Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE

Jakub Jelínek

Fundamenty informatického myšlení

Katedra softwarového inženýrství

Vedoucí bakalářské práce: Doc. RNDr. Pavel Pyrih Csc.,
Katedra matematické analýzy

Studijní program: Informatika, programování

2008

Na tomto místě bych chtěl poděkovat docentu Pyrihovi za vedení při psaní práce.

Prohlašuji, že jsem svou bakalářskou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne 21. května 2008

Jakub Jelínek

Obsah

1 Úvod.....5
1.1 Definice informatiky.....5
1.2 Definice fundamentu.....6
1.3 Definice myšlení.....7
2 Statické fundamenty.....9
2.1 Informace.....10
2.2 Čas.....13
2.3 Prostor.....16
2.4 Peníze.....19
3 Dynamické fundamenty.....21
3.1 Abstrakce.....21
3.2 Iterace a rekurze.....24
3.3 Metasyntaktické proměnné.....27
3.4 Univerzalita.....31
3.5 Jednoduchost.....33
3.6 Inspirace.....34
4 Shrnutí.....36
5 Příloha A: Implementace rekurzivního stromu.....37

Název práce: Fundamenty informatického myšlení

Autor: Jakub Jelínek

Katedra (ústav): Katedra softwarového inženýrství

Vedoucí bakalářské práce: Doc. RNDr. Pyrih Pavel CSc., Katedra matematické analýzy

e-mail vedoucího: pavel.pyrih@mff.cuni.cz

Abstrakt: V předložené práci prezentuje autor výsledky svého hledání podstatných rysů informatického myšlení, jeho fundamentů. Rozděluje fundamenty na dvě kategorie – statické a dynamické. Statické fundamenty přestavují základní stavební materiál, ze kterého je vystavěn informatický svět. Za statické fundamenty považuje informace, čas, prostor a peníze. Dynamické fundamenty vyjadřují základní „aktivní činitele", které využívají statické fundamenty a přeměňují je, čímž vytvářejí složitější elementy informatiky. Za dynamické fundamenty jsou považovány abstrakce, iterace a rekurze, metasyntaktické proměnné, univerzalita, jednoduchost a inspirace.

Klíčová slova: informatika, složitost, abstrakce, rekurze, univerzalita

Title: Fundaments of IT thinking

Author: Jakub Jelínek

Department: Department of Software Engineering

Supervisor: Doc. RNDr. Pyrih Pavel CSc., Department of Mathematical Analysis

Supervisor's e-mail address: pavel.pyrih@mff.cuni.cz

Abstract: In the propounded work author presents results of his looking for important informatics thinking features, its fundaments. He divides these fundaments into two categories – static and dynamic. Static fundaments represent basic building material that the informatics world is built from. As static fundaments are considered information, time, room and money. Dynamic fundaments express basic “active factors” which are using static fundaments and are transforming them, creating so more complicated elements of informatics. As dynamic fundaments are considered abstraction, iteration and recursion, metasyntactic variables, universality, simplicity and inspiration.

Keywords: informatics, complexity, abstraction, recursion, universality

1 Úvod

V exaktních vědách je dobrým zvykem pojmy, o kterých se chceme bavit, nejprve definovat. Rád bych se tohoto zvyku přidržel a práci zahájil definicemi slov, které se vyskytují v samotném názvu této práce – informatika, myšlení a fundament. Bylo by ovšem mylné se domnívat, že tyto definice budou podobně přesné jako např. definice v matematice, půjde spíše o to ukázat, jakým způsobem tyto pojmy budeme chápat v rámci této práce, případně, jakými způsoby jsou v různých kontextech vnímány a proč se těchto pojetí nebudeme vždy držet.

1.1 Definice informatiky

Poprvé použil slovo informatika německý autor Karl Steinbuch v knize nazvané „Informatik: Automatische Informationsverarbeitung[1]" v roce 1957. Tento pojem byl následně přijat v západní Evropě; výjimku tvořila angličtina, do níž byl pojem překládán častěji jako „computer/computing science". Anglický pojem „informatics" zavedl až ruský informatik Alexander Mikhailov v roce 1967, který definoval informatiku takto:

- „Informatika je vědní disciplína, která zkoumá strukturu a vlastnosti (ne obsah) vědeckých informací, stejně jako pravidelnosti aktivit s informacemi, jejich teorii, historii, metodologii a organizaci."

Postupem času docházelo k posunu v chápání pojmu informatika. Dnešní chápání tohoto pojmu snad přiblíží následující dvě definice, které zároveň vyhovují pojetí informatiky v rámci této práce:

- Informatika je vědní disciplína, která se zabývá získáváním, zpracováním, uchováváním a prezentováním informací; přitom se snaží o nalezení optimálních postupů, kterými dosahujeme jednotlivých cílů.
- Informatika zahrnuje zpracování informací a inženýrství počítačových systémů. Informatika studuje strukturu, chování a interakce přirozených a umělých systémů, které uchovávají, zpracovávají a zprostředkovávají informace.[2]

---

1 Překlad: „Informatika: Automatické zpracování informací"

2 Volný překlad z http://en.wikipedia.org/wiki/Informatics

Často se setkáváme s pojmem „počítačová věda“ (angl. computer science), který je do jisté míry synonymem informatiky. Avšak zatímco informatika koresponduje s výše uvedenými definicemi, jádrem počítačové vědy jsou numerické výpočty a orientace na matematiku. Mylné zdání, že počítačová věda je hlavně o počítačích, zahání informatik Edsger Dijkstra svým citátem:

„Computer science is no more about computers than astronomy is about telescopes.[3]“

1.2 Definice fundamentu

Akademický slovník cizích slov [1] vysvětluje slovo fundament jako „základ, podstatu nebo jádro“. Cílem této práce je tedy najít to podstatné, na čem stojí informatika, základní principy, které prostupují všechno další informatické myšlení.

V rámci této práce rozdělíme fundamenty do dvou kategorií – statické a dynamické.

Statické fundamenty vyjadřují „pasivní materiály“, které jsou nezbytnými surovinami pro informatiku. Za statické fundamenty považujeme:

- informace
- čas
- prostor
- peníze

Dynamické fundamenty pak vyjadřují základní „aktivní činitele“, které využívají těchto zdrojů a přeměňují je, čímž vytvářejí to složitější v celé informatice. Za dynamické fundamenty považujeme:

- abstrakci
- iteraci a rekurzi
- metasyntaktické proměnné
- univerzalitu
- jednoduchost
- inspiraci

---

3 Překlad: „Informatika se nezabývá počítači o nic více než astronomie dalekohledy.“

Fundamenty jako množiny generátorů

V matematice se zavádí pojem algebry jako množiny a operací na této množině. Formálněji řečeno, algebra je dvojice (M, A), kde M je již zmiňovaná množina a $A=\{\alpha_i\colon M^{r_i} \to M\}_{i=1}^{n}$. Pak řekneme, že daná algebra má n operací, kde i-tá operace má aritu r_i. Množinu $G \subseteq A$ nazveme generátorem algebry (M, A), pokud existuje konečná posloupnost operací, které pokud budeme aplikovat na prvky množiny G a výsledky těchto operací přidávat do množiny G, získáme nakonec množinu A.

Můžeme vidět podobnost mezi generovanou algebrou a informatikou. Statické fundamenty můžeme přirovnat ke generátorům dané algebry, dynamické fundamenty pak můžeme chápat jako množinu operací nad touto algebrou.

Alternativní pojetí

Pokud se budete ptát např. Googlu[4] na fundamenty informatiky, většinou naleznete jiný druh odpovědi. Jeden příklad za všechny:

Na stavební fakultě ČVUT se vyučuje předmět s názvem „Základy informatiky[5]“. Hlavní důraz je kladen na praktické používání informačních technologií - ovládání operačních a informačních systémů a ovládání specializovaných programů (CAD, Matlab). Fundament je tedy často chápán jako nezbytné minimum znalostí a dovedností pro praktické využití.

1.3 Definice myšlení

Wikipedie [6], otevřená encyklopedie dostupná na internetu, definuje myšlení jako „mentální proces, který umožnuje lidem modelovat svět a využívat jej efektivně podle svých cílů, plánů a tužeb. Myšlení zahrnuje intelektuální manipulaci s informacemi, jako je utváření konceptů, řešení problémů, usuzování a rozhodování.“

Způsob, jakým o věcech přemýšlíme, je určující pro to, jaké budou výsledky našeho počínání. V následujících kapitolách ukážeme, jak je možné přemýšlet o jednotlivých fundamentech a jak nám tato pojetí pomohou v našich úvahách.

Je vhodné na tomto místě upozornit na dva extrémy, které mají své místo právě u obecných úvah o myšlení a mezi nimiž bychom měli najít rozumnou rovnováhu.

---

4 Fulltextový vyhledávač na internetu, dostupný na http://www.google.com

5 Informace získána z oficiálního webu příslušné katedry: http://mat.fsv.cvut.cz/BAKALARI/ZI/ZI.html

První extrém je *nemyslet*. To znamená nashromáždit výsledky uvažování někoho druhého, nebo i své, a tyto pak aplikovat na přicházející situace, což v mnohdy je správné, rychlé a účinné řešení. Jako příklad, kdy takový postup není na místě, uvedeme příklad:

Programátor začátečník si přečte v knize prof. Töpfera [2], že „quicksort je nejpoužívanější algoritmus vnitřího třídění … a mezi algoritmy této třídy je dokonce v průměrném případě nejrychlejší.“ Z toho si udělá závěr, že „quicksort je nejrychlejší“. Pokud bude mít takový programátor napsat pro kritickou aplikaci rychlé třídění (se zaručenou dobou odezvy), zvolí s použitím své poučky quicksort. Až následně se ukáže, že kvadratická složitost quicksortu v nejhorším případě způsobuje nemalé komplikace.

