy ani nezmiňovaným) kódováním vstupů a výstupů (tj. prvků množin IN a OUT), jestliže je schopen přijmout (přečíst) kód jakéhokoli vstupu x z množiny IN a vydat k němu po konečném počtu kroků kód výstupu y z množiny OUT, pro nějž $y = p(x)$.

Stručně to vyjádříme takto: A pro každý vstup problému P (vypočte a) vydá předepsaný výstup. (O kódování se většinou vůbec nezmiňujeme, protože tiše předpokládáme nějaké přirozené kódování slovy v abecedě, posloupnostmi bitů či podobně.)

Kontrolní otázka: Podívejte se ještě jednou na zadání problému nejkratší cesty v grafu a promyslete si, zda mu skutečně jednoznačně odpovídá trojice (IN, OUT, p).

I když pomineme různé možnosti zakódování grafu řetězcem symbolů apod., výstup není pro každý vstup jednoznačně určen.

Poznámka a úmluva.
Někdy má dobrý smysl uvažovat i problémy, kde pro jeden vstup může existovat více správných výstupů, jako je tomu např. u uvedeného problému hledání nejkratší cesty v grafu (je zřejmé, že mezi dvojicí vrcholů může existovat více než jedna nejkratší cesta).
Po algoritmu, který by tento problém řešil, chceme, aby ke každému vstupu vydal (alespoň) jeden správný výstup.

Alternativně jsme tedy mohli pojem „problém“ definovat poněkud obecněji, jako trojici (IN, OUT, P), kde význam IN a OUT je stejný jako dříve a $P \subseteq IN \times OUT$ je relace, která musí splňovat, že ke každému $x \in IN$ existuje alespoň jedno $y \in OUT$ takové, že $(x, y) \in P$. (Výraz $(x, y) \in P$ lze pak chápat tak, že y je jeden z korektních výstupů pro vstup x.)
Tohoto detailu si buďme vědomi, i když jej u konkrétních problémů již nebudeme explicitně zmiňovat.

Teď si ještě uvědomme jednu důležitou věc, která možná není hned zřejmá:

Existují problémy, které nejsou algoritmicky řešitelné.

Tedy není pravda, že ke každému přesně matematicky specifikovanému problému (s přirozeným kódováním vstupů a výstupů atd.) existuje algoritmus, který ho řeší. Teď to jen zaznamenáme a později se k tomu vrátíme. Uvedeme si ale alespoň jeden příklad takového algoritmicky neřešitelného problému:

NÁZEV: Eq-CFG (*Ekvivalence bezkontextových gramatik*)

VSTUP: Dvě bezkontextové gramatiky G_1, G_2.

VÝSTUP: ANO, když $L(G_1) = L(G_2)$, NE, když $L(G_1) \neq L(G_2)$.

ANO/NE problémy, ANO/NE verze optimalizačních problémů

Všimněme si, že např. popis výstupu výše uvedeného problému Eq-CFG působí poněkud neohrabaně. Jedná se o tzv.

rozhodovací problém, neboli *ANO/NE problém*;

u takového problému je množina OUT dvouprvková, standardně chápána jako $OUT = \{\text{ANO}, \text{NE}\}$ (či $OUT = \{1, 0\}$).
U takových problémů je přirozenější a kratší definovat žádaný výstup pomocí otázky (týkající se vstupu), na kterou je odpověď buď ANO nebo NE. Např. Eq-CFG pak zapíšeme takto:

NÁZEV: Eq-CFG (*Ekvivalence bezkontextových gramatik*)

VSTUP: Dvě bezkontextové gramatiky G_1, G_2.

OTÁZKA: Platí $L(G_1) = L(G_2)$?

Jiným příkladem ANO/NE problému je problém prvočíselnosti:

NÁZEV: *Prvočíselnost*

VSTUP: Přirozené číslo x (zadané např. dekadickým zápisem).

OTÁZKA: Je x prvočíslo?

Obecně tedy budeme pro rozhodovací problémy (neboli ANO/NE problémy) používat následující schéma:

NÁZEV: *XY*

VSTUP: Zde je popsáno, co je přípustným vstupem (zadáním, instancí) našeho problému.

OTÁZKA: Zde je otázka týkající se (zadaného) vstupu, na niž je odpověď ANO nebo NE.

U ANO/NE problémů místo „algoritmus A řeší problém XY“ často také říkáme

algoritmus A rozhoduje problém XY.

ANO/NE problém, pro nějž existuje algoritmus, který jej rozhoduje, nazýváme

algoritmicky rozhodnutelný, nebo stručněji *rozhodnutelný*.

Dříve uvedený problém Eq-CFG je tedy *algoritmicky nerozhodnutelný*, stručněji *nerozhodnutelný*.

Uveďme ještě jeden ANO/NE problém, který bude hrát později důležitou roli.

NÁZEV: SAT (*problém splnitelnosti booleovských formulí*)

VSTUP: Booleovská formule v konjunktivní normální formě.

OTÁZKA: Je daná formule splnitelná (tj. existuje pravdivostní ohodnocení proměnných, při kterém je formule pravdivá)?

Kontrolní otázka: Umíte okamžitě napsat příklad vstupu (neboli instance) problému SAT, u nějž je odpověď ANO a příklad vstupu, u nějž je odpověď NE?

> *Takové otázky byste si měli klást a řešit při studiu tohoto textu aktivně sami, nejen když jste takto explicitně tázáni.*

(Zde stačí samozřejmě uvést např. velmi jednoduché formule s jedinou proměnnou: x a $y \wedge \neg y$ či podobné.)

V praxi velmi důležitou třídou problémů jsou tzv.

> *optimalizační problémy.*

U optimalizačního problému je pro každý vstup určena množina *přípustných řešení* a dále je definována určitá kriteriální funkce, která každému přípustnému řešení přiřazuje nějaké reálné číslo. Cílem je mezi všemi přípustnými řešeními pro daný vstup nalézt to, pro které je hodnota kriteriální funkce největší, nebo případně nejmenší, v závislosti na typu řešeného problému.

Příkladem optimalizačního problému je již dříve uvedený problém hledání nejkratší cesty v grafu. V tomto případě je množinou přípustných řešení množina všech cest mezi dvěma danými vrcholy, kriteriální funkcí je délka dané cesty a cílem je hodnotu kriteriální funkce minimalizovat.

Jiným příkladem je problém nalezení minimální kostry v grafu:

NÁZEV: *Minimální kostra*

VSTUP: Neorientovaný souvislý graf $G = (V_G, E_G)$, ohodnocení hran $f : E \to \mathbb{N}_+$.

VÝSTUP: Souvislý graf $H = (V_H, E_H)$, kde $V_H = V_G$ a $E_H \subseteq E_G$, a kde hodnota $\sum_{e \in E_H} f(e)$ je minimální.

V tomto případě je množinou všech přípustných řešení množina všech souvislých podgrafů H grafu G takových, že $V_H = V_G$. Hodnota kriteriální funkce pro dané přípustné řešení H je dána výrazem $\sum_{e \in E_H} f(e)$. Opět je cílem minimalizovat tuto hodnotu.

Poznamenejme, že k některým (obecným) problémům (např. optimalizačním) lze přirozeně přiřadit tzv.

rozhodovací (neboli ANO/NE) verzi problému:

např. u problému minimální kostry se vstup rozšíří o číslo c a požadovaný výstup pak bude ANO, jestliže existuje kostra s ohodnocením nejvýše rovným c, a NE v opačném případě:

NÁZEV: *Minimální kostra (ANO/NE verze)*

VSTUP: Neorientovaný souvislý graf $G = (V_G, E_G)$, ohodnocení hran $f : E \to \mathbb{N}_+$ a číslo c.

OTÁZKA: Existuje graf $H = (V_H, E_H)$, kde $V_H = V_G$ a $E_H \subseteq E_G$, který je souvislý a přitom $\sum_{e \in E_H} f(e) \leq c$?

Poznámka pro hloubavé čtenáře. V teorii algoritmické vyčíslitelnosti a složitosti se většinou uvažují jen ANO/NE problémy, pro svou jednodušší formu. Naznačme, proč to neznamená vážnou újmu na obecnosti zkoumání:
K obecnému problému (trojici (IN, OUT, p))

NÁZEV: *XY*

VSTUP: $w \in IN$

VÝSTUP: $p(w)$

můžeme definovat ANO/NE problém

NÁZEV: *XY-output-bit*

VSTUP: $w \in IN$, $i \in \mathbb{N}_+$, $h \in \{0, 1\}$.

OTÁZKA: Má $p(w)$ v „bitovém kódování“ alespoň i bitů a je v kladném případě i-tý bit roven h ?

Máme-li algoritmus rozhodující problém XY-output-bit, tak s jeho využitím snadno sestavíme algoritmus řešící problém XY. Zkuste si to promyslet!

Shrnutí: Takže už je nám jasné, co se v teorii algoritmů míní pojmem „problém“ a co to znamená, že daný algoritmus řeší daný problém; je nám v této souvislosti naprosto jasná také problematika kódování vstupů a výstupů.

Umíme specifikovat konkrétní problémy z praxe a rozlišovat mj. ANO/NE problémy, optimalizační problémy a také ANO/NE verze optimalizačních problémů. Víme, že se v teorii speciálně zaměřujeme na ANO/NE problémy (kterým se také říká rozhodovací problémy [decision problems]) a mezi nimi rozlišujeme rozhodnutelné [decidable] a nerozhodnutelné [undecidable] problémy.

5.2 Turingovy stroje

Orientační čas ke studiu této části: 4 hod.

Cíle této části:

V této části máte zvládnout programovací jazyk „velmi nízké úrovně“, mající jen jeden typ instrukcí; jde o tzv. Turingovy stroje. Samozřejmě to neznamená jen naučit se definici, ale především získat porozumění dostatečné k tomu, abyste byli schopni sestavovat (jednoduché) programy (tj. konkrétní Turingovy stroje) a pochopili, že přes svou primitivnost mohou tyto stroje implementovat tytéž algoritmy, jako každý jiný (univerzální) programovací jazyk.

Klíčová slova: *Turingovy stroje, výpočet (konečný i nekonečný), Turingovy stroje řešící problémy, Turingovy stroje rozhodující ANO/NE problémy, Turingovy stroje rozpoznávající (neboli rozhodující) jazyky*

Již jsme diskutovali fakt, že „algoritmus“ je základní pojem, který nelze definovat pomocí jiných, jednodušších pojmů; lze jej v principu jen ilustrovat na příkladech, na nichž popisujeme jeho vlastnosti.
Víme také, že pojem „algoritmus“ nijak neurčuje, jak máme zapisovat konkrétní algoritmy; lze k tomu vhodným způsobem použít přirozený jazyk, případně doplněný o obrázky, matematické symboly apod. Když se však objevily počítače, tedy stroje k provádění algoritmů, bylo potřeba exaktně specifikovat způsob zápisu algoritmů tak, aby je mohl přečíst a provádět počítač. Všichni víme, že proto byly vyvinuty programovací jazyky. Je sice mnoho různých programovacích jazyků, ale v principu jsou všechny rovnocenné: zapíšeme-li

nějaký algoritmus v jednom z nich, je jistě možné jej zapsat také v kterémkoli jiném (i když třeba komplikovaněji); dokonce lze toto „přepsání programu do jiného jazyka“ svěřit počítači, máme-li k dispozici příslušný překladač.

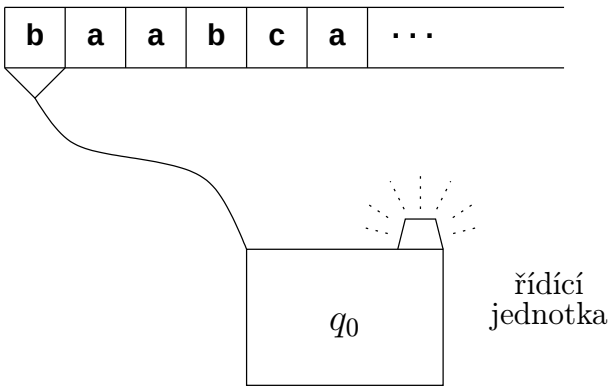
Poznámka: Hovoříme samozřejmě o univerzálních programovacích jazycích jako Java, C-čko, Fortran, Pascal, ne o některých jazycích se specifickým účelem, do nichž nejdou univerzální programovací jazyky překládat.

To nás přirozeně vede k nápadu ztotožnit pojem „algoritmus“ např. s pojmem „program v jazyce Java“. Chápeme přitom ovšem, že ztotožnění pojmu „algoritmus“ s pojmem „program v jazyce Pascal“ je ekvivalentní možnost. Podobně je ekvivalentní možnost ztotožnění pojmu „algoritmus“ s pojmem „posloupnost instrukcí strojového jazyka procesoru Pentium“. Ekvivalenci oněch možností je třeba rozumět v tom smyslu, že „co naprogramujeme v jednom jazyku, lze naprogramovat i v druhém“, byť z důvodu rychlosti a přehlednosti programování je třeba jeden z jazyků výrazně příjemnější než jiný.

Ve skutečnosti se zpřesnění zápisu algoritmů a ztotožnění pojmu „algoritmus“ a pojmu „program v určitém jazyku“ objevilo už ve třicátých letech dvacátého století, ještě před rozmachem elektronických počítačů. Objevilo se při snaze prokázat algoritmickou nerozhodnutelnost jistých problémů (z logiky a základů matematiky). Historicky prvním „univerzálním programovacím jazykem“ jsou tak *Turingovy stroje*. Seznámíme se teď s nimi, a pak se k naší diskusi vrátíme.

Alan Turing se snažil návrhem toho, čemu říkáme Turingovy stroje, formalizovat práci „výpočtáře“, pracujícího s tužkou, gumou a papírem a svou omezenou pamětí. Nepůjdeme do detailů myšlenkových pochodů Alana Turinga, ale k přiblížení navrženého modelu si nejdříve připomeneme konečný automat; zopakujeme k tomu obrázek 2.3, nyní jako obrázek 5.1. Konečný automat má tedy řídicí jednotku s konečně mnoha vnitřními stavy (tedy s omezenou pamětí) a na vstupní pásce čte (pomocí čtecí hlavy) zleva doprava zadané vstupní slovo; přitom mění vnitřní stavy podle příslušné přechodové funkce, kterou lze vlastně chápat jako množinu instrukcí.

Poznámka: V běžných programovacích jazycích je program *posloupností* instrukcí; na pořadí instrukcí záleží. Jelikož u konečného automatu je každá instrukce typu $q \xrightarrow{a} q'$, či jinak psáno $(q, a) \longrightarrow q'$, tak na pořadí instrukcí nezáleží (každá de facto předepisuje skok na specifikované „návěští“, nikdy



Obrázek 5.1:

se nepokračuje „další instrukcí v pořadí“); je ovšem potřeba říci, kde se má začít, tedy je nutné specifikovat počáteční stav (=návěští).

Turingův stroj je podobný konečnému automatu, významný rozdíl je ale popsán v následujícím:

- páska je oboustranně nekonečná; je na ní na začátku zapsáno vstupní slovo, na jehož první pozici stojí hlava, a ostatní buňky jsou prázdné, tj. je v nich zapsán speciální prázdný znak $\Box$,
- hlava (spojená s konečnou řídicí jednotkou) se může pohybovat po pásce *oběma směry* a je nejen čtecí, ale i *zapisovací* – symboly v buňkách pásky je tedy možné přepisovat, a to obecně i jinými než vstupními symboly.

Po zkušenostech s konečnými a zásobníkovými automaty nás nijak nepřekvapí následující formální definice.

Definice 5.2
Turingův stroj, zkráceně TS, je definován jako šestice $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, kde

- Q je konečná neprázdná množina *stavů*,
- Σ je konečná neprázdná množina *vstupních symbolů*,

- Γ je konečná neprázdná množina *páskových symbolů*, kde $\Sigma \subseteq \Gamma$ a v $\Gamma - \Sigma$ je (přinejmenším) speciální znak $\Box$ (prázdný znak [blank]),
- $q_0 \in Q$ je *počáteční stav*,
- $F \subseteq Q$ je množina *koncových stavů*,
- $\delta : (Q - F) \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$ je přechodová funkce.

Podobně jako např. u zásobníkového automatu můžeme konkrétní Turingův stroj zadat seznamem *instrukcí*. Tyto instrukce jsou zase dány přechodovou funkcí; místo $\delta(q, a) = (q', a', m)$ píšeme raději $(q, a) \to (q', a', m)$.

Význam instrukce $(q, a) \to (q', a', m)$ je tento:

- Tato instrukce je aplikovatelná v situaci (neboli konfiguraci), kdy řídicí jednotka je ve stavu q a hlava čte na pásce symbol a.
- Vykonání instrukce znamená následující:
  - řídicí jednotka přejde do stavu q',
  - hlava zapíše do aktuálně čtené buňky symbol a',
  - je-li $m = +1$, hlava se posune na sousední buňku pásky doprava, je-li $m = -1$, hlava se posune na sousední buňku pásky doleva, je-li $m = 0$, hlava se nikam neposune.

Podívejme se teď na následující seznam instrukcí.

$(q_1, a) \to (q_2, \bar{a}, +1)$
$(q_1, \Box) \to (q_f, \Box, 0)$
$(q_2, a) \to (q_2, a, +1)$
$(q_2, \bar{a}) \to (q_2, \bar{a}, +1)$
$(q_2, \Box) \to (q_3, \bar{a}, -1)$
$(q_3, \bar{a}) \to (q_3, \bar{a}, -1)$
$(q_3, a) \to (q_4, a, -1)$
$(q_3, \Box) \to (q_5, \Box, +1)$
$(q_4, a) \to (q_4, a, -1)$
$(q_4, \bar{a}) \to (q_1, \bar{a}, +1)$
$(q_5, \bar{a}) \to (q_5, a, +1)$
$(q_5, \Box) \to (q_f, \Box, 0)$

Tento seznam reprezentuje určitý Turingův stroj $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$.
Je zřejmé, že $Q = \{q_1, q_2, q_3, q_4, q_5, q_f\}$ a $\Gamma = \{a, \bar{a}, \Box\}$.
Které symboly jsou vstupní, tedy co je množinou Σ, není ze seznamu instrukcí zřejmé; musíme dodat, že zde bereme $\Sigma = \{a\}$.
Jako počáteční chápeme (obvykle) ten stav, kterým začíná první instrukce v seznamu. V našem případě je tedy $q_0 = q_1$; stroj raději můžeme rovnou zapsat ve formě $M = (Q, \Sigma, \Gamma, \delta, q_1, F)$.
Jak vidíme z definice přechodové funkce, koncové stavy u Turingova stroje jsou „opravdu koncové“, což znamená, že

výpočet po dosažení koncového stavu končí

(což je rozdíl oproti konečným či zásobníkovým automatům).
Koncový stav se tedy neobjeví na levé straně žádné instrukce. V našem příkladu tak snadno vyvodíme, že koncovým stavem je jedině q_f, tedy $F = \{q_f\}$. (Samozřejmě můžeme pro větší názornost informaci o množině F k seznamu explicitně dodat.)
Seznam instrukcí je ovšem především reprezentací přechodové funkce δ; např. vidíme $\delta(q_1, a) = (q_2, \bar{a}, +1)$, $\delta(q_2, a) = (q_2, a, +1)$ atd. Všimněme si přitom, že náš seznam nereprezentuje funkci δ plně, protože např. neobsahuje instrukci s levou stranou $(q_1, \bar{a})$, apod. Takové „vynechávání“ často používáme, když navrhujeme Turingův stroj (tedy „programujeme“) a jsme si jisti, že k příslušné situaci nemůže při výpočtu nad jakýmkoli vstupním slovem dojít. (Náš stroj skutečně nemůže dosáhnout konfigurace, v níž by byla řídicí jednotka ve stavu q_1 a hlava četla $\bar{a}$.) Pro úplnost si ale můžeme např. představit, že seznam je vždy automaticky doplněn tak, že pro chybějící levou stranu (q, a) se dodá instrukce $(q, a) \to (q_{fail}, a, 0)$, kde q_{fail} je speciální koncový stav znamenající „ukončení havárií“ („run-time error“).

Poznámka: Jak vidíme z definice, základní verze Turingova stroje je *deterministická*: nemůžeme mít dvě různé instrukce se stejnou levou stranou a ke každému vstupnímu slovu tak existuje jediný výpočet. To je také vlastnost, kterou u algoritmů samozřejmě předpokládáme. (Až později budeme mít důvod zkoumat také nedeterministické Turingovy stroje.)

Co je to

výpočet Turingova stroje M nad vstupním slovem w

je už čtenáři jistě jasné. Zpřesnit se tento pojem zase dá pomocí relace odvozování mezi konfiguracemi.

Konfigurace Turingova stroje je dána stavem řídicí jednotky, obsahem pásky a pozicí hlavy. Konkrétní konfiguraci stroje $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ popíšeme trojicí

(u, q, v), či zkráceně uqv, kde $u, v \in \Gamma^*$ a $q \in Q$.

Rozumíme tomu tak, že na pásce je v dané konfiguraci zapsáno slovo uv a vlevo i vpravo od něj jsou jen prázdné symboly $\Box$; řídicí jednotka je ve stavu q a hlava stojí na buňce odpovídající první pozici ve v. Všimněme si, že zápis uqv představuje stejnou konfiguraci jako $\Box^i uqv\Box^j$ pro libovolné i, j; speciálně uq (kde tedy $v = \varepsilon$) je totéž jako $uq\Box$, takže je jasné, co zde míníme „první pozicí ve v“ i v případě $v = \varepsilon$. Nepůjdeme už do dalších formálních detailů, ale předvedeme výpočet našeho výše uvedeného Turingova stroje pro vstup aaa; výpočet tedy začíná v počáteční konfiguraci $q_1\,aaa$:

$q_1\,aaa \vdash \bar{a}\,q_2\,aa \vdash \bar{a}a\,q_2\,a \vdash \bar{a}aa\,q_2\,\Box \vdash \bar{a}a\,q_3\,a\bar{a} \vdash \bar{a}\,q_4\,aa\bar{a} \vdash q_4\,\bar{a}aa\bar{a} \vdash \bar{a}\,q_1\,aa\bar{a} \vdash \bar{a}\bar{a}\,q_2\,a\bar{a} \vdash \ldots$

Jistě jste pochopili, že relace $\vdash$ znamená odvození v jednom kroku (provedení změny aktuální konfigurace provedením jedné instrukce); měli bychom psát $\vdash_M$, pokud potřebujeme zdůraznit, že se jedná o odvození podle (instrukcí) stroje M.

Kontrolní otázka: Umíte započatý výpočet dokončit? (Udělejte to!)

Jistě jste zjistili, že výpočet skončí v konfiguraci $aaaaaa\,q_f$. Navíc jste jistě pochopili, že pro jakékoli vstupní slovo $w \in \{a\}^*$ výpočet skončí v konfiguraci s obsahem pásky ww.

Pro jistotu ještě zopakujme, že každý výpočet Turingova stroje začíná v *počáteční konfiguraci*, tj. konfiguraci q_0w, kde q_0 je počáteční stav a w je slovo nad vstupní abecedou. Výpočet je buď nekonečný nebo skončí v *koncové konfiguraci*, tj. konfiguraci uqv, kde q je nějaký koncový stav a slovo uv reprezentuje obsah pásky (pro konkrétnost bereme uv tak, že první pozice v uv odpovídá nejlevější buňce pásky obsahující neprázdný symbol (tj. jiný symbol než $\Box$) a poslední pozice v uv odpovídá nejpravější buňce pásky obsahující neprázdný symbol. (Může se také stát, že $uv = \varepsilon$, tedy že páska obsahuje na závěr jen symboly $\Box$.)

Obsah pásky uv v koncové konfiguraci uqv je chápán jako *výstup*, který stroj (vypočítá a) vydá pro zadané vstupní slovo, tedy pro zadaný *vstup*.

Kontrolní otázka: Umíte teď zformulovat, co to znamená, že Turingův stroj M řeší problém P ?

Je to jasné: ke každému vstupu (tedy instanci) problému P stroj M (vypočte a vydá) problémem předepsaný výstup. Samozřejmě předpokládáme, že vstupy a výstupy problému jsou kódovány slovy v nějaké abecedě. (Tuto abecedu lze vzít jako vstupní abecedu stroje M, přičemž M může samozřejmě používat i další („pracovní“) páskové symboly.

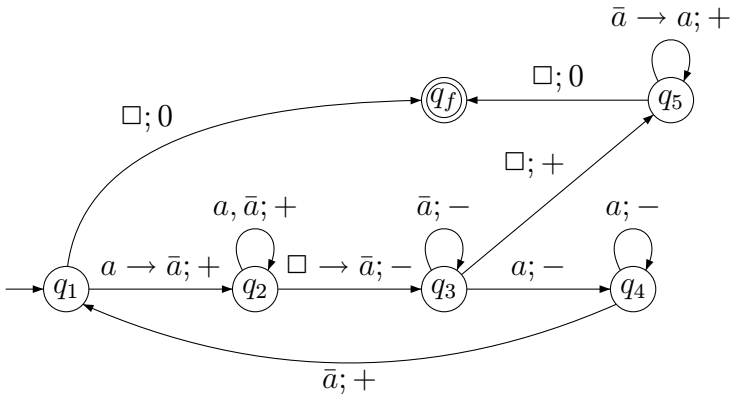
Náš výše uvedený stroj tedy vlastně řeší tento problém:

NÁZEV: *Zdvojení slov v jednoprvkové abecedě*

VSTUP: Slovo w nad abecedou $\{a\}$.

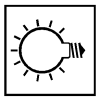
VÝSTUP: Slovo ww.

Značení: Jak jsme již uvedli, Turingovy stroje se dají zadávat seznamem instrukcí, s potřebnými dalšími informacemi, jako je specifikování vstupních symbolů, případně explicitním vyjmenováním stavů a určením počátečního a koncových stavů.
Seznam instrukcí ne náhodou připomíná počítačový program velmi nízké úrovně, dokonce s jen jedním typem instrukce. Aby byl delší program tohoto typu srozumitelný, je jistě možné a vhodné zápis řádně strukturovat, dobře okomentovat atd.
Někdy se místo seznamu instrukcí, či navíc vedle něj, může uvést zobrazení grafem, podobně jak to známe u konečných automatů. Označení hran je ovšem složitější a nutně méně srozumitelné než u konečného automatu. Např. náš stroj pro zdvojování slov je znázorněn grafem na dalším obrázku



Používáme tedy pro stručnost následující konvence. Vrcholy odpovídají stavům řídicí jednotky (počáteční stav je označený vstupující šipkou, koncové stavy jsou označeny dvojitým kroužkem); hrany reprezentují provedení jednotlivých instrukcí. Pro instrukci $(q, a) \to (q', a', m)$ vedeme hranu z vrcholu označeného q do vrcholu (označeného) q' a standardně k ní napíšeme $a \to a'; +$ když $m = +1$, $a \to a'; -$ když $m = -1$ a $a \to a'; 0$ když $m = 0$. Pokud $a' = a$, tedy symbol se nepřepisuje (tj. přepisuje se sebou samým), označujeme hranu stručněji $a; +$ (nebo $a; -$ či $a; 0$).
Jednotlivá hrana může mít více označení, protože instrukcí typu $(q, -) \to (q', -, -)$ může být více. Místo dvou označení $a; +$ a $b, +$ píšeme stručněji $a, b; +$; místo $a \to b; +$ a $a' \to b; +$ píšeme stručněji $a, a' \to b; +$. Podobně používáme zkratky samozřejmě i pro posun „$-$“ a „0“, a případně také pro tři a více podobných označení.

Čtenář může posoudit sám, který způsob (seznam instrukcí či graf) je pro něj přehlednější. V každém případě se u složitějších strojů neobejdeme bez komentářů, má-li být takový stroj (neboli program) srozumitelný. Sestrojme si teď několik dalších Turingových strojů, řešících zadané problémy.



