# Cvičenie 1 – Funkcie a while-programy

## Funkcie

Totálna funkcia – každá hodnota z prvej množiny má v druhej množine svoj (unikátny) obraz

Parciálna funkcia – nie každý prvok z prvej množiny musí mať svoj obraz v druhej množine (napr. funkcie, ktoré obsahujú delenie nulou, nie sú pre daný koreň menovateľa definované, a preto sú len parciálne)

Injektívna funkcia - žiadne 2 rôzne prvky z prvej množiny sa nezobrazujú na ten istý prvok z druhej množiny

Surjektívna funkcia - všetky hodnoty z druhej množiny sú zobrazením nejakej hodnoty z prvej množiny (1:1)

Bijektívna funkcia – je surjektívna aj injektívna zároveň

## While programy

* Imperatívne programy
* Premenné sú globálne – ak sa zmenia počas programu, a potom ich chce použiť iný program, držia si zmenenú hodnotu
* Ak nie sú inicializované, potom sa rovnajú nule

***Var*** – množina všetkých identifikátorov premenných

**Valuácia premenných** – totálna funkcia *Var → N* – stav, teda prostredie, v ktorom sa náš program vykonáva. Táto funkcia teda dostane identifikátor premennej a vráti jej celočíselnú hodnotu.

***Env*** – množina všetkých valuácii

Napr. *v1 =* {*x1* → 16; *foo* → 4} je valuácia *v1*, ktorá premennej *x1* priraďuje hodnotu 16 a premennej *foo* hodnotu 4. Všetky ostatné premenné majú hodnotu 0. Keďže *v1* je funkcia, môžeme použiť aj zápis: *v1*(*foo*)= 4 (kde *foo* je identifikátor premennej).

Keď vieme, ako sa budú naše programy vyhodnocovať, môžeme interpretovať program *P* ako **čiastočnú** (program sa môže zacykliť) funkciu, ktorá ako vstup berie počiatočnú valuáciu (pred spustením programu), a na výstupe vracia novú koncovú valuáciu, ktorá je platná po vykonaní celého programu: [*P*] : *Env → Env*.

Napr. pre program *P2*

begin

a := b + 1;

while b != zero do b:= b – 1;

end

a iniciálnu valuáciu *v1* ={*b* → 5} platí, že [*P1*](*v1*) = { *a* → 6} (*b* je po skončení programu 0, teda ju nemusíme uvádzať, a *zero* je zas 0 defaultne, keďže nie je definovaná v iniciálnej valuácií.

Takáto funkcia nám stačí na popis toho, „ako“ program počíta (ako sa iniciálna valuácia mení na koncovú valuáciu), ale nehovorí nám, „čo“ program vlastne počíta. Na to sa musíme zhodnúť, čo sú vstupy a výstupy programu, keďže program môže počítať rôzne podľa toho, čo sú jeho vstupné parametre, a čo považujeme za jeho výstup.

K programu *P* teda definujeme tzv. **sémantickú funkciu**, a to nasledovne:

, kde

Ak program pre danú valuáciu cyklí, je aj hodnota nedefinovaná. Všimnime si, že iniciálnu valuáciu tvoríme pomocou argumentov sémantickej funkcie, pričom prvý argument vkladáme do premennej , druhý do , atď. Návratová hodnota je ale vždy uložená v prvej premennej, teda v .

Toto môže byť nepraktické z hľadiska čitateľnosti, preto si definujeme „makro“, ktoré nám umožní ľahko povedať, čo sú vstupné a výstupné premenné programu – na začiatku programu toto budeme uvádzať pomocou príznakov *input* a *output*.

Ak hľadáme while-program, ktorý niečo počítača, typicky nás nezaujíma, akú má tento program zložitosť. Uprednostňujeme krátky, zrozumiteľný program, nezáležiac na jeho zložitosti (pokiaľ samozrejme vieme ukázať, že skončí a spočíta správny výsledok) pred optimalizovaný nezrozumiteľným programom.

Pri vkladaní programov do iných programov nepoužívame volanie funkcii, len použijeme názov programu ako príznak, že sa má na dané miesto skopírovať daný program. Stále platí, že všetky premenné sú naprieč celým vzniknutým programom globálne, preto môžeme chcieť v programoch („vnútorných“ či v tom „vonkajšom“ premenným vytvoriť kópiu, pokiaľ si chceme zachovať jej vstupnú hodnotu).

