Formální jazyky a automaty

HASHIM HABIBALLA - PETR VOJKOVSKÝ

Přírodovědecká fakulta OU, Ostrava

Gymnázium, Frýdek - Místek

Teoretická informatika tvoří základ celého oboru a s pomocí jejích teoretických výsledků se vytváří řada aplikovaných produktů informatiky. Ve výuce na VŠ v informatických oborech tvoří tyto disciplíny podstatnou část znalostí absolventa oboru (zvláště na netechnicky zaměřených fakultách). Vzhledem k jejich příbuznosti s matematikou a jejím formálním aparátem jsou však u studentů málo oblíbeny a pokud je jejich způsob výuky pouze formální (založený na formalizaci), pak u studentů dochází k demotivaci a memorovaní těchto vysoce logicky založených poznatků. Komplexní příprava praktických odborníků a učitelů v oblasti informatiky vyžaduje pevné základy teoretických disciplín. Je důležité vytvářet u studentů nejen statické znalosti (definice, věty, důkazy), ale také specifické dovednosti, návyky a postoje (analytické a algoritmické myšlení, strukturovaný přístup k jazykům a překladačům atd.) Z praxe učitelů (i studentů) těchto předmětů lze často slyšet, že klasická výuka založená na učení znalostí při použití formy hromadného vyučování vede k memorování bez pochopení principů a demotivaci pro další studium. Cílem by mělo být, aby každý student umět vidět vlastnosti algoritmů v jejich obecnosti, nutnost zkoumat řešitelnost a efektivnost řešení problémů. Také by měl pochopit, že používané nástroje v informatice jsou založeny na pevných strukturách souvisejících s pojmy gramatiky, jazyka a analyzátoru. Taktéž logický úsudek a jeho formalizace v matematické logice by se měl stát součástí postojů informatika. Teorie formálních jazyků, o které zde především budeme mluvit, dává prostředky pro tvorbu překladačů programovacích jazyků, pro nahrazování řetězců při hromadném zpracování textů apod. Půjde nám o to ukázat, že teoretická informatika není jen souborem těžko pochopitelných definic, vět a důkazů, ale že její podstata je poměrně jednoduchá a poskytuje důležité principy pro celou informatiku.

Vymezení toho, co je teoretická informatika může být složitý úkol. Nejtypičtějšími zástupci teoretické informatiky jsou:

1. Teorie formálních jazyků a automatů
2. Teorie vyčíslitelnosti a složitosti (souhrnně označovaná jako teorie algoritmů)
3. Logika (její informatická část zaměřená na problematiku automatizovaného odvozování)

Vzhledem k tomu, že není cílem tohoto článku udělat úvod do všech disciplín (věřím však, že by to pro čtenáře mohlo být inspirující pro další podrobné studium), omezíme se pouze na několik vět o každé z těchto oblastí.

ad 1. Tato disciplína zkoumá vlastnosti jazyků resp. jejich matematických modelů. Snaží se formalizovat intuitivní pojmy abecedy, slova, jazyka. Definuje model generátoru jazyka - gramatiky a jeho duálního pojmu akceptoru jazyka - automatu. Gramatika je souborem pravidel, které umožňují vytvářet správné věty (slova) jazyka a automat je algoritmem, který nám dává prostředek pro rozhodnutí o dané větě, zda patří do jazyka (je správně utvořenou). Na základě různě složitých (generativních) gramatik a automatů se pak rozlišují třídy jazyků od nejjednodušších - regulárních po nejsložitější - typu 0. Mezi těmito pojmy v jednotlivých třídách jazyků je dokázáno logickými a pochopitelnými prostředky mnoho vztahů. Pro praktické využití v informatikově profesi však mají největší význam dvě třídy nejjednodušších jazyků - regulárních a bezkontextových.

ad 2. Teorie vyčíslitelnosti a složitosti zkoumá vlastnosti algoritmů a to v zásadě ze dvou hlavních hledisek. Vyčíslitelnost se zabývá algoritmickou řešitelností problémů a složitost se zabývá časovou nebo prostorovou náročností řešitelných problémů. V roce 1936 Alan Turing, který je pro teoretickou informatiku klíčovou postavou, formuloval svou ideu formalizace pojmu algoritmu ve formě Turingova stroje. Tato formalizace má svůj velmi jednoduchý princip mechanismu se vstupní potenciálně nekonečnou páskou s danou abecedou a čtecí hlavou, která může zapisovat i číst na pásce a pohybovat se po jednom políčku. Tento velice jednoduchý formalismus s velkou výpočetní silou umožnil formulovat pro informatiku klíčové pojmy jako jsou rozhodnutelnost a částečná rozhodnutelnost problémů (příp. lze tyto pojmy aplikovat na funkce, množiny či jazyky).