Řešený příklad 5.1: Sestrojte TS, který jako vstup očekává binární číslo, tj. slovo z $\{0, 1\}^*$, a jako výstup má (spočítat a) vydat binární zápis trojnásobku vstupního čísla.

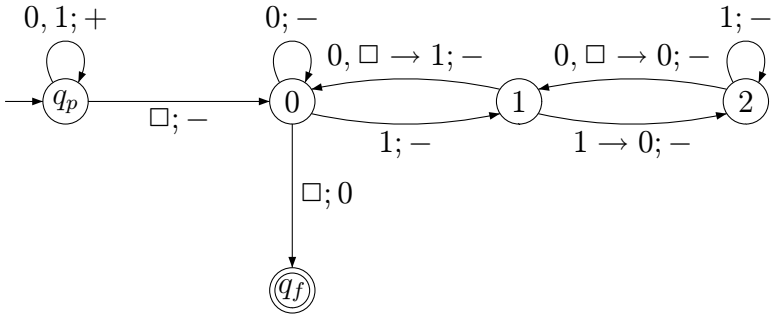
Řešení: Nejprve musíme samozřejmě sestavit algoritmus, který pak „naprogramujeme“, tedy který „implementujeme“ v instrukcích Turingova stroje. Algoritmus jistě rychle navrhneme; uvědomíme si, že je vhodné postupovat odzadu, přičemž platí:
- pokud je poslední číslice 0, v trojnásobku bude také 0 a nedojde k přenosu

(přenos bude 0),
- pokud je poslední číslice 1, v trojnásobku bude také 1 a navíc dojde k přenosu 1,
- pokud si pamatujeme přenos 1 a čteme 0, nahradíme ji 1 a vynulujeme přenos,
- atd.

Zbylá pravidla jistě snadno doplníte a uvědomíte si, že přenos může nabývat jen hodnot $0, 1, 2$.

Hodnotu přenosu si tedy TS může pamatovat stavem řídicí jednotky (tedy v omezené paměti). Stačí tedy naprogramovat počáteční fázi přechodu na pravý kraj vstupního slova a pak příslušně přepsat výše uvedená pravidla. Výpočet skončí, až stroj narazí (při zpracování zprava doleva) na $\Box$; nesmíme ale zapomenout, že na pásku se v té chvíli musí ještě zapsat aktuální přenos.

Výsledný stroj lze zachytit např. grafem na obrázku 5.2; kromě počátečního a koncového stavu použijeme tři stavy $0, 1, 2$, které odpovídají pamatovaným přenosům.



Obrázek 5.2: Turingův stroj realizující násobení třemi

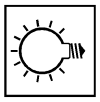
Zapišme si ještě předchozí Turingův stroj seznamem instrukcí. Přitom pro stručnost také využijeme *schémat instrukcí*, jak to již známe např. z popisů zásobníkových automatů. Stavy $0, 1, 2$ zde raději označujeme q_0, q_1, q_2, ať se výrazněji odliší od páskových symbolů $0, 1$.

$(q_p, x) \to (q_p, x, +1)$ pro $x \in \{0, 1\}$
$(q_p, \Box) \to (q_0, \Box, -1)$

$(q_0, 0) \to (q_0, 0, -1)$
$(q_0, 1) \to (q_1, 1, -1)$
$(q_0, \Box) \to (q_f, \Box, 0)$

$(q_1, y) \to (q_0, 1, -1)$ pro $y \in \{0, \Box\}$
$(q_1, 1) \to (q_2, 0, -1)$

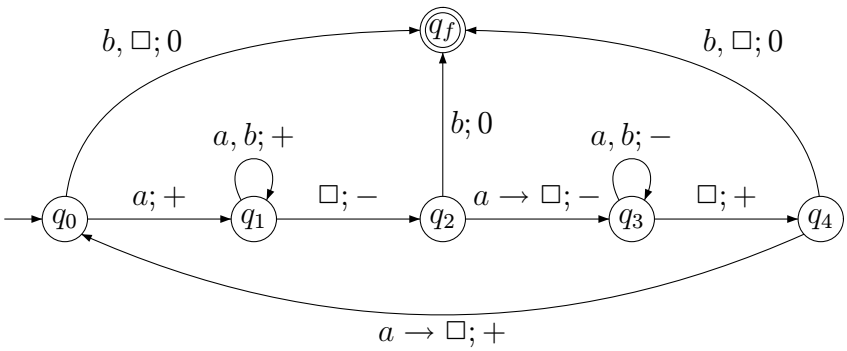
$(q_2, y) \to (q_1, 0, -1)$ pro $y \in \{0, \Box\}$
$(q_2, 1) \to (q_2, 1, -1)$



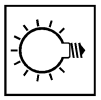
Řešený příklad 5.2: Navrhněte TS, který ze zadaného slova nad abecedou $\{a, b\}$ umaže od začátku i od konce nejdelší možné stejně dlouhé úseky znaků a (kde umazání samozřejmě znamená přepsání znaky $\Box$). Tedy např. k vstupnímu slovu *aaababaa* stroj nechá jako výstup slovo *abab*, kdežto vstupní slovo *aaabab* nezmění; ze slova *aaa* zbude ε.

Řešení: Nejprve si problém rozebereme. Pokud na začátku čte stroj $\Box$ (vstupní slovo je v tom případě ε) nebo b, může hned skončit. Pokud čte na začátku a, je nutno zkontrolovat, zda je a i na konci; v kladném případě stroj ono koncové a umaže a vrátí se na začátek, kde umaže ono počáteční a (pokud tam ještě je, totiž pokud na pásce nebylo jen slovo a). Pak stroj pokračuje stejným způsobem na zbylém slově (znovu se tedy provádí tělo určitého cyklu) atd., dokud nenarazí na znak znamenající ukončení výpočtu. Všimněme si, že Turingův stroj nemůže obecně počítat, kolik a je na začátku (či na konci) slova, protože jeho řídicí jednotka má omezenou paměť, tj. může se nacházet jen v omezeném počtu stavů. Proto jsme nuceni naprogramovat v nějaké formě onen postup „běhání zleva doprava“ a synchronizovaného mazání na začátku a na konci.

Navržený TS zase zachytíme grafem (čtenář si jej může přepsat ve formě instrukcí k procvičení). Hrany odpovídající instrukcím umazávajícím a-čka jsou v grafu zvýrazněny.



Poznámka. Všimněme si, že chybí instrukce popisující, co se má dělat, když stroj čte ve stavu q_2 znak $\square$. Jedná se opět o situaci, ke které nemůže dojít, takže to nevadí. (Samozřejmě bychom ovšem nic nezkazili, kdybychom znak $\square$ přidali např. k popisu hrany z q_2 do q_f .)



Řešený příklad 5.3: Navrhněte Turingův stroj, který pro vstupní slovo $w \in \{a, b, c\}^*$ vydá jako výstup slovo $v \in \{b, c\}^*$, které vznikne z w vypuštěním všech výskytů znaku a.

Řešení: Příklad se v prvním okamžiku může zdát triviální – TS by mohl projít celé slovo zleva doprava, přičemž by každý výskyt znaku a přepsal znakem $\square$. Je to však korektní postup? Zadání přece říká, že se znaky a mají ve w *vypustit*, nikoliv nahradit speciálními znaky. Když si také připomeneme, jak je definován výstup výpočtu, uvědomíme si, že musíme nejen přemazávat a-čka, ale také zbylé symboly „srážet k sobě“.

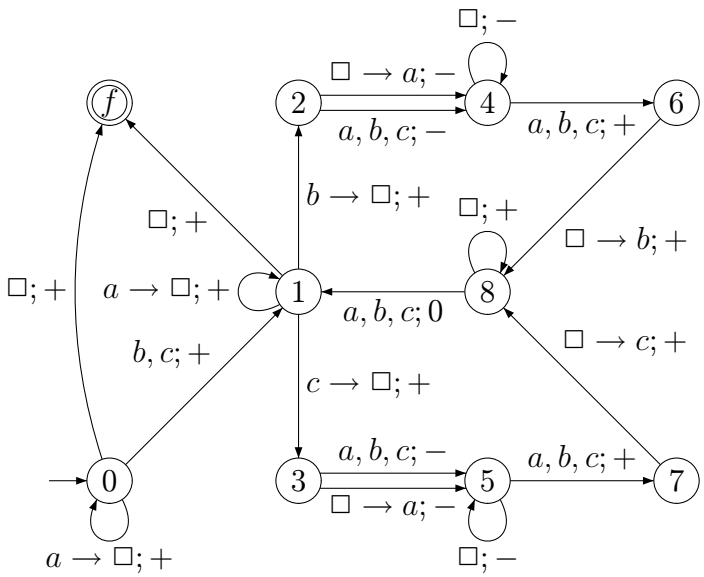
Při podrobnějším promyšlení navrhneme postup např. takto: Stroj maže všechna počáteční a-čka (jsou-li jaká); narazí-li přitom na $\square$, ukončí výpočet. Když narazí na znak b nebo c, ponechá jej (čímž získá první symbol pozdějšího výstupu), ale vpravo od něj se již chová jinak: zase sice maže úsek a-ček, přičemž naražení na $\square$ znamená konec výpočtu, ale narazí-li teď na (další) znak $x \in \{b, c\}$, přemaže ho, posune se doleva přes úsek znaků $\square$ a zapíše x za dosud vzniklý prefix výstupu. (Onen úsek znaků $\square$ může být i prázdný, pak se vlastně znak x přemaže a znovu zapíše na totéž místo.)

Když se teď ovšem stroj vydá doprava, naráží na znaky $\square$ a neví, zda-li jsou vpravo za nimi ještě jiné znaky. Toto lze ošetřit různými způsoby; jedna možnost je, že po přemazání $x \in \{b, c\}$, se stroj před putováním vlevo nejprve podívá na políčko napravo od původního x, a když je tam $\square$ (tedy x byl

posledním znakem vstupního slova), pak tam zapíše „neškodné“ a, a teprve pak se vydá doleva, aby zapsal x za dosud vytvořený prefix výstupu.

(Samozřejmě to jde i jinak. Např. místo zapisování a-čka si stroj prostě může zapamatovat v řídicí jednotce, že právě přenášený znak x byl poslední, a ušetřit si tak závěrečný kus výpočtu, který vlastně jen smaže přidané a-čko. Toto řešení by ovšem vyžadovalo zavedení dalších stavů a zvětšilo počet stavů navrhovaného Turingova stroje; my raději upřednostníme návrh menšího stroje před drobným zefektivněním výpočtu.)

Teď už tedy jsme schopni navržený postup přímočaře realizovat instrukcemi Turingova stroje; stroj zase zachytíme grafem.



S pohledem na graf ještě jednou popišme činnost stroje.

Na začátku jsou ve stavu 0 umazávány všechny znaky a. Po prvním výskytu znaku b či c stroj přejde do stavu 1, který je vlastně vstupem do hlavního pracovního cyklu. Při každém průchodu tímto cyklem z 1 zpět do 1 je přenesen jeden následující znak b (horní větev) či c (dolní větev) z původní pozice na novou (vlevo); případné předcházející znaky a jsou přemazány. Všimněme si také onoho „kouknutí se vpravo“ (ve stavu 2 či 4) a případné zapsání „pomocného“ a-čka.

Turingovy stroje řešící ANO/NE problémy

Dosud uvedené Turingovy stroje řešily obecné problémy (kde každému vstupu je přiřazen výstup z víceprvkové, typicky nekonečné, množiny). Turingovy stroje mohou samozřejmě řešit také ANO/NE problémy (u nichž je každému vstupu přiřazeno ANO nebo NE). Mohli bychom přitom např. vyžadovat, aby na pásce na závěr výpočtu zůstalo zapsáno jen ANO či NE (nebo 1 či 0). Obvyklejší (a technicky jednodušší) je ovšem požadovat, aby příslušný stroj měl koncové stavy q_{ANO} („přijmout“ [accept]) a q_{NE} („zamítnout“ [reject]); místo zapsání odpovědi ANO na pásku stroj ukončí výpočet vstupem do stavu q_{ANO}, místo zapsání odpovědi NE vstoupí do stavu q_{NE}.

Všimněme si, že takto přirozeně může definovat, co to znamená, že

Turingův stroj M rozpoznává (neboli rozhoduje) jazyk L.

Kontrolní otázka: Jak byste tedy pojem rozhodování jazyka Turingovým strojem definovali?

(Je to opravdu přímočaré: pro (každé) vstupní slovo w, které patří do L, stroj musí skončit svůj výpočet ve stavu q_{ANO}, pro vstupní slovo, které do L nepatří, musí výpočet skončit ve stavu q_{NE}.)

Otázky:

OTÁZKA 5.2: Jak velký úsek pásky může TS při svém výpočtu použít?

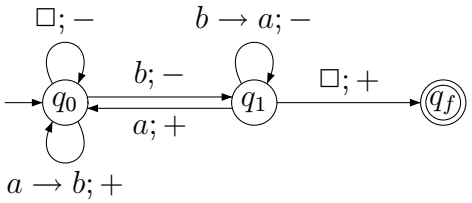
CVIČENÍ 5.3: Navrhněte TS, který zadané slovo nad abecedou $\{0, 1\}$ invertuje, tj. nuly přepíše na jedničky a naopak.

CVIČENÍ 5.4: Upravte v textu uvedený Turingův stroj pro zdvojování slov v abecedě $\{a\}$ tak, aby zdvojoval slova v abecedě $\{a, b\}$.

CVIČENÍ 5.5: Navrhněte Turingův stroj, který rozpoznává jazyk palindromů v abecedě $\{a, b\}$, tedy množinu slov $w \in \{a, b\}^*$, pro něž platí $w = w^R$ (kde w^R je zrcadlový obraz slova w).

CVIČENÍ 5.6: Popište, pro jaká vstupní slova z $\{a, b\}^*$ je výpočet následujícího Turingova stroje konečný a jak v tom případě vypadá výstup. (Stroj

samozřejmě začíná výpočet s hlavou na nejlevějším znaku vstupního slova; pokud je vstupní slovo ε, hlava čte $\Box$.)



Cvičení 5.7: Navrhněte TS, který číslo zadané ternárně nad abecedou $\{0, 1, 2\}$ vynásobí dvěma.

Cvičení 5.8: Navrhněte Turingův stroj se vstupní abecedou $\{a, b\}$ a páskovou abecedou $\{a, b, c, \Box\}$, jehož úkolem je následující.

Pro jakékoli vstupní slovo $w \in \{a, b\}^*$ musí výpočet skončit a po skončení musí být na pásce napsán výstup $\underbrace{c \ldots c}_{k}$, kde k je součet počtu výskytů podslova ab a počtu výskytů podslova ba ve w.

Návod: Zhruba řečeno, výpočet stroje musí ve slově w zjistit všechny „změny znaků“, z a na b i z b na a, a jejich počet „zapsat“ pomocí znaků c. Například pro aaa je výstup ε, pro $aaab$ je výstup c, pro $ababa$ je výstup $cccc$, pro $aabbbbaabbbba$ je výstup také $cccc$. Počet změn znaků si stroj samozřejmě nemůže pamatovat stavem řídicí jednotky, ale každou zjištěnou změnu hned „běží zapsat“ dalším c. Nezapomeňte, že původní vstup musí zmizet, tedy být přepsán znaky $\Box$.

Shrnutí: Tak už víme, co jsou Turingovy stroje, a zvládli jsme základy programovaní v tomto „programovacím jazyku velmi nízké úrovně“. Díky své jiné programátorské zkušenosti máme dobrou představu o tom, jak by se daly konstruovat takové stroje i pro složité algoritmické problémy (využitím běžných postupů jako je modulární přístup, definování podprogramů, apod.). Nebude pro nás tedy překvapením, když později ztotožníme pojmy „algoritmus“ a „Turingův stroj“.

5.3 Model RAM (Random Access Machine)

Orientační čas ke studiu této části: 3 hod.

Cíle této části:

V této části máte zvládnout jiný programovací jazyk „velmi nízké úrovně“ či vlastně „strojový jazyk“ jistého (abstraktního) počítače. Jedná se o model RAM (počítač s libovolným), které je realističtějším modelem počítačů než Turingovy stroje. Jde o pochopení instrukcí strojů RAM a jejich výpočtů. Máte porozumět tomu, že všechny algoritmy, které znáte, je možné implementovat také stroji RAM; jedná se tedy opět o univerzální výpočetní model.

Klíčová slova: *Stroje RAM, instrukce RAMu, výpočet stroje RAM, RAMy řešící problémy*

Již jsme hovořili o tom, že na Turingovy stroje se lze dívat jako na *univerální výpočetní model*; máme již dobrou představu o tom, jak by šel v principu každý algoritmus, který známe, implementovat vhodným Turingovým strojem. Teď se budeme věnovat jinému univerzálnímu výpočetnímu modelu, jenž budeme nazývat *stroje RAM*; jeho definice již byla ovlivněna tím, jak vypadají a jak pracují skutečné počítače (na rozdíl od Turingových strojů, které byly navrženy už v třicátých letech dvacátého století). Dá se říci, že se jedná o jednoduchou abstrakci reálného procesoru s jeho strojovým kódem, pracujícího s lineárně uspořádanou pamětí, tj. posloupností „buněk“ s adresami; jedná se o teoretický model, ve kterém se snažíme o maximální zjednodušení, a tak nerozlišujeme mezi nějakou „vnitřní“ a „vnější“ pamětí, atd. Název *stroje RAM* (Random Access Machine(s), v češtině někdy překládáno jako „počítač(e) s libovolným přístupem“) není zcela výstižný; nejedná se zde o nějakou náhodnost, je tím myšleno, že v jednom kroku může stroj přistoupit k buňce paměti s libovolnou adresou (na rozdíl od ryze sekvenčního přístupu u Turingových strojů, kde se v jednom kroku lze pohnout jen na sousední políčko pásky).

Poznámka: Kromě výrazů „stroj RAM“ či „RAM-stroj“ budeme užívat také jen zkratku „RAM“; budeme např. (trochu slangově) mluvit o sestrojení RAMu, porovnání dvou RAMů apod.

Do buněk paměti ukládá RAM celá čísla; jeho vstupy a výstupy jsou také posloupnosti celých čísel (uložených v buňkách vstupní a výstupní pásky). Dá se tedy říci, že RAMy mohou sloužit k řešení problémů, u nichž $IN, OUT \subseteq \mathbb{Z}^*$ ($\mathbb{Z}$ označuje množinu všech celých čísel, včetně záporných), tedy problémů, jejichž vstupy i výstupy jsou kódovány posloupnostmi celých čísel.

Poznámka: To je sice rozdíl od případu Turingových strojů, kde jsou vstupy i výstupy slova v jisté abecedě, ale z dřívější diskuse o různých kódováních vstupů a výstupů víme, že tento rozdíl je jen technického rázu a rozhodně není nijak zásadní.

Nebudeme uvádět ryze formální definici RAMů, podáme jen neformální popis, ze kterého by nicméně mělo být vše jasné. Konkrétní RAM se skládá z těchto částí (viz Obrázek 5.3):

- Programová jednotka, ve které je uložen program, tvořený konečnou posloupností instrukcí (příkazů), které budou popsány dále.

  > Jednotlivé RAMy se liší jen svým programem, dále popsaná „hardwarová“ struktura je u všech stejná. Proto konkrétní RAM můžeme ztotožnit s jeho programem, což často děláme.

- Neomezená pracovní paměť tvořená buňkami, kde každá buňka může obsahovat libovolné celé číslo. Buňky jsou očíslovány přirozenými čísly $0, 1, 2, \ldots;$. Číslo buňky se nazývá adresa buňky. Do buněk je možno zapisovat i z nich číst.

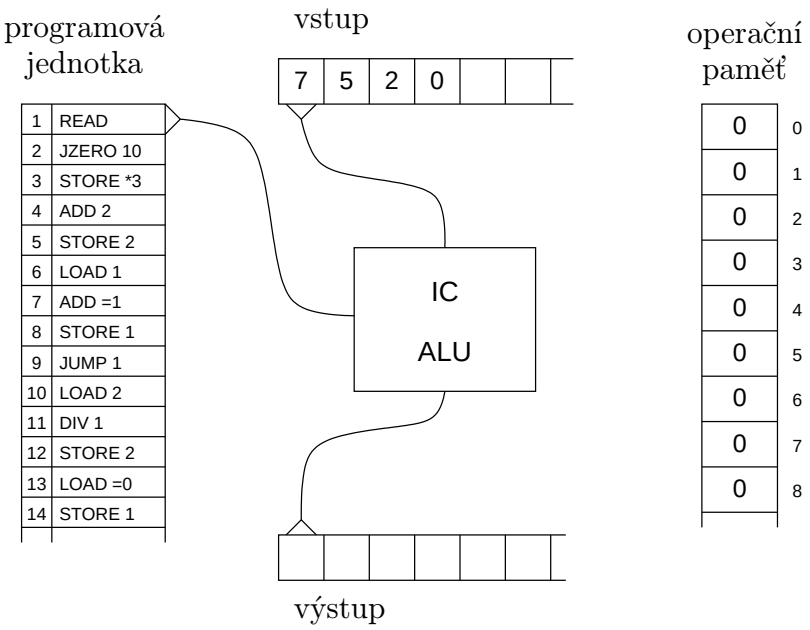
- Vstupní páska tvořena buňkami (políčky), kde každá buňka obsahuje jedno celé číslo. Z této pásky je možno pouze sekvenčně číst. Na aktuálním políčku stojí (čtecí) hlava. Základní krok v činnosti hlavy spočívá v přečtení obsahu snímaného políčka a posunutí doprava o jedno políčko.

- Výstupní páska, do jejíchž buněk se zapisují celá čísla. Na tuto pásku je pouze možné sekvenčně zapisovat (pomocí zapisovací hlavy).

- Centrální jednotka, obsahující mj. programový registr (instruction counter, IC) ukazující, která instrukce má být v daném okamžiku prováděna (programový registr prostě obsahuje pořadové číslo příslušné

instrukce). Tato instrukce se provede a programový registr se příslušně změní (např. se zvýší o 1 či se změní jinak v případě skoku. Instrukce mohou mj. předepisovat čtení ze vstupu či zapisování na výstup; centrální jednotka přistupuje na vstupní a výstupní pásku pomocí hlav zmíněných výše. Další instrukce mohou předepisovat vykonání nějakých aritmetických či logických operací (k tomu si lze představit Arithmetical-Logical-Unit ALU jako součást centrální jednotky) a práci s pamětí; k paměti se přistupuje přímo - na předepsanou (vypočtenou) adresu.

- Ještě uveďme, že buňky s adresou 0 a 1 mají zvláštní postavení: buňka 0 se nazývá *pracovní registr* (také *akumulátor*) a je „automaticky se účastní“ provádění téměř všech instrukcí; buňka 1 se nazývá *indexový registr* a slouží k dynamickému adresování, jak je vysvětleno dále.



Obrázek 5.3: Stroj RAM

V *počáteční konfiguraci* (tj. na začátku výpočtu) je na určitém počátečním úseku vstupní pásky uložen vstup: prvních n políček, pro nějaké n, obsahuje (vstupní) celá čísla $c_1, c_2, \ldots, c_n$; vstupní hlava snímá první buňku s číslem

c_1). Zbylá políčka vstupní pásky, všechna políčka výstupní pásky a všechny paměťové buňky obsahují číslo 0; programový registr ukazuje na první instrukci programu (tj. obsahuje číslo 1).

Poznámka: Na obrázku jsou vstupní čísla $7, 5, 2$. Mohlo by to ovšem být třeba $-3407, 29, 543865, -19$ apod. Všimněme si také, že podle definice RAM při přečtení vstupního čísla 0 nepozná, zda je toto číslo ještě součást vstupu nebo se už jedná o první buňku za vstupem. To je technická věc, která se snadno „programátorsky“ vyřeší (např. je každému vstupnímu číslu předřazeno informační jednobitové číslo 1). Teoretický model slouží k analýze algoritmů (zejména z hlediska složitosti, tj. časové či paměťové náročnosti, jak budeme diskutovat později) a nikoli pro skutečné praktické programování; proto definici nekomplikujeme technickými detaily jako je ten výše zmíněný.

Konfigurace (tj. stav výpočtu) se mění krok za krokem prováděním předepsaných instrukcí. Nyní uvedeme instrukce, z nichž lze sestavovat RAM-program. (Konkrétní program je např. na Obrázku 5.4.)

Tvary „operandů“ instrukcí a jejich příslušné hodnoty jsou patrny z následující tabulky (i je zápis přirozeného čísla). Za touto tabulkou pak již následuje přehled instrukcí, logicky rozdělených do několika skupin. (Označení *návěští* zde představuje přirozené číslo, udávající pořadové číslo instrukce, která bude prováděna jako následující, dojde-li ke skoku.)

Tvary operandů:

tvar	hodnota operandu
=i	přímo číslo udané zápisem i
i	číslo obsažené v buňce s adresou i
*i	číslo v buňce s adresou $i + j$, kde j je aktuální obsah indexového registru

Instrukce vstupu a výstupu (jsou bez operandu):

zápis	význam
`READ`	do pracovního registru se uloží číslo, které je v políčku snímaném vstupní hlavou, a vstupní hlava se posune o jedno políčko doprava
`WRITE`	výstupní hlava zapíše do snímaného políčka výstupní pásky obsah pracovního registru a posune se o jedno políčko doprava

Instrukce přesunu v paměti:

zápis	význam
`LOAD` *op*	do pracovního registru se načte hodnota operandu
`STORE` *op*	hodnota operandu se přepíše obsahem pracovního registru (zde se nepřipouští operand tvaru =i)

Instrukce aritmetických operací:

zápis	význam
`ADD` *op*	číslo v pracovním registru se zvýší o hodnotu operandu (tedy přičte se k němu hodnota operandu)
`SUB` *op*	od čísla v pracovním registru se odečte hodnota operandu
`MUL` *op*	číslo v pracovním registru se vynásobí hodnotou operandu
`DIV` *op*	číslo v pracovním registru se „celočíselně“ vydělí hodnotou operandu (do pracovního registru se uloží výsledek příslušného celočíselného dělení)

Instrukce skoku:

zápis	význam
`JUMP` *návěští*	výpočet bude pokračovat instrukcí určenou návěštím
`JZERO` *návěští*	je-li obsahem pracovního registru číslo 0, bude výpočet pokračovat instrukcí určenou návěštím; v opačném případě bude pokračovat následující instrukcí
`JGTZ` *návěští*	je-li číslo v pracovním registru kladné, bude výpočet pokračovat instrukcí určenou návěštím; v opačném případě bude pokračovat následující instrukcí

Instrukce zastavení:

zápis	význam
`HALT`	výpočet je ukončen („regulérně“ zastaven)

Jak lze očekávat, provedení instrukce zpravidla také znamená zvýšení programového čítače o jedničku (výpočet pokračuje prováděním bezprostředně

následující instrukce); výjimkou jsou případy, kdy dojde ke skoku (a také případ instrukce `HALT`).

Předpokládáme, že kdykoli by mělo při běhu dojít k nedefinované akci (dělení nulou, programový čítač ukazuje „mimo program“, adresa při použití operandu $*i$ vyjde záporná), výpočet se („neregulérně“) zastaví.

RAM M řeší problém $P = (IN, OUT, p)$, kde $IN, OUT \subseteq \mathbb{Z}^*$, jestliže má tuto vlastnost:
začne-li výpočet v počáteční konfiguraci se vstupem $c_1c_2 \dots c_n \in IN$, pak svůj výpočet (regulérně) skončí, přičemž na výstupu je $p(c_1c_2 \dots c_n)$.

```
 1  READ
 2  JZERO 10
 3  STORE *3
 4  ADD 2
 5  STORE 2
 6  LOAD 1
 7  ADD =1
 8  STORE 1
 9  JUMP 1
10  LOAD 2
11  DIV 1
12  STORE 2
13  LOAD =0
14  STORE 1
15  LOAD *3
16  JZERO 23
17  SUB 2
18  WRITE
19  LOAD 1
20  ADD =1
21  STORE 1
22  JUMP 15
23  HALT
```

Obrázek 5.4: Příklad programu pro stroj RAM

Na Obrázku 5.4 je příklad programu pro stroj RAM, který řeší následující problém:

VSTUP: Neprázdná posloupnost kladných celých čísel ukončená nulou.