# Cvičenie 2 – Vyčísliteľnosť funkcii

Často krát nás nezaujíma, ako vyzerá konkrétny program, ktorý nejakú funkciu počíta, ale len to, že takýto program existuje.

Zavedieme si preto pojem ***vyčísliteľný funkcia***.

## Vyčísliteľná funkcia

Taká funkcia, ktorá je sémantickou funkciou nejakého programu. Teda ak je funkcia vyčísliteľná, existuje *nejaký* program, ktorý ju počíta. Dokonca ich existuje nekonečne mnoho, keďže k základnej funkcionalite stačí pridať ľubovoľné množstvo nič nerobiacich inštrukcií. Stále ale ide obecne o parciálne funkcie, nakoľko ich programy môžu cykliť.

## Numerácie

Aby sme sa mohli vo while-programoch odvolávať na vyčísliteľné funkcie, musíme ich nejako zakódovať do číselnej hodnoty, s ktorou môžeme naďalej vo while programe pracovať. Na toto nám bude slúžiť tzv. ***numerácia***.

**Numerácia** – numerácia nejakej množiny *A* je (väčšinou totálna) surjektívna funkcia . Teda pre každý prvok existuje aspoň jeden **index** taký, že . Potom hovoríme, že *i*je indexom *a* /index nie je jednoznačný, každé *a* môže mať viacero indexov, ale vždy musí mať aspoň jeden). Obecne nie je nutné, aby bola numerácia vyčísliteľná, ale väčšinou je to nutné na to, aby sa dala na niečo ďalej použiť (aby nám bola daná numerácia na niečo platná).

Napr.: Označme *P* ako množinu všetkých prvočísel. Funkcia

je numerácia množiny *P*. Každé prvočíslo má aspoň jeden index (seba samého), a prvočíslo 3 ich má nekonečne veľa (seba samého, a všetky neprvočísla). Takáto numerácia je aj vyčísliteľná, keďže test na prvočíseľnosť vieme implementovať while programom.

Dokážeme dokonca numerovať aj všetky while programy. Intuitívne: každý program sa v počítači ukladá vo forme binárneho reťazca, ktorý môžeme interpretovať ako jedno (obrovské) číslo. Týmto kódovaním vieme teda definovať aj numeráciu programov. Takáto numerujúca funkcia bude parsovať číslo na vstupe (reprezentujúce validný program v danom jazyku) ako reťazec, a jej výsledkom bude daný program. Ak parsovanie zlyhá (reťazec nie je validný program), funkcia vráti nejakú vopred definovanú hodnotu.

Podobne môžeme všetky programy numerovať aj pomocou prvočíselného kódovania (technicky náročnejšie, ale v princípe rovnaké).

Numeráciu (unárnych) vyčísliteľných funkcii potom definujeme tak, že index *i* sa mapuje na unárnu sémantickú funkciu programu s indexom *i*. Takúto vyčísliteľnú funkciu s indexom *i* budeme potom označovať ako , a jej odpovedajúci program bude . Teda každá vyčísliteľná funkcia má aspoň jeden index (keďže existuje program, ktorý ju počíta, a ten má index), ale zároveň má nekonečne veľa indexov, keďže existuje nekonečne veľa programov, ktoré ju počítajú. Túto definíciu vieme zovšeobecniť aj pre ľubovoľný počet argumentov.

Otázka, či je veľmi ťažká (pričom ale otázka, či je jednoduchá), keďže musíme rozhodnúť, či dva rôzne programy dané indexami *i* a *j* počítajú tú istú funkciu. Teda pri tejto reprezentácii môžeme použiť klasickú číselnú rovnosť na rozhodnutie otázky „sú *i* a *j* rovnaké implementácie tej istej funkcie?“, rovnako ako môžeme použiť syntaktickú rovnosť programov. To ale neuľahčuje otázku, „sú *i* a *j* dve rôzne implementácie tej istej funkcie?“. Toto je primárny dôvod, prečo definujeme „len“ numeráciu, a nie kompletnú bijekciu (aj keď by bolo príjemné mať pre každú funkciu práve jeden unikátny identifikátor, ale na to by sme museli vedieť rozhodnúť „ťažký“ problém rovnosti funkcii).

