
KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Michal Krupka
krupka.inf.upol.cz
michal.krupka@upol.cz
17. listopadu 12, 771 46 Olomouc

Paradigmata programování 2 ◇ poznámky k přednášce

10. Srozumitelnost zdrojového kódu

verze z 20. dubna 2020

1 Úvod

Při programování nejde jen o vymyšlení a správné zapsání postupu, který má pak počítač vykonávat. Jednou napsaný program má obvykle správně fungovat mnoho let a nesmí být příliš obtížné (až nemožné, což se bohužel často děje) v něm během této doby dělat úpravy. Už během práce na první verzi je také potřeba, aby zdrojovému kódu rozuměli různí programátoři z týmu. V programování jde tedy také o *komunikaci* mezi programátory (kterými můžete klidně být pořád vy, ale v různém čase). Komunikačním médiem je zdrojový kód, a ten musí být čitelně a srozumitelně napsán. Toho se týká tato přednáška.

Výklad začíná popisem formálních pravidel (jako např. formátování) a pokračuje dalšími základními principy. Těžiště je ale v posledních kapitolách, kde se principy srozumitelnosti zdrojového kódu zabýváme více do hloubky.

Obecné poznatky z přednášky budete moci použít v libovolném programovacím jazyce. Tady ale jako příklad jazyka (pochopitelně) používám Lisp.

2 Formátování

Formátování kódu je čistě formální vlastnost. Je ale jasné, že podstatně ovlivňuje srozumitelnost. V mnoha různých učebnicích a na mnoha webových stránkách si můžete prohlédnout příklady správného a nesprávného formátování zdrojového kódu psaného například v C. Tady ukážu pár příkladů v Lispu.

Formátování zdrojového kódu můžeme chápat čistě jako problém umístění prázdných znaků, a to zejména mezer a konců řádků. Správné formátování tedy zahrnuje pravidla

- jak odsazovat řádky,
- jaké a kam vkládat mezery do řádků,
- kde ukončovat řádky,
- jak velké mají být mezery mezi řádky.

Odsazování řádků.

To slouží jako základní nástroj k orientaci ve struktuře zdrojového kódu, a to bez ohledu na to, jaké prvky textu ji ve skutečnosti určují. V Lispu například při čtení nikdo nepočítá závorky, orientuje se téměř výhradně pomocí odsazování řádků. Ve vhodném editoru je přitom snadné správné odsazování zařídit (polo)automaticky, většinou pomocí tabelátoru. To také programátorovi slouží jako kontrola. Podívejte se na tyto výrazy:

```
(let ((a 1)
(b 2))
(if (> a b)
    (print 1))
    (print 2)
    (print 3))

(let ((a 1)
(b 2))
  (if (> a b)
      (print 1)
      (print 2))
      (print 3))
```

Jak rychle poznáte, co dělají?

Pravidlo lze v odůvodněných případech porušit. Takhle by například mohla vypadat část našeho prvního interpretu Scheme ze zimního semestru:

```
;; globální (počáteční) prostředí
(let ((env (make-env (list (cons 'zero 0) (cons '+ #'+)
                           (cons '* #'*) (cons '= #'=)
                           (cons 'cons #'cons))
                     nil)))

(defun initial-env ()
  env)

(defun set-initial-env (value)
  (setf env value))

)
```

Výjimku lze podrobně zdůvodnit, nebudu se tady do toho pouštět. Příklad obsahuje i dvě výjimky z dalších pravidel; celý je takový výjimečný.

Mezery uvnitř řádků.

Výrazy Lispu jsou většinou seznamy. Pokud jsou prvky seznamu na jednom řádku, oddělují se obvykle jednou mezerou. Například tento způsob definice funkce je v Lispu nevhodný, přestože bude fungovat správně:

```
(defun f(x)
  ...)
```

Stejně jako takto zapsaná aplikace funkce:

```
(f(+ x y))
```

Pokud něco takového napíše student, vyvolává to ve mně podezření, že nechápe význam závorek v Lispu (který je jiný než ve většině jazyků) a možná by ani nechápal, proč toto:

```
(f(+ (x) y))
```

nebo toto:

```
(+x y)
```

vede k chybě.

Jinak, kromě posledních dvou příkladů ty předchozí fungovaly proto, že reader při převodu textu na seznam bere levou závorku jako začátek seznamu, bez ohledu na to, že před ní není mezera.

Kde ukončovat řádky.

Především, v žádném programovacím jazyce není vhodné psát příliš dlouhé řádky. Rozumně volené ukončování řádků společně se správným odsazováním podstatně zvyšuje čitelnost kódu. (Pro mě osobně je dobrá škola psát tyto texty, protože zde je délka řádků kódu velmi omezená.)