VÝSTUP: Odchylky jednotlivých čísel od aritmetického průměru zadané posloupnosti zaokrouhleného dolů.

Upozornění na animaci.
Na web-stránce předmětu najdete i animaci výpočtu tohoto RAMu.

Další příklad programu pro stroj RAM najdete v kapitole 7.

Poznámka: Jak už jsme zmínili, stroje RAM nejsou určeny pro skutečné praktické programování a ani při teoretických analýzách algoritmů většinou neprogramujeme RAMy do detailu. Vystačíme obvykle se strukturovaným popisem algoritmu (jako ve vyšších programovacích jazycích), z něhož by se implementace konkrétními instrukcemi RAMu dala přímočaře („mechanicky“) sestavit. Ještě se k tomu vrátíme při zkoumání výpočetní složitosti.

CVIČENÍ 5.9: Navrhněte způsob, jak byste na stroji RAM implementovali jednorozměrné pole o délce k.

CVIČENÍ 5.10: Navrhněte způsob, jak byste na stroji RAM implementovali dvourozměrné pole $k \times \ell$.

CVIČENÍ 5.11: Navrhněte způsob, jak byste na stroji RAM implementovali vykonávání rekurzivních podprogramů, tj. takových, které mohou mnohokrát rekurzivně volat samy sebe.

Shrnutí: Už tedy známe stroje RAM a alespoň trochu jsme si je „osahali“ z programátorského hlediska. Je nám jasné, jak bychom programy ve svém oblíbeném programovacím jazyce implementovali v tomto „primitivním programovacím jazyku“, kdyby to (nedejbože) bylo nutné.

5.4 Simulace mezi výpočetními modely; Churchova-Turingova teze

Orientační čas ke studiu této části: 1 hod.

Cíle této části:

V této části máte přinejmenším na neformální úrovni pochopit, co se myslí simulací mezi výpočetními modely. Speciálně si máte promyslet vzájemnou simulaci Turingových strojů a strojů RAM. Dále se máte zamyslet nad Churchovou-Turingovou tezí, tj. uvědomit si, na čem zakládáme naše ztotožňování pojmu „algoritmus“ a „Turingův stroj“ (či „RAM“ nebo „program v běžném programovacím jazyku“).

Klíčová slova: *simulace mezi různými variantami Turingových strojů, vícepáskový Turingových stroj, Turingův stroj s jednostranně nekonečnou páskou, Churchova-Turingova teze (ztotožnění pojmu „algoritmus“ a „Turingův stroj“)*

Již jsme vícekrát připomněli to, co intuitivně cítíme (opírajíce se o programátorskou zkušenost): jak Turingovy stroje tak RAMy umějí simulovat (implementovat) programy v jakémkoli běžném programovacím jazyku (a nic jiného). Nebudeme formálně definovat, co to znamená, že jeden prostředek (neboli program) simuluje jiný. Spolehneme se na „programátorské porozumění“, které si potvrdíme na následujícím případu.

Vícepáskovým Turingovým strojem míníme model, který je definován obdobně jako Turingův stroj, ale má k pásek ($k > 1$) se samostatně řízenými hlavami. Přechodová funkce bere ohled na symboly čtené hlavami ze všech pásek a určuje pohyb každé hlavy (na její pásce) zvlášť. Např instrukce 2-páskového stroje je tak typu

$$(q, a_1, a_2) \to (q', a_1', m_1, a_2', m_2) \quad (\text{kde } m_1, m_2 \in \{-1, 0, +1\})$$

a znamená: je-li řídicí jednotka ve stavu q, hlava na první pásce čte a_1 a hlava na druhé pásce čte a_2, tak řídicí jednotka přejde do stavu q', hlava na první pásce přepíše čtený symbol symbolem a_1' a pohne se podle hodnoty m_1

a hlava na druhé pásce přepíše čtený symbol symbolem a'_2 a pohne se podle hodnoty m_2.

Upozornění na animaci.
Mezi animacemi na web-stránce najdete příklad konkrétního dvoupáskového stroje.

Model vícepáskových Turingových strojů je sice obecnější než (normální jednopáskové) Turingovy stroje, ale mělo by být zřejmé, že každý k-páskový stroj je možné implementovat jednopáskovým:

Tvrzení 5.3
Každý k-páskový Turingův stroj lze simulovat standardním (jednopáskovým) Turingovým strojem.

Nebudeme to formalizovat a dokazovat, ale zase odkážeme na animaci.

Upozornění na animaci.
Na web-stránce najdete animaci ilustrující simulaci konkrétního dvoupáskového stroje jednopáskovým. Zobecnění pro jakýkoli k-páskový stroj je přímočaré.

V literatuře se často jako základní model bere varianta Turingových strojů s jen *jednostranně nekonečnou páskou*: představme si, že před vstupním slovem je zapsán speciální symbol označující začátek pásky, který nemůže být přepsán a z nějž hlava nemůže nikdy jít doleva.

Není těžké ukázat následující tvrzení.

Tvrzení 5.4
Každý Turingův stroj M s oboustranně nekonečnou páskou lze simulovat Turingovým strojem M' s jednostranně nekonečnou páskou.

Důkaz (náznak): V animaci ilustrující simulaci dvoupáskového stroje jednopáskovým se uplatnila idea vícestopé pásky. Zde si u M' představme dvoustopou pásku: když M' simuluje výpočet M v rámci vstupního slova či napravo od něj, pracuje v horní stopě; když M přechází vlevo od původního vstupního slova, M' začne pracovat v dolní stopě, kde se pohybuje opačně než M (když se hlava M pohne doleva, M' to simuluje pohybem doprava a naopak). □

Využitím posledního tvrzení také snadno odvodíme:

Tvrzení 5.5
Každý Turingův stroj lze simulovat strojem RAM.

Kontrolní otázka: Dokážete naznačit důkaz předešlého tvrzení?

(Jednostranně nekonečnou pásku přímočaře simulujeme paměťovými buňkami $2, 3, \ldots$, pozici hlavy pak hodnotou v indexregistru. Seznam instrukcí Turingova stroje přímočaře přepíšeme instrukcemi RAMu; stavy Turingova stroje jsou reprezentovány vybranými návěštími instrukcí.)

Technicky komplikovanější je ukázat opačný směr:

Tvrzení 5.6
Každý RAM lze simulovat Turingovým strojem.

Důkaz (náznak):

Upozornění na animaci.
Na web-stránce najdete detailní animaci konkrétního RAMu vícepáskovým Turingovým strojem.

Z uvedené animace lze jistě pochopit princip simulace jakéhokoli RAMu vícepáskovým Turingovým strojem; ten pak můžeme simulovat standardním jednopáskovým Turingovým strojem, jak jsme již diskutovali výše. □

Je nám tedy naprosto jasné, že Turingovy stroje a stroje RAM jsou ekvivalentní v tom smyslu, že pokud je nějaký problém řešitelný jedním z těchto modelů, pak je řešitelný i druhým.

Podobně se dá ukázat ekvivalence těchto dvou modelů s celou řadou dalších modelů ať už teoretických (jako jsou například λ-kalkulus nebo tzv. rekurzivní funkce, kterými se zde ovšem nebudeme dále zabývat) či praktických (všechny obecné programovací jazyky).

Doposud u každého modelu, který byl navržen jako formalizace pojmu „algoritmus“, se dá ukázat, že je ekvivalentní Turingovým strojům (umí řešit tytéž problémy). Toto ospravedlňuje obecné přesvědčení, že Turingův stroj (či libovolný model, který je s ním ekvivalentní) je vhodnou formalizací pojmu algoritmus. Předpokládá se, že není možné navrhnout žádný model, který by odpovídal intuitivní představě pojmu algoritmus a přitom by jej nebylo možno simulovat Turingovým strojem.

Toto přesvědčení je formulováno jako tzv. *Churchova-Turingova teze:*

> *Ke každému algoritmu je možné zkonstruovat s ním ekvivalentní Turingův stroj (s rozumným kódováním vstupů a výstupů řetězci v určité abecedě); ekvivalencí zde rozumíme podmínku, že algoritmus i Turingův stroj vydají pro tytéž vstupy tytéž výstupy.*

Bylo by možné argumentovat, že naše intuitivní chápání pojmu „algoritmus“ vyžaduje, že každý algoritmus by se měl pro každý (přípustný) vstup zastavit, tedy jeho běh by neměl být nekonečný. Pak by nebyla pravda, že každý Turingův stroj je algoritmem. Ovšem např. operační systém počítače je zajisté také algoritmem, byť neřeší nějaký problém v námi definovaném smyslu; operační systém přitom v principu může běžet donekonečna (pokud si „na vstupu nepřečte“ pokyn k zastavení). Když tedy podmínku zastavení na algoritmus neklademe, je očividné, že každý Turingův stroj je algoritmem. Proto se Churchova-Turingova teze také stručně vyjadřuje takto:

Algoritmus = Turingův stroj.

Churchova-Turingova teze není věta v matematickém slova smyslu, kterou by bylo možno dokázat. (To souvisí s naší dřívější diskusí, že „algoritmus“ je základním pojmem, nikoli odvozeným, který by byl definován pomocí jiných pojmů.) V praxi je *Churchova-Turingova teze* přijímána jako definice pojmu „algoritmus“, tedy jako *axiom*.

Poznámka: My budeme Churchovu-Turingovu tezi (tiše) využívat při důkazu algoritmické nerozhodnutelnosti problémů, jak o tom pojednáme dále.

Otázky:

Otázka 5.12: Jak byste simulovali Turingův stroj s k-prvkovou páskovou abecedou Turingovým strojem s páskovou abecedou $\{0, 1, \square\}$?

Shrnutí: Díky naší programátorské zkušenosti a promyšlením uvedených konkrétních případů je nám jasné, co je to simulace mezi výpočetními modely. Promysleli jsme si, že pojem „algoritmus“ můžeme oprávněně ztotožnit s pojmem „Turingův stroj“ (či „RAM“ nebo „program v běžném programovacím jazyku“).

5.5 Rozhodnutelnost a nerozhodnutelnost

Orientační čas ke studiu této části: 1 hod.

Cíle této části:

Máte se seznámit s některými algoritmicky nerozhodnutelnými problémy a udělat si představu o způsobu důkazu nerozhodnutelnosti. Speciálně máte pochopit pojem algoritmické převeditelnosti mezi problémy.

Klíčová slova: *rozhodnutelný problém, nerozhodnutelný problém, problém zastavení (Turingova stroje), doplňkový problém, (algoritmická) převeditelnost mezi problémy*

Již jsme zmiňovali, že existují problémy, které algoritmicky řešitelné, či algoritmicky rozhodnutelné (v případě ANO/NE problémů), nejsou. Jako příklad jsme uvedli problém

NÁZEV: Eq-CFG (*Ekvivalence bezkontextových gramatik*)

VSTUP: Dvě bezkontextové gramatiky G_1, G_2.

OTÁZKA: Jsou G_1, G_2 ekvivalentní (tj. platí $L(G_1) = L(G_2)$)?

Připomeňme si ještě definici algoritmické řešitelnosti a rozhodnutelnosti, v níž se implicitně (tedy „tiše“) předpokládá, že vstupy a výstupy jsou kódovány např. řetězci symbolů z nějaké konečné abecedy, či jiným podobným způsobem.

Definice 5.7
Problém $P = (IN, OUT, p)$ $(p : IN \rightarrow OUT)$ je *algoritmicky řešitelný*, jestliže existuje algoritmus, který pro libovolný vstup $w \in IN$ svůj výpočet skončí a vydá jako výsledek $p(w)$. V případě ANO/NE problému používáme místo „algoritmicky řešitelný“ raději pojem *algoritmicky rozhodnutelný*, či stručněji *rozhodnutelný*.

Dokážeme si teď nerozhodnutelnost konkrétního problému (použitím tzv. *metody diagonalizace*). Vstupem tohoto problému je jakýkoli Turingův stroj M

se vstupní abecedou $\{0,1\}$ a páskovou abecedou $\{0,1,\Box\}$.
(Můžeme si připomenout, že Turingovy stroje s páskovou abecedou $\{0,1,\Box\}$ de facto reprezentují všechny Turingovy stroje; již jsme se totiž zamýšleli nad tím, že stroj s libovolnou abecedou je možné přímočaře simulovat strojem s abecedou $\{0,1,\Box\}$.)
Říkáme-li, že vstupem je Turingův stroj M, rozumíme tím samozřejmě, že M je reprezentován dohodnutým kódem $Kod(M)$.

Kontrolní otázka: Umíte navrhnout nějaký přímočarý způsob, jak lze konkrétní Turingův stroj zapsat slovem v nějaké abecedě? A postačí vám k tomuto kódování abeceda $\{0,1\}$?

(Jistě jste si vzpomněli, že konkrétní Turingovy stroje jsme zadávali de facto seznamem instrukcí. Takový seznam instrukcí ovšem je vlastně řetězcem symbolů v pevně zvolené abecedě; např. konkrétní stavy jsou reprezentovány řetězci $q1, q28, q608$ apod.
A je nám navíc jasné, že symboly používané abecedy můžeme zakódovat dostatečně dlouhými bloky bitů (tedy řetězci symbolů 0 a 1 s pevnou délkou), takže je možné navrhnout přímočaré kódování tak, že $Kod(M) \in \{0,1\}^*$ pro každý Turingův stroj M.)

Takže dále budeme předpokládat, že máme dohodnuto kódování Turingových strojů (s páskovou abecedou $\{0,1,\Box\}$) pomocí řetězců z $\{0,1\}^*$.
Teď už specifikujme avizovaný problém:

NÁZEV: DHP (*Diagonální problém zastavení [Diagonal Halting Problem]*)

VSTUP: $Kod(M) \in \{0,1\}^*$, kde M je Turingův stroj se vstupní abecedou $\{0,1\}$ a páskovou abecedou $\{0,1,\Box\}$.

OTÁZKA: Zastaví se M na svůj kód? (Tedy je výpočet stroje M pro vstupní slovo $Kod(M)$ konečný?)

Poznámka: Slovo „diagonální“ odkazuje na zmíněnou metodu diagonalizace. Ta se připisuje Cantorovi a dá se mj. použít pro důkaz faktu, že množina reálných čísel má větší mohutnost než množina přirozených čísel.

DHP je jistě korektně definovaný problém: pro každý vstupní stroj M je $Kod(M)$ povoleným vstupem pro M (neboť je to řetězec v jeho vstupní abecedě $\{0,1\}$) a pro tento vstup je výpočet stroje M buď konečný (v tom

případě je odpověď na otázku ANO) nebo nekonečný (v tom případě je odpověď na otázku NE).
Můžeme se tedy zabývat otázkou, zda problém DHP je rozhodnutelný. Podívejme se teď na následující tvrzení a větu.

Tvrzení 5.8
Neexistuje Turingův stroj, který by rozhodoval problém DHP.

Věta 5.9
Problém DHP je algoritmicky nerozhodnutelný (stručněji: nerozhodnutelný).

Za chvíli se dostaneme k důkazu Tvrzení. Teď si uvědomme, že Věta plyne z Tvrzení za předpokladu přijetí Churchovy-Turingovy teze (která říká, že každý algoritmus lze implementovat Turingovým strojem).

Poznámka: Je běžnou praxí, že když pro konkrétní problém P dokážeme neexistenci Turingova stroje, který by jej rozhodoval, řekneme, že jsme dokázali, že P je nerozhodnutelný (tedy *algoritmicky* nerozhodnutelný). Přitom už nedodáváme, že to vyvozujeme z Churchovy-Turingovy teze, protože to se bere za samozřejmé. My budeme také tuto praxi používat, ale budeme si vědomi, že v tvrzeních o algoritmické nerozhodnutelnosti (či neřešitelnosti) je vždy skryt onen předpoklad platnosti Churchovy-Turingovy teze.

Jak se tedy dá dokázat Tvrzení 5.8? Předpoklad existence Turingova stroje rozhodujícího DHP se jednoduchým, ale chytrým, „trikem“ přivede k logickému sporu.

Komentář: Myšlenka následujícího důkazu připomíná tuto logickou hádanku: V malém městečku žije holič, který holí právě všechny ty muže z městečka, kteří se neholí sami. Holí se onen holič nebo ne?
(Odpověď samozřejmě nemůže být ani ANO ani NE; není prostě možné, aby takový holič existoval [když mezi muže z městečka řadíme i jeho].)

Důkaz: (Tvrzení 5.8). (Pro hloubavější čtenáře.)
Předpokládejme, že existuje Turingův stroj D rozhodující DHP; stroj D se tedy pro každý vstup $Kod(M) \in \{0,1\}^*$ zastaví, a sice ve stavu q_{ANO}, jestliže výpočet stroje M pro vstup $Kod(M)$ je konečný, a ve stavu q_{NE}, jestliže výpočet stroje M pro vstup $Kod(M)$ je nekonečný.
Uvažujme teď stroj D', který vznikne z D touto změnou: v D' použijeme místo q_{ANO} stav q_L (L jako „loop“, neboli „smyčka“), pro nějž dodáme instrukční schéma

$(q_L, a) \to (q_L, a, 0)$, kde $a \in \{0, 1, \Box\}$.

Tedy místo přechodu do koncového stavu q_{ANO} skočí stroj D' do nekonečného cyklu.
Stroj D' má vstupní abecedu $\{0, 1\}$ a můžeme navíc předpokládat, že jeho pásková abeceda je $\{0, 1, \Box\}$. (Jak už jsme vícekrát diskutovali, další páskové symboly mohou být kódovány bloky bitů dostatečné délky.)
Stroj D' má pochopitelně také svůj kód, a sice slovo $Kod(D') \in \{0, 1\}^*$.
Zeptejme se teď: zastaví se D' pro vstupní slovo $Kod(D')$?
Zjistíme, že odpověď nemůže být ani ANO ani NE.

Kontrolní otázka: Proč není možné, aby byl výpočet D' na slově $Kod(D')$ konečný? A proč není možné, aby byl výpočet D' na slově $Kod(D')$ nekonečný?

(V prvním případě by D pro vstup $Kod(D')$ skončil ve stavu q_{ANO}, tedy D' by pro vstup $Kod(D')$ prováděl nekonečný cyklus (což je spor). V druhém případě by D pro vstup $Kod(D')$ skončil ve stavu q_{NE}, tedy i výpočet D' by pro vstup $Kod(D')$ skončil (což je spor). [Výpočet stroje D' pro konkrétní vstup, v našem případě pro $Kod(D')$, nemůže samozřejmě být zároveň konečný i nekonečný.])

Takže nezbývá než vyvodit, že stroj D' vůbec neexistuje! V tom případě ovšem nemůže existovat ani stroj D rozhodující problém DHP, protože z D bychom snadno vyrobili onen D', který nemůže existovat. (Tím jsme dokončili důkaz Tvrzení 5.8.) □

Měli bychom teď poznamenat, že když se v informatice (speciálně v teorii algoritmů) řekne „základní nerozhodnutelný problém“, myslí se tím obvykle trochu jiný problém než DHP, a sice:

NÁZEV: HP (*Problém zastavení [Halting Problem]*)

VSTUP: Turingův stroj M a jeho vstup w.

OTÁZKA: Zastaví se M na w (tzn. je výpočet stroj M pro vstupní slovo w konečný)?

Samozřejmě se zase automaticky předpokládá nějaké rozumné kódování stroje M atd. Problém HP lze samozřejmě také formulovat pro RAMy, či

programy ve zvoleném programovacím jazyku apod; již víme, že takto formulované problémy jsou v zásadě ekvivalentní našemu problému HP pro Turingovy stroje.

Chceme-li dokázat nerozhodnutelnost problému HP, naskýtá se přirozeně myšlenka, že bychom nějak mohli využít již dokázané nerozhodnutelnosti problému DHP.
Můžeme uvažovat takto: Kdybychom měli algoritmus (tj. Turingův stroj, protože se tiše odkazujeme na Churchovu-Turingovu tezi) A_{HP} rozhodující problém HP, tak bychom pomocí něj snadno sestavili algoritmus A_{DHP} rozhodující DHP.

Kontrolní otázka: Jak by pracoval onen A_{DHP}, když by na vstup dostal $Kod(M)$?

(Jistě vás hned napadlo, že by A_{DHP} zavolal (jako podprogram, či proceduru, chcete-li) A_{HP}, kterému by předal $Kod(M)$ jakožto kód stroje a (totéž) slovo $Kod(M)$ jako vstupní slovo. A_{HP} by podle předpokladu skončil. Když by jako odpověď vydal (svému volajícímu programu A_{DHP}), že obdržený stroj se zastaví na obdržený vstup, tedy že M se zastaví na $Kod(M)$, tak A_{DHP} by také vydal ANO; když by A_{HP} vydal odpověď NE, tak by i A_{DHP} vydal NE.)

Jelikož ovšem víme, že žádný algoritmus A_{DHP} rozhodující DHP neexistuje, nemůže existovat ani algoritmus A_{HP} rozhodující problém HP; problém HP je tedy také nerozhodnutelný.

Dá se vcelku přirozeně říci, že jsme „(algoritmicky) převedli problém DHP na problém HP“, což nám pomohlo z nerozhodnutelnosti problému DHP vyvodit nerozhodnutelnost problému HP.
Tyto pojmy a příslušná pozorování si teď zformulujeme přesně.

Poznámka: Určitou modifikaci využijeme také v teorii složitosti, speciálně u tzv. NP-obtížnosti problémů.

Definice 5.10
Mějme ANO/NE problémy P_1, P_2. Řekneme, že problém P_1 je *algoritmicky převeditelný*, stručněji *převeditelný*, na problém P_2, což označujeme

$$P_1 \rightsquigarrow P_2,$$

jestliže existuje (převádějící) algoritmus A, který pro libovolný vstup w problému P_1 sestrojí (tzn. skončí svůj výpočet a jako výstup vydá) vstup problému P_2, označme jej $A(w)$, přičemž platí, že odpověď na otázku problému P_1 pro vstup w je ANO právě tehdy, když odpověď na otázku problému P_2 pro vstup $A(w)$ je ANO.

Samozřejmě si pojem „algoritmus“ zase můžeme nahradit pojmem „Turingův stroj“. Schématicky můžeme tedy převeditelnost $P_1 \rightsquigarrow P_2$ zachytit takto:

existuje algoritmus (TS) A tak, že

- (vstup $P1$) $w \longrightarrow \boxed{A} \longrightarrow A(w)$ (vstup $P2$)
- odpověď pro w v $P1$ $=$ odpověď pro $A(w)$ v $P2$

Již jsme tedy ukázali, že DHP$\rightsquigarrow$HP.

Kontrolní otázka: Co konkrétně dělá algoritmus, převádějící problém DHP na problém HP ?

(Velmi jednoduchou věc: vydá dvě kopie svého vstupu; tedy pro vstupní u vydá u, u.)

Rovněž jsme již de facto ukázali následující jednoduché tvrzení.

Tvrzení 5.11
Je-li $P_1 \rightsquigarrow P_2$ a problém P_1 je nerozhodnutelný, je i problém P_2 nerozhodnutelný.
(Jinými slovy: Je-li $P_1 \rightsquigarrow P_2$ a problém P_2 je rozhodnutelný, je i problém P_1 rozhodnutelný.)

Vyslovme teď jako větu náš závěr o problému HP (tj. o onom „základním nerozhodnutelném problému“)

Věta 5.12
Problém HP je nerozhodnutelný.

Nerozhodnutelnost dalších problémů se typicky ukazuje pomocí převeditelnosti z HP, resp. pomocí série převeditelností (také se říká *redukcí*) mezi problémy. Stojí za to explicitně upozornit na související očividné tvrzení.

Tvrzení 5.13
Je-li $P_1 \rightsquigarrow P_2$ a $P_2 \rightsquigarrow P_3$ (stručněji psáno $P_1 \rightsquigarrow P_2 \rightsquigarrow P_3$), tak $P_1 \rightsquigarrow P_3$.
(Tedy relace převeditelnosti mezi problémy je tranzitivní.)

Rovněž je v této souvislosti užitečný pojem doplňkového (neboli „opačného“) problému:

Definice 5.14
Mějme ANO/NE problém P. *Doplňkový problém* k problému P, označovaný $\overline{P}$, je problém, který má stejné vstupy jako P, ale výstupy ANO, NE jsou prohozeny (kde má P odpověď ANO, má $\overline{P}$ odpověď NE, a naopak).

Např. k problému HP je doplňkovým problémem $\overline{\text{HP}}$ s otázkou „Nezastaví se M na w?“ či „Je výpočet M nad vstupem w nekonečný?“, apod. (Samozřejmě není podstatná konkrétní formulace otázky, ale je podstatné to, kterým vstupům otázka přiřazuje ANO a kterým NE.)
Dále např. problém $\overline{\text{Eq-CFG}}$, označený také Non-Eq-CFG, zformulujeme třeba takto:

NÁZEV: Non-Eq-CFG (*Neekvivalence bezkontextových gramatik*)

VSTUP: Dvě bezkontextové gramatiky G_1, G_2.

OTÁZKA: Existuje slovo, které patří do jednoho z jazyků $L(G_1)$, $L(G_2)$ a nepatří do druhého? (Tedy: existuje $w \in (L(G_1) - L(G_2)) \cup (L(G_2) - L(G_1))$?

Všimněme si také triviálního faktu:

Tvrzení 5.15
Je-li problém P rozhodnutelný, je i $\overline{P}$ rozhodnutelný.
(Jinak řečeno: je-li problém P nerozhodnutelný, je i $\overline{P}$ nerozhodnutelný.)

Tedy pro každý problém P jsou buď oba problémy $P, \overline{P}$ rozhodnutelné, nebo jsou oba nerozhodnutelné. V důkazech nerozhodnutelnosti můžeme tedy využívat i doplňkových problémů. Např. z následujícího faktu plyne, že (nejen Non-Eq-CFG, ale také) Eq-CFG je nerozhodnutelný.

Fakt 5.16
HP$\rightsquigarrow$Non-Eq-CFG.

Kontrolní otázka: Umíte rychle zformulovat, co dělá algoritmus převádějící HP na Non-Eq-CFG?

(Ke vstupu M, w (TS a jeho vstup) sestrojí bezkontextové gramatiky G_1, G_2, přičemž platí: jestliže se M zastaví na w, tak $L(G_1) \neq L(G_2)$, a jestliže se M nezastaví na w, tak $L(G_1) = L(G_2)$.)

Uvedený fakt zde nedokazujeme, jen si řekneme, že jednou z možností, jak realizovat tento převod, je jít přes tzv. *Postův korespondenční problém (PKP)*; tedy HP$\rightsquigarrow$PKP$\rightsquigarrow$Non-Eq-CFG.

Poznámka: Více o tom pojednáme v rozšiřující části, kde také zkoumáme tzv. *částečnou rozhodnutelnost problémů*.

Shrnutí: Teď už máme plastičtější představu o tom, že některé problémy jsou (algoritmicky) nerozhodnutelné (včetně konkrétních praktických problémů jako je např. ekvivalence bezkontextových gramatik). Víme, že základem důkazů nerozhodnutelnosti je „chytré odvození logického sporu“ pro problém zastavení (či pro jeho diagonální verzi). Dále je nám jasné, co je to doplňkový problém k danému problému, a hlavně rozumíme pojmu „(algoritmická) převeditelnost“ mezi problémy. Chápeme, jak jí lze použít při důkazech nerozhodnutelnosti problémů využívajících již dokázané nerozhodnutelnosti jiných problémů.

Kapitola 6

Úvod do teorie vyčíslitelnosti - rozšiřující část

Orientační čas ke studiu této části: 3 hod.

