

KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

PARADIGMATA OBJEKTIVÉHO PROGRAMOVÁNÍ I

MICHAL KRUPKA



VÝVOJ TOHOTO UČEBNÍHO TEXTU JE SPOLUFINANCOVÁN
EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY

Olomouc 2008

Abstrakt

Obsahem textu jsou základy objektově orientovaného programování.

Cílová skupina

První část textu je určena studentům všech bakalářských oborů vyučovaných na katedře informatiky Přírodovědecké fakulty UP Olomouc. Chystaná druhá část textu je určena především studentům bakalářského oboru Informatika.

Obsah

1	Úvod	6
2	Od Scheme ke Common Lispu	8
2.1	Pozadí	8
2.2	Terminologie	8
2.3	Logické hodnoty a prázdný seznam	9
2.4	Vyhodnocovací proces	11
3	Common Lisp: základní výbava	15
3.1	Řízení běhu	15
3.2	Proměnné a vazby	16
3.3	Místa	20
3.4	Funkce	21
3.5	Logické operace	24
3.6	Porovnávání	25
3.7	Čísla	26
3.8	Páry a seznamy	27
3.9	Chyby	29
3.10	Textový výstup	30
3.11	Typy	31
3.12	Prostředí	32
4	Třídy a objekty	33
4.1	Třídy	33
4.2	Třídy a instance v Common Lispu	34
4.3	Inicializace slotů	40
5	Zapouzdření	43
5.1	Motivace	43
5.2	Princip zapouzdření	47
5.3	Úprava tříd <code>point</code> a <code>circle</code>	48
5.4	Třída <code>picture</code>	49
6	Polymorfismus	53
6.1	Kreslení pomocí knihovny <code>micro-graphics</code>	53
6.2	Kreslení grafických objektů	57
6.3	Princip polymorfismu	63
6.4	Další příklady	64
7	Dědičnost	70
7.1	Dědičnost jako nástroj redukující opakování v kódu	70
7.2	Určení předka při definici třídy	72

7.3	Přiblížení běžným jazykům	73
7.4	Hierarchie tříd grafických objektů	76
7.5	Přepisování metod	78
7.6	Volání zděděné metody	80
7.7	Inicializace instancí	84
7.8	Příklady	85
8	Příklady	86
8.1	Symbolické výrazy poprvé	86
8.2	Zápis a čtení symbolických výrazů	88
8.3	Symbolické výrazy a dědičnost	90
8.4	Zpráva <code>simplify</code> a volání metody předka	94
8.5	Posloupnosti	96
8.6	Zprávy fyzické úrovně	97
8.7	Zprávy logické úrovně	98
8.8	Jednorozměrná pole	99
8.9	Seznamy	103
8.10	Posloupnosti: další příklady	106
9	Události	107
9.1	Zpětná volání v knihovně <code>micro-graphics</code>	107
9.2	Jednoduché využití zpětných volání	108
9.3	Vlastnické vztahy, delegování, události	110
9.4	Implementace událostí u jednoduchých grafických objektů	111
9.5	Reakce na změny u jednoduchých objektů	114
9.6	Klikání myši u jednoduchých objektů	119
9.7	Reakce na změny u složených objektů	121
9.8	Klikání myši u složených objektů	126
9.9	Příklady	127
A	Slovníček	128
A.1	Základní výrazy	128
A.2	Odvozené výrazy	128
A.3	Makra	128
A.4	Standardní procedury — predikáty ekvivalence	129
A.5	Standardní procedury — čísla	129
A.6	Standardní procedury — logické typy	129
A.7	Standardní procedury — páry a seznamy	130
A.8	Standardní procedury — symboly	130
A.9	Standardní procedury — znaky	130
A.10	Standardní procedury — řetězce	131

A.11 Standardní procedury — vektory	131
A.12 Standardní procedury — řízení běhu	132
A.13 Standardní procedury — eval	132
A.14 Standardní procedury — vstup a výstup	132
A.15 Standardní procedury — systémové rozhraní	133
B Knihovna micro-graphics: seznam použitelných barev	134

1. Úvod

Tento text pokrývá část látky předmětu Paradigmata programování 2 (dříve Paradigmata programování 3), vyučovaného autorem na Přírodovědecké fakultě Univerzity Palackého Olomouc od roku 2005, a je určen studentům tohoto předmětu i dalším zájemcům o hlubší pochopení principů objektového programování. Chystaná druhá část tohoto textu se zabývá pokročilejšími a méně obvyklými rysy objektového programování.

Kurz Paradigmata programování 2 představuje pro většinu studentů první setkání s objektovým programováním. Jeho cílem je seznámit studenty s obecnými základy objektového programování, bez úzké vazby na konkrétní programovací jazyk. Kurz tak slouží jako protiváha předmětů, v nichž je objektové programování probíráno se zřetelem k přímému uplatnění v praxi a v nichž se studenti musejí zabývat zvláštnostmi jednotlivých v současné době v praxi používaných programovacích jazyků.

Text klade důraz na praktickou stránku problematiky, je proložen mnoha praktickými příklady, další příklady, které tvoří nedílnou součást textu, lze nalézt na webu autora. Výklad je ve většině případů koncipován metodou „zdola nahoru“, většina pojmů a technik je nejprve inspirována příkladem či problémem. Smysl každého pojmu by tak měl být čtenáři jasný dříve, než je pojem přesně vymezen.

Objektové programování obsahuje nástroje, které pomáhají programátorům zvládat práci na složitých a rozsáhlých programech. Kvůli pochopení některých důležitých principů tohoto programovacího stylu bylo tedy nevyhnutelné sáhnout po poměrně netriviálních příkladech — na takových příkladech se teprve síla objektového programování projeví. Čtenář tak může ocenit a pochopit metody, které objektové programování poskytuje, při zvládání praktických úkolů. Není nucen řešit akademické příklady, do kterých by uměle vnášel složité principy způsobem, který by v praxi neobstál. (Toto předsevzetí nebylo z pochopitelných důvodů možno dodržet stoprocentně. Proto čtenář v textu i nějaké „akademické“ příklady najde. Celkové ladění textu by ale mělo být jiné.)

Jako modelový programovací jazyk byl použit Common Lisp a jeho objektová část, Common Lisp Object System (CLOS). Použitým vývojovým prostředím je program LispWorks. Hlavní důvody této volby jsou následující:

- Studenti znají z předchozího studia příbuzný jazyk Scheme.
- Díky tomu, že se jedná o dynamicky typovaný jazyk s velmi jednoduchou syntaxí, se studenti nemusí zabývat podružnými problémy, které by je odváděly od pochopení podstatného (jako příklad mohu uvést, že jsem se nikdy nesetkal s tím, že by studenti při řešení úkolů z objektového programování měli jakékoliv dotazy ohledně syntaxe). Na závěr tohoto textu zjistíme, že se nám podařilo poměrně snadno a rychle napsat plně funkční jednoduchou objektovou 2D grafickou knihovnu.
- Common Lisp je dynamický programovací jazyk, psaní a ladění programů je velmi pohodlné a rychlé (při opravě chyby ve vykreslování okna často není nutné okno vůbec zavírat — abychom vyléčili pacienta, není nutné ho předtím zabít).
- CLOS je bohatý a současně snadno pochopitelný objektový programovací jazyk, v němž lze demonstrovat většinu běžných a mnoho neobvyklých rysů objektového programování.
- Program LispWorks je moderní vývojový nástroj obsahující všechny podstatné nástroje pro tvorbu i rozsáhlejších aplikací. Je dostupný na všech hlavních platformách (Windows, Mac OS X, Linux), k dispozici je plně funkční (s nepodstatnými omezeními) verze zdarma.

- Grafická knihovna LispWorks (CAPI) je přenositelná mezi platformami a obsahuje vše potřebné pro implementaci naší grafické knihovny.

Na webu autora je k dispozici kód a příklady k jednotlivým částem textu, i knihovna micro-graphics.

Michal Krupka

2. Od Scheme ke Common Lispu

Účelem této části je poskytnout uživatelům jazyka Scheme první informace potřebné k zahájení práce s Common Lisphem. Čtenář se dozví o základních rozdílech mezi těmito jazyky, zejména terminologických a rozdílech ve vyhodnocovacím procesu.

Další informace o Common Lispu lze nalézt v následujících částech tohoto textu, již bez vazby na jazyk Scheme. Vždy je vhodné mít po ruce nějakou příručku o Common Lispu, jako reference může dobře sloužit webová podoba standardu, [Common Lisp HyperSpec](#).

V dodatku [A](#) je uveden slovníček, ve kterém čtenář najde ke každému symbolu definovanému ve standardu R⁵RS jeho ekvivalent v Common Lispu.

2.1. Pozadí

Jakkoli jsou rozdíly mezi syntaxí jazyka Scheme a Common Lispu malé, přechod od Scheme ke Common Lispu nemusí být pro programátora rutinně pracujícího ve Scheme snadný. Podstata obou jazyků, jejich účel nebo, chceme-li, duch, jsou totiž značně rozdílné. Zatímco jazyk Scheme vznikl a je dodnes vyvíjen jako samostatný programovací jazyk — byť založený na Lispu —, jehož hlavním rysem je jednoduchost a čistota, Common Lisp přímo navazuje na rozličné předchozí dialekty Lispu, počínaje původním, uvedeným koncem padesátých let. V těchto dialektech bylo napsáno mnoho rozsáhlých programů, včetně operačních systémů pro počítače přímo navržené pro práci s Lisphem¹ a jedná se tedy o jazyk robustní, obsahující desetiletí vyvíjené a osvědčené nástroje potřebné pro tvorbu velkých projektů mnohočlennými vývojovými týmy.

Poznámka 2.1. Common Lisp lze ztěžít označit za čistý jazyk vysoké úrovně. Rozšířené tvrzení, že se jedná o „jazyk umělé inteligence,“ nepoužitelný pro praktickou práci, je zcela mylné. Ve většině implementací i ve standardu samotném je tradičně k dispozici mnoho nízkoúrovňových nástrojů, z nichž některé bychom dokonce v implementacích jiných jazyků těžko hledali — bitové operace, instrukce skoku lokálního i nelokálního, práce s ukazateli, inline assembler, přímé napojení na služby operačního systému, možnost prohlížet jednotlivě u každé funkce její zkompileovaný tvar v assembleru, lokálně v libovolné části kódu (i uvnitř funkce) nastavovat různé varianty optimalizace přeloženého kódu, dokonce i do určité míry programovat kompilátor.

Z uvedených důvodů se Common Lisp těžko mohl vyhnout jisté komplexnosti a mírné vnitřní nekonzistentnosti. Začínající programátor v Common Lispu se pohybuje rozsáhlým a nepřehledným terénem, ve kterém se z počátku nesnadno orientuje. Má k dispozici mnoho složitých, leckdy těžko srozumitelných nástrojů a jen pomalu se dobírá pochopení jejich podstaty a vzájemné souvislosti. Důvody, proč stojí za to úsilí do studia Common Lispu vložit, již byly vysvětleny. Tento text se snaží využívat poměrně značně omezenou část Common Lispu, která je popsána v této a následujících dvou částech, a stavět na čtenářových znalostech jazyka Scheme.

2.2. Terminologie

Některé základní rysy jazyků Scheme a Common Lisp jsou označovány rozdílnými termíny. Na tomto místě shrneme základní rozdíly.

Poznámka 2.2. Nepřesnost ve vyjadřování vede k nepřesnosti sdělení. Nepřesné pochopení základů mívá v programování jasný důsledek: špatně napsaný program.

¹Tzv. Lisp Machines, vyráběné koncem sedmdesátých a v osmdesátých letech, které významně ovlivnily vývoj výpočetní techniky.

Objektům, které se ve Scheme nazývají *procedury*, se v Common Lispu říká *funkce*. Ve standardu se dočteme, že *funkce* je objekt, který může být *volán* s žádným nebo více argumenty a který vytváří (vrací) nula nebo více hodnot. Je to tedy objekt, na jaké jsme zvyklí z většiny programovacích jazyků, i když s možností vrácení více než jedné hodnoty se obvykle nesetkáváme (nemluvě ovšem vůbec o základním rysu funkcionálního programování, že funkce je objekt prvního řádu, tj. objekt, se kterým lze manipulovat stejně jako s ostatními objekty). K pojmu funkce je třeba dodat, že kromě vrácení výsledku může funkce také vykonat vedlejší efekt — tato okolnost je v oblasti funkcionálního programování nepříjemnou a nevíтанou nutností, objektové programování ji naopak používá jako základní pracovní nástroj.

Víme, že v Lispu je zdrojový kód programu složen ze seznamů a dalších objektů (zejména čísel, symbolů, textových řetězců). Libovolnému objektu, který je částí zdrojového kódu programu, říkáme *výraz*. Termín *forma* má v Common Lispu jiný význam než ve Scheme (kde se používá hlavně ve spojení *speciální forma*²): Podle standardu je *forma* libovolný objekt, který je určen k vyhodnocení. Jak víme (a to je v Scheme i Common Lispu stejné), vyhodnocovat můžeme objekty jednoduché, jako jsou čísla a symboly (tzv. atomy), a objekty složené, neprázdné seznamy (prázdný seznam je atom). Vyhodnocování seznamů probíhá v Common Lispu poněkud jinak než ve Scheme; popíšeme je později.

U složených forem se první položka nazývá *operátor*. Podle operátoru rozlišujeme tři typy složených forem: *funkční formy*, *makro formy* a *speciální formy* (tento pojem tedy znamená něco jiného než stejný pojem ve Scheme). Funkční formy odlišíme od makro forem podle toho, jestli je jejich operátor jménem funkce nebo jménem makra. Posledně jmenované speciální formy jsou formy, jejichž operátor je jedním z 25 symbolů definovaných standardem. Tyto symboly se nazývají *speciální operátory*.

2.3. Logické hodnoty a prázdný seznam

V jazyce Scheme slouží k reprezentaci logických hodnot objekty `#t` (*pravda*) a `#f` (*nepravda*). V Common Lispu tuto úlohu hrají hodnoty `t` a `nil`. Narozdíl od Scheme se nejedná o hodnoty speciálního typu, ale o symboly. Zvláštností těchto symbolů je, že je pro ně stanoveno, že se vyhodnocují samy na sebe, podobně jako například čísla:

```
> nil
nil
> t
t
```

(později se setkáme s mnoha dalšími takovými symboly).

Poznámka 2.3. Na tomto místě je samozřejmě vhodné vědět, co je to *symbol*. Pokud to nevíte, zopakujte si příslušné části kurzu jazyka Scheme.

Poznámka 2.4. Zařídit, aby hodnotou symbolu byl dotýčný symbol samotný je ovšem snadné; ve Scheme by se dalo napsat například `(define a 'a)`, v Common Lispu například `(defvar a 'a)`.

V Common Lispu se stejně jako ve Scheme používají zobecněné logické hodnoty: hodnotu *pravda* (*true*) může reprezentovat libovolný objekt kromě symbolu `nil`, hodnotu *nepravda* (*false*) pak jedině symbol `nil`. V literatuře o Common Lispu i v tomto textu je zvykem psát slovo *pravda* místo konkrétnější hodnoty v případě, že je tato hodnota

²Tento termín není ve standardu R⁵RS nikde definován ani použit (kromě jedné historické zmínky), jedná se ale o mezi uživateli Scheme obecně rozšířený pojem a čtenář se s ním setkal.

podstatná pouze jako pravdivostní hodnota. V tomto kontextu se také používá slovo *nepravda* jako synonymum pro symbol `nil`.

Jako argumenty logických operací je tedy možno používat jakékoliv objekty a samotné logické operace mohou jako hodnotu *pravda* místo symbolu `t` vracet jiný, v dané situaci užitečnější objekt.

Příklad 2.5. Výraz `(and a b c)` vrací *pravdu*, pokud hodnoty `a`, `b` i `c` jsou *pravda*. Díky použití zobecněných logických hodnot je možno, aby proměnné `a`, `b`, `c` nabývaly i jiných hodnot než jenom `t` nebo `nil`, a hodnotou celého výrazu může být i něco jiného, než jen symboly `t` a `nil`. Makro `and` je například definováno tak, že pokud jsou všechny jeho argumenty *pravda* (rozumějme: různé od `nil`), vrací hodnotu posledního z nich. Výraz, který vrátí číslo 3, pokud je `x` rovno 1 a `y` rovno 2, a jinak vrátí `nil`, se tedy dá jednoduše napsat takto: `(and (= x 1) (= y 2) 3)`. Pokud bychom navíc třeba chtěli, aby v případě negativního výsledku výraz nevracel `nil`, ale nulu, můžeme použít makro `or`, které také pracuje se zobecněnými logickými hodnotami, a napsat jej takto: `(or (and (= x 1) (= y 2) 3) 0)`.

V Common Lispu není žádná zvláštní hodnota reprezentující prázdný seznam; roli prázdného seznamu hraje symbol `nil`:

```
> '()
nil
> '(1 . nil)
(1)
```

Poznámka 2.6. Symbol `nil` tak zastává funkce, na které jsou v jazyce Scheme vyčleněny tři různé objekty: logická hodnota `#f`, prázdný seznam a symbol `nil`.

Poznámka 2.7. Používání symbolu `nil` v Common Lispu ke třem různým účelům bývá často uživateli zvyklými na jazyk Scheme kritizováno. Z praktického hlediska se ale tato okolnost ukazuje být častěji spíše předností než nedostatkem. V Common Lispu je běžné, a my se s tímto přístupem budeme často setkávat, užívat symbolu `nil` jako univerzální negativní či prázdné hodnoty. To je pochopitelně umožněno tím, že může vystupovat současně jako prázdný seznam i pravdivostní hodnota *nepravda* — v jazyce Scheme se s žádnou podobně univerzální hodnotou nesetkáme.

Poznámka 2.8. Nevýhoda pojetí zvoleného v Common Lispu se může projevit například v momentě, kdy hledáme nějaký seznam a za pozitivní výsledek považujeme i seznam prázdný. V Common Lispu pak hodnota `nil` už nemůže signalizovat neúspěch. Jako příklad lze uvést třeba funkci, která by měla vrátit cestu mezi dvěma vrcholy grafu ve formě seznamu hran, nebo hodnotu *nepravda*, když taková cesta neexistuje (prázdná cesta spojující vrchol s ním samým je tedy prázdný seznam a pozitivní výsledek současně), či funkci, která má nalézt daný prvek v daném seznamu, v případě úspěchu vrátit podseznam začínající prvkem následujícím a v případě neúspěchu hodnotu *nepravda* (zde dojde k potížím, pokud je nalezený prvek na konci seznamu). Podobné problémy se ale nevyskytují příliš často a v Common Lispu je lze vždy snadno obejít. Koneckonců, oddělení hodnoty `#f` a prázdného seznamu ve Scheme stejně neřeší například problém prohledávání množiny, která může obsahovat i hodnotu `#f`.

Poznámka 2.9. Za zmínku snad ještě stojí, že v některých jazycích je definováno více hodnot reprezentujících logickou hodnotu *nepravda*. Například v Pythonu to je (mimo jiné) hodnota `none`, prázdný seznam, nebo číslo 0 libovolného typu. Další nepravdivé hodnoty lze dodefinovat.

Stejně jako v jazyce Scheme se funkcím, které vrací hodnotu *pravda* nebo *nepravda*, říká *predikáty*. V jazyce Scheme je obvyklé ukončovat názvy predikátů otazníkem, v Common Lispu písmeny „p“ nebo „-p“.

Varianta bez pomlčky se používá v případě, že koncovku přidáváme k jednoslovnému názvu (například u symbolů `equalp`, `stringp` a podobně), jinak se používá varianta s pomlčkou (`upper-case-p`).

2.4. Vyhodnocovací proces

V Common Lispu může symbol sloužit současně jako název funkce i jako název proměnné. Tyto dvě role symbolů spolu nijak nekolidují. Proto je možno napsat:

```
> (defvar list 1) ;definice proměnné list s hodnotou 1
1
> list ;v proměnné list je hodnota 1:
1
> (list 2 3) ;funkci list to však neovlivnilo:
(2 3)
```

V terminologii Common Lispu se říká, že každý symbol má *dvě vazby*: *hodnotovou* a *funkční*. Pomocí každé z těchto vazeb může být symbol svázán s nějakou hodnotou. V případě vazby hodnotové s libovolnou hodnotou, v případě vazby funkční s funkcí.

Poznámka 2.10. Záměrně nezmiňujeme případ, kdy symbol neoznačuje funkci, ale makro, nebo kdy je speciálním operátorem. Tato informace je uložena stejně jako u funkcí pomocí funkční vazby symbolu (symbol nemůže například současně označovat funkci a makro). Podle standardu je přitom hodnota svázaná funkční vazbou se symbolem označujícím makro nebo speciální operátor implementačně závislá; standard neříká, jakého typu tato hodnota je.

Je-li symbol svázán funkční vazbou s nějakou funkcí, říkáme, že je *názvem* této funkce. V případě hodnoty svázané se symbolem hodnotovou vazbou hovoříme prostě o *hodnotě symbolu* (či, volněji, *hodnotě proměnné*).

Příklad 2.11. Dvojitá role symbolů tedy umožňuje používat stejné názvy pro proměnné a funkce. Proto například můžeme napsat:

```
(defun comp (list)
  (list (car list)))
```

(pomocí makra `defun` se v Common Lispu definují funkce; viz níže uvedený doslovný překlad do Scheme). Funkce `encap-car` vrací jednoprvkový seznam obsahující první prvek seznamu `list`:

```
(encap-car '(1 2)) => (1)
```

Ve Scheme by doslovná analogie definice této funkce vypadala takto:

```
(define encap-car
  (lambda (list)
    (list (car list))))
```

Scheme
nesprávně

neboli

```
(define (encap-car list)
  (list (car list)))
```

Scheme
nesprávně

a volání `(encap-car '(1 2))` by skončilo chybou (proč?).

Poznámka 2.12. V Common Lispu je tedy možné vybírat pro proměnné názvy bez ohledu na to, jestli současně označují nějakou funkci — tak, jak jsme to viděli v případě funkce `encap-car`; u této funkce jistě neexistuje přirozenější název pro její parametr, než je symbol `list` (snad jedině symbol `cons`, který je ale rovněž názvem funkce) — není nutno sahát po náhradních názvech, jako například `lst` nebo `clist`.

Poznámka 2.13. Této možnosti se v Common Lispu poměrně často využívá; jakmile si na ni uživatel zvykne, začne přijímat nabízenou volnost spíše jako výhodu a bez obav z nedorozumění. Ve zdrojovém kódu je totiž podle pozice symbolu vždy jasné poznat, kterou z jeho dvou možných hodnot právě uvažujeme.

Poznámka 2.14. V základních implementacích Common Lispu bývá k dispozici možnost zjistit snadno ke kterékoliv funkci její dokumentaci a seznam parametrů. Například v LispWorks můžeme umístit kurzor na symbol `encap-car` a z nabídky vyvolat `Expression→Arguments`. Pokud jsme funkci `encap-car` definovali stejně jako zde, dozvíme se, že seznam parametrů této funkce je `(list)`. Ve vývojovém prostředí SLIME se nám podobná informace zobrazí automaticky během psaní volání funkce: `(encap-car...)`.

Názvy parametrů funkcí nejsou tedy pouze interní věcí autora, ale slouží i jako jednoduchá dokumentace pro uživatele. Proto je vhodné volit názvy parametrů funkce srozumitelně, s ohledem na případného uživatele. Dvoje vazby symbolů v Common Lispu umožňují pojmenovávat parametry funkcí co nejpopsnějším způsobem.

Existence dvojí vazby symbolů, kterou jsme zatím prezentovali jako jednoznačnou výhodu, nutně vede i k několika komplikacím. Kromě složitějšího vyhodnocovacího procesu jde zejména o dva případy: když chceme zavolat funkci uloženou v proměnné a když chceme zjistit funkci podle jejího názvu.

K ilustraci těchto problémů nejprve uvedme následující definici procedury na kompozici dvou procedur v jazyce Scheme:

```
(define comp  
  (lambda (a b)  
    (lambda (x)  
      (a (b x))))
```

 Scheme

neboli

```
(define (comp a b)  
  (lambda (x)  
    (a (b x))))
```

 Scheme

Analogická definice v Common Lispu by vypadala takto:

```
(defun comp (a b)  
  (defun (x)  
    (a (b x))))
```

 nesprávně

a vedla by k nepředvídatelným důsledkům, protože ve výrazu `(a (b x))` by se nepoužily aktuální *hodnoty* proměnných `a` a `b`, ale došlo by k pokusu aplikovat *funkce se jmény* `a` a `b`, o kterých není jasné, jestli by existovaly a pokud ano, co by dělaly.

Poznámka 2.15. Přečtěte si podrobně předchozí odstavec. Důvod, proč uvedený příklad v Common Lispu nefunguje, je třeba přesně pochopit.

V naší definici potřebujeme volat nikoli funkce, jejichž jména jsou `a` a `b`, ale funkce, které jsou hodnotami proměnných `a` a `b` — tedy jsou se symboly `a`, `b` svázány hodnotovou, nikoliv funkční vazbou. Pro podobné situace je v Common Lispu připravena funkce `funcall`, která volá funkci, kterou najde ve svém prvním argumentu. Správná definice funkce `comp` v Common Lispu je tedy následující:

```
(defun comp (a b)
  (lambda (x)
    (funcall a (funcall b x))))
```

Ke druhému problému: pomocí funkce `comp` a funkcí `car` a `cdr` lze snadno vyjádřit funkci, která vrací druhý prvek daného seznamu. Ve Scheme by takovou funkci (proceduru) vrátil výraz

```
(comp car cdr)
```

Scheme

V Common Lispu by vyhodnocení tohoto výrazu opět vedlo k nepředvídatelným důsledkům, protože by se v něm nepoužily funkční, ale hodnotové vazby symbolů `car` a `cdr`. Abychom získali funkce `car` a `cdr`, musíme použít speciální operátor `function`, který je navržen přesně k požadovanému účelu — nalezení funkce zadaného jména. Funkce `car` a `cdr` získáme vyhodnocením výrazů `(function car)` a `(function cdr)`. Analogický výraz v Common Lispu by tedy správně vypadal takto:

```
(comp (function car) (function cdr))
```

Pro výraz `(function name)` se používá zkratka `#'name`, takže uvedený výraz je možno napsat stručněji:

```
(comp #'car #'cdr)
```

Dvojí vazby symbolů představují jediný významný rozdíl mezi vyhodnocovacím procesem Scheme a Common Lispu.

Pro zopakování: Chceme-li zavolat funkci uloženou v proměnné `a` s argumenty `p1, ..., pn`, nestačí jako ve Scheme napsat

```
(a p1 ... pn)
```

Scheme

ale

```
(funcall a p1 ... pn)
```

Chceme-li získat funkci s názvem `f`, musíme místo prostého

```
f
```

Scheme

uvést

```
#' f
```

což je zkratka pro

```
(function f)
```

Kvůli zopakování a pro pozdější potřebu uvádíme na obrázku 1 zjednodušený popis vyhodnocovacího procesu v Common Lispu. Tento vyhodnocovací proces je stejně jako ve Scheme rekurzivní; kdekoli se v popisu na obrázku hovoří o vyhodnocení nějakého objektu, znamená to spuštění celého vyhodnocovacího procesu od začátku na tento objekt.

Vyhodnocováním objektu (kterému, jak už víme, za těchto okolností říkáme také forma), rozumíme jistý přesně definovaný proces, který tomuto objektu přiřadí hodnotu³ a má případný vedlejší efekt.

Seznam posledních změn

10. 3. 2008 Přidána poznámka o predikátech a způsobu jejich zápisu.

³V předchozím textu bylo uvedeno, že výrazy v Common Lispu mohou mít více hodnot. Pro jednoduchost zde tuto vlastnost výrazu neuvažujeme.

Vyhodnocení formy F

Je-li F symbol, výsledkem je hodnota symbolu F , vedlejší efekt není žádný.

Je-li F seznam s první položkou Op , pak je-li Op

speciální operátor nebo symbol pojmenovávající makro, výsledek včetně případného vedlejšího efektu závisí na speciálním operátoru Op , případně definici makra Op a ostatních prvcích seznamu F .