Druhý extrém je všechno se snažit *vymyslet znova*, nebo-li nemít žádné ověřené postupy, které by se daly aplikovat na známé situace. Pěkně je to vidět např. na metodikách programování. Pokud v určitém týmu není zavedena metodika, má každý programátor zpravidla jinou konvenci názvů, organizace souborů, někdy dokonce i architektury řešení. Takové projekty jsou pak těžko udržovatelné, prodlužuje se čas dokončení a rostou i náklady.

Je tedy potřeba najít rozumný způsob, jak o věcech v informatickém světě přemýšlet.

Pusťme se do toho.

2 Statické fundamenty

Statické fundamenty přestavují základní stavební materiál, ze kterého je vystavěn informatický svět. Jsou to suroviny (zdroje), které jsou omezujícím faktorem každého informatika.

Řecký myslitel Empedoklés z Akragantu (* ~493, + 433 p.n.l) hovořil o čtyřech *kořenech věcí*: nebi (vzduch), moři (voda), zemi (země), a slunci (oheň). Můžeme zde nalézt korespondenci mezi těmito kořeny a statickými fundamenty.

vzduch ► informace

voda ► čas

země ► prostor

oheň ► peníze

Informace

Informace jsou jako vzduch. Nejsou vidět, a přesto jsou všude, je jich mnoho a jsou pro člověka důležité. Informace jsou vůbec nejdůležitějším materiálem pro informatika, jsou vstupem každé jeho činnosti a následně také výstupem. Informace mají také svou hodnotu; podobně jako oheň nehoří bez vzduchu, „netečou“ peníze bez informací.

Čas

Čas můžeme přirovnat k vodě. Vždyť o čase se přece říká, že plyne jako voda. Čas je nedostatkové zboží a informatici se snaží jej co možná nejvíce ušetřit; určuje reálnou schopnost výpočetních postupů (algoritmů) být použit. Jsou totiž případy, kdy jsme schopni teoreticky dosáhnout výsledku, ovšem v čase, který nás přesahuje. Konečně voda uháší oheň, podobně jako čas snižuje hodnotu peněz (za předpokladu, že funguje inflační mechanizmus).

Prostor

Prostor je země. Máme stále více informací a potřebujeme stále více prostoru.

Peníze

Peníze jsou hybná síla lidského snažení, jsou ohněm, který nás zapaluje. Často určují, jakým směrem se bude ubírat naše rozhodování a jednání.

2.1 Informace

Podvědomou představu o tom, co je informace, má každý. Pěknou definici vyslovil filosof Gregory Bateson: „Informace je rozdíl, který znamená rozdíl.“ Znamená to, že za informaci považujeme pouze takové rozdíly, které pro nás mají nějaký smysl.

Informace versus data

V souvislosti s informacemi se objevuje také pojem *data*. Např. v počítačových časopisech můžeme často číst výzvy jako „Chraňte si svá data.“ Rozdíl mezi informací a daty je v tom, že data nemusí obsahovat žádnou informaci, nebo mohou, ale pro jejich zpracování to není rozhodující. Pokud např. zálohujeme disk, tedy data, zálohovací program kopíruje vše bit po bitu nezávisle na tom, zda je to nějaký důležitý dokument nebo náhodně vygenerovaný soubor jedniček a nul. Správně by tedy výše vzpomínaná výzva měla znít „Chraňte si své informace“, protože v nich se skrývá hodnota.

Jednotka informace

Za jednotku informace se považuje *bit* (z angl. „binary digit“, dvojková číslice). Claude E. Shannon, který je s teorií informace spojován, použil toto slovo poprvé v roce 1948, přičemž za vynálezce tohoto slova označoval Johna W. Tukeyho, který použil označení *bit* pro dvojkovou číslici o rok dříve v Bellových laboratořích.

Bit se používá jako jednotka informace, ale také jako jednotka popisující velikost dat, tedy místa, do kterého se informace ukládají. Rozdíl ukažme na příkladu: Soubor, který obsahuje tisíc písmen A, má velikost 1000 bytů[6], ale stejná informace lze popsat slovy „tisíc nul“, což zabere 72 bitů (v ASCII reprezentaci).

$$\underbrace{\overbrace{000\ldots000}^{1000\text{x}}}_{8000\,bit\mathring{u}} \qquad \underbrace{tis\acute{\imath}c\,nul}_{72\,bit\mathring{u}}$$

Počet bitů k uložení informace

Nechť atribut *A* patří do domény *Dom(A)*, která má velikost *n*. Pak počet bitů potřebných k uchování informace o hodnotě tohoto atributu určíme jako horní celou část dvojkového logaritmu velikosti jeho domény[7]:

$$b = \lceil \log_2 n \rceil$$

6 Byte = 8 bitů, jeden znak je reprezentován jedním bytem

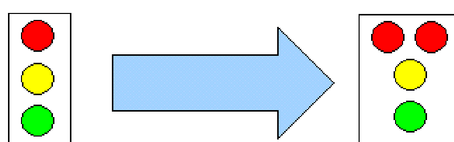
7 Nutno zdůraznit, že při výpočtu velikosti informace neuvažujeme horní celou část; velikost informace tedy může být neceločíselná.

Ilustrace: Zlomek bitu

Někdy máme sklon chápat bit jako nejmenší velikost, kterou může informace mít. Nenechme se však zmýlit - podobně jako atom, i bit můžeme dále dělit. Jak si ovšem představit např. ½ bitu? Uvažme následující situaci:

Na světelné křižovatce je semafor, který má tři světla – zelené, oranžové a červené. V každém okamžiku může svítit právě jedno (to sice neodpovídá realitě našich semaforů, protože ve skutečnosti může svítit červené a oranžové světlo současně, ale hodí se to pro tuto ilustraci).

Pro zachycení stavu semaforu potřebujeme $\log_2(3) \approx 1{,}58$ bitu, stačí nám tedy 2 bity pro uchování informace o tom, které světlo zrovna svítí. Tím ovšem nevyužijeme 0,42 bitu. Mějme dále informaci o tom, zda je pěkné počasí či nikoli.



Obrázek 1: Semafor se dvěmi červenými světly

Obohaťme tedy náš semafor o jedno červené světlo navíc. Z hlediska informace o tom, že svítí červené světlo si budou obě červená světla rovnocenná. Rovněž zachováme invariant, že v každém okamžiku svítí právě jedno světlo. Nakonec, pokud bude pěkné počasí, bude svítit levé červené světlo, a pokud nebude pěkné počasí, bude svítit pravé červené světlo. Pokud ovšem nebude červené světlo „na řadě“, nebude možné zjistit stav počasí. Informace o počasí tedy bude satisfakcí pro ty, kteří musí čekat.

Velikost bitu menší než jedna tedy odpovídá tomu, že danou informaci se dovíme pouze s určitou pravděpodobností.

Zkusme to ještě zobecnit...

Mějme atribut A, jehož doména $Dom(A)$ má velikost n a dále mějme f z intervalu (0,1). Číslo f vyjadřuje vnitřní fragmentaci, tedy rozdíl mezi velikostí ukládané informace a počtu bitů, které jsou potřeba pro její uložení:

$$b = \lceil \log_2 n \rceil = f + \log_2 n$$

Do zbylého místa se pokusíme uložit binární atribut X (potřebujeme tedy jeden bit, ale máme k dispozici pouze $f < 1$). Známe-li pravděpodobnosti, s jakými bude atribut A nabývat jednotlivých hodnot, použijeme $m = 2^b - n$ hodnot s největší pravděpodobností, které zdvojíme tak, že původní hodnota označuje $X = 0$ a duplikát označuje $X = 1$.

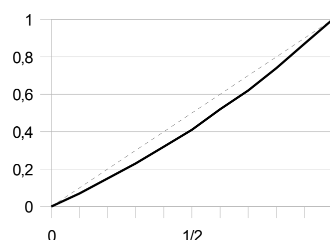
Nechť $a_1, \ldots, a_m$ jsou takto vybrané hodnoty a $p_1, \ldots, p_m$ jim odpovídající pravděpodobnosti. Pak pravděpodobnost, že se nám podaří uložit také hodnotu atributu X, určíme takto:

$$P = \sum_{i=1}^{m} p_i$$

V nejhorším případě mohou být pravděpodobnosti $p_1, \ldots, p_n$ stejné, pak

$$P = \sum_{i=1}^{m} p_i \geq \sum_{i=1}^{m} \frac{1}{n} = \frac{m}{n} = \frac{2^{f+\log_2 n} - n}{n} = \frac{2^f 2^{\log_2 n}}{n} - 1 = 2^f - 1$$

Vyneseme-li tuto nejhorší pravděpodobnost uložení do grafu (tlustá čára), zjistíme, že dosahuje (číselně) téměř velikosti daného bitového fragmentu (přerušovaná čára).