ad 3. Matematika používá logiku jako prostředek výstavby svých teorií. I když již v třicátých letech vznikly na pomezí logiky a teorie vyčíslitelnosti mnohé výsledky o rozhodnutelnosti teorií (zejména díky K. Gödelovi), význam axiomatických systémů pro informatiku rostl od šedesátých let 20. století díky umělé inteligenci. V roce 1965 Robinson formuloval rezoluční princip a otevřel tak cestu k automatizovanému dokazování vět, což dalo podnět k vytvoření mnoha systémů, které jsou založeny na matematické logice. Mají sloužit k řešení náročných úloh, které vyžadují jistou inteligenci jako je například plánování, automatizované usuzování na základě pravidel či řízení.

Zkuste se zamyslet, jak mnoho dáváme studentům z oborů jako je fyzika, chemie z jejich teoretických základů a jak málo je toho v porovnání s tím předkládáno v informatice. Nechceme vůbec zlehčovat nutnost naučit studenty základní informační gramotnost, ta však nemůže být zaměňována s oborem informatika. Studenti SŠ pak bohužel při výběru studia mohou být nepříjemně překvapeni, že informatika ani zdaleka nejsou textové editory, tabulkové procesory a databázové aplikace (a z praxe máme potvrzeno, že jich není málo). A nejen to, informatika není ani zdaleka jen praktické programování, ale má i své výše zmíněné (a velice důležité) teoretické základy, na kterých staví. To že jsou tyto základy někdy uzavřeny samy do sebe přehnaně prezentovanou formalizací (v žádném případě nechceme snižovat význam formalizace - ten je neoddiskutovatelný!) by nemělo odradit učitele i studenty, aby se pustili do studia některých vybraných problémů, které lze popsat a někdy dokonce lépe pochopit pomocí pojmů teoretické informatiky. Tento text se Vám

pokusí dát několik z mnoha příkladů, kde můžete výklad z oblasti teorie formálních jazyků použít a navíc studentům ukázat některé pokročilejší programátorské techniky (zřejmě především v nepovinných seminářích informatiky).

1 Teorie formálních jazyků a automatů

Pokusme se na úvod odhlédnout od rigorózních definic a uvědomit si následující schéma, které asi všichni známe ze základní školy z českého (anebo jakéhokoliv jiného přirozeného) jazyka:

Intuitivní příklad:
⟨Jednoduchá česká věta⟩ ::= ⟨Podmět⟩⟨Přísudek⟩⟨Předmět⟩
⟨Podmět⟩ ::= Jiří | Jan
⟨Přísudek⟩ ::= má | řídí
⟨Předmět⟩ ::= auto | firmu

Toto schéma obsahuje pravidla, která vždy položky v úhlových závorkách přepisují na posloupnosti (řetězce, uspořádaný výčet) buď stejných položek v úhlových závorkách nebo slov (symbolů). Jde vlastně o gramatiku (soubor pravidel), jak položky v úhlových závorkách (může říci proměnné) můžeme přepisovat postupně až na slova (symboly). Můžeme tedy tímto postupem dostat například takovéto jednoduché české věty:
⟨Jednoduchá česká věta⟩ ⇒ ⟨Podmět⟩⟨Přísudek⟩⟨Předmět⟩ ⇒ Jiří ⟨Přísudek⟩⟨Předmět⟩ ⇒ Jiří řídí ⟨Předmět⟩ ⇒ Jiří řídí firmu.
nebo
⟨Jednoduchá česká věta⟩ ⇒ ⟨Podmět⟩⟨Přísudek⟩⟨Předmět⟩ ⇒ Jan ⟨Přísudek⟩⟨Předmět⟩ ⇒ Jan má ⟨Předmět⟩ ⇒ Jan má auto.

Tedy postupným dosazování za proměnné podle pravidel dojdeme až na posloupnosti, které už neobsahují žádné proměnné.