Cíle kapitoly:

V této kapitole si máte ujasnit, co to je částečná rozhodnutelnost problémů, a uvědomit si, že např. problémy HP a Non-Eq-CFG jsou částečně rozhodnutelné; rovněž máte pochopit Postovu větu osvětlující vztah rozhodnutelnosti a částečné rozhodnutelnosti. Máte pochopit pojem „univerzální Turingův stroj“. Dále máte porozumět Riceově větě o nerozhodnutelnosti netriviálních vstupně/výstupních vlastností programů (či Turingových strojů) a máte se ji naučit aplikovat na konkrétní případy.

Klíčová slova: *částečně rozhodnutelné problémy, Postova věta, univerzální algoritmus (univerzální Turingův stroj), vstupně/výstupní vlastnosti programů, Riceova věta*

Částečná rozhodnutelnost

Připomeňme si, že když je problém P nerozhodnutelný, je nerozhodnutelný i jeho doplňkový problém $\overline{P}$.

Kontrolní otázka: Co je doplňkovým problémem problému $\overline{P}$?

(Samozřejmě je to problém P. Když tedy je $\overline{P}$ nerozhodnutelný, je nerozhodnutelný i P.)

Tedy např. HP i $\overline{\text{HP}}$ jsou nerozhodnutelné. Všimněme si ale jistého rozdílu mezi nimi. Když bychom dostali Turingův stroj (či počítačový program) M a jeho vstup w a měli zjistit, zda výpočet M na w je konečný, tak nás hned napadne M „spustit“ na w (tedy začít provádět [sami] či nechat provádět [počítači] výpočet M pro vstup w); když se dočkáme zastavení, budeme znát správnou odpověď Ano (v případě problému HP). Pokud M stále běží (a zabírá stále více a více paměti), nemůžeme si být stále jisti, zda jeho výpočet nakonec skončí nebo ne. U problému $\overline{\text{HP}}$ tedy tímto způsobem nemáme šanci zjistit kladnou odpověď.

Kontrolní otázka: U problémů Eq-CFG a Non-Eq-CFG je situace podobná. Pro který z nich vás napadne algoritmus, u nějž se máme možnost dočkat správné odpovědi Ano? (Alespoň si uváženě tipněte.)

Je to u Non-Eq-CFG. Není těžké navrhnout algoritmus, který pro dané slovo w a bezkontextovou gramatiku G rozhodne, zda $w \in L(G)$. (Tento problém diskutujeme v jiné části tohoto textu.) S využitím takového algoritmu (jako „podprogramu“) můžeme tedy navrhnout algoritmus, který pro zadané bezkontextové gramatiky G_1, G_2 systematicky generuje všechna slova $w_0, w_1, w_2, \ldots$ (v jejich společné terminální abecedě), přičemž pro každé w_i prověřuje, zda $w_i \in L(G_1) - L(G_2)$ či $w_i \in L(G_2) - L(G_1)$. V kladném případě algoritmus skončí, s odpovědí Ano na otázku problému Non-Eq-CFG.

Tyto úvahy přirozeně vedou k následující definici.

Definice 6.1
ANO/NE problém P je *částečně rozhodnutelný*, jestliže existuje algoritmus A, který pro každý vstup problému P, na něž je odpověď ANO, skončí a vydá (správnou) odpověď Ano a pro každý vstup problému P, na něž je odpověď NE, buď skončí se (správnou) odpovědí Ne nebo je jeho běh nekonečný.
Říkáme také, že algoritmus A *částečně rozhoduje* problém P.

Kontrolní otázka: Je každý rozhodnutelný problém také částečně rozhodnutelný?

Samozřejmě. Vidíme, že algoritmus A, který rozhoduje problém P, podle definice také částečně rozhoduje P. (Možnost nekonečného výpočtu pro vstupy

s odpovědí Ne algoritmus A prostě vůbec nevyužívá.)

Existují ovšem problémy, které jsou částečně rozhodnutelné, ale nejsou rozhodnutelné. De facto jsme již ukázali, že příklady takových problémů jsou HP a Non-Eq-CFG. Ale co např. $\overline{\text{HP}}$ a Eq-CFG? Sice nás pro žádný z nich nenapadl algoritmus, který by příslušný problém částečně rozhodoval, ale třeba by nás napadl při hlubším zamyšlení, nebo ne? Tuto otázku snadno zodpovíme, když si uvědomíme následující větu.

Věta 6.2 (Post)
ANO/NE problém P je rozhodnutelný právě tehdy, když P i $\overline{P}$ jsou částečně rozhodnutelné.

Důkaz: Když je P rozhodnutelný, pak je i $\overline{P}$ rozhodnutelný; pak ovšem P i $\overline{P}$ jsou částečně rozhodnutelné (jak jsme již diskutovali dříve).

Důkaz opačné implikace je o něco méně triviální, ale je také jednoduchý; hlavní myšlenka spočívá v tom, že k dvěma algoritmům (Turingovým strojům) lze zkonstruovat algoritmus (Turingův stroj), který výpočty obou provádí „paralelně“, tj. „střídavě sekvenčně“. Když takto „současně spustíme“ algoritmus A_1, který částečně rozhoduje P, a algoritmus A_2, který částečně rozhoduje $\overline{P}$, zaručeně dojde k situaci, kdy jeden z algoritmů skončí; když skončí A_1 s odpovědí Ano (nebo A_2 s odpovědí Ne), kombinovaný algoritmus skončí s odpovědí Ano, když skončí A_2 s odpovědí Ano (nebo A_1 s odpovědí Ne), kombinovaný algoritmus skončí s odpovědí Ne. □

Z Postovy věty tedy hned odvodíme, že problémy $\overline{\text{HP}}$ ani Eq-CFG nejsou částečně rozhodnutelné. (Protože víme, že jsou nerozhodnutelné a jejich doplňkové problémy jsou částečně rozhodnutelné.)

Univerzální algoritmus

Při diskusi algoritmu částečně rozhodujícího problém HP jsme neformálně zmínili, že „spustíme“ M na w, neboli „necháme provádět“ výpočet M pro vstup w. Tím vyjádřením vlastně myslíme, že začneme provádět (nebo necháme provádět) jistý algoritmus U, pro nějž je dvojice M, w vstupem. Algoritmus U tedy „provádí“ („interpretuje“) jakýkoli zadaný algoritmus nad zadaným vstupem. Jedná se tedy o v jistém smyslu výjimečný algoritmus. Není navržen pro specifický úkol jako je sečtení dvou čísel, setřídění databáze, řízení pračky či podobně, ale je v dobře definovaném smyslu schopen

provádět totéž, co jakýkoli jiný algoritmus; potřebuje k tomu samozřejmě mj. dostat popis (kód) algoritmu, který má provádět. Proto jej nazýváme

univerzálním algoritmem.

Kontrolní otázka: Znáte nějaké zařízení, na které lze velmi dobře nahlížet jako na realizátor univerzálního algoritmu?

(Jistě vás napadl počítač. Jeho hlavní činnost se přece dá popsat jako „Proveď zadaný algoritmus (neboli program) pro zadaný vstup (neboli data)“.)

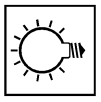
Univerzální algoritmus lze pochopitelně také prezentovat ve formě konkrétního Turingova stroje. Dokonce není příliš těžké takový stroj sestrojit, my to ale zde dělat nebudeme. Jen vyslovíme příslušnou větu.

Věta 6.3 (O univerzálním Turingovu stroji)
Lze sestrojit Turingův stroj U takový, že pro každý vstup $(Kod(M), w)$, kde M je Turingův stroj s abecedou $\{0, 1, \square\}$ a $w \in \{0, 1\}^*$, se U zastaví právě tehdy, když M se zastaví na w; v případě zastavení je navíc výstup U na vstup $(Kod(M), w)$ roven výstupu M na w.

Riceova věta

Uvedeme teď důležitou větu, jež ukazuje nerozhodnutelnost celé třídy problémů. Nebudeme ji dokazovat a vyslovíme ji jen v poněkud neformálním znění; slovo *algoritmus* (či Turingův stroj) zde „pro praktičtější vyznění“ nahrazujeme slovem *program* (v nějakém programovacím jazyku).
Věta mluví o *vstupně/výstupních vlastnostech* programů. Přibližme si, co se tím myslí. Pro každý program Pg si představme (obecně nekonečnou) tabulku s dvěma sloupci: v prvním sloupci jsou všechny přípustné vstupy programu a v druhém sloupci je ke každému vstupu w uveden buď příslušný výstup $Pg(w)$ (v případě, že výpočet Pg pro vstup w je konečný a jako výstup vydá $Pg(w)$) nebo znak $\perp$ („nedefinováno“) v případě, že výpočet Pg na w je nekonečný. Tabulka tedy zachycuje [částečné] zobrazení vstupů na výstupy, které Pg realizuje; můžeme jí říkat např. *I/O-tabulka* (I=input, O=output).

Konkrétní *vlastnost* V programů je *vstupně/výstupní*, jestliže to, zda Pg má či nemá vlastnost V je plně určeno příslušnou I/O-tabulkou programu Pg; jinými slovy: V je vstupně/výstupní právě tehdy, když každé dva programy se stejnou I/O-tabulkou buď oba vlastnost V mají nebo ji oba nemají.



Řešený příklad 6.1: Jako „programovací jazyk“ vezměme Turingovy stroje. (Slovo „program“ je tedy totéž, co „Turingův stroj“.) Zjistěte (a zdůvodněte), které z následujících vlastností (Turingových strojů) jsou vstupně/výstupní. Vlastnost je zadána otázkou,na niž je odpověď buď Ano (příslušný stroj M vlastnost má) nebo Ne (M vlastnost nemá).

a/ Zastaví se M na řetězec 001 ?

b/ Má M více než sto stavů ?

c/ Má v nějakém případě výpočet stroje M více kroků než tisícinásobek délky vstupu ?

d/ Platí, že pro libovolné n se M na vstupech délky nejvýše n vícekrát zastaví než nezastaví ?

e/ Zastaví se M na každém vstupu w za méně než $|w|^2$ kroků?

f/ Je pravda, že pro lib. vstupní slovo M realizuje jeho zdvojení ?

g/ Je pravda, že M má nejvýše sto stavů nebo více než sto stavů ?

Řešení: Zda M má či nemá vlastnost (definovanou otázkou) a/ je jistě plně určeno I/O tabulkou pro M; vlastnost a/ je tedy vstupně/výstupní.

Je zřejmé, že I/O tabulka pro M obecně neurčuje odpověď na otázku b/; jistě můžeme navrhnout konkrétní stroj M_1 např. s 2 stavy a stroj M_2 se 101 stavy, které mají stejnou I/O-tabulku. Vlastnost b/ tedy není vstupně/výstupní.

Podobně I/O tabulka pro M obecně neurčuje odpověď na otázku c/; vlastnost c/ tedy není vstupně/výstupní.

Vlastnost d/ je I/O tabulkou jistě určena, a je tedy vstupně/výstupní.

Je zřejmé, že můžeme navrhnout konkrétní stroje M_1, M_2 se stejnou I/O-tabulkou, přičemž jeden z nich vlastnost e/ má a druhý nemá. Vlastnost e/ tedy není vstupně/výstupní.

Vlastnost f/ je I/O tabulkou určena, a je tedy vstupně/výstupní.

Pozor u vlastnosti g/. Po předchozích zkušenostech nás může v první chvíli napadnout, že se nejedná o vstupně/výstupní vlastnost, když se odkazuje k počtu stavů stroje M. Když se ale podíváme důkladněji, uvědomíme si, že se

jedná o *triviální vlastnost*, neboť ji mají všechny stroje. Taková vlastnost je také de facto I/O tabulkou určena, neboť nemohou existovat dva stroje M_1, M_2 se stejnou I/O-tabulkou, kde jeden ze strojů vlastnost g/ má a druhý ji nemá. Vlastnost g/ tedy je vstupně/výstupní.

Připomeňme tedy ještě, že *vlastnost* V je *triviální*, když ji mají buď všechny programy nebo ji nemá žádný program; taková vlastnost je podle definice také vstupně/výstupní.
A teď už vyslovme Riceovu větu.

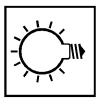
Věta 6.4 (Rice)
Jakákoli netriviální vstupně/výstupní vlastnost programů je nerozhodnutelná.

Zpřesnit se znění věty dá např. takto:
Je-li V netriviální vstupně/výstupní vlastností Turingových strojů, pak je následující problém nerozhodnutelný.

NÁZEV: *Zjišťování vlastnosti* V

VSTUP: Turingův stroj M.

OTÁZKA: Má M vlastnost V ?



ŘEŠENÝ PŘÍKLAD 6.2: Zjistěte, pro které vlastnosti z Řešeného příkladu 6.1 plyne nerozhodnutelnost z Riceovy věty.

Řešení: Zjistili jsme, že jen vlastnosti a/, d/, f/, g/ jsou vstupně/výstupní. Vlastnost g/ je triviální a snadno se přesvědčíme, že a/, d/, f/ triviální nejsou (pro každou z těchto vlastností existuje stroj, který ji má, a stroj, který ji nemá).
Takže z Riceovy věty vyvodíme nerozhodnutelnost pro vlastnosti a/, d/, f/.

Kontrolní otázka: Co myslíte, je pravda, že když pro nějakou vlastnost V programů neplyne její nerozhodnutelnost z Riceovy věty, pak je ta vlastnost rozhodnutelná (tj. problém zjišťování V je rozhodnutelný)?

Není tomu tak. Např. vlastnosti c/ a e/ z Řešeného příkladu 6.1 jsou nerozhodnutelné; jelikož ale nejsou vstupně/výstupní, neplyne jejich nerozhodnutelnost z Riceovy věty.

Shrnutí: Víme již tedy, že nerozhodnutelné problémy se dají jistým způsobem klasifikovat podle míry své nerozhodnutelnosti; uvedli jsme si alespoň (velmi hrubé) rozdělení na ty, co jsou (alespoň) částečně rozhodnutelné, a ty co nejsou (ani) částečně rozhodnutelné. Je nám jasné, že částečná rozhodnutelnost problémů P a $\overline{P}$ znamená rozhodnutelnost P (a $\overline{P}$).
Uvědomujeme si, že na počítač lze nahlížet jako na realizátor tzv. „univerzálního algoritmu“, provádějícího jakýkoli algoritmus (jehož zápis [kód] dostane).
Rozumíme Riceově větě a uvědomujeme si tak, že de facto jakákoli netriviální otázka týkající se ověřování software (jeho korektnosti, spolehlivosti apod.) je algoritmicky nerozhodnutelná.

Kapitola 7

Úvod do teorie složitosti

Cíle kapitoly:

Cílem je zvládnutí základů teorie výpočtové složitosti. Podrobnější cíle jsou patrny z cílů jednotlvých sekcí.

7.1 Složitost algoritmů

Orientační čas ke studiu této části: 3 hod.

Cíle této části:

V této části si máte uvědomit praktickou motivaci definování a zkoumání pojmů jako je časová a paměťová složitost algoritmů. Máte pochopit, proč k tomu potřebujeme (teoretický) model počítače, v našem případě RAMy; máte zvládnout přesné definice složitosti RAMu a uvědomit si i technické otázky související s měřením velikosti vstupů, s jednotkovou a logaritmickou mírou při definování „doby provádění“ jednotlivých instrukcí apod. Máte také pochopit detailní analýzu časové složitosti konkrétního algoritmu (bubblesortu).

Klíčová slova: *velikost vstupu, délka výpočtu RAMu, časová (a paměťová) složitost RAMu, časová (a paměťová) složitost algoritmu, složitost podle nejhoršího případu [worst-case complexity]*

Motivace a neformální definice pojmu „složitost algoritmu“

Již dříve jsme se zabývali tím, co je to algoritmus, co je to problém, co to znamená, že algoritmus řeší daný problém apod. Jeden a tentýž problém může být ovšem řešen řadou různých algoritmů. Pokud by počítače pracovaly nekonečně rychle, příliš by nezáleželo na tom, jaký konkrétní algoritmus použijeme, stačilo by, že by (korektně) řešil daný problém. Tak tomu ovšem není; počítače sice pracují rychle, ale ne nekonečně rychle, provedení každé instrukce trvá nějakou dobu (i když je to třeba jen jedna miliardtina sekundy). Jistě máme dobrou představu např. o tom, že dva různé algoritmy pro setřídění (velké) databáze se mohou významně lišit v době běhu; upřednostíme samozřejmě ten, který databázi setřídí do několika minut, před tím, kterému to trvá třeba půl hodiny či dokonce déle. Pro daný (algoritmicky řešitelný) problém se tedy obvykle snažíme navrhnout (či vybrat z již navržených) algoritmus, který řeší daný problém nejrychleji. Přirozenou otázkou je, jak tedy máme algoritmy porovnávat a jak určit, jak „rychlý“ je daný algoritmus, jinými slovy, jakou má „časovou složitost“ [time complexity]. Někdy nás zajímá i „paměťová složitost“ (též nazývaná „prostorová složitost“ [space complexity]), čili nároky na potřebnou paměť.

Pro konkrétnost si připomeňme problém třídění (sorting; v češtině by zde byl vhodnější termín „seřazování“):

NÁZEV: *Třídění (čísel)*

VSTUP: Konečná posloupnost přirozených čísel.

VÝSTUP: Posloupnost týchž čísel uspořádaná podle velikosti ve vzestupném pořadí.

V učebnicích se často mezi prvními algoritmy řešícími daný problém uvádí tzv. *bubblesort*, jehož základní myšlenka se dá vyjádřit takto:

- Projdi posloupnost zleva doprava, přičemž prohazuješ sousední dvojice čísel, pokud v nich větší číslo předchází menšímu.
- Tento postup procházení posloupnosti opakuj, dokud nedostaneš kompletně uspořádanou posloupnost.

Poznámka: Poznamenejme, že *bubblesort* se v učebnicích vyskytuje spíše jako odstrašující příklad (jelikož lze snadno navrhnout podstatně lepší algoritmy, jak o tom také budeme hovořit dále). My zde tento algoritmus také uvádíme jen pro jeho jednoduchost a ilustraci dále zkoumaných pojmů, nikoliv snad pro jeho „hodnotu“.

Zpřesněné vyjádření algoritmu programátorským pseudokódem by mohlo vypadat takto (pole A obsahuje členy vstupní posloupnosti, které označujeme $A[1], A[2], \ldots, A[n]$):

BUBBLESORT(A, n)
while *Nesetříděno*
do for $i \leftarrow 1$ **to** $n - 1$
do if $A[i] > A[i + 1]$
then prohoď $A[i]$ a $A[i + 1]$

(V této chvíli se náš návrh nezabývá tím, jak se přiřazuje do booleovské proměnné *Nesetříděno*.)

Kontrolní otázka: Proč je uvedený algoritmus korektní – tj. vždy skončí a výsledná posloupnost je uspořádaná?

(Stačí si uvědomit, že v prvním běhu cyklu „probublá“ největší prvek na své místo, po druhém běhu bude na svém místě zaručeně i druhý největší prvek, atd.)

Můžeme zajisté využít většího porozumění předepsanému procesu třídění a upravit (a zpřesnit) algoritmus následovně:

BUBBLESORT – progr. verze

```
▷ členy vstupní posloupnosti nejprve načteme do pole A
▷ předp., že členy jsou nenulové a hodnota 0 označuje konec vstupu
n ← 0
repeat
        n ← n + 1; READ(A[n])
  until A[n] = 0
n ← n − 1
▷ v n je uložen počet členů vstupní posloupnosti
for j ← 1 to n − 1
    do for i ← 1 to n − j
          do if A[i] > A[i + 1]
                then pom ← A[i]; A[i] ← A[i + 1]; A[i + 1] ← pom
▷ výsledná seřazená posloupnost se vypíše
for i ← 1 to n
    do WRITE(A[i])
```

Že tento algoritmus (to je výpočetní proces jím předepsaný) pro každou (konečnou) vstupní posloupnost skončí, je zde zřejmé (proč ?); uspořádanost výsledné posloupnosti plyne z našeho porozumění korektnosti, diskutovaného výše.

Jak jsme už zmínili, *bubblesort* určitě není nejlepším algoritmem řešícím problém třídění. Připomeňme si teď metodu (čili algoritmus) *heapsort*. K tomu je potřebné si připomenout datovou strukturu *halda* (heap), tj. (speciální) binární strom: každý vrchol v je ohodnocen číslem $n(v)$ (prvkem tříděné posloupnosti), přičemž je-li v' následníkem v, pak $n(v) \leq n(v')$. Zařazení dalšího prvku do haldy i výběr nejmenšího prvku z haldy se dají snadno realizovat x kroky, kde x je hloubkou haldy (stromu); při počtu vrcholů n je tedy přibližně $x = \log n$.

Poznámka: V informatice při neuvedení základu $\log n$ většinou myslíme dvojkový logaritmus $\log_2 n$. Později vysvětlíme, proč je základ logaritmu pro účely analýzy algoritmů v zásadě nepodstatný.

Důležitou myšlenkou algoritmu *heapsort* je rovněž efektivní způsob reprezentace haldy jednorozměrným polem.

Vše se dá vyčíst z dále uvedeného pseudokódu; je ovšem velmi žádoucí, ať

si čtenář běh algoritmu ilustruje (připomene) na rozumně zvoleném malém příkladu.

HEAPSORT – progr. verze

```
▷ pole H představuje haldu
▷ kon udává aktuální koncový index haldy
kon ← 0    ▷ halda je prázdná
READ(clen)
while clen ≠ 0
      do ZARAD-DO-HALDY(clen)
         READ(clen)
while kon > 0        ▷ halda není prázdná
      do clen ← VYDEJ-MIN-Z-HALDY()
         WRITE(clen)
```

ZARAD-DO-HALDY(k)

```
kon ← kon +1; H[kon] ← k; p ← kon
while p > 1 and H[⌊p/2⌋] > H[p]
      do prohoď H[⌊p/2⌋] a H[p];
         p ← ⌊p/2⌋
```

VYDEJ-MIN-Z-HALDY()

```
min ← H[1]
if kon > 1
   then H[1] ← H[kon]
kon ← kon −1
p ← 1
while 2 ∗ p + 1 ≤ kon and (H[p] > H[2 ∗ p] or H[p] > H[2 ∗ p + 1])
      do if H[2 ∗ p] ≤ H[2 ∗ p + 1]
            then prohoď H[p] a H[2 ∗ p]; p ← 2 ∗ p
            else  prohoď H[p] a H[2 ∗ p + 1]; p ← 2 ∗ p + 1
if 2 ∗ p = kon and H[p] > H[2 ∗ p]
   then prohoď H[p] a H[2 ∗ p]
return min
```

Oba algoritmy (*bubblesort* a *heapsort*) řeší náš problém třídění, přičemž *heapsort* je očividně složitější z hlediska návrhu, zápisu i porozumění (ověření

správnosti).

Kontrolní otázka: V čem je tedy *heapsort* lepší?

Opověď samozřejmě zní: *heapsort* má menší časovou složitost (náročnost) než *bubblesort*. Neformálně řečeno, *heapsort* „běhá rychleji“ než *bubblesort*. Určitým způsobem se o tom můžeme přesvědčit, naprogramujeme-li obě metody v námi oblíbeném programovacím jazyku a srovnáme běh obou programů na počítači na sadě instancí (tj. povolených vstupů) problému třídění – pro každou instanci měříme čas, který na její zpracování jednotlivé programy spotřebují.

Jistě ovšem tušíme, že obecně takový test nemusí stačit, výsledky na vybraných vstupech nemusí podávat správnou informaci o celém chování (o bězích na všech možných vstupech). Měli bychom tedy své porovnávání kvality algoritmů z hlediska časové (a paměťové) náročnosti opřít o solidnější základ.

Chtělo by to definovat pro každý algoritmus nějakou kvantitativní charakteristiku, nazvané časová složitost (či paměťová složitost), podle které pak bude možné různé algoritmy srovnávat. Složitost ovšem musí zachycovat „dobu běhu“ (či „spotřebu paměti“) globálně – tj. pro všechny přípustné vstupy, nejen pro vybranou sadu testovacích případů.

Nabízí se např. časovou složitost algoritmu prostě definovat jako funkci (zobrazení), která každému (přípustnému) vstupu přiřazuje „dobu běhu“ algoritmu na onen vstup. To má ovšem několik „vad na kráse“ (např. pak není jasné, jak srovnávat rychlost algoritmů pracujících s různými vstupy). Jako vhodnější (jednodušší a přitom postačující) se ukazuje definovat

složitost jako *funkci velikosti vstupu*.

Poznámka: Složitost (jakožto funkce) má tedy většinou nekonečný definiční obor (např. i v našem problému třídění délku vstupní posloupnosti nijak neomezujeme); nelze ji tedy zadat výčtem hodnot, ale je nutno pro ni hledat nějaký konečný popis (např. typu $3n^2 - 4n + 3$) či spíše odhad (např. typu $O(n^2)$), jak o tom budeme hovořit později.

Vstupů se stejnou velikostí n ovšem může být hodně a výpočty pro tyto jednotlivé vstupy mohou trvat různou „dobu“. Co je pak hodnotou (časové) složitosti (tj. zmíněné funkce) pro n? Pro praktické účely často stačí přístup

podle nejhoršího možného případu (worst-case),

kdy dané velikosti n přiřazuje ona funkce maximum z „dob běhu“ algoritmu na všech vstupech velikosti n.

Poznámka: V praxi se často stává, že výpočet pro běžný vstup velikosti n je mnohem rychlejší než pro ten nejhorší případ. Ovšem např. zjištění průměrné doby výpočtu na vstupech velikoti n je často podstatně náročnější než zjištění (resp. odhad) délky výpočtu u nejhoršího případu.
Přístup „podle nejhoršího případu“ poskytuje záruku, že ve skutečnosti se implementace našeho algoritmu bude chovat leda lépe, než jsme odhadli.

Musíme samozřejmě vyjasnit několik věcí – např. co je to *velikost vstupu*.

Kontrolní otázka: Jak byste např. definovali velikost vstupu u problému Třídění?

Obecně použitelné řešení je chápat *velikost daného vstupu* jako *počet bitů, které vstup zabírá* (při „přirozeném“ zakódování). Často lze však postačující výsledky analýzy složitosti dosáhnout bez nutnosti uvažovat tuto „nízkou“ úroveň (která může přidávat zbytečné technické komplikace). Např. u našeho problému Třídění je většinou postačující velikost vstupu definovat jako počet členů zadané posloupnosti (pokud se velikost čísel=členů dá rozumně chápat jako omezená; např. třídíme databázi podle číselného klíče s fixním počtem bytů).

Definice časové a paměťové složitosti algoritmu pomocí RAMu

Všimněme si teď velmi nejasného místa v dosavadních úvahách, a sice užívání pojmu „doba běhu“. Vždyť např. při různých implementacích (v různých programovacích jazycích, na různých počítačích apod.) budou „doby běhu“ jednoho a téhož algoritmu pro jeden a tentýž vstup různé! Jako nejvhodnější exaktní definování pojmu „doba běhu“ se ukazuje volba nějakého (abstraktního) modelu počítače, ke kterému se pak budeme odkazovat jako k jakémusi

referenčnímu modelu.

Dobu běhu (tj. dobu trvání výpočtu neboli délku výpočtu) konkrétního algoritmu A na daný vstup pak chápeme jako počet provedených (elementárních) instrukcí *implementace* A na zmíněném modelu počítače. Model počítače

musí tedy být dostatečně realistický, aby takto definovaná „doba běhu“ nějak rozumně vypovídala o realitě, kterou můžeme očekávat při implementaci algoritmu na fyzickém počítači.

Již dříve jsme avízovali, že jedním takovým modelem, vhodným z hlediska analýzy složitosti algoritmů, jsou stroje RAM (stručněji RAMy, či RAM-programy), se kterými jsme se seznámili v Sekci 5.3.

Nadefinujeme teď pro RAMy časovou a paměťovou složitost, jakožto funkce velikosti vstupu. Přitom se omezujeme jen na RAMy, které se pro každý vstup zastaví (provedou `HALT`, případně skončí „nereguléně“); nedefinujeme časovou ani paměťovou náročnost nekonečných výpočtů.