Graf 1: Pravděpodobnost uložení binárního atributu ve fragmentu bitu

Můžeme tedy učinit následující závěr: Velikost místa pro uložení informace menší než jeden bit přibližně odpovídá dolnímu odhadu pravděpodobnosti, že se nám podaří do tohoto místa uložit binární informaci.[8]

---

8 Platí za předpokladu, že známe pravděpodobnosti, se kterými daný atribut nabývá jednotlivých hodnot ze své domény.

2.2 Čas

V souvislosti s časem můžeme vidět dvě informaticky zajímavá témata: časovou složitost a napětí mezi strojovým časem a lidskou prací vyjádřenou dobou jejího trvání.

Časová složitost

Časová složitost výpočetního postupu vyjadřuje závislost mezi velikostí vstupních dat a času, který bude potřebný k jejich zpracování. Určujeme ji tak, že ve výpočtu najdeme nějakou elementární operaci, která je pro tento výpočet typická (např. porovnání pro třídění) a určíme počet těchto operací potřebných k výpočtu v závislosti na vstupních datech.

Příkladem výpočetního postupu může být nakreslení „stromu z hvězdiček" požadované výšky (viz obrázek). Pokud zvolíme jako elementární operaci nakreslení jedné hvězdičky, pak počet hvězdiček v závislosti na výšce stromu lze vyjádřit vztahem $n^2 - 2n + 2$. Navíc pro nás bude zajímavý pouze nejrychleji rostoucí člen, v tomto případě n^2. Řekneme, že nakreslení má kvadratickou složitost a označíme ji O(n^2).

```
   *
  ***
 *****
*******
   *
```

Obrázek 2: Strom z hvězdiček výšky 5

Notace velkého O

V předchozím textu jsme použili notaci velkého O, která se používá pro označování složitosti, aniž bychom ji nějak definovali. Nyní to napravíme formální definicí:

Mějme *f(x)* a *g(x)* funkce reálné proměnné. O výpočetním postupu, který vyžaduje *f(x)* elementárních operací řekneme, že má složitost O(g(x)), pokud

$$\exists M \in \mathbb{R}, M > 0, \exists x_0 \in \mathbb{R} : x > x_0 \Rightarrow |f(x)| \leq M |g(x)|$$

V našem příkladě je tato podmínka splněna např. pro $M = 1$ a $x_0 = 1$.

Třídy časové složitosti

Podle definice by mělo jistě smysl říci, že nějaký výpočet má složitost $O(4x^2 + 2)$. Takový výpočet má ovšem také složitost $O(x^2)$, což je mnohem přehlednější. Snažíme se tedy používat jednodušší zápis.

Stejně tak můžeme o výpočtu se složitostí $O(x^2)$ říci, že má složitost $O(x^5)$ a podle

definice to bude také pravda. Tím ovšem ztratíme cennou informaci o skutečné složitosti. Snažíme se tedy uvádět co nejnižší třídu (měřeno rychlostí asymptotického růstu).

Následující tabulka poskytuje přehled některých tříd složitostí a příklady výpočtů, které mají tuto časovou složitost.

Značení	**Název**	**Příklad**
O(1)	Konstantní	Určení parity čísla
O(log n)	Logaritmická	Hledání prvku v setříděném seznamu půlením intervalu
O(n)	Lineární	Hledání prvku v nesetříděném seznamu
O(n log n)	Lineárně logaritmická	Třídění seznamu heapsortem Rychlá Fourierova transformace
$O(n^2)$	Kvadratická	Bublinkové třídění Diskrétní Fourierova transformace
O(n!)	Faktoriální	NP-úplné úlohy řešené hrubou silou

Tabulka 1: Přehled časových složitostí

Multiplikativní konstanta

Nutno poznamenat, že dva algoritmy, které mají stejnou třídu složitosti, nemusí běžet stejnou dobu. Každý z nich může mít jinou *multiplikativní konstantu*, což je infimum z čísel M uvažovaných v definici symbolu *O*.

Je-li *f(x)* počet elementárních operací algoritmu a *O(g(x))* jeho složitost, pak se multiplikativní konstanta *c* často určí takto:

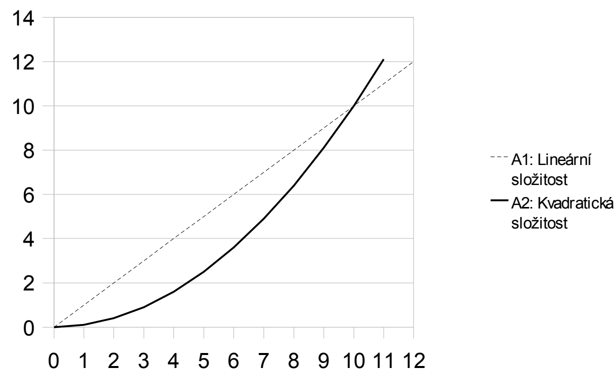
$$c = \lim_{x \to \infty} \frac{f(x)}{g(x)}$$

Dále musíme mít na paměti, že není nutně pravda, že algoritmus, který patří do nižší třídy složitosti, je vždy rychlejší než algoritmus ve třídě vyšší. To platí pro situace, kdy velikost zpracovávaných dat je malá.

Uvažme dva algoritmy A1 a A2, které řeší stejnou úlohu, a jim odpovídající počty elementárních operací

$$f_1(x) = x$$
$$f_2(x) = \frac{1}{10} x^2$$

Na následujícím grafu můžeme pozorovat, že algoritmus A2 je rychlejší, dokud velikost vstupních dat nepřesáhne hodnotu 10.



Graf 2: Ilustrace složitosti na malých datech

Čas: stroj versus člověk

V dnešní době umí počítače počítat mnohem rychleji než člověk. Rychlejší výpočetní postupy zpravidla vyžadují složitější konstrukci výpočtu (programu). Vždy je nutné zvážit, do jaké míry potřebujeme efektivní program a jak se to projeví na úsilí jej vytvořit. Toto úsilí roste nejen se snižující se třídou složitosti algoritmu, ale také s klesající multiplikativní konstantou.

Přijměme předpoklad, že naprogramovat bublinkové třídění je dvakrát jednodušší než naprogramovat třídění haldou. Pokud chceme setřídit jednou 10 000 položek, zvolíme pravděpodobně bublinkové třídění, protože čas potřebný ke konstrukci programu a jeho běhu je menší. Uvedené hodnoty doby konstrukce a běhu jsou pouze odhadem a mají sloužit pouze pro ilustraci.

$$\overbrace{\underbrace{10\,minut}_{Konstrukce}+\underbrace{10\,sekund}_{Běh}}^{Bublinkové\,třídění}<\overbrace{\underbrace{20\,minut}_{Konstrukce}+\underbrace{2\,sekundy}_{Běh}}^{Třídění\,haldou}$$

Pokud ovšem budeme tento výpočet provádět denně po dobu jednoho roku, zjistíme, že se nám už vyplatí implementovat třídění haldou.

$$71\,minut=\overbrace{\underbrace{10\,minut}_{Konstrukce}+\underbrace{365\cdot 10\,sekund}_{Běh}}^{Bublinkové\,třídění}>\overbrace{\underbrace{20\,minut}_{Konstrukce}+\underbrace{365\cdot 2\,sekundy}_{Běh}}^{Třídění\,haldou}=32\,minut$$

2.3 Prostor

Dostatek prostoru je důležitým faktorem nejen pro informatiku. Můžeme sledovat snahu vytvářet stále větší úložnou kapacitu. Zároveň stále postupující miniaturizace naznačuje snahu šetřit i skutečným prostorem v serverovnách a datových skladech. Zároveň přibývá informací, které je potřeba ukládat a také potřeba tyto data umět rychle načítat a měnit (příkladem mohou být on-line transakce).

Prostor tedy můžeme chápat dvojím způsobem, jako velikost pamětí a skutečný prostor.

Prostorová složitost

V souvislosti s výpočetními postupy (algoritmy) zkoumáme, jaké budou nároky na paměť. Tyto nároky jsou dvojí: paměť potřebná pro uložení programového kódu a paměť potřebnou pro zpracování dat. Paměť potřebná pro uložení programového kódu je zpravidla zanedbatelná vzhledem k velikosti dat. Závislost velikosti vstupních dat na velikosti paměti potřebné k jejich zpracování nazýváme prostorová (nebo paměťová) složitost. Pokud program nevyžaduje více paměti než zabírají zpracovávaná data, říkáme, že pracuje *in-situ* (z lat., čti sitú, na místě).

Prostorovou složitost vyjadřujeme podobně jako časovou složitost notací velkého O v závislosti na vstupních datech.

Často platí, že pokud má algoritmus k dispozici málo prostoru, počítá déle, protože některé hodnoty je nucen zapomenout a v případě potřeby je počítat znova. Naopak, pokud má k dispozici dostatek místa, může jej využít a dokončit výpočet dříve.

