Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika

ŠTÁTNY PEDAGOGICKÝ ÚSTAV
NATIONAL INSTITUTE FOR EDUCATION

Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika

Počítačové systémy 4

Predmet: Počítačové systémy

Línia: Vlastný odborový kontext informatiky a informatickej výchovy

Operačný program VZDELÁVANIE

EURÓPSKA ÚNIA
Európsky sociálny fond

Európska únia
Európsky sociálny fond

Moderné vzdelávanie pre vedomostnú spoločnosť/Projekt je spolufinancovaný zo zdrojov ES

Počítačové systémy 4

Identifikácia modulu

Aktivita projektu:	1.2 Vzdelávanie nekvalifikovaných učiteľov informatiky na 2. stupni ZŠ a na SŠ
Línia aktivity:	Vlastný odborový kontext informatiky a informatickej výchovy
Predmet:	Počítačové systémy

Garant predmetu:

RNDr. Peter Gurský, PhD.
ÚINF PF UPJŠ, Košice
peter.gursky@upjs.sk

Autori:

RNDr. František Galčík, PhD.
ÚINF PF UPJŠ, Košice
frantisek.galcik@upjs.sk

doc. RNDr. Stanislav Krajči, PhD.
ÚINF PF UPJŠ, Košice
stanislav.krajci@upjs.sk

Zaradenie modulu

Modul nadväzuje na prvé tri moduly predmetu Počítačové systémy. Účastníci ho absolvujú vo štvrtom semestri vzdelávania. Tento modul využíva poznatky získané vo viacerých predmetoch predchádzajúceho štúdia (Programovanie, Algoritmy a štruktúry údajov, Počítačové systémy, Matematika pre učiteľov informatiky 1 až 3).

Abstrakt modulu

Modul sa skladá z dvoch samostatných častí. Prvá časť nazvaná *Interpretery a kompilátory* priblíži základné idey, údajové štruktúry a algoritmy využívané interpretermi pri interpretovanom vykonávaní programov. Stručne budú priblížené aj niektoré činnosti vykonávané pri kompilácii programov. Dôraz je kladený na realizáciu výpočtov s využitím zásobníka, mechanizmus volania podprogramov a algoritmický preklad aritmetických výrazov na postupnosť jednoduchších inštrukcií. Druhá kapitola s názvom *Kódovanie* oboznámi čitateľa s dôvodmi a základnými princípmi kódovania informácií. Jej prvá časť sa venuje bezprefixovému kódovaniu včítane jeho grafického znázornenia vo forme (spravidla binárneho) stromu. Definuje sa tu i pojem entropie dôležitý okrem iného aj ako dobrá aproximácia odhadu tzv. ceny kódovania. Na záver prvej časti sú ukážky dvoch známych algoritmov kódovania, a to Shannonovho-Fanovho a optimálneho Huffmanovho. Druhá časť kapitoly je venovaná samoopravným kódom.

Obsah

Úvod

Milá čitateľka, milý čitateľ, počas predchádzajúceho štúdia ste sa zoznámili s mnohými základnými princípmi, pojmami a algoritmami rôznych častí tak širokej oblasti, akou informatika nesporne je. V tomto module využijeme veľa z nich. Modul sa skladá z dvoch (zdanlivo) samostatných častí: „Interpretery a kompilátory“ a „Kódovanie“. Pre obe časti je spoločná idea, že „prepísanie“ čohosi do inej formy vedie často k oveľa efektívnejším riešeniam. V prvej časti budú to „čohosi“ počítačové programy. Ukážeme si, ako program z vyššieho programovacieho jazyka „prepísať“ do iného programovacieho jazyka s jednoducho realizovateľnými inštrukciami tak, aby tento program robil presne to isté. Pozrieme sa tiež, ako vykonávanie tohto „prepísaného“ programu môžeme odsimulovať pomocou iného programu. V druhej časti sa pod tým „čohosi“ budú skrývať údaje. Údaje, ktoré je treba ukladať na pamäťové média či prenášať počítačovou sieťou. V tejto časti si ukážeme, že „prepísaním“ údajov (ich zakódovaním) do inej formy dokážeme použiť oveľa menej pamäťového miesta, než by sa na prvý pohľad zdalo. Dokonca ich prepisom do „inej“ formy dokážeme v prípade, že niečo zlyhá, poškodené údaje (uložené na pamäťovom médiu alebo prednášané sieťou) spätne zrekonštruovať, resp. toto poškodenie odhaliť.

Cieľ modulu

Po absolvovaní tohto modulu sa od účastníka vzdelávania očakáva, že

- bude rozumieť základným princípom činnosti interpreterov a spôsobu preloženia programu z vyššieho programovacieho jazyka do nižšieho,
- porozumie dôvodom i základným princípom kódovania informácií a zoznámi sa s niektorými konkrétnymi typmi kódovania.

Vstupné vedomosti

Požadované prerekvizity

Všetky moduly Programovanie 1 až 9, Algoritmy a štruktúry údajov 1 a 2, Počítačové systémy 1 a 2, Matematika pre učiteľov informatiky 1 až 3.

Predpokladané vstupné vedomosti, skúsenosti a zručnosti

Predpokladáme, že účastník vzdelávania:

- vie napísať jednoduchý program, ovláda algoritmy a údajové štruktúry prebrané v predmete Algoritmy a štruktúry údajov, pozná architektúru procesora Intel x86 a základné príkazy v jazyku asemblera,
- je zoznámený so základnými matematickými pojmami (včítane logaritmu) a je schopný upravovať algebraické výrazy

Interpretery a kompilátory

Zamýšľali ste sa niekedy nad tým, čo všetko sa deje „v počítači“ počas behu programu, s ktorým práve pracujete, či akým spôsobom sa tento program vykonáva?

V moduloch *Programovanie* sme sa naučili vytvárať vlastné programy v jazyku *Object Pascal* v prostrediach Delphi a Lazarus. Z pohľadu týchto modulov sa za behom programu skrýva postupné vykonávanie príkazov, ktoré sú reakciou na aktivitu a vstupy používateľa. Zvyčajne sú tieto príkazy napísané programátorom v niektorom z vyšších programovacích jazykov (napr. Pascal, C/C++, Java, PHP, Python, ...). Jednotlivé príkazy vyššieho programovacieho jazyka pracujú s premennými, realizujú rôzne výpočty, pomocou podmienkových príkazov (if-y) umožňujú ovplyvniť „čo sa stane ďalej“, s využitím cyklov dokážeme opakovať nejakú postupnosť príkazov, atď. Možnosti sú široké. Vieme tiež, že „slušný“ zdrojový kód a aj jeho vykonávanie je rozčlenené do podprogramov – procedúr a funkcií, ktoré sa môžu navzájom volať s rôznymi hodnotami parametrov.

Spustený program (aj ten napísaný v Lazaruse alebo Delphi) beží na počítači. V predchádzajúcich moduloch *Počítačové systémy* sme sa dozvedeli, že počítač je v skutočnosti len „kopa“ navzájom poprepájaných elektrických obvodov realizujúcich jednoduché aritmeticko-logické operácie a umožňujúcich uchovanie bitových hodnôt (v registroch, cache pamäti, operačnej pamäti alebo inom pamäťovom médiu). Jadrom počítača je procesor. Ak odhliadneme od hardvérových detailov, na procesor môžeme pozerať ako na zariadenie schopné postupne vykonávať jednoduché strojové inštrukcie. To, ktoré inštrukcie je schopný procesor vykonávať, a to, ako sú zakódované, záleží od konkrétnej architektúry procesora. Ak si vezmeme architektúru procesora Intel x86, pomocou strojových inštrukcií dokážeme prekopírovať obsah medzi registrami a pamäťou, vykonať jednoduché aritmeticko-logické operácie nad registrom, atď. Niektoré registre majú špeciálny význam – napr. IP (Instruction Pointer) obsahuje adresu strojovej inštrukcie, ktorá sa má vykonať ako nasledujúca. Keďže programovanie pomocou číselných kódov strojových inštrukcií je nepohodlné (tak sa robilo len pri úplne prvých počítačoch), programátori na ich zápis používajú jednoduchý mnemonický jazyk asemblera. Takto zapísané inštrukcie ide programom (nazývaným asembler) viac-menej priamočiaro zakódovať („transformovať“) do číselných inštrukcií strojového kódu. Najväčšou nevýhodou programov napísaných v strojovom kóde je ich slabá prenositeľnosť, keďže sú zviazané s konkrétnou architektúrou procesora. A rôznych architektúr procesorov je dnes veľmi veľa.

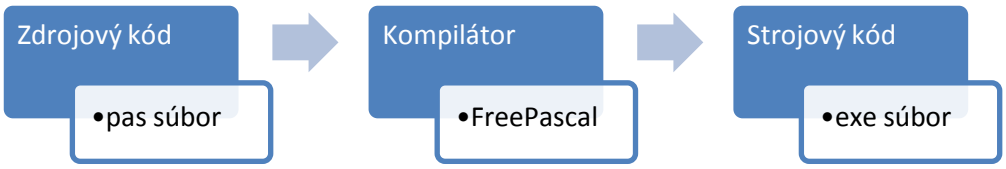
Z predchádzajúcich modulov teda doposiaľ poznáme dva pohľady na programy - dva svety: „svet vyšších programovacích jazykov“ (Pascalu) a „svet hardvéru, procesora a strojových inštrukcií“. Cieľom tejto časti tohto modulu je pozrieť sa na „miesto“, kde sa tieto svety stretávajú, a na problémy, ktoré sa v tejto oblasti riešia. Táto oblasť je veľmi rozsiahla a každý pokus o jej ucelenejšie priblíženie presahuje rozsah tohto modulu. Pozrieme sa preto len na niektoré zaujímavé problémy, ktoré nám túto oblasť aspoň trochu priblížia. Pokúsime sa načrtnúť odpovede na otázky: Ako to vyzerá v pamäti počas behu programu? Čo sa stane, keď zavolám vo svojom programe procedúru alebo funkciu? Akým spôsobom sa program vo vyššom programovacom jazyku preloží do strojového kódu („exe“-čka)?

1.1 Interpreter vs. kompilátor

S pojmom **kompilácia** sme sa už stretli v moduloch venovaných programovaniu. Vždy, keď sme v prostredí Lazarus nechali spustiť program, ako jedna z prvých činností sa udiala kompilácia zdrojového kódu (projektu). V prostredí Lazarus kompiláciu realizuje samostatný program – kompilátor FreePascal-u. Tento kompilátor je súčasťou Lazarus-u a o jeho existencii pri základnom programovaní ani nemusíme tušiť. Lazarus ho spúšťa automaticky pri každom spustení vytváranej aplikácie. Ak zdrojový kód aplikácie nie je korektný, výsledkom kompilácie sú chybové hlášky (tie Lazarus zobrazí pekne v okienku so správami o kompilácii). Naopak, ak je zdrojový kód bez chýb, výsledkom kompilácie je vytvorenie spustiteľného programu – „exéčka“. O jeho vytvorení sa môžeme presvedčiť tak, že sa pozrieme do priečinka, v ktorom máme uložený projekt. Až keď máme ako

výsledok kompilácie spustiteľný súbor, prostredie Lazarus ho automaticky spustí a program sa začne vykonávať. **Kompilátorom** nazývame počítačový program, ktorý transformuje zdrojový kód v programovacom jazyku (zdrojový jazyk) do iného počítačového jazyka (cieľový jazyk). Najčastejšie je cieľovým jazykom kompilácie strojový kód alebo jazyka asemblera. Najčastejším dôvodom kompilácie je vytvorenie spustiteľného programu. Činnosť kompilátora, podobne ako práca tlmočníka, nie je jednoduchá. Program, ktorý je výsledkom kompilácie, musí dodržiavať pravidlá cieľového jazyka a zároveň musí vykonávať (vyjadrovať) presne to isté, čo program zapísaný v zdrojovom jazyku. A aby toho nebolo málo, býva zvykom, že kým zdrojový jazyk má zvyčajne bohaté a komplexné výrazové prostriedky (cykly, podprogramy, premenné, ...), cieľový jazyk umožňuje používať len niekoľko veľmi jednoduchých typov inštrukcií (viď jazyk asemblera). Kvôli rôznorodosti dnešných počítačových architektúr a operačných systémov (zo strojového kódu sa často volajú služby operačného systému, ktoré sú špecifické pre jednotlivé operačné systémy) pre mnohé programovacie jazyky nájdeme kompilátory z týchto jazykov do strojového kódu viacerých platforiem.

Na stránke http://wiki.freepascal.org/Platform_list je možné vidieť, pre ktoré operačné systémy a procesory (CPU) existuje kompilátor FreePascal-u.



Obrázok 1: Schéma kompilácie

Určite ste sa stretli s programami, ktoré sa nešíria vo forme samostatne spustiteľných súborov – „exéčiek“, ale na ich spustenie ste potrebovali mať nainštalovaný špeciálny program, pomocou ktorého sa tieto programy dali spustiť. Napríklad, ak chceme spustiť Java programy v súboroch s príponou *jar*, potrebujeme mať nainštalovaný *Java Runtime Environment* (JRE). Na spustenie projektu v Imagine (v súbore s príponou *imp*) potrebujeme mať nainštalovaný program Imagine. Na to, aby sa nám na webovej stránke spustila aplikácia vo „Flash“-ku, potrebujeme mať nainštalovaný *Adobe Flash player*. Na spustenie PHP skriptu nám treba napríklad webový server podporujúci PHP. Programy, pomocou ktorých sa tieto programy spúšťajú a vykonávajú, sú príkladmi interpreterov. **Interpreter** je počítačový program, ktorý priamo vykonáva príkazy zapísané v nejakom programovacom jazyku. Interpreter si môžeme predstaviť ako program, ktorý číta zdrojový kód riadok za riadkom a postupne vykonáva príkazy v jednotlivých riadkoch zdrojového kódu. Uvedomme si, že pri interpretovanom vykonávaní programu príkazy vykonáva program, t.j. iný softvér, kým pri spustení spustiteľného programu v strojovom kóde sú inštrukcie vykonávané priamo procesorom, t.j. hardvérom. Výhodou distribúcie programov vo forme, ktorá má byť interpretovaná interpreterom, je oproti distribúcii samostatne spustiteľných programov nezávislosť na konkrétnej počítačovej platforme. Keďže existuje JRE pre Windows, Linux aj mobilné telefóny, po vytvorení Java programu, ho dokážeme spustiť na každej z týchto platforiem. Presnejšie, dokážeme ho spustiť na každej z platforiem, pre ktorú bol vytvorený interpreter programov v jazyku Java. Nevýhodou je nutná prítomnosť interpretera kvôli spusteniu programu.

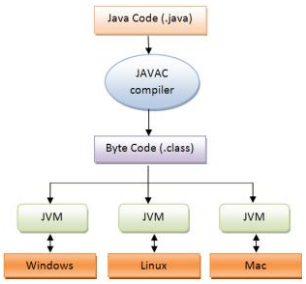
V prípade Javy, ako bude vysvetlené neskôr, sa neinterpretuje priamo zdrojový kód, ale program v prechodnom binárnom tvare nazývanom byte-code (bajt-kód).

V súčasnosti existuje viacero typov interpreterov. Najstarším typom interpreterov sú interpretery, ktoré priamo vykonávajú (interpretujú) zdrojový kód. Do tejto skupiny môžeme zaradiť QBasic, PHP alebo webovými prehliadačmi interpretovaný JavaScript. Nevýhodou týchto interpreterov je to, že pri každom spustení programu sa musí vykonávať pomerne náročné spracovanie a analýza zdrojového kódu.

V texte budeme využívať nespisovný (slangový) výraz „parsovanie“ na označenie nejakej formy analýzy zdrojového kódu, resp. textu. Dôvodom je to, že v IT komunite je tento výraz zaužívaný a jeho pomerne dlhé slovenské ekvivalenty sa v odbornej komunite používajú len zriedka.

Riešením problému s náročnou analýzou zdrojového kódu („parsovaním“ zdrojového kódu) vznikol ďalší typ interpreterov. Zdrojový kód sa najprv kompilátorom preloží do formy prechodného programu (zvyčajne v binárnom tvare), ktorá umožňuje jeho efektívnejšie interpretovanie (vykonávanie). Jazyk prechodnej formy sa zvyčajne skladá z jednoduchých inštrukcií nezávislých od konkrétnej architektúry alebo platformy. Pri tomto type interpreterov sa program najčastejšie distribuuje v prechodnej forme, nie v zdrojovom kóde. V prípade Javy sa v jar súboroch

Kompilácia a vykonávanie programov v Jave:



nachádza program v jazyku prechodnej formy (v binárnom tvare) nazývanom *byte-code* (bajt-kód). Program v byte-code je interpretovaný tzv. Java virtuálnym strojom (*Java Virtual Machine*), ktorý je súčasťou JRE (*Java Runtime Environment*). Populárnou platformou sa dnes stala aj platforma *.Net* z dielne spoločnosti Microsoft. Podobne ako v Jave, aj tu sa zdrojový kód kompiluje do prechodnej formy v jazyku CIL (*Common Intermediate Language*). Pri spustení sa potom programy v prechodnej forme interpretujú virtuálnym strojom, ktorý je súčasťou súboru programov nazývaných *.Net Framework*. Kompilácia programov do prechodnej formy je v súčasnosti využívaná mnohými interpretermi.

Hlavnou nevýhodou interpretovaného vykonávania programov je spomalenie spôsobené vykonávaním rôznych podporných činností a analýzy vykonávaného kódu. Významné zlepšenie (zrýchlenie) priniesli interpretery, ktoré neinterpretujú priamo program v jazyku prechodnej formy. Pred samotným interpretovaním sa časť programu v prechodnej forme, ktorá sa má najbližšie vykonávať, interne prekompiluje do strojového kódu a až ten sa vykoná (priamo na hardvéri). Tento spôsob kompilácie sa označuje ako **Just-In-Time kompilácia** a je využívaný aj vo virtuálnych strojoch (interpreteroch) Javy a .Net Framework-u.

1.2 Interpreter „zásobníkového“ jazyka

Aby sme si priblížili pozadie fungovania interpreterov a sčasti aj kompilátorov, definujeme si vlastný programovací jazyk s menom **DSL** (*DVUi Stack Language*). Jazyk DSL bude nezávislý od architektúry (na rozdiel od jazyka asemblera) a zároveň dostatočne jednoduchý na to, aby sme napísanie jeho interpretera zvládli s doterajšími programátorskými znalosťami. Programovací jazyk DSL bude mať tieto vlastnosti:

- podporované budú len niektoré celočíselné operácie a premenné,
- v každom riadku môže byť len jeden príkaz (pozor, na rozdiel od Pascalu nepoužívame bodkočiarky na oddelenie či ukončenie príkazov),
- každý riadok môže začínať návestím – slovno-číselným pomenovaním riadka, ktoré je ukončené dvojbodkou,
- názov každého identifikátora v programe (návestie riadka, premennej, podprogramu, …) je neprázdna postupnosť písmen anglickej abecedy, číslic a znaku _ (podčiarkovník), ktorá nereprezentuje číslo
- programovací jazyk nie je „case-sensitive“, t.j. pri použití identifikátorov nezáleží na veľkosti písmen.