## Univerzálna funkcia

Čo ak máme funkciu , ktorú ale dopredu nepoznáme, a chceme umožniť jej vloženie ako vstup pre funkciu . môže ako vstupný parameter brať index nejakej vyčísliteľnej funkcie, ale bez znalosti jej programu nemôžeme so súčasnými znalosťami skonštruovať program pre . Každý index funkcie v sebe ale kóduje zároveň aj jej program. Potrebujeme teda nejaký „interpret“, ktorý by dokázal „spustiť“ funkciu danú jej indexom.

Takýto interpret sa nazýva **univerzálna funkcia**:

ktorá berie za parametre index funkcie, ktorú má spustiť, a vstup, s ktorým ju má spustiť. Dôkaz vyčísliteľnosti tejto funkcie je pomerne technický, ale jej existencia a vyčísliteľnosť nie je prekvapivá (predstavme si to ako napísanie interpretu Pythonu v Pythne – pomerne pracné, ale možné). Univerzálna funkcia však nie je totálna. Ak cyklí, cyklí aj univerzálna funkcia.

Aby sme mohli univerzálnu funkciu používať vo while programoch, definujme si jej makro

kde sa univerzálna funkcia nahradí programom, ktorý ju počíta. Môžeme si rovno definovať aj makro pre volanie univerzálnej funkcie, teda zápis

Príklad: Je funkcia vyčísliteľná?

Áno. Uvažujme program . Najskr sa spustí funkcia s indexom *x* a vstupom *x*. Pokiaľ sa tento výpočet skončí, vrátime *y*, inak samozrejme celý program cyklí.

*Príklad*: Dokážte, že

*Dôkaz*:

Použili sme: definíciu univerzálnej funkcie, definíciu sémantickej funkcie while programu, a fakt, že nedefinované premenné sú v každej valuácií nastavené defaultne na nulu.

## Veta o parametrizácií

Keď už vieme, že môžeme funkcie „spúšťať“. Asi by sme chceli ešte ukázať, že ich vieme aj kombinovať. Inak povedané, že existuje spôsob, akým vieme nejaké existujúce funkcie/programy spojiť do jedného pomocou iného programu. Teda, že existuje „kompilátor“, ktorý dokáže pracovať s programami, modifikovať ich, a tvoriť nové.

Za týmto účelom existuje **veta o parametrizácií** (ktorej dôkaz je opäť primárne o práci s kódovaním programov):

*„Existujú totálne vyčísliteľné funkcie “*

Intuitívne môžeme túto vetu chápať tak, že funkcia predstavuje istú formu vzoru/predlohy/šablóny, do ktorej „kompilátor“ *s* dosadzuje hodnoty , a vyrába z nej nový program. Teda transformácia/kombinácia programov, ktorú sme schopní popísať v šablóne , sa dá vykonať programovo pomocou *s*.

*Príklad*: Ukážte, že funkcia *g(i, j)*, kde je totálne vyčísliteľná.

*Riešenie*: Ako vidíme, funkcie *g* berie dve funkcie a vracia novú funkciu, ktorá počíta ich súčet. Teda *g*  nepočíta samotný súčet, ale vracia program, ktorý po spustení (s ďalšími parametrami) ten súčet spočíta.

Uvažujme nasledujúci program ako našu šablónu:

Potom = . Použitím vety o parametrizácií teda môžeme definovať = (*c* je známa konštanta – identifikátor šablóny). Keďže *s* je totálne vyčísliteľná, bude aj *g* totálne vyčísliteľná (spustíme *s* so vstupnými argumentami a konštantou *c*).

## Nevyčísliteľné funkcie

Existuje mnoho funkcií, ktoré nie sú vyčísliteľné.

*Príklad*: Ukážte, že funkcia *halt(i)* nie je vyčísliteľná:

*Dôkaz – sporom:*

Pre spor predpokladajme, že funkcia *halt* je vyčísliteľná. Teda existuje index *h* taký, že . Potom uvažujme funkciu *flip(i):*

Funkcia *flip* je zjavne tiež vyčísliteľná (spustí sa funkcia *halt*, a podľa výsledku program buď cyklí, alebo skončí), teda existuje index *f* taký, že .