Definice 7.1
Velikostí vstupu stroje RAM rozumíme počet buněk (vstupní pásky), které daný vstup zabírá.
Délka výpočtu RAMu M pro konkrétní vstup se definuje jako počet provedení instrukcí, které M pro daný vstup vykoná, než se zastaví.
Časovou složitostí RAMu M rozumíme funkci $T_M : \mathbb{N} \to \mathbb{N}$, kde $T_M(n)$ znamená délku výpočtu M nad vstupem velikosti n v nejhorším případě; tedy $T_M(n) = max\{\, k \mid k$ je délka výpočtu M nad (nějakým) vstupem velikosti $n\,\}$.

Definice 7.2
Velikostí paměti RAM-stroje M (potřebné při výpočtu) pro konkrétní vstup rozumíme číslo $p + 1$, kde p je maximum z adres buněk, jež jsou během výpočtu (nad daným vstupem) navštíveny.
Paměťovou složitostí (nebo též *prostorovou složitostí*) RAM-stroje M rozumíme funkci $S_M : \mathbb{N} \to \mathbb{N}$, kde $S_M(n)$ znamená velikost potřebné paměti při výpočtu M nad vstupem velikosti n v nejhorším případě; tedy $S_M(n) = max\{\, k \mid k$ je velikost paměti potřebné při výpočtu M nad (nějakým) vstupem velikosti $n\,\}$.

Teď už je tedy jasnější, co vlastně míníme složitostí algoritmů:

> *Časovou složitostí algoritmu A rozumíme časovou složitost RAMu implementujícího A; podobně pro paměťovou složitost.*

Poznámka: Uvědomme si, že uvedená „definice“ složitosti algoritmu je nutně neformální, nemáme totiž přesnou definici toho, co je to implementace

algoritmu RAMem. Je nám ale jistě jasné, o co jde: Abychom mohli mluvit o složitosti algoritmu A, musí být popsán do té míry podrobnosti, která umožňuje přímočaré přepsání ve formě instrukcí RAMu.
Brzy se také dostaneme k tomu, že složitost algoritmu většinou zkoumáme a odhadujeme „jen zhruba“; to umožňuje nejít při popisu algoritmů do přílišných detailů, které nejsou potřebné pro účel naší analýzy (a jistě tak pro běžné účely analýzy složitosti není nutné algoritmus přepsat až do instrukcí RAMu).

Měli bychom si být vědomi ještě jedné technické věci. Existuje totiž opodstatněná námitka proti uvedeným definicím složitosti RAMu, tvrdící, že takto definovaná délka výpočtu (počítající s provedenou instrukcí RAMu jako s jednotkou) a takto definovaná použitá paměť (počítající s buňkou paměti RAMu jako s jednotkou) může v konkrétním případě dávat přehnaně optimistická očekávání ohledně chování skutečné realizace na počítači.

Kontrolní otázka: Tušíte podstatu této námitky? (Částečně jsme se toho dotkli, když jsme hovořili o tom, jak měříme velikost vstupu problému Třídění.)

Asi jste si uvědomili, že „zakopaný pes“ je v tom, že nijak neomezujeme velikost čísla, které je možno uložit do jednotlivé buňky paměti! Přitom velikost paměti zabrané danou buňkou (a čas elementární operace pracující s danou buňkou) chápeme jako jednotku, jinými slovy uvažujeme tzv.

jednotkovou (či uniformní) míru.

Uvedené definice tedy umožňují „šikovně šetřit paměť“ kódováním celé série čísel (např. matice čísel) číslem jediným. Podobnými „triky“ se dá šetřit i čas výpočtu (počet provedených instrukcí) a získané výsledky pak neodpovídají realitě. Nám samozřejmě nejde o to, vymýšlet tyto „šikovné triky“, musíme si ale být problému vědomi. Pokud podobný nežádoucí efekt hrozí, uvažuje se místo jednotkové míry

míra logaritmická:

je-li v buňce uloženo číslo z, počítá se, že je takto zabrána paměť velikosti $\lceil \log_2(|z|+1)+1 \rceil$, tedy počet bitů potřebných k zapsání z (zhruba tedy $\lceil \log_2 |z| \rceil$). Podobně i cena (čas) provedení jedné instrukce není 1, ale je

úměrná velikosti (binárního) zápisu čísel, se kterými se při provádění instrukce operuje.

V našich algoritmech pro Třídění ovšem žádné uvedené triky nepoužíváme, a předpokládáme, že velikost (binárního zápisu) tříděných čísel je rozumně omezena; v tom případě při analýze složitosti těchto algoritmů můžeme užívat jednotkovou míru (při zachování realističnosti výsledků). (Předpokládáme tak tedy, že doba provedení operací jako načtení čísla, porovnání dvou čísel apod., je omezena konstantou.)

Implementace algoritmu Bubblesort pomocí RAMu

Již jsme říkali, že při analýze složitosti konkrétních algoritmů nám jde většinou o „hrubé odhady“, které nevyžadují popisovat algoritmy na úrovni instrukcí RAMu. Pro ilustraci si ovšem alespoň jeden algoritmus na tak nízké úrovni popišme.

Přepíšeme algoritmus *bubblesort* až do RAM-instrukcí a budeme analyzovat časovou složitost výsledného RAMu. Výsledkem vcelku přímočarého „přeložení“ dříve uvedeného pseudokódu (*Bubblesort – progr. verze*) do „jazyka“ RAM může být níže uvedený program (předpokládáme v něm, že vstupní posloupnost není prázdná).

Vlastní RAM-program, tvořený posloupností 69 instrukcí je uveden v prvním „sloupci“. V druhém sloupci je tentýž program v poněkud srozumitelnější podobě – užívá symbolických návěští a symbolického adresování ($N = 2$, $J = 3$, $HM = 4$, $I = 5$, $IPJ = 6$, $POM = 7$, $X = 8$, $A = 8$; člen $A[1]$ bude uložen v buňce 9, $A[2]$ v buňce 10, $A[3]$ v buňce 11 atd.). Třetí sloupec obsahuje komentáře, ze kterých by mělo být patrno, že se skutečně jedná o „překlad“ uvedené verze *bubblesortu*. (Připomeňme, že všechny paměťové buňky mají na začátku hodnotu 0.)

Bubblesort, RAM-verze:

```
01  LOAD   =1              LOAD   =1          ; do indexreg. se vloží 1,
02  STORE  1               STORE  1           ; tj. první volný index pole A
03  READ         Cykl-vst: READ               ; načtení dalšího vstupu
04  JZERO  10              JZERO  Kon-vst     ; 0 znamená konec vstupu
05  STORE  *8              STORE  *A          ; A[indexreg] ← vstup
06  LOAD   1               LOAD   1           ;
07  ADD    =1              ADD    =1          ; indexreg. se zvýší o 1
```

```
08  STORE  1              STORE  1          ;
09  JUMP   3              JUMP   Cykl-vst    ;
10  LOAD   1    Kon-vst:  LOAD   1          ;
11  SUB    =1             SUB    =1         ;
12  STORE  2              STORE  N          ; N obsah. počet vst. čísel
13  LOAD   3     Cykl-1:  LOAD   J          ;
14  ADD    =1             ADD    =1         ; J ← J + 1
15  STORE  3              STORE  J          ;
16  LOAD   2              LOAD   N          ;
17  SUB    3              SUB    J          ;
18  JZERO  58             JZERO  Vystup     ; skok při J = N
19  ADD    =1             ADD    =1         ;
20  STORE  4              STORE  HM         ; HM (hor. mez) ← N − J + 1
21  LOAD   =0             LOAD   =0         ;
22  STORE  5              STORE  I          ; I ← 0
23  LOAD   5     Cykl-2:  LOAD   I          ;
24  ADD    =1             ADD    =1         ; I ← I + 1
25  STORE  5              STORE  I          ;
26  ADD    =1             ADD    =1         ;
27  STORE  6              STORE  IPJ        ; IPJ ← I + 1
28  LOAD   4              LOAD   HM         ;
29  SUB    5              SUB    I          ;
30  JZERO  13             JZERO  Cykl-1     ; skok při I = HM
31  LOAD   5              LOAD   I          ;
32  STORE  1              STORE  1          ;
33  LOAD   *8             LOAD   *A         ;
34  STORE  8              STORE  X          ; X ← A[I]
35  LOAD   6              LOAD   IPJ        ;
36  STORE  1              STORE  1          ;
37  LOAD   8              LOAD   X          ;
38  SUB    *8             SUB    *A         ;
39  JGTZ   41             JGTZ   Prohod     ; skok při X > A[I + 1]
40  JUMP   23             JUMP   Cykl-2     ;
41  LOAD   5     Prohod:  LOAD   I          ;
42  STORE  1              STORE  1          ;
43  LOAD   *8             LOAD   *A         ;
44  STORE  7              STORE  POM        ; POM ← A[I]
45  LOAD   6              LOAD   IPJ        ;
46  STORE  1              STORE  1          ;
47  LOAD   *8             LOAD   *A         ;
48  STORE  8              STORE  X          ; X ← A[I + 1]
49  LOAD   5              LOAD   I          ;
50  STORE  1              STORE  1          ;
51  LOAD   8              LOAD   X          ;
52  STORE  *8             STORE  *A         ; A[I] ← X
```

```
53  LOAD   6              LOAD   IPJ        ;
54  STORE  1              STORE  1          ;
55  LOAD   7              LOAD   POM        ;
56  STORE  *8             STORE  *A         ; A[I+1] ← POM
57  JUMP   23             JUMP   Cykl-2     ;
58  LOAD   =2    Vystup:  LOAD   =1         ;
59  STORE  1              STORE  1          ; indexreg. ← 1
60  LOAD   *8  Cykl-vys:  LOAD   *A         ;
61  WRITE                 WRITE              ; write(A[indexreg.])
62  LOAD   2              LOAD   N          ;
63  SUB    1              SUB    1          ;
64  JZERO  69             JZERO  Konec      ; skok při indexreg. = N
65  LOAD   1              LOAD   1          ;
66  ADD    =1             ADD    =1         ;
67  STORE  1              STORE  1          ; indexreg. se zvýší o 1
68  JUMP   60             JUMP   Cykl-vys   ;
69  HALT          Konec:  HALT              ;
```

Spočtěme nyní, kolik instrukcí bude provedeno při zpracování vstupu velikosti n (v nejhorším možném případě).

První (vstupní) fáze výpočtu, od začátku po první příchod na instrukci s návěštím `Cykl-1`, zřejmě zabere „čas“ (tj. počet provedení instrukcí)

$$T_1 = 2 + 7n + 2 + 3 = 7n + 7\,.$$

Podobně třetí (výstupní) fáze, od skoku na `Vystup` po `Konec`, zabere zřejmě čas

$$T_3 = 2 + 9(n-1) + 5 + 1 = 9n - 1\,.$$

Složitější je analyzovat zbylou (prostřední) fázi, od prvního skoku na `Cykl-1` po skok na `Vystup`. Také s přihlédnutím k naší progr. verzi *bubblesortu* není ovšem zase tak obtížné odvodit, že ona prostřední fáze trvá

$$T_2 = \Big(\sum_{j=1}^{n-1}\Big(10 + \Big(\sum_{i=1}^{n-j} 34\Big) + 8\Big)\Big) + 6\,.$$

Poznamenejme, že vždy předpokládáme ten horší (tj. delší k zpracování) případ, kdy skutečně dojde k prohazování prvků (tj. provádí se „podprogram“ `Prohod`).

Výraz pro T_2 můžeme upravovat standardní manipulací se sumami například takto:

$$T_2 = 6 + \sum_{j=1}^{n-1} 18 + \sum_{j=1}^{n-1} \sum_{i=1}^{n-j} 34 = 6 + 18(n-1) + \sum_{j=1}^{n-1} 34(n-j) =$$

$$= 18n - 12 + 34 \sum_{j=1}^{n-1} n - 34 \sum_{j=1}^{n-1} j \,.$$

Dosadíme-li za první sumu $n(n-1)$ a za druhou sumu $(1+2+\ldots+n-1) = (n/2)(n-1)$, odvodíme pak již přímočarými úpravami vztah

$$T_2 = 17n^2 + n - 12 \,.$$

Celkový čas potřebný pro zpracování vstupu velikosti n je tedy $T_1+T_2+T_3 = 17n^2 + 17n - 6$. Označíme-li náš RAM-stroj M a jeho časovou složitost T_M, ukázali jsme tak, že pro každé $n \geq 1$ je

$$T_M(n) = 17n^2 + 17n - 6 \,.$$

Shrnutí: Naše dřívější intutivní porozumění pojmu složitost algoritmu je nyní postaveno na pevné bázi. Víme, že (časová nebo paměťová) složitost algoritmu je funkcí velikosti vstupu a vztahuje se k nějakému referenčnímu modelu počítačů; v našem případě jsme použili RAMy. Jsme si vědomi technických souvislostí (měření velikosti vstupu, jednotková a logaritmická míra, přístup „podle nejhoršího případu“) a chápeme, že konkrétní algoritmus se dá přepsat až do instrukcí RAMu, na nichž se můžeme pokusit příslušnou složitost detailně analyzovat. Naštěstí už ovšem také víme, že obvykle se popis algoritmů a analýza jejich složitosti nedělá takhle detailně, protože nám pro praktické závěry stačí daleko hrubší odhady složitosti algoritmů (o nichž budeme mluvit v následující části).

7.2 Asymptotická složitost, odhady řádového růstu funkcí

Orientační čas ke studiu této části: 3 hod.

Cíle této části:

V této části máte porozumět značení $O(.)$, $o(.)$, $\Theta(.)$, které se používá při běžné analýze složitosti algoritmů a vyjadřuje odhady řádového růstu příslušných funkcí. Máte si uvědomit vztahy mezi základními funkcemi objevujícími se při analýze složitosti algoritmů a zvládnout rychlé vyvozování těchto vztahů v konkrétních případech.

Klíčová slova: *řádový růst funkcí, zanedbávání konstantních faktorů, asymptotické chování, značení $O(.)$, $o(.)$, $\Theta(.)$, lineární, kvadratické a další funkce, polynomiální funkce, exponenciální funkce*

Při analýze časové složitosti *bubblesortu* jsme viděli, že přesné (algebraické) vyjádření funkce T_M (pro příslušný RAM M) není technicky úplně triviální úkol ani u tohoto jednoduchého programu. U větších a komplikovanějších programů by takový postup byl podstatně náročnější.

Pro naše účely (srovnávání algoritmů) naštěstí přesné vyjádření časové složitosti není nutné; většinou postačí „rozumný“ odhad příslušné funkce T_M. (Totéž se týká paměťové složitosti S_M, ale my se zaměřujeme přednostně na časovou složitost.)

Poznámka: Samozřejmě slovem „odhad“ míníme „horní odhad“ (neřekneme-li jinak). Pokračujeme tak v přístupu „podle nejhoršího případu“. Chceme totiž, aby (skutečné) chování algoritmu (z hlediska dob běhu, či spotřeby paměti) nebylo horší než říká výsledek naší analýzy; pokud je ve skutečnosti lepší než říká náš (horní) odhad, je to pro nás jen pozitivní.

Teď se dostáváme k nejdůležitějšímu obratu, používanému v analýzách složitostí algoritmů:

> *funkce $f(n)$ a $c \cdot f(n)$ (kde $c > 0$) bereme jako ekvivalentní, tj. „stejně rychle rostoucí“!*

To vypadá na první pohled jako velmi velké „zhrubnutí“ našich analýz; např. mezi odhady $0.0001n^2$, $17n^2$, $10000000n^2$ nerozlišujeme, všechny jsou prostě typu „n^2 krát nějaká kladná konstanta“. Může takhle hrubá analýza vůbec přinést nějaké smysluplné výsledky, které mají praktický smysl?
V praxi se ukazuje, že ano. My se k tomu vrátíme, teď si však nejprve zavedeme značení, které slouží k elegantnímu vyjádření onoho „zanedbávání

konstantních faktorů“.
Běžně je používáno značení z následující definice.

Definice 7.3
Pro libovolné funkce $f, g : \mathbb{N} \to \mathbb{N}$ znamená zápis $f \in O(g)$ či $f(n) \in O(g(n))$ toto:

$$(\exists k \in \mathbb{N})(\exists n_0 \in \mathbb{N})(\forall n \geq n_0) : \ f(n) \leq k \cdot g(n)\,.$$

Někdy se také píše $f = O(g)$ či $f(n) = O(g(n))$. Uvedené zápisy čteme jako „f patří do $O(g)$“ nebo „f je v $O(g)$“ nebo „f je $O(g)$“ apod. Říkáme také, že „$f(n)$ roste řádově nejvýše jako $g(n)$“ apod.

Je-li tedy $f(n) \in O(g(n))$, slouží funkce $g(n)$ jako určitý horní odhad (řádového růstu) funkce $f(n)$. Jak vidíme, neznamená to nutně, že platí $\forall n : f(n) \leq g(n)$. Mluvíme zde o horních odhadech v onom „hrubém smyslu“, odhadujeme pouze „řádový růst“, při němž zanedbáváme přesnou hodnotu konstantních faktorů.

Kontrolní otázka: Je vám např. jasné, že $17n^2 \in O(0.0001n^2)$?

Jistě ano. Určitě totiž platí $(\exists k \in \mathbb{N})(\forall n \geq 0) : \ 17n^2 \leq k \cdot 0.0001n^2$. (Stačí vzít $k \geq 170000$.)

Proč je ale v definici vlastně ono $\ldots (\exists n_0 \in \mathbb{N})(\forall n \geq n_0) \ldots$? (V našem předchozím příkladu jsme tiše brali $n_0 = 0$.) Chceme samozřejmě, aby při našich analýzách (odhadujících řádový růst složitosti) byla např. funkce $g(n) = n^2$ horním odhadem pro funkci $f(n) = n + 1$. Ovšem jelikož $g(0) = 0$, nepomůže v tomto případě vynásobení jakkoli velkou konstantou k „přebití“ $f(0) = 1$.

Poznámka: Jinými slovy se dá říci, že $f(n) \in O(g(n))$ právě tehdy, když pro (dostatečně velkou) konstantu k platí $f(n) \leq k \cdot g(n))$ pro skoro všechna n, tedy s případným konečným počtem výjimek.
Ještě jinak lze $f \in O(g)$ vyjádřit takto: existují konstanty $c, d \geq 0$ takové, že

$$\forall n \in \mathbb{N} : f(n) \leq c \cdot g(n) + d\,.$$

(Rozmyslete si, že uvedená vyjádření jsou opravdu ekvivalentní.)

Snadno si také uvědomíme následující „tranzitivitu“:

Tvrzení 7.4
Jestliže $f \in O(g)$ a $g \in O(h)$, tak $f \in O(h)$.

V našem konkrétním příkladu u *bubblesortu* jsme odvodili $T_M(n) \leq 17n^2 + 17n - 6$. Připomeňme, že jsme uvažovali jen neprázdné vstupy, takže bychom měli přesněji psát: $\forall n \geq 1 : T_M(n) \leq 17n^2 + 17n - 6$.

Poznámka: U jakéhokoli stroje M musí samozřejmě být $T_M(n) \in \mathbb{N}$ pro všechny velikosti [přípustných] vstupů n. Někdy při analýzách pracujeme i s funkcemi, jejichž hodnoty jsou pro některé argumenty záporné (např. $17n^2 + 17n - 6$ pro $n = 0$) či necelé (např. $0.5n$, $n^{\frac{1}{2}}$, $n \log_2 n$ apod.). V takových případech tiše předpokládáme, že záporné hodnoty jsou nahrazeny nulou a necelé hodnoty jsou zaokrouhlovány.

Odvodili jsme tedy $T_M(n) \in O(17n^2 + 17n - 6)$. Když se ale zamyslíme, zjistíme, že toto vyjádření říká vlastně totéž jako stručnější $T_M(n) \in O(n^2)$.

Kontrolní otázka: Proč?

Stačí ukázat, že $17n^2 + 17n - 6 \in O(n^2)$ a $n^2 \in O(17n^2 + 17n - 6)$. Máme ovšem $17n^2 + 17n - 6 \leq c_1 n^2$ např. pro $c_1 = 34$ a $n^2 \leq c_2(17n^2 + 17n - 6) + d_2$ např. pro $c_2 = 1$ a $d_2 = 6$.

Když si to promyslíme, uvědomíme si, že pro získání odhadu typu $T_M(n) \in O(n^2)$ jsme *bubblesort* jistě nemuseli fyzicky naprogramovat jako RAM – když nám nejde o přesné hodnoty konstant (a tedy nerozlišujeme, zda určitý úsek algoritmu se dá implementovat pěti, padesáti, či pěti sty instrukcemi RAMu), dokážeme využitím naší programátorské zkušenosti časovou složitost odhadnout např. již z pseudokódu.
(Promyslete si to!)

Podívejme se pořádněji na to, jak jsem ukázali jistou ekvivalenci funkcí $17n^2 + 17n - 6$ a n^2. Vede nás k to k zavedení dalšího značení a vyvození následujícího tvrzení.

Definice 7.5
Zápis $f \in \Theta(g)$ (či $f(n) \in \Theta(g(n))$) znamená, že $f \in O(g)$ a $g \in O(f)$.

Značení Θ tedy vyjadřuje, že příslušné funkce rostou „řádově stejně rychle“.

Tvrzení 7.6
Pro libovolný polynom $a_d n^d + a_{d-1} n^{d-1} + \ldots + a_1 n + a_0$, *kde* $a_d > 0$, *platí:*

$$a_d n^d + a_{d-1} n^{d-1} + \ldots + a_1 n + a_0 \in \Theta(n^d)\,.$$

Objeví-li se tedy při našich analýzách (řádového růstu složitosti) polynom, je v něm rozhodující člen s největším exponentem. (Při vynásobení dostatečně velkou konstantou jistě „přebije“ všechny ostatní členy dohromady.)

Náš závěr z (hrubé) analýzy *bubblesortu* zněl „bubblesort je v $O(n^2)$“, což bereme jako zkratku pro „časová složitost algoritmu *bubblesort* je v $O(n^2)$“. Připomeňme ale, že O vyjadřuje *horní odhad* (řádového růstu složitosti), takže toto vyjádření nevylučuje, že je možné uvedený horní odhad zlepšit. Ve skutečnosti je ovšem funkce n^2 pro *bubblesort* nejen horním, ale i spodním (řádovým) odhadem, a můžeme tedy přesněji říkat „bubblesort je v $\Theta(n^2)$“.

Kontrolní otázka: Proč se odvozený horní odhad n^2 pro bubblesort nedá zlepšit?

Zde je to celkem jednoduché: program *bubblesort* jsme přímo uvedli tak, že provádí řádově n-krát jistý cyklus, v jehož těle probíhá „průměrně $\frac{n}{2}$-krát“ jistý vnořený cyklus. Takže např. i pro již seřazenou vstupní posloupnost délky n provede bubblesort řádově n^2 kroků. (To by se zásadně nezměnilo, ani kdybychom vždy testovali, zda už nemůžeme skončit, protože k žádnému prohození nedošlo. Např. na opačně seřazenou posloupnost by zase bylo potřeba řádově n^2 kroků, takže pro každé n existuje (alespoň jeden) vstup velikosti n, pro nějž se vykoná řádově n^2 kroků.)

Kontrolní otázka: Umíte teď odhadnout časovou složitost *heapsortu*?

Jistě není problém z našeho dříve uvedeného popisu zjistit, že „heapsort je v $O(n \log n)$“. Dalším promyšlením bychom odvodili, že se odhad nedá zlepšit, že tedy „heapsort je v $\Theta(n \log n)$“, ale již zjištění horního odhadu $O(n \log n)$ umožňuje závěr, že „heapsort je (přinejmenším asymptoticky) rychlejší než bubblesort“.
Je totiž zřejmé, že $(n \log n)$ roste „řádově (striktně) pomaleji“ než n^2: Nejenže n^2 je horním odhadem pro $(n \log n)$, tedy $(n \log n) \in O(n^2)$, ale je to „ostrý“ horní odhad, protože $n^2 \notin O(n \log n)$.

Kontrolní otázka: Umíte zdůvodnit, že $n^2 \notin O(n \log n)$?

Ať vynásobíme funkci $(n \log n)$ jakoukoli konstantou, pro dostatečně velké hodnoty n ji n^2 „přebije“. Přesněji vyjádřeno: $(\forall k)(\exists n_0)(\forall n \geq n_0) : kn \log n < n^2$. (Zdůvodněte, proč $k \log n < n$ pro dostatečně velká n.)

K vyjádření onoho ostrého horního odhadu se také používá značení o (jedná se o malé-o, na rozdíl od velkého-O):

Definice 7.7
Zápis $f \in o(g)$, či $f(n) \in o(g(n))$, znamená, že
$(\forall k \in \mathbb{N})(\exists n_0 \in \mathbb{N})(\forall n \geq n_0) : k \cdot f(n) < g(n)$.

Poznámka: Dá se také ukázat, že $f(n) \in o(g(n))$ se dá ekvivalentně vyjádřit jako

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0\,.$$

Shrňme si dosud definované zápisy (odhadů růstu funkcí) spolu s jejich neformálním vyjádřením:

- $f \in O(g)$ .... f roste nejvýše tak rychle jako g,
- $f \in o(g)$ .... f roste (striktně) pomaleji než g,
- $f \in \Theta(g)$ .... f roste stejně rychle jako g.

Poznámka: Ještě jednou zdůrazněme, že naše „hrubé“ odhady se zabývají *asymptotickým chováním* (složitosti algoritmu), tedy chováním pro „velké“ hodnoty. Když tedy např. víme, že algoritmus A má složitost $O(n \log n)$ a algoritmus B má složitost $\Theta(n^2)$, tak to ještě neznamená, že je lepší A použít za všech okolností. Konstanta skrytá v uvedeném $O(n \log n)$ může být totiž mnohonásobně větší než konstanta skrytá v $\Theta(n^2)$, což může znamenat, že pro ty vstupy (omezené velikosti), které nás u příslušného problému zajímají, je B ještě pořád rychlejší než A. (To ale není případ bubblesortu a heapsortu.)

Funkce, které se běžně objevují v analýzách složitostí algoritmů, jsou např.

n, $n \log n$, n^2, n^3, 2^n, $n!$, n^n apod.

Je zřejmé, že funkce jsou zde uspořádány podle růstu. Platí $f_i(n) \in o(f_j(n))$, je-li f_i v seznamu vlevo od f_j. Je ovšem velmi užitečné provést si srovnání hodnot těchto funkcí pro konkrétní n, která se dají očekávat jako velikosti vstupů. (Zkuste si to provést!)

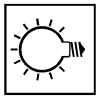
Značení: Běžně se užívají následující názvy:

- $f(n) \in \Theta(\log n)$ .... *logaritmická* funkce,

- $f(n) \in \Theta(n)$ .... *lineární* funkce,
- $f(n) \in \Theta(n^2)$ .... *kvadratická* funkce,
- $f(n) \in O(n^c)$ pro nějaké $c > 0$ .... *polynomiální* funkce,
- $f(n) \in \Theta(c^n)$ pro nějaké $c > 1$ .... *exponenciální* funkce.

Příklad: (ilustrující rozdíl v růstu některých funkcí)
- pro funkci $f(n) \in \Theta(n)$ platí: pokud n vzroste na dvojnásobek, tak hodnota $f(n)$ taktéž vzroste zhruba na dvojnásobek.
- pro funkci $f(n) \in \Theta(n^2)$ platí: pokud n vzroste na dvojnásobek, tak hodnota $f(n)$ vzroste zhruba na čtyřnásobek.
- ovšem pro funkci $f(n) \in \Theta(2^n)$ platí: pokud n vzroste (pouze) o 1, tak hodnota $f(n)$ vzroste zhruba na dvojnásobek. To ilustruje *obrovský rozdíl* mezi polynomiálními a exponenciálními funkcemi.

Již jsme říkali, že funkcí $\log n$ běžně rozumíme $\log_2 n$, tedy logaritmus při základu 2. Zmínili jsme také, že základ logaritmů v analýze složitosti de facto nehraje roli. Teď toto tvrzení zpřesníme.