Základné príkazy jazyka DSL:

- `DEFINT meno`
  - definícia premennej: vytvorí premennú celočíselného typu so zadaným menom,
- `GOTO navestie`
  - príkaz skoku: presunie vykonávanie programu na príkaz označený zadaným návestím,
- `premenna := DSL_numerický_výraz`
  - príkaz priradenia: do premennej, ktorej identifikátor je naľavo od `:=` uloží hodnotu určenú numerickým výrazom napravo od `:=`,
- `IF premenna THEN prikaz`
  - podmienkový príkaz: ak hodnota premennej, ktorej identifikátor sa nachádza medzi `IF` a `THEN`, je **nenulová**, potom sa vykoná príkaz nachádzajúci sa za kľúčovým slovom `THEN`
- `IFNOT premenna THEN prikaz`
  - podmienkový príkaz: ak hodnota premennej, ktorej identifikátor sa nachádza medzi `IF` a `THEN`, je **nulová**, potom sa vykoná príkaz nachádzajúci sa za kľúčovým slovom `THEN`

Namiesto logických hodnôt **TRUE**/**FALSE** používame numerické určenie pravdivostnej hodnoty rovnakým spôsobom, ako to je v programovacom jazyku C/C++. Hodnota 0 reprezentuje logickú hodnotu **FALSE**, nenulová hodnota logickú hodnotu **TRUE**.

Numerickým výrazom v jazyku DSL môže byť konkrétna celočíselná hodnota – literál (hodnota výrazu je daná touto hodnotou), identifikátor premennej (hodnotou výrazu je aktuálna hodnota uložená v tejto premennej) alebo volanie podprogramu (hodnotou výrazu je číselná hodnota vrátená ako výsledok volania podprogramu - funkcie). Teda na rozdiel od Pascalu sa na pravej strane príkazov priradenia nemôžu

(kvôli jednoduchosti implementácie) nachádzať aritmetické operácie.

Základné vstupno-výstupné operácie nám v jazyku DSL poskytnú príkazy:

- `PRINT(premenna)`
  - vypíše aktuálnu hodnotu premennej
- `premenna := GETINPUT`
  - od používateľa načíta hodnotu do zadanej premennej

Na dočasné uloženie hodnôt v jazyku DSL môžeme použiť okrem premenných aj údajovú štruktúru zásobník. Zásobník v jazyku DSL však bude poskytovať aj príkazy umožňujúce realizáciu jednoduchých aritmetických výpočtov:

- `PUSH(premenna)` alebo `PUSH(hodnota)`
  - vloží na vrch zásobníka aktuálnu hodnotu premennej, resp. zadanú hodnotu
- `premenna := POP`
  - vyberie prvok z vrchu zásobníka a jeho hodnotu uloží do zadanej premennej
- `STACKADD, STACKSUB, STACKMUL, STACKDIV`
  - vyberie z vrchu zásobníka 2 hodnoty: prvá vybraná nech je B a druhá vybraná nech je A, a na vrch zásobníka vloží hodnotu A+B, A-B, A*B, resp. A div B
- `STACKLEQ, STACKEQ, STACKGEQ`
  - vyberie z vrchu zásobníka 2 hodnoty: prvá vybraná nech je B a druhá vybraná nech je A, a na vrch zásobníka vloží hodnotu 1, ak A≤B, A=B, resp. A≥B, inak 0
- `STACKNEG`
  - zneguje hodnotu na vrchu zásobníka - ak tam bola 0, bude tam 1 a ak tam bola nenulová hodnota, bude tam 0

Na jednoduchú aritmetiku môžeme v jazyku DSL použiť tieto príkazy:

- `premenna1 := PRED(premenna2)`
  - do premennej `premenna1` uloží hodnotu uloženú v premennej `premenna2` zníženú o 1
- `premenna1 := SUCC(premenna2)`
  - do premennej `premenna1` uloží hodnotu uloženú v premennej `premenna2` zvýšenú o 1

Na prvý pohľad je zrejmé, že takto navrhnutý jazyk DSL má podstatne menší počet príkazov oproti tomu, čo poznáme z jazyka Pascal. Na druhej strane je tento jazyk podobne, ako je to charakteristické pre vyššie programovacie jazyky ako Pascal, nezávislý na konkrétnej architektúre - nehovoríme tu nič o registroch, prístupe do pamäte, spôsoboch adresácie či iných hardvérových špecifikách. Stačí nám jednoduchý koncept premennej a zásobníka s aritmeticko-logickými operáciami. Pozrime sa, ako by mohol vyzerať program, ktorý bude od používateľa načítavať čísla až dokiaľ sa nezadá ako vstup číslo 0. Po skončení načítavania program vypíše súčet zadaných čísel.

Inšpiráciou pre túto časť jazyka DSL bol p-kód - strojový kód hypotetického procesora, tzv. „zásobníkového stroja“. P-kód bol špecifikovaný Niklausom Wirthom - tvorcom programovacieho jazyka Pascal. Neskoršie implementácie prvých kompilátorov jazyka Pascal nekompilovali pascalovský kód do strojového kódu, ale do p-kódu. Program v p-kóde bol následne buď interpretovaný alebo kompilovaný do strojového kódu pre konkrétnu cieľovú platformu. Na p-kód môžeme dnes pozerať aj ako na jazyk prechodnej formy interpretovaných jazykov.

Jazyk DSL:	Jazyk PASCAL:

DEFINT cislo
DEFINT sucet
 	
sucet := 0
start_cyklu:
cislo := **GETINPUT**
PUSH(cislo)
PUSH(sucet)
STACKADD
sucet := **POP**
IF cislo **THEN GOTO** *start_cyklu*
PRINT(sucet)
	var cislo: Integer;
	var sucet: Integer;
	begin
	  sucet := 0;
	  repeat
	  readln(cislo);
	  sucet := sucet + cislo;
	
	  until cislo=0
	  writeln(cislo);
	end;

Úloha 1.1	Odsimulujte vykonávanie vyššie uvedeného programu v jazyku DSL podľa opisu príkazov jazyka DSL.
Úloha 1.2	Vytvorte program v jazyku DSL, ktorý od používateľa načíta číslo N a vypíše súčet čísel od 1 po N. Program vyskúšajte v pripravenom interpreteri jazyka DSL.

Riešením úlohy môže byť napríklad nasledujúci program v jazyku DSL:

```
DEFINT n
DEFINT sucet

sucet := 0
n := GETINPUT
start_cyklu:
PUSH(n)
PUSH(sucet)
STACKADD
sucet := POP
n := PRED(n)
IF n THEN GOTO start_cyklu
PRINT(sucet)
```

Príkaz GOTO je jediný príkaz v jazyku DSL (a tiež v prvých programovacích jazykoch), ktorý umožňuje meniť riadenie toku programu. S jeho pomocou dokážeme vytvoriť ekvivalenty všetkých programových konštrukcií (cykly, podmienky, vetvenia, …), ktoré poznáme z jazyka Pascal. Táto inštrukcia sa ľahko prekladá do jazyka asemblera Intel x86 - zodpovedá inštrukcii skoku JMP.

Všimnime si v ňom niekoľko špecifík jazyka DSL. Jazyk DSL neumožňuje jedným príkazom realizovať sčítanie obsahu dvoch premenných, avšak celkom dobre si vieme poradiť so zásobníkom s aritmetickými operáciami (i keď namiesto jedného príkazu v Pascale, v jazyku DSL potrebujeme 4 príkazy). Príkazy realizujúce výpočet `n+sucet` v tomto programe majú tú vlastnosť, že po ich vykonaní je obsah výpočtového zásobníka rovnaký, ako bol pred začiatkom výpočtu, t.j. výpočet po sebe nezanecháva „žiadne stopy“. To je veľmi dôležité v prípade, kedy by bol tento výpočet realizovaný ako podčasť (podvýpočet) nejakého iného výpočtu a zásobník by už obsahoval hodnoty nejakého „rozrobeného“ výpočtu. Ďalším špecifikom jazyka DSL je to, že na riadenie následnosti vykonávania príkazov (riadenie toku programu) musíme používať príkaz skoku - príkaz GOTO, resp. v kombinácií s príkazom IF podmienkový príkaz skoku. Takýto spôsob riadenia toku programu je blízky jazyku asemblera a strojovému kódu, v ktorých sa ďalšia inštrukcia na vykonanie určuje zmenou obsahu registra IP (Instruction Pointer).

Môžete predpokladať, že máte k dispozícii hotovú postupnosť príkazov, ktorá do nejakej vami určenej premennej vypočíta pravdivostnú hodnotu splnenia podmienky cyklu.

Úloha 1.3	Vytvorte návod („algoritmus“), podľa ktorého by ste vedeli každý jednoduchý for-cyklus prepísať do jazyka DSL. V menších skupinkách potom podobným spôsobom vytvorte návody aj pre pascalovské konštrukcie for-downto-cyklus, if-then-else, while-do-cyklus, repeat-until-cyklus, atď.

Štruktúra jednoduchého Pascalovského for-cyklu je nasledovná:

```
for i:=a to b do prikaz(y);
```

Prepis tejto konštrukcie do jazyka DSL by mohol vyzerať napríklad takto:

```
DEFINT porovnanie
i := a
opakuj:
PUSH(i)
PUSH(b)
STACKLEQ
porovnanie := POP
IFNOT porovnanie THEN GOTO koniec
... príkazy cyklu v jazyku DSL ...
```

```
i := SUCC(i)
GOTO opakuj
koniec:
...
```

Po vytvorení schém („návodov") na prevod základných programových konštrukcií z vyššieho programovacieho jazyka (napr. Pascalu), by sme už mali zvládať pretransformovať skoro každý program v tomto jazyku do jazyka DSL. Podobne sa pri kompilácii z vyššieho programovacieho jazyka do jazyka asemblera zdrojový kód prekladá použitím „predpripravených" transformačných schém.

Úloha 1.4

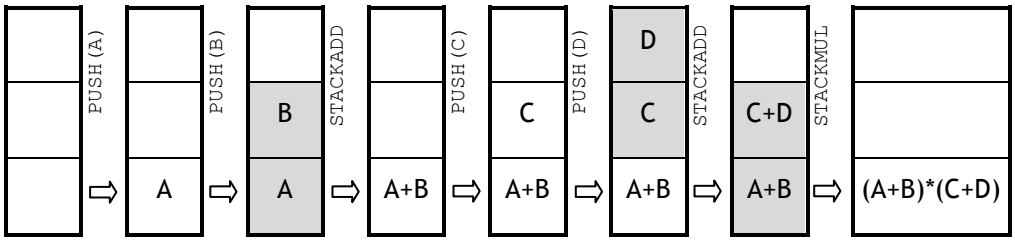
Napíšte postupnosť príkazov jazyka DSL, ktorá bez použitia pomocných premenných zrealizuje pascalovský príkaz priradenia: `A := (A+B)*(C+D);`

Riešením predchádzajúcej úlohy môže byť takáto postupnosť inštrukcií.

```
PUSH(A)
PUSH(B)
STACKADD
PUSH(C)
PUSH(D)
STACKADD
STACKMUL
A := POP
```

Odsimulovaním jednotlivých inštrukcií s konkrétnymi hodnotami premenných ľahko nadobudneme presvedčenie o správnosti tejto postupnosti príkazov. Ak by sme sa chceli o jej správnosti presvedčiť formálnejšie, tak tieto inštrukcie odsimulujeme so všeobecnými hodnotami:

Na to, ako systematicky previesť aritmeticky výraz do postupnosti príkazov, sa pozrieme v kapitole 1.4.



Úloha 1.5

Diskutujte: Ak jazyk Pascal patrí medzi vyššie programovacie jazyky a jazyk asemblera medzi nižšie programovacie jazyky, kam by ste zaradili jazyk DSL?

Pravdepodobne dôjdete k názoru, že jazyk DSL je niekde „medzi". Má len veľmi jednoduché programové konštrukcie, no zároveň je nezávislý od architektúry. Tieto vlastnosti ho robia napríklad dobrým kandidátom pre prechodný jazyk využívaný interpretermi, v ktorých sa najprv zdrojový kód kompiluje do prechodného jazyka a až program v tomto jazyku (zapísaný v binárnom tvare) sa interpretuje.

Po zoznámení sa s jazykom DSL, je čas pozrieť sa na to, ako by sme si interpreter tohto programovacieho jazyka naprogramovali. Vykonávanie programu spočíva v postupnom realizovaní jednotlivých príkazov programu. Preto je celkom prirodzená stratégia vytvoriť si podprogram (procedúru), ktorý zrealizuje jeden príkaz (v našom prípade jeden riadok) v zdrojovom kóde programu. Takýto prístup nám umožní navyše aj jednoduché krokovanie programu. Medzi jednotlivými krokmi interpretovania príkazov programu si potrebujeme uchovať všetko, čo charakterizuje aktuálny stav behu programu - potrebujeme si uchovať aktuálny **kontext** vykonávania príkazov. Inšpirovať sa môžeme činnosťou počítačového procesora. V ňom sú jednotlivé inštrukčné cykly z vonkajšieho pohľadu nezávislé.

Pri hibernovaní počítača tiež dochádza k uloženiu aktuálneho stavu celého systému, ktorý sa po návrate z hibernácie obnoví. Keďže obsah disku je trvalo uložený (nestratí sa po vypnutí počítača), pri hibernovaní potrebujeme uložiť najmä aktuálny obsah operačnej pamäte.

V každom inštrukčnom cykle sa na základe obsahu registrov a pamäte (kontextu vykonávania), ktorý je výsledkom predchádzajúcej činnosti procesora, zrealizuje zmena obsahu registrov a pamäte podľa inštrukcie, ktorej číselný kód sa nachádza na adrese určenej obsahom registra IP, t.j. inštrukcie, ktorá je taktiež určená obsahom registra a pamäte (kontextom). Ak si všimneme jednotlivé príkazy jazyka DSL, uvidíme, že každý stav výpočtu (kontext vykonávania) je charakterizovateľný aktuálnym obsahom definovaných premenných a obsahom zásobníka. Pri interpretovaní programu ešte potrebujeme vedieť, ktorý príkaz (inštrukcia) sa má vykonať ako ďalší. Inšpirujúc sa registrom IP v architektúre Intel x86, postačí nám medzi údajové štruktúry uchovávajúce kontext pridať celočíselnú premennú, ktorá bude uchovávať číslo riadka s príkazom, ktorý sa má vykonať ako ďalší. Výsledkom predchádzajúcich úvah môže byť takáto údajová štruktúra záznamového typu charakterizujúca aktuálny stav výpočtu (kontext):

```
type
  TPremenna = record
    Meno: string;
    Hodnota: Integer;
  end;

  TKontext = record
    RiadokInstrukcie: Integer;

    Premenne: array[1..MaxPremenne] of TPremenna;
    PocetPremennych: Integer;

    Zasobnik: array[1..MaxZasobnik] of Integer;
    VelkostZasobnika: Integer;
  end;

var
  Kontext: TKontext;
```

Ak parameter procedúry alebo funkcie označíme kľúčovým slovom **var**, tento parameter bude odovzdaný referenciou. V porovnaní s odovzdávaním parametra hodnotou (to sme používali v moduloch *Programovanie*) premenná predstavujúca parameter nie je lokálnou premennou podprogramu, ale **zastupuje** premennú, ktorá bola v tomto parametri uvedená pri volaní podprogramu.

```
procedure
 PRef(var X:Integer)
begin
  X := 10;
end;

procedure
 PHod(X:Integer)
begin
  X := 10;
end;

...
var Y: Integer;
...
Y := 20;
//v Y je 20
PHod(Y);
//v Y je 20
PRef(Y);
//v Y je 10
```

S využitím tejto údajovej štruktúry sa jadrom interpretera stane procedúra, ktorá na základe obsahu záznamovej premennej `Kontext` vykoná ďalší príkaz programu. Základná schéma tejto procedúry môže vyzerať napríklad takto:

```
procedure Krok(KodProgramu: TRiadky; var Kontext: TKontext);
var
  Prikaz: string;
begin
  Prikaz := KodProgramu[Kontext.RiadokInstrukcie];
  VykonajPrikaz(Prikaz, Kontext);
  if Prikaz nie je GOTO then
    Inc(Kontext.RiadokInstrukcie)
  else
    Kontext.RiadokInstrukcie := VratRiadokNavestia(Prikaz);
end;
```

V tejto procedúre sa volá procedúra *VykonajPrikaz*, ktorá podľa aktuálneho kontextu vykonávania tento príkaz interpretuje (vykoná). Kvôli prehľadnosti programu je v tejto procedúre najlepšie vykonať len základné „rozparsovanie“ príkazu (textová analýza reťazca s príkazom za účelom zistenia, aký príkaz sa tam nachádza) a podľa toho zavolať ďalšiu procedúru, ktorá už vykoná pre daný príkaz špecifické parsovanie a následne aj tento príkaz interpretuje. Náročnosť implementácie parsovania zdrojového kódu záleží aj od toho, ako je programovací jazyk navrhnutý. Napríklad, ak by sme dovolili mať v jednom riadku viac príkazov (ako to je v Pascale), určite by bola implementácia náročnejšia, než v našom prípade, kedy je v každom riadku len jeden príkaz. Aj pridanie blokov programu (begin/end) s možnosťou vnárania by viedlo k náročnejšej implementácii parsovania – príslušné begin-y a end-y by sme museli výpočtovo náročne párovať. Príkladom jazyka s nenáročným parsovaním je jazyk asemblera. Každá inštrukcia sa nachádza v jednom riadku a skladá sa z mena, za ktorým nasleduje čiarkami oddelený zoznam

parametrov inštrukcie. Inštrukcia môže byť navyše označená aj návestím.

Úloha 1.6	Navrhnite, ako by ste implementovali interpretovanie (vykonanie) príkazov `PUSH`, `STACKMUL`, `STACKDIV`, `GOTO` a príkazu priradenia.
Úloha 1.7	Na základe návrhov v predchádzajúcej úlohe doplňte chýbajúce implementácie vykonania príkazov `STACKMUL` a `STACKDIV` v pripravenom interpreteri jazyka DSL. Implementácie doplňte v procedúre *VykonajVolanie*, ktorá ako parametre dostane názov podprogramu (inštrukcie), pole hodnôt parametrov a počet parametrov. Do programu doplňte aj vlastnú inštrukciu (napr. `STACKABS`, ktorá hodnotu na vrchu zásobníka nahradí jej absolútnou hodnotou, či `PRINT` s ľubovoľne veľa parametrami).
Úloha 1.8	Diskutujte: Ako zložité môže byť naprogramovať prekladač (kompilátor) z vyššieho programovacieho jazyka do jazyka DSL a z jazyka DSL do jazyka asemblera Intel x86?

