Príloha – postrehy z riešenia
Informatické postupy automatizovaného vytvárania logických tabuliek

Miroslav Hájek, septima (2017 / 18)

Motivácia

Pri zisťovaní všetkých pravdivostných hodnôt logického výrazu sme sa v triede zhodli na tom, že je to veľmi zdĺhavé a veľmi ľahké na pomýlenie sa. Keďže práca s binárnou logikou je základom všetkých počítačov, položil som si otázku ako napísať program, ktorý to zvládne za nás.

Zápis a Gramatika

Pretým, než počítač začne niečo počítať musíme mu dodať dáta v jemu zrozumiteľnej podobe. Aby však bolo používateľské rozhranie jednoduché, musíme tiež zabezpečiť premenu matematického zápisu do série samostatných lexém. Na klávesnici je celkom obtiažne nájsť matematické symboly a preto ich nahradím za ASCII znaky podľa následujúcej tabuľky gramatiky.

Kategória	Matematický zápis	Nahradzujúce značky	Výpočet v počítači
Operand	$[A-Z]$	[A – Z][a – z]{>=1}	–
Unárny operátor	$\neg$	~	`not A`
Binárne operátory z jedného znaku	$\wedge$	&	`A and B`
	$\vee$	|	`A or B`
Binárne operátory z dvoch znakov	$\Longrightarrow$	=>	`not A or B`
	$\Longleftrightarrow$	<=>	`(A and B) or (not A and not B)`

Povedzme, že máme takýto logický výraz: $\neg(C \vee B) \implies (\neg A \wedge B)$

Jeho „strojovo čitateľný“ zápis by vyzeral takto: ~(C | B) => (~A & B)

ale môžeme zapísať aj zrozumiteľnejšie výrazy: (~auto & mačka) <=> (auto & ~mačka)

Lexikálna analýza

Premena vyššie uvedeného zápisu na jednotlivé lexikálne nedeliteľné prvky (tokeny) sa nazýva tokenizácia. Je riešená pomocou *konečného stavového automatu*, čo je len množstvo podmienok, ktoré sa pri prečítaní rozhodnú, ktorej značke (lexému) patrí daný znak (prípadne sa pozrie o pár znakov dopredu a ak sa potvrdí hypotéza, tak sa znaky „pohltia“). Napravo je zoznam tokenov vytvorených z výrazu:

$C \implies \neg(A \wedge B)$

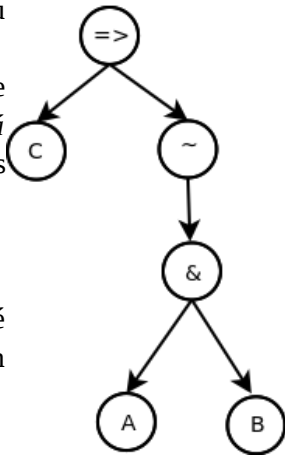
Typ	Lexém
Premenná	C
Operátor	=>
Premenná	~
Zátvorka	(
Premenná	A
Operátor	&
Premenná	B
Zátvorka	)

Sémantická analýza

Vďaka tomu, že sme v predchádzajúcom kroku dešifrovali význam znakov v reťazci dáme ich teraz do dátovej štruktúry, tak aby sme s nimi mohli ľahko manipulovať pri výpočtoch. Ak by sme mali veľmi komplikovanú komplexnú rekurzívne definovanú gramatiku určite sa oplatí použiť *abstraktný syntaktický*

strom. Takýto prístup však pre málo značiek vytvorí nepotrebnú komplexnosť. Najlepší prístup je znovu zmeniť formu dát.

Z bežného života sme zvyknutí písať operandy okolo operátorov a prioritu zaznačíme zátvorkami alebo pravidlami, čo sa nazýva ako *infixová notácia*.
(napr. 1 + 2 * 3) Narozdiel od toho, počítač potrebuje mať pri vykonávaní operácie jednoducho prístupné všetky potrebné operandy. Na to nám poslúži *postfixová notácia* (napr. 1 2 3 * +), kvôli jej využívaniu *zásobníka*. Premenu na takýto zápis zabezpečí *algoritmus Shunting-yard* (zoraďovacej stanice).
[Dijkstra, E.W. (1961) - https://ir.cwi.nl/pub/9251].



Syntaktický strom

Funguje na princípe zásobníka operátorov. Všetky operandy sú prepisované na výstup, iba operátory sa hromadia a sú „poslané“ na výstup, len ak to dovoľuje ich priorita udaná zátvorkami a počtom ich operandov.

Infix `C => ~(A & B)`

Postfix `C A B & ~ =>`

Vytvorenie tabuliek a Výpočet

Problém so symbolickým výpočtom logických tabuliek je ten, že pre akúkoľvek premennú je potrebné vyhodnotiť všetky kombinácie hodnôt, ktoré môže výraz nadobúdať. Počet riadkov v tabuľke, *n*, je určený vzťahom (počet premenných = *x*): $n = 2^x$.

Premenné teda spolu vytvoria *n* dvojkových čísel. Keď ich chceme generovať po stĺpcoch, tak pozorujeme, že sa 0 na 1-tku mení s periódou násobkov dvojky.

$$p = \frac{2^x}{k}; 2|k \wedge (2 \leq k \leq 2^x)$$

Pri troch premenných dostaneme 3 vektory pravdivostných hodnôt a postupne ich zadanými operáciami spočítame. Ešte je potrebné dodať, že pri implementácii sú premenné (A, B, C) zamenené za indexy vektorov v tabuľke (0, 1, 2) a odkazovanie pokračuje pri vykonávaní všetkých ďalších operácii.

A	0.	`[0, 0, 0, 0, 1, 1, 1, 1]`
B	1.	`[0, 0, 1, 1, 0, 0, 1, 1]`
C	2.	`[0, 1, 0, 1, 0, 1, 0, 1]`
A & B	0. & 1. = 3.	`[0, 0, 0, 0, 0, 0, 1, 1]`
~(A & B)	~3. = 4.	`[1, 1, 1, 1, 1, 1, 0, 0]`
C => ~(A & B)	2. => 4. = 5.	`[1, 1, 1, 1, 1, 1, 1, 0]`

Implementácia – Funkčný program

Tento rozbor som založil na konkrétnom programe, ktorý som napísal, aby som sa konečne mohol zbaviť nekreatívnych a zdĺhavých vypisovacích úloh (alebo aby som si ich mohol aspoň ľahko skontrolovať). Zdrojový kód v programovacom jazyku Python nájdete na:

https://github.com/etakerim/Math-ComputationalUtilities/blob/master/apps/logicke-tabulky.py