Jeden složený výraz, pokud není příliš dlouhý, můžeme napsat na jeden řádek, jako třeba tady:

```
(if a b c)
```

Jinak ho formátujeme, jak je pro příslušný výraz obvyklé:

```
(if a
    b
  c)
```

Obecně máme vždy dvě volby: buď celý výraz napsat na řádek, nebo ho celý dělit podle zvyklostí. Sem tam se připouštějí výjimky, jednu z nich (která se jistě dá dobře zdůvodnit) jste viděli v příkladě nahoře:

```
(list (cons 'zero 0) (cons '+ #'+)
      (cons '* #'*) (cons '= #'=)
      (cons 'cons #'cons))
```

Mezery mezi řádky.

V rámci jedné definice (tedy např. v `defun`-výrazu nebo `defmacro`-výrazu) by nikdy neměl být prázdný řádek. Mezi definicemi je obvykle právě jeden řádek.

3 Jména

Čitelnost programu podstatně závisí na správné volbě jmen proměnných, funkcí atd. Programovací jazyky obvykle povolují rozmanitěji utvořená jména, než je vhodné. Při volbě jména je třeba dbát na to, aby

- jméno bylo utvořeno podle daných formálních pravidel,
- bylo konzistentní s ostatními jmény,
- bylo dostatečně popisné.

Formální pravidla používání jmen jsou v různých jazycích různá, vždy je třeba vycházet z pravidel pro příslušný jazyk (a případně projekt, firmu atd.). V Lispu jsou v zásadě dvě: jména se píší malými písmeny (přestože příslušné symboly mají pak názvy složené z velkých písmen), slova se většinou oddělují pomlčkou (taková pravidla používají i jazyky COBOL, Forth a CSS), a názvy globálních proměnných začínají a končí hvězdičkou.

Konzistentnost jmen usnadňuje pochopení účelu, kterému jméno slouží. V Lispu například (pokud nepoužíváme *místa*) máme ve zvyku začínat názvy všech mutátorů datových struktur předponou „`set-`“. Konstruktory datových struktur v Lispu zase obvykle začínají „`make-`“. Globální proměnné ovlivňující formát výstupu zase všechny začínají na „`*print-`“. (Common Lisp ovšem není zrovna příkladem konzistentnosti názvů, to je dáno historickým vývojem a kvůli zpětné kompatibilitě se s tím nedá nic moc dělat.) Pro selektory datových struktur jsme žádnou pojmenovávací konvenci nezaváděli, ale v jiných jazycích často začínají předponou „`get`“.

Při používání kódů a knihoven z různých zdrojů si ovšem musíme zvyknout na rozdílný styl.

Popisnost jmen je neméně důležitá. Stojí za to věnovat vymyšlení vhodného jména energii a čas. Porovnejte tyto definice. Která je pochopitelnější?

```
(defun fun (a b c d)
  ...)

(defun point-distance (pt1-x pt1-y pt2-x pt2-y)
  ...)
```

Velmi popisné názvy se obvykle používají pro veřejné funkce, tedy ty, které píšeme jako vstupní body např. knihovny a které uvádíme v dokumentaci. Čím více hrají funkce a proměnné pomocnou roli, tím mohou být jejich názvy stručnější a méně srozumitelné. Při tomto zkracování bychom ale neměli být příliš horliví, ať se ve svém vlastním kódu později vyznáme.

4 Idiomy

V jazykovědě se *idiomem* rozumí rčení, které chápáno doslova dává jiný než zamýšlený smysl nebo ho nedává vůbec. V češtině jsou idiomy třeba „mít za ušima“ nebo „dostat do těla“. Různé jazyky mívají dost rozdílné idiomy (například v angličtině „call it a day“ nebo „let the chips fall where they may“.)

V programování se idiomem rozumí ustálený způsob řešení nějakého problému v konkrétním programovacím jazyce. Může být dokonce pro neznalé jazyka nesrozumitelný či zdánlivě nesmyslný, podstatné ale je, že ze všech možných řešení by se měl používat právě on, aby byl na první pohled vidět účel dané části kódu. *Idiomatické programování*, neboli *programování v idiomech* je programování s využitím idiomů, kde to jde. Idiomy se v různých programovacích jazycích liší a programátor by neměl přenášet idiomy jednoho jazyka do druhého.

Toto je například idiomatický způsob, jak inicializovat prvky pole `a` délky `n` v jazyce C:

```
for( i = 0; i < n; i++ )
  a[i] = n;
```

Až na nepatrné rozdíly (např. napsat `++i` místo `i++`) by programátor vždy měl používat tento způsob, i když si jistě dovede představit deset jiných (např. použít `while`-cyklus).