λ -výraz definující funkci Fun nebo symbol pojmenovávající funkci Fun , vyhodnotí se zleva doprava všechny prvky seznamu F počínaje druhým a zavolá se funkce Fun s argumenty rovnými hodnotám těchto vyhodnocení. Hodnotou výrazu bude hodnota vrácená funkcí Fun , během výkonu této funkce také může dojít k vedlejším efektům.

něco jiného, dojde k chybě.

Je-li F něco jiného, hodnotou je F , vedlejší efekt není žádný.

Obrázek 1: Zjednodušený vyhodnocovací proces v Common Lispu

3. Common Lisp: základní vybava

Common Lisp je jazyk s velmi jednoduchou syntaxí, kterou jsme mohli v podstatě celou vysvětlit na několika stránkách předchozí části textu. Na druhé straně ovšem je Common Lisp objemná knihovna nástrojů na tvorbu rozsáhlých programů. V jazyce je 978 jmen speciálních operátorů, maker, funkcí a dalších symbolů, z nichž některé jsou podloženy hlubokými poznatky a principy. Abychom mohli při výkladu objektového programování Common Lisp používat jako účinný pomocný nástroj, musíme si jej alespoň do určité míry osvojit. Tato kapitola se věnuje vysvětlení části Common Lispu, která postačí ke zvládnutí základů objektového programování, probíraných v tomto textu.

V této kapitole představíme část ze zmíněných 978 symbolů. Některé z nich podrobně vysvětlíme, u jiných odkážeme čtenáře na definici ve standardu. U každého symbolu ze základní vybavy očekáváme, že o jeho existenci bude čtenář vědět a že jej bude schopen (s případným nahlédnutím do standardu) správně, rychle a účinně použít.

3.1. Řízení běhu

Základním operátorem, který slouží k podmíněnému vyhodnocení výrazů je speciální operátor `if`. Jeho zjednodušená syntax je následující:

```
(if test-form then-form else-form) => result
```

Speciální operátor `if` vyhodnocuje nejprve formu `test-form` a podle výsledku pak buď `then-form` nebo `else-form`: je-li hodnota `test-form` pravda, vyhodnotí formu `then-form` a jako výsledek vrátí její hodnotu, je-li nepravda, vyhodnotí `else-form` a vrátí její hodnotu.

Poznámka 3.1. V Common Lispu, na rozdíl od některých jiných (zejména procedurálních) programovacích jazyků, není rozdíl mezi *příkazem* a *výrazem*. Proto lze jak k podmíněnému vykonání příkazu, tak k podmíněnému získání hodnoty výrazu použít tentýž operátor `if` (například v jazyce C je nutno k prvnímu použít klíčové slovo `if`, ke druhému ternární operátor `?:`). Toto dvojí použití (které lze samozřejmě i kombinovat) ilustrujeme na jednoduchém příkladě: pokud bychom k číslu `x` chtěli přičíst číslo `1`, nebo `-1` v závislosti na tom, zda je číslo `y` nezáporné, nebo záporné, mohli bychom napsat

```
(if (>= y 0)
    (+ x 1)
    (+ x -1))
```

což by odpovídalo podmíněnému vykonání příkazu, nebo

```
(+ x (if (>= y 0) 1 -1))
```

což je podmíněné získání hodnoty. Pro úplnost uvedeme ještě jednu možnost, která se také může v některých situacích hodit:

```
(funcall (if (>= y 0) #' + #' -) x 1)
```

Užitečnými variantami speciálního operátoru `if` jsou makra `when` a `unless`. Jejich definice a příklady použití (včetně souvislosti se speciálním operátorem `if` a makrem `cond`) lze najít ve standardu.

V Common Lispu existuje mnoho maker umožňujících iterace (cykly) nejrůznějších typů. Jsou to například makra `dotimes` a `dolist`. Jejich použití je někdy pohodlnější než použití rekurze. První slouží k iteraci přes všechna čísla větší nebo rovna nule a menší než zadaná hodnota. Výraz

```
(dotimes (x 5)
  (print x))
```

vytiskne následujících pět řádků:

```
0
1
2
3
4
```

Makro `dolist` vytváří cyklus, v němž je daná proměnná navázána postupně na všechny prvky daného seznamu. Výraz

```
(dolist (x '(2 -1 3))
  (print x))
```

vytiskne

```
2
-1
3
```

Poznámka 3.2. Makra `dotimes` a `dolist` patří jednoznačně do imperativního programovacího stylu. Je nemožné vymyslet příklad využití těchto maker v jinak čistě funkcionálním kódu (proto jsme použili funkci `print`, která má vedlejší efekt — vytisknutí daného objektu). Tato makra ovšem lze definovat čistě funkcionálními prostředky. Hlubavější čtenář to může zkusit.

Poznámka 3.3. Funkci `print` vysvětlíme později.

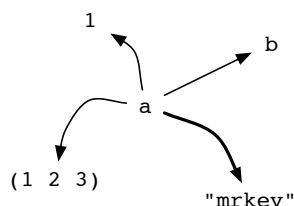
Speciální operátor `quote` i jeho zkratka `'` jsou čtenáři již dostatečně známy.

Symbody do základní výbavy: `if`, `when`, `unless`, `cond`, `dotimes`, `dolist`, `quote`

3.2. Proměnné a vazby

Každý symbol může být vybaven množstvím *hodnotových vazeb*, které jej svazují s libovolnými hodnotami (víme už, že symboly mají i funkční vazby, ty však pro nás nyní nejsou podstatné) — těmto vazbám se podle standardu říká *proměnné*, symbolu, kterému vazba přísluší, se říká *název* (nebo také *jméno*) proměnné. Někdy se také pro jednoduchost ale nepřesně ztotožňuje proměnná se svým symbolem, tj. se svým názvem.

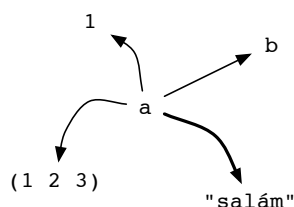
Ze všech hodnotových vazeb symbolu je aktivní vždy pouze jedna; ta určuje, jakou má příslušný symbol v daný moment hodnotu. O ostatních vazbách říkáme, že jsou aktuální vazbou *zastíněny*.



Obrázek 2: Vazby symbolu

Poznámka 3.4. Hodnotové vazby symbolu si můžeme představit jako provázky spojující tento symbol s různými hodnotami. Pokud je tedy například symbol `a` svázán se čtyřmi hodnotami — číslem 1, symbolem `b`, řetězcem `"mrkev"` a seznamem `(1 2 3)`, můžeme si tuto skutečnost představit tak, jak je uvedeno na Obrázku 2.

Pokud je tučně vyznačená vazba právě aktivní, bude aktuální hodnotou symbolu `a` řetězec `"mrkev"`. Říkáme také, že tento řetězec je *hodnotou* proměnné `a`. Kdybychom v tento moment změnili hodnotu proměnné `a` na něco jiného, řekněme na řetězec `"salám"` (což lze udělat voláním `(setf a "salám")`, jak uvidíme podrobněji později), vypadal by výsledek tak, jak vidíme na Obrázku 3.



Obrázek 3: Vazby symbolu po změně aktuální vazby

Hodnotové vazby mohou být dvou typů: *lexikální*, nebo *dynamické*. Tyto typy vazeb se liší okolnostmi, za kterých vazby vznikají, zanikají a za kterých jsou aktivní. Základním typem vazby v Common Lispu je vazba lexikální⁴, dynamické vazby se používají pouze za určitých okolností a my se s nimi v tomto textu nesetkáme.

Každá hodnotová vazba symbolu je vytvářena speciálními operátory `let` a `let*`, které popíšeme níže, a na začátku volání funkce, je-li symbol uveden jako její parametr. Každá nová vazba je vždy lexikální, vyjma případů, kdy programátor rozhodne jinak.

Lexikální vazba nikdy nezaniká, ale je aktivní pouze v části programu určené blokem textu ve zdrojovém kódu — buď částí těla operátoru `let` nebo `let*`, nebo částí volané funkce. Mimo tento blok textu není možno žádným způsobem hodnotu lexikální vazby získat nebo nastavit, a to ani v rámci kódu, který je z tohoto bloku textu volán.

Poznámka 3.5. Lokální proměnné ve většině programovacích jazyků jsou lexikální.

Základním nástrojem pro práci s vazbami je speciální operátor `let`, jehož zjednodušená syntax je následující:

```
(let ((var init-form)*) form*) => result
```

`var`: symbol

`init-form`: forma

`form`: forma

`result`: hodnota vrácená poslední formou `form`

⁴I když se v literatuře občas dočteme opak.

Speciální operátor `let` vytváří nové vazby symbolů `var` a inicializuje je na hodnoty, které získá postupným vyhodnocením forem `init-form`. Vytvořené vazby jsou obecně lexikální (výjimky zde nebudeme rozebírat). Vytvořené lexikální vazby, jak už bylo řečeno, nikdy nezanikají, ale jsou aktivní pouze v části kódu, která je označena jako `form*`. Výrazy `form*` se postupně vyhodnotí, výsledek posledního se vrátí jako výsledek celého `let`-výrazu.

Místo seznamu (`var init-form`) lze napsat pouze `var`. Význam je stejný, jako kdybychom napsali (`var nil`).

Příklad 3.6. Například:

```
CL-USER 1 > (let ((a t) b)
              (list a b))
(T NIL)
```

Jelikož nové vazby jsou aktivní uvnitř `let`-výrazu, dochází k zastínění případných vazeb vnějších:

```
CL-USER 2 > (let ((a 1))
              (cons (let ((a 2)) a) a))
(2 . 1)
```

V tomto případě vnitřní `let`-výraz vytvořil vazbu symbolu `a` na číslo 2, která zastínila vnější vazbu tohoto symbolu na číslo 1.

Poznámka 3.7. Při opakovaném vyhodnocení téhož `let`-výrazu se vytvářejí stále nové vazby na tytéž symboly a původní vazby zůstávají v platnosti. Tuto skutečnost ilustrujeme později, v odstavci věnovaném funkcím.

Speciální operátor `let*` má stejnou syntax jako operátor `let`. Liší se od něj tím, že nové vazby symbolů `var` nejsou aktivní pouze v oblasti `form*`, ale stanou se aktivními vždy ihned poté, co je získána hodnota příslušné `init-form`. Jinými slovy, při vyhodnocování každé `init-form` už jsou aktivní vazby všech předchozích `var`.

Příklad 3.8. Rozdíl mezi operátory `let` a `let*` demonstrujeme na příkladě:

```
CL-USER 3 > (let ((a 1))
              (let ((a 2)
                    (b a))
                b))
1
```

— v momentě, kdy byla inicializována vazba symbolu `b`, byla aktivní vnější vazba symbolu `a`.

```
CL-USER 4 > (let ((a 1))
              (let* ((a 2)
                     (b a))
                b))
2
```

— v momentě, kdy byla inicializována vazba symbolu `b`, už byla aktivní vnitřní vazba symbolu `a`.

Poznámka 3.9. Existují další způsoby, jak vytvořit novou vazbu. Kromě navazování parametrů na konkrétní hodnoty při volání funkce, které podrobněji rozebereme později, to je například u maker `dotimes` a `dolist`, která již byla probrána; v obou příkladech, na kterých jsme práci těchto maker ilustrovali, se vytváří nová vazba symbolu `x`.

O lexikálních vazbách souhrnně říkáme, že mají *lexikální rozsah platnosti*. Tento pojem můžeme použít i na jiné objekty než jsou vazby.

V každém okamžiku běhu programu je u každého symbolu aktivní nejvýše jedna vazba. Souhrn aktivních lexikálních vazeb se nazývá aktuální *lexikální prostředí*.

Poznámka 3.10. Prostedí si lze představit (a pravděpodobně tak bývá implementováno) jako tabulku se dvěma sloupci: jedním pro symbol, druhým pro jeho hodnotu.

Poznámka 3.11. Chceme-li místo vytváření nové vazby symbolu změnit hodnotu aktivní vazby, můžeme použít makro `setf`. K němu se dostaneme později.

V souvislosti s proměnnými je třeba zmínit ještě makro `defvar`. Jeho zjednodušená syntax je tato:

```
(defvar name [form]) => name
name: symbol form: forma
```

Makro `defvar` vytvoří zvláštní druh vazby symbolu `name`. Tato vazba má časově neomezenou platnost — nikdy nezaniká. Navíc je viditelná ze všech míst programu.

Poznámka 3.12. Proměnné definované makrem `defvar` mají tedy podobné vlastnosti jako tzv. globální proměnné v jiných programovacích jazycích.

Proměnným definovaným makrem `defvar` se říká *dynamické* (někdy též *speciální*) *proměnné*.

Je-li uvedena forma `form` a proměnná `name` dosud nemá žádnou hodnotu, forma `form` bude vyhodnocena a výsledek bude nastaven jako hodnota nově vzniklé vazby symbolu `name`. Má-li proměnná `name` hodnotu, forma `form` se vůbec nevyhodnotí.

Příklad 3.13. Demonstrace uvedené vlastnosti makra `defvar`:

```
CL-USER 5 > (defvar *x* 1)
*x*

CL-USER 6 > *x*
1

CL-USER 7 > (defvar *x* 2)
*x*

CL-USER 8 > *x*
1
```

Tento efekt makra `defvar` zjevně nemusí být vždy tím, který očekáváme. Pokud například vyhodnotíme výraz `(defvar *x* (fun))`, pak objevíme a opravíme chybu ve funkci `fun` a výraz znovu vyhodnotíme, bude proměnná `*x*` stále obsahovat původní nechtěnou hodnotu. Proto, pokud víme dopředu, že chceme, aby se vždy při načtení souboru s touto definicí hodnota proměnné `*x*` aktualizovala, použijeme místo uvedeného výrazu raději dvojici výrazů `(defvar *x*) (setf *x* (fun))`.

Makro `defvar` obsahuje jednu záludnost. Symboly, které jsou pomocí tohoto makra zavedeny jako dynamické proměnné, nemohou mít lexikální vazby. Pokud tyto symboly použijeme (například v `let`-výrazu) jako název nové proměnné, nevytvoří se lexikální, ale dynamická vazba. Abychom se vyhnuli komplikacím spojeným s používáním dynamických vazeb, zavedeme dvě pravidla pro používání dynamických proměnných:

Pravidla pro používání dynamických proměnných

1. Za názvy dynamických proměnných budeme volit pouze symboly začínající a končící hvězdičkou,
2. u těchto symbolů nebudeme nikdy vytvářet nové vazby.

Poznámka 3.14. Připomeňme si, jakým způsobem lze vytvářet nové vazby symbolů, abychom věděli, co všechno nám druhý bod zakazuje: k vytváření nových vazeb slouží operátory `let` a `let*`, nové vazby vznikají při volání funkcí tak, že se navážou parametry na argumenty (o tom viz níže v podkapitole o funkcích), makra `dotimes` a `dolist` také vytvářejí nové vazby.

Symboly do základní výbavy: `let`, `let`, `defvar`*

3.3. Místa

V Common Lispu se k vykonání vedlejšího efektu často používá makro `setf`. Například:

```
CL-USER 28 > (defvar *a*)
*a*

CL-USER 29 > (setf *a* (cons 0 2))
(0 . 2)

CL-USER 30 > (setf (car *a*) 1)
1

CL-USER 31 > *a*
(1 . 2)
```

Uvedená dvě volání makra `setf` tedy vykonávají rozdílné akce. První nastavuje hodnotu aktuální vazby symbolu `*a*`, druhé mění hodnotu `car` nějakého tečkového páru.

Příklad 3.15. Je důležité dobře pochopit, co znamená, že makro `setf` *nastavuje hodnotu aktuální vazby symbolu*. Začátečníci zvyklí programovat v procedurálních jazycích se často diví výsledku následujícího pokusu:

```
CL-USER 8 > (defvar *a*)
*a*

CL-USER 9 > (setf *a* 1)
1

CL-USER 10 > (defun set-a (a b)
               (setf a b))
```

```

SET-A

CL-USER 11 > (set-a *a* 2)
2

CL-USER 12 > *a*
1

```

Mnoho výrazů, jejichž vyhodnocení vede k získání obsahu nějakého místa v paměti, lze v kombinaci s makrem `setf` současně použít k modifikaci tohoto místa. Kromě výše uvedených dvou typů výrazů (výrazu `a` a výrazu `(car a)`) jsou to zejména výrazy, které pracují s položkami strukturovaných dat různých typů (vektorů, polí, párů, posloupností).

Výraz, který lze současně použít k získání hodnoty `a` a v kombinaci s makrem `setf` k jejímu nastavení, se nazývá *místo* (*place*). V Common Lispu je definováno mnoho typů míst, jejich výčet může zájemce najít v části 5.1.2 standardu. Kromě toho je ve standardu vždy u každého symbolu, který lze v kombinaci s makrem `setf` jako místo použít, tato skutečnost uvedena.

Poznámka 3.16. Zvídavější čtenáře zaujme, že Common Lisp poskytuje mechanismy, jak nové typy míst dodefinovat.

Makro `setf` použité uvedeným způsobem vždy vrací jako svůj výsledek nastavovanou hodnotu — tedy svůj druhý parametr.

Makro `setf` lze použít s více parametry k nastavení hodnot více míst současně. V takovém případě vrací vždy hodnotu posledního z těchto míst. Například:

```

CL-USER 12 > (defvar *a*)
*A*

CL-USER 13 > (setf *a* (cons 1 2))
(1 . 2)

CL-USER 14 > (setf (car *a*) 3 (cdr *a*) 4)
4

CL-USER 15 > *a*
(3 . 4)

```

Symbol do základní výbavy: `setf`

3.4. Funkce

Základní operací prováděnou s funkcemi je volání (aplikace) funkce s nějakými daty. Těmto datům se v Common Lispu říká *argumenty* funkce.

Příklad 3.17. Ve formě `(cons 1 2)` je tedy funkce `cons` volána s argumenty 1, 2, neboli seznam argumentů v tomto volání je `(1 2)`.

Pokud budeme dále hovořit o definici funkce, budeme používat termín *parametr*. Ten neoznačuje konkrétní data, se kterými je funkce volána, ale proměnnou, na kterou je v průběhu výkonu funkce některý z argumentů navázán.

Poznámka 3.18. V jiných jazycích se používá jiná terminologie, například formální a aktuální parametry.

Při definici nových funkcí (například pomocí operátorů `defun`, `lambda` a `labels`, které popíšeme za chvíli), je třeba uvést informaci o jejich parametrech. Tuto informaci uvádíme formou tzv. *obyčejného λ -seznamu*, kterým může v našem zjednodušeném případě být seznam symbolů, které se v nesmí opakovat.

Symbole obsažené v λ -seznamu se nazývají *parametry* dané funkce. Pokud λ -seznam funkce obsahuje n parametrů, musí být funkce volána právě s n argumenty. Při volání funkce se vytvoří nové lexikální vazby parametrů na pořadím jim odpovídající argumenty. Rozsah platnosti těchto vazeb je podobný jako u speciálních operátorů `let` a `let*` a zahrnuje celé tělo definované funkce.

Příklad 3.19. Funkce `car`, `cdr` a `cons` by tedy mohly mít následující λ -seznamy:

```
(x)
(x)
(x y)
```

Podívejme se nyní na to, jakými způsoby lze definovat nové funkce. Základním prostředkem je makro `defun`, jehož zjednodušená syntax je následující:

```
(defun function-name lambda-list form*)
  => function-name
function-name: symbol
ordinary-lambda-list: obyčejný  $\lambda$ -seznam
form: forma
```

Makro `defun` vytváří novou funkci jménem *function-name* (nastavuje tedy hodnotu funkční vazby symbolu *function-name* na tuto funkci), jejíž seznam parametrů je specifikován obyčejným λ -seznamem *lambda-list* a tělo se skládá z forem *form*. Při volání této funkce se vytvoří nové vazby parametrů λ -seznamu tak, jak bylo specifikováno výše, a ve vzniklém prostředí se postupně vyhodnotí všechny formy *form*. Hodnota poslední formy *form* bude hodnotou celého tohoto volání.

Příklad 3.20. Definice funkce na sečtení všech celých čísel od a do b :

```
(defun sum-numbers (a b)
  (* (+ a b) (- b a -1) 1/2))
```

Po vyhodnocení této definice pak tuto funkci můžeme volat jménem `sum-numbers`:

```
CL-USER 7 > (sum-numbers 1 10)
55
```

Poznámka 3.21. Z předchozí části již víme, že získat tuto funkci můžeme vyhodnocením výrazu `(function sum-numbers)`, což je ve zkratce `#'sum-numbers`.

K vytváření bezejmenných funkcí slouží operátor `lambda`. Zjednodušená syntax:

```
(lambda lambda-list form*) => result  
ordinary-lambda-list: obyčejný  $\lambda$ -seznam  
form: forma  
result: výsledná funkce
```

Operátor `lambda` vytváří novou funkci, jejíž seznam parametrů je specifikován obyčejným λ -seznamem *lambda-list* a tělo se skládá z forem *form*. Narozdíl od makra `defun` nestanovuje pro novou funkci žádné jméno, ale vrací ji jako svou hodnotu. Při volání této funkce se naváží všechny parametry λ -seznamu jak bylo specifikováno výše a pak se postupně vyhodnotí všechny formy *form*. Hodnota poslední formy *form* bude hodnotou celého tohoto volání.

Poznámka 3.22. Makro `defun` si tedy lze zhruba představit tak, že nejprve pomocí makra `lambda` vytvoří novou funkci a potom tuto funkci nějak uloží jako hodnotu funkční vazby symbolu `function-name`. Pokročilejší programátoři v Common Lispu ale vědí, že to není všechno, co makro `defun` dělá.

λ -výrazy stejné syntaxe lze uvést i na prvním místě vyhodnocovaného seznamu. V takovém případě se funkce definovaná tímto výrazem přímo zavolá — viz popis vyhodnocovacího procesu uvedený v předchozí části.

Příklad 3.23. Například výraz

```
((lambda (x) (+ x (if (< x 0) -1 1))) (fun))
```

přičte k výsledku volání funkce `fun` jedničku, pokud je nezáporný, jinak od něj jedničku odečte.

Speciální operátor `labels` lze použít k lokální definici funkcí. Tento operátor definuje funkce, které jsou platné pouze v určitém lexikálním rozsahu. Každá definice lokální funkce má stejnou syntax jako u makra `defun`. Vlastnosti definovaných funkcí jsou rovněž stejné jako u makra `defun`. Lexikální rozsah platnosti těchto definic zahrnuje celý `labels`-výraz. Proto se definované lokální funkce mohou vzájemně rekurzivně volat.

Příklad 3.24. Možná definice iterativní verze funkce na výpočet faktoriálu:

```
(defun fact (n)  
  (labels ((fact-iter (n accum)  
            (if (<= n 1)  
              accum  
              (fact-iter (- n 1) (* n accum))))))  
  (fact-iter n 1)))
```

Funkci definovanou lokálně speciálním operátorem `labels` lze pomocí jejího názvu získat stejně jako funkci definovanou globálně makrem `defun` — použitím speciálního operátoru `function`.

Obvyklý způsob volání funkce je použitím jejího názvu nebo λ -výrazu na prvním místě vyhodnocovaného seznamu (viz popis vyhodnocovacího procesu v předchozí části textu). Další možností je použít funkci `funcall` nebo `apply`, jejichž popis lze najít ve standardu.

Funkce vytvářené operátory `defun`, `lambda` a `labels` jsou *lexikální uzávěry* — veškeré volné lexikální proměnné použité v těle těchto funkcí se budou interpretovat v lexikálním prostředí, v němž byly použity, a to i v případě, že tyto funkce budou volány v jiném prostředí. Lexikální uzávěry tedy mají možnost číst a měnit hodnoty lexikálních vazeb, které byly aktivní, když byly tyto uzávěry definovány, a to i když už program výraz, který tyto vazby definoval (`let` nebo `let*-výraz`, nebo tělo volané funkce), opustil.

Ještě jinak řečeno: v těle lexikálního uzávěru tyto vazby opět aktivní budou, neboť toto tělo se textově nachází uvnitř bloku kódu, ve kterém aktivní jsou — vzpomeňme si, že lexikální vazby nikdy nezanikají.

Například v této definici

```
(defun test (x)
  (cons (lambda () x)
        (lambda (val) (setf x val)))))
```

každé volání funkce `test` vytváří novou lexikální vazbu symbolu `x` a každá z dvojic funkcí, které tímto voláním postupně vznikají, se tak prostřednictvím symbolu `x` odkazuje na jinou jeho vazbu:

```
CL-USER 9 > (setf a (test 1))
(#<anonymous interpreted function 2171C1B2> . #<anonymous interpreted function 2171C00A>)

CL-USER 10 > (setf b (test 2))
(#<anonymous interpreted function 216FA27A> . #<anonymous interpreted function 216FA0D2>)

CL-USER 11 > (funcall (car a))
1

CL-USER 12 > (funcall (car b))
2

CL-USER 13 > (funcall (cdr a) 3)
3

CL-USER 14 > (funcall (car a))
3

CL-USER 15 > (funcall (car b))
2
```

Symboly do základní výbavy: `defun`, `lambda`, `labels`, `function`, `funcall`, `apply`

3.5. Logické operace

O logických hodnotách v Common Lispu jsme se už zmínili. Ve všech operátorech, které pracují s logickými hodnotami, symbol `nil` reprezentuje hodnotu *nepřavda* a všechny ostatní objekty hodnotu *pravda*. Tak může například výsledek logických operací kromě informace o kladném výsledku přinášet také nějakou užitečnou hodnotu.

Funkce `not` neguje svůj argument. Mohla by být definována takto:

```
(defun not (x)
  (unless x t))
```

Makra `and` a `or` pracují podle pravidel *zkráceného vyhodnocování logických operací*, tj. nevyhodnocují všechny své argumenty, ale postupují od prvního tak dlouho, dokud nenarazí na hodnotu, která rozhodne o výsledku. Tuto hodnotu ihned vrátí a ve vyhodnocování dalších argumentů už nepokračují. U makra `and` rozhodne o výsledku první nepravdivá hodnota, u makra `or` první pravdivá. Pokud makro `and` nebo `or` dojde ve vyhodnocování až na konec seznamu argumentů, vrátí hodnotu posledního z nich.

Poznámka 3.25. Makra `and` a `or` lze tedy používat i jako operátory řídící běh programu. Takto by například mohla být implementována funkce `find-true`, která najde první prvek daného seznamu, který není roven `nil`:

```
(defun find-true (list)
  (and list
    (or (car list)
        (find-true (cdr list)))))
```

Poznámka 3.26. V předchozí kapitole jsme upozornili na to, že názvy predikátů, tj. funkcí, které vracejí (zobecněnou) logickou hodnotu je obvyklé psát s příponou „p“ nebo „-p“.

Symbolsy do základní výbavy: `and`, `or`, `not`

3.6. Porovnávání

Univerzální predikát na porovnávání dat neexistuje. Vždy záleží na účelu, ke kterému data používáme.

Poznámka 3.27. Představa, že by mohl existovat jeden univerzální porovnávací predikát (tj. funkce, která rozhoduje, zda jsou objekty „stejně“ nebo ne), je scestná. Predikát na porovnávání seznamů bude jistě vypadat jinak, než predikát na porovnávání seznamů, reprezentujících množiny (tj. seznamů, u nichž nám nezáleží na pořadí prvků).

Příklad 3.28. Pokud programujeme čistě funkcionálně, můžeme seznamy vytvořené dvojím voláním výrazu `(list 1 2)` považovat za totožné. Pokud ale pracujeme s vedlejším efektem, už to obecně nelze:

```
CL-USER 16 > (setf a (list 1 2))
(1 2)

CL-USER 17 > (setf b (list 1 2))
(1 2)

CL-USER 18 > (setf (car a) 3)
3

CL-USER 19 > a
(3 2)

CL-USER 20 > b
(1 2)
```

Příklad 3.29. Čísla 1 a 1.0 můžeme z matematického hlediska považovat za sobě rovná, z implementačního hlediska ale sobě jistě rovná nejsou — jsou různých typů, zabírají různé místo v paměti, pro práci s nimi se používají různé instrukce procesoru atd.

V Common Lispu jsou zavedeny čtyři univerzální porovnávací predikáty. My budeme využívat dva z nich: predikát `eq1` a predikát `equalp`. Oba přijímají dva argumenty a oba vracejí hodnotu *pravda*, pokud si tyto argumenty jsou v jistém smyslu rovny.