Dobrým příkladem této závislosti časové a prostorové složitosti je výpočet Fibonacciho čísel, tj. posloupnosti $\{Fib(i)\}_{i=0}^{\infty}$, pro kterou platí:

$$
\begin{aligned}
&Fib(0)=0\\
&Fib(1)=1\\
&Fib(n)=Fib(n-1)+Fib(n-2)
\end{aligned}
$$

n-té Fibonacciho číslo můžeme hledat rekurzivně bez toho, že bychom si pamatovali již vypočítané hodnoty, čímž ušetříme paměť (žádnou nebudeme potřebovat), ale budeme déle počítat. Alternativou je ukládat si jednou vypočítané hodnoty, což sice zabere nějaké místo, ale ušetří čas. Srovnání nabízí následující tabulka:

Výpočet fib_n	Bez ukládání mezivýsledků	S ukládáním mezivýsledků
Prostorová složitost	žádná	O(n)
Časová složitost (měříme v počtu součtů, které musíme vykonat)	$O(a^n)$ (označíme-li počet součtů potřebných k výpočtu Fib(n) symbolem $\bar{n}$, pak $\bar{0}=\bar{1}=0$ $\bar{n}=\overline{n-1}+\overline{n-2}+1$ $\bar{n}$ tedy roste přibližně stejně rychle jako Fib(n))	n – 1 (pro každé číslo kromě jedničky sečteme jednou dvě předchozí)

Tabulka 2: Prostorová a časová složitost různých výpočtů Fibonacciho čísel

Komprese

Ne vždy se podaří uložit informace do místa, které svou velikostí odpovídá velikosti dané informace. Například, pokud chceme ukládat česká slova o délce 5, zvolíme pravděpodobně pět bytů paměti, což odpovídá 2^{40} možných hodnot, což je číslo, které přesahuje počet všech slov ve všech jazycích na světě.

Komprese je proces hledání úspornější reprezentace nějaké informace. V případě ztrátové komprese se navíc snažíme vybrat pouze ty informace, které jsou pro nás nejvíce zajímavé (např. slyšitelné frekvence při kompresi zvuku).

V souvislosti s kompresí vyslovil Hillis [5] tuto definici informačního obsahu: „Obsah informace v posloupnosti bitů je roven délce nejmenšího počítačového programu, který je schopen tyto bity vygenerovat.“

Přehled fyzických nosičů informací

Zastavme se na chvíli u toho, kolik místa informace zabírají. Dosud jsme chápali prostor jako počet bitů potřebných k uchování dané informace. Nyní se zkusme podívat na věc z hlediska skutečného prostoru. V následující tabulce porovnáme některé datové nosiče, bude nás přitom zajímat jejich rozměr, hmotnost a samozřejmě také kapacita. Pro zajímavost určíme:

- kolik váží jeden TB dat
- kolik dat se vejde do krabice od banánů; pro jednoduchost budeme předpokládat, že nosiče zaplní všechen prostor krabice. Vnitřní rozměr banánové krabice je asi 52x37x24 cm, tedy objem je přibližně 46 000 cm^3.

Nosič	Disketa 3,5"	CD ROM[9]	Flash Disk[10]
Kapacita (MB)	1,44	700	1000
Hmotnost (g)	19	16 / 59 /85	3
Rozměr (mm)	90x93x3	120x120x1 / 125x143x5 / 125 x 143x10	40x15x2
Objem (cm^3)	25	14 / 89 / 179	1
Nosičů potřebných na 1 GB (~ 2^{20} MB)	728 178	1 497	1049
Hmotnost 1 TB (kg)	13 835	24 / 88 / 127	3
GB dat v jedné banánové krabici	2,65	2 300 / 362 / 180	46 000

Tabulka 3: Přehled vybraných datových nosičů

Uvedená tabulka má posloužit jako malá ilustrace toho, že rozdíly mezi uvedenými nosiči jsou značné. Zatímco 1 TB na disketách váží téměř 14 tun, na flash discích je možné jej odnést v batohu. Podobně, banánová krabice plná disket má stejnou kapacitu jako 4 disky CD. Nakonec, ve stejné krabici je možné nashromáždit 46 TB dat, pokud použijeme flash disky.

9 Uvedené hodnoty odpovídají po řadě následujícím variantám: (1) bez obalu, (2) v úzkém obalu, (3) v normálním obalu

10 Konkrétně měřeno na flash disku A-DATA 1GB MyFlash PD15 FlashDrive USB2.0

2.4 Peníze

O penězích se říká, že hýbou světem. Peníze motivují člověka od přírody líného, aby pracoval, nebo dokáží přimět člověka k rozhodnutím, které jsou proti jeho přesvědčení.

Celá ekonomie se v podstatě točí kolem peněz a chceme-li dělat informatiku ekonomicky, nebo její poznatky využít hospodárně, nesmíme při našem přemýšlení zapomínat na peníze. Na půdě Vysoké školy ekonomické v Praze mají jednu oblíbenou frázi, že totiž „o peníze jde až na prvním místě."

Náklady (obětované) příležitosti

Pro následující úvahy zavedeme nyní pojem náklady (obětované) příležitosti, v angl. literatuře známý pod označením *opportunity costs*.

Náklady (obětované) příležitosti odpovídají hodnotě nejhodnotnější činnosti, které se musíme vzdát ve prospěch jiné činnosti.

Pro lepší pochopení uveďme příklad: Představme si, že se rozhodujeme, zda raději zůstaneme déle v práci, nebo půjdeme do divadla. Pokud zvolíme divadlo, náklady, které budou s tímto rozhodnutím spojené, jsou samozřejmě cena vstupenky (takové náklady jsou označovány jako *explicitní*), ale dalším nákladem bude ušlá mzda, kterou jsem musel tomuto rozhodnutí obětovat (takové náklady jsou označovány jako *implicitní*).

Vyzbrojeni znalostí principu nákladů obětované příležitosti se můžeme podívat, jakou roli hrají peníze při rozhodování v informatice.

Juniorská práce

Platí obecné pravidlo, že schopní lidé jsou drazí. Platí to obecně, my si to můžeme ukázat na příkladu programátorů. Jsme-li jako manažer IT projektu postaveni před rozhodnutí, zda danou část aplikace bude programovat junior nebo senior, musíme zvážit náklady obou variant. Řekněme dopředu, že neexistuje obecné pravidlo a záleží na konkrétní situaci. Mějme následující případ:

Danou úlohu bude junior řešit 80 hodin, následně bude provedena seniorem revize v rozsahu 5 hodin, a dalších 15 hodin bude junior zapracovávat připomínky, které vznikly při revizi. Pokud by danou úlohu řešit senior, stráví nad řešením 40 hodin a žádná revize nebude potřeba.

Předpokládejme dále, že cena jedné hodiny juniora je 150 Kč a seniora 500 Kč. Pokud bychom měli k dispozici tyto údaje, můžeme vypočítat hodnotu obou variant:

$$cena_{junior+revize} = 95\,hodin \times 150\,Kč/hodinu + 5\,hodin \times 500\,Kč/hodinu = 16\,750\,Kč$$
$$cena_{senior} = 40\,hodin \times 500\,Kč/hodinu = 20\,000\,Kč$$

Pokud tedy vyjádříme explicitní náklady obou variant, vychází nám, že bude lepší nechat pracovat juniora pod dohledem seniora. Předpokládejme však dále, že výsledek této činnosti bude po dokončení životaschopný a za každý den svého provozu vytvoří hodnotu ve výši 500 Kč. Začnou-li programátoři v obou variantách pracovat v pondělí a předpokládáme-li 40 hodinový pracovní týden, bude průběh prací odpovídat následujícímu schématu:

Po	Út	St	Čt	Pá	So	Ne	Po	Út	St	Čt	Pá	So	Ne	Po	Út	St	Čt	Pá	So	Ne
Práce juniora							**Práce juniora**							**Revize + oprava**		**S**	**T**	**A**	**R**	**T**
Práce seniora					**S**	**T**	**A**	**R**	**T**											

Z uvedeného diagramu je vidět, že dílo bude hotové o 11 a půl dne dříve v případě, že bude zhotoveno seniorem, a snadno dopočítáme, že za tuto dobu můžeme provozem získat 5 750 Kč. Tento zisk odpovídá nákladům obětované příležitosti varianty s juniorem. Celkové náklady obou variant pak vycházejí následovně:

$$cena_{junior+revize} = mzdy + náklady\ obětované\ příležitosti = 16\,750 + 5\,750 = 22\,500\,Kč$$
$$cena_{senior} = mzdy = 20\,000\,Kč$$

Je vidět, že v tomto případě přihlédnutí s nákladům obětované příležitosti obrátilo situaci a dříve nevýhodná alternativa se nyní jeví jako ta výhodnější.

Další příklady z praxe

Na tomto místě zmíníme ještě další dvě oblasti, ve kterých se vyplatí přemýšlet a zejména počítat – při volbě programovacího jazyka a při rozhodování, zda optimalizovat. V těchto případech musíme odpovědět na následující otázky:

- Je levnější rychleji napsat programový kód za cenu horšího výkonu?
- Přinese optimalizace alespoň takový užitek, aby se zaplatila?

Pokud o těchto věcech přemýšlíme z pohledu peněz, pak nám bude jasnější, proč se tolik prosazuje programování v Javě nebo .NETu oproti např. C++, nebo proč se vlastně neoptimalizuje, není-li to nezbytně potřeba.

3 Dynamické fundamenty

3.1 Abstrakce

Abstrakce je proces zobecnění (nebo výsledek tohoto procesu), při kterém omezujeme množství informací nějaké myšlenky nebo jevu, obvykle proto, abychom zachovali pouze informace, které jsou pro nás v danou chvíli podstatné. Abstrakce je proces skrývání detailů, při kterém vynikají pouze důležité rysy.