Čo sa však stane, ak použije index funkcie *flip* ako vstup pre funkciu *flip*? Buď sa funkcia zacyklí, alebo nie:

1. - spor: ak sa flip zacyklí, tak sa nezacyklí
2. - spor: ak sa flip nezacyklí, tak sa zacyklí

nemôže byť vyčísliteľná funkcia.

*Príklad*: Ukážte, že funkcia *f(x, y)* nie je vyčísliteľná.

*Riešenie*: dôkaz by sme mohli viesť podobne, ako pre *halt*, avšak nakoľko ten už máme, môžeme si dokazovanie pomocou neho zjednodušiť:

Pre spor predpokladajme, že *f* je vyčísliteľná. Potom ale platí, že *halt(i) = f(i, 1)*. Teda ak existuje program pre *f*, existuje program aj pre *halt*. To sme ale v predchádzajúcom príklade vyvrátili, takže *f* nemôže byť vyčísliteľná.

Práve sme teda dokázali, že pre while programy neexistuje algoritmus, ktorý by vždy presne v konečnom čase určil, či program zastaví, alebo nie. Takéto tvrdenie sa dá potom dokázať aj pre iné dostatočne silné programovacie jazyky. Veľa užitočných programov toto obmedzenie však nemá . pokiaľ sa obmedzíme napr. len na for-cykly so známym počtom opakovaní, naše program vždy zaručene zastavia, a stále budeme schopní zoraďovať čísla alebo prehľadávať grafy, ale nebudeme môcť naimplementovať napr. Collatzovu domnienku (viď originál prvé cvičenie).

Prečo sa teda nezaoberáme len totálnymi funkciami? Z praktického hľadiska by určite bolo fajn, keby máme programovací jazyk, v ktorom by sme mohli napísať programy pre všetky totálne funkcie. Bohužiaľ, (vyčísliteľný) jazyk, ktorý by dokázal popísať všetky totálne funkcie, tiež existovať nemôže.

*Príklad*: Dokážte, že neexistuje efektívna numerácia *totálne vyčísliteľných* funkcií s univerzálnou funkciou. Teda, že neexistuje numerácia s vyčísliteľnou univerzálnou funkciou , kde sú všetky totálne vyčísliteľné funkcie.

*Dôkaz – sporom*:

Predpokladajme, že existuje táto numerácia a univerzálna funkcia . Potom uvažujme funkciu . Funkcia je určite totálna ( je totálna, keďže počíta totálne funkcie), a je tiež vyčísliteľná (keďže je vyčísliteľná). Teda funkcia musí byť súčasťou takejto numerácie, keďže je totálna a vyčísliteľná, teda pre nejaký index *o*. Potom ale platí, že . Z tohto sporu potom vyplýva, že náš predpoklad bol nesprávny, a takáto univerzálna funkcia existovať nemôže.

Takýto dôkaz nutne nehovorí, že numerácia totálnych funkcií neexistuje. Dôležitá vlastnosť je tu vyčísliteľnosť. Ak by sme nepožadovali vyčísliteľnosť, tak funkcia ako skonštruovať môžeme, lebo nemám k dispozícií . Z praktického hľadiska by nám bol ale takýto výsledok dosť zbytočný, keďže pomocou neho nič „nenaprogramujeme“.

# Cvičenie 3 – Rozhodnuteľnosť

Keď už máme zavedené, čo to znamená niečo počítať (while programy), a predstavu o tom, čo všetko sa dá a nedá spočítať (vyčísliteľné a nevyčísliteľné funkcie), môžeme sa zaoberať ďalšími dôležitými otázkami: *Čo je riešiteľný problém? Existujú rôzne druhy „riešiteľnosti“?*

Na to si musíme najskôr pripomenúť, čo je problém.

**Problém** – problém A je ľubovoľná podmnožina prirodzených čísel , napr. množina , alebo (indexy funkcií, ktoré pre vstup 1 vracajú 42).

V informatike však často potrebujeme pracovať s inými typmi objektov, než sú prirodzené čísla, napr. s grafmi, alebo logickými formulami. Tie sa ale dajú kódovať pomocou prirodzených čísel (graf napr. ako matica susednosti), a teda sa môžeme často stretnúť aj s takto definovanými problémami: , alebo .