V Lispu například pro větvení s jednou větví místo

```
(if a (print b) nil)
```

nebo dokonce

```
(if a (print b))
```

Píšeme zásadně

```
(when a (print b))
```

To činí program mnohem čitelnějším, protože čtenáře ani na chvilku nenapadne hledat druhou větev. (Nemluvím teď o další výhodě makra `when`, že na místě `(print b)` připouští více výrazů.) Neidiomatický kód narušuje čtenářovo očekávání: když vidí operátor `if`, automaticky bude kromě první větve hledat i druhou. V kombinaci s dalšími prohřešky (nesprávné formátování) může vést k nesprávnému pochopení programu.

Místo

```
(if (not a) (print b))
```

je nejlepší napsat

```
(unless a (print b))
```

Ještě méně idiomatické je napsat místo `(when a (print b))` toto:

```
(and a (print b))
```

Přestože z pohledu programu je to přesně totéž (!), význam pro programátora je úplně jiný. Narušili jsme jeho očekávání, protože u operátoru `and` čekal logické výrazy.

5 Říkat, co máme na mysli

Než začneme psát funkci, měli bychom nejprve mít jasnou představu, co má dělat. Jaký algoritmus nebo obecně postup by měla vykonávat. Poté řešíme neméně obtížnou otázku: jak postup zapsat v daném programovacím jazyce. To je značně ošidná fáze, protože se vlastně snažíme udělat překlad naší představy funkce do příslušného programovacího jazyka. Kvůli snížení chybovosti a zvýšení čitelnosti

by tento překlad měl být co nejmenší. Měli bychom programem vyjádřit pokud možno co nejvěrněji, co chceme říci.

Upravit funkci tak, aby byla co nejsrozumitelnější, dá někdy více práce, než ji napsat, ale tu práci je dobré vykonat.

Jako pomocníka ovšem potřebujeme kvalitní programovací jazyk: potřebujeme, aby měl dostatek **vyjadřovacích prostředků**. Základním rysem očekávaným od programovacího jazyka samozřejmě je, že v něm musí jít zapsat libovolný algoritmus. To obecné (*general-purpose*) programovací jazyky splňují (některé jednoúčelové ne). Bohatství vyjadřovacích prostředků ovšem hraje také významnou roli. Uvedu několik příkladů.

Představme si, že chceme do proměnné `a` uložit obsah proměnné `plus`, je-li číslo n kladné, a nulu, pokud není. V programovacích jazycích, ve kterých máme možnost napsat podmíněný výraz jako výraz s návratovou hodnotou, můžeme napsat přesně to, co jsme chtěli udělat:

```
(setf a (if (> n 0) plus 0))
```

To lze ještě lehce zpřehlednit, pokud známe lispovou funkci `plusp`:

```
(setf a (if (plusp n) plus 0))
```

(Zde je to nepodstatné, ale kdyby místo proměnné `n` byl dlouhý výraz, pomohlo by to čitelnosti.)

V jazyce bez „funkcionálního `if`“ bychom museli před napsáním řešení provést malou kompilaci v hlavě a dát `if` dopředu:

```
(if (plusp n)
    (setf a plus)
  (setf a 0))
```

Nejde jen o to, že tento výraz je delší než předchozí (už se mi zdálo vhodnější rozdělit ho na víc řádků), ale že už nevyjadřuje přesně zadání, tedy to, jakou akci jsme chtěli do programu napsat. Museli jsme ji nejdřív přeložit do programovacího jazyka.

Další příklad můžeme vzít z minulého semestru. Kdy je hodnota čistý seznam? Když je to symbol `nil` *nebo* pár a jeho *cdr* je čistý seznam. A doslova napsáno:

```
(defun proper-list-p (x)
  (or (null x)
      (and (consp x)
           (proper-list-p (cdr x)))))
```

Můžete se podívat do přednášky 6 z minulého semestru na tři jiné a o dost hůře čitelné varianty tohoto predikátu. Hůře čitelné jsou proto, že nekopírují doslova naši definici řečenou (téměř) přirozeným jazykem. (Poznámka: kruhové seznamy jsme v prvním semestru nebrali v úvahu; tato funkce by na nich cyklila.)

Ještě jeden příklad úspěšného zapsání našich myšlenek do zdrojového kódu. Týká se minulých dvou přednášek, a to definice hlavové normální formy. Podle definice, výraz je *hlavová normální forma*, když je

1. proměnná nebo
2. abstrakce nebo
3. aplikace, jejíž hlava není abstrakce a je hlavová normální forma.

Když si uvědomíme, že výraz nemůže být nic jiného než proměnná, abstrakce nebo aplikace, nemusíme ve třetím bodě už testovat, zda je výraz aplikace. A můžeme napsat:

```
(defun hnfp (expr)
  (or (variablep expr)
      (abstractionp expr)
      (and (not (abstractionp (appl-head expr)))
           (hnfp (appl-head expr)))))
```

Vidíme, že až na drobnou úpravu slovosledu jsme zapsali přesně znění definice.