Nacházíme mnoho aplikací principu abstrakce také v informatice. Podívejme se nejprve na příklady, jak abstrakce přímo pracuje s některými statickými fundamenty.

Abstrakce času

Pokud počítáme s časovými hodnotami, často nepotřebujeme znát přesnou hodnotu, nebo ani nejsme schopni ji určit, případně takové určení pro nás nemá smysl. Příkladem může být jeden tik procesoru, jehož délka je závislá taktovací frekvenci, nebo doba čtení určitého množství dat z disku. Pro další uvažování má pak smysl říkal, že operace si vyžádá tři čtení z disku, místo abychom určovali přesný čas.

Abstrakce prostoru

Podobně můžeme abstrahovat prostor. Pokud např. rozdělíme disk na bloky o velikosti řádově 512 – 4096 B, můžeme pak uvažovat o těchto blocích stejně, ať mají jakoukoli velikost. Odhlížíme tedy od detailů (velikost bloku) a můžeme se soustředit pouze na podstatné rysy (to, že jsou jednotlivé bajty seskupeny do bloků stejné velikosti).

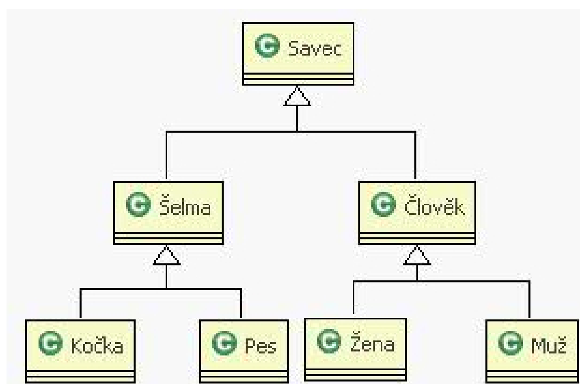
Abstrakce v OOP

V terminologii filozofů je abstrakce chápána jako myšlenkový proces, při kterém jsou ideje oddělovány od objektů. V objektově orientovaném programování (OOP) jsou naopak objekty středem zájmu abstrakce, zatímco ideje (jsou-li nějaké) ne.

Abstrakci můžeme v OOP vysledovat hned dvojí. Jednak abstrakci v rámci hierarchie dědičnosti, jednak koncept OOP jako výsledek evoluce programovacích jazyků, kde během této evoluce dochází k postupné abstrakci stroje, pro který se programuje.

Na obrázku 3 vidíme hierarchii dědičnosti, v jejímž kořenu je třída Savec, která je nejobecnější (na nejvyšším stupni abstrakce). Její potomci Šelma a Člověk přidávají k vlastnostem savce další specifické rysy, dolní řada tříd pak představuje již velmi konkrétní třídy. Abstrakce tedy postupuje odspoda nahoru (ve směru šipek). Diagram

také připomíná základoškolskou nad/podřazenost slov.



Obrázek 3: Ukázka hierarchie dědičnosti tříd

Postup abstrakce v programovacích jazycích popisuje Eckel ve své knize [3]:

„Každý programovací jazyk představuje určitou abstrakci. Lze říci, že složitost problému, který jsme schopni řešit, je úzce spojena s druhem a kvalitou abstrakce. Nejnižší stupeň abstrakce strojového jazyka počítače představuje jazyk symbolických adres (assembler). Mnoho tzv. \`imperativních\` jazyků, které následovaly ve vývoji (např. Fortran, BASIC či jazyk C), byly abstrakcemi jazyka symbolických adres. Jakkoli tyto jazyky přinesly významná zlepšení oproti assembleru, míra jejich abstrakce stále vyžaduje, aby programátor uvažoval spíše v rovině počítačových struktur než v rovině struktur vlastního řešení problému.“

Dále Eckel vysvětluje, že v těchto jazycích je třeba transformovat problém z reálného světa do pojmů, kterým rozumí stroj, což může být obtížné. Pak pokračuje:

„Objektově orientovaný přístup je o krok dál, neboť poskytuje programátorovi nástroje pro reprezentaci prvků z prostoru problémů ... Základní myšlenka spočívá v tom, že program lze přizpůsobovat popisu problému přidáváním nových typů objektů, takže když potom čtete kód popisující řešení, čtete zároveň slova popisující samotný problém. Takovýto druh abstrakce je mnohem pružnější a účinnější, než všechny abstrakce předcházející.“

Funkční abstrakce

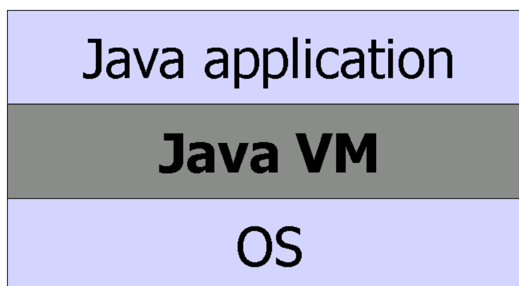
Hillis [5] popisuje pojem funkční abstrakce takto: „Jakmile vymyslíme, jak realizovat požadovanou funkci, můžeme mechanizmus uložit do \`černé skříňky\` nebo do \`stavebního bloku\` a přestat o něm přemýšlet. Funkci představovanou stavebním blokem můžeme využívat znovu a znovu, bez jakékoli zmínky o detailech toho, co je uvnitř. Metoda funkční abstrakce je základem návrhu počítačů.“ Podobně fungují také

funkce v programování – je definováno, jaké má vstupní parametry a co vrátí. Jak je implementována, nás jako uživatele nemusí zajímat.

Virtuální stroje

Virtualizace je téma dneška. Musíme však připomenout, že počátky virtualizace sahají až do 60. let minulého století k počítači IMB360. Při konstrukci operačního systému CP-67[11] ve snaze zjednodušit víceuživatelský systém vznikla myšlenka, že by operační systém vytvořil virtuální stroj pro každého uživatele a následně by na každém takovém stroji běžel již jednouživatelský systém.

Příkladem virtuálního stroje z dnešní doby může být Java Virtual Machine (JVM). Pro zajištění přenositelnosti se vkládá mezi operační systém a samotnou aplikaci ještě Javovský virtuální stroj[12], který má stejné vlastnosti a chování na všech platformách. Virtuální stroj je tedy abstrakcí operačního systému.



Obrázek 4: Abstrakce OS pomocí JVM

11 Tento OS byl později přejmenován na VM/370.

12 JVM je program, který umí interpretovat instrukce Java aplikace.

3.2 Iterace a rekurze

Iterace a rekurze jsou důležitou součástí myšlenkových i výpočetních postupů. Mají některé společné rysy a v některých se odlišují. Společnou charakteristikou může být na první pohled zřejmé opakování určitého postupu. Podívejme se nyní na charakteristiku iterace a rekurze podrobněji.

Iterace

Iterace spočívá v tom, že opakujeme určitý postup, přičemž výstup jednoho opakování, které se také nazývá iterace, slouží jako vstup opakování dalšího. Takovým postupem tedy opracováváme původní vstup až do doby, kdy jsme s výsledkem spokojeni. Iterativní přístup nám přináší jisté zjednodušení oproti variantě, že bychom chtěli výsledku dosáhnout přímo.

Iterace v Unified Process

Zásada iterace je jeden ze tří základních axiomů metodiky Unified Process[13] používané pro tvorbu softwarových systémů. Zapojení iterace do této metodiky popisují Arlow a Neustadt [4]:

> „Metodika UP je konečně iterativní a přírůstková. Iterativní aspekt znamená rozklad projektu na menší podprojekty (iterace), které systému dodávají funkce dávkově ... Jinými slovy to znamená, že tvoříme software v procesu postupných upřesňování našeho konečného záměru ... Historie ukazuje, že člověku se obecně vzato řeší lépe menší problémy než větší.“

Rekurze

Rekurze spočívá v tom, že aplikujeme určitý postup v rámci jeho samotného. V matematice to např. znamená, že definujeme funkci pomocí výrazu obsahující sebe sama. Nejznámějším příkladem může být rekurzivní definice faktoriálu:

$$0! = 1$$
$$n! = n \times (n-1)!$$

Pro rekurzi je důležité uvést, kdy má skončit. Tato *ukončovací podmínka* je v uvedené definici přítomna na prvním řádku.

Nutno poznamenat, že faktoriál není nutné definovat rekurzivně, tedy že rekurzivita není nějaká vlastnost, kterou funkce má, nebo ne. Pro srovnání uveďme ekvivalentní

13 Čtenář se také může setkat s pojmem Rational Unified Process (RUP), který je komerční variantou UP

definici bez rekurze:

$$n!=\prod_{i=1}^{n} i$$

Na tuto definici se můžeme dívat také jako na iteraci. V n opakováních se provádí násobení pořadovým číslem tohoto opakování, vstupem první iterace je jednička a výstupem poslední je výsledek.