Funkce `eq1`, zjednodušeně řečeno, zkoumá, zda jsou zadané objekty totožné podle těch nejprísnejších pravidel, jaká mají v rámci Common Lispu smysl. U objektů shodných podle funkce `eq1` se můžeme spolehnout na to, že zůstanou shodné, ať s nimi budeme dělat cokoli.

Příklad 3.30. Například:

```
CL-USER 21 > (eq1 'a 'a)
T

CL-USER 22 > (eq1 1 1)
T

CL-USER 23 > (eq1 (list 1 2) (list 1 2))
NIL

CL-USER 24 > (eq1 1 1.0)
NIL
```

Funkce `equalp` naopak považuje dva objekty za stejné, pokud mají, opět zhruba řečeno, stejný obsah. Nerozlišuje také malá a velká písmena v řetězcích. Pokud jsou dva objekty `equalp`, jsou také určitě `eq1`.

Příklad:

```
CL-USER 25 > (equalp (list 1 2) (list 1 2))
T

CL-USER 26 > (equalp "ahoj" "AHOJ")
T

CL-USER 27 > (equalp 'ahoj "AHOJ")
NIL

CL-USER 28 > (equalp 1 1.0)
T
```

Přesný popis funkcí `eq1` a `equalp` je ve standardu.

Symboły do základní výbavy: `eq1`, `equalp`

3.7. Čísla

Základní funkce pro práci s čísly jsou následující:

Predikáty `=`, `/=`, `<=`, `>=` slouží k porovnávání dvou a více čísel. Funkce `+`, `-`, `*`, `/` implementují základní aritmetické operace. Akceptují jeden (v případě funkcí `+` a `*` dokonce žádný) a více argumentů.

K dispozici je mnoho různých reálných funkcí, například `min`, `max`, `abs`, `signum`, `sin`, `cos`, `tan`, `exp`, `expt`, `sqrt`, `log` a dalších. Konstanta `pi` obsahuje číslo π .

Celočíselné dělení a zaokrouhlování provádějí například funkce `floor`, `ceiling`, `round`, `truncate` (různé typy celočíselného dělení spojené se zaokrouhlováním — vracejí vždy dvě hodnoty, podíl a zbytek; v tomto textu se ale druhými a dalšími hodnotami funkcí nezabýváme) a `mod`, `rem` (dva typy zbytku po dělení).

Symbole do základní výbavy: `=`, `/=`, `<=`, `>=`, `+`, `-`, `*`, `/`, `min`, `max`, `abs`, `signum`, `sin`, `cos`, `tan`, `exp`, `expt`, `sqrt`, `log`, `pi`, `floor`, `ceiling`, `round`, `truncate`, `mod`, `rem`

3.8. Páry a seznamy

Základní funkce pracující s tečkovými páry — v Common Lispu se pro ně používá spíše název *cons* — čtenář i čtenářka již většinou zná: funkce `cons` vytváří nový pár, funkce `car` a `cdr` získávají hodnoty jeho složek. Hodnoty `(car nil)` a `(cdr nil)` jsou navíc definovány jako `nil` — při jejich získávání tedy nedojde k chybě.

Poznámka 3.31. Funkce `car` a `cdr` tedy nepracují jen s páry, ale se všemi seznamy včetně prázdného. Dejme si na tuto okolnost pozor, někdy může vést k nepříjemným chybám.

Funkce `car` a `cdr` definují místa, takže složky párů lze modifikovat pomocí makra `setf`. Pokus o nastavení `car` nebo `cdr` symbolu `nil` ovšem vede k chybě.

Funkce, které zjišťují hodnoty složek zřetězených párů jsou `caar`, `cadr`, `cdar`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdaar`, `cdadr`, `cdar`, `cddr`, `caaaar`, `caaaadr`, `caaaar`, `caaaadr`, `caaaar`, `caaaadr`, `caaaar`, `caaaadr`, `caaaar`, `caaaadr`, `caaaar`, `caaaadr`. Všechny tyto funkce akceptují jako parametr i symbol `nil` a definují místa obdobně jako funkce `car` a `cdr`.

K získávání hodnot obecných složek zřetězených párů slouží funkce `nth` a `nthcdr`. Funkce `nth` a `nthcdr` akceptují jako parametr i symbol `nil`. Funkce `nth` navíc definuje místo.

Seznam je v Common Lispu libovolný pár nebo symbol `nil` (který reprezentuje prázdný seznam). Mezi seznamy se tedy počítají i tzv. *tečkované seznamy*, což jsou všechny seznamy `x`, pro něž některá z hodnot `(nthcdr n x)` není seznam. Objekt

```
(1 2 3 4 5 . 10)
```

je tedy tečkovaný seznam. Tečkované seznamy zde zmiňujeme pouze pro úplnost, v dalším se s nimi nesetkáme. Totéž platí i o tzv. kruhových seznamech.

Funkce `list` vytváří nové seznamy. Funkce `copy-list` kopíruje daný seznam. Funkce `append` spojuje libovolný počet seznamů. Funkce `last` vrací konec daného seznamu zadané délky, funkce `butlast` začátek. K jednotlivým prvkům seznamu lze také přistupovat pomocí funkcí `first`, `second`, `third`, `fourth`, `fifth`, `sixth`, `seventh`, `eighth`, `ninth`, `tenth`, které také definují místo. Funkce `null` testuje, zda je daný objekt nulový seznam (je tedy ekvivalentní funkci `not`; volba je věcí stylu).

Funkce `reverse` slouží k obrácení seznamů.

Funkce `mapcar` je základní funkce, která aplikuje danou funkci na všechny prvky seznamu a shromažďuje výsledky volání do nového seznamu:

```
CL-USER 10 > (mapcar (lambda (x) (+ x 1))
                     '(0 1 2 3))
(1 2 3 4)
```

Pokud zadanou funkci lze volat s více argumenty, můžeme funkci `mapcar` zadat více seznamů:

```
CL-USER 12 > (mapcar #'cons '(a b c) '(1 2 3))
((A . 1) (B . 2) (C . 3))

CL-USER 13 > (mapcar #'+ '(1 2 3) '(4 5 6) '(7 8 9))
(12 15 18)
```

Příklad 3.32. Skalární součin vektorů `u` a `v` reprezentovaných seznamy:

```
(apply #'+ (mapcar #'* u v))
```

Pokud bychom matici reprezentovali seznamem řádků, z nichž každý by byl reprezentován seznamem, vypadal by součin matice `M` a vektoru `v` takto:

```
(mapcar (lambda (row)
          (apply #'+ (mapcar #'* row v)))
        M)
```

Funkce `find` a `find-if` rozhodují, zda je daný prvek, nebo prvek s danou vlastností přítomen v zadaném seznamu. Zjednodušená syntax:

```
(find element list) => result
(find-if predicate list) => result

element: libovolný objekt
predicate: funkce
list: seznam
result: nalezený prvek nebo nil
```

Funkce `find` vrátí *element*, pokud jej najde v seznamu *list*. Pokud ne, vrátí `nil`. Příklad:

```
CL-USER 1 > (find 2 '(1 2 3))
2

CL-USER 2 > (find 4 '(1 2 3))
NIL
```

Funkce `find` používá k porovnávání prvku *element* a prvků seznamu *list* funkci `eql`. Proto:

```
CL-USER 3 > (find (cons 1 2) '((1 . 2) (3 . 4)))
NIL
```

Funkce `find-if` vrátí první prvek seznamu *list* takový, že když se na něj aplikuje funkce *predicate*, výsledek aplikace bude *pravda*. Pokud takový prvek v seznamu *list* nenajde, vrátí `nil`. Příklad:

```
CL-USER 4 > (find-if (lambda (x)
                      (> x 4))
                  '(2 4 6 8))
6

CL-USER 5 > (find-if (lambda (x)
                      (< x 2))
                  '(2 4 6 8))
NIL
```

Funkce `remove` a `remove-if` slouží k odstranění prvků ze seznamu. Příklad:

```
(remove 2 '(1 2 3)) => (1 3)
(remove-if (lambda (x) (> x 5))
           '(1 4 7 10 6 2))
=> (1 4 2)
```

Funkce `length` zjišťuje délku seznamu. Funkce `every` testuje, zda všechny prvky posloupnosti vyhovují danému predikátu. Podobně jako například funkce `mapcar` je také schopna pracovat s predikáty, které přijímají více argumentů a více posloupnostmi.

Symbolsy do základní výbavy: `cons`, `car`, `cdr`, `caar`, `cadr`, `cdar`, `cddr`, `caaar`, `caadr`, `cadar`, `caddr`, `cdaar`, `cdadr`, `cddar`, `cdddr`, `caaaar`, `caaadr`, `caadar`, `caaddr`, `cadaar`, `cadadr`, `caddar`, `cadddr`, `cdaaar`, `cdaadr`, `cdadar`, `cdaddr`, `cddaar`, `cddadr`, `cdddar`, `cddddr`, `nth`, `nthcdr`, `list`, `copy-list`, `append`, `last`, `butlast`, `first`, `second`, `third`, `fourth`, `fifth`, `sixth`, `seventh`, `eighth`, `ninth`, `tenth`, `reverse`, `mapcar`, `find`, `find-if`, `remove`, `remove-if`, `length`, `every`

3.9. Chyby

Každý program se může dostat do situace, se kterou jeho autor dopředu nepočítal a která vyžaduje zásah uživatele (ať už jiné části programu nebo člověka). Takovýmto stavům se říká *vyjíměčné stavy* a dochází k nim hlavně v důsledku nějaké chyby (programu, operačního systému, disku apod.). Jsou to stavy, u kterých je nevhodné, aby program bez informace z vnějšku pokračoval v práci. V našem textu budeme používat jednu funkci, která signalizuje, že k vyjíměčnému stavu došlo. Je to funkce `error` a má následující zjednodušenou syntax:

```
(error string)
```

Tato funkce signalizuje chybu, jejíž popis je v řetězci *string*. Současně dojde k předčasnému zastavení běhu programu.

Příklad 3.33. Bezpečná funkce na výpočet faktoriálu:

```
(defun fact (n)
  (unless (and (typep n 'integer) (>= n 0))
    (error "Factorial needs a non-negative integer as its
argument."))
  (labels ((fact-iter (n accum)
    (if (<= n 1)
      accum
      (fact-iter (- n 1) (* n accum)))))
    (fact-iter n 1)))
```

(volání `(typep n 'integer)` zjišťuje, zda je hodnota proměnné `n` celé číslo; více v odstavci o typech).

Test:

```
CL-USER 1 > (fact -1)

Error: Factorial needs a non-negative integer as its argument.
1 (abort) Return to level 0.
2 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed, or :?
for other options

CL-USER 2 : 1 >
```

3.10. Textový výstup

K textovému výstupu slouží zejména funkce `print` a `format`. Jednodušší je funkce `print`, která při volání s jedním argumentem tento argument vytiskne do standardního výstupu (v `LispWorks` je jím buď okno `Listeneru`, pokud jsme funkci zavolali z něj, nebo záložka `Output`). Například vyhodnocením výrazu

```
(print (list (cons 1 2) (cons 1 2)))
```

vytiskneme

```
((1 . 2) (1 . 2))
```

Funkce také objekt vrací jako svůj výsledek, takže v případě tisku do okna `Listeneru` jej uvidíme vytisknutý dvakrát.

Funkce `format` se používá k vytištění formátovaného textu. Jako parametr se jí uvádí tištěný řetězec, v němž je možno použít nejrůznější direktivy uvedené vždy znakem `~`. Nás budou zajímat direktivy `~s` a `~a`, pomocí nichž lze do řetězce umístit (a tedy vytisknout) libovolný objekt, jenž je použit jako parametr funkce `format`, a direktiva `~%`.

Direktiva `~s` tiskne objekty tak, aby byl výsledek použitelný ve zdrojovém textu programu (stejně jako u funkce `print`), tedy včetně znaků indikujících typ objektů, jako

jsou například uvozovky u řetězců (*Simple Print*). Direktiva `~a` je tiskne přijatelněji pro oko, bez těchto znaků (*Aesthetic Print*). Direktiva `~%` způsobí přechod na další řádek (ten je ale možno také zajistit odřádkováním přímo v řetězci).

První argument funkce `format` určuje cíl tisku. Pokud jako tento argument použijeme symbol `t`, bude funkce tisknout do standardního výstupu, použijeme-li symbol `nil`, funkce vytištěný text vrátí ve formě řetězce jako svou hodnotu.

Příklad 3.34. Volání

```
(format t "~%List ~s and string ~s" (list 1 1) "Ahoj")
```

přejde ve standardním výstupu na nový řádek a vytiskne

```
List (1 1) and string "Ahoj"
```

(slovo "Ahoj" je v uvozovkách).

Příklad 3.35. Volání

```
(let ((n -1))
  (format nil
    "Number ~s is ~a"
    n
    (if (>= n 0)
      "non-negative"
      "negative")))
```

vrátí jako výsledek řetězec

```
"Number -1 is negative"
```

(slovo "negative" není v uvozovkách).

Obě uvedené funkce umožňují také textový výstup do souborů i jinam, tyto možnosti jsou ale do určité míry implementačně závislé a nejsou pro tento text podstatné.

Symbolsy do základní výbavy: `print`, `format`

3.11. Typy

V Common Lispu je propracovaný systém typů. *Typ* je obecně definován jako libovolná množina objektů a každý typ lze nějakým způsobem označit. V objektovém programování budeme potřebovat pouze základní typy. Kromě již uvedených jsou to například typy `symbol`, `function`, `cons`, `number`, `integer`, `string`. Typ `t` je definován jako typ, který obsahuje všechny objekty, typ `nil` je prázdný. Typ `null` obsahuje pouze symbol `nil`.

Poznámka 3.36. Tak dostává symbol `nil` další roli. Kromě symbolu, prázdného seznamu a hodnoty *nepravda* ještě označuje typ.

Funkce `typep` rozhoduje, zda je daný objekt daného typu:

```
CL-USER 1 > (typep (lambda (x) (+ x 1)) 'function)
T

CL-USER 2 > (typep "abc" 'string)
T

CL-USER 3 > (setf type 'number)
NUMBER

CL-USER 4 > (typep 10 type)
T

CL-USER 5 > (typep 10 nil)
NIL
```

Symbol do základní výbavy: `typep`

3.12. Prostředí

V této podkapitole uvedeme pouze tři symboly užitečné při interaktivní práci s Common Lispem: `*`, `**`, `***`. V příkazovém řádku lze využívat proměnných `*`, `**`, `***` k odkazování na předchozí výsledky. Například:

```
CL-USER 1 > (+ 2 3)
5

CL-USER 2 > (+ 4 5)
9

CL-USER 3 > (* * **)
45
```

Symbole do základní výbavy: ``, `**`, `***`*

Poslední změny

- 10. 3. 2008 Přidána poznámka o způsobu psaní predikátů do odstavce o logických operacích. Přidány funkce `remove`, `remove-if`, `length` do odstavce o seznamech.

4. Třídy a objekty

Pojetí pojmu objekt se v různých programovacích jazycích liší. V tomto textu budeme vycházet z představy, kterou zavedl SmallTalk a která je dodnes v objektových jazycích více či méně dodržována.

Objekt je ucelený souhrn dat (*abstraktní datová struktura*), který je charakterizován třemi základními vlastnostmi: zapouzdřením, polymorfismem a dědičností. Těmito pojmy se budeme zabývat v následujících částech, nyní uvedeme pouze základní definice.

Každý objekt obsahuje kromě svých dat i kód, který s těmito daty pracuje. Tomuto kódu se říká *metody*. Metoda je zvláštní druh funkce, která se spustí, kdykoliv chce uživatel vykonat s objektem nějakou akci. Každý objekt může obsahovat více metod, podle toho, jaké akce s ním lze provádět.

Spouštění metod se děje pomocí mechanismu *zasílání zpráv*. Každá metoda objektu má své jméno; pokud chceme nějakou metodu zavolat, pošleme objektu zprávu téhož jména. Pokud tedy zašleme objektu zprávu A, systém mezi jeho metodami nalezne metodu A a spustí ji. Této metodě se říká *obsluha zprávy A*, procesu zavolání obsluhy dané zprávy se říká její *obsloužení*.

Zpráva zasílaná objektu může obsahovat argumenty podobně, jako může argumenty obsahovat volání funkce. Obsluha zprávy (metoda) má pak tyto argumenty k dispozici (podobně jako funkce). Obsluha zprávy také může vracet hodnotu jako svůj výsledek.

Poznámka 4.1. V Common Lispu je na rozdíl od této užší definice zvykem nazývat objektem jakákoli data. V případě, že by mohlo dojít k nedorozumění, je tedy někdy třeba blíže specifikovat, v jakém smyslu zrovna o objektech hovoříme. Proto budeme někdy používat zpřesňující termíny, jako *objekt Common Lispu (lispový objekt)*, *objekt ve smyslu objektového programování* a podobně.

Poznámka 4.2. Každé zaslání zprávy, které změní vnitřní stav objektu, znamená vedlejší efekt. V samém srdci objektového programování tak stojí princip vedlejšího efektu, který je v přímém rozporu se zásadami funkcionálního programování.

4.1. Třídy

Data v objektech jsou (podobně jako u struktur v jazyce C) rozdělena do jednotlivých pojmenovaných položek. V Common Lispu se těmto položkám říká *sloty*. Každý slot objektu má své (v rámci objektu) jedinečné jméno.

Třída je, ve své jednodušší podobě, popis objektu. Obsahuje jednak seznam názvů slotů objektu a jednak definici všech jeho metod. Při běhu programu slouží třída jako předloha k vytváření nových objektů. Objekt, jehož popisem je daná třída, se nazývá *přímou instancí* této třídy.

Poznámka 4.3. Uvedenou definici třídy v následujících kapitolách ještě rozšíříme. Mimo jiné definujeme pojem instance třídy, který je obecnější než pojem přímé instance. Pokud v následujícím textu použijeme pojem instance, vztahuje se to k tomuto obecnějšímu pojmu (a tedy i k pojmu přímé instance).

Poznámka 4.4. Zopakujme si, v čem jsou shodné a v čem se mohou lišit dvě přímé instance téže třídy:

1. Dvě přímé instance téže třídy obsahují stejnou sadu slotů, tedy mají stejný počet slotů stejných názvů. Hodnoty těchto slotů však mohou být různé.
2. Dvě přímé instance téže třídy obsahují stejné metody.

Třída je datový typ. Ve staticky typovaných programovacích jazycích lze třídy používat ke specifikaci typů proměnných, v dynamicky typovaných programovacích jazycích pak lze na třídy použít nástroje k dynamickému zjišťování typu dat (v Common Lispu například funkci `typep`). Tyto možnosti jsou ovšem k dispozici i ve staticky typovaných jazycích (konstrukce “is”), do kterých tak objektové programování vnáší prvky dynamického typování.

4.2. Třídy a instance v Common Lispu

Podívejme se, jak jsou obecné pojmy z předchozích podkapitol realizovány v Common Lispu. Nové třídy se definují pomocí makra `defclass`, které specifikuje seznam slotů třídy, a pomocí makra `defmethod`, které slouží k definici metod instancí třídy.

Nové objekty se vytvářejí pomocí funkce `make-instance`. Ke čtení hodnoty slotu objektu slouží funkce s názvem `slot-value`, který lze v kombinaci s výrazem `setf` použít i k nastavování hodnot slotů.

Zprávy se v Common Lispu objektům zasílají pomocí stejné syntaxe, jakou se v tomto jazyce volají funkce.

Zjednodušená syntax makra `defclass` je následující:

```
(defclass name () slots)

name: symbol (nevyhodnocuje se)
slots: seznam symbolů (nevyhodnocuje se)
```

Symbol *name* je název nově definované třídy, symboly ze seznamu *slots* jsou názvy slotů této třídy.

Poznámka 4.5. Prázdný seznam za symbolem *name* je součástí zjednodušené syntaxe. V dalších kapitolách, až se dozvíme více o třídách, ukážeme, co lze použít místo něj.

Příklad 4.6. Definice třídy `point`, jejíž instance by obsahovaly dva sloty s názvy *x* a *y* by vypadala takto:

```
(defclass point ()
  (x y))
```

Definovali-li jsme novou třídu (zatím pouze třídu bez metod, k jejichž definici se dostaneme vzápětí), měli bychom se naučit vytvářet její přímé instance. V Common Lispu k tomu používáme funkci `make-instance`, jejíž zjednodušená syntax je tato:

```
(make-instance class-name)


class-name: symbol
```

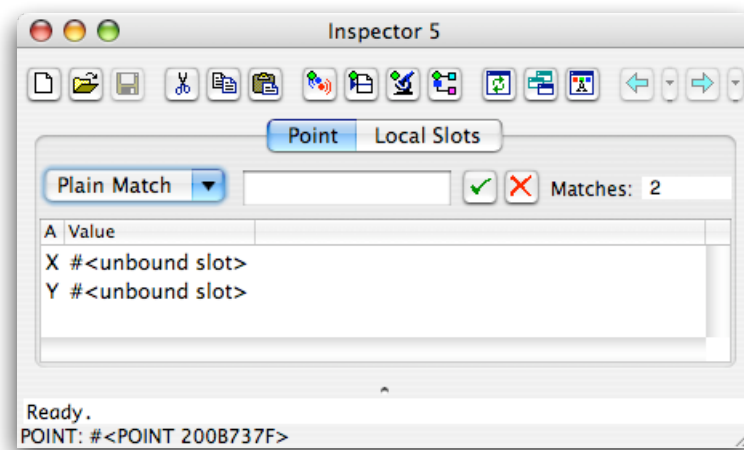
Funkce `make-instance` vytvoří a vrátí novou přímou instanci třídy, jejíž jméno najde ve svém prvním parametru. Všechny sloty nově vytvořeného objektu jsou neiniciovány a každý pokus získat jejich hodnotu skončí chybou (později si řekneme, jak se získávají hodnoty slotů a pak to budeme moci vyzkoušet).

Příklad 4.7. Pokud jsme definovali třídu `point` tak, jak je uvedeno v předchozím příkladě, můžeme nyní vyzkoušet vytvoření její instance:

```
CL-USER 2 > (make-instance 'point)
#<POINT 200DC0D7>
```

Poznámka 4.8. Pokud není v tento moment čtenáři jasné, proč jsme ve výrazu `(make-instance 'point)` symbol `point` kvotovali, měl by si uvědomit, že `make-instance` je funkce a zopakovat si základy vyhodnocovacího procesu v Common Lispu.

Výsledek volání není příliš čitelný, v prostředí LispWorks si jej ale můžeme prohlédnout v inspektoru. Pokud v Listeneru klikneme na tlačítko s mikroskopem () , objeví se okno obsahující údaje o posledním výsledku, jak je vidět na Obrázku 4.



Obrázek 4: Neinicializovaná instance třídy `point` v inspektoru

Text `#<unbound slot>` u názvů jednotlivých slotů znamená, že sloty jsou neinicializované. Můžeme jim ale pomocí prostředí LispWorks zkusit nastavit hodnotu. Klikneme-li na některý ze zobrazených slotů pravým tlačítkem, můžeme si v objevivší se nabídce vybrat volbu “Slots->Set...” tak, jak je znázorněno na Obrázku 5 a novou hodnotu slotu nastavit.

K programovému čtení hodnot slotů slouží funkce `slot-value`, k jejich nastavování symbol `slot-value` v kombinaci s makrem `setf`. Syntax je následující:

```
(slot-value object slot-name)
```

object: objekt

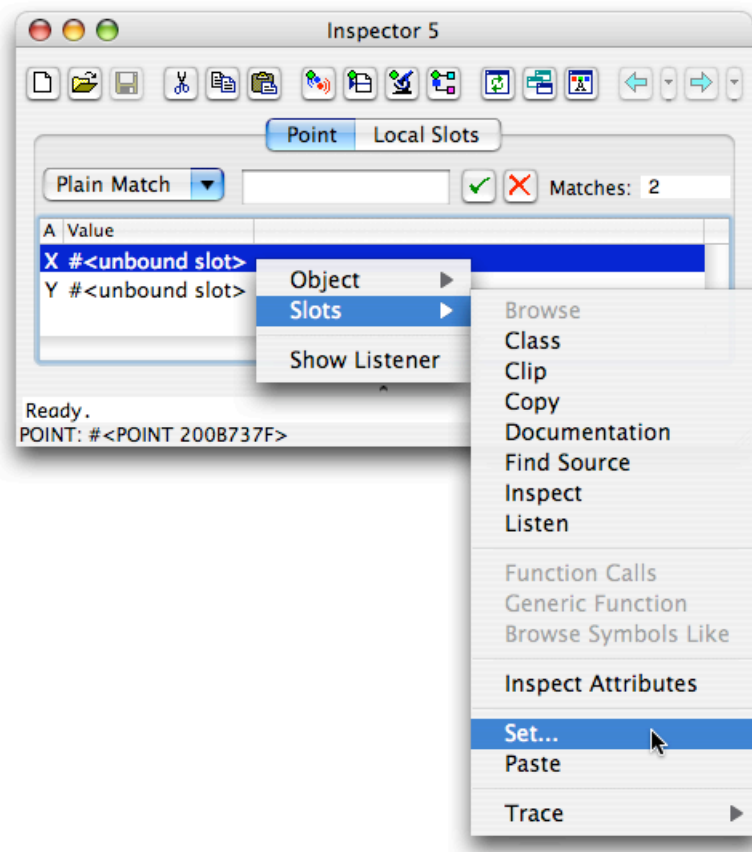
slot-name: symbol

Příklad 4.9 (práce s funkcí `slot-value`). Vyzkoušejme si práci s funkcí `slot-value`. Nejprve vytvořme instanci již definované třídy `point` a uložíme ji do proměnné `pt`:

```
CL-USER 2 > (setf pt (make-instance 'point))
#<POINT 216C0213>
```

Poznámka 4.10. Použití proměnné, kterou jsme dříve nedefinovali (jako v tomto případě proměnné `pt`), je povoleno pouze k experimentálním účelům v příkazovém řádku. Na jiných místech je vývojové prostředí nepovoluje. Každou proměnnou je třeba buď definovat jako lexikální (například pomocí speciálního operátoru `let`), nebo jako dynamickou (makrem `defvar`).

Nyní zkusme získat hodnotu slotu `x` nově vytvořené instance:



Obrázek 5: Nastavování hodnoty slotu v inspektoru

```
CL-USER 3 > (slot-value pt 'x)

Error: The slot X is unbound in the object #<POINT 216C0213>
(an instance of class #<STANDARD-CLASS POINT 200972AB>).
  1 (continue) Try reading slot X again.
  2 Specify a value to use this time for slot X.
  3 Specify a value to set slot X to.
  4 (abort) Return to level 0.
  5 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed, or :?
for other options
```

Vidíme, že došlo k chybě; slot x není v nově vytvořeném objektu inicializován. Z chybového stavu se dostaneme napsáním `:a` a zkusíme hodnotu slotu nejprve nastavit:

```
CL-USER 4 : 1 > :a

CL-USER 5 > (setf (slot-value pt 'x) 10)
10
```

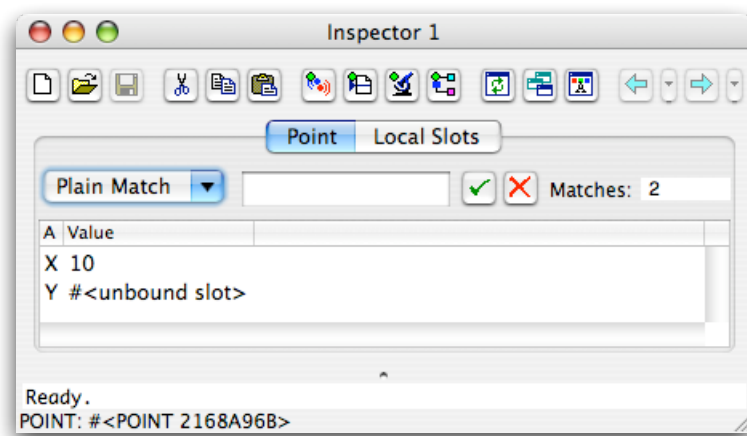
Nyní již funkce `slot-value` chybu nevyvolá a vrátí nastavenou hodnotu:

```
CL-USER 6 > (slot-value pt 'x)  
10
```

Nové hodnoty slotů lze také ověřit pomocí inspektoru. Získejme nejprve obsah proměnné `pt`:

```
CL-USER 7 > pt  
#<POINT 216C0213>
```

a stiskneme tlačítko s mikroskopem. Výsledek vidíme na Obrázku 6.