Můžeme tedy formulovat laické definice pojmů gramatiky a jazyka následovně:

Gramatika je soubor výchozích symbolů, proměnných a pravidel pro vytváření vět.

Jazykem je množina všech vět, které jsou vytvořeny v souladu s gramatikou.

Intuitivní příklad:

Budeme-li uvažovat o tom, co je to jazyk, zkusme vyjít z gramatiky z předchozího příkladu na jednoduchou českou větu. Jazyk L je pak množina: L = { Jiří má auto, Jiří má firmu, Jiří řídí auto, Jiří řídí firmu, Jan má auto, Jan má firmu, Jan řídí auto, Jan řídí firmu} Jazyk má tedy 8 prvků (správně utvořených vět). Uvědomte si, že tento příklad je velice jednoduchý. Jednoduchá česká věta se rozvinula na posloupnost podmětu, přísudku a předmětu. Každá z těchto tří proměnných se mohla podle gramatiky rozvinout do dvou symbolů (slov). To tedy dává 2^3 možných vět. Jazyk je tedy v tomto případě konečný, neboť má konečně mnoho prvků. Například věta Auto řídí Jan by podle této gramatiky nepatřila do jazyka, protože není utvořena podle pravidel (nedodržuje slovosled daný prvním

pravidlem)!

Další pojem, který zkoumá tato teorie, je automat. Automat má zjišťovat, zda věta patří do určitého jazyka. Například automat, který zjišťuje zda věta je jednoduchá česká věta podle příkladu, by pracoval tak, že by si větu rozdělil do slov a pak zjistil zda první slovo je podmět, druhé přísudek a třetí předmět. Pokud by to tak bylo, pak by odpověděl, že slovo patří do jazyka. U uvedeného výkladu berte prosím v úvahu, že jeho účel je být jasným a jednoduchým vysvětlením, v žádném případě ne formální definicí. Skript a učebnic, které přesně a exaktně matematicky definují tyto pojmy teorie formálních jazyků je mnoho a ani by se příliš pro mladší studenty nehodily. Pravděpodobně by je spíše od dalších pokusů, zkoumat tuto teorii, odradily. Poznatky teorie formálních jazyků mají význam pro obory informatiky, neboť s využitím jejich poznatků jsou vybudovány aplikované produkty informatiky jako jsou vývojové nástroje (programovací jazyky a jejich prostředí pro tvorbu počítačových aplikací), databázové dotazovací jazyky (jazyky umožňující v databázích vyhledávat či měnit informace), značkovací jazyky jako je XML, HTML (jazyky popisující strukturovaně dokumenty - například webovské stránky) apod.