Ještě bych dodal, že pohled na to, co znamená čitelnost kódu, se bude v různých programovacích jazycích lišit. Bude se to ale lišit i v rámci jednoho jazyka a jednoho programu, protože na různých místech programu můžeme řešit různé problémy. Můžeme používat různé nástroje a knihovny; každá knihovna si může stanovovat své zvyklosti a idiomy: význam a pořadí parametrů funkcí, způsob práce s prostředky, typické pořadí volání funkcí apod. Tak v rámci jednoho programovacího jazyka tak trochu používáme několik velmi podobných, ale přece jen drobně se lišících jazyků.

Lisp jde v tomto ještě o krok dál, protože ke zvýšení srozumitelnosti kódu nám umožňuje připravit si vlastní makra. (Víme, že samotný Lisp je do značné míry pomocí maker budován, například operátor `when`, o kterém jsme před chvilkou hovořili, je makro, které nám někdo kvůli zvýšení srozumitelnosti připravil.) Makra umožňují přetvořit Lisp na sice podobný, ale přece jen jiný jazyk. To má své výhody i nevýhody.

Co nejvěrněji formulovat do zdrojového kódu programu naše myšlenky patří k důležitým pokročilým programátorským schopnostem. Je až překvapivé, jak málo programátorů to umí nebo se o to aspoň snaží. Na konci přednášky bude obsáhlejší příklad.

6 Další pravidla

Následující programátorská pravidla se netýkají přímo čitelnosti kódu, ale výrazně ji ovlivňují. Jde víceméně o náhodný výběr, jistě by se našla i další.

Organizace zdrojového kódu. Jde jak o organizaci kódu do různých souborů, tedy tzv. *modularitu*, tak i o organizaci částí kódu v rámci jednoho souboru.

Komentáře. Je-li potřeba kód okomentovat, aby byl srozumitelnější, může to být příznak toho, že není dostatečně čitelně napsán. Překvapivě často zjistíme, že kód lze přepsat tak, aby komentář nebyl nutný. Pokud to nejde, samozřejmě komentujeme.

Používání abstrakcí. O tom jsme mluvili už minulý semestr na 2. přednášce. Abstrakce umožňují říct, *co* kód dělá bez nutnosti popisovat, *jak* to dělá.

Krátké funkce. Základní programátorské pravidlo říká, že funkce by neměly mít větší počet řádků než nějaký limit (který se pro různé jazyky může lišit). Ještě lepší pravidlo zní: *Žádná funkce by neměla dělat víc než jednu věc. Jinak je třeba ji rozdělit na víc funkcí.* To samozřejmě zvyšuje čitelnost programu.

Volba jazyka. Tohle pravidlo je snad nejdůležitější a přitom značně podceňované. K psaní programu bychom měli zvolit takový jazyk, který vytvoření čitelného zdrojového kódu umožňuje.

7 Příklad: množiny a relace poprvé

Pokusíme se naprogramovat jeden jednoduchý matematický příklad a jeden příklad z oblasti dolování dat (*data mining*). Nejprve to uděláme sice kvalitně, ale bez ohledu na požadavek „říkat, co mám na mysli“. Postupně pak budeme program vylepšovat směrem k větší srozumitelnosti, ke konci i s použitím pokročilých technik Lispu. V průběhu práce zjistíme, že příklady spolu souvisejí.

Nejdřív tedy trochu matematiky. Řekněme, že máme zobrazení $f\colon X \to Y$. Pro množinu $B \subseteq Y$ definujeme *vzor* (*pre-image*) *množiny* B *při zobrazení* f jako množinu všech prvků množiny X, které se zobrazením f zobrazí do B. Vzor množiny B při zobrazení f označujeme $f^{-1}(B)$ (neplést s inverzním zobrazením!). Matematickým zápisem tedy můžeme napsat:

$$f^{-1}(B) = \{x \in X \mid f(x) \in B\}.$$

K nalezení této množiny je potřeba projít všechny prvky množiny X, na každý aplikovat zobrazení f a podívat se, zda výsledek leží v množině B. Pokud bychom množinu reprezentovali seznamem a zobrazení funkcí, můžeme vzor množiny naprogramovat například těmito třemi způsoby:

```
(defun f-preimage (fun B X)
  (let ((result '()))
    (dolist (el X)
      (when (member (funcall fun el) B)
        (push el result)))
    result))

(defun f-preimage (fun B X)
  (cond ((null X) '())
        ((member (funcall fun (car X)) B)
         (cons (car X) (f-preimage fun B (cdr X))))
        (t (f-preimage fun B (cdr X)))))

(defun f-preimage (fun B X)
  (remove-if-not (lambda (el) (member (funcall fun el) B))
                 X))
```

Poznámky:

- Symboly pro matematické množiny tady záměrně píšu velkým písmenem, i když to není obvyklé. Hle, odůvodněné porušení zvyklosti...
- Funkce `member` je jako funkce `find`, ale najde i hodnotu `nil`, protože vrací něco jiného než přímo hledaný prvek. Lze ji tak přesně použít jako predikát zjišťující, zda daný prvek leží v množině (je-li množina reprezentována seznamem).