Obrázek 6: Instance třídy `point` po změně hodnoty slotu `x`

Zbývá vysvětlit, jak se v Common Lispu definují metody objektů. Jak jsme již řekli, všechny přímé instance jedné třídy mají stejnou sadu metod. Metody jsou zvláštní druh funkce, proto se definují podobně. V Common Lispu je k definici metod připraveno makro `defmethod`. Jeho syntaxe (ve zjednodušené podobě, jak ji uvádíme na tomto místě), je stejná jako u makra `defun` s tou výjimkou, že u prvního parametru je třeba specifikovat jeho třídu. Tím se metoda definuje pro všechny instance této třídy.

```
(defmethod message ((object class) arg1 arg2 ...)  
  expr1  
  expr2  
  ... )  
  
message: symbol  
object: symbol  
class: symbol  
arg1: symbol  
expri: výraz
```

Symbol `class` určuje třídu, pro jejíž instance metodu definujeme, symbol `message` současně název nové metody i název zprávy, kterou tato metoda obsluhuje. Výrazy `expr1`, `expr2` atd. tvoří *tělo metody*, které stejně jako tělo funkce definuje kód, který se provádí, když je metoda spuštěna. Symbol `object` je během vykonávání těla metody navázán na objekt, jemuž byla zpráva poslána, symboly `arg1`, `arg2`, atd. na další argumenty, se kterými byla zpráva zaslána.

Jak již bylo řečeno, syntax zaslání zprávy je stejná jako syntax volání funkce:

```
(message object arg1 arg2 ...)  
  
message: symbol  
object: výraz  
argi: výraz
```

Symbol *message* musí být názvem zprávy, kterou lze zaslat objektu vzniklému vyhodnocením výrazu *object*. Zpráva je objektu zaslána s argumenty, vzniklými vyhodnocením výrazů *arg1*, *arg2* atd. Stejně jako u volání funkce jsou výrazy *object*, *arg1*, *arg2* atd. vyhodnoceny postupně zleva doprava.

Příklad 4.11. Řekněme, že potřebujeme zjišťovat polární souřadnice bodů. Správný objektový přístup řešení této úlohy je následující: definovat nové zprávy, které budeme bodům k získání těchto informací zasílat.

Definujme tedy nové metody pro třídu *point*: metodu *r*, která bude vracet vzdálenost bodu od počátku (první složku jeho polárních souřadnic), a metodu *phi*, která bude vracet odchylku spojnice bodu a počátku od osy *x* (tedy druhou složku polárních souřadnic bodu)

Metoda *r* počítá vzdálenost bodu od počátku pomocí Pythagorovy věty:

```
(defmethod r ((point point))  
  (let ((x (slot-value point 'x))  
        (y (slot-value point 'y)))  
    (sqrt (+ (* x x) (* y y)))))
```

Poznámka 4.12. Pokud nechápete, co znamená *(point point)* v této definici, podívejte se znovu na syntax makra *defmethod*. Zjistíte, že první položkou tohoto seznamu je symbol, na nějž bude při vykonávání těla metody navázán příjemce zprávy, zatímco druhou položkou je název třídy, pro jejíž instance metodu definujeme. Že jsou obě tyto hodnoty stejné, nevádí, v Common Lispu mohou být názvy proměnných a tříd stejné.

Po zaslání zprávy *r* bodu bychom tedy měli obdržet jeho vzdálenost od počátku. Vytvořme si na zkoušku instanci třídy *point* a nastavme jí hodnoty slotů *x* a *y* na 3 a 4:

```
CL-USER 8 > (setf pt (make-instance 'point))  
#<POINT 200BC6A3>  
  
CL-USER 9 > (setf (slot-value pt 'x) 3  
                  (slot-value pt 'y) 4)  
4
```

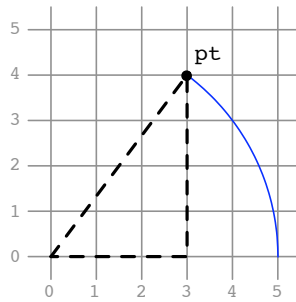
Vytvořený objekt reprezentuje geometrický bod, který je znázorněn na Obrázku 7.

Nyní zkusme získat vzdálenost tohoto bodu od počátku zasláním zprávy *r* naší instanci (připomeňme, že zprávy se objektům zasílají stejnou syntaxí jakou se volají funkce, tedy výraz *(r pt)* znamená zaslání zprávy *r* objektu *pt*):

```
CL-USER 10 > (r pt)  
5.0
```

Tento výsledek by měl být správně, jelikož $\sqrt{3^2 + 4^2} = 5$.

Podobně definujme metodu *phi* (pochopení vyžaduje trochu matematických znalostí):



Obrázek 7: Bod o souřadnicích (3, 4)

```
(defmethod phi ((point point))
  (let ((x (slot-value point 'x))
        (y (slot-value point 'y)))
    (cond ((> x 0) (atan (/ y x)))
          ((< x 0) (+ pi (atan (/ y x)))))
    (t (* (signum y) (/ pi 2))))))
```

Další zkouška:

```
CL-USER 11 > (phi pt)
0.9272952
```

Tangens tohoto úhlu by měl být roven 4/3 (viz Obrázek 7):

```
CL-USER 12 > (tan *)
1.3333333
```

Pro úplnost ještě definujeme metody pro nastavení polárních souřadnic bodu. Narozdíl od předchozích budou tyto metody vyžadovat zadání argumentů. Vzhledem k tomu, že každá z nich mění obě kartézské souřadnice bodu současně, bude užitečné napsat nejprve metodu pro současné nastavení obou polárních souřadnic.

```
(defmethod set-r-phi ((point point) r phi)
  (setf (slot-value point 'x) (* r (cos phi))
        (slot-value point 'y) (* r (sin phi)))
  point)
```

Metody `set-r` a `set-phi` tuto metodu využijí (přesněji řečeno, zprávu `set-r-phi` zasílají):

```
(defmethod set-r ((point point) value)
  (set-r-phi point value (phi point)))

(defmethod set-phi ((point point) value)
  (set-r-phi point (r point) value))
```

Poznámka 4.13. Metody `set-r-phi`, `set-r` a `set-phi` vracejí vždy jako výsledek parametr `point`. Tento přístup budeme volit ve všech metodách, které mění stav objektu: vždy budeme jako výsledek vracet měněný objekt. Důvodem je, aby šlo objektu měnit více hodnot v jednom výrazu:

```
(set-r (set-phi pt pi) 1)
```

Nyní můžeme instancím třídy `point` posílat zprávy `set-r-phi`, `set-r` a `set-phi` a měnit tak jejich polární souřadnice. Vyzkoušejme to tak, že našemu bodu `pt` pošleme zprávu `set-phi` s argumentem `0`. Tím bychom měli zachovat jeho vzdálenost od počátku, ale odchylka od osy x by měla být nulová.

Zaslání zprávy `set-phi` s argumentem `0`:

```
CL-USER 13 > (set-phi pt 0)
#<POINT 200BC6A3>
```

Test polohy transformovaného bodu:

```
CL-USER 14 > (slot-value pt 'x)
5.0

CL-USER 15 > (slot-value pt 'y)
0.0
```

Výsledek je tedy podle očekávání (nová poloha bodu je na druhém konci modrého oblouku na Obrázku 7).

4.3. Inicializace slotů

Ukažme si ještě jednu možnost makra `defclass`. V předchozích odstavcích jsme si všimli, že když vytvoříme novou instanci třídy, jsou všechny její sloty neinicializované a při pokusu o získání jejich hodnoty před jejím nastavením dojde k chybě. To se někdy nemusí hodit. Proto makro `defclass` stanovuje možnost, jak specifikovat počáteční hodnotu slotů nově vytvářené instance.

V obecnější podobě makra `defclass` je jeho syntax následující:

```
(defclass name () slots)

name: symbol (nevyhodnocuje se)
slots: seznam (nevyhodnocuje se)
```

Prvky seznamu `slots` mohou být buď symboly, nebo seznamy. Je-li prvkem tohoto seznamu symbol, je jeho význam takový, jak již bylo řečeno, tedy specifikuje název slotu instancí třídy, který není při vzniku nové instance inicializován. Je-li prvkem tohoto seznamu seznam, musí být jeho tvar následující:

```
(slot-name :initform expr)

slot-name: symbol
expr: výraz
```

V tomto případě specifikuje symbol `slot-name` název definovaného slotu. Výraz `expr` je vyhodnocen pokaždé při vytváření nové instance třídy a jeho hodnota je do příslušného slotu instance uložena.

Příklad 4.14. Upravme definici třídy `point` tak, aby byly sloty `x` a `y` nových instancí inicializovány na hodnotu 0:

```
(defclass point ()
  ((x :initform 0)
   (y :initform 0)))
```

Jak můžeme snadno zkusit, sloty nových instancí jsou nyní inicializovány:

```
CL-USER 1 > (setf pt (make-instance 'point))
#<POINT 20095117>

CL-USER 2 > (list (slot-value pt 'x) (slot-value pt 'y))
(0 0)
```

Příklad 4.15. Nyní definujeme další třídu, jejíž instance budou reprezentovat geometrické útvary. Bude to třída `circle`. Jak známo, geometrie každého kruhu je určena jeho počátkem a poloměrem. Proto budou mít instance této třídy dva sloty. Slot `center`, který bude obsahovat instanci třídy `point` a slot `radius`, který bude obsahovat číslo. Každý z těchto slotů bude při vytvoření nové instance automaticky inicializován.

```
(defclass circle ()
  ((center :initform (make-instance 'point))
   (radius :initform 1)))
```

Teď již necháme na čtenáři, aby si sám zkusil vytvořit novou instanci této třídy a prohlédl její sloty.

Příklad 4.16. Pokud pošleme zprávu objektu, který pro ni nemá definovanou metodu (obsahu této zprávy), dojde k chybě. Můžeme si to ukázat tak, že pošleme zprávu `phi` instanci třídy `circle`:

```
CL-USER 3 > (phi (make-instance 'circle))

Error: No applicable methods for #<STANDARD-GENERIC-FUNCTION
PHI 21694CFA> with args (#<CIRCLE 216C3CF3>)
  1 (continue) Call #<STANDARD-GENERIC-FUNCTION PHI 21694CFA>
again
  2 (abort) Return to level 0.
  3 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed, or :?
for other options
```

V tomto hlášení o chybě je třeba všimnout si hlavně textu „No applicable methods“, který znamená, že jsme poslali zprávu objektu, který pro ni nemá definovanou obsluhu (metodu).

Vzhledem k tomu, že syntax zasílání zpráv je v Common Lispu stejná jako syntax volání funkce či aplikace jiného operátoru, nemohou se zprávy jmenovat stejně jako funkce, makra, nebo speciální operátory. Proto následující definice vyvolá chybu (`set` je funkce Common Lispu):

```
CL-USER 5 > (defmethod set ((point point) coord value)
              (cond ((eql coord 'x) (set-x point value))
                    ((eql coord 'y) (set-y point value))))

Error: SET is defined as an ordinary function #<Function SET
202D54A2>
  1 (continue) Discard existing definition and create generic
function
  2 (abort) Return to level 0.
  3 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed, or :?
for other options
```

Pokud bychom se pokusili poslat objektu zprávu, pro niž jsme nedefinovali metodu pro žádnou třídu, Common Lisp vůbec nepochopí, že se snažíme poslat zprávu, a bude volání interpretovat jako použití neexistujícího operátoru:

```
CL-USER 7 > (pho (make-instance 'circle))

Error: Undefined operator PHO in form (PHO (MAKE-INSTANCE
(QUOTE CIRCLE))).
  1 (continue) Try invoking PHO again.
  2 Return some values from the form (PHO (MAKE-INSTANCE (QUOTE
CIRCLE))).
  3 Try invoking something other than PHO with the same argu-
ments.
  4 Set the symbol-function of PHO to another function.
  5 Set the macro-function of PHO to another function.
  6 (abort) Return to level 0.
  7 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed, or :?
for other options
```

5. Zapouzdření

5.1. Motivace

Začneme několika motivačními příklady.

Příklad 5.1. Vytvořme nejprve novou instanci třídy `point`:

```
CL-USER 1 > (setf pt (make-instance 'point))
#<POINT 21817363>
```

a předpokládejme, že na nějakém místě programu omylem nastavíme hodnotu slotu `x` této instance na `nil`:

```
CL-USER 2 > (setf (slot-value pt 'x) nil)
NIL
```

Po nějaké době, na jiném místě našeho programu, pošleme objektu `pt` zprávu `r`.

Poznámka 5.2. Než budete číst dál, zkuste odhadnout, co přesně se stane a proč.

Ano, dojde k chybě:

```
CL-USER 3 > (r pt)

Error: In * of (NIL NIL) arguments should be of type NUMBER.
  1 (continue) Return a value to use.
  2 Supply new arguments to use.
  3 (abort) Return to level 0.
  4 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed, or :?
for other options
```

Poznámka 5.3. Než budeme pokračovat ve výkladu o zapouzdření, uděláme malou odbočku. Většina čtenářů tohoto textu stráví s prostředím `LispWorks` nejméně jeden semestr. Stojí tedy za to věnovat nějaký čas bližšímu seznámení s ním. Tento čas se vám mnohonásobně v budoucnu vrátí.


Během studia tohoto textu a programování v `Common Lispu` bude jistě každý dělat mnoho chyb. Je tedy účelné naučit se efektivně prostředí `LispWorks` v těchto situacích používat.

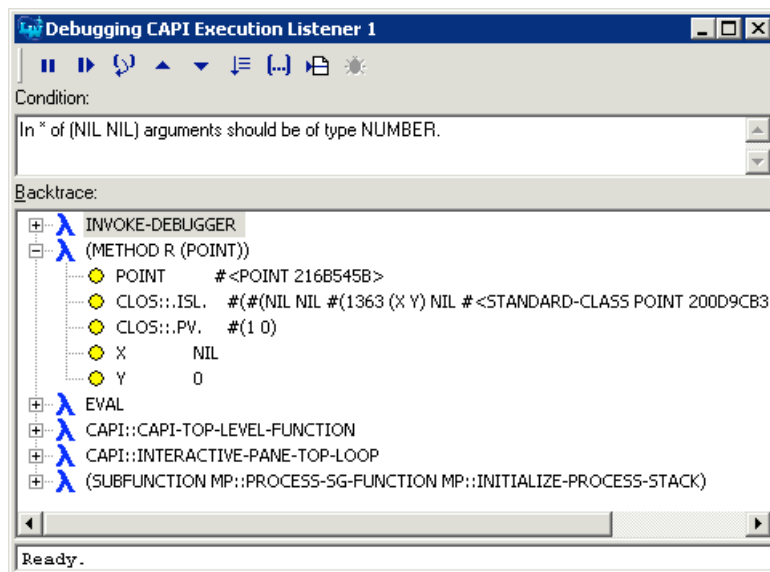
Příklad 5.4. Předchozí příklad skončil chybovým stavem, k němuž došlo po zaslání zprávy `r` objektu `pt`. Hlášení o chybě, „`In * of (NIL NIL) arguments should be of type NUMBER`“, je třeba číst takto: *při volání funkce `*` s argumenty `(NIL NIL)` došlo k chybě, protože argumenty volání nejsou čísla*. Jinými slovy, pokoušíme se násobit symbol `nil`.

Nyní máme několik možností, co ve vzniklém chybovém stavu dělat. Především se z něj můžeme dostat napsáním `:a`. Tím se chybový stav ukončí a výkon programu definitivně zastaví. To by ale byla škoda, protože tím bychom opustili prostředí, které je nachystáno k tomu, aby nám pomohlo chybu v programu nalézt.

V chybovém stavu nám prostředí nabízí očíslovaný seznam akcí, které můžeme podniknout a (případně) spustit program od místa, kde se zastavil. Tuto možnost oceníme

zejména při práci s většími programy, proto ji tady uvedeme pouze pro úplnost: V našem případě jsou zajímavé hlavně akce 1 a 2, po jejichž volbě (napsáním `:c 1` nebo `:c 2`) budeme moci buď zadat jiný výsledek funkce `*` nebo jiné argumenty, se kterými pak bude opět zavolána.

Užitečnější pro začátečníky je možnost pomocí grafického uživatelského rozhraní LispWorks využít chybové situace k hledání příčiny chyby. V situaci, ve které se nyní nacházíme, je totiž stále k dispozici zásobník pozastaveného procesu s informacemi o volaných funkcích, jejich argumentech a lokálních proměnných. Tento zásobník si můžeme prohlédnout v debuggeru (ladiči) po stisknutí tlačítka s beruškou () . Okno debuggeru je vidět na obrázku 8.



Obrázek 8: Výpis zásobníku v debuggeru s detailem metody `r`

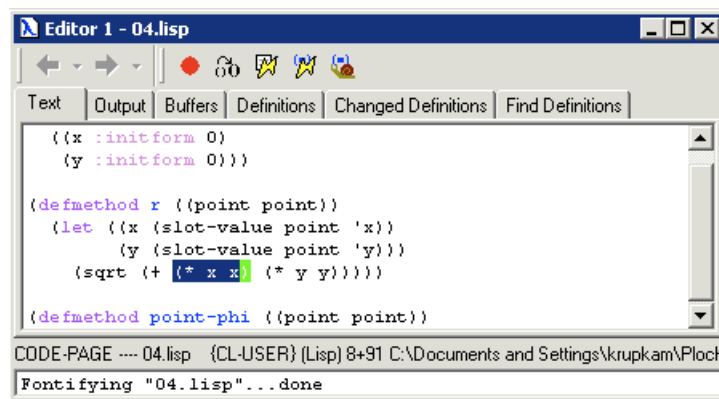
V hlavní části okna je zobrazen aktuální zásobník volání funkcí. Vespod tohoto zásobníku najdeme funkce, které spustil systém poté, co jsme stisknutím klávesy Enter zadali vyhodnocení výrazu. Jako poslední z těchto funkcí vidíme funkci `eval`, která již způsobila zaslání zprávy `r` a spuštění příslušné metody. Tu vidíme hned nad funkcí `eval` v detailnějším zobrazení. Na vrcholu zásobníku je funkce `invoke-debugger`, kterou spustil systém, když došlo k chybě, a která pozastavila vykonávání kódu.

V detailnějším zobrazení volání metody `r` vidíme parametry metody (v našem případě jediný, parametr `point`) s jejich hodnotami a lokální proměnné (ty jsou dvě, `x` a `y`, a jsou vytvořeny speciálním operátorem `let` v našem zdrojovém kódu metody) opět s jejich hodnotami. Další lokální proměnné (v tomto případě `CLOS::ISL` a `CLOS::PV`) přidal do metody systém a my si jich nemusíme všimnout.

Vidíme například, že lokální proměnná `x` má hodnotu `nil`, což bylo zdrojem vzniklé chyby. Také si můžeme všimnout, že hodnotou parametru `point` je nějaká instance třídy `point`. Pokud se chceme podívat podrobněji jaká, můžeme na řádek s parametrem `point` zaklikat a zobrazit si ji v inspektoru (ten už známe, proto tady není zobrazen), který nám mimo jiné ukáže, že instance má ve slotu `x` nepřípustnou hodnotu `nil`.

Pokud zaklikáme na řádek metody `r` samotné, systém najde její zdrojový kód a přesně označí místo, kde došlo k chybě. To je vidět na Obrázku 9.

Tímto způsobem nám prostředí LispWorks umožňuje najít místo v programu, kde došlo k chybě i odhalit její bezprostřední příčinu.



Obrázek 9: Místo v metodě `r`, kde došlo k chybě

Příklad 5.5. Příčinou chyby tedy bylo nesprávné nastavení slotu `x` objektu `pt` na hodnotu `nil`. Kdy a kde k němu ale došlo? V tom nám už debugger neporadí; mohlo to být na libovolném místě programu, ve kterém slot `x` nastavujeme, o několik řádků výše, nebo o hodinu dříve. Tady může začít hledání, které může u velkých a příliš spleťtých programů trvat hodně dlouho.

Porovnejme tuto situaci s následující. Nejprve ale opravme náš objekt `pt` a nastavme jeho slot `x` na nějakou přípustnou hodnotu:

```
CL-USER 4 > (setf (slot-value pt 'x) 1)
1
```

Teď se pokusme udělat podobnou chybu s tím rozdílem, že nyní nastavíme na `nil` jednu z polárních souřadnic bodu `pt`, řekněme úhel.

Poznámka 5.6. Co se stane?

```
CL-USER 5 > (set-phi pt nil)
```

```
Error: In COS of (NIL) arguments should be of type NUMBER.
```

- 1 (continue) Return a value to use.
- 2 Supply a new argument.
- 3 (abort) Return to level 0.
- 4 Return to top loop level 0.

```
Type :b for backtrace, :c <option number> to proceed, or :?
for other options
```

Chybové hlášení sice není příliš srozumitelné (i když jeho smysl už chápeme — došlo k pokusu počítat kosinus z hodnoty `nil`), ale pomocí debuggeru jsme schopni velmi rychle odhalit příčinu chyby, kterou je zaslání zprávy `set-phi` s nepřípustnou hodnotou argumentu.

Rozdíl mezi uvedenými dvěma případy: zatímco v tomto jsme byli upozorněni na problém v momentě, kdy jsme jej vytvářeli, v předchozím proběhlo nastavení nepřípustné hodnoty bez povšimnutí a upozornění jsme byli až na druhotnou chybu — tedy chybu způsobenou předchozí chybou. Tento rozdíl může způsobit ztrátu několika hodin, ba dokonce dní programátorovy práce, o vlivu na jeho nervovou soustavu nemluvě.

Přitom, z pohledu zvenčí by člověk řekl, že koncepčně není mezi uvedenými případy podstatný rozdíl; koneckonců, ať se jedná o kartézské nebo polární, vždy jsou to prostě jen souřadnice. Rozdíl je v tom, jak jsou tyto případy implementovány (v jednom se hodnota nastavuje přímo, ve druhém zasíláním zprávy a nějakým výpočtem). Chyba je tedy na straně programátora.

Závěr: není dobré umožnit uživateli nastavovat hodnoty slotů v objektech bez kontroly jejich konzistence.

Poznámka 5.7. Lze oprávněně namítnout, že v silně staticky typovaném jazyce by uvedený problém vůbec nenastal, protože kompilátor by nedovolil do proměnné číselného typu uložit nečíselnou hodnotu. To je sice pravda, ale pro příklad, kdy by uložit nepřípustnou hodnotu do slotu umožnil i kompilátor staticky typovaného jazyka, nemusíme chodit daleko. Například u instancí třídy `circle` jsou (logicky) jako hodnoty slotu `radius` povolena pouze nezáporná čísla. Pokus uložit do tohoto slotu v průběhu programu vypočítané záporné číslo ale žádný kompilátor neodhalí.

Příklad 5.8. Předpokládejme, že jsme se (ať už z jakýchkoliv důvodů) rozhodli změnit reprezentaci geometrických bodů tak, že místo kartézských souřadnic budeme ukládat souřadnice polární. Změněná definice třídy `point` by vypadala takto:

```
(defclass point ()
  ((r :initform 0)
   (phi :initform 0)))
```

Položme si otázku: Co všechno bude nutno změnit v programu, který tuto třídu používá? Odpověď je poměrně jednoduchá. Pokud změníme definici metod `r`, `phi` a `set-r-phi` následujícím způsobem:

```
(defmethod r ((point point))
  (slot-value point 'r))

(defmethod phi ((point point))
  (slot-value point 'phi))

(defmethod set-r-phi ((point point) r phi)
  (setf (slot-value point 'r) r
        (slot-value point 'phi) phi)
  point)
```

nebudeme muset v programu měnit už žádné jiné místo, ve kterém pracujeme s polárními souřadnicemi bodů. Horší to bude s kartézskými souřadnicemi. S těmi jsme dosud pracovali pomocí funkce `slot-value`. Všechna místa programu, na kterých je napsáno něco jako jeden z těchto čtyř výrazů:

```
(slot-value pt 'x)
(slot-value pt 'y)
(setf (slot-value pt 'x) value)
(setf (slot-value pt 'y) value)
```

bude třeba změnit. Budeme tedy muset projít celý (možná že dost velký) zdrojový kód programu a všude, kde to bude potřeba, provést příslušnou úpravu.

Závěr: pokud používáme k práci s objekty mechanismus zasílání zpráv a nepřistupujeme k jejich vnitřním datům přímo, je náš program lépe připraven na změny vnitřní reprezentace dat.

Příklad 5.9. Představme si, že píšeme uživatelskou dokumentaci ke třídám `point` a `circle`, které jsme zatím naprogramovali. Při našem současném řešení bychom museli v dokumentaci popisovat zvlášť práci s kartézskými a polárními souřadnicemi bodů, protože s každými se pracuje jinak: Kartézské souřadnice se zjišťují pomocí funkce `slot-value` s druhým parametrem rovným symbolu `x` nebo `y`, zatímco ke čtení souřadnic polárních používáme zprávy `r` a `phi`.

Podobně, ke čtení poloměru kružnice používáme funkci `slot-value` s druhým parametrem rovným symbolu `radius`. Výsledkem by bylo, že by si uživatel musel pamatovat dvojí způsob práce s našimi objekty, což by mělo obvyklé nepříjemné důsledky (musel by častěji otvírat dokumentaci, asi by dělal více chyb a podobně), které by se zmnohonásobily u většího programu (který by neobsahoval dvě třídy, ale sto, které by obsahovaly mnohem více slotů a metod atd.).

V našem případě si musí uživatel našich tříd pamatovat, která data se čerpají přímo ze slotů a která jsou vypočítaná a získávají se zasláním zprávy. Když odhlédneme od toho, že tato vlastnost dat se může časem změnit (viz předchozí příklad), nutíme uživatele, aby se zabýval detaily, které pro něj nejsou podstatné.

Závěr: Při návrhu třídy je třeba myslet na jejího uživatele a práci mu pokud možno co nejvíce zpříjemnit a usnadnit.

5.2. Princip zapouzdření

Uvedené tři příklady nás motivují k pravidlu zvanému *princip zapouzdření*, které budeme vždy důsledně dodržovat.

Princip zapouzdření

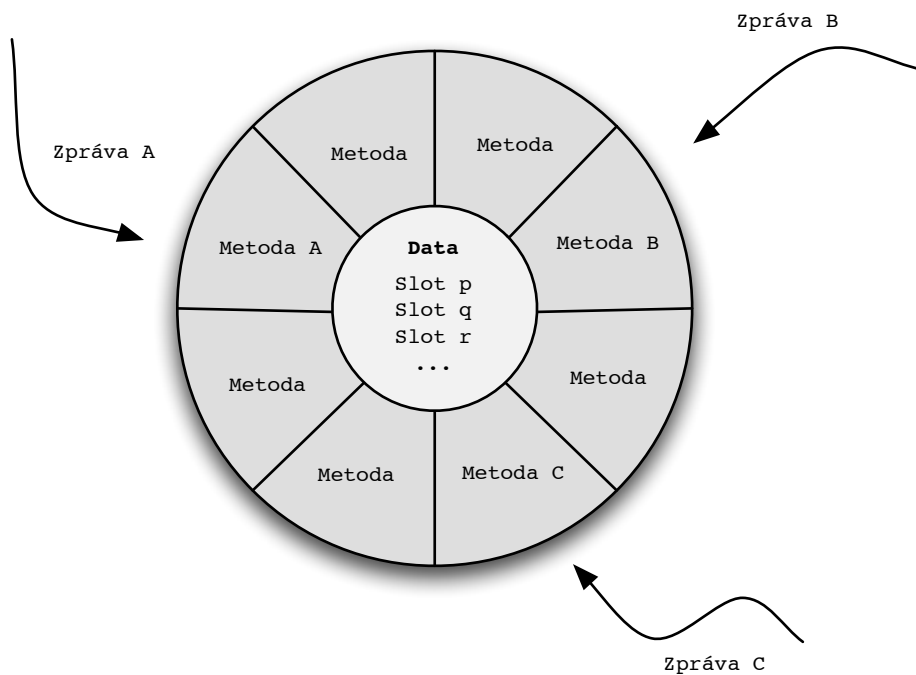
Hodnoty slotů objektu smí přímo číst a měnit pouze metody tohoto objektu. Ostatní kód smí k těmto hodnotám přistupovat pouze prostřednictvím zpráv objektu zasílaných.