ŘEŠENÝ PŘÍKLAD 7.1: Pro konstanty $a, b > 1$ ukažte, že $\log_a n \in \Theta(\log_b n)$.

Řešení: Stačí si uvědomit, že $\log_a n = (\log_a b)(\log_b n)$.

Otázky:

OTÁZKA 7.1: Co můžeme usoudit ze vztahu $f(n) \in O(g(n))$ pro porovnání hodnot $f(10)$ a $g(10)$?

OTÁZKA 7.2: Může v případě $f(n) \in O(g(n))$ platit $\forall n : f(n) > g(n)$?

OTÁZKA 7.3: Může v případě $f(n) \in o(g(n))$ platit $\forall n : f(n) > g(n)$?

CVIČENÍ 7.4: Uvažme problém sečtení dvou (velmi velkých) dekadicky zapsaných čísel. Jaká je velikost vstupu pro vstupní čísla x_1, x_2 ? (Stačí řádový odhad.)

CVIČENÍ 7.5: Průměr z daných $n > 1$ čísel spočítáme následující funkcí:

```
PRŮMĚR(X, n)
    z ← 0.0
    for i ← 1 to n
        do z ← z + X[i]
    return z/n
```

Určete, kolik tato funkce PRUMER vykoná aritmetických operací v závislosti na n.

CVIČENÍ 7.6: Určete, kolik průchodů vnitřním cyklem provede pro vstup n následující jednoduchý program.

```
ALG1(n)
    for i ← 1 to n * n
        do for j ← 1 to i
               do print "jeden průchod"
```

Je to $\Theta(n^2)$ nebo $\Theta(n^3)$ nebo $\Theta(n^4)$?

CVIČENÍ 7.7: Udejte správný asymptotický vztah mezi funkcemi $\sqrt{n}$ a $\log n$.

CVIČENÍ 7.8: Která z těchto funkcí roste nejrychleji?

a) $1000n$ b) $n \cdot \log n$ c) $n \cdot \sqrt{n}$

CVIČENÍ 7.9: Rozhodněte, které z následujících vztahů mezi funkcemi proměnné n jsou platné.

a) $\log n \in O(\sqrt{n})$

b) $\sqrt{n} \in O(n)$

c) $2^n \in O(n^n)$

d) $2^n \in O(n^{1024})$

e) $n! \in O(2^n)$

f) $n^{\log n} \in O(n^{1024})$

Cvičení 7.10: Seřaďte následující tři funkce podle rychlosti jejich růstu, od nejpomalejšího růstu po nejrychlejší.

a) $n + \sqrt{n} \cdot \log n$

b) $n \cdot \log n$

c) $\sqrt{n} \cdot \log^2 n$

Cvičení 7.11: Seřaďte následující tři funkce podle rychlosti jejich růstu, od nejpomalejšího růstu po nejrychlejší.

a) 2^n

b) $2^{\sqrt{n}}$

c) $n!$

Cvičení 7.12: Seřaďte následující tři funkce podle rychlosti jejich růstu, od nejpomalejšího růstu po nejrychlejší.

a) $n/2005$

b) $\sqrt{n} \cdot 3n$

c) $n + n \cdot \log n$

Cvičení 7.13: Seřaďte následující tři funkce podle rychlosti jejich růstu, od nejpomalejšího růstu po nejrychlejší.

a) $(\log n)^n$

b) n^n

c) $2^{\sqrt{n}}$

Shrnutí: Teď už bezpečně ovládáme značení $O(.)$, $o(.)$, $\Theta(.)$ a chápeme jeho význam: umožňuje stručně vyjadřovat informace získané z běžné analýzy složitosti algoritmů, při níž nemusíme jít do přílišných detailů popisu algoritmů (a nemusíme je popisovat na úrovni instrukcí RAMu. Máme dobrou představu o markantním rozdílu mezi polynomiálními a exponenciálními funkcemi z hlediska jejich růstu. Zvládáme rychle porovnávat běžné funkce vyskytující se v analýzách složitosti.

7.3 Polynomiální algoritmy, třídy složitosti, třída PTIME

Orientační čas ke studiu této části: 2 hod.

Cíle této části:

V této části si máte promyslet pojem „složitost problému“ a umět jej definovat pomocí „tříd složitosti“. Speciálně si máte uvědomit, proč je třída PTIME (obsahující problémy řešitelné polynomiálními algoritmy) brána jako aproximace „prakticky zvládnutelných problémů“. Máte si rovněž promyslet příklady konkrétních problémů z PTIME.

Klíčová slova: *polynomiální algoritmus, třídy složitosti, třída* PTIME*, polynomiálně ekvivalentní výpočetní modely, robustnost třídy* PTIME

Polynomiální algoritmy

Ze své praxe všichni jistě známe řadu různých algoritmů pro řešení problémů. Když bychom se podívali podrobněji na ty, které jsou běžně používány (jako programy běžící na počítačích), zjistili bychom, že se většinou jedná o algoritmy s *polynomiální časovou složitostí*; takový algoritmus má tedy složitost v $O(n^c)$ pro nějakou konstantu c. Typicky je přitom c velmi malé; zkušenost ukazuje, že když se pro přirozeně definovaný praktický problém podaří nalézt *polynomiální algoritmus* (čímž zkráceně označujeme algoritmus s polynomiální časovou složitostí), pak se většinou podaří nalézt i algoritmus se složitostí $O(n^6)$ či lepší.

Poznámka: Metodami, které zde nebudeme diskutovat, se dá ukázat, že pro libovolné c existuje problém, který se nedá řešit v čase $O(n^c)$ (tedy algoritmem s časovou složitostí $O(n^c)$), ale dá se řešit v čase $O(n^{c+1})$. Jedná se ovšem o uměle zkonstruované problémy pro účel důkazu takového tvrzení.
V praxi se neobjevují problémy, pro něž bychom znali algoritmy s časovou složitostí např. $O(n^{60})$, ale ne lepší.

Problémy, pro něž existují polynomiální algoritmy se obecně považují za *(prakticky) zvládnutelné problémy*. Bohužel je mnoho problémů z běžné praxe, pro něž polynomiální algoritmy nejsou známy či se dokonce ví, že takové algoritmy neexistují. Po našich zkušenostech s růstem funkcí víme, že máme-li algoritmus, který není polynomiální (typicky má exponenciální či ještě vyšší složitost), nemůžeme doufat v jeho použití pro větší vstupy (protože bychom se u nich „nedožili konce výpočtu“).
Máme-li tedy problém k řešení, první otázkou je, jestli jsme schopni pro něj navrhnout (nebo nalézt v literatuře apod.) polynomiální algoritmus. Až v druhé fázi se pak zabýváme tím, zda se podaří navrhnout takový algoritmus se složitostí např. $O(n^3)$ nebo $O(n^2)$ či ještě lépe $O(n \log n)$ či úplně ideálně $O(n)$.

Kontrolní otázka: Proč u standardního praktického problému nemůžeme doufat, že se ho podaří řešit ještě lépe, tedy v čase $o(n)$?

(U „normálního“ problému musí řešící algoritmus přinejmenším přečíst vstup, což zabere $\Theta(n)$ kroků.)

Teď si alespoň vyjmenujeme několik problémů, pro které známe polynomiální algoritmy (v $O(n^c)$ pro malé c).

- třídění

- vyhledávání (a vkládání apod.) v lineárně či stromově (grafově) implementovaných datových strukturách

- aritmetické operace (násobení či dělení [velkých] čísel apod.)

- ekvivalence (deterministických) konečných automatů

- problém příslušnosti k bezkontextovému jazyku (vstup: bezkontextová gramatika G a slovo w; otázka: je $w \in L(G)$?)

- problém nalezení nejkratší cesty v grafu

- problém minimální kostry v grafu

- dále specifikované problémy

NÁZEV: *Výběr aktivit*

VSTUP: Množina konečně mnoha aktivit $\{1, 2, \ldots, n\}$ s pevně určenými časovými intervaly (s_1, f_1), (s_2, f_2), ..., (s_n, f_n), kde $(\forall i, 1 \leq i \leq n) : s_i < f_i$

VSTUP: Množina obsahující největší možný počet vzájemně kompatibilních aktivit (tj. aktivit s vzájemně se nepřekrývajícími intervaly)

NÁZEV: *Optimalizace násobení řetězce matic*

VSTUP: Posloupnost matic $A_1, A_2, \ldots, A_n$.

VÝSTUP: Plně uzávorkovaný součin $A_1 A_2 \ldots A_n$ tž. při použití standardního algoritmu násobení matic se vykoná minimální počet skalárních násobení.

NÁZEV: LCS (*problém nejdelší společné podposloupnosti*)

VSTUP: Dvě posloupnosti v, w v nějaké abecedě Σ.

VÝSTUP: Nejdelší společná podposloupnost posloupností v, w.

Poznámka: Navrhnout polynomiální algoritmy není pro některé z uvedených problémů triviální úkol. Většinou se ale přímočaře uplatní obecné

postupy používané při návrhu efektivních (tj. rychlých polynomiálních) algoritmů, jako jsou např. přístupy „rozděl a panuj“, tzv. dynamické programování, „hltavý [greedy] přístup“ (u některých optimalizačních problémů), apod.

Složitost problémů

Nepřímo jsme při naší diskusi narazili na to, že vedle „složitosti algoritmů“ se potřebujeme vyjadřovat i o „složitosti problémů“. Intuitivně cítíme, že různé problémy mohou být různě „složité“; co to ale je ona složitost problémů? Víme-li např. o algoritmu (tedy RAMu), který daný problém řeší a má složitost $\Theta(n^3)$, je toto $\Theta(n^3)$ jen určitým horním odhadem „skutečné“ složitosti problému (můžeme říci „složitost problému je nejvýše kubická“). Je např. možné, že nalezneme jiný algoritmus, který řeší náš problém a který má složitost $O(n^2)$; tím jsme náš dosud známý odhad zlepšili (víme pak už, že „složitost problému je nejvýše kvadratická“).

Na určení *horního odhadu* $O(f(n))$ složitosti problému stačí tedy pro daný problém navrhnout algoritmus s časovou složitostí $O(f(n))$.

Poznámka: Jen poznamenejme, že daleko těžší je ukazovat *spodní odhady* složitosti problému, tedy např. to, že *neexistuje* algoritmus, který daný problém řeší v čase $o(f(n))$ apod.

Jistě nás teď napadne, že složitost problému bychom mohli definovat jako složitost „optimálního“ algoritmu, který daný problém řeší. Při exaktní definici onoho pojmu „optimální“ ovšem vzniknou jisté komplikace. Proto se pojem složitosti problému nedefinuje přímo, ale zavádějí se tzv. třídy složitosti, definované níže.

Definice 7.8
Pro funkci $f : \mathbb{N} \to \mathbb{N}$ rozumíme *třídou časové složitosti* $\mathcal{T}(f)$, též značenou $\mathcal{T}(f(n))$, množinu těch problémů, které jsou řešeny RAMy s časovou složitostí v $O(f)$.

Třídou prostorové složitosti $\mathcal{S}(f)$, též $\mathcal{S}(f(n))$, rozumíme třídu těch problémů, které jsou řešeny RAMy s prostorovou složitostí v $O(f)$.

Důležitá úmluva: V předchozí definici a všude, kde hovoříme o třídách složitosti vztahujících se k RAMu jako k referenčnímu modelu, máme vždy při

odkazu na složitost RAMu na mysli *logaritmickou míru*, pokud není uvedeno jinak. Jde o to, aby takto definované třídy složitosti rozumně odpovídaly realitě.

Poznámka: Podobně bychom mohli zavést definice tříd složitosti i pro jiné výpočetní modely (např. pro Turingovy stroje). V takovém případě však dostaneme *jiné* třídy složitosti, proto je třeba vždy uvést, k jakému výpočetnímu modelu se dané třídy složitosti vztahují.

Řekneme-li tedy například, že problém P patří do $\mathcal{T}(n^2)$ (taky se v této souvislosti říká „časová složitost P je v $O(n^2)$“ či ještě stručněji „P je v $O(n^2)$“), říkáme tím, že existuje algoritmus s nejvýš kvadratickou časovou složitostí, který řeší problém P (přesněji řečeno: existuje RAM s nejvýš kvadratickou časovou složitostí v logaritmické míře, který řeší problém P).

Poznámka: Připomeňme, že pro příslušnost problému k dané třídě složitosti je důležitý i způsob kódování vstupu (a ten je tedy nutno chápat jako součást definice problému).

Všimněme si, že platí

$$P \in \mathcal{T}(f) \ \wedge \ f \in O(g) \ \Rightarrow \ P \in \mathcal{T}(g)$$

Takže např.

$$\mathcal{T}(n) \subseteq \mathcal{T}(n \cdot log\, n) \subseteq \mathcal{T}(n^2) \subseteq \mathcal{T}(n^3) \subseteq \mathcal{T}(2^n)$$

Třída PTIME

Už jsme nepřímo hovořili o třídě

$$\mathsf{PTIME} = \bigcup_{k=0}^{\infty} \mathcal{T}(n^k)$$

obsahující všechny problémy řešitelné polynomiálními algoritmy. Zmínili jsme také, proč se třída PTIME považuje za třídu *zvládnutelných problémů*.
Teď ještě alespoň zaznamenejme

robustnost třídy PTIME,

což znamená nezávislost na zvoleném (rozumném) referenčním modelu počítačů. Když bychom totiž jako referenční model vzali např. Turingovy stroje, třída PTIME by se nezměnila.

Kontrolní otázka: Jak byste přímočaře definovali, co je to časová složitost T_M Turingova stroje M ?

Podíváme-li se totiž znovu na vzájemnou simulaci RAMů a Turingových strojů, kterou jsme diskutovali dříve, zjistíme mj., že počet kroků Turingova stroje M' simulujícího RAM M je (jen) polynomiálně větší než počet kroků M; přesněji řečeno, pro časové složitosti strojů M a M' platí: $T_{M'}(n) \in O((T_M(n))^c)$ pro (malé) pevné c. Říkáme také, že RAMy a Turingovy stroje jsou *polynomiálně ekvivalentní*, neboli simulují se navzájem „s nejvýš polynomiální ztrátou".

Mluvili jsme o tom, že PTIME je robustní z hlediska všech „rozumných" referenčních modelů. Obecně se rozumnými modely myslí ty, u nichž analýza algoritmů (v nich „naprogramovaných") dává realistické výsledky pro praxi (alespoň na úrovni naší „hrubé" analýzy řádového růstu složitosti). Technicky lze definovat jako rozumné ty modely, jež jsou polynomiálně ekvivalentní modelu RAM (myslí se samozřejmě s logaritmickou mírou, jak bylo dohodnuto v úmluvě za definicí 7.8). V literatuře se ovšem často v uvedené definici „rozumnosti" odkazuje k historicky prvnímu modelu, tedy k Turingovým strojům.

Shrnutí: Připomněli jsme si, že známe nejrůznější polynomiální algoritmy používané v praxi a víme, že otázka polynomiální složitosti je většinou první otázkou kladenou při zvažování řešení konkrétního problému. Rozumíme pojmu „třída složitosti" (jejími prvky jsou problémy) a chápeme také vlastnost robustnosti třídy PTIME vzhledem k referenčním výpočetním modelům.

7.4 Nedeterministické polynomiální algoritmy, třída NPTIME

Orientační čas ke studiu této části: 2 hod.

Cíle této části:

V této části si máte důkladně promyslet pojem nedeterministického polynomiálního algoritmu a definici třídy NPTIME. Máte se také naučit prokazovat příslušnost konkrétních problémů k NPTIME.

Klíčová slova: *nedeterministický polynomiální algoritmus, třída* NPTIME*, robustnost třídy* NPTIME*, třída* coNP

Vzpomeňme si na problém SAT:

NÁZEV: SAT (*problém splnitelnosti booleovských formulí*)

VSTUP: Booleovská formule v konjunktivní normální formě.

OTÁZKA: Je daná formule splnitelná (tj. existuje pravdivostní ohodnocení proměnných, při kterém je formule pravdivá)?

To je příklad problému, pro nějž se zatím nikomu nepodařilo nalézt polynomiální algoritmus ani prokázat, že takový algoritmus neexistuje.

Kontrolní otázka: Navrhnete rychle (nejvýš) exponenciální algoritmus pro SAT, tedy algoritmus se složitostí $O(2^n)$?

Jistě vás napadl algoritmus následujícího typu:

Vstup: formule F s proměnnými $x_1, x_2, \ldots, x_m$,

Postup:

---

systematicky generuj všechna pravdivostní ohodnocení
$val : \{x_1, x_2, \ldots, x_m\} \to \{0, 1\}$

a pro každé z nich ověř, zda $F(x_1, x_2, \ldots, x_m)$ je při dosazení hodnot $val(x_i)$ za x_i $(i = 1, 2, \ldots, m)$ pravdivá; když je pravdivá, skonči s odpovědí ANO.

(Když jsi prošel všech 2^m ohodnocení a pro žádné nebyla formule pravdivá, tak) skonči s odpovědí NE.

Všimněme si, že ověření pravdivosti F pro konkrétní ohodnocení lze provést rychle, tj. polynomiálně; dokonce v čase $O(n)$. Exponencialita algoritmu spočívá v tom, že je exponenciálně mnoho případů (zde pravdivostních ohodnocení), které algoritmus ověřuje.

Podobný jev můžeme pozorovat např. u problému

NÁZEV: IS (*problém nezávislé množiny [Independent Set] (také nazývaný problém anti-kliky [anti-clique])*)

VSTUP: Neorientovaný graf G (o n vrcholech); číslo k ($k \leq n$).

OTÁZKA: Existuje v G nezávislá množina velikosti k (tj. množina k vrcholů, z nichž žádné dva nejsou spojeny hranou)?

(Jedná se o ANO/NE verzi optimalizačního problému.)

Zde totiž můžeme procházet (generovat) všechny k-prvkové podmnožiny množiny vrcholů grafu G a u každé z nich rychle ověřit, zda je nezávislá.

Ukázalo se, že mnoho přirozených (ANO/NE-verzí) praktických problémů, u nichž neznáme polynomiální algoritmus, má podobnou vlastnost: ke každému vstupu existuje (lze generovat) hodně (přesněji $O(2^{n^c})$) „potenciálních řešení“, přičemž pro každé z nich se dá rychle zjistit, zda se jedná o „skutečné řešení“. V případě, že existuje alespoň jedno „skutečné řešení“, je odpověd na vstup problému ANO, v případě, že „skutečné řešení“ neexistuje, je odpověď NE.

To vedlo k zavedení pojmu *nedeterministického polynomiálního algoritmu*. Pojem „nedeterministický algoritmus“ nás po zkušenostech s nedeterministickými konečnými automaty a nedeterministickými zásobníkovými automaty jistě nepřekvapí.

Kontrolní otázka: Jak byste definovali nedeterministický Turingův stroj?

(Povolíme prostě více instrukcí se stejnou levou stranou $(q, a) \rightarrow \ldots$. Výpočtů pro jeden vstup tak může být více.)

Na úrovni vyššího programovacího jazyka si můžeme představit zavedení speciální příkazové konstrukce „nedeterministického výběru“: com_1 [] com_2 znamená, že se nedeterministicky zvolí buď provedení příkazu com_1 nebo com_2.

Uveďme příklad nedeterministického algoritmu pro problém SAT:

vstup: formule F s proměnnými $x_1, x_2, \ldots, x_m$

$x_1 := 0$ [] $x_1 := 1$
$x_2 := 0$ [] $x_2 := 1$
...
$x_m := 0$ [] $x_m := 1$

vyhodnoť $F(x_1, x_2, \ldots, x_m)$; když 1 (true), vydej ANO, když 0 (false), vydej NE.

Všimněme si, že když je vstupní formule splnitelná, existuje *alespoň jeden výpočet* uvedeného nedeterministického algoritmu, který *vydá* ANO. Není-li formule splnitelná, pak *každý výpočet vydá* NE. Můžeme si představit, že úspěšný výpočet spočívá v „uhodnutí" skutečného řešení a jeho následném ověření.
Tento nedeterministický algoritmus tedy „rozhoduje" problém SAT, ve smyslu následující definice. (Tato definice nás po zkušenostech s definicí přijímání slov nedeterministickými konečnými a zásobníkovými automaty nijak nepřekvapí.)

Definice 7.9
Nedeterministický algoritmus A „rozhoduje ANO/NE problém" P, jestliže:

- Pro vstup problému P, na nějž je odpověď ANO, *alespoň jeden výpočet* nedeterministického algoritmu A *vydá* ANO.
- Pro vstup problému P, na nějž je odpověď NE, *každý výpočet* nedeterministického algoritmu A *vydá* NE.

(Pro úplnost můžeme dodat podmínku, že každý výpočet skončí vydáním ANO nebo NE.)

V souvislosti s uvedeným nedeterministickým algoritmem pro SAT jsme také zmínili polynomialitu. Vztahuje se to samozřejmě ke složitosti, kterou u nedeterministických algoritmů pro každý vstup určuje nejdelší možný výpočet. (Čili zase používáme přístup „podle nejhoršího případu".)

Definice 7.10
Časovou složitostí nedeterministického algoritmu A (implementovaného ve zvoleném nedeterministickém referenčním modelu) rozumíme funkci $T_A : \mathbb{N} \to \mathbb{N}$, kde $T_A(n)$ je délkou nejdelšího výpočtu A pro vstup velikosti n.

Nedeterministický algoritmus nazveme *polynomiální*, jestliže jeho časová složitost je $O(n^c)$ pro nějaké c.

Třída

NPTIME

je třídou všech ANO/NE problémů, které jsou rozhodovány nedeterministickými polynomiálními algoritmy.

Poznámka: Ani jsme se explicitně nezmiňovali, ke kterému referenčnímu modelu se odvoláváme, když hovoříme o složitosti nedeterministických algoritmů. To ovšem nevadí, protože NPTIME má stejnou vlastnost *robustnosti*, o níž jsme hovořili u PTIME.

Kontrolní otázka: Umíte prokázat příslušnost problému IS (nezávislá množina) k NPTIME?

(Algoritmus nejprve (sérií nedeterministických výběrů) zvolí k vrcholů z grafu G a poté ověří, zda tyto vrcholy tvoří nezávislou množinu.)

Uveďme si teď příklady problémů, které jsou v NPTIME, spolu s náznakem myšlenek příslušných algoritmů.

NÁZEV: *Složenost čísla*

VSTUP: Přirozené číslo ℓ.

OTÁZKA: Je číslo ℓ složené?

Algoritmus nedeterministicky zvolí číslo x takové, že $1 < x < \ell$, a poté ověří, zda $\ell \bmod x = 0$. Tj. algoritmus nedeterministicky hádá netriviální dělitel čísla ℓ.

NÁZEV: CG (*Barvení grafu*)

VSTUP: Neorientovaný graf G a číslo k.

OTÁZKA: Je možné graf G obarvit k barvami (tj. existuje přiřazení barev vrcholům tak, aby žádné dva sousední vrcholy nebyly obarveny stejnou barvou)?

V tomto případě algoritmus nejprve nedeterministicky zvolí přiřazení barev jednotlivým vrcholům a poté ověří, zda se jedná o korektní obarvení.

NÁZEV: *Isomorfismus grafů*

VSTUP: Dva neorientované grafy G a H.

OTÁZKA: Jsou grafy G a H isomorfní?

Algoritmus nedeterministicky zvolí bijekci mezi vrcholy zadaných grafů a ověří, zda se jedná o isomorfimus.

NÁZEV: HK (*problém hamiltonovské kružnice*)

VSTUP: Neorientovaný graf G.

OTÁZKA: Existuje v G hamiltonovská kružnice (tj. uzavřená cesta, procházející každým vrcholem právě jednou)?

Algoritmus nedeterministicky zvolí množinu k hran, kde k je počet vrcholů G, a pak ověří, zda se jedná o kružnici, kde každý vrchol je navštíven právě jednou.
Podobně se prokáže příslušnost k NPTIME u následujícího problému.

NÁZEV: HC (*problém hamiltonovského cyklu*)

VSTUP: Orientovaný graf G.

OTÁZKA: Existuje v G hamiltonovský cyklus (tj. uzavřená cesta, procházející každým vrcholem právě jednou)?

NÁZEV: *Subset-Sum*

VSTUP: Multimnožina přirozených čísel $M = \{x_1, x_2, \ldots, x_n\}$ a přirozené číslo s.

OTÁZKA: Existuje podmnožina multimnožiny M, která dává součet s?

Algoritmus nedeterministicky zvolí podmnožinu multimnožiny M a ověří, zda součet čísel v této multimnožině je s.

Otázky:

Otázka 7.14: Proč se v definici třídy NP (to je zkratka pro NPTIME) omezujeme jen na rozhodovací (tedy ANO/NE) problémy?

Otázka 7.15: Je pravda, že když ANO/NE-problém P patří do PTIME, tak doplňkový problém $\overline{P}$ také patří do PTIME?

Otázka 7.16: Plyne z definice NPTIME, že když ANO/NE-problém P patří do NPTIME, tak doplňkový problém $\overline{P}$ také patří do NPTIME?

Poznámka: Doplňkové problémy k problémům z NP tvoří (duální) třídu, která se označuje coNP. (Vypadá to, že se jedná o různé třídy, ale dokázáno to není.)

Shrnutí: Ujasnili jsme si pojem nedeterministického polynomiálního algoritmu; nedeterminismus nám nedělal velké potíže díky předchozím znalostem nedeterministických automatů. Umíme také rozpoznávat příslušnost jednotlivých problémů k NPTIME.

7.5 Polynomiální převeditelnost, NP-úplné problémy

Orientační čas ke studiu této části: 2 hod.

Cíle této části:

Máte porozumět definici polynomiální převeditelnosti mezi problémy, definici NP-úplných problémů a způsobu prokázování NP-obtížnosti problémů. Máte pochopit specifikace konkrétních NP-úplných problémů.

Klíčová slova: *polynomiální převeditelnost,* NP*-těžké a* NP*-úplné problémy, Cookova věta*

Je PTIME =NPTIME?

Kontrolní otázka: Je vám jasné, že PTIME $\subseteq$ NPTIME?

(Jistě ano. Je nám jasné, že technicky vzato je deterministický algoritmus speciálním případem nedeterministického, jen prostě nevyužívá nedeterministický výběr.)

Otázka, zda je uvedená inkluze vlastní, tj. zda PTIME $\subsetneq$ NPTIME nebo zda PTIME = NPTIME, je jedním z největších otevřených problémů teoretické informatiky (i matematiky obecně).
Většina odborníků se přiklání k první možnosti, tj. že existují problémy v NPTIME, které není možné rozhodovat deterministickými polynomiálními algoritmy; dosud se to však nepodařilo dokázat.

Když se v praxi začaly objevovat problémy, o nichž se zjistilo, že jsou v NPTIME, ale nikdo o nich neuměl zjistit, zda jsou v PTIME, inspirovalo to široký výzkum v této oblasti. Jedno z vyústění tohoto zkoumání bylo zavedení definice tzv. NP-*úplných problémů*, v jistém smyslu „nejtěžších problémů v NPTIME “.
K uvedení definice nejdříve potřebujeme pojem *polynomiální převeditelnosti* mezi problémy. Je to speciální případ algoritmické převeditelnosti (jak ji známe z úvodu do teorie vyčíslitelnosti); požadujeme jen navíc, aby převádějící algoritmus byl polynomiální.

Polynomiální převeditelnost

Definice 7.11
Mějme ANO/NE problémy P_1, P_2. Řekneme, že problém P_1 je *polynomiálně převeditelný* na problém P_2, což označujeme

$$P_1 \lhd P_2,$$

jestliže existuje (převádějící) *polynomiální* algoritmus A, který pro libovolný vstup w problému P_1 sestrojí vstup problému P_2, označme jej $A(w)$, přičemž platí, že odpověď na otázku problému P_1 pro vstup w je ANO právě tehdy, když odpověď na otázku problému P_2 pro vstup $A(w)$ je ANO.