Všechny tři varianty fungují, jak mají, ale liší se čitelností. První je napsána procedurálně pomocí cyklu a bude tedy srozumitelnější pro toho, kdo nerozumí dobře rekurzi. Druhá je funkcionální. Třetí taky a navíc s výhodou využívá funkci `remove-if-not` (funkce `remove-if-not` vypustí z daného seznamu prvky, které nesplňují danou podmínku; nechám tam tedy ty, které ji splňují.) Žádná z funkcí ale nevyhovuje přísnějším požadavkům srozumitelnosti zdrojového kódu.

Touto poznámkou zatím vzor množiny opustíme a podíváme se na obraz. Pro množinu $A \subseteq X$ je *obraz* (*image*) *množiny* A *při zobrazení* f množina obrazů všech prvků množiny A. Značí se $f(A)$ (neplést s obrazem prvku) a takto ji můžeme zapsat matematickým jazykem:

$$f(A) = \{f(x) \mid x \in A\}$$

Tady jsou dvě možnosti implementace:

```
(defun f-image (fun A)
  (let ((result '()))
    (dolist (el A)
```

```
      (push (funcall fun el) result))
    (delete-duplicates result)))

(defun f-image (fun A)
  (delete-duplicates (mapcar fun A)))
```

(Funkce `delete-duplicates` je destruktivní verze funkce `remove-duplicates`. Tady si ji můžeme dovolit, protože pracuje s čerstvě vytvořenými a nikde jinde nepoužitými seznamy.)

Tady zjevně ve stručnosti a pochopitelnosti vede druhá verze, stejně ale na první pohled není vidět, co má dělat — čtenář si musí udělat opačným směrem překlad, o kterém jsem mluvil dříve.

A teď druhý příklad. Máme množiny X a Y a mezi nimi binární relaci I. Prvky množiny X budeme chápat jako *objekty* a prvky množiny Y jako *stributy*. Pokud je prvek $x \in X$ v relaci I s prvkem $y \in Y$, řekneme, že *objekt x má atribut y*.

Pro libovolnou množinu atributů B definujeme množinu objektů $B^{\downarrow}$ jako množinu objektů, které mají všechny atributy z množiny B. Tedy

$$B^{\downarrow} = \{x \in X \mid x \text{ je v relaci } I \text{ s každým prvkem množiny } B\}.$$

Pro množinu objektů A definujeme množinu atributů $A^{\uparrow}$ jako množinu všech atributů, které má každý objekt z A. Neboli

$$A^{\uparrow} = \{y \in Y \mid \text{každý prvek množiny } A \text{ je v relaci } I \text{ s prvkem } y\}.$$

Naším úkolem je implementovat $^{\uparrow}$ a $^{\downarrow}$ jako funkce v Lispu. Množiny budeme opět reprezentovat seznamy. Pro jednoduchost množiny X a Y uložíme do globálních proměnných `*X*` a `*Y*`. Bude to užitečné pro experimentování v Listeneru. (Časem sem dodám ukázku, jak pak můžeme funkce snadno upravit, aby byly stále použitelné jednoduše a přitom nezávisely na globálních proměnných.) Za objekty i atributy budeme brát přirozená čísla od nuly po danou horní mez. V našem příkladě budou nastaveny takto:

```
(defvar *X* '(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18))
(defvar *Y* '(0 1 2 3 4 5 6 7))
```

Budeme tedy pracovat s devatenácti objekty osmi atributy.

Relaci I uložíme do globální proměnné `*I*` jako dvourozměrné pole. Na pozici (x, y) v poli bude jednička, právě když objekt x má atribut y:

```
(defvar *I*
  (make-array '(19 8)
              :initial-contents '((0 0 0 0 0 1 1 0)
                                  (0 1 0 1 0 1 0 0)
                                  (1 1 1 1 0 0 1 0)
                                  (1 1 1 1 0 1 0 0)
                                  (1 0 0 0 0 1 0 0)
                                  (1 1 1 1 0 1 1 0)
                                  (1 0 0 0 0 0 0 0)
                                  (1 1 1 0 0 0 0 0)
                                  (1 1 1 0 0 1 0 1)
                                  (1 1 1 1 1 1 0 1)
                                  (1 1 0 0 0 0 0 0)
                                  (0 1 0 0 0 0 0 0)
                                  (0 1 0 0 0 0 0 0)
                                  (1 1 1 1 0 1 1 0)
                                  (1 1 1 1 0 1 0 0)
                                  (1 1 1 1 1 1 0 0)
                                  (1 1 1 0 0 1 0 1)
                                  (1 1 1 1 1 1 0 0)
                                  (1 1 1 1 1 1 1 1))))
```

Jde o pole 19×8. Volbou `:initial-contents` jsme rovnou při definici inicializovali jeho obsah.