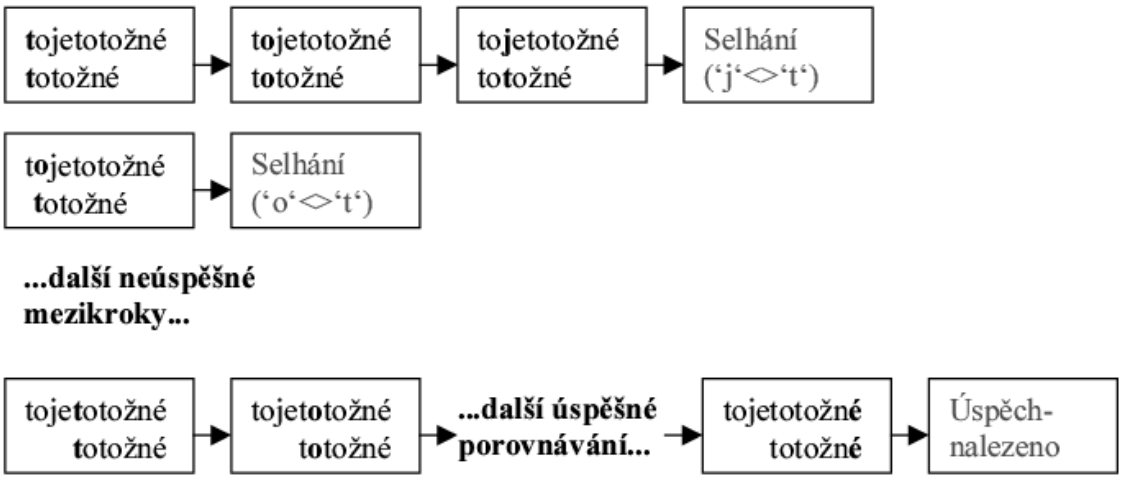
2 Regulární jazyky a konečné automaty

Nejjednoduššími jazyky, které zkoumá teorie formálních jazyků, jsou jazyky regulární (resp. jazyky rozpoznatelné konečnými automaty). Pro intuitivní představu lze tento pojem přiblížit například na znacích ? a *, které znají uživatelé operačních systémů (může jít i o jiné znaky, ale princip bude podobný). Pokud si s pomocí těchto znaků chcete například vypsat soubory začínající řetězcem "Ma" s příponou ".txt" (Matematika.txt, Makro.txt atp.), pak nemusíte řešit tuto situaci ručním hledáním, ale automatizujete si tuto činnost pomocí hledání řetězce "Ma*.txt". Zde hvězdička má pro operační systém význam výskytu libovolné kombinace (i prázdné) symbolů abecedy. Tento výraz pak vytváří celou množinu slov (jazyk), která chceme vyhledat. Notace regulárních jazyků (generovaných regulárními výrazy) je ve své formální definici samozřejmě trochu složitější, ale ve své podstatě umožňuje právě takovéto činnosti - několikanásobné opakování stejných symbolů, více různých slov atd.

I když pojmy teorie jazyků a automatů zní odtažitě, podívejme se na jednu úlohu, kterou asi řešil každý čtenář tohoto článku a tou je vyhledávání v textu. Hned poté si ukážeme, jak je možné takovou myšlenku ilustrativně vysvětlit pomocí pojmu konečného automatu. Pro tuto úlohu existují různě efektivní a složité algoritmy. Pokusme se rozebrat nejprve ten nejjednodušší, který nás zřejmě napadne (jde o algoritmus brute-force, tj. hrubá síla).

Intuitivní příklad: Vezměme si slovo "totožné". Jak bychom realizovali jeho vyhledávání například v textu "tojetotožné".

Algoritmy, které postupně načítají text a hledají výskyt slova, samozřejmě využívají