1.3 Volanie a beh podprogramov

Základom dobrého programátorského štýlu je rozumné rozdelenie programu a jeho funkcionality do menších jednotiek - podprogramov (procedúr a funkcií). Okrem toho už hotové podprogramy (napr. rôzne knižničné procedúry a funkcie) výrazne zjednodušujú vývoj ďalších programov. Rozšírme preto jazyk DSL o možnosť vytvárania a volania podprogramov. Procedúry a funkcie nebudeme vzájomne rozlišovať - každý podprogram bude funkciou, t.j. bude vracať nejakú hodnotu. Záleží len na zdrojovom kóde, ktorý volá podprogram, či vrátenú hodnotu nejako využije, t.j. priradí ju do premennej. Podprogramy budeme definovať pomocou kľúčového slova `SUBROUTINE`, ktoré označuje začiatok definície podprogramu a je nasledované hlavičkou podprogramu, a kľúčového slova `END`, ktoré označuje koniec definície podprogramu. Spôsob vytvárania podprogramov v jazyku DSL si ozrejmíme na príklade podprogramu *Mocnina*, ktorý počíta hodnotu a^n, kde a a n sú parametre podprogramu:

```
SUBROUTINE Mocnina(a, n)
  result := 1
  cyklus:
  IFNOT n THEN EXIT
  PUSH(result)
  PUSH(a)
  STACKMUL
  result := POP
  n := PRED(n)
  GOTO cyklus
END

DEFINT c
c := Mocnina(4, 3)
PRINT(c)
```

Všimnime si, že na vrátenie hodnoty z podprogramu používame (podobne ako v Object Pascale) preddefinovanú premennú `result` a to takým spôsobom, že hodnota v nej uložená v dobe ukončenia vykonávania podprogramu sa berie ako podprogramom vrátená hodnota. Inšpirujúc sa jazykom Pascal, pre vytváranie a vykonávanie podprogramov v jazyku DSL budú platiť tieto pravidlá:

- každá premenná definovaná v podprograme je lokálna premenná, t.j.

prestáva existovať po ukončení vykonávania podprogramu (po návrate z podprogramu),

- hlavička podprogramu sa skladá z mena podprogramu, za ktorým sa môže v okrúhlych zátvorkách nachádzať čiarkami oddelený zoznam mien parametrov – parametre sú z pohľadu vykonávania programu lokálne premenné, ktorých obsah je určený hodnotami parametrov pri volaní podprogramu,
- premenné v hlavnom programe nie sú prístupné v podprogramoch, hlavný program je „špeciálnym“ podprogramom, ktorý sa spúšťa automaticky ako štartovací podprogram,
- každé vykonávanie podprogramu má vlastný výpočtový zásobník,
- príkazom GOTO môžeme skočiť len na návestie definované v danom podprograme,
- príkazom EXIT je možné predčasne ukončiť vykonávanie podprogramu,
- po návrate z podprogramu vykonávanie programu pokračuje príkazom, ktorý nasleduje za príkazom, ktorý spôsobil volanie podprogramu.

Úloha 1.9	V jazyku DSL napíšte podprogram na výpočet faktoriálu.
Riešenie	**SUBROUTINE** Faktorial(n) result := 1 cyklus: **IFNOT** n **THEN EXIT** **PUSH**(result) **PUSH**(n) **STACKMUL** result := **POP** n := **PRED**(n) **GOTO** cyklus **END**

Pri programovaní interpretera s podporou vytvárania a volania podprogramov sú spomedzi všetkých skôr spomenutých vlastností podprogramov najdôležitejšie tieto dve: (I) nezávislosť vykonávania podprogramov a (II) pokračovanie vo výpočte po návrate z volaného podprogramu.

Pripomeňme, že v jazyku DSL nemáme globálne premenné a každý podprogram má vlastný výpočtový zásobník. Preto prvú vlastnosť vieme vyriešiť tak, že každé volanie podprogramu bude znamenať akoby spustenie nového behu programu. Z programátorského hľadiska to znamená vytvorenie nového kontextu vykonávania. Tento kontext však nebude uchovávať celkový stav behu programu (obsahy všetkých premenných, zásobníkov podprogramov,...), ale len stav výpočtu volaného podprogramu. Volaný podprogram sa stane vykonávaným podprogramom a novovytvorený kontext bude kontextom pre vykonávanie všetkých príkazov podprogramu až do momentu návratu z tohto podprogramu.

Druhá dôležitá charakteristika volania podprogramov je to, že po návrate z vykonávania volaného podprogramu výpočet volajúceho podprogramu pokračuje ďalej. Toto pokračovanie vo výpočte znamená návrat ku predchádzajúcemu kontextu vykonávania príkazov, ktorý prestal byť aktívny pri volaní podprogramu. Aby sme sa k predchádzajúcemu kontextu mohli vrátiť, musí byť niekde uložený. Nesmieme však zabúdať ani na to, že môže vzniknúť situácia, že podprogram A volá podprogram B a podprogram B počas svojho behu volá podprogram C. V okamihu, keď sa vykonávajú príkazy podprogramu C, kdesi musí byť uložený kontext behu podprogramu B vo chvíli volania podprogramu C. Zároveň musí byť niekde uložený aj kontext behu podprogramu A vo chvíli volania podprogramu B. To je dôležité preto, aby keď skončí vykonávanie podprogramu C, mohol vo svojom behu pokračovať podprogram B. A podobne keď skončí podprogram B, treba pokračovať v behu podprogramu A. Z predchádzajúcich úvah vyplýva, že vhodným riešením problému volania podprogramov je:

- pri každom volaní podprogramu uchovať aktuálny kontext vykonávania a vytvoriť nový „aktívny“ kontext, podľa obsahu ktorého sa bude interpretovať volaný podprogram,
- pri návrate z volaného podprogramu (po skončení jeho vykonávania) obnoviť ako „aktívny“ kontext ten kontext, ktorý bol uložený pri volaní podprogramu.

Úloha 1.10

Diskutujte: Akú údajovú štruktúru, resp. akým spôsobom by išlo implementovať uchovávanie a obnovu kontextov pri volaniach podprogramov? Porovnajte výhody a nevýhody jednotlivých návrhov.

Všimnime si, že nech je postupnosť volaní podprogramov akákoľvek, v každom okamihu sa vykonáva len naposledy volaný podprogram a kontexty ďalších podprogramov v „postupnosti volaní“ sú uložené. V postupnosti volaní sa nový prvok (nové volanie) objaví vždy na konci. Ukončenie vykonávania podprogramu má za následok skrátenie „postupnosti volaní“ o jeden prvok z jej konca. Analogická situácia je s kontextami podprogramov, ktoré je treba uchovávať a obnovovať pri volaní, resp. návrate z podprogramov. Tu je dôležité si uvedomiť, že kontext, ktorý bol uložený ako posledný, bude obnovený ako prvý.

Úloha 1.11

Vytvorme postupnosť uchovávaní a obnovovaní kontextov pri spustení podprogramu **A**, ktorého zdrojový kód je uvedený nižšie.

```
SUBROUTINE A
  B
  C
END
```

```
SUBROUTINE B
  D
  E
END
```

```
SUBROUTINE C
  B
  D
END
```

U podprogramov **D** a **E** (bez uvedenia zdrojového kódu) predpokladáme, že sa v nich nenachádza žiadne volanie podprogramov.

Postupnosť volaní v jednotlivých fázach vykonávania:

A
A**B**
AB**D**
A**B**
AB**E**
A**B**
A
A**C**

...

To, že obnovený (vybraný) je vždy najneskôr uložený kontext, je presne to, čím je charakteristická údajová štruktúra zásobník. V zásobníku je tiež naposledy vložený prvok vždy vybraný najskôr. O tejto štruktúre vieme, že je veľmi jednoducho a efektívne implementovateľná. Pridanie aj odobranie prvku vieme v zásobníku zrealizovať v čase *O(1)*, t.j. v konštantnom čase. Kontexty (otvorených, neukončených) podprogramov preto budeme pri volaní podprogramov ukladať do tzv. **zásobníka volaní** (ang. call-stack). Prvky zásobníka volaní sa zvyknú nazývať rámcami zásobníka (ang. stack-frame). Zásobník volaní (kontextov vykonávaní) môžeme implementovať pomocou poľa záznamov:

```
ZasobnikVolani: array[1..MaxVolani] of TKontext;
VyskaZasobnikaVolani: Integer = 0;
```

Premenná *VyskaZasobnikaVolani* uchováva aktuálny počet prvkov uložených v zásobníku volaní.

Úloha 1.12

Analýzou zdrojového kódu interpretera jazyka DSL preskúmajte, akým spôsobom je implementované volanie podprogramov. Zamerajte sa na procedúry *PridajDoZasobnikaVolani*, *VykonajVolanie* a záverečné príkazy procedúry *VykonajPrikaz*.

Vysvetlenie: Procedúra *VykonajPrikaz* implementuje rozparsovanie (rozdelenie na podreťazce) príkazu volania podprogramu. Pomocou funkcie *PripravHodnotyParametrov* sa podľa aktuálneho kontextu (obsahu premenných) reťazce zodpovedajúce parametrom volania vyhodnotia – určia sa ich príslušné číselné hodnoty. S týmito hodnotami sa volá procedúra *VykonajVolania*. Najprv sa overuje, či nejde o volanie niektorého „zabudovaného“ podprogramu (napr. POP, PUSH, PRINT, atď.). Ak je volaný programátorom (v kóde programu v jazyku DSL) definovaný podprogram, zavolá sa procedúra *PridajDoZasobnika*, ktorá vytvorí kontext v zásobníku volaní a v ňom lokálne premenné pre jednotlivé parametre volaného podprogramu.

Samotné spracovanie volania podprogramu sa skladá z týchto častí:

- **rozparsovanie príkazu** – rozdelíme reťazec na podreťazce zodpovedajúce jednotlivým častiam príkazu: názov volaného podprogramu, výrazy určujúce hodnoty parametrov volania (funkcia *RozparsujVolanie*),
- **vyhodnotenie parametrov** – výrazy (podreťazce zodpovedajúce parametrom volania podprogramu) sa vyhodnotia podľa aktuálneho kontextu vykonávania (funkcia *PripravHodnotyParametrov*),
- **overenie korektnosti volania** – overíme, či podprogram so zadaným menom bol definovaný a ak áno, či zoznam (počet a typ) parametrov volania je kompatibilný s parametrami podprogramu uvedenými v hlavičke definície podprogramu (v našej implementácii je táto časť spojená s volaním podprogramu),
- **volanie podprogramu** - pri volaní „zabudovaného“ podprogramu jeho okamžité vykonanie, pri volaní podprogramu definovaného v zdrojovom kóde vytvorenie nového kontextu vykonávania pre volaný podprogram vrátane vytvorenia lokálnych premenných pre parametre volania.

Ukážku toho, ako implementovať volanie podprogramov a zásobník volaní vo forme zdrojového kódu, možno nájsť v pripravenom interpreteri jazyka DSL.

Zásobník volaní nemusíme využiť len na uloženie kontextov pri volaní podprogramov, ale aj na uloženie aktuálne vykonávaného kontextu. Stačí zaviesť dohodu, že kontext na vrchu zásobníka volaní, je aktuálne vykonávaný kontext. V našom prípade je tak aktuálny kontext v `ZasobnikVolani[VyskaZasobnikaVolani]`. Prázdny zásobník volaní pri tejto implementácii znamená, že vykonávanie programu je ukončené.

V interpreteri jazyka DSL sú údaje o programátorom definovaných podprogramoch uložené v poli záznamov `Podprogramy`. Toto pole je naplnené procedúrou *InicializaciaVykonavania* pred vykonaním prvého príkazu programu v procedúre *Krok*.

Nespomínali sme ešte, ako by sa implementovalo samotné volanie programátorom definovaného podprogramu. Pred samotným volaním potrebujeme zistiť, či vôbec volaný podprogram existuje a ak áno, aké má parametre a na akom riadku (kde) začína vykonávanie tohto podprogramu. Inými slovami potrebujeme nájsť definíciu podprogramu. Máme dve možnosti. Prvá je pri každom volaní podprogramu prehľadať zdrojový kód a pokúsiť sa nájsť definíciu volaného podprogramu. Zjavne táto možnosť nie je veľmi efektívna. Druhá možnosť je vytvoriť si zoznam s údajmi o programátorom definovaných podprogramoch (napr. pole záznamov). Tento zoznam si môžeme vybudovať buď pred začiatkom vykonávania programu alebo dynamicky počas spracovávania riadkov programu počas jeho vykonávania. V tomto druhom prípade vždy, keď narazíme na riadok začínajúci slovom `SUBROUTINE` vieme, že ide o začiatok definície podprogramu a na základe rozparsovania tohto riadka získame základné údaje o tomto podprograme. Nájdením najbližšieho ďalšieho riadka so slovom `END` nájdeme koniec tohto podprogramu. Nevýhodou tohto prístupu je to, že v programe môžeme volať len tie podprogramy, na ktorých definície sme počas vykonávania natrafili. Teda definícia volaných podprogramov sa v zdrojovom kóde musí nachádzať vždy pred miestom ich volania. Takýto spôsob používa napr. interpreter jazyka PHP. Podobný mechanizmus je použitý aj pri kompilovaní v Pascale – odtiaľ pochádza pravidlo, že procedúry a funkcie musia byť definované v zdrojom kóde prv, než je v zdrojovom kóde príkaz, kde sa volajú. Keď už máme definíciu volaného podprogramu, pri jeho volaní nám ostáva vytvoriť nový kontext v zásobníku volaní a vytvoriť v ňom lokálne premenné pre jednotlivé

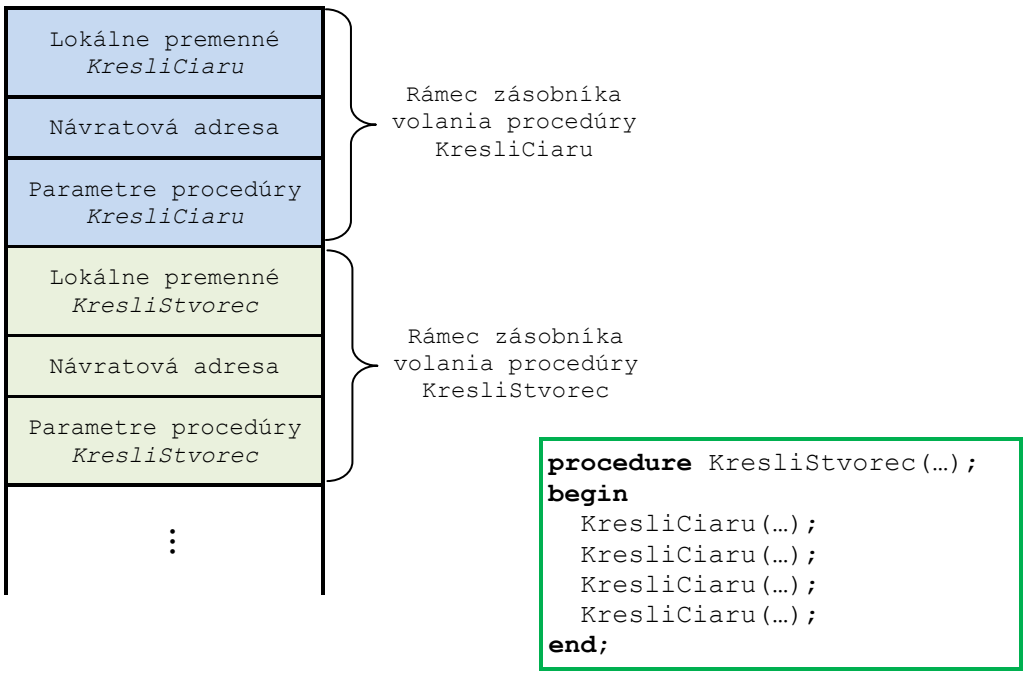
Ak chceme v Pascale zavolať podprogram definovaný neskôr, stačí niekde pred miestom volania uviesť hlavičku podprogramu nasledovanú kľúčovým slovom **forward**.

parametre podprogramu naplnené podľa hodnôt, s ktorými sa podprogram volá.

Ak sa pozorne zamyslíme, ako to všetko vyzerá v zásobníku volaní, prídeme na to, že v skutočnosti nepotrebujeme mať výpočtový zásobník v každom kontexte – vystačíme si s jedným zásobníkom pre celé vykonávanie podprogramu. To vďaka tomu, že prvky sa do zásobníka pridávajú, resp. odoberajú len na jednom konci. Pre prípad, že by podprogram po skončení nechal v zásobníku nejaké prvky navyše alebo by pri svojej činnosti odobral nejaké prvky, ktoré tam nevložil, stačí do kontextu pridať informáciu o aktuálnom počte prvkov v zásobníku z pohľadu podprogramu. Pri práci so zásobníkom potom treba už len kontrolovať, či tento počet neklesne pod 0, resp. po skončení vykonávania podprogramu odobrať zo zásobníka prvky, ktoré tam „po ňom ostali“. Dokonca aj premenné by sme za istých podmienok nemuseli vytvárať ako samostatné prvky kontextu, ale mohli by sme ich uchovávať vo výpočtom zásobníku.

Ešte sme si nespomenuli, ako podprogram môže vrátiť hodnotu. V našej implementácii interpretera jazyka DSL na tento účel používame globálnu premennú `NaposledyVratenaHodnota`. Ďalšou možnosťou je túto hodnotu uložiť na vyhradené miesto v kontexte volajúceho podprogramu.

Mechanizmus zásobníka volaní sa nevyužíva len pri interpretovaní programov, ale aj pri behu skompilovaných programov. Z pohľadu programátora je jeho prítomnosť a činnosť neviditeľná – transparentná. Pri vytváraní interpretera jazyka DSL sú lokálne premenné uložené v kontextoch vykonávaní a tým pádom „de facto“ v zásobníku volaní. Podobná situácia je pri behu skompilovaných programov, kedy sa lokálne premenné uchovávajú v zásobníku volaní.



Obrázok 2: Ukážka obsahu zásobníka volaní pri behu skompilovaného programu.

Činnosť zásobníka volaní v pascalovských programoch môžeme najlepšie vidieť pri krokovaní programu v okne ladenia „Zásobník volaní“. Aby sme mohli krokovať bežiaci program v prostredí Lazarus, musíme najprv bežiaci program zastaviť počas behu. Na zastavenie programu na určitom mieste slúži tzv. miesto prerušenia - ang. breakpoint. Breakpoint-y sa v prostredí Lazarus (aj Delphi) vytvárajú kliknutím naľavo od riadka (do lišty s číslovaním riadkov), na ktorom sa má vytvoriť. Riadky s breakpointami sú podfarbené červenou farbou. V programe môžeme mať viacero breakpoint-ov. Ak chceme breakpoint z riadka odstrániť, stačí opäť kliknúť naľavo od tohto riadka.