## Rozhodnuteľné problémy

Definujeme triedu rozhodnuteľných (rekurzívnych) problémov (množín):

Množina A je rekurzívna práve vtedy, keď existuje totálne vyčísliteľná funkcia taká, že

Teda pre každý vstup vždy skončí a vracia jednotku len ak vstup patrí do A → podľa výsledku teda vieme presne rozhodnúť, či prvok (vstup) do množiny patrí, alebo nie.

### Ako dokázať, že je množina rekurzívna? Teda, že je problém rozhodnuteľný?

Najjednoduchší spôsob je nájsť totálnu funkciu , ktorá náš problém rozhoduje. Pokiaľ toto riešime pre konkrétny problém, znamená to vyriešiť daný problém (opäť nás nezaujíma zložitosť nášho riešenia, iba to, že je totálne). Napr. množina je rekurzívna, lebo riešením tohto problému je jednoduché overenie, že zvyšok po delení prvku 13-timy je rovný 0.

### Ako dokázať, že množina nie je rekurzívna?

Aby sme našli nejaké množiny, ktoré nie sú rekurzívne, zameriame sa na iné objekty, o ktorých „neexistencií“ už vieme – nevyčísliteľný funkcie.

Uvažujme množinu (zastaví pre vstup *x*). Množinu K budeme nazývať problém zastavenia. Množina K nie je rekurzívna.

*Dôkaz*: Pre spor predpokladajme, že K rekurzívna je, teda existuje funkcia , ktorá ju rozhoduje. Potom si ale môžeme všimnúť, že *halt =* , a teda ak vyčísliteľná je, je vyčísliteľná aj funkcia *halt*. O tej sme však už dokázali, že vyčísliteľná nie je. Dochádza teda k sporu, a K nemôže byť rekurzívna.

Existenciu tejto nerekurzívnej množiny môžeme ďalej použiť pri argumentácii o iných podobných množinách (viď originál.)

Dokazovanie nám uľahčuje Prvá Riceova veta.

## Prvá Riceova veta

Neformálne si môžeme všimnúť, že rovnosť dvoch funkcií nie je vyčísliteľná, teda ak by sme ju pri vyhodnocovaní nášho problému museli vyriešiť, asi sa nebude jednať o rekurzívnu množinu (toto je len intuitívny argument). Formálne sa táto vlastnosť nazýva **zachovávanie funkcií**.

**Zachovávanie funkcií** – hovoríme, že množina zachováva funkcie, ak pre každú funkciu platí, že množina buď obsahuje všetky jej indexy, alebo žiaden. Napr. množina funkcie zachováva (funkcia pre vstup 1 vracia 42 bez ohľadu na to, aký index tejto funkcie uvažujeme). Na druhej strane, množina K funkcie nezachováva. Uvažujme funkciu, ktorá cyklí pre všetky možné vstupy okrem jednej konštanty *c*, pričom táto konštanta je zároveň aj jedným z indexov jej programov. Potom , ale všetky ostatné indexy tejto funkcie do K nepatria.

**Prvá Riceova veta** potom hovorí, že ak je množina A netriviálna (nie je prázdna a nie je ani N) a zachováva funkcie, nie je rekurzívna. je zjavne netriviálna (neobsahuje napr. index prázdnej funkcie, teda nie je N, ale obsahuje indexy funkcií, ktoré konštantne vracajú 42, teda nie je prázdna), a zároveň, ako už bolo spomenuté, zachováva funkcie (formálny dôkaz: Nech *a* a *b* sú dva rôzne indexy tej istej funkcie. Potom , a táto hodnota je buď 42, alebo nie, teda ak *a* patrí do tak aj *b* patrí do , a opačne). Teda podľa prvej Riceovej vety nie je množina rekurzívna.

Ako sme ale videli na príklade množiny K, nie všetky ne-rekurzívne množiny zachovávajú funkcie, teda prvá Riceova veta sa nedá použiť vždy (vtedy musíme použiť dôkazy podobné tomu, aký sme použili pôvodne).

## Čiastočne rozhodnuteľné problémy

Existuje teda nejaký medzistupeň medzi rozhodnuteľným a nerozhodnuteľným problémom? Napr. problém zastavenie nie je rekurzívny, ale pre každý program, ktorý zastavuje, vieme potvrdiť, že skutočne zastaví. Problémom je, že nevieme potvrdiť aj nezastavenie. Avšak ak je odpoveď pozitívna, vieme ju dokázať.