Kontrolní otázka: Umíte ukázat tranzitivitu relace $\lhd$, tedy, že z $P_1 \lhd P_2 \lhd P_3$ plyne $P_1 \lhd P_3$?

Základem je uvědomit si, že v polynomiálním čase vyrobí převádějící algoritmus výstup (jen) polynomiální velikosti a že složení dvou polynomů je zase polynom (byť vyššího stupně). Když tedy $P_1 \triangleleft P_2$ díky algoritmu A_1 se složitostí $O(n^{c_1})$ a $P_2 \triangleleft P_3$ díky algoritmu A_2 se složitostí $O(n^{c_2})$ tak (sekvenčním) spojením algoritmů A_1, A_2 vznikne algoritmus A se složitostí $O(n^{c_1 c_2})$, který prokazuje $P_1 \triangleleft P_3$.

NP-úplnost

Definice 7.12
Problém Q nazveme NP-*těžkým*, pokud každý problém ve třídě NP lze na problém Q polynomiálně převést, tedy pokud platí $\forall P \in \mathsf{NPTIME} : P \triangleleft Q$.
Problém Q nazveme NP-*úplným*, pokud je NP-těžký a náleží do třídy NP.

Z této definice je zřejmé, že pokud bychom nalezli (deterministický) polynomiální algoritmus rozhodující některý (kterýkoliv) NP-těžký problém, znamenalo by to, že existuje (deterministický) polynomiální algoritmus pro každý problém v NP, tedy že PTIME =NPTIME. Proto není radno se pokoušet o nalezení polynomiálního algoritmu pro problém, o kterém se ví, že je NP-těžký. Zatím se taková věc nepodařila ani těm nejchytřejším výzkumníkům.

O tom, jak se prokazuje příslušnost k NPTIME, již víme. Jak se ale prokazuje NP-obtížnost konkrétních problémů? Podobně jako v případě nerozhodnutelnosti. Tam se ukázal jako základní nerozhodnutelný problém „halting problem“ HP (či DHP) a nerozhodnutelnost dalších se ukazuje hlavně pomocí (algoritmické) převeditelnosti.
Cook (1971) ukázal „základní NP-těžký problém“, který je ovšem také v NP a je tedy NP-úplný:

Věta 7.13 (Cook)
Problém SAT je NP-*úplný.*

Dále už slouží polynomiální převeditelnost:

Tvrzení 7.14
Jestliže $P_1 \triangleleft P_2$ a P_1 je NP-*těžký, pak P_2 je rovněž* NP-*těžký; když je navíc P_2 v* NPTIME*, je* NP-*úplný.*

Upozornění na animaci.
Na web-stránce předmětu najdete např. animaci prokazující SAT $\triangleleft$ IS. (Ve

skutečnosti je tam ukázáno 3-SAT $\triangleleft$ IS, kde 3-SAT je verze problému SAT, která je také NP-úplná, jak se ještě zmíníme později.)

Důsledek: Problém IS je NP-úplný.

Bez důkazů jen zaznamenáme NP-úplnost dalších problémů.

Tvrzení 7.15
Problémy jsou CG, HK, HC, Subset-Sum jsou NP*-úplné.*

Poznamejme, že o problému „Složenost čísla“ se od r. 2002 ví, že je v PTIME (což se spíš formuluje pro doplňkový problém: prvočíselnost je v PTIME). O problému „Izomorfismus grafů“ se neví, zda je v PTIME, ale ani se neví, zda je NP-úplný.

Uveďme ještě několik příkladů NP-úplných problémů.

NÁZEV: TSP (*problém obchodního cestujícího (*ANO/NE *verze)*)

VSTUP: množina „měst“ $\{1, 2, \ldots, n\}$, přir. čísla („vzdálenosti“) d_{ij} ($i = 1, 2, \ldots, n$, $j = 1, 2, \ldots, n$); dále číslo ℓ („limit“).

OTÁZKA: existuje „okružní jízda“ dlouhá nejvýše ℓ, tj. existuje permutace $\{i_1, i_2, \ldots, i_n\}$ množiny $\{1, 2, \ldots, n\}$ tž. $d(i_1, i_2) + d(i_2, i_3) + \ldots + d(i_{n-1}, i_n) + d(i_n, i_1) \leq \ell$?

NÁZEV: 3-SAT (*problém SAT s omezením na 3 literály*)

VSTUP: Booleovská formule v konjunktivní normální formě, kde v každé klauzuli (tj. v každém konjunktu) jsou právě 3 literály (literál je buď proměnná nebo její negace).

OTÁZKA: Je daná formule splnitelná (tj. existuje pravdivostní ohodnocení proměnných, při kterém je formule pravdivá)?

NÁZEV: 3-CG (*problém barvení grafu třemi barvami*)

VSTUP: Neorientovaný graf $G = (V, E)$.

OTÁZKA: Lze G obarvit třemi barvami, tzn. existuje zobrazení $c : V \to \{col_1, col_2, col_3\}$ takové, že $\forall\{v_1, v_2\} \in E : c(v_1) \neq c(v_2)$?

Poznámka: Problém 3-SAT je speciálním případem obecnějšího problému SAT a podobně 3-CG je speciálním případem problému CG. Ukazuje se, že tyto problémy zůstávají NP-úplné, i když se omezíme jen na instance určitého konkrétního typu. To může být výhodné při hledání vhodných redukcí (tj. převeditelností) při dokazování NP-obtížnosti dalších problémů. Zejména problém 3-SAT je k tomuto účelu často využíván.

NÁZEV: ILP (*problém celočíselného lineárního programování*)

VSTUP: Matice A typu $m \times n$ a sloupcový vektor b velikosti m, jejichž prvky jsou celá čísla.

OTÁZKA: Existuje celočíselný sloupcový vektor x (velikosti n) tž. $Ax \geq b$?

Jedná se rovněž o NP-úplný problém. Snadno se ukáže, že je NP-těžký (např. převodem 3-SAT $\triangleleft$ ILP), ale na rozdíl od dříve uvedených problémů je obtížnější prokázat, že ILP $\in$ NPTIME. Zhruba řečeno, dá se ukázat, že pokud řešení nerovnosti $Ax \geq b$ existuje, existuje i řešení „dostatečně malé" – jeho zápis je polynomiální vzhledem k zápisu A a b; řešení se tedy dá v polynomiálním čase „uhodnout" a ověřit.

CVIČENÍ 7.17: Když chceme např. ukázat, že IS$\triangleleft$HC, pomocí (převádějícího) algoritmu A, co musí A konkrétně splňovat?

CVIČENÍ 7.18: Promyslete si předchozí otázku pro další dvojice dříve uvedených problémů.

Shrnutí: Perfektně jsme si promysleli polynomiální převeditelnost mezi problémy, a její využití v definici a prokazování NP-obtížnosti (a NP-úplnosti). Známe dobře příklady NP-úplných problémů.

Kapitola 8

Úvod do teorie složitosti - rozšiřující část

Orientační čas ke studiu této části: 1 hod.

Cíle kapitoly:

Cílem je přidat několik poznatků k složitosti problémů.

Klíčová slova: *třída* PSPACE*, Savitchova věta (implikující* PSPACE =NPSPACE*),* PSPACE*-úplné problémy, problém kvantifikovaných booleovských formulí (QBF), dokazatelně nezvládnutelné problémy,* EXPSPACE*-úplné problémy, superexponenciální problémy (např. rozhodování pravdivosti v Presburgerově aritmetice)*

Třída PSPACE

Předmětem našeho prvořadého zájmu je *časová* složitost algoritmů a problémů. Už v případě konkrétních algoritmů a problémů ovšem může mít dobrý smysl zkoumat také *prostorovou* (neboli *paměťovou*) složitost a speciálně vztah časové a prostorové složitosti (daný algoritmus může jít např. zrychlit jen za cenu zvýšení paměťové náročnosti a naopak).

Zkoumají se samozřejmě i třídy problémů vymezené prostorovou složitostí. Čtenář si jistě snadno doplní definice tříd PSPACE a NPSPACE a všimne si zřejmé inkluze $\mathsf{PSPACE} \subseteq \mathsf{NPSPACE}$. Pro tyto třídy se ovšem ví, že platí i inkluze obrácená (a je tedy $\mathsf{PSPACE} = \mathsf{NPSPACE}$); to ihned plyne z následující věty (jen poznamenejme, že věta platí obecněji—pro naše účely však postačuje uvedené znění):

Věta 8.1 (Savitch, 1970)
Je-li problém P rozhodován nedeterministickým Turingovým strojem s prostorovou složitostí $O(n^k)$, pak je také rozhodován deterministickým Turingovým strojem s prostorovou složitostí $O(n^{2k})$.

Všimněme si, že pro libovolnou funkci f je $\mathcal{T}(f(n)) \subseteq \mathcal{S}(f(n))$ (např. Turingův stroj očividně navštíví při výpočtu nejvýše tolik políček, kolik udělá kroků). Měl by teď už být zřejmý vztah

$$\mathsf{PTIME} \subseteq \mathsf{NPTIME} \subseteq \mathsf{PSPACE} = \mathsf{NPSPACE}.$$

Přes velké úsilí vědecké komunity, nemůžeme dosud vyloučit nejen možnost $\mathsf{PTIME} = \mathsf{NPTIME}$, ale dokonce ani $\mathsf{PTIME} = \mathsf{PSPACE}$, byť se tyto možnosti jeví velmi „nepravděpodobnými“.

Podobně jako u NP-úplnosti, lze definovat tzv. PSPACE-úplné problémy; definici napíšeme obecněji:

Definice 8.2
O problému P řekneme, že je *$\mathcal{C}$-těžký*, kde $\mathcal{C}$ je nějaká třída problémů, jestliže pro každý $P' \in \mathcal{C}$ platí $P' \triangleleft P$. Je-li navíc $P \in \mathcal{C}$, říkáme, že P je *$\mathcal{C}$-úplný*.

Známým příkladem PSPACE-úplného problému je problém

NÁZEV: QBF (*problém pravdivosti kvantifikovaných booleovských formulí*)

VSTUP: formule $(\exists x_1)(\forall x_2)(\exists x_3)(\forall x_4)\ldots(\exists x_{2n-1})(\forall x_{2n})\mathcal{F}(x_1, x_2, \ldots, x_{2n})$, kde $\mathcal{F}(x_1, x_2, \ldots, x_{2n})$ je booleovská formule v konjunktivní normální formě.

OTÁZKA: je daná formule pravdivá?

Poznámka: Problém QBF je někdy označován i zkratkou Q-SAT pro zdůraznění vztahu k problému SAT.

Pravidelné střídání existenčních a univerzálních kvantifikátorů samozřejmě není nutné; každá kvantifikovaná formule se ovšem dá převést do uvedené formy.

QBF často slouží jako jakýsi výchozí PSPACE-úplný problém. PSPACE-úplnost (či PSPACE-obtížnost) dalších problémů se pak dokazuje využitím polynomiální převeditelnosti, jak to známe u třídy NP.

Zaznamenejme alespoň, že

> problém ekvivalence *nedeterministických* konečných automatů (a podobně problém ekvivalence regulárních výrazů)

je PSPACE-úplný problém.

Dokazatelně nezvládnutelné problémy

Jestliže prokážeme NP-obtížnost či PSPACE-obtížnost nějakého problému, říkáme, že je *prakticky nezvládnutelný* (intractable). Je totiž jasné, že navrhneme-li algoritmus, který řeší (přesně ten) daný problém, a neobjevíme-li přitom geniální „trik“, na který dosud nikdo nepřišel, bude mít algoritmus zřejmě exponenciální (či ještě horší) složitost. Obecně používat algoritmus (resp. odpovídající program) budeme moci jen na velmi malá vstupní data. Teoreticky ovšem pořád ještě možnost rychlého algoritmu (založeného na „geniálním triku“) existuje.

Poznámka: Samozřejmě je možné, že exponenciální algoritmus ve skutečnosti chceme použít jen na malá data, anebo ho třeba používáme na data, při nichž se neprojeví ona (worst case) exponenciální složitost. V tomto smyslu praktická nezvládnutelnost (obecného) problému ještě neznamená, že ho v praxi nemůže počítačový program úspěšně řešit v pro nás zajímavých případech. (Existují i jiné možnosti, jak v praxi úspěšně řešit i „nezvládnutelný“ problém. Zmíníme se o tom v partiích o aproximačních a pravděpodobnostních algoritmech.)

Známe ovšem i tzv. *dokazatelně nezvládnutelné* problémy (provably intractable problems), tj. ty, u nichž máme *dokázáno*, že pro ně neexistují polynomiální algoritmy. Definujeme-li např. třídy

$$\mathsf{EXPTIME} = \bigcup_{k=0}^{\infty} \mathcal{T}(2^{n^k}) \,, \qquad \mathsf{EXPSPACE} = \bigcup_{k=0}^{\infty} \mathcal{S}(2^{n^k})$$

pak EXPTIME-těžký či EXPSPACE-těžký problém je takovým dokazatelně nezvládnutelným problémem.

Dá se totiž ukázat, že inkluze $\mathsf{PTIME} \subset \mathsf{EXPTIME}$, $\mathsf{PSPACE} \subset \mathsf{EXPSPACE}$ jsou skutečně vlastní (tzn., že neplatí rovnost). (My to zde ale dokazovat nebudeme.)

Např. následující problém je EXPSPACE-úplný:

NÁZEV: RE^2 (*ekvivalence regulárních výrazů s mocněním*)

VSTUP: dva regulární výrazy, v nichž je možné použít mocnění (tzn. je možno psát α^2 místo $\alpha \cdot \alpha$).

OTÁZKA: reprezentují zadané výrazy tentýž jazyk?

Poznamenejme, že stejný problém pro standardní regulární výrazy (bez mocnění) je PSPACE-úplný. (Připomeňme si, že složitost je funkcí velikosti vstupu; mocnění v regulárních výrazech umožňuje exponenciálně zkrátit některé výrazy bez mocnění.) Čtenář by si měl být schopen vyvodit, že problém ekvivalence dvou *nedeterministických* konečných automatů je tedy také PSPACE-úplný. (Pro deterministické konečné automaty ovšem samozřejmě známe polynomiální algoritmus; připomeňme si, že přechodem od nedeterministického automatu k deterministickému se obecně nemůžeme vyhnout exponenciálnímu nárůstu počtu stavů.)

Existují samozřejmě i dokazatelně těžší (superexponenciální) problémy. Ilustrujme je příkladem problému Presburgerovy aritmetiky (rozhodování pravdivosti formulí teorie sčítání).

NÁZEV: ThAdd (*problém pravdivosti teorie sčítání*)

VSTUP: formule jazyka 1. řádu užívající jediný „nelogický“ symbol – ternární (tj. 3-ární) predikátový symbol $PLUS$ ($PLUS(x, y, z) \Leftrightarrow_{df} x + y = z$).

OTÁZKA: je daná formule pravdivá pro množinu $\mathbb{N} = \{0, 1, 2, \ldots\}$, kde $PLUS(a, b, c)$ je interpretováno jako $a + b = c$?

Zde vůbec není zřejmé, že existuje algoritmus, který daný problém řeší. Presburger ukázal takový algoritmus ve 20. létech 20. století (jedním z cílů tzv. Hilbertova programu bylo ukázat podobný algoritmus i s připuštěním predikátu pro násobení – nemožnost řešení tohoto úkolu ukázal později Gödel). Mnohem později bylo ukázáno, že každý algoritmus, řešící problém ThAdd má složitost minimálně 2^{2^n}.

Presburger ukázal algoritmus využitím tzv. metody eliminace kvantifikátorů. Elegantní důkaz umožňují také výsledky, které známe z teorie konečných automatů.

Věta 8.3
Existuje algoritmus rozhodující problém ThAdd.

Důkaz: (Idea.) (Pro hloubavé čtenáře.) Představme si třístopou pásku, kde v každé stopě je řetězec nul a jedniček, tj. binární zápis čísla. Na pásku samozřejmě můžeme hledět jako na jednostopou s tím, že povolené *symboly abecedy* jsou uspořádané *trojice* nul a jedniček. Snadno nahlédneme, že existuje konečný automat, který přijímá právě ta slova v „abecedě trojic“, která mají tu vlastnost, že součet čísla v první stopě s číslem v druhé stopě je roven číslu ve třetí stopě. Ihned je to jasné při čtení odzadu; pak si stačí připomenout, že regulární jazyky jsou uzavřeny na zrcadlový obraz.

Z dalších uzávěrových vlastností snadno vyvodíme, že pro libovolnou formuli $\mathcal{F}(x_1, x_2, \ldots, x_n)$ jazyka ThAdd která *neobsahuje kvantifikátory*, lze zkonstruovat kon. automat $A_{\mathcal{F}}$ přijímající právě binární zápisy těch n-tic čísel (na n-stopé pásce), pro které je $\mathcal{F}(x_1, x_2, \ldots, x_n)$ pravdivá.

Pro formuli $(\exists x_n)\mathcal{F}(x_1, x_2, \ldots, x_n)$ je možné zkonstruovat automat, který přijímá právě binární zápisy těch $(n-1)$-tic čísel (dosazených za $x_1, x_2, \ldots, x_{n-1}$), pro které je $(\exists x_n)\mathcal{F}(x_1, x_2, \ldots, x_n)$ pravdivá: automat pracuje na $(n-1)$-stopé pásce, ale simuluje $A_{\mathcal{F}}$ tak, že obsah n-té stopy nedeterministicky hádá! (Pak ho samozřejmě lze převést na ekvivalentní deterministický automat.)

Jelikož $(\forall x_n)\mathcal{F}(x_1, x_2, \ldots, x_n)$ je ekvivalentní $\neg(\exists x_n)\neg\mathcal{F}(x_1, x_2, \ldots, x_n)$, načrtli jsme takto postup, který k formuli jazyka ThAdd v prenexní formě postupnou aplikací zmíněných konstrukcí sestrojí konečný automat, který přijme prázdné slovo právě tehdy, když výchozí formule je pravdivá. □

Příloha A

Řešení příkladů

Cvičení 1.1:

a) $aaa, aab, aba, abb, baa, bab, bba, bbb$

b) slovo v je pochopitelně bráno jako celek, tedy výsledek dosazení $v = ab$ do výrazu musíme psát $(ab)^3 \cdot ba \cdot (bba)^2$ (nikoliv $ab^3 \cdot ba \cdot (bba)^2$); správný výsledek je tedy $ababbabbabba$

c) $00, 01, 10$

d) $\varepsilon, 0, 00, 001, 0010$

e) $\varepsilon, 0, 10, 010, 0010$

Cvičení 1.2: Je důležité si nejprve uvědomit, že slovo neobsahující aa uvedenou podmínku automaticky splňuje; nesplňují ji totiž jen ta slova, která obsahují nějaký výskyt podslova aa, za nímž nenásleduje b (tedy následuje a nebo nenásleduje nic, jelikož se jedná o konec slova). Správnou odpovědí je tedy seznam $\varepsilon, a, b, ab, ba, bb, aab, aba, abb, bab$.

Otázka 1.3: Ne, není totiž konečná, což je důležitá podmínka.

Otázka 1.4: Ano, přeneseně, třeba jako jazyk všech neprázdných slov nad abecedou $\{0, 1, \ldots, 9\}$; pro jednoznačnost můžeme případně vyloučit slova začínající nulami, s výjimkou jednoznakového slova 0.

Otázka 1.5: Nelze.

Otázka 1.6: Velký – prázdný jazyk nemá v sobě žádné slovo, je to prázdná množina, kdežto prázdné slovo je slovem jako každé jiné, jen má nulovou délku.

Otázka 1.7: Jen pokud je $L = \emptyset$ či $L = \{\varepsilon\}$ (v obou případech je $L^* = \{\varepsilon\}$).

Otázka 1.8*: Ne. Podle definice každé slovo $u \in (L^*)^*$ lze rozdělit na několik částí, z nichž každá patří do L^*; tedy u lze psát $u = v_1 v_2 \dots v_n$ (pro nějaké n), kde každé v_i je z L^*. Jelikož podle definice lze každé v_i rozdělit na několik částí, z nichž každá patří do L, můžeme i původní u (rovnou) rozdělit na (menší) části, z nichž každá patří do L. Prokázali jsme tak, že každé $u \in (L^*)^*$ patří do L^*; celkově tedy $L^* = (L^*)^*$.

Cvičení 1.9: Slova ε, 10 a celé slovo 101110110.

Cvičení 1.10: $\{11001, 110000, 011101, 0111000\}$

Cvičení 1.11:

- Pro $L_1 \cup L_2$: $\varepsilon, a, b, aa, bb, aaa$
- Pro $L_1 \cap L_2$: $b, aa, bb, aba, bbb, aaaa$
- Pro $L_1 - L_2$: $aab, baa, aabb, abab, baba, bbaa$
- Pro $\overline{L_1}$: a, ab, ba, aaa, abb, bab

Cvičení 1.12: Třeba $L_1 = \{\varepsilon\}$ a $L_2 = \{1\}$.

Cvičení 1.13*: Ne, jen všechna ta slova, co nekončí lichým počtem 0.

Cvičení 1.14: Ze slov neobsahujících žádný výskyt 1 je to jen ε. Pro jediný výskyt 1 máme v průniku slova $01, 10$. Pro dva výskyty znaku 1 máme v průniku slova 0011, 0101, 0110, 1001, 1010, 1100; atd.

Cvičení 1.15: $a, b, abba, abbb, abbabba, bbaa, bbab, bbaabba$

Cvičení 1.16*: Do $L_0 \cdot L_0$ patří právě ta slova u, pro něž existuje rozdělení $u = vw$, v němž platí $|v|_c \neq |v|_d$ a $|w|_c \neq |w|_d$. Tedy u nemůže být např. tvaru $u = cdcd \dots cdc$ ani tvaru $u = dcdc \dots dcd$. Když ovšem je tvaru $u = cdcd \dots cd$ (tedy $(cd)^i$ pro $i \geq 1$) nebo tvaru $u = dcdc \dots dc$ (tedy $(dc)^i$ pro $i \geq 1$) nebo obsahuje podslovo cc či dd, pak se patřičně rozdělit dá. (Ověřte!)

Cvičení 1.17: Takové posloupnosti, ve kterých je sudý rozdíl mezi počtem stisků tlačítka A a počtem stisků tlačítka B.

Cvičení 1.18: Takové posloupnosti, ve kterých jsou počty stisků tlačítka A liché, B liché a C sudé nebo A sudé, B sudé a C liché.

Cvičení 1.19: Je to jazyk všech těch slov, v nichž každý úsek nul (který nelze prodloužit) má sudou délku a každý úsek jedniček má délku dělitelnou třemi.

Cvičení 1.20: Slova ze samých nul nebo ta slova, která mají jediný znak 1 právě uprostřed, tj. ε, 0, 00, 000, ..., 1, 010, 00100, ...

Cvičení 1.21: Třeba pro $L_1 = \{1\}$, $L_2 = \{11\}$ a $L_3 = \{1, 11\}$ vyjde $L_1 \cap L_2 = \emptyset$, ale $111 \in L_1 \cdot L_3$ i $111 \in L_2 \cdot L_3$.

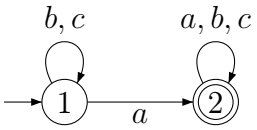
Cvičení 2.5: Můžeme uvažovat např. takto: Vidíme, že každé slovo w z $L_{q_2}^{toAcc}$, tedy splňující $q_2 \xrightarrow{w} q_2$, je buď ε nebo končí 1, tedy je tvaru $w = u1$ pro libovolné u, nebo je tvaru $w = u00$, přičemž v tomto posledním případě je $q_2 \xrightarrow{u} q_2$. To můžeme vyjádřit tak, že $L_{q_2}^{toAcc} = \{w \mid$ nejdelší sufix w, který je řetězcem nul, tedy je prvkem $\{0\}^*$, má sudou délku$\}$.
Jazyk přijímaný automatem, tedy $L_{q_1}^{toAcc}$ je pak očividně tvořen těmi slovy, která splňují uvedenou podmínku charakterizující jazyk $L_{q_2}^{toAcc}$ a zároveň obsahují alespoň jeden výskyt symbolu 1.

Otázka 2.8: Může, potom bude přijato i prázdné slovo, pro nějž výpočet skončí již v počátku.

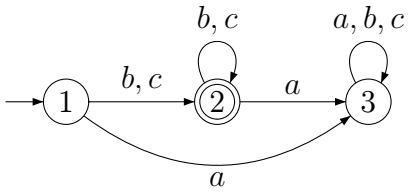
Otázka 2.9: Ano, automat nemusí mít žádný přijímající stav nebo přijímající stav nemusí být dosažitelný.

Otázka 2.10: Ano, je. Samozřejmě pro pevně dané uspořádání abecedy.

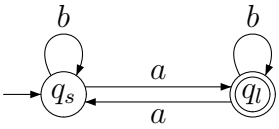
Cvičení 2.11: Existuje:



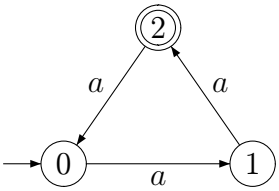
Cvičení 2.12: Ano.



Cvičení 2.13: Čtení znaku b nemění stav:



Cvičení 2.14: Počítání na cyklu délky 3:

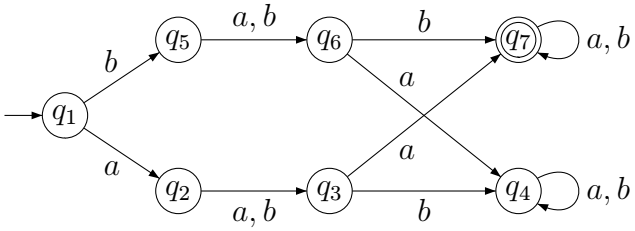


Cvičení 2.15: Právě taková, ve kterých počet výskytů znaku a mínus počet výskytů b dává zbytek 2 po dělení 3.

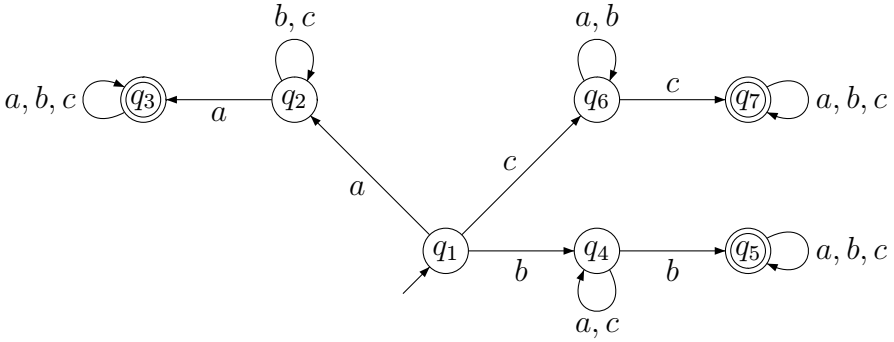
Cvičení 2.16: Všechna taková, ve kterých se vyskytuje a a po jeho prvním výskytu následuje sufix „a“, „aa“ nebo „b“ (a nic víc).

Cvičení 2.17: Prostřední a ten vpravo. Automat vlevo se dostává do přijímajícího stavu jen když délka dává zbytek 2 po dělení 3.

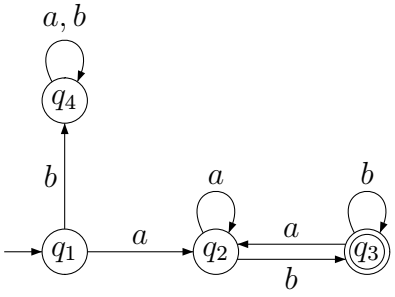
Cvičení 2.18:



Cvičení 2.19:



Cvičení 2.20:

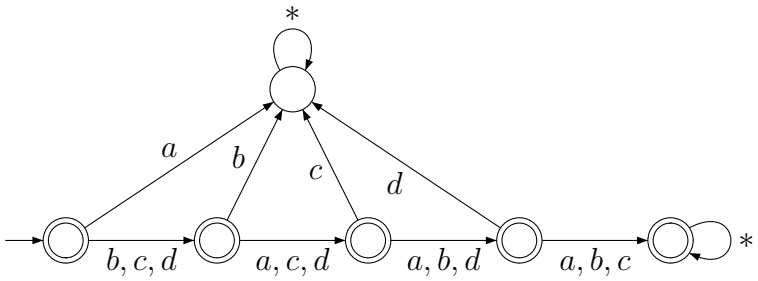


Cvičení 2.21: Ano, počítáme paritu výskytů pro a i b zvlášť podle Příkladu 2.1 a uděláme automat pro sjednocení těchto dvou jazyků.