K principu zapouzdření nás motivovaly tři základní důvody. Zopakujme si je:

1. Uživateli je třeba zabránit modifikovat vnitřní data objektu, protože by jej mohl uvést do nekonzistentního stavu.
2. Změna vnitřní reprezentace dat objektu by měla uživateli přinášet co nejmenší komplikace.
3. Rozhraní objektů by mělo být co nejjednodušší a nejsnadněji použitelné. Uživatelé nezajímají implementační detaily.

Poznámka 5.10. Jedná se o konkrétní realizaci obecnějšího principu, který je třeba dodržovat ve všech programovacích stylech, jakmile je program rozdělen na moduly, nikoliv pouze v objektovém programování. Tento princip bývá obecně nazýván principem *oddělení rozhraní a implementace* (v jiných souvislostech se také hovoří o *abstraktních bariérách* či *vrstevné architektuře programu*).

Data uvnitř objektu lze tedy měnit pouze pomocí zpráv objektu zasílaných. Momentální hodnota dat v objektu je také nazývána jeho *vnitřním stavem*. Princip zapouzdření je znázorněn na Obrázku 10.



Obrázek 10: Objekt

5.3. Úprava tříd `point` a `circle`

Příklad 5.11. Upravme definici třídy `point` tak, aby odpovídala principu zapouzdření (metody pracující s polárními souřadnicemi zde neuvádíme, protože žádnou změnu nevyžadují):

```
(defclass point ()
  ((x :initform 0)
   (y :initform 0)))

(defmethod x ((point point))
  (slot-value point 'x))

(defmethod y ((point point))
  (slot-value point 'y))

(defmethod set-x ((point point) value)
  (unless (typep value 'number)
    (error "x coordinate of a point should be a number"))
  (setf (slot-value point 'x) value)
  point)

(defmethod set-y ((point point) value)
  (unless (typep value 'number)
    (error "y coordinate of a point should be a number"))
  (setf (slot-value point 'y) value)
  point)
```

Příklad 5.12. Upravme také třídu `circle`. U slotu `radius` budeme postupovat stejně jako u slotů `x` a `y` třídy `point`. Zveřejníme jeho obsah tím, že definujeme zprávy pro čtení a zápis jeho hodnoty:


```
(defmethod radius ((c circle))
  (slot-value c 'radius))

(defmethod set-radius ((c circle) value)
  (when (< value 0)
    (error "Circle radius should be a non-negative number"))
  (setf (slot-value c 'radius) value)
  c)
```

U slotu center je zbytečné, aby mohl uživatel jeho hodnotu nastavovat; proto mu dáme pouze možnost ji číst:

```
(defmethod center ((c circle))
  (slot-value c 'center))
```

Různé změny u kruhu tak bude možné dělat pomocí jeho středu, například nastavit polohu:

```
CL-USER 9 > (make-instance 'circle)
#<CIRCLE 200CB05B>

CL-USER 10 > (set-x (center *) 10)
10
```

5.4. Třída picture

Další využití zapouzdření ukážeme na příkladě třídy picture. Smysl této třídy je podobný jako smysl příkazu „Group“ v různých grafických aplikacích. Je to nástroj, který umožňuje spojit několik grafických objektů do jednoho a usnadnit tak operace prováděné na všech těchto objektech současně.

Příklad 5.13. Instance třídy picture budou obsahovat seznam podřízených grafických objektů. Definice třídy by tedy mohla vypadat takto:

```
(defclass picture ()
  ((items :initform '())))
```

Metody ke čtení a nastavování hodnoty slotu items:

```
(defmethod items ((pic picture))
  (slot-value pic 'items))

(defmethod set-items ((pic picture) value)
  (unless (every (lambda (elem)
                    (or (typep elem 'point)
                        (typep elem 'circle)
                        (typep elem 'picture))))
    value)
    (error "Picture elements are not of the desired type."))
  (setf (slot-value pic 'items) value)
  pic)
```

Metoda `set-items` podobně jako předtím například metoda `set-radius` třídy `circle` nejprve testuje zda jsou nastavovaná data konzistentní, v tomto případě, zda jsou všechny prvky seznamu `items` správného typu (že proměnná `items` obsahuje seznam, otestuje funkce `every` — můžete vyzkoušet sami):

```
CL-USER 1 > (setf list (list 0 0 0))
(0 0 0)

CL-USER 2 > (setf pic (make-instance 'picture))
#<PICTURE 200E83CB>

CL-USER 3 > (set-items pic list)

Error: Picture elements are not of desired type.
1 (abort) Return to level 0.
2 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed, or :?
for other options
```

Pokud z této chybové hlášky nepochopíme o jakou chybu jde, můžeme, jak bylo ukázáno dříve, spustit ladění kliknutím na tlačítko s beruškou.

Poznámka 5.14. Metodu `set-items` bude třeba přepracovat kdykoliv definujeme novou třídu grafických objektů. To je jistě nešikovné. Nápravu sjednáme později, v Kapitole 7.

Takto definovaná třída `picture` bude fungovat správně, ale nebude ještě dostatečně odolná vůči uživatelským chybám. Uživatel má stále ještě možnost narušit konzistenci dat jejích instancí.

Poznámka 5.15. Jak? Přijďte na to sami?

Pokračujme tedy v příkladě. Vložme do proměnné `list` seznam složený z grafických objektů a uložíme jej jako seznam prvků již vytvořenému obrázku `pic`:

```
CL-USER 5 > (setf list (list (make-instance 'point) (make-
instance 'circle)))
(#<POINT 2008A1DB> #<CIRCLE 2008A1BF>)

CL-USER 6 > (set-items pic list)
(#<POINT 2008A1DB> #<CIRCLE 2008A1BF>)
```

Jaké jsou prvky obrázku `pic`?

```
CL-USER 7 > (items pic)
(#<POINT 2008A1DB> #<CIRCLE 2008A1BF>)
```

To nás jistě nepřekvapí. Nyní upravme seznam `list`:

```
CL-USER 8 > (setf (first list) 0)
0
```

Co bude nyní tento seznam obsahovat?

```
CL-USER 9 > list
(0 #<CIRCLE 2008A1BF>)
```

A co bude nyní v seznamu prvků obrázku `pic`?

```
CL-USER 10 > (items pic)
(0 #<CIRCLE 2008A1BF>)
```

Poznámka 5.16. Jste schopni vysvětlit, co se stalo? Než budete pokračovat dále, je to třeba pochopit.

Obrázek `pic` nyní obsahuje nekonzistentní data. Abychom problém odstranili, změníme metodu `items` tak, aby nevracela seznam uložený ve slotu `items`, ale jeho kopii. Podobně, při nastavování hodnoty slotu `items` v metodě `set-items` do slotu uložíme kopii uživatelem zadaného seznamu. Tímto dvojím opatřením zaříkáme, že uživatel nebude mít k seznamu v tomto slotu přístup, pouze k jeho prvkům.

```
(defmethod items ((pic picture))
  (copy-list (slot-value pic 'items)))

(defmethod set-items ((pic picture) value)
  (unless (every (lambda (elem)
                  (or (typep elem 'point)
                      (typep elem 'circle)
                      (typep elem 'picture))))
    value)
    (error "Picture elements are not of desired type."))
  (setf (slot-value pic 'items) (copy-list value))
  pic)
```

Poznámka 5.17. Není třeba dodávat, že pokud uživatel nedodrží princip zapouzdření, budou tato bezpečnostní opatření neúčinná.

Několik testů:

```
CL-USER 12 > (setf list (list (make-instance 'point) (make-
instance 'circle)))
(#<POINT 200D7A23> #<CIRCLE 200D7A07>)

CL-USER 13 > (set-items pic list)
(#<POINT 200D7A23> #<CIRCLE 200D7A07>)

CL-USER 14 > (setf (first list) 0)
0

CL-USER 15 > (setf (second (items pic)) 0)
0

CL-USER 16 > (items pic)
(#<POINT 200D7A23> #<CIRCLE 200D7A07>)
```

Vidíme, že nyní je všechno v pořádku — ani následná editace seznamu posílaného jako parametr zprávy `set-items`, ani editace seznamu vráceného zprávou `items` nenaruší vnitřní data objektu `pic`.

Poznámka 5.18. Bezpečnost kódu by měla být pro programátora prioritou. Proto jsme si v našem programu dovolili přistoupit k opatření, které lze považovat za neefektivní: při každém přístupu ke slotu `items` vytváříme nový seznam. V reálných situacích (ale opravdu vždy až praxe ukáže, že je to nutné!) je proto možno přijmout kompromis, který povede z hlediska rychlosti programu a jeho nároků na paměť k efektivnějšímu řešení, ale současně sníží jeho odolnost vůči chybám. V našem případě by kompromisní řešení vypadalo tak, že bychom zůstali u původní implementace metod `items` a `set-items` (tj. bez použití funkce `copy-list`) a v dokumentaci bychom jasně napsali, že jako autoři třídy `picture` neručíme za následky, které bude mít přímá editace seznamu prvků obrázků.

6. Polymorfismus

6.1. Kreslení pomocí knihovny `micro-graphics`

V této podkapitole představíme jednoduchou grafickou knihovnu `micro-graphics`, napsanou v prostředí LispWorks čistě pro účely tohoto textu, kterou budeme používat k vykreslování našich grafických objektů.

Objektové systémy obvykle vytvářejí většinu svých nástrojů včetně grafických tak, že implementují a poskytují objektové rozhraní k jiným, procedurálně napsaným knihovnám. Při implementaci kreslení grafických objektů budeme postupovat stejně; roli oné nižší procedurální vrstvy bude hrát právě knihovna `micro-graphics`, jejíž služby mají čistě procedurální charakter.

V této podkapitole popíšeme pouze základní funkce knihovny, o dalších se zmíníme vždy až to bude potřeba. Kompletní popis rozhraní knihovny najdeme v Dodatku B.

Poznámka 6.1. Tato podkapitola tedy tematicky nepatří do oblasti objektového programování. Knihovna `micro-graphics` je čistě procedurální.

Knihovna `micro-graphics` se do prostředí LispWorks načítá tak, že se načte (funkcí `load` nebo z menu) její soubor `load.lisp`. K dispozici dává následující základní funkce:

```
(mg:display-window) => window
```

Funkce `mg:display-window` vytvoří a zobrazí nové grafické okno. Jako výsledek vrací odkaz na toto okno, který je třeba používat jako parametr v ostatních funkcích, jež s oknem pracují. Nové okno má několik kreslicích parametrů, které lze zjišťovat pomocí funkce `mg:get-param` a nastavovat pomocí funkce `mg:set-param`. Souřadnice v okně se udávají v pixelech, jejich počátek je v levém horním rohu okna, hodnoty druhé souřadnice se zvětšují směrem dolů.

```
(mg:get-param window param) => value
```

window: hodnota vrácená funkcí `mg:display-window` *param*: symbol

Funkce `mg:get-param` vrací hodnotu kreslicího parametru *param* okna *window*. Pro nás jsou důležité tyto parametry:

- :thickness** Tloušťka čáry v pixelech. Ovlivňuje funkce `mg:draw-circle` a `mg:draw-polygon`, pokud není nastaven parametr `:fillp`. Počáteční hodnota: 1.
- :foreground** Barva inkoustu. Ovlivňuje funkce `mg:draw-circle` a `mg:draw-polygon`. Počáteční hodnota: `:black`.
- :background** Barva pozadí. Ovlivňuje funkci `mg:clear`. Počáteční hodnota: `:white`.
- :filledp** Zda kreslit kruhy a polygony vyplněné. Ovlivňuje funkce `mg:draw-circle` a `mg:draw-polygon`. Počáteční hodnota: `nil`.
- :closedp** Zda spojit poslední a první vrchol polygonu. Ovlivňuje funkci `mg:draw-polygon`, pokud není nastaven parametr `filledp`. Počáteční hodnota: `nil`.

Přípustnými hodnotami parametrů `:foreground` a `:background` jsou všechny symboly, které v grafickém systému LispWorks pojmenovávají barvu. Jejich seznam lze zjistit funkcí `color:get-all-color-names`, nebo, pokud uvedeme část názvu barvy, kterou chceme použít, funkcí `color:apropos-color-names`. Vzorkovník barev je také součástí Dodatku B.

Poznámka 6.2. Barvy lze také v LispWorks vytvářet z komponent pomocí zabudovaných funkcí `color:make-rgb`, `color:make-hsv`, `color:make-gray`. Zájemci se mohou na tyto funkce podívat do dokumentace.

Kreslicí parametry lze nastavovat funkcí `mg:set-param`.

```
(mg:set-param window param value) => nil
```

window: hodnota vrácená funkcí `mg:display-window`
param: symbol
value: hodnota

Funkce `mg:set-param` nastavuje kreslicí parametr *param* okna *window* na hodnotu *value*. Význam kreslicích parametrů je uveden u funkce `mg:get-param`. Nové kreslicí parametry ovlivňují způsob kreslení do okna (funkcemi `mg:clear`, `mg:draw-circle`, `mg:draw-polygon`) od momentu, kdy byly nastaveny.

```
(mg:clear window) => nil
```

window: hodnota vrácená funkcí `mg:display-window`

Funkce `mg:clear` vymaže celé okno *window* barvou aktuálně uloženou v kreslicím parametru `:background`.

```
(mg:draw-circle window x y r) => nil
```

window: hodnota vrácená funkcí `mg:display-window`
x, *y*, *r*: čísla

Funkce `mg:draw-circle` nakreslí do okna *window* kruh se středem o souřadnicích *x*, *y* a poloměrem *r*. Kruh se kreslí barvou uloženou v kreslicím parametru `:foreground` okna *window*. Kreslicí parametr `:filledp` okna *window* udává, zda se bude kruh kreslit vyplněný. Pokud není nastaven, bude se kreslit pouze obvodová kružnice čarou, jejíž tloušťka je uložena v kreslicím parametru `:thickness` okna *window*.

```
(mg:draw-polygon window points) => nil
```

window: hodnota vrácená funkcí `mg:display-window`
points: seznam čísel

Funkce `mg:draw-polygon` nakreslí do okna *window* polygon s vrcholy danými parametrem *points*. Tento parametr musí obsahovat seznam sudé délky, jako prvky se v něm musí střídát *x*ové a *y*ové souřadnice vrcholů polygonu. Kreslí se barvou uloženou v kreslicím parametru `:foreground` okna *window*. Kreslicí parametr `:filledp` okna *window* udává, zda se bude polygon kreslit vyplněný. Pokud není nastaven, budou se kreslit pouze úsečky spojující jednotlivé vrcholy polygonu čarou, jejíž tloušťka

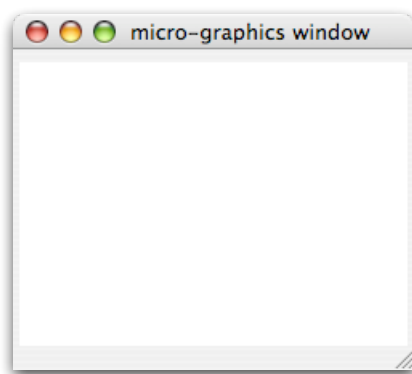
je uložena v kreslicím parametru `:thickness` okna *window*. Kreslicí parametr `:closedp` okna *window* určuje, zda se má nakreslit i úsečka spojující poslední bod polygonu s prvním. Pokud je nastaven kreslicí parametr `:filledp`, kreslicí parametr `:closedp` se ignoruje.

Příklad 6.3. Vyzkoušejme některé ze služeb knihovny *micro-graphics*. Nejprve knihovnu načteme do prostředí *LispWorks* volbou nabídky „Load...” a souboru `load.lisp` (při volbě nabídky „Load...” musí být aktivní okno s příkazovým řádkem, nikoliv okno s editovaným souborem, jinak se načte tento soubor).

Nejprve zavoláme funkci `mg:display-window` a výsledek uložíme do proměnné:

```
CL-USER 13 > (setf w (mg:display-window))
#<MG-WINDOW 216B76F3>
```

Otevře se nově vytvořené okno knihovny *micro-graphics*, jak vidíme na Obrázku 11.



Obrázek 11: Prázdné okno knihovny *micro-graphics*

Výsledek tohoto volání (v našem případě zapisovaný prostředím jako `#<MG-WINDOW 216B76F3>`) slouží pouze jako identifikátor okna, který budeme používat při dalších voláních funkcí knihovny. Žádný jiný význam pro nás nemá.

Pomocí funkce `mg:get-param` můžeme zjistit přednastavené kreslicí parametry:

```
CL-USER 14 > (mapcar
               (lambda (p)
                 (list p (mg:get-param w p)))
               '(:thickness :foreground :background
                 :filledp :closedp))
((:THICKNESS 1) (:FOREGROUND :BLACK) (:BACKGROUND :WHITE)
 (:FILLEDP NIL) (:CLOSEDP NIL))
```

Kreslicí parametry můžeme nastavit funkcí `mg:set-param`. Zkusme tedy změnit barvu pozadí a potom pomocí funkce `mg:clear` okno vymazat:

```
CL-USER 15 > (mg:set-param w :background :green)
NIL

CL-USER 16 > (mg:clear w)
NIL
```



Obrázek 12: Okno knihovny micro-graphics po změně barvy pozadí

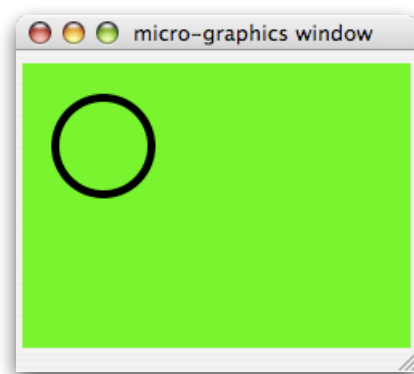
Pokud jsme to udělali dobře, pozadí okna se přebarví na zeleno (Obrázek 12).

Jako další krok otestujeme funkci `mg:draw-circle`. Víme, že tato funkce vyžaduje jako parametry údaje o středu a poloměru vykreslovaného kruhu. Kromě toho její výsledek ovlivňují kreslicí parametry `:foreground`, `:thickness` a `:filledp`. Nastavme tedy nejprve například parametr `:thickness` a zavolejme funkci `mg:draw-circle`:

```
CL-USER 17 > (mg:set-param w :thickness 5)
NIL

CL-USER 18 > (mg:draw-circle w 50 50 30)
NIL
```

V okně se objeví černý nevyplněný kruh tloušťky 5 (Obrázek 13).



Obrázek 13: Okno knihovny micro-graphics po nakreslení kružnice

Na závěr testu ještě změníme parametry `:filledp` a `:foreground` a zkusíme pomocí funkce `mg:draw-polygon` nakreslit trojúhelník:

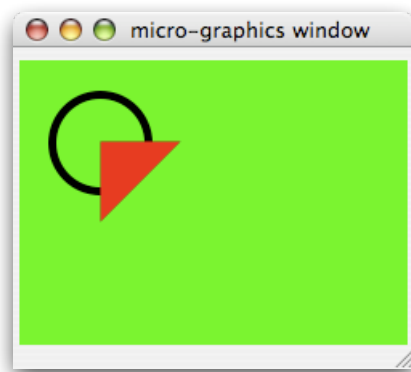
```
CL-USER 19 > (mg:set-param w :foreground :red)
NIL

CL-USER 20 > (mg:set-param w :filledp t)
NIL
```



```
CL-USER 21 > (mg:draw-polygon w '(50 50 100 50 50 100))  
NIL
```

Výslednou podobu okna vidíme na Obrázku 14.



Obrázek 14: Okno knihovny `micro-graphics` po nakreslení trojúhelníka

Poznámka 6.4. Postup kreslení pomocí knihovny `micro-graphics` si můžeme zjednodušeně představit tak, že knihovna společně s operačním systémem (jeho grafickou nadstavbou) zjistí pomocí identifikátoru okna (v našem příkladě uloženém v proměnné `w`) adresy míst v paměti, která je třeba modifikovat, aby uživatel na obrazovce viděl nakreslený grafický útvar. Potom tato místa v paměti modifikuje.

Tento popis se vztahuje především na starší a jednodušší operační systémy, případně jejich grafické nadstavby. V ostatních je kreslení složitější (kreslicí algoritmy realizuje grafická karta, nekreslí se přímo na obrazovku, ale do bufferu, jehož obsah se teprve na obrazovku přenese atd.). V každém případě se ale jedná o jednorázovou akci, která se provede pouze v okamžiku, kdy zavoláme nějakou kreslicí funkci (v našem případě `mg:clear`, `mg:draw-circle`, nebo `mg:draw-polygon`). Neměli bychom se proto divit, když nám po manipulacích s oknem (změně rozměrů, překrývání jinými okny a podobně) obrázek postupně zmizí.

6.2. Kreslení grafických objektů

Když jsme se naučili používat procedurální grafickou knihovnu `micro-graphics`, zkusíme ji využít ke kreslení našich grafických objektů.

Budeme pokračovat v objektovém přístupu; proto budeme grafické objekty kreslit tak, že jim budeme posílat zprávy a necháme je, aby vykreslení pomocí knihovny `micro-graphics` již provedly samy ve svých metodách.

Knihovna `micro-graphics` má procedurální rozhraní, výsledek kreslení je závislý na hodnotách, které je nutno nastavit před kreslením. Při objektovém přístupu požadujeme, aby výsledky akcí prováděných s objekty byly pokud možno závislé pouze na vnitřním stavu objektů (a hodnotách parametrů zpráv objektům zasílaných) — tím se zajistí relativní samostatnost objektů a jejich použitelnost v jiné situaci. Proto budou informace o způsobu kreslení (barva, tloušťka pera a podobně) součástí vnitřního stavu objektů, stejně jako informace o okně, do nějž se objekty mají kreslit.

Poznámka 6.5. To je v souladu s principy objektového programování i s intuitivní představou: například barva kruhu je zjevně jeho vlastnost, kruh by tedy měl údaj o ní nějakým způsobem obsahovat a při kreslení by na ni měl brát ohled.

Příklad 6.6. Uvedme nejprve definici třídy `window`, jejíž instance budou obsahovat informace o okně knihovny `micro-graphics`, do něhož lze kreslit naše grafické objekty, (slot `mg-window`) a další údaje. Mezi tyto údaje patří:

- grafický objekt, který se do okna vykresluje (slot `shape`),
- barva pozadí okna (slot `background`).

Definice třídy:

```
(defclass window ()
  ((mg-window :initform (mg:display-window))
   (shape :initform nil)
   (background :initform :white)))
```

Definice metod, které přistupují k jednotlivým slotům:

```
(defmethod mg-window ((window window))
  (slot-value window 'mg-window))

(defmethod shape ((w window))
  (slot-value w 'shape))

(defmethod set-shape ((w window) shape)
  (set-window shape w)
  (setf (slot-value w 'shape) shape)
  w)

(defmethod background ((w window))
  (slot-value w 'background))

(defmethod set-background ((w window) color)
  (setf (slot-value w 'background) color)
  w)
```

Metoda `redraw` vykreslí obsah okna tak, že nejprve zjistí ze slotu `background` barvu pozadí, touto barvou obsah okna vymaže (funkcí `mg:clear`) a nakonec pošle grafickému objektu ve slotu `shape` zprávu `draw`:

```
(defmethod redraw ((window window))
  (let ((mgw (slot-value window 'mg-window)))
    (mg:set-param mgw :background (background window))
    (mg:clear mgw)
    (when (shape window)
      (draw (shape window)))))
  window)
```

Vidíme, že zpráva `draw` má následující syntax:

```
(draw circle)
```

Po jejím zaslání grafickému objektu by se měl tento objekt vykreslit do svého okna. Obsluhu této zprávy bude tedy třeba definovat pro všechny třídy grafických objektů.

Příklad 6.7. Implementaci kreslení začneme u třídy `circle`. Z předchozího příkladu víme, že je třeba definovat zprávu `draw`: Po jejím obdržení by se měl kruh `circle` vykreslit. To bude vyžadovat přidání slotů a metod třídy `circle`:

```

(defclass circle ()
  ((center :initform (make-instance 'point))
   (radius :initform 1)
   (color :initform :black)
   (thickness :initform 1)
   (filledp :initform nil)
   (window :initform nil)))

(defmethod color ((c circle))
  (slot-value c 'color))

(defmethod set-color ((c circle) value)
  (setf (slot-value c 'color) value)
  c)

(defmethod thickness ((c circle))
  (slot-value c 'thickness))

(defmethod set-thickness ((c circle) value)
  (setf (slot-value c 'thickness) value)
  c)

(defmethod filledp ((c circle))
  (slot-value c 'filledp))

(defmethod set-filledp ((c circle) value)
  (setf (slot-value c 'filledp) value)
  c)

(defmethod window ((c circle))
  (slot-value c 'window))

(defmethod set-window ((c circle) value)
  (setf (slot-value c 'window) value)
  c)

```

Poznámka 6.8. To je trochu zbytečně moc stejného psaní (které v programování není nikdy vítáno). Později se dozvíme, jak si v těchto případech lze ušetřit práci.

Slot `window` bude obsahovat instanci třídy `window`. Napíšeme si ještě užitečnou metodu, která zjistí z okna kruhu příslušný odkaz na okno knihovny `micro-graphics`:

```

(defmethod shape-mg-window ((c circle))
  (when (window c)
    (mg-window (window c))))

```

Metoda `draw` třídy `circle` bude sestávat ze dvou částí:

1. Nastavení kreslicích parametrů okna podle hodnot slotů kruhu,
2. vykreslení kruhu (funkcí `mg:draw-circle`).

Bude rozumné definovat kód pro tyto dva úkony zvlášť. Jelikož programujeme objektově, definujeme dvě pomocné zprávy, jejichž obsluha tyto úkony provede. První z nich nazveme `set-mg-params`. Příslušná metoda bude vypadat takto:

```
(defmethod set-mg-params ((c circle))
  (let ((mgw (shape-mg-window c)))
    (mg:set-param mgw :foreground (color c))
    (mg:set-param mgw :thickness (thickness c))
    (mg:set-param mgw :filledp (filledp c)))
  c)
```

Zprávu pro vlastní vykreslení nazveme do-draw. Metoda bude vypadat takto:

```
(defmethod do-draw ((c circle))
  (mg:draw-circle (shape-mg-window c)
                  (x (center c))
                  (y (center c))
                  (radius c))
  c)
```

K dokončení už zbývá pouze definovat vlastní metodu draw. Ta ovšem bude jednoduchá:

```
(defmethod draw ((c circle))
  (set-mg-params c)
  (do-draw c))
```

Test:

```
CL-USER 1 > (setf w (make-instance 'window))
#<WINDOW 217F04BF>

CL-USER 2 > (setf circ (make-instance 'circle))
#<CIRCLE 218359FB>

CL-USER 3 > (set-x (center circ) 100)
#<POINT 2183597B>

CL-USER 4 > (set-y (center circ) 100)
#<POINT 2183597B>

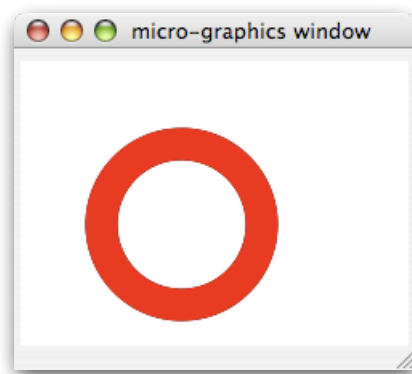
CL-USER 5 > (set-radius circ 50)
#<CIRCLE 218359FB>

CL-USER 6 > (set-color circ :red)
#<CIRCLE 218359FB>

CL-USER 7 > (set-thickness circ 20)
#<CIRCLE 218359FB>

CL-USER 8 > (set-shape w circ)
#<WINDOW 217F04BF>

CL-USER 9 > (redraw w)
#<WINDOW 217F04BF>
```



Obrázek 15: Červené kolečko

Výsledek by měl odpovídat Obrázku 15.