Volanie funkcie
`x := funkcia(a, b);`
po preklade do jazyka asemblera vyzerá nejako takto:

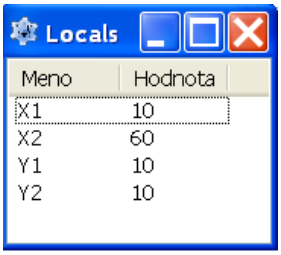
```
push b
push a
call funkcia
add esp, 8
mov x, eax
```

Všimnime si, že pred samotným volaním sa do zásobníka ukladajú hodnoty parametrov. Kým my sme pri implementácii interpretera uvažovali samostatný zásobník pre každý kontext, v skompilovaných programoch sa využíva len jeden zásobník - v ňom sú informácie o volaní, návratová adresa (kde má pokračovať program po návrate z podprogramu), lokálne premenné. Po návrate z podprogramu sa register ESP obsahujúci adresu vrchu zásobníka upraví tak, že sa pôvodne vložené hodnoty `a` a `b` z neho odstránia. Vrátená hodnota je uložená v registri EAX. Poznamenajme, že v niektorých architektúrach zásobník rastie od menších adries k vyšším, v iných zase od vyšších adries k nižším. Keďže údaje (premenné) a návratové adresy sú v tom istom zásobníku, pri neopatrnom programovaní môže mať náš program zraniteľnosť „stack buffer overflow“ (http://en.wikipedia.org/wiki/Stack_buffer_overflow). To je nebezpečné najmä pri rôznych internetových aplikáciách prijímajúcich údaje po sieti od klientov.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  KresliStvorec(10, 10, 50);
50 end;
```

Obrázok 3: Riadok s aktívnym miestom prerušenia (breakpointom)

Okno ladenia: Lokálne premenné



Breakpoint spôsobuje, že počas behu programu sa vždy pred vykonaním príkazu na riadku s aktívnym breakpoint-om zastaví beh program. Riadok, pred ktorého vykonaním sa zastavil beh programu, je označený v lište naľavo od zdrojového kódu zelenou šípkou. Po pozastavení programu vieme o aktuálnom stave behu programu získať mnoho zaujímavých informácií. Ak v zdrojovom kóde podržíme kurzor myši nad premennou, ktorá existuje v práve aktívnom kontexte vykonávania, zobrazí sa nám jej aktuálna hodnota. Pre zobrazenie všetkých lokálnych premenných v aktívnom kontexte vykonávania môžeme použiť okno „Lokálne premenné“, ktoré otvoríme cez ponuku menu **Zobraziť > Okná ladenia > Lokálne premenné** (Ctrl+Alt+L). Cez podmenu **Zobraziť > Okná ladenia** je možné zobraziť viacero zaujímavých okien s údajmi o aktuálnom stave behu programu:

- **Zásobník volaní** – aktuálny obsah zásobníka volaní, pri každom neukončenom volaní v zásobníku je zobrazený názov podprogramu spolu s hodnotami parametrov volania,
- **Assembler** – príkaz, ktorý sa má najbližšie vykonať, zapísaný ako postupnosť inštrukcií v jazyku asemblera vrátane prekladu týchto inštrukcií do strojového kódu v binárnom tvare,
- **Registers** – aktuálny obsah registrov procesora,
- **Pozorovania** – umožňuje napísať výraz, ktorý sa vyhodnotí podľa obsahu premenných v aktuálnom kontexte vykonávania.

Prostredie Lazarus tak, ako všetky vývojárske nástroje podporujúce ladenie programov (slangovo označované ako debugovanie programov), umožňuje krokovanie programu, t.j. postupné vykonávanie príkazov po jednom. Krokovanie je možné ovládať cez podmenu **Spustiť.** Praktickejšie je však krokovanie ovládať klávesovými skratkami:

- **Krok dnu (F7)** – vykoná príkaz, ktorý sa má vykonať ako ďalší v poradí; ak tento príkaz obsahuje volanie podprogramu (procedúry alebo funkcie), krokovanie sa **presunie do** zdrojového kódu podprogramu,
- **Krok cez (F8)** – vykoná príkaz, ktorý sa má vykonať ako ďalší v poradí; ak tento príkaz obsahuje volanie podprogramu, podprogram sa **vykoná bez** krokovania jeho príkazov,
- **Spustiť (F9)** – spustí beh programu bez krokovania, beh však môže prerušiť ďalší breakpoint, na ktorý sa pri behu programu príde.

Úloha 1.13

V prostredí Lazarus otvorte a krokujte pripravený testovací program z prostredia Moodle. Všímajte si, ako sa mení obsah okna „Zásobník volaní“.

1.4 Preklad a vyhodnotenie aritmetických výrazov

Jazyk DSL tak, ako sme ho doposiaľ definovali, je dosť nepraktický. Napríklad obyčajné vynásobenie premennej `x` konštantou `10` vyžaduje 4 príkazy jazyka DSL:

```
PUSH(x)
PUSH(10)
STACKMUL
x := POP
```

Oveľa jednoduchšie by bolo napísať `x := x * 10` ako v jazyku Pascal. Jazyk DSL však nepodporuje aritmetické výrazy. Doprogramovať do interpretera jazyka DSL

Rekuzia

Možno vám pri uvažovaní o tom, ako funguje mechanizmus volania podprogramov, napadla otázka: Podprogram **A** môže volať podprogram **B**. Môže ale podprogram **A** niekde vo svojom vykonávaní opäť volať podprogram **A** (sám seba)? Odpoveď je áno. Treba si uvedomiť, že v tomto prípade volanie podprogramu **A** vytvára nový kontext – tento aktuálny kontext s kontextom v zásobníku volaní nemajú takmer nič spoločné. Spoločný je iba zdrojový kód, ktorý sa ale vykonáva podľa obsahu premenných v kontexte. A ten je v týchto dvoch vykonávaniach podprogramov rôzny.

V jazyku DSL môžeme vytvoriť takýto rekurzívny podprogram počítajúci faktoriál čísla na základe vzťahu rekurentnej matematickej definície

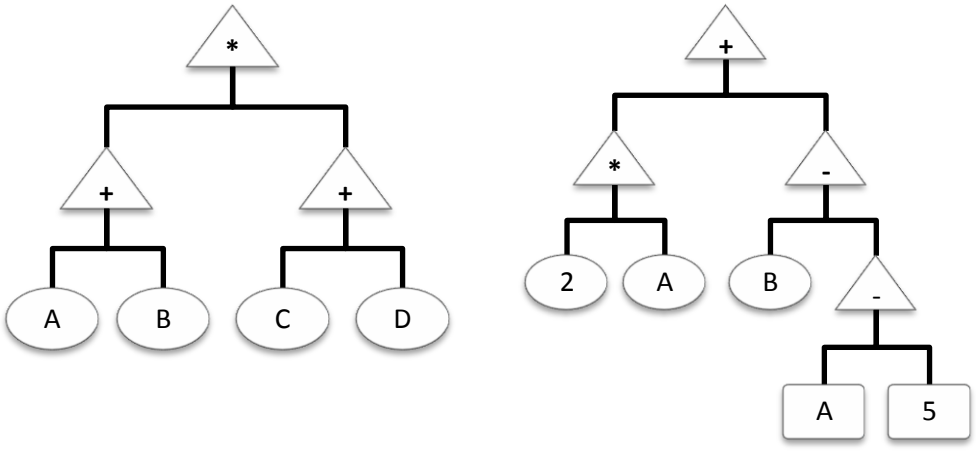
n! = n.(n-1)!

```
SUBROUTINE Fakt(n)
  result := 1
  IFNOT n THEN EXIT

  PUSH(n)
  n := PRED(n)
  result := Fakt(n)
  PUSH(result)
  STACKMUL

  result := POP
END
```

rozparsovanie a vyhodnocovanie jednoduchých aritmetických výrazov (napr. vynásobenie premennej konštantou, pripočítanie hodnoty, ...) je síce prácna, ale principiálne o nič náročnejšia úloha než to, s čím sme sa doposiaľ stretli. Riešenie tohto problému by viedlo k pridaniu veľkého množstva testov a if-ov do zdrojového kódu, v ktorých by sa analyzovali a vyhodnocovali špeciálne formy podporovaných aritmetických výrazov. Ako by sa však naprogramovalo vyhodnotenie ľubovoľného aritmetického výrazu? Poznamenajme, že v tomto prípade už môžeme mať dočinenia s výrazmi, kde sú pokope rôzne aritmetické binárne i unárne operácie s rôznymi prioritami, zátvorky či volania funkcií. Pri implementácií interpreterov existuje viacero možností ako riešiť vyhodnocovanie aritmetických výrazov. Intuitívne je jasné, že celý aritmetický výraz sa musí nejako rozložiť do postupnosti vyhodnotenia jednoduchších podvýrazov tak, ako keď „ručne“ vyhodnocujeme nejaký zložitejší aritmetický výraz. Výsledkom vyhodnotenia podvýrazov sú akési medzivýsledky, ktoré potom použijeme pri ďalšom výpočte. Zvyčajne prvý nápad je uložiť ich do pomocných premenných. Príkaz `V := (A+B)*(C+D)` môžeme rozložiť do postupnosti jednoduchších podvýrazov:

```
POM1 := A+B
POM2 := C+D
V := POM1 * POM2
```

Po skúsenosti s úlohou 1.4 v kapitole 1.2 vieme, že na vyhodnotenie tohto výrazu nám vôbec netreba pomocné premenné a schopnosť interpretovať jednoduché aritmetické výrazy. Vystačíme si so zásobníkom s aritmetickými operáciami. Otázka je, či to je tak len v tomto prípade alebo naozaj dokážeme vyhodnotenie každého aritmetického výrazu zrealizovať s využitím takéhoto zásobníka. Ukážeme, že áno. Naozaj vyhodnotenie každého aritmetického výrazu je možné zapísať ako postupnosť takých `PUSH` a `STACK`XYZ príkazov, že po skončení vykonávania tejto postupnosti príkazov sa v zásobníku vypočíta hodnota aritmetického výrazu na základe aktuálnych hodnôt premenných (kontextu). Dokonca táto postupnosť príkazov bude mať tú vlastnosť, že po skončení výpočtu bude obsah zásobníka úplne rovnaký ako pred jeho začiatkom s tou výnimkou, že na vrchu zásobníka pribudne nový prvok s hodnotou výsledku vyhodnotenia. Existuje viacero algoritmov, pomocou ktorých vieme túto postupnosť príkazov pre zadaný aritmetický výraz automaticky skonštruovať. V ďalšom si na príklade ukážeme princíp jedného z nich. Poznamenajme, že ak by sme popri vytváraní postupnosti príkazov tieto príkazy aj ihneď vykonávali, dostali by sme algoritmus na vyhodnotenie aritmetického výrazu.



Obrázok 4: Aritmetické stromy

V module *Algoritmy a štruktúry údajov* sme si predstavili údajovú štruktúru strom. Táto štruktúra sa vtedy (okrem iného) ukázala ako veľmi dobrý prostriedok na uloženie hierarchických údajov (napr. rodokmeňov). Ďalším využitím stromov sú tzv. aritmetické stromy, ktoré umožňujú zachytiť (uložiť) aritmetické výrazy vo forme, ktorá je výhodná na efektívnu analýzu a spracovanie výrazu. **Aritmetický strom** je

binárny strom (každý uzol má nanajvýš dve deti). Aritmetické operácie sú v aritmetickom strome uložené vo vnútorných uzloch stromu, konštantné hodnoty a premenné (presnejšie odkazy na premenné, resp. mená premenných) v listoch stromu. Na obrázku 4 môžeme vidieť príklady aritmetických stromov pre výrazy `(A+B)*(C+D)` a `2*A+B-(A-5)`. Kvôli prehľadnosti uzly aritmetických operácií a uzly hodnôt/premenných vizuálne rozlišujeme.

Ako môžeme vidieť, v koreni stromu sa nachádza operátor, ktorý by sme pri vyhodnotení výrazu aplikovali ako posledný (operátor s najnižšou prioritou). Tento operátor nám rozdelí výraz na dva podvýrazy, ktorým zodpovedajú ľavý a pravý (aritmetický) podstrom koreňa. Rovnaká situácia je v každom jednom podstrome, resp. jemu zodpovedajúcom podvýraze. Na systematické skonštruovanie aritmetického stromu *T* pre zadaný aritmetický výraz *V* môžeme použiť nasledujúci algoritmus:

1. Ak je celý výraz uzatvorený v zátvorkách, potom odstráň vonkajšie zátvorky, t.j. napríklad výraz `(1+3*(a-5))` zmeň na `1+3*(a-5)`.
2. Ak je výraz číselná hodnota, potom strom výrazu je uzol s tohto hodnotou.
3. Ak výraz pozostáva len z mena premennej, potom strom výrazu je uzol s menom tejto premennej.
4. Nájdi vo výraze *V* operátor s najnižšou prioritou, ktorý nie je uzavretý v zátvorkách:
    a. Rozdeľ výraz *V* na podvýraz V_1 naľavo od neho a podvýraz V_2 napravo od neho.
    b. Aplikuj celý tento algoritmus na skonštruovanie aritmetického stromu T_1 pre podvýraz V_1.
    c. Aplikuj celý tento algoritmus na skonštruovanie aritmetického stromu T_2 pre podvýraz V_2.
    d. Vytvor strom T tak, že koreňom je uzol obsahujúci rozdeľujúci operátor, ktorého ľavým podstromom je strom T_1 a pravým podstromom je strom T_2.

Tento algoritmus na systematickú konštrukciou aritmetického stromu je príkladom rekurzívneho algoritmu, keďže na zrealizovanie krokov 4.b a 4.c tohto algoritmu aplikujeme tento algoritmus na podvýrazy (algoritmus vo svojom opise využíva samého seba).

Úloha 1.14

Spoločne aplikovaním algoritmu skonštruujte aritmetický strom pre výraz `(A+3*B)*5+4*A`. V pracovných skupinách samostatne aplikujte algoritmus na výraz, ktorý vám určí iná pracovná skupina.

Po vytvorení stromu aritmetického výrazu aplikovaním nasledujúceho algoritmu skonštruujeme postupnosť príkazov realizujúcich vyhodnotenie výrazu s použitím zásobníka s aritmetickými operáciami:

1. Ak koreň obsahuje meno premennej p, postupnosť príkazov realizujúcich výpočet je: `PUSH(p)`
2. Ak koreň obsahuje číselnú hodnotu h, postupnosť príkazov realizujúcich výpočet je: `PUSH(h)`
3. Ak koreň obsahuje aritmetickú operáciu O: MUL(*), ADD(+), DIV(/) alebo SUB(-), potom:
    a. Vytvor postupnosť príkazov P_1 realizujúcich vyhodnotenie ľavého podstromu koreňa.
    b. Vytvor postupnosť príkazov P_2 realizujúcich vyhodnotenie pravého podstromu koreňa.
    c. Postupnosť príkazov realizujúcich vyhodnotenie výrazu je:
    príkazy P_1
    príkazy P_2
    `STACK`O (t.j. `STACKMUL`, `STACKADD`, `STACKDIV` alebo `STACKSUB`)

Úloha 1.15

Podľa vyššie uvedeného postupu pre zadaný aritmetický strom vytvorte postupnosť operácií, ktoré vedú k jeho vyhodnoteniu.

1.5 Štruktúrované programovacie jazyky

Ak by sme v jazyku DSL skúsili napísať väčší program, výsledný program by bol s veľkou pravdepodobnosťou neprehľadný. Asi najväčším zdrojom neprehľadnosti by bol príkaz skoku - GOTO. Pomocou tohto príkazu totiž môžeme skočiť na ľubovoľné miesto v podprograme, čím sa stráca nadhľad nad celým programom. Podobný problém sa objavil v 60-tych rokov, kedy sa vo vývoji softvéru prejavila tzv. softvérová kríza. Jednou z odpovedí na neprehľadnosť zdrojového kódu vtedajších programovacích jazykov využívajúcich varianty GOTO inštrukcií bol vznik a nárast popularity tzv. štruktúrovaných programovacích jazykov (medzi ne patrí Pascal aj C). Základnom myšlienkou štruktúrovaných programovacích jazykov je rozdelenie programového kódu a jeho vykonávania do blokov, ktoré sú jeho stavebnými jednotkami. Jednotlivé bloky vytvárajú hierarchickú štruktúru toku riadenia programu. Napr. v Pascale sa tieto bloky vytvárajú pomocou kľúčových slov `begin` a `end`. Na štruktúrovaný program sa môžeme pozerať aj ako na akúsi skladačku. Stavebné štruktúry príkazov sú troch typov: postupnosť (`begin-end` blok), opakovanie (`while-do`, `repeat-until`, `for`-cyklus) a výber (`if-then-else`, `case-of`). Vezmime si príkaz `while`-cyklu v jazyku Pascal. Jeho štruktúra je:

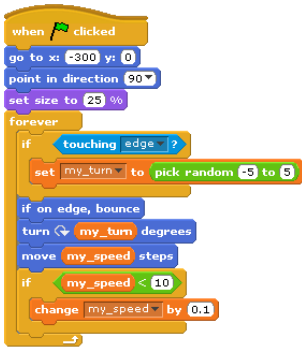
```
while logický výraz do príkaz;
```

Sémantika (význam) tohto príkazu je taká, že `príkaz` sa má opakovať tak dlho, kým platí `logický výraz`. Za časť `príkaz` môžeme dosadiť buď konkrétny jeden príkaz alebo blok príkazov uzavretých medzi kľúčové slová `begin` a `end`. V tomto bloku príkazov môžu byť ľubovoľné príkazy - napr. aj príkaz `while`. To, čo je dôležité všimnúť si na tejto štruktúre, je skutočnosť, že bez GOTO neexistuje žiaden spôsob, ako by sa riadenie programu mohlo dostať z bloku `príkaz` na inštrukciu mimo tento blok iným spôsobom, než je ukončenie cyklu (splnením podmienky alebo príkazom break), a zároveň nie je možnosť, aby sa riadenie dostalo na niektorý príkaz v bloku `príkaz` bez toho, aby sa začal vykonávať `while`-cyklus a s ním aj celý blok `príkaz`. Schematicky možno syntax takejto štruktúry zapísať formulkou:

```
<statement> ::= "while" <expression> "do" <statement>
```

Podobným spôsobom možno popísať celú syntax programovacieho jazyka. Tento spôsob zápisu sa nazýva **Backus-Naurova forma** (skrátene BNF).

Reálne programátor nikdy nerozmýšľa nad tým, že štruktúrovaný program tvorí pomocou takýchto pravidiel. Skladanie programu „zo stavebných dielikov“ pripomínajúcich Puzzle je perfektne spracované v detskom programovacom jazyku Scratch (http://scratch.mit.edu/).



Úloha 1.16

Na stránke http://www.fh-jena.de/contrib/fb/gw/gmueller/Kurs_halle/pas_bnf.html si pozrite pravidlá pre syntax jazyka Pascal zapísané v BNF notácii (terminálne a neterminálne symboly nie sú rozlišované). Diskutujte o význame jednotlivých pravidiel. Ide týmito pravidlami zachytiť programami, ktoré ste v minulosti vytvárali?

Na tvorenie štruktúrovaného programu sa môžeme pozerať ako na skladanie Puzzle alebo akúsi prepisovaciu hru. V každom okamihu je program postupnosť slov dvoch typov: terminálov (napr. `"while"`), ktoré nesmieme už ďalej prepisovať, a neterminálov (napr. `<expression>`), ktoré sa prepisujú podľa syntaktických pravidiel. Význam vyššie uvedenej formulky syntaktického pravidla je ten, že kedykoľvek sa vo vytváranom programe objaví neterminál `<statement>` (miesto, kde sa ako programový podblok očakáva nejaký príkaz) aplikovaním pravidla môžeme neterminál `<statement>` prepísať na `"while" <expression> "do"`

<statement>. Z pravidiel pre Pascal ľahko vydedukujeme, že terminálne symboly sú kľúčové slová jazyka a identifikátory. Neterminály sú naopak „rozrobené“ časti programu, ktoré ešte treba doprogramovať. Ďalšou dôležitou vlastnosťou neterminálov je to, že pre jeden neterminál je zvyčajne definovaných viacero pravidiel - záleží len na programátorovi, aké „prepisovacie“ pravidlo zvolí, aby výsledný program robil to, čo má. Program je hotový, keď už niet čo prepisovať, t.j. program sa skladá len z terminálov.

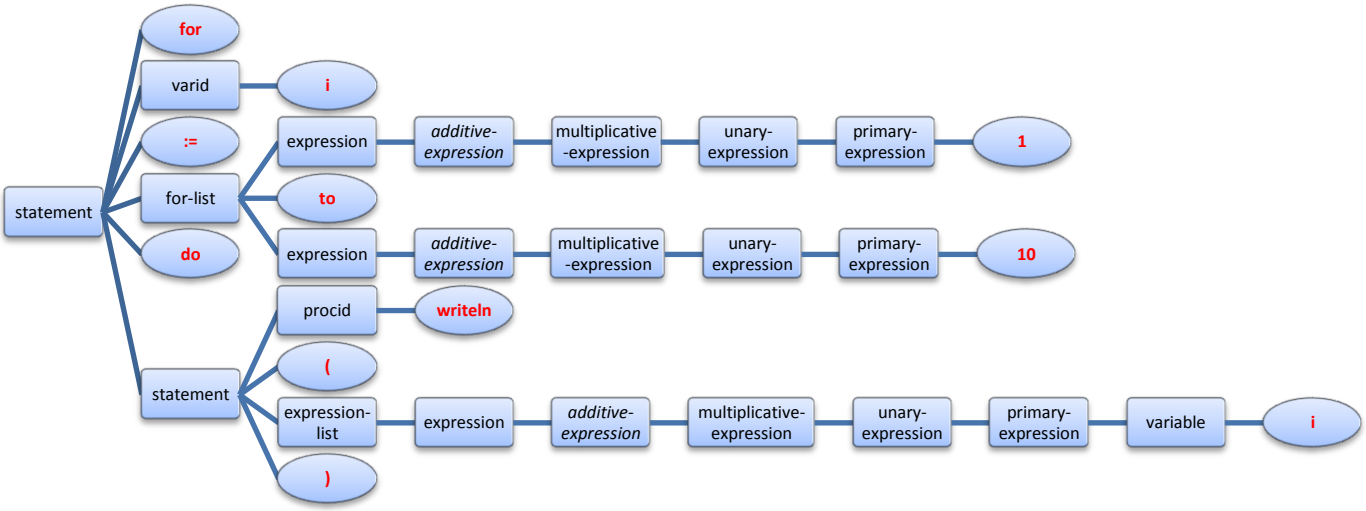
Úloha 1.17

Uvažujme jednoduchý fragment programu:

```
for i:= 1 to 10 do writeln(i);
```

Aplikovaním akých syntaktických pravidiel jazyka Pascal dokážeme vytvoriť tento kód z neterminálu <statement>?

Pripomeňme, že programy v štruktúrovanom programovacou jazyku majú hierarchické usporiadanie. Ak si spomenieme na modul *Algoritmy a štruktúry údajov*, vždy keď máme nejaké hierarchické usporiadanie, tak sa určite odniekiaľ vynoria stromy. A nie je tomu inak ani tu. Podobne ako aritmetický výraz vieme reprezentovať aritmetickým stromom, aj zdrojový kód štruktúrovaného programovacieho jazyka vytvára strom. Tento strom sa nazýva **AST - abstraktný syntaktický strom**.



Obrázok 5: Abstraktný syntaktický strom pre
`for i:= 1 to 10 do writeln(i);`

Nie je náhoda, že AST je tak trochu podobný aritmetickým stromom. Veď aritmetický výraz je z istého uhla pohľadu „programom“ na výpočet nejakej numerickej hodnoty.

Ako môžeme vidieť na obrázku, neterminály sú v abstraktnom syntaktickom strome umiestnené vo vnútorných uzloch a terminály v listoch. „Rozvetvenie“ neterminálu predstavuje aplikáciu niektorého z „prepisovacích“ syntaktických pravidiel programovacieho jazyka. Napríklad rozvetvenie koreňa vzniklo aplikovaním pravidla:

<statement> ::= "for" <varid> ":=" <for-list> "do" <statement>

Úloha 1.18

Podľa syntaktických pravidiel v jazyka Pascal v BNF notácií vytvorte AST pre jednoduchý fragment programu (neterminál <expression> v AST nerozvíjajte).

Vytvorenie prvých kompilátorov bola veľmi náročná činnosť. Pre jazyk zdrojového kódu sa všetky časti kompilátora programovali „na mieru“. Našťastie výsledky v oblasti kompilátorov tento proces do značnej miery zautomatizovali. Základom tejto automatizácie je možnosť zapísať syntax programovacieho jazyka pomocou syntaktických pravidiel (napríklad v BNF). S využitím týchto pravidiel potom

dokážeme pre zadaný vstupný zdrojový kód algoritmicky skonštruovať jemu zodpovedajúci AST. Hlavnou výhodou AST je zachytenie štruktúry programu. Vďaka tomu je možná efektívna analýza zdrojového kódu a jeho následný preklad do iného počítačového jazyka. Spracovanie zdrojového kódu prebieha v niekoľkých fázach:

- **skenovanie (lexikálny analýzátor/skener)** – realizuje sa lexikálna analýza, kedy sa zdrojový kód rozdelí na postupnosť tokenov, ktoré zodpovedajú terminálnym symbolom,
- **parsovanie (syntaktický analyzátor/parser)** – realizuje sa syntaktická analýza postupnosti tokenov, pri ktorej sa identifikuje syntaktická štruktúra programu a zvyčajne sa buduje AST,
- **sémantická analýza** – kontroluje sa kompatibilita typov, vytvára sa tabuľka symbolov, analyzuje sa korektnosť programu,
- **optimalizácia** – aplikujú sa transformácie, ktoré zachovávajú funkčnosť programu, no vedú k tomu, aby preložený (cieľový) kód bol efektívnejší,
- **generovanie kódu** – samotné generovanie cieľového kódu.

Spomínate si na pravidlo, že názov premennej (identifikátor) v Pascale nesmie začínať číslom? Toto pravidlo nie je len tak. Vďaka nemu dokáže skener po prečítaní prvého znaku tokenu („slova“) rozlíšiť, či ide o názov identifikátora alebo o číselný literál (konkrétnu numerickú hodnotu). Skenovanie je tak oveľa rýchlejšie.

Čo sme sa naučili

Poznáme pojem interpreter a kompilátor, rozumieme základným princípom ich činnosti. Dozvedeli sme sa o základných údajových štruktúrach, ktoré vieme využiť pri implementovaní jednoduchého interpretera. Vieme, že zásobník volaní je dôležitá údajová štruktúra používaná tak v interpreteroch ako aj v skompilovaných programoch na realizáciu volania podprogramov.

1.6 Literatúra

Základné materiály:

[1] Wirth, N. (1988) *Algoritmy a Dátové štruktúry.* Bratislava: Alfa 1988.
[2] Blaho, A. (2006) *Informatika pre stredné školy, Programovanie v Delphi*, SPN, Bratislava
[3] Blaho, A., Salanci, Ľ. (2009) Programovanie 1 až 9, študijné materiály DVUI, ŠPÚ, Bratislava
[4] Galčík, F., Winczer, M. (2010) Algoritmy a štruktúry údajov 1 a 2, študijné materiály DVUI, ŠPÚ, Bratislava

Internetové zdroje:

[5] http://en.wikipedia.org/wiki/Compiler
[6] http://en.wikipedia.org/wiki/Interpreter_(computing)

2. Kódovanie

S priateľom sme sa dohodli, že spolu pôjdeme trebárs na koncert. Zrazu nám však do toho niečo prišlo, a my mu to potrebujete dať vedieť. Máme, pravdaže, viacero možností — zavoláme mu, napíšeme SMS, e-mail, možno i list, alebo za ním prosto zájdeme. Bez ohľadu na to, ktorú z nich si vyberieme, proces prenosu tejto informácie má takéto fázy:

1. Informáciu, ktorú máme na začiatku v hlave len vo forme myšlienky, musíme najprv sformulovať do viet, slov, hlások. Tento proces sa obvykle deje viac-menej automaticky, jeho prítomnosť si však môžeme dobre uvedomiť vtedy, keď náš priateľ nevie po slovensky.

2. Ak chceme takto verbalizovanú informáciu napísať do listu, musíme najprv hlásky nahradiť príslušnými písmenami doplnenými o interpunkciu či iné grafické značky, teda do takej formy, aby ju papier dokázal preniesť. Takýto „preklad“ sa však deje aj v ďalších prípadoch: Pri osobnom kontakte sa hlásky, existujúce zatiaľ len v našich predstavách, pomocou hlasiviek menia na akustické vlny, pri telefonáte sa tieto vlny navyše pomocou telefónneho aparátu transformujú do prúdu elektrických signálov. V prípade SMS alebo e-mailu síce ide tiež o elektrické signály, tam sa však na ne premení postupnosť stláčaných klávesov. Vo všetkých týchto prípadoch teda ide o zakódovanie informácie do formy, ktorá je prenositeľná nami vybraným médiom.

„Papier znesie všetko.“
(anonym)

3. V tejto fáze nastáva samotný prenos zakódovaného signálu. V prípade telefonátu či SMS sa tak deje prostredníctvom telefónnej siete, pri e-maile využívame internet, pri obyčajnom liste služby pošty. Aj obyčajná priama osobná komunikácia však potrebuje médium, ide predsa o vlnenie vzduchu.

4. Keď sa už (akýmkoľvek spôsobom) zakódovaná správa dostane k nášmu priateľovi, ten ju potrebuje rozkódovať, čiže vrátiť do pôvodnej podoby. Prichádzajúca SMS sa zmení na prúd písmen, elektrické signály sa pomocou telefónneho prístroja znovu stanú akustickými vlnami a jedny i druhé sa prostredníctvom očí či uší zmenia na hlásky, slová a vety.

5. Posledná fáza sa deje v mozgu nášho priateľa, ktorý sa pokúša vzniknuté zrakové či sluchové vnemy interpretovať — zmeniť ich (v ideálnom prípade) späť na naše pôvodné myšlienky.

Na to, aby sa náš priateľ náš odkaz naozaj dozvedel, nesmie žiadna z týchto fáz zlyhať.

Úloha 2.1	Popíšte, čo znamená zlyhanie v každej z uvedených fáz pri každom z uvedených spôsobov komunikácie.
Ukážka riešenia	Rozoberme si prípad klasického listu: 1. Neschopnosť formulovať svoje myšlienky do viet je prejavom nedostatočného ovládania jazyka — obvykle cudzieho, ale v prípade veľmi zložitých myšlienok (hoci zrušenie koncertu nie je práve ten prípad) i vlastného. 2. Ak človek ovláda jazyk len slovom, ale nie písmom, zlyháva v druhej etape. Neschopnosť zapísať už sformulované myšlienky do písomnej podoby je spôsobená buď tým, že odosielateľ neovláda pravopis, alebo tým, že vôbec písať nevie (to by si však tento spôsob komunikácie zrejme nezvolil).

„O čom nemožno hovoriť,
o tom sa musí mlčať.“
(Ludwig Wittgenstein)

3. Tu nesie vinu pošta. Buď list nedoručí vôbec, alebo ho dopraví poškodený natoľko, že jeho obsah je dešifrovateľný iba čiastočne, ba dokonca vôbec.
4. Zlyhanie v tejto fáze znamená neschopnosť prečítať napísaný text, čiže premeniť písmená na (či už nahlas vyslovené, alebo len myslené) hlásky, a poskladať ich do slov a viet. Vina však nemusí byť len v prijímateľovi, možno len nie je schopný rozlúštiť nečitateľný odosielateľov škrabopis.
5. Podobne ako pri prvej fáze, aj tu je neschopnosť interpretovať spôsobená obvykle nedostatočným ovládaním jazyka, môže to však byť i chabým zmyslom pre poetiku.

Do ktorej fázy by ste ako zlyhanie zaradili dysgrafiu a dyslexiu?

„Co tím chtěl básník říci?“
(anonym)

Ľahko vidieť, že podobný proces sa deje pri ľubovoľnej inej informácii, a to i v prípade, že jej odosielateľom či príjemcom nie je človek.

Úloha 2.2	Aj kopírovanie súboru je prenos informácie. Akej? Kto alebo čo je v tomto prípade jej odosielateľom a kto alebo čo jej príjemcom? Ako v tomto prípade vyzerajú horeuvedené fázy?
Aktivita 2.1	Nájdite ďalšie príklady prenosu informácie. Rozdeľte každý z nich na príslušné fázy. Ako v týchto prípadoch vyzerajú zlyhania?

Všimnime si, že prvá a posledná fáza sú skôr psychologického rázu (takže v prípade, že na koncoch reťazca nie sú ľudia, ich môžeme dokonca zanedbať), nebudeme sa nimi preto ďalej zaoberať. Sústredíme sa však na zvyšné tri. Keďže zakódovanie a rozkódovanie (už aj intuitívne) úzko súvisia, budeme sa im venovať naraz v prvej časti, kým druhá časť bude zameraná na samotný prenos zakódovanej informácie.

2.1. Zakódovanie a rozkódovanie

2.1.1. Bezprefixové kódovanie

Pri zakóduvaní vlastne nahradzujeme nejaké objekty z istej množiny symbolov S objektmi tiež z nejakej množiny C. Kým na charaktere prvkov množiny S veľmi nezáleží, prvky množiny C musia byť také, aby ich bolo možné prenášať. Asi najprirodzenejšie je, že to budú akési postupnosti čo najjednoduchších signálov. Aby sme s touto množinou vedeli rozumne pracovať, bolo by vhodné, aby tieto elementárne signály boli čo najľahšie identifikovateľné, inými slovami, aby boli z nejakej dopredu definovanej a pomerne malej množiny. Túto množinu môžeme oprávnene nazvať jazyk a jej prvky znaky. Množinu C potom môžeme chápať ako množinu niektorých reťazcov tohto jazyka, t. j. konečných postupností jeho znakov. Ak teda jazyk označíme napríklad L, tak C je podmnožinou množiny L^* všetkých slov jazyka L. Jej prvky preto nazveme kódové slová. Kódovanie potom bude vlastne isté zobrazenie z množiny S do množiny C. Zakódovaný symbol bude treba po prenose rozkódovať, teda prijatým slovám treba spätne priradiť symboly. Za (pomerne silného) predpokladu, že pri prenose sa kódové slová nepoškodili, je teda rozkódovanie tiež zobrazenie, a to z množiny C do množiny S. Bolo by, samozrejme, veľmi vhodné, aby sme zakódovaný symbol rozkódovaním naozaj odhalili, čiže ak kódovanie označíme e a rozkódovanie d, musí pre každý symbol s platiť $d(e(s)) = s$. Inými slovami, zobrazenie d musí byť k zobrazeniu e inverzné. To znamená, že rozkódovanie je kódovaním úplne určené, a stačí sa preto sústrediť len na samotné kódovanie.

Ilustrujme si tieto pojmy na konkrétnom príklade: Množina S symbolov, ktoré budeme kódovať, nech je $\{\heartsuit, \diamondsuit, \clubsuit, \spadesuit\}$, elementárne signály nech sú 0 a 1, čiže $L = \{0, 1\}$. Kó-

dovanie e_1 nech je dané napríklad ľavou tabuľkou, rozkódovanie d_1 (ako jeho inverzné zobrazenie) potom pravou:

s	$e_1(s)$
♥	00
♦	01
♣	10
♠	11

c	$d_1(c)$
00	♥
01	♦
10	♣
11	♠

Správa, ktorú chceme preniesť, je však iba zriedkavo tvorená jediným symbolom. Vzhľadom na linearitu času ide obvykle o ich konečnú postupnosť. Každý z nich zakódujeme nejakým slovom, ktoré vzápätí prenesieme príjemcovi. Ten ich postupne rozkóduje a spojením týchto kódov do jedného celku našu správu zrekonštruuje. Napríklad pri kódovaní z nášho príkladu sa pri správe ♦♣♦♠ jej jednotlivé znaky postupne zakódujú do kódových slov 01, 10, 01 a 11, takže celkovo je správa zakódovaná do reťazca 01100111. Keďže všetky naše kódové slová majú dva znaky, túto zakódovanú správu môžeme jednoducho spätne rozdeliť na dvojice znakov — 01 + 10 + 01 + 11, tie postupne rozkódovať na symboly ♦ + ♣ + ♦ + ♠, a tak zrekonštruovať pôvodnú správu ♦♣♦♠.

Ako sme si už všimli, predchádzajúce kódovanie má istú špeciálnu vlastnosť — všetky kódové slová majú rovnakú dĺžku (a to 2). Takéto kódovanie nazveme blokové. Ako však ukazuje nasledujúci príklad, do úvahy však prichádzajú aj iné možnosti:

s	$e_2(s)$
♥	10
♦	0
♣	110
♠	111

V kódovaní e_2 sa správa ♦♣♦♠ zakóduje do reťazca 0 + 110 + 0 + 111, t. j. 01100111. Aj tu musí túto postupnosť jej príjemca najprv rozložiť na kódové slová, je to však v porovnaní s predchádzajúcim príkladom náročnejšie:

- Keďže zakódovaná správa sa začína znakom 0, ako prvý symbol správy do úvahy prichádza jedine ♦.
- Zvyšok zakódovanej správy je potom 1100111. Jeho prvý znak je 1, mohlo by teda ísť o niektorý zo symbolov ♥, ♣, alebo ♠. Avšak druhý znak je 1, symbol ♥ to teda nebude. Nebude to však ani ♠, lebo tretí znak je 0. Druhý symbol správy je teda ♣.
- Zvyšok zakódovanej správy je 0111. Keďže sa začína znakom 0, tretí symbol musí byť opäť ♦.
- Zvyšok zakódovanej správy je 111, a ten nekorešponduje so žiadnym symbolom okrem ♠.
- Tým sa však reťazec zakódovanej správy plne vyčerpal, správa je teda rozkódovaná.

Situácia sa však môže skomplikovať, ako si ukážeme na nasledujúcom príklade:

s	$e_3(s)$
♥	0
♦	1
♣	10
♠	11

Za tohto kódovania sa naša správa ♦♣♦♠ zakóduje do reťazca $1+10+1+11$, t. j. 110111. Pri rozkódovaní túto postupnosť treba rozdeliť na kódové slová (z ktorých potom rozkódovaním dostaneme symboly pôvodnej správy). Problém však je, že to môžeme urobiť viacerými spôsobmi — okrem vysielateľom správy zamýšľaného rozkladu $1+10+1+11$ vyhovuje napríklad aj $1+1+0+1+11$, čo vedie k správe ♦♦♥♦♠. Toto kódovanie teda nevyhovuje.

Úloha 2.3	Nájdite všetky správy, ktoré sa v kódovaní e_3 zakódujú do postupnosti 110111.

Kódovanie e_3 by sme, prirodzene, mohli upraviť tak, že by sme medzi jednotlivé kódované slová vložili nejaký oddeľovač, napríklad ␣, a teda správa by po takomto zakódovaní vyzerala $1␣10␣1␣11$, resp. systematickejšie $1␣10␣1␣11␣$. To by však znamenalo, že by sa jazyk kódových slov zväčšil z dvojprvkového $\{0,1\}$ na trojprvkový $\{0,1,␣\}$ a kódovanie by sa zmenilo takto:

s	$e_4(s)$
♥	$0␣$
♦	$1␣$
♣	$10␣$
♠	$11␣$

Tu už podobný zmätok nehrozí, ak však chceme ostať pri pôvodnom dvojprvkovom jazyku (napríklad z technických dôvodov), kódovanie e_4 je nevyhovujúce.

Vráťme sa však ku kódovaniu e_2. Keď sme hľadali jednoznačnosť rozkladu zakódovanej správy na kódové slová, postupne sme vylučovali symboly, ktoré s prečítanou časťou nekorešpondovali, a to až dovtedy, kým neostalo jediné prípustné slovo. V istom zmysle sme však mali šťastie, že sa kódové slová vylučovaných symbolov na istom mieste od prečítanej časti začali líšiť. Ak by sme totiž takto postupovali pri kódovaní e_3, už po prečítaní prvého znaku zakódovanej správy 110111 by sme nevedeli, či ide už o celé kódové slovo symbolu ♦, alebo ešte len o začiatok kódového slova symbolu ♣ alebo ♠.

Slovo α jazyka L nazývame prefixom slova β, ak existuje slovo γ jazyka L také, že $\beta = \alpha\gamma$.

Takže napríklad slovo 1, ktoré je v kódovaní e_3 kódovým slovom symbolu ♦, je prefixom slova 10, ktoré v tomto kódovaní kóduje ♣, pričom použijúc označenie z definície $\alpha = 1$, $\beta = 10$ a $\gamma = 0$. Podobne je slovo 1 prefixom slova 11, ktoré kóduje ♠. Naproti tomu však pri kódovaní e_2 žiadnu takúto dvojicu nájsť nevieme.

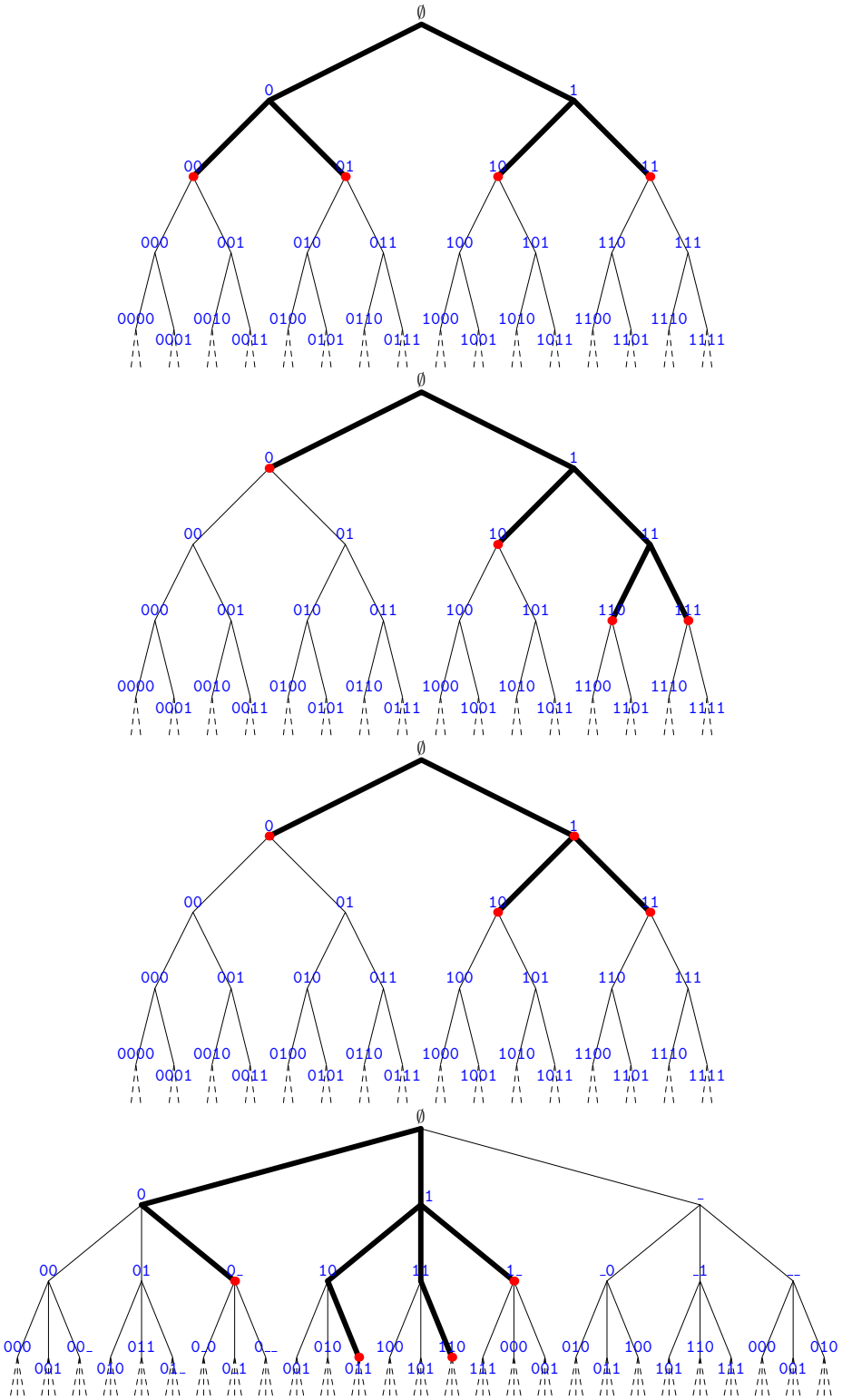
Množinu kódových slov nazývame bezprefixová, ak neexistuje dvojica jej rôznych prvkov c_1 a c_2, že c_1 je prefixom c_2.
Kódovanie nazveme bezprefixové, ak množina jeho kódových slov je bezprefixová.

Množina $\{10,0,110,111\}$ je teda bezprefixová, čiže aj kódovanie e_2 je bezprefixové. Kódovanie e_3 však bezprefixové nie je, lebo množina $\{0,1,10,11\}$ jeho kódových slov nie je bezprefixová.

Úloha 2.4	Je kódovanie e_1 bezprefixové? Pre ktoré ďalšie blokové kódovania platí toto tvrdenie?
Úloha 2.5	Je kódovanie e_4 bezprefixové?

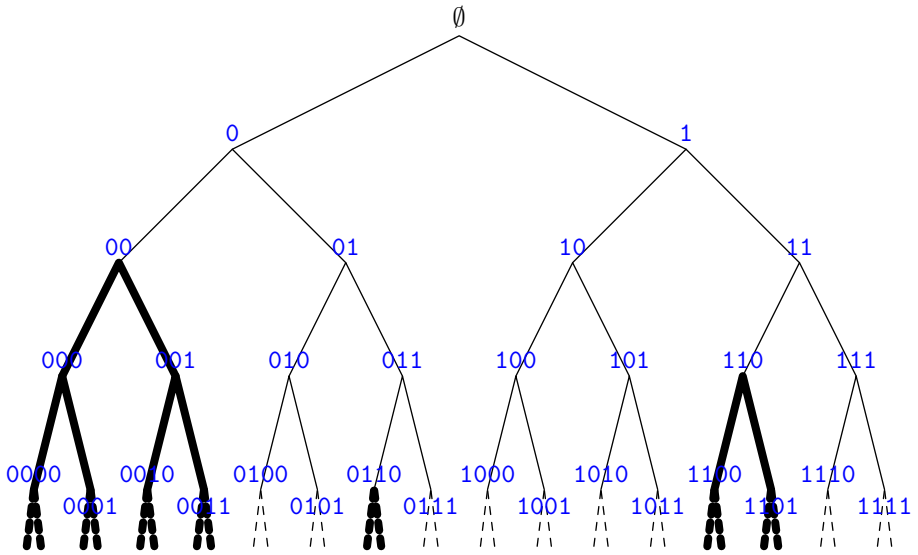
Každé kódovanie môžeme zobraziť aj obľúbeným informatickým spôsobom — vo forme stromu. Jeho uzlami budú všetky slová daného jazyka, pričom slová rovnakej dĺžky sú na tej istej úrovni (špeciálne jeho koreň zodpovedá prázdnemu slovu), a (orientovanou) hranou sú spojené také dva vrcholy, z ktorých druhý vznikne z prvého pridaním

práve jedného znaku. V prípade, že jazyk je n-prvkový, má teda každý uzol n potomkov, špeciálne pre náš jazyk $\{0, 1\}$ ide o binárny strom. Uzly, ktoré zodpovedajú kódovým slovám, vyznačíme farebne. Stromy (presnejšie, ich relevantné časti) našich kódovaní teda vyzerajú takto:



Z takéhoto obrázku môžeme ľahko vidieť, či je nejaké slovo prefixom iného. V takom prípade totiž uzol kratšieho slova leží na spojnici uzla dlhšieho slova s koreňom, alebo, inými slovami, uzly oboch slov ležia na spoločnej vetve. Ľahko vidieť, že v prípadoch stromov bezprefixových kódovaní e_1, e_2 a e_4 taký prípad nenastáva, kým v prípade e_3, ktoré nie je bezprefixové, áno.

Pri takomto grafickom vyjadrení si môžeme všimnúť istú dôležitú vec. Kvôli korektnosti a jednoduchosti sa pritom obmedzíme len na jazyk $\{0, 1\}$ a jeho slová dĺžky ohraničenej pevným, ale dostatočne veľkým číslom h (tak, aby bolo väčšie než dĺžka najdlhšieho kódového slova), takže všetky vetvy budú mať dĺžku h. Takýchto vetiev, čiže slov dĺžky h, je 2^h — na každom z h miest sa môže vyskytnúť jeden z dvojice znakov 0 alebo 1. Ku kódovému slovu s dĺžkou d si teraz všimnime počet vetiev, ktoré sú jeho prefixom — budeme hovoriť, že ich pokrýva. Každá z takých vetiev má prvých d znakov zhodných s naším slovom, a teda fixovaných, avšak každý zo zvyšných $h - d$ znakov môže byť ľubovoľný z dvojice 0 alebo 1. To však znamená, že pokrývaných vetiev je 2^{h-d}, a teda naše slovo s dĺžkou d pokrýva $\frac{2^{h-d}}{2^h}$, čiže $\frac{1}{2^d}$ z počtu všetkých vetiev.



Na týchto obrázkoch vidíme, že slovo s dĺžkou 2 pokrýva štvrtinu vetiev, slovo s dĺžkou 3 osminu, slovo s dĺžkou 4 šestnástinu a tak ďalej. Pomer pokrývaných vetiev teda klesá exponenciálne s dĺžkou slova.

Navyše, ak je naše kódovanie bezprefixové, každá vetva je pokrytá najviac jedným kódovým slovom, takže ak majú kódové slová dĺžky $d_1, \ldots, d_n$, musí platiť

$$\sum_{i=1}^{n} \frac{1}{2^{d_i}} \leq 1.$$

Úloha 2.6	Overte platnosť tohto vzťahu pre binárne a bezprefixové kódovania e_1 a e_2.

2.1.2. Cena kódovania

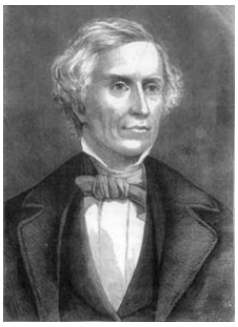
Všimnime si teraz jedno z najznámejších kódovaní — morseovku. Každému písmenu anglickej (t. j. latinskej) abecedy je priradený akýsi kód zložený z bodiek a čiarok.

Úloha 2.7	Koľko prvkov má jazyk morseovky?
Riešenie	Odpoveď, ktorá sa hneď ponúka — dva: bodku a čiarku —, je nepravdivá! Aby bola správa rozkódovateľná, medzi jednotlivými zakódovanými písmenami musíme urobiť aj malú medzeru. Rovnako ako sme nevyhovujúce kódovanie e_3 upravili na bezprefixové (a teda vyhovujúce) e_4, aj tu preto do jazyka i na koniec každého slova formálne pridáme nejaký oddeľovač. Tým sa kódovanie stane bezprefixovým, a teda aj rozkódovateľným.

Toto kódovanie nie je blokové: napríklad také `e` alebo `t` sú kódované jedným znakom (bez spomínaného oddeľovača), kým niektoré písmená dokonca štyrmi znakmi. Prečo

p.	frekvencia	kód	d.
e	12,702 %	·	1
t	9,056 %	−	1
a	8,167 %	·−	2
o	7,507 %	−−−	3
i	6,966 %	··	2
n	6,749 %	−·	2
s	6,327 %	···	3
h	6,094 %	····	4
r	5,987 %	·−·	3
d	4,253 %	−··	3
l	4,025 %	·−··	3
c	2,782 %	−·−·	4
u	2,758 %	··−	3
m	2,406 %	−−	2
w	2,360 %	·−−	3
f	2,228 %	··−·	4
g	2,015 %	−−·	3
y	1,974 %	−·−−	4
p	1,929 %	·−−·	4
b	1,492 %	−···	4
v	0,978 %	···−	4
k	0,772 %	−·−	3
j	0,153 %	·−−−	4
x	0,150 %	−··−	4
q	0,095 %	−−·−	4
z	0,074 %	−−··	4

Samuel Finley Breese Morse
(1791–1872)



(http://en.wikipedia.org/wiki/Samuel_Morse)

toľká nespravodlivosť? Odpoveď je jednoduchá – hoci to na samotnej abecede nijako nepoznať, niektoré písmená sa v jazyku vyskytujú častejšie, iné menej často. V angličtine, Morseovej rodnej reči, sú frekvencie písmen približne také, ako uvádza tabuľka. Zrejmá (i keď nie úplne dokonalá) korelácia medzi frekvenciou výskytu písmena a dĺžkou jeho kódového slova nemôže byť náhodná, dá sa teda predpokladať, že Morse kódy písmenám prideľoval práve na základe podobnej tabuľky frekvencií. Prečo to robil, prečo neuprednostnil napríklad blokový kód, ktorý sa rozkóduva omnoho jednoduchšie?

Po vysvetlenie sa vráťme k našim dvom kódovaniam e_1 a e_2. Predpokladajme pritom, že naše štyri elementárne symboly majú v nami vysielaných správach takéto frekvencie (ich súčet je, samozrejme, 1) a kvôli prehľadnosti zopakujme aj obe kódovania:

s	frekvencia s	$e_1(s)$	$e_2(s)$
♥	0,3	00	10
♦	0,4	01	0
♣	0,2	10	110
♠	0,1	11	111

Zakódujme teraz 10-symbolovú správu ♣♦♥♥♠♦♣♦♥♦, ktorá výskytom jednotlivých symbolov zodpovedá uvedenej tabuľke. V kódovaní e_1 dostávame zakódovanú správu 10010000110110010001, ktorá má dĺžku 20. Na jeden symbol teda pripadajú priemerne 2 znaky, čo, samozrejme, nie je vôbec prekvapivé, keďže ide o blokové kódovanie dvojznakovými slovami, ktoré frekvencie úplne ignoruje. Kódovanie e_2 sa však, ako vidíme z tabuľky, snaží rešpektovať „morseovkovú“ zásadu, že častejším znakom zodpovedajú kratšie kódové slová. V ňom zakódovaná správa je 1100101011101100100. Tá má dĺžku 19, na jeden symbol teda pripadá $1{,}9$ znaku. Už pri takejto pomerne krátkej správe sme teda ušetrili jeden znak! Ľahko vidieť, že počet ušetrených znakov rastie priamo úmerne s dĺžkou správy, v ktorej početnosť symbolov zodpovedá uvedenej frekvenčnej tabuľke. Kódovanie e_2 je teda úspornejšie.

Ako však takúto úspornosť merať? A neexistuje ešte úspornejšie kódovanie? Možno cenu kódovania nejako ohraničiť?