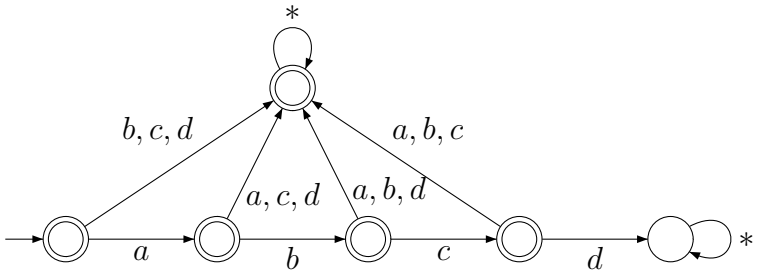
Cvičení 2.22: Ano, stačí dvoustavový automat. Také lze využít automat z předchozí úlohy, jen patřičně pozměníme přijímající stavy. Jak?

Cvičení 2.23: Ano, ale už se nám tento automat bude obtížně kreslit, protože má 102 stavů; musí „počítat“ prvních 100 znaků, a pak přijmout všechna delší slova.

Cvičení 2.24:



Cvičení 2.25:



Cvičení 2.26: Nejmenší takový KA má a) 8, b) 7 stavů.

Cvičení 2.27: Nejmenší takový KA má 5 stavů.

Otázka 2.31: Protože II vznikl sloučením stavů $4, 5$ a mezi nimi vedly přechody znakem b. Nyní se tyto dva přechody sloučí do jedné smyčky.

Otázka 2.32: Neukáže, naopak bude postup obvykle dosti dlouhý, protože bude muset rozložit množinu stavů až na jednotlivé jednoprvkové podmnožiny.

Cvičení 2.33: Ano, postupem minimalizace začneme s rozkladem $\{\{1, 3\}, \{2\}\}$ a hned v první iteraci rozlišíme stavy $1, 3$ přechodem znakem 0.

Cvičení 2.34: Ano, již po dvou iteracích dojde k rozlišení všech stavů.

Cvičení 2.36: 4 stavy, je nutno počítat paritu počtů jak a, tak b.

Cvičení 2.37: Již po dvou iteracích postupu minimalizace dojde k rozložení na jednotlivé stavy zvlášť; hledaná slova jsou tedy délky nejvýše dvě.

Cvičení 2.38: Spojit 12; 348; 567.

Cvičení 2.39: Spojit 12; 348; 5; 67.

Cvičení 2.40: 5 stavů, 2×2 jsou nutné k rozpoznávání parity počtů a a b jeden další pro odlišení prázdného slova.

Cvičení 2.41: Vyjdeme ze základního automatu na $3 \times 3 = 9$ stavech, který počítá výskyty a a b do dvou (tj. jako 0, 1, mnoho) a na vhodných místech má přijímající stavy. Tento automat pak minimalizujeme. Výsledek má 7 stavů.

Cvičení 2.42: Obdobně předchozímu 5 stavů.

Cvičení 2.43: Obdobně předchozímu 9 stavů.

Otázka 2.44: Ne, protiřečilo by to větě 2.16.

Cvičení 2.45: Samozřejmě ne, ten první nepřijímá prázdné slovo, kdežto druhý ano.

Cvičení 2.46: Ano, po minimalizaci je ten druhý stejný jako první.

Otázka 2.47: Ano, počítat počet znaků modulo daná konstanta konečný automat umí.

Otázka 2.48: Ano, takovou omezenou informaci si konečný automat snadno může zapamatovat.

Otázka 2.49: Ne, vidíme, že tady by se např. při dlouhém počátečním úseku a-ček automat bez pamatování jejich počtu neobešel. Samozřejmě zpřesnit to zase můžeme pomocí nekonečně mnoha kvocientů.

Otázka 2.50: Ano, jedná se o booleovskou kombinaci podmínek, pro něž snadno umíme sestrojit konečný automat. Je samozřejmě užitečné si připomenout, že logická formule $A \Rightarrow B$ je ekvivalentní formuli $\neg A \vee B$.

Otázka 2.51: Ne, regulární podmínka „končí baa“ nám nepomůže, kvocientů je pořád očividně nekonečně mnoho. Automat prostě musí počítat např. počáteční a-čka, nemůže se „spoléhat“, že slovo skončí baa.

Otázka 2.52: Ano; nesmíme soudit povrchně např. podle podobnosti s předchozím příkladem. Rozebráním podmínky snadno zjistíme, že si stačí pamatovat, zda počet a-ček je $0, 1$ či ≥ 2 a zda počet b-ček je $0, 1$ či ≥ 2. (Co je např. kvocient jazyka podle slova aa ?)

Otázka 2.53: Ne, např. počáteční úsek a^i by si automat musel pamatovat přesně, aby byl připraven na možný sufix bba^i. [Kvocienty podle a, aa, aaa, ... jsou zase různé.]

Otázka 2.54: Ne, opět aplikujeme podobnou úvahu jako u předchozích příkladů.

Otázka 2.55: Ano! V jednoprvkové abecedě je přece každé slovo rovno svému zrcadlovému obrazu.

Otázka 2.56: Ne. Analogickými úvahami jako výše.

Otázka 2.57: Ano. V jednoprvkové abecedě to znamená jen sudou délku slova.

Otázka 2.58: Ne. Opět jsou kvocienty podle a^i a a^j pro $i \neq j$ očividně různé.

Otázka 2.59: Ano. Očividně stačí počítat počet jednotlivých symbolů jen do 100. [To ovšem neznamená, že minimální automat má $(101)^2$ stavů; proč ne?]

Otázka 2.60*: Ne, ani jednoprvková abeceda nepomůže. Měli bychom to vytušit, i když třeba nejsme sběhlí v teorii čísel. [Co to je prvočíslo ovšem jistě všichni víme.] Už kdyby totiž dva kvocienty podle a^i, a^j pro $i < j$ měly být stejné, tak by pro každé k mělo být $i + k$ prvočíslo právě tehdy, když $j + k$ je prvočíslo. Jinými slovy, když bychom k libovolnému prvočíslu $p_0 \geq i$ přičetli $j-i$, tak bychom měli dostat prvočíslo $p_1 = p_0 + (j-i)$. Pak ovšem i $p_2 = p_1 + (j-i) = p_0 + 2(j-i)$ je prvočíslo, $p_3 = p_2 + (j-i) = p_0 + 3(j-i)$ je prvočíslo, atd. Tedy i $p_0 + p_0(j-i)$ je pak prvočíslo – to je ovšem spor, neboť $p_0 + p_0(j-i)$ můžeme vyjádřit jako součin $p_0 \cdot (j-i+1)$, kde oba členy jsou větší než 1.

Otázka 2.65: Nemusí být – definice nám povoluje se neustále přesouvat po ε-přechodech a nečíst vstup. Takový výpočet však nikdy k přijetí slova nevede, takže pro nás nemá praktický význam.

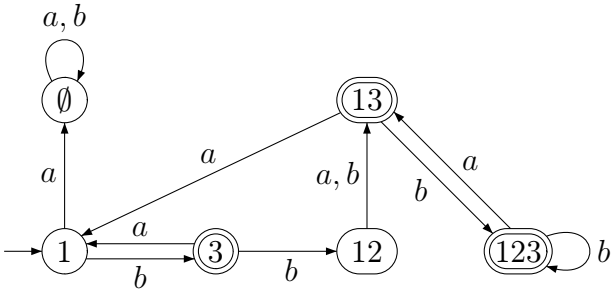
Otázka 2.66: Vznikne de facto stejný automat – vzniklé stavy budou reprezentovat jednoprvkové množiny obsahující původní stavy; metoda by ovšem odstranila případné nedosažitelné stavy.

Otázka 2.67: Právě tehdy, když se pro některou posloupnost vstupních znaků u každý příslušný výpočet „zasekne“ – tedy nepřečte celé u a skončí ve stavu, v němž není definován přechod pro následující písmeno z u ani ε-přechod.

Otázka 2.68: Pochopitelně nemůže, neobsahuje samozřejmě žádný původní přijímající stav; končí v něm všechny „zaseklé“ výpočty.

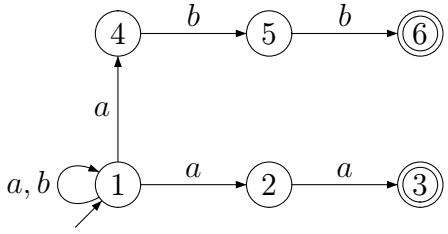
Cvičení 2.69: Nikdy, již první přechod znakem a není definován.

Cvičení 2.70: Asi jste konstruovali tabulku; ta by měla být ekvivalentní následujícímu grafu.



Cvičení 2.71: Kromě ε a „bb“ všechna. (Viz sestrojený deterministický automat.)

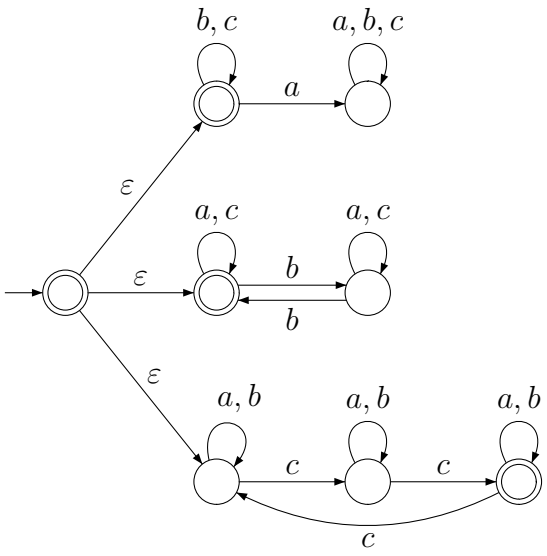
Cvičení 2.72: Přirozeně využijeme nedeterminismu:



Cvičení 2.73:

	a	b
$\leftrightarrow 1,2,3$	$1,2,3$	$1,2$
$1,2$	$2,3$	2
$\leftarrow 2,3$	$1,2,3$	1
2	$2,3$	$\emptyset$
1	2	2
$\emptyset$	$\emptyset$	$\emptyset$

Cvičení 2.74: Asi nejpřímočařejší je toto řešení:



Dokážete tento automat převést na deterministický?

Cvičení 2.75: Třeba *bab*.

Cvičení 2.76: Třeba *abb*.

Cvičení 2.77:

	a	b
$\leftrightarrow 1,3$	$1,2$	$1,2$
$1,2$	$1,2,3$	2
$\leftarrow 1,2,3$	$1,2,3$	$1,2$
2	$1,3$	$\emptyset$
$\emptyset$	$\emptyset$	$\emptyset$

Cvičení 2.78:

	a	b
$\rightarrow 1$	2	2
2	2, 3	$\emptyset$
$\leftarrow 2, 3$	1, 2, 3	1
$\emptyset$	$\emptyset$	$\emptyset$
$\leftarrow 1, 2, 3$	1, 2, 3	1, 2
1, 2	2, 3	2

Cvičení 2.79*: Slova začínají b, končí ba, opakují se $\leq 2\times$ a za sebou.

Otázka 2.81: Ne, třeba $(0+1)^*$ a $(0+00+1)^*$ označují stejný jazyk. Každý regulární jazyk lze popsat nekonečně mnoha regulárními výrazy.

Cvičení 2.82: Třeba takto $a^*(c+\varepsilon)b^*$.

Cvičení 2.83: Třeba takto $aa^*(c+\varepsilon)b^*b$.

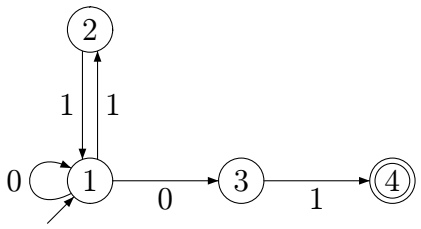
Cvičení 2.84: Třeba takto $aa^*(cc+c+\varepsilon)b^*b$.

Cvičení 2.85: Nejsou, třeba první jazyk obsahuje prázdné slovo, kdežto druhý ne.

Cvičení 2.86: Nejsou, třeba slova v prvním jazyku mohou končit znakem 1, kdežto v druhém jazyku ne.

Cvičení 2.87*: Např. $(00+100+010+1010)^*$

Cvičení 2.89: Zde:



Cvičení 2.90: Jen přidáme smyčku s 0 nad stav 3.

Cvičení 2.91*: $(\varepsilon + 1 + 11)(01 + 011 + 001 + 0011)^*(\varepsilon + 0 + 00)$

Cvičení 2.92: $(((b + c)^* + (a + c)^*)c)^*(b^* + a^*)$

Cvičení 2.93: $((b + c + a(b + c))^*(\varepsilon + a)$

Cvičení 2.94: c^*

Cvičení 2.95: $c^*(abb^* + b^*)^*$

Cvičení 2.96*: $((c+\varepsilon)(a+b)(a+b)^*)^*(c+\varepsilon)\, aa\, ((c+\varepsilon)(a+b)(a+b)^*)^*(c+\varepsilon)$

Cvičení 2.97*:

a) Nejkratší je ε a nejdelší 01100110, neboť jazyk L nedovoluje opakovat stejný znak za sebou více než dvakrát.

b) Protože $1 \in K - L$ a $010101 \in L - K$.

c) 10101 jednoznačně.

Otázka 3.2: Ano, třeba gramatika $S \longrightarrow SS \mid \varepsilon$ generuje jen prázdné slovo, ale to lze odvodit nekonečně mnoha derivacemi.

Otázka 3.3: Prázdný, protože z neterminálu S nevygenerujeme žádné terminální slovo.

Cvičení 3.4:

- $$\begin{array}{lcl} S & \longrightarrow & AaB \mid \varepsilon \\ A & \longrightarrow & AA \mid aSb \mid \varepsilon \mid SB \\ B & \longrightarrow & SA \mid ac \end{array}$$

- $S \Rightarrow AaB \Rightarrow AAaB \Rightarrow aSbAaB \Rightarrow abAaB \Rightarrow abaB \Rightarrow abaSA \Rightarrow abaAaBA \Rightarrow abaaBA \Rightarrow abaaacA \Rightarrow abaaacSB \Rightarrow abaaacB \Rightarrow abaaacac$
- $S \Rightarrow AaB \Rightarrow AaSA \Rightarrow AaSSB \Rightarrow AaSSac \Rightarrow AaSac \Rightarrow AaAaBac \Rightarrow AaAaacac \Rightarrow Aaaacac \Rightarrow AAaaacac \Rightarrow Aaaacac \Rightarrow aSbaaacac \Rightarrow abaaacac$

Cvičení 3.5: $S \longrightarrow \varepsilon \mid aSa \mid bSb$.

Cvičení 3.6: Žádná, v odvození z S se nikdy nezbavíme výskytu neterminálu S nebo C.

Cvičení 3.7: Ne, druhá generuje třeba slovo „*abba*“, které v první odvodit nelze.

Cvičení 3.8:

- G_1 : $S \longrightarrow AbaabA, \qquad A \longrightarrow \varepsilon \mid aA \mid bA$
- G_2 : $S \longrightarrow A \mid AbAbAbS, \qquad A \longrightarrow \varepsilon \mid aA$
- G_3 : $S \longrightarrow \varepsilon \mid aSa \mid bSb$
- G_4 : $S \longrightarrow \varepsilon \mid 0S0 \mid T, \qquad T \longrightarrow \varepsilon \mid 1T$
- G_5 : $S \longrightarrow 01 \mid 011 \mid 0S1 \mid 0S11$

Cvičení 3.9: Jazyk je generován např. gramatikou

$$\begin{array}{rcl} S & \longrightarrow & S_1C \mid AS_2 \\ S_1 & \longrightarrow & \varepsilon \mid aS_1b \\ S_2 & \longrightarrow & \varepsilon \mid bS_2c \\ A & \longrightarrow & \varepsilon \mid aA \\ C & \longrightarrow & \varepsilon \mid cC \end{array}$$

Cvičení 3.10: Gramatika očividně jednoznačná není. Např. slovo aac lze odvodit levou derivací $S \Rightarrow AaB \Rightarrow aB \Rightarrow aac$, ale také levou derivací $S \Rightarrow AaB \Rightarrow AAaB \Rightarrow AaB \Rightarrow aB \Rightarrow aac$. (Také např. $S \Rightarrow AaB \Rightarrow SBaB \Rightarrow BaB \Rightarrow SAaB \Rightarrow AaB \Rightarrow aB \Rightarrow aac$.)

Cvičení 3.11: Např. $S \longrightarrow aSb \mid Sb \mid b$

Cvičení 3.12: $S \to \varepsilon \mid aaSaa \mid abSba \mid baSab \mid bbSbb$

Cvičení 3.13*: $S \to aTa \mid bTb \mid \varepsilon$, $T \to aUa \mid bUb \mid a \mid b$, $U \to aSa \mid bSb$ (T generuje palindromy, jejichž délka modulo 3 je rovna jedné, U generuje palindromy, jejichž délka modulo 3 je rovna dvěma.)

Cvičení 3.14: Negenerují, neboť v G_2 neodvodíme ε.

Cvičení 3.15: Negenerují; v G_2 neodvodíme $aaaabb$, v G_1 ano.

Cvičení 3.16: b),d),e),f)

Cvičení 3.17*: Například

$$\begin{array}{lcl} S & \longrightarrow & \varepsilon \mid bS \mid cS \mid TU \\ T & \longrightarrow & aTbb \mid abb \\ U & \longrightarrow & TU \mid \varepsilon \mid cS \end{array}$$

Otázka 3.18: Musí mít možnost udělat na začátku ε-krok (případně více ε-kroků), kterým(i) vyprázdní zásobník; například má instrukci $(q_0, \varepsilon, Z_0) \to (q, \varepsilon)$ pro počáteční stav q_0, počáteční zásobníkový symbol Z_0 a nějaký stav q.

Otázka 3.19: Ke každému přechodu („instrukci“) $q \xrightarrow{a} q'$ konečného automatu sestrojíme analogickou instrukci $(q, a, Z_0) \to (q', Z_0)$ zásobníkového automatu (obsah zásobníku se nemění) a pro každý přijímající stav q konečného automatu dodáme instrukci $(q, \varepsilon, Z_0) \to (q, \varepsilon)$.

Cvičení 3.20: $(q_0, a \times (a+a), A) \vdash (q_0, a \times (a+a), B) \vdash (q_0, a \times (a+a), B \times C) \vdash (q_0, a \times (a+a), C \times C) \vdash (q_0, a \times (a+a), a \times C) \vdash (q_0, \times(a+a), \times C) \vdash (q_0, (a+a), C) \vdash (q_0, (a+a), (A)\,) \vdash (q_0, a+a), A)\,) \vdash \ldots \vdash (q_0, \varepsilon, \varepsilon)$

Cvičení 3.21: $(r_1, \varepsilon, Z) \to (r_1, \varepsilon)$
$(r_1, x, Z) \to (r_1, x)$, pro $x \in \{a, b\}$
$(r_1, c, z) \to (r_1, z)$, pro $z \in \{Z, a, b\}$
$(r_1, x, y) \to (r_1, xy)$, pro $x, y \in \{a, b\}$
$(r_1, \varepsilon, x) \to (r_2, x)$, pro $x \in \{a, b\}$
$(r_2, x, x) \to (r_2, \varepsilon)$, pro $x \in \{a, b\}$
$(r_2, c, x) \to (r_2, x)$, pro $x \in \{a, b\}$

Cvičení 3.22: Jedná se o jazyk $L = \{\, u \in \{a, b, c\}^* \mid$ po vynechání všech výskytů symbolu c z u dostaneme slovo ve tvaru $w(w)^R \,\}$, pro nějž jste měli zkonstruovat zásobníkový automat v jiném příkladu.

Cvičení 3.23: Jedná se opět o jazyk $L = \{\, u \in \{a, b, c\}^* \mid$ po vynechání všech výskytů symbolu c z u dostaneme slovo ve tvaru $w(w)^R \,\}$.

Cvičení 3.24: Dá se aplikovat standardní konstrukce zásobníkového automatu ekvivalentního zadané gramatice.

Cvičení 3.25*: Jedná se o jazyk $L = \{w \in \{a, b\}^* \mid |w|_a = |w|_b\}$. Deterministický automat může mít stavy q_a a q_b. Když bude řídicí jednotka ve stavu q_a, tak v dosud přečteném úseku u vstupního slova platí $|w|_a \geq |w|_b|$, pro stav q_b bude platit $|w|_a \leq |w|_b|$. Se zbytkem konstrukce už si musíte poradit sami.

Cvičení 4.1: Po odstranění neterminálů, které není možno přepsat na terminální slovo dostaneme gramatiku:

$$\begin{array}{lcl} S & \longrightarrow & aSb \mid aDaS \mid \varepsilon \\ C & \longrightarrow & CC \mid cS \\ D & \longrightarrow & aSb \mid cD \mid aEE \\ E & \longrightarrow & bD \end{array}$$

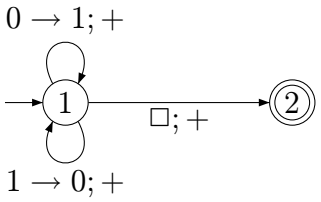
Po následném odstranění nedosažitelných neterminálů dostaneme:

$$\begin{array}{lcl} S & \longrightarrow & aSb \mid aDaS \mid \varepsilon \\ D & \longrightarrow & aSb \mid cD \mid aEE \\ E & \longrightarrow & bD \end{array}$$

Cvičení 4.5: regulární: L_2, L_3
bezkontextové neregulární: L_1, L_6
nebezkontextové: L_4, L_5, L_7

Otázka 5.2: Libovolně velký, páska je (potenciálně nekonečná) a její využití při výpočtech není nijak omezeno.

Cvičení 5.3:



Cvičení 5.4: Toto zobecnění je přímočaré; stavy „přenášející kopii písmene doprava“ teď ovšem budou dva, protože stroj si musí pamatovat, který symbol právě přenáší.

Cvičení 5.5: Lze využít (tj. patřičně modifikovat) stroj z Řešeného příkladu 5.2.

Cvičení 5.6: Výpočet se zastaví na všech slovech kromě ε. Pokud je vstupem neprázdný řetězec a-ček, výstup z něj vznikne přepsáním posledního a na b; jinak je výstup roven vstupu.

Cvičení 5.9: Vyhradíme volný úsek paměti délky k a počátku p a potom budeme k prvku $a[j]$ přistupovat jako k paměťovému místu na adrese $p + j$ (využitím indexregistru).

Cvičení 5.10: Vyhradíme volný úsek paměti délky $k\ell$ a počátku p. Přistupovat budeme k $a[i,j]$ na adrese $p + i\ell + j$.

Cvičení 5.11: Ve vymezeném úseku paměti implementujeme zásobník pro lokální proměnné a návraty. (Podobně to dělá běžná CPU.)

Otázka 5.12: Původní symboly reprezentujeme bloky s dostatečným počtem „bitů“; simulující stroj musí provedení jedné instrukce simulovaného stroje implementovat pomocí kroků, v nichž načte celý blok do řídicí jednotky (má tedy více stavů) a pak patřičně změní reprezentaci simulovaného stavu v řídicí jednotce, zapíše nový obsah aktuálního bloku a přesune se na sousední blok předepsaným směrem.

Otázka 7.1: Vůbec nic.

Otázka 7.2: Ano, např. máme $n + 1 \in O(n)$.

Otázka 7.3: Ne, může to platit jen pro konečně mnoho n.

Cvičení 7.4: Dekadický zápis čísla má délku zhruba $\log_{10} x$. Takže můžeme říci, že velikost vstupu je $\Theta(\log x_1 + \log x_2)$ (nebo např. $\Theta(\log(max\{x_1, x_2\})$.) (Jak víme, na základu logaritmu nezáleží.)

Cvičení 7.5: Cyklus `for` se vykoná právě n-krát, takže n-krát se i přičte hodnota do z a n-krát se inkrementuje i. Dále mezi aritmetické operace počítáme $n+1$ porovnání $i < n$ a jedno závěrečné dělení. Celkem tedy máme $3n + 2$ aritmetických operací. Řádově je to $\Theta(n)$.

Cvičení 7.6: Vnější cyklus se jasně iteruje n^2-krát. Počet iterací vnitřního cyklu však závisí na proměnné i vnějšího cyklu. Vnitřních iterací proto proběhne součtem $\sum_{i=1}^{n^2} i = 1 + 2 + 3 + \ldots + n^2 - 1 + n^2$. Někteří z vás možná ví, že součet této řady je $\frac{1}{2}n^2(n^2 + 1)$, ale my si výsledek umíme asymptoticky odvodit sami

$$1 + 2 + 3 + \ldots + n^2 \leq n^2 + n^2 + \ldots + n^2 = n^2 \cdot n^2 = n^4\,,$$

ale na druhou stranu

$$1+2+\ldots+\frac{1}{2}n^2+\ldots+n^2 \geq \frac{1}{2}n^2+\frac{1}{2}n^2+1+\ldots+n^2 \geq \frac{1}{2}n^2\cdot\frac{1}{2}n^2 = \Theta(n^4)\,.$$

Počet průchodů vnitřním cyklem tedy je $\Theta(n^4)$.

Cvičení 7.7: $\log n \in o(n^{0.5})$ (protože $\lim_{n\to\infty} \frac{\log n}{n^{0.5}} = 0$).

Cvičení 7.8: (c)

Cvičení 7.9: a), b), c)

Cvičení 7.10: $c) \prec a) \prec b)$

Cvičení 7.11: $b) \prec a) \prec c)$

Cvičení 7.12: $a) \prec c) \prec b)$

Cvičení 7.13: $c) \prec a) \prec b)$

Otázka 7.14: Kdyby bylo pro daný vstup více možných výstupů nedeterministického algoritmu než jen Ano/Ne, nemohli bychom rozumně definovat, jaký výstup k onomu vstupu tedy přiřadíme.

Otázka 7.15: Ano. U rozhodovacího (deterministického) algoritmu stačí prohodit vydávané odpovědi Ano a Ne.

Otázka 7.16: Neplyne. Navíc si všimněme, že např. u problému SAT lze v pozitivním případě uhodnout a ověřit splňující pravdivostní ohodnocení. Není ale vůbec jasné, zda existuje něco takového, co by se dalo „uhodnout a ověřit“ k prokázání nesplnitelnosti dané formule.

Cvičení 7.17: K instanci problému IS, tedy k zadanému grafu G a číslu k musí sestrojit instanci problému HC, tedy orientovaný graf G' tak, že je zaručeno, že když v G existuje nezávislá množina velikosti k, tak v G' existuje hamiltonovský cyklus, a když v G nezávislá množina velikosti k neexistuje, tak v G' neexistuje hamiltonovský cyklus; navíc musí mít A polynomiální časovou složitost.

Literatura

[Chy84] Michal Chytil. *Automaty a gramatiky.* SNTL Praha, 1984.

[Gru97] Jozef Gruska. *Foundations of Computing.* Intern. Thomson Computer Press, 1997.

[HU78] J. Hopcroft and J. Ullman. *Formálne jazyky a automaty.* Alfa Bratislava, 1978.

[Kuč83] Luděk Kučera. *Kombinatorické algoritmy.* SNTL Praha, 1983.

[MvM87] Molnár, Češka, and Melichar. *Gramatiky a jazyky.* Alfa-SNTL, 1987.

[Sip97] Michael Sipser. *Introduction to the Theory of Computation.* PWS Publish. Comp., 1997.