Táto myšlienka je formalizovaná v koncepte **čiastočne rozhodnuteľných problémov** (resp. **rekurzívne spočetných množín** – R.E. – recursively enumerable). R.E. množiny sa dajú definovať niekoľkými spôsobmi, pričom tieto definície sú ekvivalentné, avšak nie každá je v každej situácii rovnako praktická.

1. Množina A je R.E., ak existuje vyčísliteľná funkcia *f* taká, že *A = domain(f)*. Teda A je definičným oborom nejakej vyčísliteľnej funkcie.
2. Množina A je R.E., ak je prázdna, alebo existuje (totálne) vyčísliteľná funkcia *f* taká, že *A = range(f)*. Teda A je oborom hodnôt nejakej (totálne) vyčísliteľnej funkcie, alebo inak povedané, existuje funkcia *f*, ktorá je numeráciou množiny A. (*f* sa často uvažuje ako totálna, ale v skutočnosti to nie je nutné – z netotálnej numerácie sme potom schopní vyrobiť pomocou malého triku aj totálnu)

Každá rekurzívna množina je aj R.E. Keďže pre rekurzívnu máme jasne spočítateľné, kedy do nej prvok patrí a kedy nie, nie je problém skonštruovať funkciu, ktorá sa zacyklí, keď do danej množiny prvok nepatrí, a tým splníme podmienku prvej definície. Niekedy sa teda môžeme v zadaní stretnúť s pojmami ako „množina je R.E. ale nie je rekurzívna“, aby bolo jasné, že pre takúto množinu nie len nemusí ale ani nemôže existovať exaktná rozhodovacia funkcia. Na druhej strane, keď sa povie, že množina je R.E., neznamená to automaticky, že nemôže byť aj rekurzívna.

### Ako dokázať, že množina je R.E.?

Podobne, ako pri rekurzívnych množinách, ak máme daný konkrétny problém, najľahšie je nájsť funkciu, ktorá by spĺňala jednu z našich definícií. Napr. pre problém zastavenia je to triviálne: pokiaľ naša funkcia *f* pre vstup *x* spustí pomocou univerzálnej funkcie , bude platiť, že *domain(f) = K*. Pokiaľ riešime nejaké obecnejšie tvrdenie, musíme zase vychádzať z definície, podobne, ako pri rekurzívnych množinách (akurát si môžeme vybrať, ktorú definíciu použijeme, prípadne ich môžeme aj striedať).

**R.E. množiny nie sú uzavreté na doplnok**. Sporom: Uvažujme netriviálnu R.E. množinu *B* a jej doplnok . Pre spor predpokladajme, že je tiež R.E.. Potom existujú numerácie a podľa definície (2) – pre jednoduchosť ich nazvyme a . Uvažujme nasledujúci program:

input : x; output : y;

begin

found := 0; n := 0;

while found = 0 do

begin

if r1(n) = x || r2 (n) = x do

found := 1;

n := n + 1;

end

if r1(n-1) = x do y := 1; else y := 0;

end

Tento program potom definuje rozhodovaciu funkciu pre množinu B. Prečo? O každom čísle *x* vieme, že buď patrí do B alebo patrí do . Teda sa nutne musí objaviť v obore hodnôt alebo (a samozrejme sa nemôže objaviť v oboch súčasne). Môžeme teda postupne prechádzať obe numerácie a hľadať, v ktorej z nich sa nachádza *x*. Ak je v B, vraciame 1, ak v , vraciame 0.

Ukázali sme tak, že ak by boli všetky R.E. množiny uzavreté na doplnok, tak sú všetky rekurzívne. To by ale znamenalo, že aj problém zastavenia je rekurzívny, čo je spor.

Zaujímavý dôsledok tejto vlastnosti je aj to, že sme práve ukázali existenciu množín, ktoré nie sú ani rekurzívne, ani R.E. – sú to (okrem iného) aj doplnky nerekurzívnych R.E. množín. Teda doplnok problému zastavenia, , nie je ani R.E. (dáva zmysel aj intuitívne – sú to programy, ktoré nezastavujú nad svojím vlastným indexom – teda nemám spôsob, ako potvrdiť, že program nezastaví.).