Zda má objekt o atribut a nyní zjistíme funkcí `has`:

```
(defun has (o a)
  (= 1 (aref *I* o a)))
```

Test (můžete zkontrolovat podle obsahu proměnné `*I*` výše):

```
CL-USER 1 > (has 6 2)
NIL

CL-USER 2 > (has 2 6)
T
```

Napsat funkce pro $^{\uparrow}$ a $^{\downarrow}$ podle jejich definice by nemělo být těžké. Verze pomocí cyklů:

```
(defun up (A)
  (let ((result '()))
```

```
    (dolist (y *Y*)
      (let ((add t))
        (dolist (x A)
          (unless (has x y)
            (setf add nil)))
        (when add (push y result))))
    result))

(defun down (B)
  (let ((result '()))
    (dolist (x *X*)
      (let ((add t))
        (dolist (y B)
          (unless (has x y)
            (setf add nil)))
        (when add (push x result))))
    result))
```

Poznámka: vnitřní cykly by nemusely dobíhat do konce; v momentě, kdy nastavíme proměnnou `add` na `nil`, je možné z vnitřního cyklu vyskočit. Pro jednoduchost to tady nedělám, i když by to šlo.

Testy (opět porovnejte s tabulkou nahoře; nezapomeňte, že indexujeme od nuly):

```
CL-USER 5 > (up '(1 2 3))
(1 3)

CL-USER 6 > (down '(1 3))
(1 2 3 5 9 13 14 15 17 18)
```

Funkcionální verze:

```
(defun up (A)
  (remove-if-not (lambda (y)
                   (every (lambda (x)
                            (has x y))
                          A))
                 *Y*))

(defun down (B)
  (remove-if-not (lambda (x)
                   (every (lambda (y)
                            (has x y))
                          B))
                 *X*))
```

(Funkce `every` testuje, zda každý prvek seznamu splňuje daný predikát.)

A testy:

```
CL-USER 5 > (up '(1 2 3))
(1 3)

CL-USER 6 > (down '(1 3))
(1 2 3 5 9 13 14 15 17 18)
```

Tím je náš úkol vyřešen? Částečně ano. Všechny zde napsané funkce dělají, co mají. Máme ale výhrady vůči jejich srozumitelnosti. V dalších částech se je budeme snažit přepisovat, aby lépe vyjadřovaly, co jsme měli na mysli.

8 Příklad: více deklarativnosti

Všimněte si, že nesrozumitelnost našich funkcí byla dána tím, že se příliš zabývaly technickými detaily. A čím více se jimi zabývaly, tím méně srozumitelné byly. Dá se to říct i tak, že se příliš zabývaly tím, *jak* něčeho dosáhnout, místo aby ukazovaly, *čeho* chceme dosáhnout. To je rozdíl mezi tzv. *imperativním* a *deklarativním* stylem programování.

Imperativním stylem popisujeme krok po kroku, jaké akce je třeba vykonat k dosažení výsledku.

Deklarativní styl je založen na tom, že pouze řekneme co má být výsledkem, aniž bychom explicitně popisovali, jak ho dosáhnout.

Myslím, že z toho, co jsme zatím v této přednášce naprogramovali, je jasné, že varianty funkcí, které byly napsány více funkcionálním stylem, se deklarativnímu stylu více blížily.

Všimněte si, že matematický jazyk, kterým jsme popisovali operace, jež jsme chtěli naprogramovat, je čistě deklarativní. Například tady:

$$A^{\uparrow} = \{y \in Y \mid \text{každý prvek množiny } A \text{ je v relaci } I \text{ s prvkem } y\}$$

říkáme, že množina $A^{\uparrow}$ se skládá z prvků, které jsou v relaci s každým prvkem množiny A. Nepopisujeme, jak množinu sestrojit. To je nám jedno, můžete ji sestrojit, jak chcete, hlavně, že bude sestrojená. Matematici se algoritmy moc nezabývají.

Matematický popis je ovšem pro řešení našich problémů vhodný, protože se týkají matematických pojmů (přestože část řešení pak bude použita v informatice, konkrétně dolování dat). Programátor zde musí matematickým pojmům, jako je obraz množiny při zobrazení či operace $^{\uparrow}$ a $^{\downarrow}$, rozumět a také by měl rozumět jejich matematického popisu.