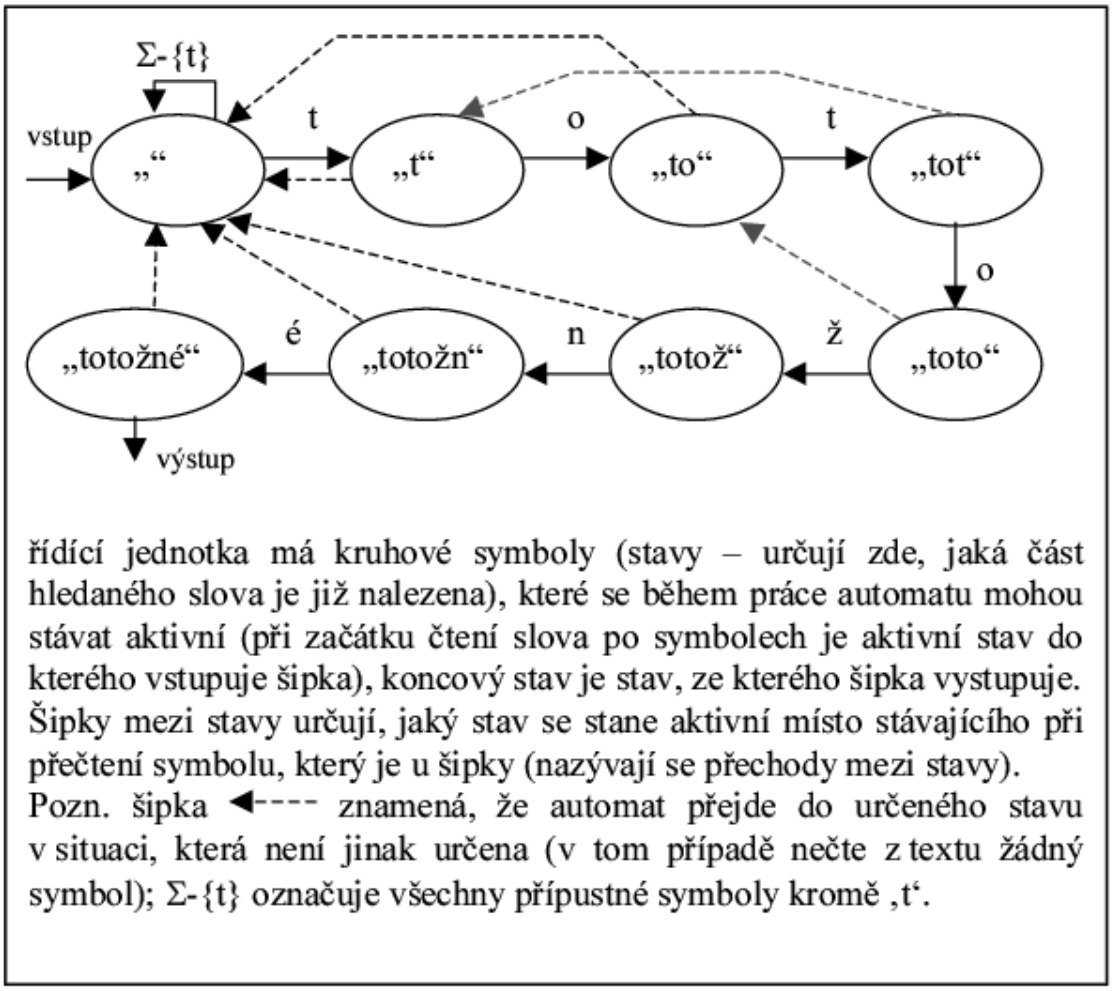


Obrázek 1: Algoritmus "hrubá síla"

knihoven funkcí. Tyto knihovny jistě obsahují již připravené algoritmy porovnání řetězců apod. Přesto půjdeme-li až na jádro způsobu nalezení slova bez použití těchto pomůcek, musí se číst postupně znaky textu a srovnávat - jde o první písmeno hledaného slova 't'? Pokud ano, dále srovnávej zda souhlasí následující písmena... Pokud projdeme celé slovo totožný, aniž by se v právě načítaném úseku textu něco lišilo, pak můžeme skončit a říct, že "totožný" se v textu vyskytuje a pokud ne, pak se vrátíme na další písmeno textu a celý postup opakujeme. Toto je zhruba řečeno algoritmus brutální síly. Jeho postup probíhá zkráceně takto (srovnávání textu a slova - obrázek 1):

Vidíte, že uvedený algoritmus je skutečně "hrubý". Nevyžaduje sice příliš mnoho uvažování, ale na druhou stranu je poměrně "hloupý", protože se vždy vrací v textu na následující symbol a vůbec nevyužívá informaci o tom, co již v hledaném slově úspěšně srovnal. Proto by bylo rozumné zkusit navrhnout algoritmus, který by se již nemusel nikdy vracet v textu na symboly, které již srovnával. Pokusme se tuto myšlenku ilustrovat pomocí pojmů teorie formálních jazyků.

Z hlediska teorie formálních jazyků, jde o vyhledávání regulárního výrazu (mimochodem velice strukturálně jednoduchého - tyto výrazy mají obecně mnohem větší sílu než pro náš ilustrativní příklad), který můžeme realizovat pomocí konečného automatu. Pokud se nebudeme snažit o exaktní formalizaci, lze si automat představit jako stroj. Má pásku, na kterou můžete zapisovat po symbolech slova, která chcete rozpoznat, zda patří do jazyka nebo ne. Dále má řídící jednotku se stavy, které si můžete představit jako žárovky, které se rozsvítí, pokud je automat právě v tom konkrétním stavu. Z pásky umí automat číst pouze po jednotlivých symbolech a nemůže se vracet zpět na již přečtené symboly. Je důležité, abyste si uvědomili, že automat si nemůže nic pamatovat. Vždy vidí jen jeden symbol ze zkoumaného slova. Dále automat ví, jak reagovat na situaci pokud je v nějakém stavu a na pásce vidí určitý symbol. Vždy je pak výsledkem takové akce přechod do nějakého stavu (může jít i o stejný stav). Nakonec má automat k dispozici informaci, ve kterém stavu má



Obrázek 2: Přechodová funkce automatu

začít a který stav je koncový (nebo více stavů) a pokud se dostane do takového stavu, pak je slovo z jazyka. Pokusme se tento automat reprezentovat s pomocí stavového diagramu (pro ty kteří se již setkali s teorií zdůrazňujeme, že jde o automat zobecněný s e-přechody - přerušovanými čarami, což nesnižuje obecnost):