### Step counter

Počas dokazovania R.E. budeme často potrebovať simulovať výpočet programu pre konečný počet krokov (teda intuitívne „spustiť program s timeoutom“). Tento simulátor sa nazýva *Step counter Sc*, a ide v podstate o upravenú verziu univerzálnej funkcie (opäť je to totálne vyčísliteľná funkcia).

Uvažujme množinu . Pomocou definície (1) opäť jednoducho ukážeme, že množina je R.E. – stačí spustiť funkciu na vstupe pre hodnotu 1 a počkať, či vráti 42. Ak nie, cyklí. Čo ak by sme ale túto množinu trochu upravili: – nemáme pevne daný vstup, pre ktorý sa má 42 vrátiť. Triviálne by sme mohli spúšťať funkciu *i* pre rôzne vstupy, a hľadať 42, pokiaľ by sme ale narazili na vstup, kde *i* cyklí, nemáme sa ako posunúť ďalej.

Teda musíme pokusy vo vhodnom čase zastaviť, a skúšať iné hodnoty:

input i;

begin

n := 0; found := 0;

while found = 0 do

begin

x := n; k:= 0;

while k != n + 1 do:

begin

if Sc(i, x, k) = 1 and phi\_i(x) = 42 do

found := 1;

k := k + 1; x := x -1;

end

n := n + 1;

end

end

(Predpokladáme, že logické spojky v *if* sa vyhodnocujú lenivo – teda sa spočíta až vtedy, keď step counter dopredu potvrdí, že výpočet s týmto vstupom skončí.)

Vieme, že ak *i*patrí do , existuje vstup *x* a počet krokov *k* taký, že zastaví do *k* krokov a vráti 42. Na to, aby sme ho našli, potrebujeme vyskúšať všetky dvojice hodnôt *x* a *k*. Teda potrebujeme postupne prechádzať tabuľku dvojíc hodnôt. Pokiaľ by sme však niečo také skúšali robiť po stĺpcoch či riadkoch, nepodarí sa nám to, pretože tie sú nekonečné. Môžeme ich však prechádzať po diagonálach – najskôr začneme kombináciou (0, 0), potom na druhej diagonále (n = 1) skúsime  
(0, 1) a (1, 0). Na tretej (n = 2) vyskúšame (0, 2), (1, 1), a (2, 0), atď.

Takýmto spôsobom vieme, že nech už je kombinácia *k* a *x*, ktorá nám vráti 42, akákoľvek, nachádza sa na nejakej diagonále, a my sa k nej v konečnom čase prepracujeme, a nájdeme ju. Ak taká dvojica neexistuje, budeme samozrejme tabuľku stále ďalej, a program sa nám efektívne zacyklí. Zostrojili sme teda funkciu, ktorej definičný obor je .

Takáto operácia je pomerne bežná pri práci s R.E. množinami, teda často predpokladáme existenciu tzv. projekčných funkcií a , ktoré nám dokážu unikátne „z jedného čísla vyrobiť dve“ (máme bijekciu medzi číslami a dvojicami čísel). Teda máme napr. *id(0, 0) = 0*, a (3) = 2.

Pomocou step counteru a projekcie môžeme potom napr. ukázať, ako z netotálnej numerácie v definícií (2) urobiť totálnu (a teda, že sú definície bez a s totálnosťou ekvivalentné). Nech A je neprázdna množina, *r* je index jej netotálnej numerácie a *a*je nejaký prvok z tejto množiny. Uvažujme potom napr. takýto program:

input x; output y;

begin

if Sc(r, pi1(x), pi2(x)) = 1 do

y := phi\_r(pi2(x));

else

y := a;

end

Takáto numerácia vždy zastaví, no a ak pôvodná numerácia vracala nejaký prvok *y* pre vstup *x* po *k* krokoch, tak táto numerácia ho vráti pre vstup *id(x, k)*. Teda obor hodnôt tejto numerácie je zhodný s oborom hodnôt pôvodnej numerácie.

Ako dokázať, že množina nie je R.E.?

Okrem iného, použitím ďalších Riceových viet.