Obrázek v okně můžeme kdykoli překreslit zavoláním

```
(redraw w)
```

Příklad 6.9. Budeme pokračovat kreslením obrázků, tedy instancí třídy `picture`. Jako první si položíme otázku: jakou zprávu definovat pro instance této třídy, aby se po jejím přijetí vykreslily? Jako vhodný kandidát se samozřejmě nabízí zpráva `draw`, kterou jsme již použili u třídy `circle`.

Snadno zjistíme, že nám CLOS tuto volbu umožní:

```
(defmethod draw ((pic picture))
  (dolist (item (reverse (items pic)))
    (draw item))
  pic)
```

Než zjistíme důsledky této definice, podívejme se na právě napsanou metodu. Procházíme v ní všechny prvky obrázku `pic` od posledního k prvnímu (díky funkci `reverse`) a každému posíláme zprávu `draw`. Metoda by tedy opravdu měla vykreslit všechny prvky obrázku, přičemž objekty, které jsou v seznamu prvků obrázku vpředu, by měly překrývat objekty více vzadu.

Vyzkoušejme kreslení obrázku na příkladě. Vytvoříme instanci třídy `picture`, která bude obsahovat několik soustředných kruhů se střídajícími se barvami. Jelikož to bude trochu pracné, napíšeme si pomocnou funkci:

```
(defun make-bulls-eye (x y radius count window)
  (let ((items '())
        (step (/ radius count))
        (blackp t)
        circle)
    (dotimes (i count)
      (setf circle (set-window
                    (set-filledp
                     (set-color
                      (set-radius (make-instance 'circle)
                                   (- radius (* i step))))
                      (if blackp :black :light-blue))
                    window)))
```

```

        t)
        window))
    (set-y (set-x (center circle) x) y)
    (setf items (cons circle items)
      blackp (not blackp)))
    (set-items (make-instance 'picture) items)))

```

Funkce `make-bulls-eye` nejprve vytvoří obrázek (instanci třídy `picture`), pak v cyklu vytvoří zadaný počet kruhů, nastaví jim potřebné parametry a shromáždí je v seznamu. Tento seznam pak nastaví jako seznam prvků obrázku. Vytvořený obrázek vrátí jako výsledek.

Test:

```

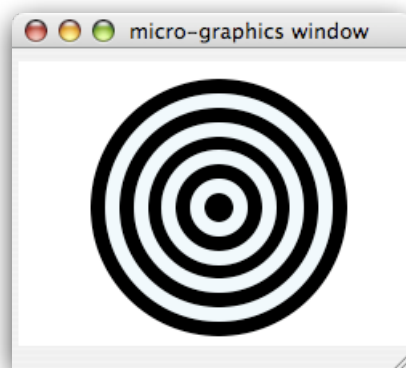
CL-USER 17 > (setf w (make-instance 'window))
#<WINDOW 200CB967>

CL-USER 18 > (setf eye (make-bulls-eye 125 90 80 9 w))
#<PICTURE 20105CB3>

CL-USER 19 > (draw eye)
#<PICTURE 20105CB3>

```

Výsledné okno je na Obrázku 16. Kruhy v obrázku jsou vyplněné (mají nastaveno



Obrázek 16: Terč

`filledp` na `t`), výsledného efektu je dosaženo jejich překrytím.

Poznámka 6.10. Zkuste v předchozím příkladě vykreslit okno voláním `redraw w` a komentujte výsledek.

Příklad 6.11. Jak těžké nyní bude nakreslit dva terče vedle sebe? Podívejme se na to:

```

CL-USER 20 > (setf w (make-instance 'window))
#<WINDOW 200E040F>

CL-USER 21 > (setf eye1 (make-bulls-eye 60 90 40 5 w))
#<PICTURE 218721BB>

CL-USER 22 > (setf eye2 (make-bulls-eye 185 90 40 5 w))
#<PICTURE 21833C93>

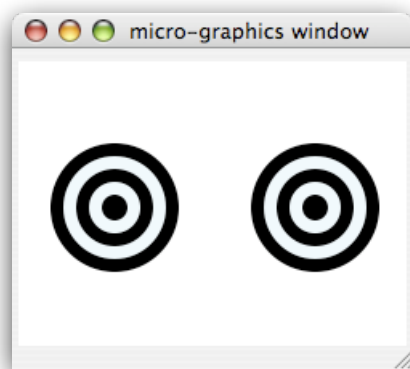
```

```
CL-USER 23 > (setf pic (make-instance 'picture))
#<PICTURE 2181811F>

CL-USER 24 > (set-items pic (list eye1 eye2))
#<PICTURE 2181811F>

CL-USER 25 > (draw pic)
#<PICTURE 2181811F>
```

A výsledek je na Obrázku 17.



Obrázek 17: Dva terče vedle sebe

6.3. Princip polymorfismu

Aniž bychom o tom hovořili, využili jsme v předchozích dvou příkladech *princip polymorfismu*, který nám usnadnil práci.

Princip polymorfismu v objektovém programování

Různé objekty mohou mít definovány různé obsluhy téže zprávy.

U jazyků založených na třídách jsou metody definovány pro třídy. Pro tyto jazyky lze princip polymorfismu upřesnit takto:

Princip polymorfismu pro jazyky založené na třídách

Různé třídy mohou mít definovány pro tutéž zprávu různé metody.

Princip polymorfismu jsme v předchozích dvou příkladech využili dvakrát. Poprvé při definici metod pro zprávu `draw`, podruhé při jejím zasílání.

Při definici metod tříd `circle` a `picture` nám princip polymorfismu umožnil definovat pro každou z těchto tříd metody téhož názvu, ale s různou implementací — metody `draw` ve třídách `circle` a `picture` mají různé definice. Systém umožňuje pojmenovat akce, které se liší provedením (implementací), ale nikoli významem, stejným názvem.

V metodě `draw` třídy `picture` posíláme zprávu `draw` prvkům obrázku, aniž bychom dopředu znali jejich třídu. Teprve v momentě, kdy je zpráva zaslána, rozhodne systém podle třídy příjemce, jakou metodu má zavolat. To je druhé využití principu polymorfismu v uvedených dvou příkladech.

Příklad 6.12. Kdyby nebylo principu polymorfismu, musely by mít metody pro vykreslení objektů v různých třídách různé názvy. Kdyby tyto názvy byly například `circle-draw` pro třídu `circle` a `picture-draw` pro třídu `picture`, musela by definice metody `picture-draw` vypadat takto:

```
(defmethod picture-draw ((pic picture))
  (dolist (item (reverse (items pic)))
    (cond ((typep item 'circle) (circle-draw item))
          ((typep item 'picture) (picture-draw item))))))
```

Kromě toho, že je tato definice delší než původní definice, má ještě jednu nevýhodu: kdykoliv bychom definovali novou třídu grafických objektů s metodou pro vykreslování, museli bychom modifikovat i metodu `picture-draw` třídy `picture`. Za chvíli například definujeme třídu `polygon`. V případě, že nemáme k dispozici princip polymorfismu, bychom její metodu pro vykreslení museli pojmenovat jedinečným názvem, například `polygon-draw`, a upravit i metodu `picture-draw`:

```
(defmethod picture-draw ((pic picture))
  (dolist (item (reverse (items pic)))
    (cond ((typep item 'circle) (circle-draw item))
          ((typep item 'picture) (picture-draw item))
          ((typep item 'polygon) (polygon-draw item))))))
```

Změna na jednom místě programu by tedy znamenala nutnost změny i na dalších místech. Této nutnosti nás princip polymorfismu zbavuje.

Poznámka 6.13. V objektových programovacích jazycích je princip polymorfismu obvykle přítomen. Předchozí příklad proto není reálný. Při používání procedurálního programovacího stylu se s podobnými jevy ale setkáváme.

Poznámka 6.14. Jak za chvíli uvidíme, po definici třídy `polygon` budeme stejně muset předefinovat jinou metodu třídy `picture`: metodu `set-items`. Tuto nepříjemnost vyřešíme lépe až pomocí dědičnosti.

Příklad 6.15. Dbejme na přesnost vyjadřování. Následující dvě charakterizace principu polymorfismu nejsou správně: „Dva různé objekty mohou po přijetí téže zprávy vrátit různé výsledky,“ „dva různé objekty mohou po přijetí téže zprávy vykonat různé akce.“

První definice není správná, protože k tomu, aby byly po zaslání zprávy vráceny různé výsledky není nutné, aby měly objekty pro zprávu definovány různé metody. Uvažte například zprávu `color` zaslanou dvěma různým instancím třídy `circle`.

Druhá definice rovněž není správná. Necháme na čtenáři, aby zvážil, proč.

6.4. Další příklady

Příklad 6.16. Knihovna `micro-graphics` nabízí možnost kreslení polygonů. Je tedy přirozené definovat `polygon` jako objekt v našem objektovém grafickém systému.

Z našeho pohledu jsou polygony kromě obrázků dalším typem grafických objektů, které obsahují jiné grafické objekty jako své prvky. Polygon je tvořen seznamem bodů, kreslí se jako lomená čára, tyto body spojující. Obdélníky, čtverce i trojúhelníky jsou polygony.

Základní definice třídy `polygon` je tedy velmi podobná definici třídy `picture`. Kromě slotu `items` ovšem podobně jako u třídy `circle` definujeme další sloty, které budou obsahovat informace potřebné ke kreslení, a metody pro přístup k nim:


```

(defclass polygon ()
  ((items :initform '())
   (color :initform :black)
   (thickness :initform 1)
   (filledp :initform nil)
   (closedp :initform t)
   (window :initform nil)))

(defmethod items ((poly polygon))
  (copy-list (slot-value poly 'items)))

(defmethod set-items ((poly polygon) value)
  (unless (every (lambda (elem)
                   (typep elem 'point))
                value)
    (error "Invalid polygon element type."))
  (setf (slot-value poly 'items) (copy-list value))
  poly)

(defmethod color ((p polygon))
  (slot-value p 'color))

(defmethod set-color ((p polygon) value)
  (setf (slot-value p 'color) value)
  p)

(defmethod thickness ((p polygon))
  (slot-value p 'thickness))

(defmethod set-thickness ((p polygon) value)
  (setf (slot-value p 'thickness) value)
  p)

(defmethod closedp ((p polygon))
  (slot-value p 'closedp))

(defmethod set-closedp ((p polygon) value)
  (setf (slot-value p 'closedp) value)
  p)

(defmethod filledp ((p polygon))
  (slot-value p 'filledp))

(defmethod set-filledp ((p polygon) value)
  (setf (slot-value p 'filledp) value)
  p)

(defmethod window ((p polygon))
  (slot-value p 'window))

(defmethod set-window ((p polygon) value)
  (setf (slot-value p 'window) value)
  p)

```

```
(defmethod shape-mg-window ((shape polygon))
  (when (window shape)
    (mg-window (window shape))))
```

Kreslení polygonu navrhne podobně jako u třídy `circle`. Parametry okna knihovny `micro-graphics`, které ovlivní kreslení, jsou `:foreground`, `:thickness`, `:filledp` a `:closedp`, hodnoty všech zjišťujeme z příslušných slotů. V metodě `do-draw` musíme nejprve souřadnice bodů polygonu zpracovat do tvaru, který vyžaduje funkce `mg:draw-polygon`, pak ji můžeme zavolat.

```
(defmethod set-mg-params ((poly polygon))
  (let ((mgw (shape-mg-window poly)))
    (mg:set-param mgw :foreground (color poly))
    (mg:set-param mgw :thickness (thickness poly))
    (mg:set-param mgw :filledp (filledp poly))
    (mg:set-param mgw :closedp (closedp poly))
    poly))

(defmethod do-draw ((poly polygon))
  (let (coordinates)
    (dolist (point (reverse (items poly)))
      (setf coordinates (cons (y point) coordinates))
      (setf coordinates (cons (x point) coordinates)))
    (mg:draw-polygon (shape-mg-window poly)
                     coordinates))
  poly)

(defmethod draw ((poly polygon))
  (set-mg-params poly)
  (do-draw poly))
```

Příklad 6.17. Jednoduchý příklad práce s polygonem:

```
CL-USER 1 > (setf w (make-instance 'window))
#<WINDOW 200A5D9F>

CL-USER 2 > (setf p (make-instance 'polygon))
#<POLYGON 216FBD3B>

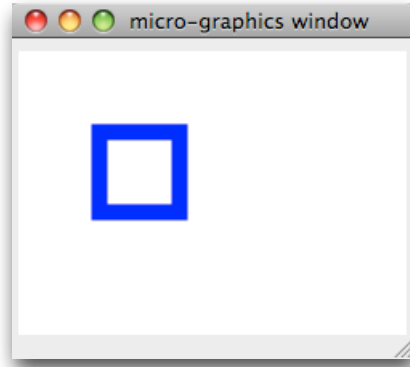
CL-USER 3 > (set-items
             p
             (list (move (make-instance 'point) 50 50)
                   (move (make-instance 'point) 100 50)
                   (move (make-instance 'point) 100 100)
                   (move (make-instance 'point) 50 100)))
#<POLYGON 216FBD3B>

CL-USER 4 > (set-color (set-thickness p 10) :blue)
#<POLYGON 216FBD3B>

CL-USER 5 > (set-shape w p)
#<WINDOW 200A5D9F>
```

```
CL-USER 6 > (redraw w)
#<WINDOW 200A5D9F>
```

Pokus by měl skončit stejně jako na Obrázku 18.



Obrázek 18: Modrý čtverec

Příklad 6.18. Po definici třídy `polygon` je ještě potřeba přidat tuto třídu k seznamu tříd, jejichž instance je povoleno ukládat do seznamu prvků obrázků. Proto musíme změnit definici metody `set-items` třídy `picture`:

```
(defmethod set-items ((pic picture) value)
  (unless (every (lambda (elem)
                  (or (typep elem 'point)
                      (typep elem 'circle)
                      (typep elem 'picture)
                      (typep elem 'polygon))))
    value)
    (error "Picture elements are not of desired type."))
  (setf (slot-value pic 'items) (copy-list value))
  pic)
```

Vidíme, že definice třídy `polygon` vyvolala potřebu změnit definici metody třídy `picture`. Takové závislosti mezi různými částmi zdrojového kódu programu jsou nežádoucí, protože mohou snadno vést k chybám. V další kapitole potíží vyřešíme pomocí principu dědičnosti.

Nyní uvedeme další příklady operací, které má smysl provádět se všemi grafickými objekty bez ohledu na to, jakého typu tyto objekty jsou: posunutí, rotaci a změnu měřítka.

Příklad 6.19. *Posunutí* je operace, která změní polohu grafického objektu na základě zadaných přírůstků souřadnic. Objekt posuneme tak, že mu pošleme zprávu `move`, jejíž syntax je následující:

```
(move object dx dy)
```

object: grafický objekt, jemuž zprávu posíláme
dx, dy: čísla

Čísla dx a dy jsou přírůstky na ose x a y , o něž chceme objekt *object* posunout.

Je zřejmé, že zatímco z hlediska uživatele není podstatné, jaký grafický objekt posouváme, obsluha zprávy *move* bude u objektů různých tříd různá. Definice metod se tedy budou u různých tříd lišit:

```
(defmethod move ((pt point) dx dy)
  (set-x pt (+ (x pt) dx))
  (set-y pt (+ (y pt) dy))
  pt)

(defmethod move ((c circle) dx dy)
  (move (center c) dx dy)
  c)

(defmethod move ((pic picture) dx dy)
  (dolist (it (items pic))
    (move it dx dy))
  pic)
```

U třídy *point* jednoduše přičítáme přírůstky dx a dy ke kartézským souřadnicím bodu. Kruh posouváme tak, že posouváme jeho střed. U obrázku a polygonu posouváme všechny grafické objekty, které obsahuje (v případě polygonu jsou to body, u obrázku libovolné grafické objekty — v případě obrázku tedy posíláme zprávu *move* různým grafickým objektům, aniž bychom znali jejich typ, tedy aniž bychom věděli, jaká z definovaných metod se vykoná).

Příklad 6.20. *Rotate* je otočení objektu o daný úhel kolem daného bodu. Definujeme zprávu *rotate* s následující syntaxí:

```
(rotate object angle center)

object: grafický objekt
angle: číslo
center: instance třídy point
```

Zde *angle* je úhel, o který chceme objekt otočit, a *center* střed rotace.

V obsluze zprávy *rotate* u třídy *point* budeme postupovat tak, že bod nejprve posuneme tak, aby střed rotace splýval s počátkem souřadnic, pak změníme jeho polární souřadnice a posuneme jej zpět. Implementace tohoto postupu bude následující:

```
(defmethod rotate ((pt point) angle center)
  (let ((cx (x center))
        (cy (y center)))
    (move pt (- cx) (- cy))
    (set-phi pt (+ (phi pt) angle))
    (move pt cx cy)
    pt))
```

Metoda pro třídu *circle* již pouze správně otočí střed kruhu, u třídy *picture* otočíme všechny objekty v obrázku:

```

(defmethod rotate ((c circle) angle center)
  (rotate (center c) angle center)
  c)

(defmethod rotate ((pic picture) angle center)
  (dolist (it (items pic))
    (rotate it angle center))
  pic)

```

Příklad 6.21. Další základní geometrickou transformací je *změna měřítka*, kterou realizujeme pomocí zprávy `scale`. Podrobnosti najdete ve zdrojovém kódu k této části textu.

Poslední změny

- 23. 3. 2008 Přidána Poznámka 6.14 a Příklad 6.18.
- 12. 3. 2008 Rozšířen Příklad 6.16 (polygon), přidán Příklad 6.17 (ukázka kreslení polygonu).
- 11. 3. 2008 Drobná změna v definici třídy `window` a její metody `redraw`: Počáteční hodnota slotu `shape` je `nil`, metoda `redraw` na to bere ohled. Tím je z textu prozatím odstraněna zmínka o třídě `empty-shape`.
- 10. 3. 2008 Přidána poznámka o změně měřítka na konec.

7. Dědičnost

VŠUDE MÍSTO “HIERARCHIE” DÁT “HIERARCHIE PODLE DĚDIČNOSTI”,
“STROM DĚDIČNOSTI” APOD.

EMPTY-SHAPE, FULL-SHAPE

7.1. Dědičnost jako nástroj redukující opakování v kódu

Pokusme se nejprve pochopit příčiny častého opakování stejného kódu v implementaci grafických objektů z předchozí kapitoly.

Začněme slotem `window` a souvisejícími metodami ve všech dosud definovaných třídách. Slot `window` má u všech grafických objektů, bez ohledu na jejich třídu, stejný účel: uchovávat odkaz na okno, do kterého se bude objekt vykreslovat. Metody, které s ním pracují, jsou tři. Připomeňme jejich implementaci například ve třídě `point`:

```
(defmethod window ((pt point))
  (slot-value pt 'window))

(defmethod set-window ((pt point) value)
  (setf (slot-value pt 'window) value)
  pt)

(defmethod shape-mg-window ((pt point))
  (when (window pt)
    (mg-window (window pt))))
```

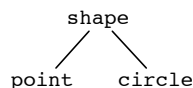
Metody `window` a `set-window` realizují zapouzdření slotu `window` grafického objektu, metoda `shape-mg-window` zjišťuje okno knihovny `micro-graphics`, do kterého se má objekt vykreslit. Všechny tři metody jsou u všech tříd grafických objektů implementovány přesně stejně, jako u třídy `point`. To je samozřejmě velmi nevýhodné řešení, se kterým se nelze spokojit.

Důvod, proč se u grafických objektů různých tříd některé sloty a metody opakují, je následující: všechny grafické objekty, bez ohledu na třídu, které jsou přímými instancemi, mají některé společné vlastnosti. Jednou z nich je, jak jsme zjistili, například to, že ve slotu `window` nesou informaci o okně, do něž mají být vykreslovány, a že mají implementovány tři metody, které s tímto slotem pracují. Společných vlastností grafických objektů za chvíli najdeme víc.

Tuto souvislost lze shrnout jednoduše: *Body, kružnice, obrázky, polygony a prázdné a plné objekty jsou všechno grafické objekty.*

Objektové programování umožňuje zachytit tento jev explicitně, pomocí *principu dědičnosti*. Tento princip je založen tom, že u tříd lze specifikovat stromovou hierarchii⁵. Lze definovat obecné třídy, které (pokud jsou chápány jako množiny objektů) jsou nadmnožinami tříd jiných.

V našem případě lze definovat obecnou třídu `shape`, jejímiž instancemi budou všechny grafické objekty a která bude specifikovat vlastnosti společné všem grafickým objektům. U tříd konkrétnějších (například `point` nebo `circle`) pak již nebudeme tyto společné vlastnosti definovat znovu, ale pouze stanovíme, že konkrétnější třídy jsou podmnožinami obecné třídy `shape`.



Obrázek 19: Strom tříd grafických objektů, první verze

Část vznikající stromové hierarchie našich tříd můžeme vidět na Obrázku 19.

Každá třída především představuje množinu objektů. Třída `shape` je na uvedeném obrázku umístěna nad třídami `point` a `circle`, jelikož je jejich nadmnožinou. Naopak, třídy `point` a `circle` jsou podmnožinami třídy `shape` (každý bod je grafický objekt, každý kruh je grafický objekt).

O vztahu nadmnožina–podmnožina mezi třídami (tedy předchůdce–následník v jejich stromové hierarchii) se hovoří jako o vztahu *předek–potomek* (*ancestor–descendant*, *predecessor–successor*), někdy také *nadtřída–podtřída* (*superclass–subclass*).

Z množinového vztahu mezi podtřídou a nadtřídou vyplývá, že *každá instance třídy je současně instancí její nadtřídy*. Tomuto pravidlu se říká *pravidlo is-a*. Jedná se o v podstatě triviální fakt, který ale má zásadní důležitost při navrhování jakékoliv stromové hierarchie tříd. Aby náš návrh tříd věrně modeloval vztahy mezi reálnými objekty, musí toto pravidlo bezpodmínečně splňovat.

Poznámka 7.1. Při návrhu hierarchie tříd se nesmíme nechat svést okamžitou výhodností nějakého řešení. Vždy musíme mít na mysli principiální souvislosti. Jen tak můžeme doufat, že náš návrh obstojí i v budoucnu, poté, co jej bude nutno upravovat podle nově vzniklých (a nepředpokládaných) požadavků.

Poznámka 7.2. Pravidlo *is-a* není jediným pravidlem, které je třeba při návrhu tříd dodržet, je ale základní a nejdůležitější.

Příklad 7.3. Každý bod je grafickým útvarem (*every point is a shape*), každý kruh je grafickým útvarem. Proto strom na Obrázku 19 pravidlo *is-a* splňuje. Podobně například každý automobil je vozidlo, nebo každý pes je savec. Proto podle pravidla *is-a* lze definovat třídu automobilů jako podtřidu třídy vozidel a třídu psů jako podtřidu třídy savců.

Příklad 7.4. Vztah stěrač–automobil, neodpovídá pravidlu *is-a*, proto nelze definovat třídu stěračů jako podtřidu třídy automobilů. Koneckonců, má stěrač kola? Totéž platí také třeba pro vztah čtverec–úsečka, úsečka–bod.

Poznámka 7.5. Instance třídy musí mít všechny vlastnosti, které mají instance její nadtřídy (viz poznámku o stěrači a kolech v předchozím příkladě). Tento poznatek je důsledkem pravidla *is-a* a může pomáhat při ověřování správnosti návrhu hierarchie tříd. Neplatí ovšem bohužel obecně, někdy se může stát, že instance podtřídy některé vlastnosti instancí nadtřídy ztratí. K tomuto problému se vrátíme za chvíli.

Přímým předkem dané třídy je třída, která je jejím předkem a nemá žádného potomka, který by byl rovněž jejím předkem. Třída je *přímým potomkem* dané třídy, je-li tato třída jejím přímým předkem.

Poznámka 7.6. Mezi třídou a jejím přímým předkem tedy v hierarchii tříd není žádná jiná třída. Podobně se v hierarchii tříd nevyskytuje žádná třída mezi třídou a jejím přímým potomkem.

Nyní můžeme také upřesnit vztah mezi pojmy instance a přímé instance. Prvky tříd (v množinovém smyslu) se v objektovém programování nazývají jejich *instancemi*. Ke každému objektu ale existuje jediná třída, jíž je objekt *přímou instancí*. Je to vždy taková třída, že daný objekt již není instancí žádného jejího potomka.

⁵Cizí slovo *hierarchie* je použito v přeneseném významu, v němž označuje stupňovitou soustavu (hodnot, hodnotí apod.).

7.2. Určení předka při definici třídy

Definici třídy pomocí makra `defclass` je nyní potřeba rozšířit, aby umožnila určit místo definované třídy v hierarchii tříd.

Nová definice makra `defclass` je následující:

```
(defclass name parents slots)

name: symbol
parents: prázdný seznam nebo jednoprvkový seznam obsahující symbol
slots: seznam
```

Rozšíření předchozí definice spočívá v tom, že jako druhý argument makra `defclass` (parametr `parents`) lze kromě prázdného seznamu uvést i seznam obsahující jeden prvek. Pokud této možnosti využijeme, musí být tímto prvkem symbol označující třídu. Nově vytvářená třída se pak stane přímým potomkem této třídy.

Poznámka 7.7. V seznamu `parents` lze uvést i více přímých předků nové třídy. Tato volba, kterou CLOS umožňuje, by měla za důsledek, že by hierarchie tříd měla složitější strukturu než strukturu stromu. Tento jev se nazývá *vícenásobná dědičnost*. V této části textu se budeme zabývat pouze jednoduchou dědičností.

Definice tříd `shape`, `point` a `circle` podle předchozí podkapitoly (tedy ve struktuře z Obrázku 19 a s přesunem slotu `window` do třídy `shape`) by tedy vypadala takto:

```
(defclass shape ()
  ((window :initform nil)))

(defclass point (shape)
  ((x :initform 0)
   (y :initform 0)
   (color :initform :black)
   (thickness :initform 1)))

(defclass circle (shape)
  ((center :initform (make-instance 'point))
   (radius :initform 1)
   (color :initform :black)
   (thickness :initform 1)
   (filledp :initform nil)))
```

U třídy `shape` tedy nespécifikujeme žádného přímého předka, protože tato třída je na vrcholu stromu tříd grafických objektů. Bezprostředním předkem tříd `point` a `circle` je třída `shape`.

Podobně jako definice slotů přesuneme ze tříd `point` a `circle` do třídy `shape` i metody, které s nimi pracují. Proto metody `window`, `set-window` a `shape-mg-window` definujeme pro třídu `shape`:

```
(defmethod window ((shape shape))
  (slot-value shape 'window))

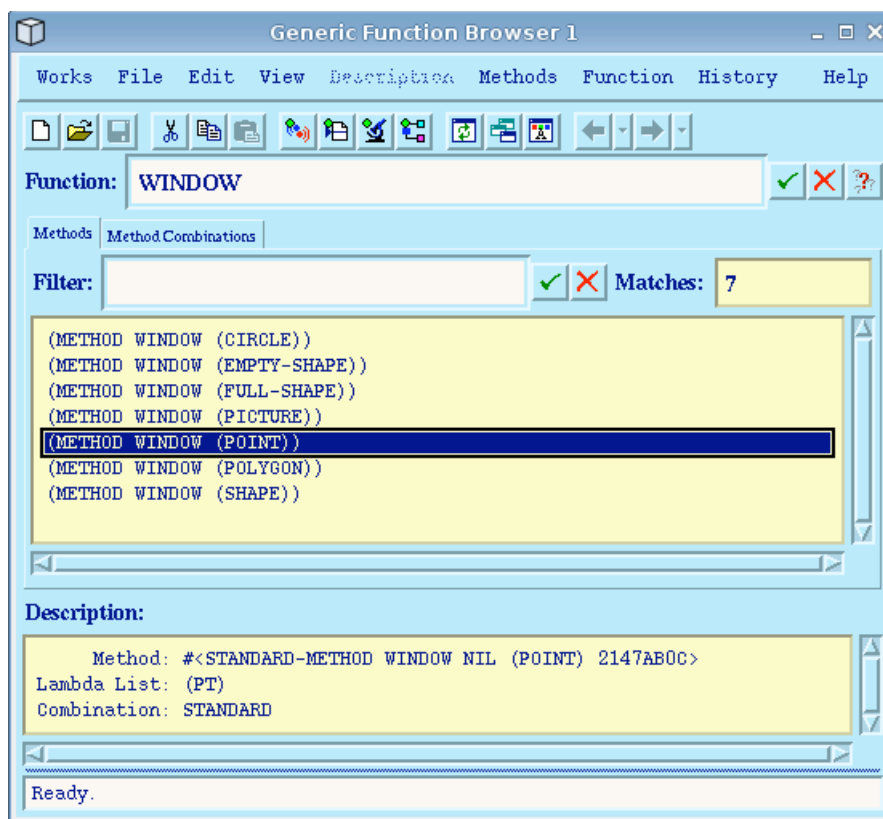
(defmethod set-window ((shape shape) value)
```



```
(setf (slot-value shape 'window) value)
shape)

(defmethod shape-mg-window ((shape shape))
  (when (window shape)
    (mg-window (window shape))))
```

Pozor, případné dřívější definice těchto metod pro jiné třídy zůstávají v platnosti. O tom se lze v prostředí LispWorks přesvědčit například pomocí nástroje „Generic Function Browser“, který umí ukázat všechny metody dané zprávy. Pokud máme například definovány všechny třídy a metody z předchozí kapitoly a pak vyhodnotíme třeba zde uvedenou definici metody `window` pro třídu `shape`, prohlížeč nám ukáže všechny aktuální definice metod pro zprávu `window` (viz Obrázek 20).