Co nám tento diagram říká (obrázek 2)? Stavy vyjadřují, jakou část hledaného slova jsme již úspěšně načetli. Na počátku po vstupu do automatu nejprve čekáme na symbol 't', kterým slovo začíná (to je ona smyčka $\Sigma - \{t\}$). Po načtení 't' se dostáváme do příslušného stavu. Pokud přijde 'o' pokračujeme v úspěšném porovnávání, ale pokud ne, pak se musíme rozhodnout, kam se vrátit, abychom sice nic z textu znovu zbytečně nečetli, ale zároveň abychom tak neopominuli již načtenou úspěšnou část slova. V tomto případě se můžeme vrátit až na počátek. Podívejme se ale, co se děje, jsme-li již ve stavu "tot". V tom případě se nemůžeme vrátit úplně na počátek, protože bychom tak ignorovali, že jsme již úspěšně načetli symbol 't'. Musíme se tedy do příslušného stavu nastavit, abychom tak neporušili

Obrázek 3: Algoritmus Knuth-Morris-Pratt

potenciální možnost vyhledání slova v textu. Stejná "nestandardní" situace nastává ve stavu "toto". Musíme vzít v úvahu, že i přes neúspěch pro "totož" jsme se již dostali do stavu, kdy je načteno "to" a může potenciálně přijít "tot"! Naznačme schématicky, jak pracuje tento vylepšený algoritmus (obrázek 3).

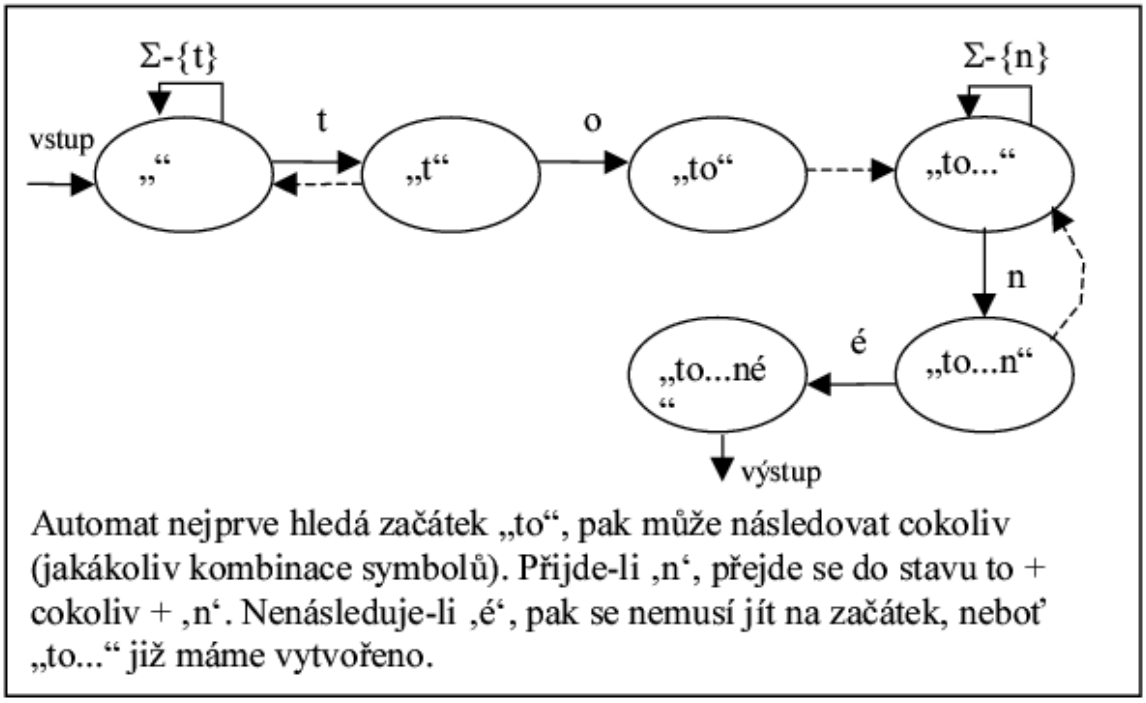
Tento automat nám vlastně poskytuje jakési "know-how", díky němuž se zbavíme základního nedostatku (z hlediska efektivity algoritmu) a to je nutnost vracet se v textu vždy na další symbol. Při tomto postupu nikdy již čtený symbol v textu nemusíme znovu načítat. To jistě v rozsáhlých textech zrychlí hledání. Cena za to je nutnost zjistit si, kam se musím vrátit v automatu při selhání. Jelikož to však činíme jen jednou na počátku, u rozsáhlých textů se to vyplatí. Popsaný postup je vlastně teoreticky popsaným algoritmem známým jako Knuth-Morris-Prattův algoritmus (KMP). Lze jej naprogramovat a navíc poměrně jednoduše - je pouze třeba zkonstruovat postup vytváření "automatu" - jinak řečeno tabulky, která popisuje kam se vrátit z jednotlivých stavů - jako algoritmus (není to složité - v podstatě jde o problém prohledávání slova v sobě samém a tím zjištění - kolik symbolů v jednotlivých částech slova se shoduje s jeho začátkem). Ukázku můžete vidět na následujícím ilustračním programu (procedura InitNext provádí diskutované vytvoření přechodové funkce).

```
var next : array[1..255] of byte; a, p : string; M, N : byte;

procedure InitNext;
var i, j: integer;
begin
  i := 1; j := 0; next[1] := 0;
  repeat
  if (j = 0) or (p[i] = p[j]) then
    begin
      inc(i); inc(j); next[i] := j;
    end
  else j := next[j];
  until (i > M);
end;
```

```
function KMPSearch:integer;
 var i,j:integer;
begin
  i := 1; j := 1; InitNext;
  repeat
    if (j = 0) or (a[i] = p[j]) then
      begin
        inc(i); inc(j);
      end
    else j := next[j];
  until (i > N) or (j > M);
  if j > M then KMPSearch := i - M else KMPSearch := i;
end;

begin
  a := 'tojetotozne'; N := length(a);
  p := 'totozne'; M := length(p);
  writeln('Nalezeno na pozici:',KMPSearch);
end.
```