Rekurze vs. důkaz matematickou indukcí

Připomeňme si, jak vypadá důkaz matematickou indukcí. Chceme-li tímto postupem dokázat tvrzení *P(x)*, použijeme následující schéma:

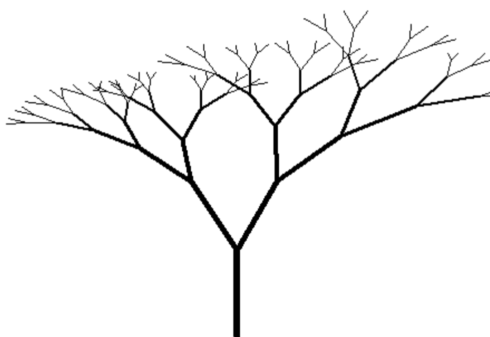
$$\exists n_0 \in \mathbb{N} [P(n_0) \wedge (\forall n \in \mathbb{N}, n > n_0 : P(n-1) \Rightarrow P(n))] \Rightarrow \forall n \geq n_0 : P(n)$$

Pokud platí tvrzení *P(x)* pro nějaké počáteční n_0 a umíme ukázat, že platnost tohoto tvrzení pro předchozí hodnotu implikuje platnost pro další hodnotu, máme dokázáno tvrzení *P(x)* pro všechny hodnoty větší nebo rovny n_0. Můžeme vidět podobnost s rekurzí v tom smyslu, že počáteční podmínka matematické indukce odpovídá ukončovací podmínce rekurze a indukční krok odpovídá rekurzivnímu kroku.

Rekurze v přírodě

V duchu kapitoly 3.6, která pojednává o inspiraci jako důležitém principu, který se promítá do informatiky, nyní ukážeme, že také rekurze je princip, který není umělou konstrukcí matematiků nebo informatiků, ale že pochází z dílny „matky přírody“.

Pro ilustraci použijeme příklad stromu. Na strom se můžeme dívat jako na rekurzivně definovanou strukturu. Definujme tedy strom takto: Strom je kmen, ze kterého vyrůstají dva menší stromy. Velikost těchto menších stromů a směr jejich růstu vzhledem ke kmeni určuje výsledný tvar stromu.



Obrázek 5: Obrázek vygenerovaný z rekurzivní definice stromu

V příloze A této práce je možné najít program v jazyce R, který implementuje popsanou rekurzivní definici takového stromu. Výstupem tohoto programu je výše uvedený obrázek.

Efektivita rekurzivních algoritmů

Rekurzivní algoritmy mají tu výhodu, že úlohy, které mají jednoduchý rekurzivní popis, lze pomocí rekurzivních algoritmů jednoduše (a přímočaře) naprogramovat. Ukazuje se však, že bezmyšlenkovité použití rekurze často vede k snížení efektivity takového algoritmu, případně k praktické nepoužitelnosti. Největší problém nastává v okamžiku, kdy se rekurzivní definice odkazuje sama na sebe více než jednou, jako je tomu třeba v případě výpočtu Fibonacciho čísel. Jak jsme již naznačili v kapitole 2.3, má rekurzivní algoritmus exponenciální časovou složitost.

Efektivita iterací

Opět si můžeme povšimnout souvislosti mezi rekurzí a iterací a dívat se na výpočet Fibonacciho čísel iterativně. Vstupem první iterace budou první dvě Fibonacciho čísla, která jsou součástí definice. Každá iterace dostane dvě po sobě jdoucí Fibbonaciho čísla a vypočítá následující. Tím dosáhneme lineární časové a konstantní paměťové složitosti, tedy ještě lepší výsledek, než v případě rekurzivního výpočtu s ukládáním mezivýsledků uvedeném v kapitole 2.3. Následující UML diagram znázorňuje popsaný postup:

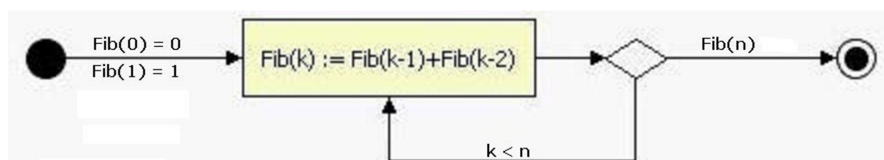


Diagram 1: Iterativní výpočet n-tého Fibonacciho čísla

Je tedy důležité si uvědomit, že rekurze je sice mocný nástroj, ale musí se užívat opatrně, abychom mohli plně využít její síly. Více příkladů problémů s efektivitou rekurzivních algoritmů je možno najít v knize [2], kapitola „Efektivita rekurzivních algoritmů“.

3.3 Metasyntaktické proměnné

„Metasyntaktická proměnná“ je označení zažitého názvu pro nespecifikovanou entitu, jejíž přesná povaha záleží na kontextu. Metasyntaktické proměnné používáme, pokud potřebujeme entitu pojmenovat, ale na přesném jménu nám nezáleží. Chceme-li např. ukázat použití různých modifikátorů viditelnosti v signatuře metody, není důležité, jak se daná metoda jmenuje, ale jaké je umístění toho jména v rámci signatury:

```
void foo()                       /* Package visibility        */
private void foo(int bar)        /* Only this class           */
protected void foo(long bar)     /* This class and subclasses */
public void foo(char bar)        /* Everyone                  */
```

Obrázek 6: Příklad použití metasyntaktické proměnné "foo"

Ještě než se podíváme na konkrétní příklady metasyntaktických proměnných podrobněji, uvědomme si, že to není záležitost pouze informatická. V matematice se např. používají písmena *a, b, c* pro konstanty, *f, g, h* pro funkce, *i, j, k, l* pro indexy, *m, n* pro počet, *p, q, r, s* pro přímky nebo jako parametry, *u, v, w* pro vektory a nakonec *x, y, z* pro proměnné. Na následující ilustraci ze světa matematiky může čtenář porovnat, jakým dojmem působí výraz, který se uvedenými konvencemi řídí, a výraz, který ne.

$$\sum_{i=1}^{n} \frac{f(x_i)}{|\vec{u}_i|} \quad \times \quad \sum_{f=1}^{i} \frac{x(u_f)}{|\vec{x}_f|}$$

Foo

Foo je nejčastěji používaná metasyntaktická proměnná, která má univerzální použití. Může označovat hodnoty, funkce, objekty, soubory a další. Někdy se vznik tohoto slova vysvětluje tak, že vzniklo z prvních písmen fráze „File Or Object“ (tj. soubor nebo objekt). Jiná teorie hledá vznik tohoto slova v době druhé světové války. Více informací o etymologii tohoto slova lze nalézt v RFC 3092[14].

Foo je zástupcem skupiny metasyntaktických proměnných, které souhrnně označujeme jako „nesmyslná slova“. Tato slova se mohou vyskytovat jednotlivě, nebo mohou být seskupena do posloupností. Konkrétní posloupnost se zpravidla vztahuje k určité skupině lidí, kteří ji vymysleli a používají ji. Následující tabulka uvádí další příklady z této skupiny:

---

14 Dostupné např. na http://rfc.net/rfc3092.html (v anglickém jazyce)

bar	Druhá nejčastěji používaná metasyntaktická proměnná. Často se užívá ve spojení s foo, např. foobar.
baz, qux, quux, ...	Pokračování řady foo, bar ...
foo, bar, bletch	Používáno na Univerzitě Waterloo.
foo, bar, fum	Používáno v XEROX PARC.[15]
oogle, foogle, boogle	Údajně běžné v Anglii. Obě série mohou pokračovat s dalšími počátečními souhláskami.
zork, gork, bork	
shme	Používáno na Univerzitě Berkeley.
zxc	Používáno na Univerzitě Cambridge.

Tabulka 4: Přehled "nesmyslných" metasyntaktických proměnných

Jehla v kupce sena

Známé přirovnání o hledání jehly v kupce sena existuje i anglickém jazyce. To má za následek, že anglická slova pro jehlu (needle) a kupku sena (haystack) jsou také používána jako metasyntaktické proměnné. Použití těchto slov je nejčastěji vidět v situaci, když popisujeme funkci, která něco prohledává. Pak needle označuje hledaný předmět a haystack místo, kde budeme hledat. Tento princip je možné vidět např. v dokumentaci jazyka PHP (viz např. http://cz.php.net/manual/cs/function.strpos.php).

Další slova, která pocházejí z přirozeného jazyka a zároveň hrají roli metasyntaktické proměnné, ukazuje následující tabulka:

spam, ham, eggs	Základní metasyntaktické proměnné používané programátory jazyka Python. Odkazují se na scénku skupiny Monty Python[16], ve které se pár v restauraci snaží vybrat z jídelníčku jídlo neobsahující „spam“[17]
pippo, pluto, paperino	Používáno v Itálii. „Pippo“ je italské jméno pro Goofyho (postava z kresleného filmu Walta Disneyho). „Paperino“ označení pro kačera Donalda.
aap, noot, mies	Používáno v Holandsku. První slova, která se děti ve škole učí číst.

Tabulka 5: Přehled metasyntaktických proměnných s původním významem

Číselné metasyntaktické proměnné

Nastávají situace, kdy někde potřebujeme uvést číslo, jehož hodnota pro nás není důležitá. Použijeme tedy „zástupné číslo“, metasyntaktickou proměnnou, kdy číslo

15 Výzkumné centrum společnosti XEROX v Palo Alto, Kalifornie, USA.

16 Monty Python je britská komediální skupina založená v roce 1969.

17 Úplný text scénky na http://www.detritus.org/spam/skit.html

zastupuje číslo. Příklady proměnných z této skupiny jsou v následující tabulce, uspořádané vzestupně (tedy ne podle frekvence užití).

17	Na MIT[18] označováno jako „nejvíce náhodné číslo“.
23	Užívá se často jako příklad celočíselné hodnoty, zejména ve snaze vyhnout se případné konotaci spojené s číslem 42.
37	Číslo 37 bývá často vybíráno lidmi, které požádáte o číslo mezi 10 a 50, ve kterém nejsou dvě stejné číslice. Proto často reprezentuje číslo, o kterém chceme tvrdit, že je náhodné.