Napsat naše funkce čitelněji bychom tedy mohli tak, že se zápisem přiblížíme deklarativnímu matematickému popisu definovaných množin, který jsme použili například k uvedené definici množiny $A^{\uparrow}$.

Matematický popis množiny, který tady používáme, je dvojího typu. První je tento:

$$\{x \in X \mid x \text{ splňuje podmínku } P\}.$$

Podmínku P můžeme chápat jako funkci, která prvku množiny X přiřadí hodnotu *Pravda* nebo *Nepravda*. Zápisem je zadána množina všech prvků množiny X, pro které je hodnota podmínky rovna *Pravdě*.

Druhý typ popisu množiny je složitější a zahrnuje v sobě typ první:

$$\{f(x) \mid x \in X \text{ a splňuje podmínku } P\}.$$

Tady máme navíc nějaké zobrazení f z množiny X, řekněme $f : X \to Y$. Popisovaná množina je podmnožinou množiny Y. Skládá se z prvků vzniklých aplikací funkce f na všechny prvky množiny X splňující podmínku P.

Podmínka je obecnější než podmínka předchozí, protože za zobrazení f můžeme vzít identitu na množině X.

K vytvoření množiny podle uvedeného popisu tedy potřebujeme zadat

- množinu X,
- podmínku (predikát) pro její prvky
- a nepovinně další funkci, která se na prvky množiny X splňující podmínku aplikuje.

Takto se tedy dá definovat nová množina matematickým popisem:

```
(defun make-set (superset test &optional (map #'identity))
  (delete-duplicates (mapcar map (remove-if-not test superset))))
```

Funkce `identity` je lispová implementace identity. Pro každý argument vrací nezměněný týž argument (zkuste si to v Listeneru). (Funkci `make-set` lze vylepšit; na druhou stranu by to ale bylo na úkor čitelnosti.)

Podívejme se, jak se teď dají napsat naše funkce `f-preimage`, `f-image`, `up`, `down` pomocí funkce `make-set`.

```
(defun f-preimage (fun B X)
  (make-set X (lambda (el) (member (funcall fun el) B))))

(defun f-image (fun A)
  (make-set A #'identity fun))

(defun up (A)
  (make-set *Y*
            (lambda (y)
              (every (lambda (x) (has x y))
                     A))))

(defun down (B)
  (make-set *X*
            (lambda (x)
              (every (lambda (y) (has x y))
                     B))))
```

Jak je hodnotíte? Já vcelku kladně, jako mírný posun k matematickému způsobu zadání množiny, a tedy jako čitelnější verze než ty předchozí (to ocení hlavně ti, kdo matematickému jazyku rozumí). Naštěstí náš programovací jazyk ještě neřekl poslední slovo.

9 Od organizace kódu k syntaxi

Řešení, kterými jsme ukončili předchozí kapitolu, znamenají asi maximum čitelnosti, kterého lze dosáhnout v každém programovacím jazyce disponujícím lexikálními uzávěry. Pokud chceme napsat funkce ještě čitelněji a srozumitelněji (tedy, pro programátory znalé matematické symboliky množin, jak jsem psal), což bychom měli chtít, musíme se poohlédnout po dalších nástrojích. Ty už se budou v různých jazycích lišit. V našem jazyce jsou takovým nástrojem makra.

Můžeme to brát tak, že vhodné funkce jsme už napsali — mám na mysli funkci `make-set`, která dělá explicitně to, co bylo v předchozích verzích skryto: vytvoření nové množiny s danými vlastnostmi podle matematických zvyklostí. Dalším krokem by tedy mohlo být nalezení *syntaktického* nástroje, tedy zapsání už ekvivalentního kódu jinak, čitelněji. Jak jsem psal, to už by se v každém jazyce dělalo jinak (pokud by to šlo). U nás to tedy bude pomocí maker.

Za největší překážku čitelnosti dosud nalezeného řešení můžeme považovat použití uzávěrů. Například podmínku „pro každý prvek $y \in B$ platí, že x má y jako atribut“ jsme museli napsat jako

```
(every (lambda (y) (has x y))
       B)
```

Zde je ona „kompilace z hlavy do programu“ výrazně přítomna.

Chceme tedy čitelněji zapsat podmínku s velkým kvantifikátorem ($\forall$). Ta mívá obvykle takový tvar:

$$\text{Pro každé } x \in A \text{ platí } P(x)$$

Takže nejvýstižnější by bylo asi něco takového:

```
(for-all x :in A :it-holds (P x))
```

Takové makro ovšem umíme:

```
(defmacro for-all (symbol &key (in nil inp) (it-holds nil))
  (unless inp (error "FOR-ALL: IN must be specified"))
  `(every (lambda (,symbol)
            ,it-holds)
          ,in))
```

Použil jsem i test, zda uživatel nezapomněl uvést základní množinu (klíčem `:in`). Nepovinný parametr je tedy ve skutečnosti povinný. (K této zajímavosti se časem ještě vrátím.)