Příklad, který jsme pravě prezentovali, je samozřejmě velice jednoduchý. Konečné automaty mají větší možnosti než jen rozpoznávání pevných řetězců. Regulární výrazy, které k nim přísluší, mohou nahrazovat části slov libovolnými kombinacemi apod. Například lze sestrojit automat, který by vyhledával text se slovem, které začíná na "to" a končí na "né"- tedy např. totožné, toporné, topné apod. Takový automat by vypadal následovně (obrázek 4).

Dalším typickým příkladem využití konečných automatů je vyhledávání několika různých slov v textu najednou nebo naopak slov, která splňují více podmínek najednou. Právě zde se dostáváme opět již k diskutované formalizaci. Pokud pochopíte, jak automat pracuje a jak je sestrojovat, můžete po důkladnějším studiu teorie formálních jazyků a automatů používat její formální aparát - tedy například algoritmy na sestrojování sjednocení, průniku automatů apod. Pokud Vás zaujal tento ilustrativní příklad, poznali jste, jak může být pro studenta zajímavé používat tento formalismus například v algoritmizaci při výuce vyhledávacích algoritmů. Věříme, že tato teorie přímo vybízí k hledání dalších zajímavých úloh, na kterých se mohou studenti seznámit s pochopitelnými a efektivními programátorskými technikami a zároveň tak poznat svět teoretické informatiky. Lze k tomu využít již zmiňované učebnice, ať již v elektronické nebo fyzické podobě. Také Internet je velkým zdrojem inspirace pro další práci s konečnými automaty a regulárními výrazy.



Obrázek 4: Automat pro složitější výraz

3 Implementace konečného automatu

Kromě zmiňovaného algoritmu KMP, lze samozřejmě velice jednoduše realizovat simulaci jakéhokoliv konečného automatu. Takovýto příklad bude zřejmě výchozím, pokud se rozhodnete pro vlastní zkoumání teorie formálních jazyků a automatů nebo její výuku.

Implementace konečného automatu vlastními silami.

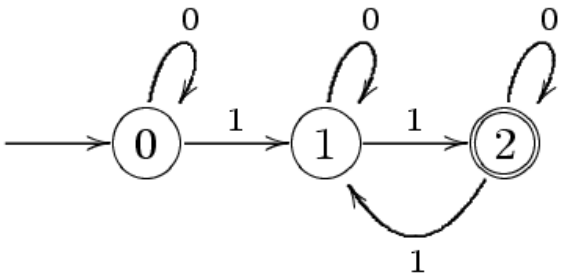
Nechť abeceda je dvouprvková množina $A = \{0, 1\}$. Automat nechť rozpoznává slova obsahující sudý počet jedniček (obrázek 5). Množina stavů $Q = 0, 1, 2$, kde 0 je počáteční stav, 2 je koncový.