Obrázek 20: Prohlížeč generických funkcí

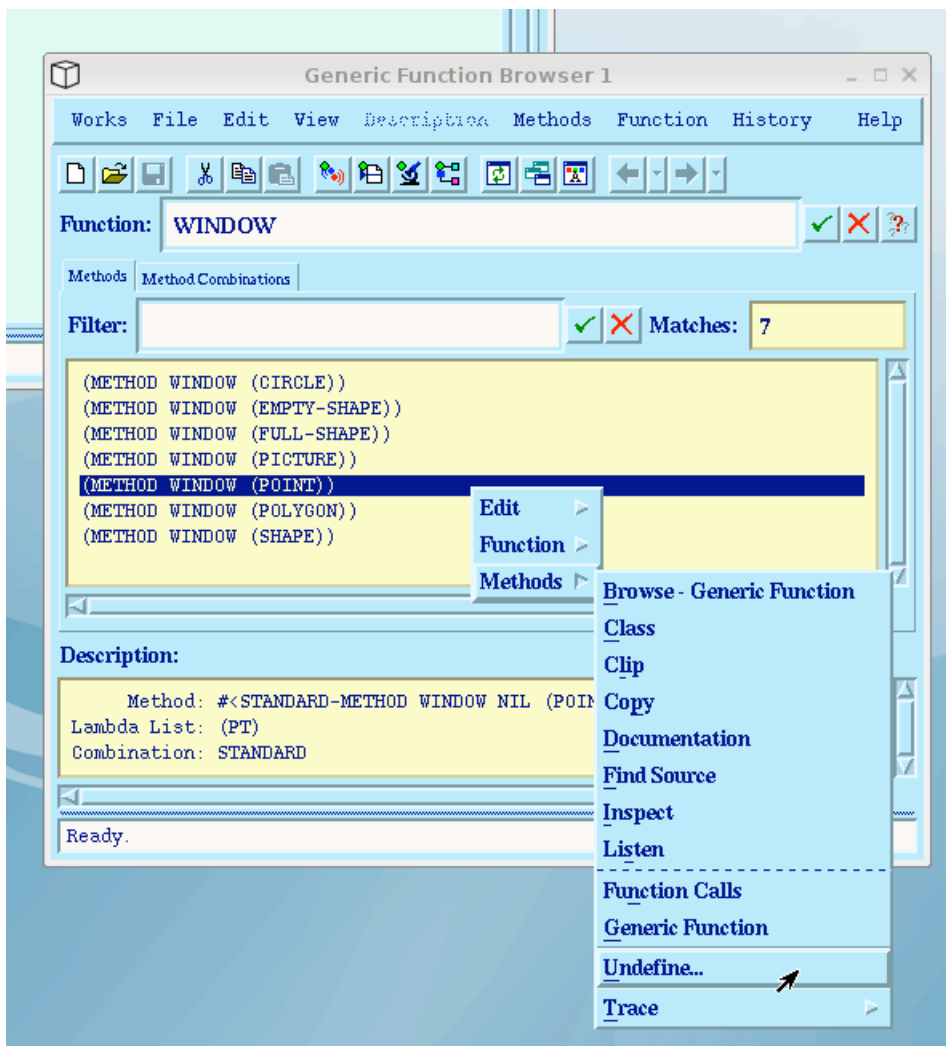
Pomocí tohoto nástroje nyní můžeme odstranit všechny metody, které potřebujeme. Obrázek 21 ukazuje, jak to lze udělat například s metodou `window` třídy `point`.

Pokud máme k dispozici zdrojový text metody, jejíž definici chceme zrušit, můžeme také použít jiný způsob: pomocí kontextové nabídky u definice samé (Obrázek 22), nebo odkazu na ni v seznamu definic (Obrázek 23).

Poznámka 7.8. Jinou možností, jak se zbavit nežádoucího stavu prostředí je samosebou ukončení aplikace a její opětovné spuštění. Jakkoliv je tato metoda účinná, její časté používání není vhodné — nevede nás totiž k pochopení problému, který řešíme.

7.3. Přiblížení běžným jazykům

Abychom se přiblížili současným základním v praxi používaným objektovým jazykům a osvojili si způsob práce v nich, přijmeme na nějakou dobu několik omezení,



Obrázek 21: Odstranění metody v prohlížeči generických funkcí

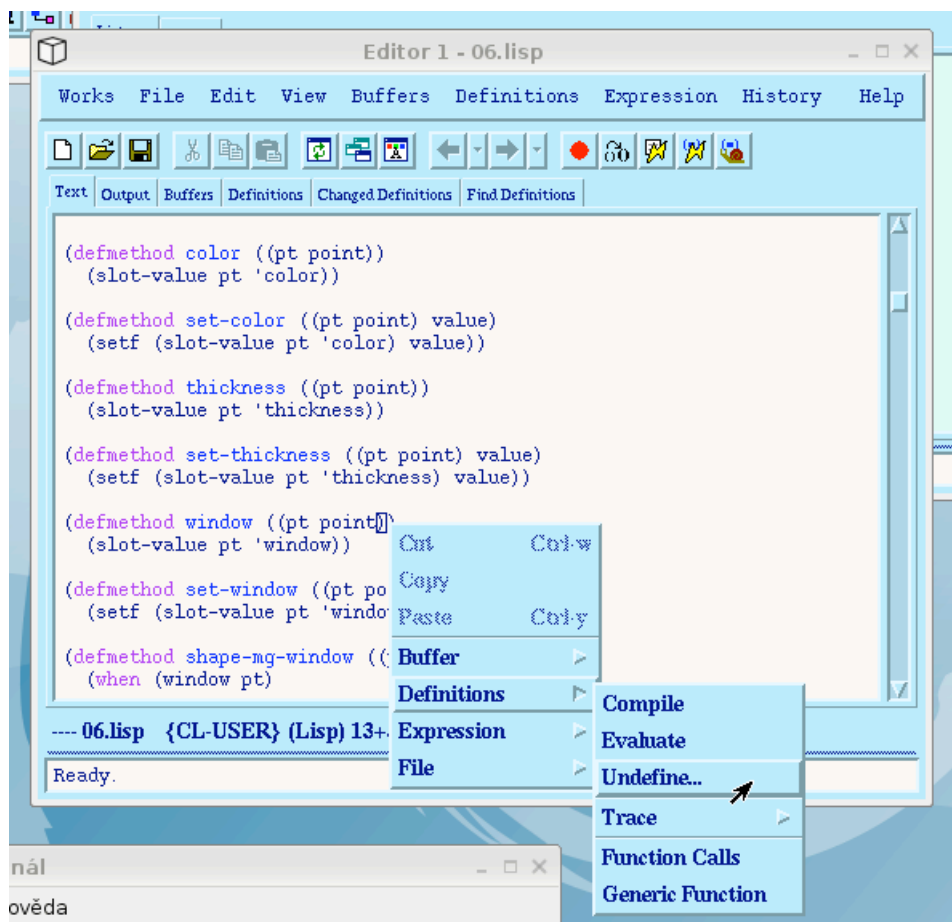
která tyto jazyky na rozdíl od Common Lispu stanovují.

Při definici metod budeme respektovat stromovou hierarchii tříd. Pokud pro danou zprávu definujeme metody pro několik tříd, vždy definujeme i metodu specializovanou na nějakého předka těchto tříd. Napodobíme tím situaci, se kterou se setkáváme ve staticky typovaných objektových jazycích, jako jsou například C++, objektový Pascal, Java, C#.

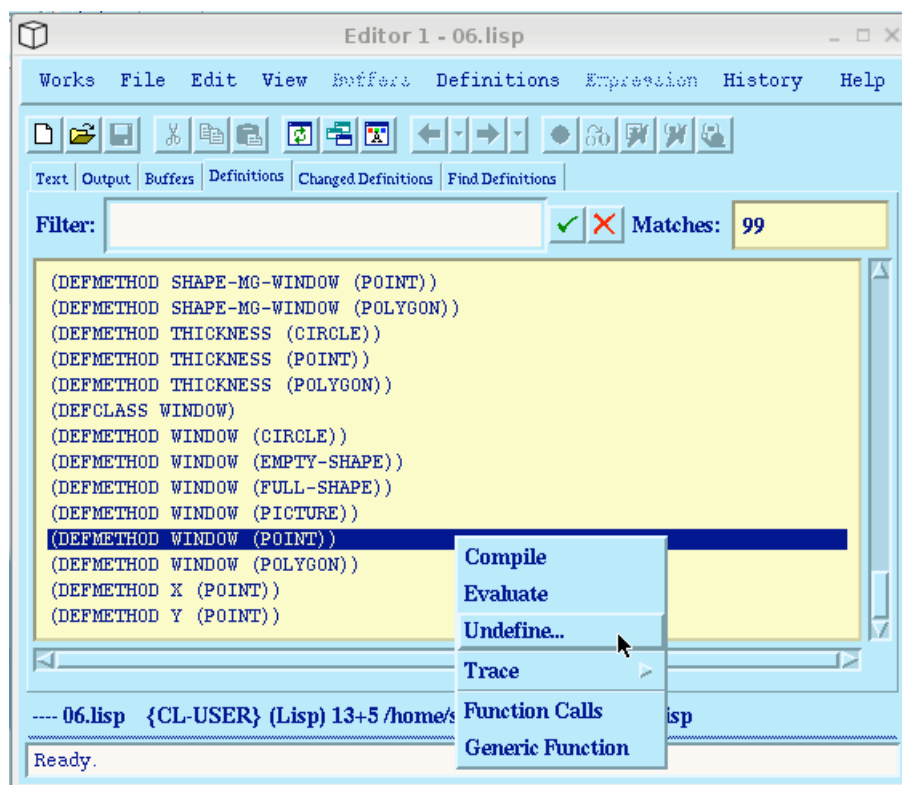
Metoda specializovaná na předka ostatních tříd nemusí dělat nic užitečného. V některých případech je dokonce vhodné zabránit jejímu volání. V Closu to můžeme řešit tak, že v kódu metody vyvoláme chybu. V některých objektových jazycích je možné takové metody označit jako *abstraktní*. V takových případech pak není nutné kód metody vůbec psát, o vyvolání chyby se postará kompilátor. V tomto textu budeme termín abstraktní metoda používat pro metody, které není vhodné přímo volat, ale které je nutno přepsat v podřizovaných třídách.

Poznámka 7.9. Zavádění abstraktních metod jinak není v Closu nutné, protože při definici metod nemusíme respektovat stromovou hierarchii. V tom se Clos podobá objektovým jazykům založeným na posílání zpráv, jako je například SmallTalk.

Syntax většiny objektových jazyků vyžaduje definovat metody současně s definicí třídy. V těchto jazycích se říká (a tímto způsobem se uvažuje), že metody *patří třídám* (a jsou tak i vnitřně implementovány). Abychom tento přístup napodobili, budeme vždy všechny metody dané třídy uvádět bezprostředně za její definicí.



Obrázek 22: Odstranění metody v editoru



Obrázek 23: Odstranění metody v seznamu definic

7.4. Hierarchie tříd grafických objektů

Využijeme získaných poznatků k uspořádání tříd grafických objektů do stromové hierarchie podle dědičnosti. Naším cílem bude eliminovat co nejvíce opakujícího se kódu z předchozího návrhu, současně ale nikdy neztratíme nadhled a budeme stále přihlížet k pravidlu *is-a*.

Provedené změny budou zpětně kompatibilní. Veškerý kód napsaný pro třídy grafických objektů z předchozí kapitoly bude fungovat i v nové verzi.

Poznámka 7.10. Uživatel naší nové verze již ale bude počítat s námi zavedenou hierarchií tříd. Pokud budeme chtít zachovat zpětnou kompatibilitu, nebude ji možné v budoucnu měnit. Proto je třeba věnovat návrhu velkou pozornost.

Nejprve se podívejme na sloty a metody, které bude vhodné přesunout do třídy `shape`. Kromě slotu `window` jsme ve více než jedné třídě definovali následující sloty: `color`, `thickness`, `filledp`, `items`. Metody, které s těmito sloty bezprostředně pracují (realizují jejich zapouzdření), jsou `color`, `set-color`, `thickness`, `set-thickness`, `filledp`, `set-filledp`, `items`, `set-items`. U těchto slotů a metod je tedy možné zvážit přesunutí do třídy `shape`.

U slotu `color` a souvisejících metod je situace poměrně jasná. Tento slot a tyto metody slouží k implementaci barvy grafických objektů. Jistě má smysl, aby každý grafický objekt nesl informaci o barvě, kterou je vykreslován.

Poznámka 7.11. Pro přesnost: v některých speciálních případech není informace o barvě grafického objektu využita. To platí zejména pro třídy `empty-shape` a `picture`. Barva prázdného grafického objektu se nikdy při jeho vykreslování neprojeví, u instancí třídy `empty-shape` tedy nemá smysl informaci o barvě udržovat. U obrázku je situace podobná, protože se každý jeho prvek kreslí svou vlastní barvou. Kvůli zachování struktury stromu dědičnosti ale tyto výjimky nebudeme brát v úvahu. (V dalším také uvidíme, že u třídy `picture` lze informaci o barvě s výhodou využít jinak).

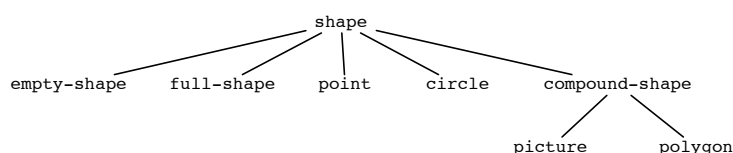
Z podobných důvodů jako u informace o barvě přesuneme i sloty `thickness` a `filledp` a související metody do třídy `shape`.

Co provést se slotem `items`? Tento slot, který najdeme u tříd `picture` a `polygon`, slouží k uložení objektů, ze kterých se instance těchto tříd skládají (v případě `polygonů` to mohou být body, u obrázků libovolné grafické objekty). U ostatních tříd nemá smysl tento slot definovat — jistě není vhodné a logické hovořit o seznamu podobjektů u bodů nebo kružnic.

Co tedy mají `polygony` a `obrázky` společného? V obou případech se jedná o objekty, které se skládají z jiných objektů, tedy o složené objekty (pravidlo *is-a*: `polygon` i `obrázek` je složený objekt). Má tedy smysl zavést společného předka tříd `polygon` a `picture` a metody společné těmto třídám přesunout do něj. Uvidíme, že těchto metod nebude málo a že se struktura našeho zdrojového kódu pročistí.

Třidu složených grafických objektů nazveme `compound-shape`.

Strom dědičnosti tříd grafických objektů je znázorněn na Obrázku 24.



Obrázek 24: Strom dědičnosti tříd grafických objektů

Funkčnost tříd `polygon` a `picture`, která souvisí s tím, že se jedná o složené objekty (například, ale nejen, získávání a nastavování seznamu obsažených objektů) bude pomocí třídy `compound-shape` oddělena od funkčnosti pro tyto třídy specifické. To je příklad obecnějšího principu, podle něhož se různé třídy používají k implementaci různých funkcností. V dobře navrženém systému tříd je jasné a přehledně rozdělena zodpovědnost za různé úkoly mezi jednotlivé třídy.

Poznámka 7.12. V objektovém programování tedy třídy hrají roli modulů.

Uvedme upravené definice našich tříd a některých metod a všimněme si, jak se proti předchozí kapitole všechno zjednodušilo.

Třída `shape` a základní metody:

```
(defclass shape ()
  ((color :initform :black)
   (thickness :initform 1)
   (filledp :initform nil)
   (window :initform nil)))

(defmethod window ((shape shape))
  (slot-value shape 'window))

(defmethod set-window ((shape shape) value)
  (setf (slot-value shape 'window) value)
  shape)

(defmethod shape-mg-window ((shape shape))
  (when (window shape)
    (mg-window (window shape))))

(defmethod color ((shape shape))
  (slot-value shape 'color))

(defmethod set-color ((shape shape) value)
  (setf (slot-value shape 'color) value)
  shape)

(defmethod thickness ((shape shape))
  (slot-value shape 'thickness))

(defmethod set-thickness ((shape shape) value)
  (setf (slot-value shape 'thickness) value)
  shape)

(defmethod filledp ((shape shape))
  (slot-value shape 'filledp))

(defmethod set-filledp ((shape shape) value)
  (setf (slot-value shape 'filledp) value)
  shape)
```

Třída `point`. Metody pro práci s kartézskými a polárními souřadnicemi neuvádíme, protože se nezměnily. Metody, které pracují se sloty `color` a `thickness`, již nepotřebujeme, protože jsou přesunuty do třídy `shape`.

```
(defclass point (shape)
  ((x :initform 0)
   (y :initform 0)))
```

Třída `circle`. Metody `radius`, `set-radius` a `center` jsou stejné jako dříve, proto je neuvádíme.

```
(defclass circle (shape)
  ((center :initform (make-instance 'point))
   (radius :initform 1)))
```

Nově zavedená třída `compound-shape` obsahuje slot `items` a metody pro práci s ním. Tyto metody jsou poněkud složitější, proto zatím uvedeme pouze definici třídy:

```
(defclass compound-shape (shape)
  ((items :initform '())))
```

Třídě `polygon` zbyde ze všech slotů pouze slot `closedp`. Zatím uvedeme pouze definici třídy. Metody pro práci se slotem `closedp` jsou stejné jako dříve.

```
(defclass polygon (compound-shape)
  ((closedp :initform t)))
```

U třídy `picture` se pokusíme o malé vylepšení. U některých obrázků je výhodné moci nastavovat současně jednu barvu všem jejich prvkům. V dalším textu ukážeme, jak je možné tuto novou službu implementovat. Jelikož se tato možnost nehodí ve všech případech, učiníme tuto službu volitelnou: zavedeme slot `propagate-color-p`, který bude obsahovat informaci, zda má obrázek při nastavení své barvy nastavit také barvu obsažených grafických útvarů. Vzhledem k tomu, že chceme zachovat zpětnou kompatibilitu, bude počáteční hodnota tohoto slotu `nil`.

V této podkapitole ukážeme pouze novou definici třídy `picture`:

```
(defclass picture (compound-shape)
  ((propagate-color-p :initform nil)))
```

7.5. Přepisování metod

Víme, že podle principu polymorfismu, lze v každé třídě definovat jinou obsluhu téže zprávy. Současně jsme zavedli omezení, že pokud jsou metody pro tutéž zprávu definovány ve více třídách musí být definovány i pro nějakého jejich předka.

To vede často k tomu, že objekt, kterému je poslána zpráva, má na výběr mezi více metodami, které je možno jako obsluhu zprávy vykonat (každý objekt totiž může být instancí více tříd, z nichž každá může mít příslušnou metodu definovanou). V takové situaci objektový systém vykoná metodu, definovanou pro nejspecifičtější (tedy ve stromu dědičnosti nejnižší stojící) třídu, jíž je objekt instancí.

Konkrétněji: pokud je objektu zaslána zpráva, objektový systém hledá její obsluhu nejprve ve třídě, jíž je objekt přímou instancí. Pokud ji tam najde, zavolá ji. Nenajde-li ji, pokračuje ve hledání metody v bezprostředním předku třídy. Takto pokračuje, dokud metodu nenajde.

Poznámka 7.13. Popsaným způsobem postupuje většina objektových jazyků. Existují ale výjimky; například v programovacím jazyce Beta se nehledají metody v hierarchii dědičnosti ze spodu nahoru, ale naopak. V Common Lispu je toto chování do značné míry programovatelné.

Každá třída má tedy k dispozici veškerou funkčnost svých předků. Říkáme, že třída *dědí* metody po svých předcích.

Nejsme-li spokojeni se zděděnou metodou, můžeme ve třídě definovat metodu novou. V takovém případě říkáme, že ve třídě zděděnou metodu *přepisujeme*.

Ukažme si nejprve uvedený princip na transformacích grafických objektů, tedy na metodách `move`, `rotate` a `scale`.

Především rozhodneme, jak tyto metody definujeme ve třídě `shape`. U instancí této třídy nemáme dostatek informací k tomu, abychom mohli uvést konkrétní definice. Máme tedy pouze dvě rozumné možnosti:

1. definovat metody tak, aby nic nedělaly,
2. definovat metody tak, aby vyvolaly chybu.

Při volbě mezi těmito možnostmi je třeba brát v úvahu dopad na implementátora potomků třídy `shape`.

Výhodou první možnosti je, že usnadňuje definici a testování potomků třídy `shape`. Čím více metod je definováno tímto způsobem, tím rychleji jsme schopni implementovat potomka této třídy, který sice není úplně funkční, ale dá se testovat. Tato možnost tedy pomáhá dodržovat důležitou programátorskou zásadu: udržovat program co nejčastěji ve zkompileovatelném a spustitelném stavu.

Druhá možnost připadá v úvahu v situaci, kdy by metoda, která nedělá nic, mohla uvést aplikaci do nekonzistentního stavu. Pokud je tedy charakter metody takový, že po jejím provedení je uživatel závislý na jejím výsledku, je třeba autora nové třídy tímto způsobem donutit, aby tuto metodu přepsal.

Poznámka 7.14. Ve druhém případě se tedy jedná o abstraktní metody, ve smyslu uvedeném výše.

Metody `move`, `rotate` a `scale` pro třídu `shape` definujeme tak, že pouze vrátí transformovaný grafický útvar jako svou hodnotu:

```
(defmethod move ((shape shape) dx dy)
  shape)

(defmethod rotate ((shape shape) angle center)
  shape)

(defmethod scale ((shape shape) coeff center)
  shape)
```

Ve třídách `empty-shape` a `full-shape` tyto metody přepisovat nemusíme, protože implementace ve třídě `shape` dělá to, co je pro tyto třídy třeba (nic).

Ve třídách `point` a `circle` zůstaneme u původní implementace — metody třídy `shape` tedy přepíšeme.

U polygonů a obrázků vidíme, že původní implementace je v obou třídách stejná. Přesuneme ji tedy do třídy `compound-shape` (a přepíšeme původní metody třídy `shape`). Nejdříve si ale napíšeme pomocnou metodu `send-to-items`, kterou mnohokrát využijeme a která slouží k poslání téže zprávy všem prvkům složeného grafického objektu.


```

(defmethod send-to-items ((shape compound-shape)
                          message
                          &rest arguments)
  (dolist (item (items shape))
    (apply message item arguments))
  shape)

(defmethod move ((shape compound-shape) dx dy)
  (send-to-items shape #'move dx dy)
  shape)

(defmethod rotate ((shape compound-shape) angle center)
  (send-to-items shape #'rotate angle center)
  shape)

(defmethod scale ((shape compound-shape) coeff center)
  (send-to-items shape #'scale coeff center)
  shape)

```

7.6. Volání zděděné metody

Víme, že díky dědičnosti je metoda definovaná pro třídu automaticky definovaná i pro všechny její podtřídy. Pokud ale v některé podtřídě tuto metodu přepíšeme novou metodou, spustí se při zaslání příslušné zprávy instanci této podtřídy tato nová metoda. V některých případech je vhodné použít kombinovaný přístup: v podtřídě původní metodu využít, ale přitom ji rozšířit o nový kód, pro danou podtřídu specifický.

Podívejme se nejprve na metodu `set-window`. Tuto metodu jsme již definovali ve třídě `shape` způsobem, který je vyhovující pro většinu grafických objektů:

```

(defmethod set-window ((shape shape) value)
  (setf (slot-value shape 'window) value)
  shape)

```

V předchozí kapitole jsme se ale u třídy `picture` s touto definicí nespokojili:

```

(defmethod set-window ((shape picture) value)
  (dolist (item (items shape))
    (set-window item value))
  (setf (slot-value shape 'window) value)
  shape)

```

V metodě `set-window` v třídě `picture` jsme tedy nejprve nastavili okno všem objektům obsaženým v obrázku a pak jsme teprve nastavili okno i jemu.

Takto definovanou metodu `set-window` pro třídu `picture` lze beze zbytku zkopírovat i do vytvářené nové verze našeho grafického systému. Bude to ale mít dvě podstatné vady:

1. Opakovaný kód. Poslední dva řádky metody jsou přesně stejné jako tělo metody přepisované (tj. metody třídy `shape`).

2. Porušení zásady zapouzdření. Metoda `set-window` třídy `picture` vychází ze znalosti vnitřní konstrukce třídy `shape`, tedy toho, že její instance po obdržení zprávy `set-window` nastavují hodnotu slotu `window`.

Pokud by se v budoucnu změnila implementace metody `set-window` třídy `shape`, bylo by zřejmě nutné stejně změnit i její implementaci ve třídě `picture`, a to z obou uvedených důvodů.

Objektové jazyky nabízejí způsob, jak se těchto nedostatků zbavit. Uvnitř těla libovolné metody můžeme na zvoleném místě zavolat prepisovanou metodu.

V Common Lispu je za tímto účelem zavedena lokální funkce `call-next-method`. Pokud tuto funkci zavoláme (bez parametrů) v těle metody, objektový systém zavolá metodu předka, kterou prepisujeme.

Pomocí funkce `call-next-method` tedy lze odstranit oba uvedené nedostatky metody `set-window`:

```
(defmethod set-window ((pic picture) value)
  (dolist (item (items pic))
    (set-window item value))
  (call-next-method))
```

Implementaci je dále možno zjednodušit pomocí metody `send-to-items`:

```
(defmethod set-window ((pic picture) value)
  (send-to-items pic #'set-window value)
  (call-next-method))
```

V našem systému už zbývají převést do nové podoby dvě věci: kreslení grafických objektů a práce se seznamem prvků složených grafických objektů. Pojdme se stručně na oba problémy podívat.

U některých tříd postupovalo v první verzi kreslení grafických objektů podle stejného vzoru: nejprve se metodou `set-mg-params` nastavily potřebné grafické parametry okna knihovny `micro-graphics` a potom se objekt metodou `do-draw` vykreslil. Tento postup je vhodný k tomu, aby byl definován obecně ve třídě `shape`:

```
(defmethod draw ((shape shape))
  (set-mg-params shape)
  (do-draw shape))
```

Autoři potomků třídy `shape` nyní nemusí prepisovat metodu `draw`, pouze, pokud je třeba, metody `set-mg-params` a `do-draw`.

Metodu `set-mg-params` napíšeme tak, že nastaví všechny parametry okna podle hodnot uložených ve slotech. Tento přístup zbaví některé třídy nutnosti metodu prepisovat:

```
(defmethod set-mg-params ((shape shape))
  (let ((mgw (shape-mg-window shape)))
    (mg:set-param mgw :foreground (color shape))
    (mg:set-param mgw :filledp (filledp shape))
    (mg:set-param mgw :thickness (thickness shape)))
  shape)
```

Metoda `do-draw` nemůže ve třídě `shape` dělat nic. Zbývá tedy rozhodnout, zda je vhodné definovat ji jako abstraktní. Přikloníme se k prázdné implementaci:

```
(defmethod do-draw ((shape shape))
  shape)
```

Poznámka 7.15. Rozhodnutí nenutit autora potomka třídy `shape` k přepsání této metody není jednoznačně správné a je motivováno obecným přístupem používaným v Common Lispu; v některých objektových jazycích je zvykem programátory více nutit k určitým postupům (to se týká hlavně potomků jazyka C: C++, C#, Java). V těchto jazycích bychom zřejmě spíše použili abstraktní metodu.