Na rozdiel od symbolov správ, ktoré môžu mať (a veľmi často i majú) rôzne frekvencie, znaky kódovej abecedy (t. j. 0 a 1) máme plne pod kontrolou, nemáme preto dôvod uprednostňovať žiaden z nich. Oboma by sa teda malo začínať rovnaké množstvo zakódovaných správ. Napríklad ak majú naše symboly ♥, ♦, ♣ a ♠ frekvencie postupne $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$ a $\frac{1}{8}$, kódové slovo pre ♥ sa bude začínať napríklad znakom 0 a kódové slová pre ostatné tri symboly znakom 1. Keďže znakom 0 začína jediné kódové slovo, to už predlžovať netreba, pri zvyšnej trojici však musíme pokračovať. Symbol ♦ má rovnakú frekvenciu ako súčet frekvencií ♣ a ♠, začiatok jeho kódového slova teda bude (napríklad) 10, kým začiatok zvyšných dvoch 11. Ani kódové slovo pre ♦ už netreba predlžovať, treba však ešte rozlíšiť ♣ a ♠. Jedno preto predĺžime na 110, druhé na 111. Kódovanie teda vyzerá takto:

znak	frekvencia	kódové slovo
♥	$1/2$	0
♦	$1/4$	10
♣	$1/8$	110
♠	$1/8$	111

Všimnime si, že v súlade s predchádzajúcimi úvahami pre každý symbol je medzi jeho frekvenciou f a dĺžkou d jeho kódového slova vzťah $f = \frac{1}{2^d}$, resp. inverzne $d = -\log_2 f$. To teda znamená, že priemerná dĺžka slova bude súčtom výrazov tvaru $f \cdot (-\log_2 f)$, t. j. $-f \log_2 f$, čiže

$$-\frac{1}{2}\cdot\log_2\frac{1}{2} - \frac{1}{4}\cdot\log_2\frac{1}{4} - \frac{1}{8}\cdot\log_2\frac{1}{8} - \frac{1}{8}\cdot\log_2\frac{1}{8} =$$

$$= -\frac{1}{2}\cdot(-1) - \frac{1}{4}\cdot(-2) - \frac{1}{8}\cdot(-3) - \frac{1}{8}\cdot(-3) = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{3}{8} = \frac{9}{4} = 2{,}5.$$

Pravdaže, nie pre každú frekvenciu f je číslo $-\log_2 f$ prirodzené. Aby to teda mohla byť dĺžka slova, namiesto hodnoty $-\log_2 f$ vezmeme najbližšie väčšie prirodzené číslo (jeho tzv. hornú celú časť, označenú $\lceil -\log_2 f \rceil$). To teda znamená, že cenu každého kódu možno zdola ohraničiť nasledujúcou hodnotou:

Symbol H tu neznamená hlásku „há“, ide o grécke veľké písmeno eta.

Ak máme rozdelenie frekvencií $F = \langle f_1, f_2, \ldots, f_n \rangle$ (pre nejaké $n \geq 2$), číslo

$$\mathrm{H}_2(F) = (-f_1 \log_2 f_1) + (-f_2 \log_2 f_2) + \cdots + (-f_n \log_2 f_n) = -\sum_{i=1}^{n} f_i \log f_i.$$

nazývame entropiou (presnejšie 2-entropiou) rozdelenia frekvencií F.

Úloha 2.8 Vypočítajte entropiu $\mathrm{H}_2(F)$, ak $F = \langle 0{,}3; 0{,}4; 0{,}1; 0{,}2 \rangle$. Porovnajte s ňou ceny predchádzajúcich kódovaní e_1 a e_2.

V prípade optimálneho kódovania však môžeme jeho cenu pomocou entropie odhadnúť aj zhora, ako o tom hovorí slávna Shannonova veta o bezstratovom kódovaní:

Nech F je rozdelenie frekvencií a P je cena optimálneho kódovania pre F. Potom platí

$$\mathrm{H}_2(F) \leq P < \mathrm{H}_2(F) + 1.$$

Vieme už, že prvú nerovnosť spĺňa každé kódovanie. Ak však spĺňa i druhú, nazývame ho suboptimálne. V nasledujúcich dvoch statiach si ukážeme dva zaujímavé algoritmy na nájdenie suboptimálneho kódovania, v druhom prípade pôjde dokonca o kódovanie optimálne.

2.1.3. Shannonovo-Fanovo kódovanie

Prvý algoritmus je založený na už uvedenej myšlienke delenia postupnosti frekvencií na úseky s čo „najrovnakejšími“ súčtami. Pochádza zo 40. rokov a jeho dvoma nezávislými autormi sú Claude Shannon a Robert Fano. Uvedieme si ho na nasledujúcom príklade:

Claude Elwood Shannon (1916–2001)

(http://en.wikipedia.org/wiki/Claude_Shannon)

Najprv si uvedomme, že na konkrétnej podobe kódovaných symbolov vlastne vôbec nezáleží, pre kódovanie sú dôležité len ich frekvencie. Predpokladajme, že tie sú usporiadané do nerastúcej postupnosti

$$\langle 0{,}22; 0{,}18; 0{,}15; 0{,}15; 0{,}10; 0{,}08; 0{,}07; 0{,}02; 0{,}02; 0{,}01 \rangle$$

(samozrejme, súčet členov tejto postupnosti je 1). Skúsme teraz túto postupnosť rozdeliť na dva úseky s čo najbližšími súčtami. Keďže členov postupnosti je desať, do úvahy prichádza týchto deväť deliacich bodov:

1. Ak je deliace miesto po prvom člene, vzniknuté postupnosti sú $\langle 0{,}22 \rangle$ a $\langle 0{,}18; 0{,}15; 0{,}15; 0{,}10; 0{,}08; 0{,}07; 0{,}02; 0{,}02; 0{,}01 \rangle$. Ich súčty sú $0{,}22$ a $0{,}78$, ich rozdiel je $-0{,}56$.
2. Ak je deliace miesto po druhom člene, vzniknuté postupnosti sú $\langle 0{,}22; 0{,}18 \rangle$ a $\langle 0{,}15; 0{,}15; 0{,}10; 0{,}08; 0{,}07; 0{,}02; 0{,}02; 0{,}01 \rangle$. Ich súčty sú $0{,}40$ a $0{,}60$, ich rozdiel je $-0{,}20$.
3. Ak je deliace miesto po treťom člene, vzniknuté postupnosti sú $\langle 0{,}22; 0{,}18; 0{,}15 \rangle$ a $\langle 0{,}15; 0{,}10; 0{,}08; 0{,}07; 0{,}02; 0{,}02; 0{,}01 \rangle$. Ich súčty sú $0{,}55$ a $0{,}45$, ich rozdiel je $0{,}10$.
4. Ak je deliace miesto po štvrtom člene, vzniknuté postupnosti sú $\langle 0{,}22; 0{,}18; 0{,}15; 0{,}15 \rangle$ a $\langle 0{,}10; 0{,}08; 0{,}07; 0{,}02; 0{,}02; 0{,}01 \rangle$. Ich súčty sú $0{,}70$ a $0{,}30$, ich rozdiel je $0{,}40$.

Robert Mario Fano (1917)

(http://www.youtube.com/watch?v=sjnmcKVnLi0)

5 Ak je deliace miesto po piatom člene, vzniknuté postupnosti sú $\langle 0{,}22; 0{,}18; 0{,}15; 0{,}15; 0{,}10\rangle$ a $\langle 0{,}08; 0{,}07; 0{,}02; 0{,}02; 0{,}01\rangle$. Ich súčty sú $0{,}80$ a $0{,}20$, ich rozdiel je $0{,}60$.

6 Ak je deliace miesto po šiestom člene, vzniknuté postupnosti sú $\langle 0{,}22; 0{,}18; 0{,}15; 0{,}15; 0{,}10; 0{,}08\rangle$ a $\langle 0{,}07; 0{,}02; 0{,}02; 0{,}01\rangle$. Ich súčty sú $0{,}88$ a $0{,}12$, ich rozdiel je $0{,}76$.

7 Ak je deliace miesto po siedmom člene, vzniknuté postupnosti sú $\langle 0{,}22; 0{,}18; 0{,}15; 0{,}15; 0{,}10; 0{,}08; 0{,}07\rangle$ a $\langle 0{,}02; 0{,}02; 0{,}01\rangle$. Ich súčty sú $0{,}95$ a $0{,}05$, ich rozdiel je $0{,}90$.

8 Ak je deliace miesto po ôsmom člene, vzniknuté postupnosti sú $\langle 0{,}22; 0{,}18; 0{,}15; 0{,}15; 0{,}10; 0{,}08; 0{,}07; 0{,}02\rangle$ a $\langle 0{,}02; 0{,}01\rangle$. Ich súčty sú $0{,}97$ a $0{,}03$, ich rozdiel je $0{,}94$.

9 Ak je deliace miesto po deviatom člene, vzniknuté postupnosti sú $\langle 0{,}22; 0{,}18; 0{,}15; 0{,}15; 0{,}10; 0{,}08; 0{,}07; 0{,}02; 0{,}02\rangle$ a $\langle 0{,}01\rangle$. Ich súčty sú $0{,}99$ a $0{,}01$, ich rozdiel je $0{,}98$.

To, že sú si súčty najbližšie, vlastne znamená, že absolútna hodnota ich rozdielu je najmenšia. V postupnosti $\langle -0{,}60; -0{,}20; 0{,}10; 0{,}40; 0{,}60; 0{,}76; 0{,}90; 0{,}94; 0{,}98\rangle$ rozdielov z predchádzajúcich deviatich možností rozdelenia má najmenšiu absolútnu hodnotu tretí člen $0{,}10$, hľadaná možnosť je teda tretia. To teda znamená, že kódové slová prvých troch frekvencií budú začínať jedným znakom, kým zvyšné tým druhým:

frekvencia	začiatok kódového slova
$0{,}22$	0
$0{,}18$	0
$0{,}15$	0
$0{,}15$	1
$0{,}10$	1
$0{,}08$	1
$0{,}07$	1
$0{,}02$	1
$0{,}02$	1
$0{,}01$	1

Kým budeme pokračovať, všimnime si, že spomínaná postupnosť rozdielov je rastúca (isteže, veď v každom kroku presunieme nejaké kladné číslo z druhej podpostupnosti do prvej). Ak v nej teda prídeme k nejakému kladnému členu (prvých pár môže byť záporných), v hľadaní ideálnej možnosti už pokračovať netreba, lebo rozdiel (rovný svojej absolútnej hodnote) sa už bude iba zväčšovať. To teda znamená, že naše hľadanie sme mohli ukončiť už po tretej možnosti.

Vráťme sa opäť k určovaniu kódových slov. Opäť budeme pokračovať rovnakým spôsobom, tentoraz však už pre prvú vzniknutú podpostupnosť $\langle 0{,}22; 0{,}18; 0{,}15\rangle$. Tu máme len dve možnosti delenia:

1 Ak je deliace miesto po prvom člene, vzniknuté postupnosti sú $\langle 0{,}22\rangle$ a $\langle 0{,}18; 0{,}15\rangle$. Ich súčty sú $0{,}22$ a $0{,}33$, ich rozdiel je $-0{,}11$.

2 Ak je deliace miesto po druhom člene, vzniknuté postupnosti sú $\langle 0{,}22; 0{,}18\rangle$ a $\langle 0{,}15\rangle$. Ich súčty sú $0{,}40$ a $0{,}15$, ich rozdiel je $0{,}25$.

Prvá možnosť je lepšia, čo znamená, že prvé kódové slovo predĺžime na `00` a druhé dve na `01`:

frekvencia	začiatok kódového slova
0,22	00
0,18	01
0,15	01
0,15	1
0,10	1
0,08	1
0,07	1
0,02	1
0,02	1
0,01	1

Nasleduje jednočlenná postupnosť $\langle 0{,}22 \rangle$. Tá sa rozdeliť nedá, ale – na druhej strane – si uvedomme, že kódové slovo s príslušným začiatkom 00 je jediné, jeho predĺžením by sme cenu výsledného kódovania iba zbytočne zväčšovali. Kódové slovo 00 tak môžeme považovať za definitívne. Úplne analogicky tak budeme robiť pri každej jednočlennej postupnosti.

Pri nasledujúcej dvojčlennej postupnosti $\langle 0{,}18; 0{,}15 \rangle$ máme jedinú možnosť rozdelenia, tá je zrejme ideálna. Začiatok kódového slova prislúchajúceho prvému členu tak predĺžime na 010, pri druhom člene to bude 011. Obe vzniknuté jednočlenné postupnosti už ošetriť vieme – pri každej doterajší začiatok kódového slova prehlásime za jeho konečnú podobu:

frekvencia	začiatok kódového slova
0,22	00 ✓
0,18	010 ✓
0,15	011 ✓
0,15	1
0,10	1
0,08	1
0,07	1
0,02	1
0,02	1
0,01	1

Poďme na nasledujúcu postupnosť – $\langle 0{,}15; 0{,}10; 0{,}08; 0{,}07; 0{,}02; 0{,}02; 0{,}01 \rangle$. Opäť hľadajme čo „najspravodlivejšie“ rozdelenie:

1. Ak je deliace miesto po prvom člene, vzniknuté postupnosti sú $\langle 0{,}15 \rangle$ a $\langle 0{,}10; 0{,}08; 0{,}07; 0{,}02; 0{,}02; 0{,}01 \rangle$. Ich súčty sú $0{,}15$ a $0{,}30$, ich rozdiel je $-0{,}15$.
2. Ak je deliace miesto po druhom člene, vzniknuté postupnosti sú $\langle 0{,}15; 0{,}10 \rangle$ a $\langle 0{,}08; 0{,}07; 0{,}02; 0{,}02; 0{,}01 \rangle$. Ich súčty sú $0{,}25$ a $0{,}20$, ich rozdiel je $0{,}05$.

Keďže sme dospeli ku kladnému rozdielu, ďalej už, ako vieme, postupovať netreba. Ideálne miesto rozdelenia na dve podpostupnosti je teda druhé. To znamená, že kódové slová prvých dvoch príslušných frekvencií budú mať začiatky 10 a zvyšné na 11:

frekvencia	začiatok kódového slova
0,22	00 ✓
0,18	010 ✓
0,15	011 ✓
0,15	10
0,10	10
0,08	11
0,07	11
0,02	11
0,02	11
0,01	11

Analogicky postupujeme ďalej. Výsledné Shannonovo-Fanovo kódovanie je takéto:

frekvencia	kódové slovo
0,22	00
0,18	010
0,15	011
0,15	100
0,10	101
0,08	110
0,07	1110
0,02	11110
0,02	111110
0,01	111111

Úloha 2.9	Vypočítajte cenu tohto kódovania. Porovnajte ju s príslušnou entropiou.
Úloha 2.10	Dokončite vynechané iterácie algoritmu.
Úloha 2.11	Nájdite Shannonovo-Fanovo kódovanie písmen latinskej abecedy s vyššie uvedenými frekvenciami ich výskytov v angličtine.

Výhodou Shannonovho-Fanovho kódovacieho algoritmu je jeho intuitívnosť a priamočiarosť. Žiaľ, nie vždy je ním nájdené kódovanie optimálne. Napríklad pre postupnosť $\langle 0{,}35; 0{,}17; 0{,}17; 0{,}16; 0{,}15\rangle$ sú nájdené kódové slová $\langle 00, 01, 10, 110, 111\rangle$, cena tohto kódovania je teda

$$0{,}35 \cdot 2 + 0{,}17 \cdot 2 + 0{,}17 \cdot 2 + 0{,}16 \cdot 3 + 0{,}15 \cdot 3 = 2{,}31.$$

Existuje však iné kódovanie, a to napríklad s kódovými slovami $\langle 0, 100, 101, 110, 111\rangle$, ktorého cena je

$$0{,}35 \cdot 1 + 0{,}17 \cdot 3 + 0{,}17 \cdot 3 + 0{,}16 \cdot 3 + 0{,}15 \cdot 3 = 2{,}30.$$

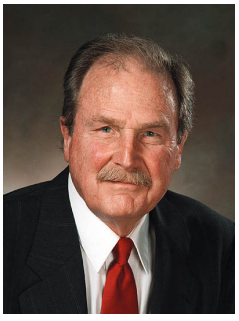
Shannonovo-Fanovo kódovanie je zaujímavé i svojou štruktúrou. Tento algoritmus je rekurzívny, ergo induktívny, avšak jeho parametre (konkrétne dĺžky vznikajúcich podpostupností) sú určené až dátami. Na spracovanie postupnosti istej dĺžky teda potrebujeme vedieť spracovať postupnosti ľubovoľných menších dĺžok.

2.1.4. Huffmanovo optimálne kódovanie

Aj ďalší algoritmus má induktívny charakter, tu použitá indukcia je však úplne klasická – spracovanie postupnosti s danou dĺžkou sa prevádza na spracovanie postupnosti o jeden člen kratšej.

Autorom tohto algoritmu je Fanov žiak David Huffman, ktorý ho vymyslel v roku 1951 práve na popud svojho slávneho učiteľa. Aj Huffmanovo kódovanie si predvedieme na príklade: Vezmime si opäť nerastúcu postupnosť frekvencií

David Albert Huffman (1925–1999)



(http://www.adeptis.ru/vinci/m_part5_2.html)

$$F_{10} = \langle 0{,}22; 0{,}18; 0{,}15; 0{,}15; 0{,}10; 0{,}08; 0{,}07; 0{,}02; \underline{0{,}02}; \underline{0{,}01}\rangle.$$

Túto desaťčlennú postupnosť zredukujeme na deväťčlennú tak, že najmenšie dve frekvencie – $0{,}02$ a $0{,}01$ – nahradíme ich súčtom $0{,}03$, ktorý pripíšeme (napríklad) na koniec:

$$F_9 = \langle 0{,}22; 0{,}18; 0{,}15; 0{,}15; 0{,}10; 0{,}08; 0{,}07; \underline{0{,}02}; \underline{0{,}03}\rangle.$$

Opäť redukujeme rovnakým spôsobom: dve najmenšie frekvencie – $0{,}02$ a $0{,}03$ – nahradzujeme ich súčtom $0{,}05$ a pripíšeme ho na koniec:

$$F_8 = \langle 0{,}22; 0{,}18; 0{,}15; 0{,}15; 0{,}10; 0{,}08; \underline{0{,}07}; \underline{0{,}05}\rangle.$$

Analogicky postupujeme ďalej. Všimnime si, že, samozrejme, súčet každej takejto postupnosti je vždy 1, veď len zlučujeme niektoré sčítance:

$$F_7 = \langle 0{,}22; 0{,}18; 0{,}15; 0{,}15; \underline{0{,}10}; \underline{0{,}08}; 0{,}12\rangle.$$

$$F_6 = \langle 0{,}22; 0{,}18; 0{,}15; \underline{0{,}15}; \underline{0{,}12}; 0{,}18\rangle.$$

Za zmienku stojí, že sme tu mali dve možnosti výberu hodnoty $0{,}15$. Výsledné kódovanie sa tým síce zmení, jeho cena (a teda ani prípadná optimalita) však nie.

$$F_5 = \langle 0{,}22; 0{,}18; \underline{0{,}15}; \underline{0{,}18}; 0{,}27\rangle.$$

Aj tu máme dve možnosti, obe fungujú rovnako dobre.

$$F_4 = \langle \underline{0{,}22}; \underline{0{,}18}; 0{,}27; 0{,}33\rangle.$$

$$F_3 = \langle \underline{0{,}27}; \underline{0{,}33}; 0{,}40\rangle.$$

$$F_2 = \langle 0{,}40; 0{,}60\rangle.$$

Ďalšia redukcia už zrejme nemá zmysel, dostali by sme jednočlennú postupnosť $\langle 1\rangle$. Našťastie ani nie je potrebná, veď dve hodnoty vieme rozdeliť optimálne (až na poradie) jediným spôsobom: prvé kódové slovo bude napríklad `0`, druhé `1`. Kratšie to zrejme nepôjde. Označme pre každé n z množiny $\{2, 3, \ldots, 10\}$ kódovú postupnosť K_n. Takže máme

$$K_2 = \langle 0, 1\rangle.$$

Zopakujme, že postupnosť F_2 vznikla z F_3 tak, že sa jej prvé dva členy nahradili ich súčtom, ktorý sa stal koncom postupnosti F_2. Tento súčet má v K_2 kódové slovo `1`, takže prvým dvom členom v F_3 priradíme kódové slová `10` a `11`. Dostávame teda

$$K_3 = \langle \underline{10}, \underline{11}, 0\rangle.$$

Postupnosť F_3 vznikla tak, že sme z postupnosti F_4 odobrali (opäť) jej prvé dva členy a nahradili ich koncom F_3. Ten má v K_3 kódové slovo `0`, prvé dva členy F_4 teda budú mať kódové slová `00` a `01`. Takže

$$K_4 = \langle \underline{00}, \underline{01}, 10, 11\rangle.$$

Postupnosť F_4 vznikla z postupnosti F_5 nahradením jej tretieho a štvrtého člena posledným členom F_4. Keďže ten má v K_4 kódové slovo `11`, v F_5 bude mať tretí člen kódové slovo `110` a štvrtý `111`. To znamená, že

$$K_5 = \langle 00, 01, \underline{110}, \underline{111}, 10\rangle.$$

Analogicky postupujeme ďalej:

$$K_6 = \langle 00, 01, 110, 100, 101, 111\rangle.$$

$$K_7 = \langle 00, 01, 110, 100, \underline{1110}, \underline{1111}, 101\rangle.$$

$$K_8 = \langle 00, 01, 110, 100, 1110, 1111, \underline{1010}, \underline{1011}\rangle.$$

$$K_9 = \langle 00, 01, 110, 100, 1110, 1111, 1010, \underline{10110}, \underline{10111}\rangle.$$

$$K_{10} = \langle 00, 01, 110, 100, 1110, 1111, 1010, 10110, \underline{101110}, \underline{101111}\rangle.$$

Dá sa dokázať (to však presahuje priestorové možnosti tohto materiálu), že pre každé n z množiny $\{2, 3, \ldots, 10\}$ je K_n optimálne kódovanie k postupnosti frekvencií F_n. Výsledné Huffmanovo kódovanie je teda takéto:

frekvencia	kódové slovo
0,22	00
0,18	01
0,15	110
0,15	100
0,10	1100
0,08	1111
0,07	1010
0,02	10110
0,02	101110
0,01	101111

Úloha 2.12	Vypočítajte cenu tohto kódovania. Porovnajte ju s cenou Shannonovho-Fanovho kódovania (z predchádzajúce state) a s entropiou.
Úloha 2.13	Pri vytváraní postupnosti F_4 si vyberte druhú možnosť a nájdite príslušné kódovanie. Vypočítajte jeho cenu a porovnajte ho s cenou predchádzajúceho kódovania.
Úloha 2.14	Pri vytváraní postupnosti F_6 si vyberte druhú možnosť a nájdite príslušné kódovanie. Vypočítajte jeho cenu a porovnajte ho s cenou predchádzajúceho kódovania.
Úloha 2.15	Nájdite Huffmanovo kódovanie písmen latinskej abecedy s vyššie uvedenými frekvenciami ich výskytov v angličtine.

2.2. Prenos zakódovanej informácie

2.2.1. Registrovanie a opravenie chyby

Vieme už, ako danú informáciu zakódovať — transforomovať do prenositeľnej formy — i rozkódovať — urobiť opačnú transformáciu. Prvý z týchto procesov používame, keď sme vysielateľom informácie, druhý v prípade, že sme jej príjemcom. Čo sa však medzitým so zakódovanou informáciou deje? Ak jej vysielateľ a jej príjemca sídlia na rôznych miestach, informáciu bolo treba nejakým spôsobom preniesť od jedného k druhému (veď sme ju kódovali práve preto, aby bol tento proces jednoduchší). Treba si však uvedomiť, že tento prenos nemusí byť dokonalý, mali by sme sa preto proti jeho potenciálnym chybám vyzbrojiť.

Problém prenosu informácie, pravdaže, nie je len záležitosťou počítačov. Iste sme sa s ním už neraz stretli napríklad pri ceste osobákom, keď staničný hlásateľ, vedomý si nekvality ozvučenia predpotopnými chrčiacimi amplión mi, informáciu o príchode či odchode vlaku radšej dvakrát zopakoval. Môžeme si od neho vziať poučenie: Prenášanú informáciu jednoducho zopakujeme, takže trebárs namiesto slova 10010 budeme prenášať slovo 1001010010. Prijímateľ (o ktorom, samozrejme predpokladáme, že je o takomto duplikovaní informovaný) tak vie, že prijímané slovo je len polovicou prijatého reťazca — a je jedno, či prvou, alebo druhou, keďže sú rovnaké.

Pri nedokonalom prenose však môžu nastať dva problémy:

- Ak má prijaté slovo nepárny počet znakov, a teda nemožno ho rozdeliť na polovice, zrejme sa (aspoň) jeden znak stratil alebo pribudol.
- Problém však môže nastať aj v prípade, že počet znakov ostane zachovaný, no polovice prijatej správy sa nezhodujú — ktorá z nich je potom platná?

Prenosové chyby prvého typu sú v podstate neopraviteľné. Obvykle však máme popri samotnej správe aj informáciu o jej veľkosti, takže vieme zistiť, že nedošla v poriadku. V takom prípade je asi najjednoduchšie správu ignorovať a proces jej prenosu radšej zopakovať. V ďalšom sa preto budeme zaoberať iba druhým typom chyby — keď nastane zmena znaku 1 na znak 0 alebo naopak. To teda znamená, že budeme uvažovať len blokové kódy.

Ak teda príjemca prijme napríklad správu 1001011010, vie, že nastala nejaká prenosová chyba, nevie však zistiť aká, a teda ju nevie ani opraviť. Tomuto problému sa však môžeme do istej miery vyhnúť — stačí uvedenú správu poslať trikrát za sebou, takže v našom prípade 100101001010010. Príjemca si prijatú správu (ktorá má, ako predpokladáme, správnu dĺžku) rozdelí na tretiny a navzájom ich porovná. V prípade, že pri prenose nastala jediná chyba, dve z týchto tretín ostanú nepoškodené. Stačí mu teda porovnať všetky tri tretiny a vybrať tú, ktorá sa zhoduje s aspoň jednou ďalšou tretinou (v prípade jednej chyby nastáva zhoda s práve jednou, a ak je prenos bezchybný, zrejme s oboma). Napríklad ak prijal správu 100101011010010,

ktorej tretiny sú 10010, 10110 a 10010 (čiže prvá a tretia sa zhodujú), pôvodná správa bola (za predpokladu jedinej chyby) 100101001010010. V tomto prípade teda príjemca môže chybu nielen zaregistrovať, ale aj opraviť.

Môže sa, samozrejme, stať, že chýb pri prenose sa stane viac, pri (oproti dĺžke prenášanej správy) dostatočne malej pravdepodobnosti jednej nechcenej zmeny znaku je však pravdepodobnosť dvoch či viacerých chýb zanedbateľná.

Úloha 2.16	Ak je dĺžka prenášanej správy n a pravdepodobnosť zmeny znaku je p, zistite pravdepodobnosť, že nastane k zmien znakov.
Riešenie	Kvôli jednoduchosti najprv predpokladajme, že chyba nastane pri prvých k znakoch. Tento jav teda nastane s pravdepodobnosťou $$\underbrace{p \cdot p \cdots p}_{k} \cdot \underbrace{(1-p) \cdot (1-p) \cdots (1-p)}_{n-k} = p^k (1-p)^{n-k}.$$ Rovnaký súčin (hoci v rôznom poradí činiteľov) však dostaneme pri ľubovoľnom rozmiestnení k chýb. Takýchto rozmiestnení je $\binom{n}{k}$, pričom každé dve rozmiestnenia sa navzájom vylučujú. Hľadaná pravdepodobnosť k zmien znakov je teda $$\binom{n}{k} p^k (1-p)^{n-k}$$ (čo je, mimochodom, k. člen rozkladu $((1-p)+p)^n = 1$). Takže napríklad pri 10-znakovej správe a pravdepodobnosti zmeny znaku $\frac{1}{100}$ je pravdepodobnosť bezchybného prenosu $$\binom{10}{0} \left(\frac{1}{100}\right)^0 \left(1-\frac{1}{100}\right)^{10-0} = \left(\frac{99}{100}\right)^{10} \approx 0{,}90438,$$ pravdepodobnosť jednej chyby $$\binom{10}{1} \left(\frac{1}{100}\right)^1 \left(1-\frac{1}{100}\right)^{10-1} = \frac{1}{10} \left(\frac{99}{100}\right)^{9} \approx 0{,}09135$$ a pravdepodobnosť dvoch chýb $$\binom{10}{2} \left(\frac{1}{100}\right)^2 \left(1-\frac{1}{100}\right)^{10-2} = \frac{45}{10000} \left(\frac{99}{100}\right)^{8} \approx 0{,}00415.$$

2.2.2. Kontrolné súčty

Opakovanie správy však nie je jediná možnosť, ako výrazne zmenšiť pravdepodobnosť chyby. Veľmi známou metódou je i kontrola parity. Keď posielame nejakú správu, na jej koniec pridáme jeden znak, a to tak, aby výsledná predĺžená správa obsahovala párny počet znakov 1. Napríklad správu 1101001 upravíme na správu 11010010 pridaním nuly (aby sme dostali párny počet jednotiek), kým správu 0101001 upravíme na správu 010100101 pridaním jednotky. Ak prijímateľovi príde správa s nepárnym počtom jednotiek, bude vedieť, že nastala nejaká chyba (nie však to, na ktorom mieste).

Na kontrole parity je však založená i metóda, ktorou jednu chybu vieme nielen zaregistrovať, ale i opraviť. Kvôli jednoduchosti predpokladajme, že naša správa má 21 znakov, a teda jej znaky môžeme zobraziť do obdĺžnika 7×3. Nech je to táto správa:

```
0 1 0 0 0 0 1
1 1 0 1 0 0 0
0 1 1 0 0 1 1
```

Teraz ku každému riadku i každému stĺpcu doplníme kontrolný znak, ktorým doplníme príslušný počet jednotiek na párne číslo. Do pravého dolného rohu nového obdĺžnika 8×4 navyše doplníme znak, ktorým doplníme na párne číslo počet jednotiek v celom pôvodnom obdĺžniku:

0	1	0	0	0	0	1	0
1	1	0	1	0	0	0	1
0	1	1	0	0	1	1	0
1	1	1	1	0	1	0	1

Vyjadrovanie typu „kontrolný znak, ktorým doplníme príslušný počet jednotiek na párne číslo“ je dosť ťažkopádne. Môžeme ho však podstatne zjednodušiť pomocou operácie $\oplus$ („xor“, resp. exkluzívne alebo), ktorá je definovaná takto:

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Ak si navyše uvedomíme, že ide vlastne o sčítanie zvyškových tried pri delení dvomi, ľahko vidieť, že operácia $\oplus$ je komutatívna i asociatívna, a môžeme ju preto pokojne nazývať súčet. Pridaný znak na koniec každého riadka či stĺpca je teda súčtom jeho členov a budeme ho nazývať kontrolný súčet. Podobne posledný pridaný znak je kontrolným súčtom celej tabuľky.

Predpokladajme teraz, že pri prenose predĺženej 32-znakovej správy nastala jedna chyba a prijatá správa je takáto:

0	1	0	0	0	0	1	0
1	1	1	1	0	0	0	1
0	1	1	0	0	1	1	0
1	1	1	1	0	1	0	1

Ľahko vidíme, že nesedia kontrolné súčty v druhom riadku a treťom stĺpci:

0	1	0	0	0	0	1	0
1	1	1	1	0	0	0	1
0	1	1	0	0	1	1	0
1	1	1	1	0	1	0	1

Chyba teda nastala v ich prieniku a môžeme ju ľahko opraviť:

0	1	0	0	0	0	1	0
1	1	0	1	0	0	0	1
0	1	1	0	0	1	1	0
1	1	1	1	0	1	0	1

Je zaujímavé, že zaregistrovať vieme aj dve (ba dokonca až tri) chyby, tie už však opraviť nemusíme vždy vedieť. Napríklad pri prijatej správe

0	1	0	1	0	0	1	0
1	1	0	1	0	0	0	1
0	0	1	0	0	1	1	0
1	1	1	1	0	1	0	1

nesedia tieto kontrolné súčty:

0	1	0	1	0	0	1	0
1	1	0	1	0	0	0	1
0	0	1	0	0	1	1	0
1	1	1	1	0	1	0	1

Oprava by však mohla dopadnúť dvoma rovnocennými spôsobmi:

0	0	0	1	0	0	1	0
1	1	0	1	0	0	0	1
0	0	1	1	0	1	1	0
1	1	1	1	0	1	0	1

0	1	0	0	0	0	1	0
1	1	0	1	0	0	0	1
0	1	1	0	0	1	1	0
1	1	1	1	0	1	0	1

2.2.3. Hammingova vzdialenosť

Pod vzdialenosťou dvoch objektov obvykle rozumieme nejaké nezáporné číslo. Ak je objektov viacero, takéto číslo je priradené každej z nich vytvorenej dvojici. To teda znamená, že ak objekty tvoria množinu A, vzdialenosť bude zobrazenie z kartézianskeho súčinu $A \times A$ (ten obsahuje dvojice objektov z A) do množiny nezáporných čísel. Nie však hocijaké, musí spĺňať isté vlastnosti. Napríklad takú, že vzdialenosť objektu od seba samého je nulová, kým jeho vzdialenosť od každého iného objektu je kladná. Inou vlastnosťou je tzv. symetria – vzdialenosť medzi dvoma objektmi nezávisí od toho, z ktorej strany ju meriame. Asi najzaujímavejšou vlastnosťou je tzv. trojuholníková nerovnosť – vzdialenosť medzi dvoma objektmi nesmie presiahnuť súčet ich vzdialeností od ľubovoľného tretieho objektu. Matematicky možno tieto vlastnosti zhrnúť do definície:

A čo jednosmerky? ;-)

Nech M je množina. Potom zobrazenie d z množiny $M \times M$ do nezáporných reálnych čísel nazývame vzdialenosť, ak spĺňa nasledujúce vlastnosti:

- $d(x, y) = 0$ práve vtedy, keď $x = y$.
- $d(x, y) = d(y, x)$.
- $d(x, y) \leq d(x, z) + d(z, y)$.

Určite najpoužívanejšou vzdialenosťou je euklidovská, založená na Pytagorovej vete: Ak sú $\langle x_1, y_1 \rangle$ a $\langle x_2, y_2 \rangle$ body v rovine, ich euklidovská vzdialenosť je

$$d(\langle x_1, y_1 \rangle, \langle x_2, y_2 \rangle) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}.$$

(Dá sa ľahko ukázať, že táto funkcia skutočne spĺňa všetky tri vlastnosti z definície.) V tomto prípade je množina M z definície vzdialenosti rovná množine $\mathbb{R} \times \mathbb{R}$ dvojíc reálnych čísel, vzdialenosť však môžeme definovať aj na iných množinách.

V predchádzajúcej stati sme pracovali s kódovými slovami danej dĺžky (napríklad 21 alebo 32) vytvorenými zo znakov `0` a `1`. Prirodzeným kandidátom na vzdialenosť medzi nimi je počet nevyhnutných zmien, ktorými z jedného slova vytvoríme druhé. Napríklad slová `110101` a `100110` (samozrejme, sú rovnakej dĺžky) sa líšia práve na troch miestach (druhom, piatom a šiestom), čo znamená, že ich vzdialenosť bude 3. Za predpokladu, že znak `0` stotožníme s číslom 0 a znak `1` s číslom 1, môžeme tento pojem vyjadriť aj matematicky: Ak a a b sú (rovnaké alebo rôzne) znaky/čísla `0`/0 alebo `1`/1, číslo $|a - b|$ má hodnotu 0 vtedy, keď sú čísla a a b rovnaké, kým hodnota 1 sa nadobúda práve vtedy, ak sa líšia (buď platí $a = 0$ a $b = 1$, alebo $a = 1$ a $b = 0$). Vzdialenosť teda môžeme definovať takto:

Nech n je kladné prirodzené číslo a $a_1 a_2 \ldots a_n$ a $b_1 b_2 \ldots b_n$ sú slová z abecedy $\{0, 1\}$. Hammingovou vzdialenosťou týchto dvoch slov nazývame hodnotu

$$d(a_1 a_2 \ldots a_n, b_1 b_2 \ldots b_n) = |a_1 - b_1| + |a_1 - b_1| + \cdots + |a_n - b_n| = \sum_{i=1}^{n} |a_i - b_i|.$$

Napríklad Hammingova vzdialenosť spomínaných slov 110101 a 100110 je

$$d(110101, 100110) = |1-1|+|1-0|+|0-0|+|1-1|+|0-1|+|1-0| = 0+1+0+0+1+1 = 3.$$

Do tohto súčtu každá zmena prispeje hodnotou 1 a každá ne-zmena hodnotou 0, výsledok je teda práve počet zmien medzi slovami.

Úloha 2.17	Ukážte, že Hammingova vzdialenosť spĺňa podmienky definície vzdialenosti.

Ak nejaký (samozrejme, blokový) kód obsahuje dve slová s Hammingovou vzdialenosťou 1 (napríklad 11000 a 11010), v prípade niektorých prijatých správ (tu napríklad 11010) nedokážeme zistiť, či ide o pôvodné slovo, alebo či náhodou nenastala chyba (tu v štvrtom znaku). To však znamená, že ak chceme vedieť zaregistrovať aspoň jednu chybu, Hammingova vzdialenosť každých dvoch slov musí byť aspoň 2. To bol napríklad prípad kódu s kontrolným súčtom.

Ak zasa máme v kóde nejaké dve slová s Hammingovou vzdialenosťou 2 (napríklad 11000 a 11011), pri niektorých správach (tu napríklad 11010) síce chybu dokážeme zaregistrovať (ak, pravdaže, samotné slovo 11010 nie je kódové), ale nedokážeme ju opraviť. Takže ak máme byť schopní nielen zaregistrovať, ale i opraviť aspoň jednu chybu, Hammingova vzdialenosť každých dvoch slov musí byť aspoň 3.

Tieto poznatky môžeme prirodzene zovšeobecniť:

- Blokový kód vie zaregistrovať n chýb, ak každé jeho dve slová majú Hammingovu vzdialenosť aspoň $n+1$.
- Blokový kód vie opraviť n chýb, ak každé jeho dve slová majú Hammingovu vzdialenosť aspoň $2n+1$.

2.3. Literatúra

[1] R. Togneri, C. J. S. DeSilva: *Fundamentals of information theory and coding design*, Chapman & Hall/CRC, 2002, ISBN: 978-1-58488-310-3

[2] J. Adámek: *Kódování a teorie informace*, skriptá ČVUT, ISBN 80-01-00661-1

[3] D. Olejár, M. Stanek: *Úvod do teórie kódovania*, http://www.dcs.fmph.uniba.sk/texty/codebook.pdf

Čo sme sa naučili v tomto module

Zhrnutie

Tento modul zoznámil s

- interpretermi, kompilátormi a princípmi ich práce,
- dôvodmi i základnými princípmi kódovania informácií a s niektorými konkrétnymi typmi kódovania.

Tento študijný materiál vznikol ako súčasť národného projektu Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika v rámci Aktivity „Vzdelávanie nekvalifikovaných učiteľov informatiky na 2. stupni ZŠ a na SŠ“.

Autori © RNDr. František Galčík, PhD.
doc. RNDr. Stanislav Krajči, PhD.

Názov Ďalšie vzdelávanie učiteľov základných škôl a stredných škôl v predmete informatika

Podnázov Počítačové systémy 4

Študijný materiál prešiel recenzným pokračovaním.

Recenzenti doc. Ing. Peter Fabián, PhD.
Ing. Radoslav Gargalík

Počet strán 40

Náklad 300 ks

Prvé vydanie, Bratislava 2010

Vydal Štátny pedagogický ústav, Pluhová 8, 830 00 Bratislava, v súčinnosti s Univerzitou Pavla Jozefa Šafárika v Košiciach, Univerzitou Komenského v Bratislave, Univerzitou Konštantína Filozofa v Nitre, Univerzitou Mateja Bela v Banskej Bystrici a Žilinskou univerzitou v Žiline

Vytlačil BRATIA SABOVCI, s r.o., Zvolen

ISBN 978-80-8118-064-4