42	Číslo 42 se často používá pro inicializaci proměnných nebo jako návratová hodnota. Je vůbec nejdůležitějším příkladem číselné metasyntaktické proměnné.
	Toto číslo bylo zpopularizováno v knize Douglase Adamse „Stopařův průvodce po galaxii“, ve které má počítač nazvaný „Hlubina myšlení“, největší počítač všech dob, určit odpověď na „otázku života, vesmíru a vůbec všeho.“ Po dlouhém počítání odpovídá počítač: „42“.
	Jako zajímavost můžeme uvést, že SWI-PROLOG odpovídá na otázku X (pouze volná proměnná) následujícím výpisem:
	`?- X.`
	`% ... 1,000,000 ............ 10,000,000 years later`
	`% >> 42 << (last release gives the question)`

47	Někdy se používá namísto čísla 42, převážně členy „47 society“. Tato skupina tvrdí, že číslo 47 je nejčastější číslo ve vesmíru.[19]
69	69 se často používá jako ukázkové číslo. Jeho popularita je zajištěna, neboť vyvolává asociaci s jednou sexuální polohou. Nutno také připomenout, že na mnoha kalkulačkách je 69 největší číslo, z kterého lze spočítat faktoriál.
1701	Číslo 1701 je používáno programátory, kteří jsou fanoušci televizního seriálu Star Trek, neboť vesmírné plavidlo v tomto seriálu mělo označení NCC-1701.
CAFE BABE	Číslo 3 405 691 582 má v hexadecimální soustavě zápis CAFEBABE. Používá se pro vyplnění 64bitové hodnoty, pokud chceme označit, že je chybná nebo se nevyužitá. Vyskytuje se také na začátku každé zkompilované třídy jazyka Java.
DEAD BEEF	Číslo 3 735 928 559 má v hexadecimální soustavě zápis DEADBEEF. Použití je stejné jako v předchozím případě.
DEAD C0DE	Číslo 3 735 929 054 má v hexadecimální soustavě zápis DEADC0DE. Toto číslo je velmi oblíbené u hackerů. Číslo umístěné ve zdrojovém kódu indikuje, že se tento kód již nepoužívá nebo že nefunguje.

Tabulka 6: Přehled číselných metasyntaktických proměnných

18 Massachusetts Institute of Technology

19 Více informací o „47 society“ lze nalézt na http://www.47.net/47society/. Autor se od názoru této skupiny distancuje.

Alice a Bob

Široké uplatnění metasyntaktických proměnných nacházíme v kryptografii a počítačové bezpečnosti. Existuje velké množství jmen používaných pro označení jednotlivých účastníků. Tato jména používáme při popisu protokolů nebo nastalých situací.

V situaci, kdy jeden účastník chce druhému předat zprávu, označujeme odesilatele Alice a příjemce Bob. Je-li v komunikaci zahrnuto více než dvě strany, pak se někdy používají jména Carol, Dave, Ellen, Frank... pro další účastníky. Důležité je, že počáteční písmena těchto jmen jsou v abecedním pořadí.

Vedle těchto „hodných“ účastníků se vyskytují útočníci. Pasivního útočníka, který pouze odposlouchává zprávy, které posílá Alice Bobovi, ale nemodifikuje je, nazýváme Eva (angl. Eve, podle eavesdropper = odposlech). Pokud útočník zprávy nejen poslouchá, ale také modifikuje, nahrazuje vlastními zprávami nebo přehrává staré zprávy, je nazýván Mallory (z angl. malicious = zlomyslný, zákeřný). Někdy se pro něj také používají jména Marvin aMallet, případně také Trudy (z angl. intruder = vetřelec).

Nakonec potřebujeme v bezpečnostních protokolech důvěryhodné třetí strany. V této souvislosti se používají následující tři jména:

- Trent – důvěryhodná (angl. trusted) třetí strana, jejíž přesná role závisí na konkrétním protokolu
- Victor – ověřovatel (angl. verifier)
- Sam – důvěryhodný server

Nakonec zmiňme Waltera (z angl. warden = hlídač), který je někdy potřeba, aby určitým způsobem dohlížel nad Alicí a Bobem, pokud to závažnost situace vyžaduje.

3.4 Univerzalita

Z etymologického hlediska se slovo „univerzalita“ skládá ze dvou kořenů:

- *unum* = jedno
- *vertere* = obracet se

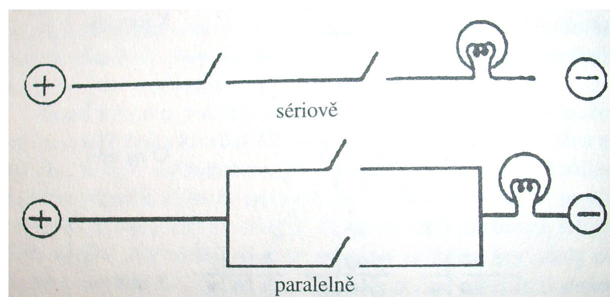
Na univerzitní půdě můžeme také slýchávat spojení těchto slov „ad unum vertere“, což znamená „k jednomu se obraceti“, případně „přivádět v jednotu“.

Univerzalita tedy ukazuje, že zdánlivě různé věci se nakonec „v jedno obrátí“. Jednoduchým příkladem může být sčítání: Někdo může sčítat v desítkové soustavě, někdo ve dvanáctkové, ale nakonec se ukáže, že oboje ščítání má stejnou výpočetní sílu, tedy že není možné jedním způsobem vypočítat něco jiného a že lze jeden postup převést na druhý.

Univerzalita logických hradel

Konstruktér počítačů W. D. Hillis popisuje ve své knize [5] důležitost booleovské logiky a její použití v počítačích. Zároveň také ukazuje, že způsob, jakým budou logické hodnoty ve skutečnosti reprezentovány, stejně jako způsob, jakým budou provedeny logické operace, může být velmi různorodý.

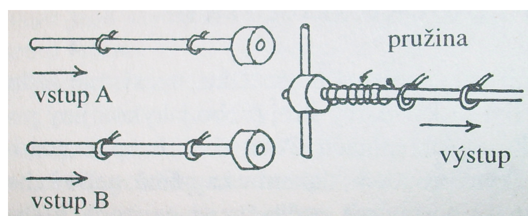
Kromě prakticky použitelného případu, kdy signály jsou reprezentovány určitými hodnotami napětí elektrického proudu a logické operace prováděny pomocí tranzistorů, uvádí Hillis tři další možnosti, jak je možné booleovskou logiku implementovat. Postupně představíme všechny tři. Uvedené ilustrace pocházejí z knihy [5].



Obrázek 7: Logický součin a součet

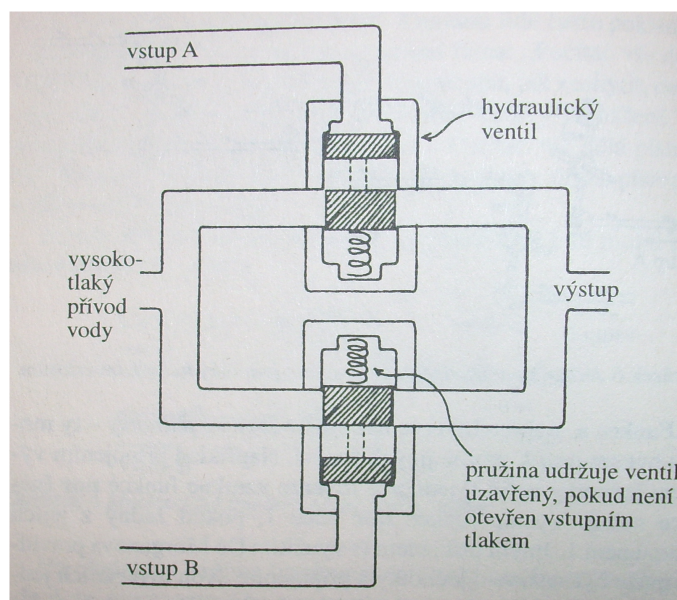
První způsob implementace logických hradel využívá spínače (tlačítka), žárovky a zdroj napětí. Vstupy logických operací jsou reprezentovány stisknutím tlačítka, výstup operace indikuje žárovka. V případě logického součinu implementovaného sériovým zapojením svítí žárovka, pouze pokud jsou oba vypínače sepnuty. Podobně pro logický

součet, který implementujeme pomocí paralelního zapojení, platí, že žárovka svítí, pokud je sepnut alespoň jeden vypínač.



Obrázek 8: Mechanická implementace logického součtu

Druhý způsob implementace využívá mechanický pohyb. Obrázek 8 ukazuje možnou implementaci logického součtu. Poloha tyčí určuje jejich hodnotu – pokud je tyč vlevo, má hodnotu 0, pokud je tyč vpravo, má hodnotu 1. Pokud tedy alespoň jedna ze vstupních tyčí A nebo B bude vpravo (tj. bude mít hodnotu 1), bude mít také výstup hodnotu 1 – bude také vpravo.



Obrázek 9: Hydraulický logický součet

Třetí způsob implementace logických hradel využívá schopnosti kapality šířit tlak. Pokud je v trubce tlak (tak silný, aby pohnul ventilem), má hodnotu 1, jinak 0. V uvedeném příkladu logického součtu se do výstupní trubky (vpravo) dostane tlak, pokud na alespoň jednom ze vstupů bude také tlak.