## Druhá Ricevoa veta

Ak A je netriviálna a rešpektuje funkcie (teda nie je rekurzívna podľa prvej Ricevoej vety), a zároveň platí, že existujú dve funkcie a také, že a , pričom je rozšírením , tak A nie je R.E.

Funkcia *g* rozširuje funkciu *f*, ak je *g* definovaná pre všetky hodnoty, pre ktoré je definovaná *f*, ale môže byť definovaná aj pre hodnoty, pre ktoré *f* definovaná nie je (teda ).

*Príklad*: funkcia *halt* je rozšírením funkcie

(*halt* je definovaná aj keď nezastaví.)

Akákoľvek funkcia je rozšírením prázdnej funkcie (môžeme pridať hocičo). Jediným rozšírením totálnej funkcie je táto funkcia sama (nemáme čo viac pridať).

Čo to znamená? Znamená to, že ak existuje netotálna funkcia, ktorá patrí do A, ale rozšírením tejto netotálnej funkcie sa viem dostať „von“ z A, tak A nie je R.E. To okrem iného znamená, že táto Riceova veta sa nedá použiť na dôkaz množiny . Keďže všetky funkcie v tejto množine sú totálne, nemám ich ako rozšíriť.

Dá sa ale použiť na dôkaz množiny . Uvažujme prázdnu funkciu a jej rozšírenie – identitu. Prázdna funkcia nie je totálna, teda patrí do , ale identita je totálna, teda do nepatrí. teda nemôže byť R.E. (samozrejme treba ešte overiť, že je triviálna a rešpektuje funkcie, to je ale ľahké, keďže sme práve našli funkciu, ktorá tam patrí a funkciu, ktorá tam nepatrí, a aj je funkcia netotálne, všetky jej indexy sú netotálne).

Pri hľadaní vhodnej funkcie na dôkaz je dobré začať s „najmenšou“ (najmenej definovanou) funkciou, ktorá patrí do A. To je často práve prázdna funkcia (ak prázdna funkcia patrí do netriviálnej množiny, a množina zachováva funkcie, tak nemôže byť R.E. – ľubovoľná funkcie, ktorá nie je v A, bude rozšírením prázdnej funkcie, a teda bude splňovať podmienky). Ak do množiny prázdna funkcia nepatrí, dá sa „najmenšia“ funkcia často dovodiť z definície. Tiež sa dá zamerať na konečné vs. nekonečné funkcie.

## Tretia Riceova veta

Ak A je netriviálna a rešpektuje funkcie (teda nie je rekurzívna) a zároveň existuje funkcia  taká, že , a pre každé konečné zúženie platí , tak A nie je R.E..

**Zúženie** – opačný pojem k rozšíreniu – odoberám prvky miesto pridávania. Konečné zúženie je zúženie, ktorého definičný obor je konečný.

*Príklad*: Funkcia, ktorá rozhoduje problém zastavenia pre prvých 1000 programov, je konečným zúžením funkcie *halt* (a je dokonca vyčísliteľná, lebo všetky funkcie s konečným definičným oborom sú vyčísliteľné). Prázdna funkcia má jediné zúženie – samú seba.

Čo to znamená? Ak máme nejakú funkciu, ktorá patrí do A, ale všetky jej „konečné podčasti“ do A nepatria, tak A nie je R.E. Taktiež si všimnime, že ak množina obsahuje prázdnu funkciu, nedá sa táto veta použiť, lebo prázdna funkciu je konečným zúžením ľubovoľnej funkcie.

Dá sa ale použiť na dôkaz množiny . Identita je totálna funkcia, ktorá patrí do , ale jej ľubovoľné konečné zúženie nie je totálne, teda do nepatrí ( dokonca neobsahuje žiadnu konečnú funkciu, takže je úplne jedno, akú funkciu z si vyberieme, žiadne konečné zúženie tam nikdy nebude). Netriviálnosť a zachovávanie funkcií platí, teda nie je R.E.

Pri hľadaní vhodnej funkcie pre dôkaz je dobré zasa hľadať čo najjednoduchšiu neprázdnu funkciu, ktorá patrí do danej množiny. Častokrát si ale človek vystačí s identitou alebo rôznymi konštantnými funkciami. Dôležité ale je, že funkcia musí mať nekonečný definičný obor, inak je sama sebe konečným zúžením, a teda nikdy nemôže ako dôkaz fungovať.