Rozpoznané slovo automat ohlásí jako 'SHODA'.

Implementace v jazyce Pascal:

Čteme jednotlivé znaky ze standardního vstupu pomocí funkce `ReadKey`. Ta ale vrací načtený znak. Např. Převedení znaku číslice na „numerickou podobu“ v DEC (např. `'0'` na 0) realizuje funkce `znc(z)`.

Tabulku přechodové funkce vyjádříme pomocí dvourozměrného pole. Automat při své činnosti vypisuje stavy, ve kterých se právě nachází.



Obrázek 5: Automat "Sudý počet jedniček"

```
uses crt; var c:char;{ Vstup }
    q, sym:integer; { Počáteční stav }
    delta:array[0..2,0..1] of integer; { Přechodová funkce - 3 x 2 }

function znc(z:char):integer;
begin
  znc := ord(z) - ord('0');
end;

begin
  clrscr;
  delta[0, 0] := 0; delta[0, 1] := 1;
  delta[1, 0] := 1; delta[1, 1] := 2;
  delta[2, 0] := 2; delta[2, 1] := 1;
  c := ReadKey;
  write('Vstup symbol:',c);
  while c <> chr(13) do
  begin
    sym := znc(c);
    if ((sym = 0) or (sym = 1)) then
    begin
      q := delta[q, sym];
      writeln('-Jsem ve stavu:',q);
      c := ReadKey;
      if c <> chr(13) then write('Vstup symbol:',c);
    end
    else
    begin
      writeln('-znak nepatri do abecedy!');
      Exit;
    end;
  end;
```

```
  if q = 2 then writeln('SHODA')
  else writeln('NESHODA');

end.
```

Právě takové typy úloh, které jsou nejen výukou teorie, ale umožňují studentům vidět i použití a implementaci, jsou ve výuce používány. Bohužel převažující platforma MS-Windows neposkytuje příliš mnoho přirozeného využití regulárních výrazů, to je spíše doména operačního systému Unix. Ten již obsahuje řadu prostředků, které umožňují definovat regulární výrazy (např. sed, gawk) a na těch je založena výuka v seminářích z informatiky. Studenti mají samozřejmě zařazeny i teoretické otázky. Poměrně snadno pochopitelné jsou i převody automatů a regulárních výrazů, které jsou rovněž ve výuce zahrnuty. Věříme, že právě takto aplikačně pojatá výuka i na střední škole může studentům poodhalit mnoho pojmů a otázek teorie formálních jazyků. Je to výuka spíše intuitivní a neformální, ale právě takový přístup by mohl přinést zlepšení povědomí i přístup k teoretické informatice.

Literatura

[1] ČEŠKA, M., RÁBOVÁ, Z. Gramatiky a jazyky. VUT Brno, 1992
http://www.fit.vutbr.cz/study/courses/TI1/public/gj-1.3.pdf (2003)

[2] HOPCROFT, J.E., ULLMAN, J. D. Introduction to Automata theory, Languages and Computation. Addison-Wesley, Reading (Mass.), 1979

[3] HABIBALLA, H. Regulární a bezkontextové jazyky I. Ostravská univerzita, Ostrava, 2003

[4] CHYTIL, M. Automaty a gramatiky. Matematický seminář SNTL Praha, 1984

[5] JANČAR, P. Teorie jazyků a automatů. VŠB TU Ostrava,
http://www.cs.vsb.cz/jancar/ (2003)

[6] KOUBKOVÁ, A., PAVELKA, J. Úvod do teoretické informatiky. Matfyz press, Praha 1998.

Kontaktní adresa:

Hashim Habiballa
katedra informatiky a počítačů
Přírodovědecká fakulta Ostravské Univerzity
30.dubna 22, 701 03 Ostrava 1,
tel.: +420596160241,
e-mail: habiballa@volny.cz, www: http://www.volny.cz/habiballa/

Petr Vojkovský
Gymnázium Petra Bezruče ve Frýdku-Místku
ČSA 517, 738 02 Frýdek-Místek