Tento příklad měl ilustrovat, že přes různorodost implementace logických hradel umí všechna tato hradla v konečném důsledku pouze vyhodnocovat logické operace. Ukazuje se tedy, že logické hradlo je univerzální princip, který není svázán s nějakou konkrétní implementací a každá implementace se nakonec „v jedno obrátí“.

3.5 Jednoduchost

Jakkoli je informatika věda složitá a komplikovaná, nesmíme zapomínat na jeden důležitý princip – jednoduchost. Při naší práci bychom si měli vždycky uvědomovat, že složitost není cílem naší práce, nýbrž jakýmsi nutným průvodcem. Jsou ovšem situace, kdy se můžeme složitosti vyhnout.

Extrémní programování, jeden z možných přístupů k tomu, jak vyvíjet software, má jedno heslo, které se týká právě jednoduchosti: „Simpliest thing that could posibly work." Tedy, „nejjednodušší věc, která by mohla eventuálně fungovat." Extrémní programování nás nabádá, abychom začali něčím jednoduchým a bude-li to později třeba (tj. bude-li to zákazník chtít), můžeme to rozšířit. Není nic smutnějšího, než programovat funkcionalitu, o které vám pak zákazník řekne, že ji nepotřebuje (a případně nezaplatí).

Chtěl bych uvést několik příkladů z praxe, kdy zbytečná složitost řešení zapříčiňuje zbytečné plýtvání zdrojů:

- psát programový kód v nízkoúrovňovém jazyce v okamžiku, kdy optimalizace na výkon není třeba; místo toho je lepší využít vyšší programovací jazyk, který umožní rychlejší napsání kódu za cenu nižšího výkonu, pokud toto snížení není kritické.
- nakupovat hardware nebo software, které poskytuje funkcionalitu, kterou nevyužijeme; výběrem jednodušších zařízení ušetříme peníze, protože nemusíme platit za zbytečné vlastnosti.

Na závěr této kapitolky bych chtěl uvést tři citáty, které vyjadřují některé aspekty myšlenky jednoduchosti:

> „Dělejte věci jednoduše: tak jednoduše, jak je to jen možné, ale ne jednodušeji."
> *Albert Einstein*

> „Jednoduchost znamená dosáhnout maximálního efektu s minimálním úsilím."
> *Koichi Kawana, architekt botanických zahrad*

> „Dokonalosti je dosaženo, ne když není nic, co bychom mohli přidat, ale když není nic, co bychom mohli odebrat." *Antoine de Saint-Exupery*

3.6 Inspirace

Ve zprávě o stvoření světa, která je zachycena hned v první knize Bible, čteme toto: „Bůh viděl, že všechno, co učinil, je velmi dobré." (Genesis 1,31a). Ať už má čtenář na stvoření světa jakýkoli názor, jistě bude souhlasit, že člověk se často nechává inspirovat stvořením, tím, co vidí kolem sebe v přírodě. Let ptáků inspiroval člověka ke konstrukci letadla. Na několik dalších inspirací v souvislosti s informatikou se nyní podíváme blíže. Půjde o kvantové počítání, neuronové sítě a evoluční programování.

Kvantové počítání

Jedna z myšlenek kvantového počítání spočívá v tom, že atomy seskupující se do molekul zaujmou vždy přesnou polohu v podstatě okamžitě. Pokud bychom chtěli tuto polohu vypočítat na dnešním počítači, zabralo by to určitý čas, a to tím více, čím přesnějšího výsledku bychom chtěli dosáhnout.

Otázkou tedy je, zda tuto geniální schopnost atomů (nebo jiných částic) jsme schopni využít při konstrukci výpočetního stroje. V každém případě jsme fascinováni touto schopností, inspirováni sestrojit něco podobného.

Neuronové sítě

Při zkoumání funkce lidského mozku byla objevena struktura buněk, které si mezi sebou posílají signály. Každá taková buňka na příchozí signály reaguje odesláním nebo neodesláním svého signálu dál. Jednotlivé buňky se nazývají neurony a jejich struktura se pak nazývá neuronová síť.

Fakt, že lidský mozek je schopen nejen počítat, ale také přemýšlet, inspiruje informatiky k tomu, aby vytvářeli umělé neurony (nebo je simulovali na počítači) a spojovali je do sítí. Přestože výsledky, kterých se tím dosáhne, jsou v porovnání se skutečným mozkem nezajímavé, stále je o neuronové sítě velký zájem.

Evoluční programování

Evoluční/genetické programování/algoritmy jsou pojmy, které všechny vycházejí ze stejného principu. Využívají myšlenku, že organizmy v přírodě se vyvíjejí pomocí křížení, náhodných mutací a přirozeného výběru. Tento evoluční postup se používá zejména v situacích, kdy není snadné hledat řešení nějakého problému přímo, takže vygenerujeme množinu možných řešení, které necháme mezi sebou křížit, mutovat a

vybíráme z nich ty lepší, čímž se nakonec (s trochou štěstí) dostaneme k řešení, které se od optimálního příliš neliší.

Použití tohoto principu popisuje Hillis [5] na vývoji programu pro třídění čísel. Své výsledky komentuje takto: „Použil jsem matematické testy, abych dokázal, že vzniklé třídící programy třídí bezchybně, ale věřím ještě více procesu, který je vytvořil, než matematickým testům. Vím totiž, že každý z vyvinutých třídících programů pochází z dlouhé řady programů, jejichž přežití záviselo na schopnosti třídit."

Problémem, který komplikuje použití evolučního vývoje, je fakt, že přirozený výběr neuznává význam mutací, jejichž význam je vidět až v rámci určité skupiny mutací. Tuto myšlenku ilustruje Ryrie [7]: „Za příklad si vezměme vývoj oka. Došlo-li by nejprve k mutaci, která vytvořila slzný kanálek, byla by při přirozeném výběru tato mutace zachována, dokud by nedošlo k dalším mutacím, jež by vedly ke vzniku řas, víčka, rohovky, čoček atd? Nebo by při přirozeném výběru organizmus se slzným kanálkem zanikl, protože by neexistovaly žádné ostatní složky zrakového systému a protože slzný kanálek by byl sám o sobě neužitečný?"

4 Shrnutí

Co říci závěrem? Autor si je vědom toho, že práce jistě nevyčerpala téma a existují oblasti, které může někdo považovat za fundamentální a nebyly zmíněny. Podobně jako jedna algebra může mít různé množiny generátorů (třeba i disjunktní), tak i někdo jiný může vidět fundamenty informatiky v něčem jiném.

Pokud tedy někoho toto téma zaujalo a měl by na něj třeba jiný názor, může použít tuto práci jako odrazový můstek pro své další bádání.

5 Příloha A: Implementace rekurzivního stromu

Následující program v jazyce R kreslí strom podle rekurzivní definice. Pro spuštění je potřeba si vykonat následující kroky:

1. Stáhnout instalační program jazyka R z http://www.r-project.org/.
2. Nainstalovat a spustit prostředí jazyka R.
3. V menu `File > Open script...` otevřít soubor obsahující program.
4. V menu `Edit > Run all` spustit program.

Zdrojový kód

Zde uvedený zdrojový kód je také obsažen na přiloženém CD v souboru `tree.R`.

```
#Rekurzivní funkce, která kreslí strom
tree <- function(x, y, len, depth = 6, dir = 0) {
     x2 <- x + sin(dir) * len;
     y2 <- y + cos(dir) * len;

     #Nakreslíme kmen
     lines(c(x,x2),c(y,y2),lwd=depth)

     if (depth > 0) {
           len2 <- len * rnorm(1,len_mean,len_sd)

           dir2a <- dir + rnorm(1,dir_mean,dir_sd)
           dir2b <- dir - rnorm(1,dir_mean,dir_sd)

           #Nakreslíme větve = menší stromy
           tree(x2,y2, len2, depth - 1, dir2a)
           tree(x2,y2, len2, depth - 1, dir2b)
     }
}

#Abychom mohli reprodukovat výsledek, zafixujeme náhodu
set.seed(42);

#Připravíme si místo pro kreslení
frame()
plot.window(c(0,400),c(0,600))

#Nastavíme míru zkracování a vychylování podstromů
len_mean <- 0.7
len_sd <- 0.1

dir_mean <- 0.3
dir_sd <- 0.05

#Nakreslíme strom
tree(200,50,150)
```

Literatura

[1] Petráčková V., Kraus J. a kolektiv.: *Akademický slovník cizích slov*, Academia, Praha, 2000, str 194.

[2] Töpfer P.: *Algoritmy a programovací techniky*, Prometheus, Praha, 1995.

[3] Eckel B.: *Myslíme v jazyku C++*, Grada, Praha, 2000, str. 26n.

[4] Arlow J., Neustadt I.: *UML a unifikovaný proces vývoje aplikací*, Computer Press, Brno, 2003, str. 28n .

[5] Hillis W. D.: *Vzor v kameni*, Academia, Praha, 2003, str. 30 a 102.

[6] *Wikipedie*, http://en.wikipedia.org/wiki/Thinking, citováno dne 11. února 2008.

[7] Ryrie, C. C.: *Základy teologie*, BIBLOS, Třinec, 1994, str. 199.