Teď si můžeme vychutnat snadnost použití makra `for-all`:

```
CL-USER 3 > (for-all x :in '(1 2 3) :it-holds (plusp x))
T

CL-USER 4 > (for-all x :in '(1 2 3) :it-holds (evenp x))
NIL
```

Když jsme v tom, napíšeme si i existenční kvantifikátor, i když ho nebudeme potřebovat. Nazveme ho `for-some`, aby dávalo anglicky smysl použít opět konzistentně klíč `:it-holds`:

```
(defmacro for-some (symbol &key (in nil inp) (it-holds t))
  (unless inp (error "FOR-SOME: IN must be specified"))
  `(some (lambda (,symbol)
           ,it-holds)
         ,in))
```

Test:

```
CL-USER 5 > (for-some x :in '(-1 -2 3) :it-holds (plusp x))
T

CL-USER 6 > (for-some x :in '(-1 -2 -3) :it-holds (plusp x))
NIL

CL-USER 7 > (for-all x :in '(1 2 3)
                       :it-holds (for-some y :in '(1 2 3)
                                             :it-holds (<= x y)))
T

CL-USER 8 > (for-all x :in '(1 2 3)
                       :it-holds (for-some y :in '(1 2 3)
                                             :it-holds (< x y)))
NIL
```

Podobným způsobem napíšeme makro na popis množiny matematickým jazykem. Makro používá funkci `make-set`.

```
(defmacro set-of (form &key (where form wherep) in (such-that t))
  `(make-set ,in
             (lambda (,where)
               ,such-that)
             (lambda (,where)
               ,form)))
```

Makro zahrnuje obě varianty popisu množiny. V případě, že uživatel použil jako první argument symbol (je to tedy ta jednodušší varianta použitá například ve funkci `f-preimage`), neměl by použít argument `where`. (Opět bychom mohli dát test na začátek makra.)

Smysl makra je zřejmý z jeho použití:

```
CL-USER 9 > (set-of x :in '(1 2 3 4) :such-that (oddp x))
(1 3)

CL-USER 10 > (set-of (* x x) :where x :in '(1 2 3 4)
                             :such-that (oddp x))
(1 9)

CL-USER 11 > (set-of x :in '(1 2 3 4)
                       :such-that (for-some y :in '(3 4 5 6)
                                              :it-holds (<= y x)))
(3 4)
```

Chtěl jsem teď popsat, co každý ze tří testů počítá, ale proč bych to dělal, je to přímo čitelné ze zadaných výrazů.

Další verze našich čtyř funkcí; teď už vůči jejich srozumitelnosti může těžko kdo něco namítat:

```
(defun f-inv-image (fun B X)
  (set-of x :in X :such-that (member (funcall fun x) B)))

(defun f-image (fun A)
  (set-of (funcall fun x) :where x :in A))

(defun up (A)
  (set-of y :in *Y*
            :such-that (for-all x :in A :it-holds (has x y))))

(defun down (B)
  (set-of x :in *X*
            :such-that (for-all y :in B :it-holds (has x y))))
```

Jaký udělat z dosažených výsledků závěr? Samozřejmě jsme využili vyjadřovací schopnosti, tedy bohatost výrazových prostředků našeho jazyka. Makra se ukázala být vhodným a silným nástrojem. Co je ale podstatnější a čeho jste si možná nevšimli, je to, že úsilí dosáhnout co nejsrozumitelnějšího zápisu našich čtyř funkcí nás dovedlo k hlubšímu pochopení *struktury řešeného problému*. Pochopili jsme, že problém má vlastně tři vrstvy:

1. vrstvu logiky (kvantifikátory),
2. vrstvu množinovou (popis množiny matematickým jazykem)
3. a teprve nad nimi vrstvu pro práci s daty (to mluvím o funkcích `up` a `down`).

Toto pochopení je velice cenné a dovedlo nás k lepšímu strukturování zdrojového kódu a jeho rozdělení na skupiny souvisejících funkcí.

V souvislosti s použitím maker jste si asi všimli, že vedlo k mírnému posunutí našeho Lispu k jinému programovacímu jazyku, a to k jazyku schopnému vyjadřování deklarativním matematickým jazykem. To má své výhody i nevýhody, jak už jsem psal. Programátor, který by měl s naším zdrojovým kódem pracovat, se musí kromě Lispu učit ještě další jeho úpravy. Je tedy vždy třeba zvážit, zda to stojí za to. (Programátoři v Lispu jsou na to ovšem zvyklí.)

Někdy v budoucnu ještě přidám třešinku v podobě následující kapitoly. Zatím ale text končí zde.