Všimněme si, že nám vznikají dva druhy metod: jedny jsou určeny spíše k tomu, aby je uživatel volal (tj. aby objektům zasílal příslušné zprávy; to se týká metody `draw`), zatímco u druhých se to neočekává (`set-mg-params`, `do-draw`). Metody druhého typu jsou pouze připraveny k tomu, aby byly v nově definovaných třídách přepsány. V některých jazycích se metodám druhého typu říká *chráněné metody* (*protected methods*). Toto rozdělení bude v další části textu ještě výraznější.

Podívejme se nyní na implementaci kreslení u potomků třídy `shape`. U třídy `circle` není nutno přepisovat ani metodu `draw`, ani metodu `set-mg-params`. Stačí pouze definice metody `do-draw` tak, jak byla uvedena v předchozí kapitole.

U třídy `point` je kreslení poněkud netypické — tato třída ignoruje obsah slotu `filledp` a před kreslením nastavuje hodnotu příslušného grafického parametru knihovny `micro-graphics` na `t`. To je vhodná příležitost k volání zděděné metody v metodě `set-mg-params`:

```
(defmethod set-mg-params ((pt point))
  (call-next-method)
  (mg:set-param (shape-mg-window pt) :filledp t)
  pt)
```

Metodu `draw` třídě `point` nedefinujeme, metoda `do-draw` zůstává stejná jako dříve.

U třídy `empty-shape` není nutno ohledně kreslení definovat nic. Stačí implementace zděděná po třídě `shape`. Naopak třída `full-shape` je značně netypická; přepisujeme metodu `set-mg-params` i `do-draw`:

```
(defmethod set-mg-params ((shape full-shape))
  (mg:set-param (shape-mg-window shape)
                :background
                (color shape))
  shape)

(defmethod do-draw ((shape full-shape))
  (mg:clear (shape-mg-window shape))
  shape)
```

U instancí třídy `polygon` je třeba při nastavování grafických parametrů okna nastavit i parametr `closedp`. Proto přepíšeme metodu `set-mg-params`:

```
(defmethod set-mg-params ((poly polygon))
  (call-next-method)
  (mg:set-param (shape-mg-window poly)
                 :closedp
                 (closedp poly))
  poly)
```

(všimněte si volání zděděné metody). Metoda `do-draw` je stejná jako dříve.

Kreslení instancí třídy `picture` můžeme nechat beze změny.

Teď se podívejme na práci se slotem `items` u složených grafických objektů. Metodu `items`, kterou jsme dříve definovali pro třídy `polygon` a `picture` zvlášť, můžeme beze změny přesunout do třídy `compound-shape`:

```
(defmethod items ((shape compound-shape))
  (copy-list (slot-value shape 'items)))
```

Metody `set-items` dělaly v původním návrhu v každé ze tříd `polygon` a `picture` něco trochu jiného, najdeme mezi nimi ale podobnosti: každá z nich nejprve testovala typovou správnost nastavovaného seznamu položek, pak jej uložila do slotu `items` a všem jeho prvkům správně poslala zprávu `set-window`. Rozdíl tedy spočíval pouze v testování typu prvků seznamu. To můžeme vydělit do zvláštní metody `check-items` a zbytek nechat společný:

```
(defmethod set-items ((shape compound-shape) value)
  (check-items shape value)
  (setf (slot-value shape 'items) (copy-list value))
  (send-to-items shape #'set-window (window shape))
  shape)
```

Úkolem metody `check-items` bude otestovat, zda všechny prvky daného seznamu mají typ požadovaný pro prvky daného složeného grafického objektu (pro `polygony` jsou to body, pro obrázky libovolné grafické objekty) a v případě negativního výsledku vyvolat chybu. Tuto metodu můžeme obecně napsat tak, aby prošla všechny prvky seznamu a každý otestovala zvlášť v metodě `check-item`, která již bude implementována pro obrázky a `polygony` zvlášť.

```
(defmethod check-items ((shape compound-shape) item-list)
  (dolist (item item-list)
    (check-item shape item))
  shape)
```

Nyní máme na výběr, zda metodu `check-item` napsat ve třídě `compound-shape` jako prázdnou (tj. aby nedělala nic), nebo jako abstraktní (tj. aby vyvolala chybu). V tomto případě poprvé bez váhání použijeme druhou možnost. Pokud by totiž někdo navrhoval dalšího potomka třídy `compound-shape`, je nezbytné, aby tuto metodu přepsal — v případě, že by tato metoda nekontrolovala typ nastavovaných prvků složeného objektu, mohla by způsobit nekonzistenci dat.

```
(defmethod check-item ((shape compound-shape) item)
  (error "Method check-item has to be rewritten"))
```

Po této reorganizaci zbývá třídám `polygon` a `picture` pouze přepsat metodu `check-item` (všimněte si také dalšího výrazného zjednodušení tohoto testu ve třídě `picture`, možnému díky zavedení třídy `shape`):

```
(defmethod check-item ((p polygon) item)
  (unless (typep item 'point)
    (error "Invalid polygon element type."))
  p)

(defmethod check-item ((pic picture) item)
  (unless (typep item 'shape)
    (error "Invalid picture element type."))
  pic)
```

Zbývá ještě dořešit již zmiňovanou možnost současného nastavení barvy všech prvků obrázku. Zde ukážeme hotový kód a necháme na čtenáři, aby si jej sám rozebral.

```
(defmethod send-to-items-set-color ((p picture) color)
  (send-to-items p #'set-color color)
  p)

(defmethod set-propagate-color-p ((p picture) value)
  (setf (slot-value p 'propagate-color-p) value)
  (when value
    (send-to-items-set-color p (color p)))
  p)

(defmethod set-color ((p picture) color)
  (call-next-method)
  (when (propagate-color-p p)
    (send-to-items-set-color p (color p)))
  p)

(defmethod set-items ((p picture) items)
  (call-next-method)
  (when (propagate-color-p p)
    (send-to-items-set-color p (color p)))
  p)
```

7.7. Inicializace instancí

Nově vytvářené instance je někdy třeba inicializovat složitějším způsobem, než jak to umožňuje volba `:initform` v definici třídy. U většiny programovacích jazyků k tomu slouží, zvláštní metody, nazývané *konstruktory*, v Common Lispu je možné použít metodu `initialize-instance`.

Funkce `make-instance`, která slouží k vytváření nových instancí tříd, vždy nejprve novou instanci vytvoří a pak jí pošle zprávu `initialize-instance`. V obsluze této zprávy tedy může nově vytvářený objekt provést inicializaci, na které nestačí volba `:initform` v definici třídy.

Poznámka 7.16. Zpráva `initialize-instance` patří ke zprávám, které objektům nezasíláme, ale pouze přepisujeme jejich metody.

Definice metody `initialize-instance` má následující tvar:

```
(defmethod initialize-instance ((var class) &key)
  ... kód metody ...)
```

Všimněme si symbolu `&key` v definici metody. Tento symbol je třeba v definici vždy uvést, jinak dojde k chybě. Význam symbolu `&key` pro nás ale není důležitý.

V metodě `initialize-instance` pro každou třídu je vždy nutno umožnit inicializaci instance definovanou v rodičích této třídy. Vždy je tedy nutno volat lokální funkci `call-next-method`.

Příklady použití metody `initialize-instance` najdete v kódu k této části textu.

7.8. Příklady

Kód k této části textu obsahuje složitější příklady použití dědičnosti. Doporučujeme čtenářům si tento kód projít.

8. Příklady

Nyní na chvíli opustíme budovaný grafický systém a uvedeme další příklady.

8.1. Symbolické výrazy poprvé

Naším cílem bude vytvořit jednoduchý systém na symbolické manipulaci s algebraickými výrazy, který bude umět tyto výrazy derivovat, dosazovat jeden výraz do druhého a částečně zjednodušovat.

Základní výrazy, se kterými budeme pracovat, budou číselné konstanty, proměnné a binární součty, součiny, rozdíly a podíly výrazů.

Příklad 8.1. Příklady výrazů: 1 , x , $x + y + 1$, $x * ((y - z) + 2/y)$.

Budeme se snažit celý systém vybudovat čistě funkcionálně, tedy bez vedlejšího efektu. Žádná metoda nebude své argumenty modifikovat, ale bude podle potřeby vytvářet nové objekty.

Poznámka 8.2. Toto řešení tedy umožní spojit výhody dvou základních paradigmat: objektového a funkcionálního. Pokud vám někdy někdo bude říkat, že jsou tato paradigmatata neslučitelná, vzpomeňte si na tento příklad.

Základní třídy výrazů budou následující: třída `const`, jejíž instance budou reprezentovat číselné konstanty, třída `var` s instancemi reprezentujícími proměnné (x , y a podobně) a třídy složených (binárních) výrazů `+-expr`, `--expr`, `*-expr` a `/-expr`, jejichž instance budou reprezentovat součty, rozdíly, součiny a podíly jednodušších výrazů.

V této podkapitole definujeme třídy výrazů bez určení bezprostředního předka, tedy stejným způsobem jako v kapitolách předcházejících kapitole o dědičnosti. Pak si všimneme, jak lze výhodně využít principu dědičnosti.

Je zřejmé, že každý z výrazů musí obsahovat nějaká data: konstanta svou hodnotu, proměnná svůj název (k tomuto účelu použijeme symbol) a binární výrazy dva podvýrazy. S použitím specifikace `:initform` uvedeme výchozí hodnoty těchto dat. Definice tříd by tedy mohly vypadat takto:

```
(defclass const ()
  ((value :initform 0)))

(defclass var ()
  ((name :initform 'x)))

(defclass +-expr ()
  ((expr-1 :initform (make-instance 'const))
   (expr-2 :initform (make-instance 'const))))

(defclass --expr ()
  ((expr-1 :initform (make-instance 'const))
   (expr-2 :initform (make-instance 'const))))

(defclass *-expr ()
  ((expr-1 :initform (make-instance 'const))
   (expr-2 :initform (make-instance 'const))))

(defclass /-expr ()
  ((expr-1 :initform (make-instance 'const))
   (expr-2 :initform (make-instance 'const))))
```

Vidíme, že definice tříd `+-expr`, `--expr`, `*-expr` a `/-expr` se podobají jak vejce vejci. Zatím tuto skutečnost pouze zaznamenáme a zkusíme pro tyto třídy definovat požadované metody.

Zprávu `deriv` budeme objektům reprezentujícím výrazy posílat s jedním parametrem, kterým bude proměnná (instance třídy `var`). Tento parametr bude určovat, podle jaké proměnné chceme výraz derivovat. Výsledkem zaslání zprávy by měl být nově utvořený výraz, který reprezentuje derivaci výrazu původního podle zadané proměnné.

Metody pro třídu `expr` a `var` (derivací konstanty je nula, derivací proměnné je 1 nebo 0, podle toho, zda derivujeme podle téže proměnné):

```
(defmethod deriv ((expr const) var)
  (make-instance 'const))

(defmethod deriv ((expr var) var)
  (let ((result (make-instance 'const)))
    (setf (slot-value result 'value)
          (if (eql (slot-value expr 'name)
                  (slot-value var 'name))
              1
              0)))
  result))
```

Dále derivace součtu (instance třídy `+-expr`):

```
(defmethod deriv ((expr +-expr) var)
  (let ((result (make-instance '+-expr)))
    (setf (slot-value result 'expr-1)
          (deriv (slot-value expr 'expr-1) var)
          (slot-value result 'expr-2)
          (deriv (slot-value expr 'expr-2) var))
    result))
```

Vidíme, že derivace součtu je definována rekurzivně, přesně tak, jak ji známe z matematiky.

Definujte sami metodu `deriv` i pro rozdíl, součin a podíl a sledujte, co mají společného. Všimněme si, že uvedené metody jsou opravdu čistě funkcionální.

Nyní definujme pro výrazy zprávu `expr-subst`. Tuto zprávu budeme výrazům zasílat se dvěma parametry: proměnnou a výrazem (oba tyto parametry jsou opět výrazy). Výsledkem by měl být výraz, který vznikne z původního výrazu (příjemce zprávy) tak, že se za proměnnou dosadí zadaný výraz.

Poznámka 8.3. Použili jsme název `expr-subst`, protože příhodné názvy `subst` a `substitute` jsou již v Common Lispu obsazeny.

Příklad 8.4. Když například objektu, reprezentujícímu výraz $x + y(x + z + 1)$ pošleme zprávu `expr-subst` s parametry, reprezentujícími výrazy x a $u - 2$, měli bychom dostat objekt, reprezentující výraz $(u - 2) + y((u - 2) + z + 1)$. Tento objekt, stejně jako objekt původní, je instancí třídy `+-expr`.

Metody zprávy `expr-subst` bude snadné definovat pro konstanty a proměnné:

```
(defmethod expr-subst ((expr const) var substituent)
  expr)

(defmethod expr-subst ((expr var) var substituent)
  (if (eq1 (slot-value expr 'name)
          (slot-value var 'name))
      substituent
      var))
```

Definice pro součet výrazů bude opět rekurzivní:

```
(defmethod expr-subst ((expr +-expr) var substituent)
  (let ((result (make-instance '+-expr)))
    (setf (slot-value result 'expr-1)
          (expr-subst (slot-value expr 'expr-1)
                      var
                      substituent))
    (setf (slot-value result 'expr-2)
          (expr-subst (slot-value expr 'expr-2)
                      var
                      substituent))
    result))
```

Opět si zkuste tuto metodu definovat pro rozdíl, součin a podíl.

8.2. Zápis a čtení symbolických výrazů

Při práci se zatím napsaným kódem rychle zjistíme, že se poměrně obtížně testuje a ladí — alespoň srovnáme-li to s možnostmi testování funkcí, které nepracují s objekty ve smyslu objektového programování, ale se seznamy, symboly, čísly, textovými řetězci a podobně. Je to především tím, že na rozdíl od většiny ostatních datových typů nejsou objekty ve smyslu objektového programování snadno čitelné (jejich obsah nelze zjistit přímo z textového výstupu příkazového řádku) a nedají se ani snadno při ladění zadávat (je potřeba je zadat voláním funkce `make-instance`).

Poznámka 8.5. To je jedna z nepříjemností, se kterými se při přechodu od funkcionálního k objektovému programování setkáme a kvůli které objektové programování vyžaduje k efektivní práci mnohem důmyslnější vývojová prostředí. V tomto příkladě se s tím ještě celkem snadno vyrovnáme, protože v něm i se standardními objekty pracujeme zásadně funkcionálně, v dalších příkladech už to tak snadné nebude.

Abychom tento problém vyřešili, zavedeme ke každému symbolickému výrazu jeho snadno čitelnou reprezentaci pomocí čísla, symbolu, nebo seznamu.

Konstanty (instance třídy `const`) budeme reprezentovat jejich hodnotami (tedy čísly), proměnné (instance třídy `var`) jejich názvy (tedy symboly) a binární výrazy seznamy zapisujícími jejich obsah v prefixovém tvaru. Každé třídě definujeme metodu `representation`, která k výrazu vypočte jeho reprezentaci. Implementace bude jednoduchá:

```
(defmethod representation ((expr const))
  (slot-value expr 'value))
```



```

(defmethod representation ((expr var))
  (slot-value expr 'name))

(defmethod representation ((expr +-expr))
  `(+ ,(representation (slot-value expr 'expr-1))
      ,(representation (slot-value expr 'expr-2))))

(defmethod representation ((expr --expr))
  `(- ,(representation (slot-value expr 'expr-1))
      ,(representation (slot-value expr 'expr-2))))

(defmethod representation ((expr *-expr))
  `( * ,(representation (slot-value expr 'expr-1))
      ,(representation (slot-value expr 'expr-2))))

(defmethod representation ((expr /-expr))
  `( / ,(representation (slot-value expr 'expr-1))
      ,(representation (slot-value expr 'expr-2))))

```

Dále definujeme funkci `parse`, která naopak z reprezentace výraz vytvoří. Nejprve několik pomocných definic:

```

(defvar *const-expr-class* 'const)
(defvar *var-expr-class* 'var)
(defvar *+-expr-class* '+-expr)
(defvar *--expr-class* '--expr)
(defvar *-expr-class* '*-expr)
(defvar */-expr-class* '/-expr)

```

Poznámka 8.6. Důvod, proč si definujeme globální proměnné na názvy tříd aritmetických výrazů zatím není zcela zřejmý. Je ale jasné, že tímto způsobem snadno zvýšíme modifikovatelnost našeho řešení.

```

(defun make-binary-expr (name expr-1 expr-2)
  (let ((result (make-instance
                    (cond
                     ((eql name '+) *+-expr-class*)
                     ((eql name '-') *--expr-class*)
                     ((eql name '*) *-expr-class*)
                     ((eql name '/') */-expr-class*))))))
    (setf (slot-value result 'expr-1)
          (parse expr-1)
          (slot-value result 'expr-2)
          (parse expr-2))
    result))

(defun make-const (value)
  (let ((result (make-instance *const-expr-class*)))
    (setf (slot-value result 'value) value)
    result))

(defun make-var (name)
  (let ((result (make-instance *var-expr-class*)))

```

```

      (setf (slot-value result 'name) name)
      result))

(defun parse (repr)
  (cond
    ((typep repr 'number) (make-const repr))
    ((typep repr 'symbol) (make-var repr))
    ((typep repr 'list) (apply 'make-binary-expr repr))
    ((typep repr 'expression) repr)))

```

Nyní můžeme většinu uvedených metod napsat čitelněji pomocí funkce `parse`. Ukážeme si to pouze na dvou metodách funkce `deriv`, ostatní případy přenecháme čtenářově pšli:

```

(defmethod deriv ((expr var) var)
  (parse
    (if (eql (slot-value expr 'name)
              (slot-value var 'name))
        1
        0)))

(defmethod deriv ((expr *-expr) var)
  (let ((expr-1 (slot-value expr 'expr-1))
        (expr-2 (slot-value expr 'expr-2)))
    (parse `(+ (* ,(deriv expr-1 var) ,expr-2)
                (* ,expr-1 ,(deriv expr-2 var))))))

```

Příklad 8.7. Definujte metodu `deriv` i pro třídy `--expr` a `/-expr`.

8.3. Symbolické výrazy a dědičnost

Shrňme si nyní základní nedostatky dosud uvedené implementace symbolických výrazů a všimněme si, že všechny porušují jednu základní programátorskou zásadu:

1. definice tříd `+ -expr`, `--expr`, `* -expr` a `/ -expr` jsou velmi podobné,
2. definice metod `expr-subst` pro tyto třídy jsou velmi podobné,
3. definice metod `representation` pro tyto třídy jsou velmi podobné.

Zmíněnou zásadou je samosebou požadavek nikdy nepsat tentýž kód na více míst programu současně. Pokud se ve zdrojovém textu programu opakuje vícekrát tentýž vzor, je třeba (pokud to programovací jazyk a vývojové prostředí dovolí) program upravit tak, aby vzor uváděl pouze na jednom místě a z ostatních míst se na něj odkazoval.

V objektovém programování se na řešení těchto problémů používá dědičnost — pokud by mělo více tříd obsahovat tentýž jev (data téhož typu a účelu, stejné nebo podobné metody), pokusíme se hierarchii tříd přeorganizovat tak, abychom tento jev umístili do jedné třídy, kterou učiníme společným předkem všech těchto tříd.

Třídy `+ -expr`, `--expr`, `* -expr` a `/ -expr` bezpochyby mají něco společného. Pokud prozkoumáme podobná místa jejich definic a definic jejich metod, snadno uhadneme, že jejich společným rysem je to, že instance všech těchto tříd jsou *binární výrazy*. Je tedy přirozené, definovat těmto třídám společného předka `binary-expression`, na nějž všechny společné rysy těchto tříd převedeme.

Poznámka 8.8. Při úpravách tohoto typu je třeba být velmi opatrný a nespokojit se s pouhým nalezením společných prvků tříd a jejich převedením na společného předka. Aby návrh hierarchie tříd dobře fungoval i v budoucnu, je nutné, aby měl co nejpřísnější logiku. Základním testem jeho správnosti by mělo být již dříve zmíněné kritérium *is-a*, které by mělo být bezvýhradně splněno. V našem případě tomu tak nepochybně je.

Poznámka 8.9. Přesto je možné vymyslet zobecnění našeho příkladu, v němž náhle podobné vztahy mezi třídami přestanou platit. Hlavní výhrada kritiků objektového programování a – zejména dědičnosti je, že téměř žádná hierarchie tříd neobstojí ve zkoušce časem. Dědičnost je současně velmi mocný, současně ale problematický příspěvek objektového programování; jako s každým silným nástrojem je i s ní potřeba zacházet opatrně a s rozmyslem. K této problematice se ještě vrátíme. Nevýhodou mnoha učebnic objektového programování je, že demonstrují dědičnost na jednoduchých, ale v praxi málo použitelných příkladech. V tomto textu se pokoušíme tomuto problému vyhnout, je to ale za tu cenu, že uváděné příklady jsou často rozsáhlé a složité.

Jednoduchou aplikací pravidla *is-a* také zjistíme, že všechny třídy symbolických výrazů by měly mít společného předka, který bude zahrnovat funkčnost všem symbolickým výrazům společnou. Definujeme proto třídu `expression`, již budou všechny ostatní třídy potomky. Brzy také uvidíme, že některé z námi vytvářených metod bude mít smysl definovat i pro tuto třídu. Změněné definice základních tříd:

```
(defclass expression ()
  ())

(defclass const (expression)
  ((value :initform 0)))

(defclass var (expression)
  ((name :initform 'x)))

(defclass binary-expression (expression)
  ((expr-1 :initform (make-instance 'const))
   (expr-2 :initform (make-instance 'const))))

(defclass +-expr (binary-expression)
  ())

(defclass --expr (binary-expression)
  ())

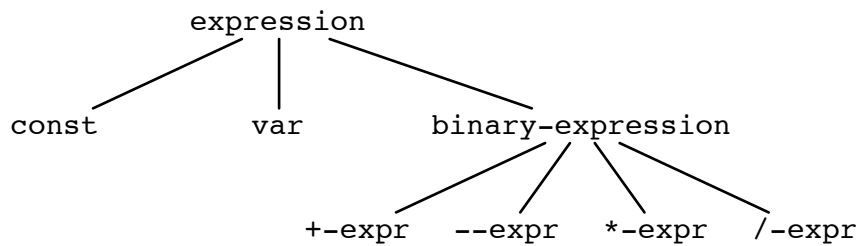
(defclass *-expr (binary-expression)
  ())

(defclass /-expr (binary-expression)
  ())
```

Strom dědičnosti těchto tříd je na [Obrázku 25](#).

Jelikož chceme při definici metod respektovat stromovou hierarchii, musíme pro zprávy `deriv`, `expr-subst` a `representation` definovat metody specializované na třídu `expression`. U všech si musíme rozmyslet, zda je definujeme jako abstraktní, nebo jestli umožníme jejich volání.

U metody `deriv` je situace jasná. Pokud neznáme konkrétní tvar symbolického výrazu, nevíme, jak jej derivovat. Proto si nemůžeme dovolit, aby tato metoda vracela



Obrázek 25: Strom dědičnosti tříd algebraických výrazů

nějakou hodnotu. Jakákoliv hodnota by to byla, byla by chybná. Proto tuto metodu implementujeme jako abstraktní:

```
(defmethod deriv ((expr expression) var)
  (error "Method deriv has to be rewritten"))
```

Metoda `expr-subst` bude také abstraktní, jelikož u obecného výrazu nelze rozhodnout, jaký výraz z něj po dosazení za některou proměnnou vznikne:

```
(defmethod expr-subst ((expr expression) var substituent)
  (error "Method expr-subst has to be rewritten"))
```

Poznámka 8.10. Zde by se mohl čtenář pozastavit, proč děláme takovou vědu z návrhu metod třídy `expression`, když stejně víme, že je všechny budeme v podřízených třídách přepisovat. Je třeba si ale uvědomit, že se někdy v budoucnu může stát, že budeme (my, nebo dokonce někdo jiný) nějaké nové potomky třídě `expression` přidávat. Na takové případy je vhodné náš systém připravit, a to i přesto, že nejsme schopni odhadnout (a neměli bychom se o to ani pokoušet), co se v budoucnu může stát. Metody `deriv` a `expr-subst` ukazují jedno z možných řešení, které je pro tuto situaci snad nejvhodnější: donutit budoucího tvůrce nových potomků třídy `expression` k přepsání těchto metod s tím, že pokud tak neučiní, nebudou mu metody fungovat.

U metody `representation` zvolíme jiný postup. Vzhledem k tomu, že jsme funkci `parse` navrhli tak, aby uměla pracovat i s objekty třídy `expression`, není nutné tuto metodu psát jako abstraktní:

```
(defmethod representation ((expr expression))
  expr)
```

Poznámka 8.11. Toto řešení je v tomto případě vhodnější a má také proti předchozímu řešení významnou výhodu: při vytvoření nového potomka třídy `expression` není nezbytné metodu `representation` ihned přepisovat, protože její správná, byť nedokonalá, funkce je zajištěna již třídou nadřízenou. Tento přístup nám tak pomáhá upravovat kód po co nejmenších krocích a udržovat jej co nejčastěji ve funkčním stavu. Metodu `representation` přepíšeme až v momentě, kdy se nám to bude hodit.

Zavedení třídy `expression` také umožní vylepšit funkci `parse` :

```
(defun parse (repr)
  (cond
    ((typep repr 'number) (make-const repr))
    ((typep repr 'symbol) (make-var repr))
    ((typep repr 'list) (apply 'make-binary-expr repr))
    ((typep repr 'expression) repr)
    (t (error "Cannot parse ~s" repr))))
```

Vraťme se nyní ke třídě `binary-expression` a pokusme se napravit problémy uvedené na začátku tohoto odstavce pod čísly 2. a 3. Podívejme se, jak byla definována metoda `expr-subst` pro třídu `+-expr`:

```
(defmethod expr-subst ((expr +-expr) var substituent)
  (let ((result (make-instance '+-expr)))
    (setf (slot-value result 'expr-1)
          (expr-subst (slot-value expr 'expr-1)
                      var
                      substituent)
          (slot-value result 'expr-2)
          (expr-subst (slot-value expr 'expr-2)
                      var
                      substituent))
    result))
```

Metody `expr-subst` pro třídy `--expr`, `*-expr` a `/-expr` mají implementace velmi podobné, protože všechny využívají pouze sloty `expr-1` a `expr-2`, které jsou definovány v třídě `binary-expression`. Jediné, v čem se liší, je použití symbolů pro typ výsledného výrazu. Proto lze místo čtyř definic napsat pouze jednu pro třídu `binary-expression`:

```
(defmethod expr-subst ((expr binary-expression)
                      var substituent)
  (let ((result (make-instance (type-of expr))))
    (setf (slot-value result 'expr-1)
          (expr-subst (slot-value expr 'expr-1)
                      var
                      substituent)
          (slot-value result 'expr-2)
          (expr-subst (slot-value expr 'expr-2)
                      var
                      substituent))
    result))
```

U zprávy funkce `representation` je problém podobný. Připomeňme si například metodu třídy `+-expr`:

```
(defmethod representation ((expr +-expr))
  `(+ ,(representation (slot-value expr 'expr-1))
      ,(representation (slot-value expr 'expr-2))))
```

Ostatní metody potomků třídy `binary-expression` se od této lišily pouze v symbolu v první položce vráceného seznamu. Zde je tedy možno opět přesunout většinu kódu do metody třídy `binary-expression`, jen je potřeba rozhodnout, jakým způsobem si tato metoda zjistí ke každému výrazu jeho symbol. Jedno z řešení je definovat na to novou zprávu:

```
(defmethod bin-expr-symbol ((expr binary-expression))
  (error "Method bin-expr-symbol has to be rewritten."))
```

```

(defmethod bin-expr-symbol ((expr +-expr))
  '+)

(defmethod bin-expr-symbol ((expr --expr))
  '-)

(defmethod bin-expr-symbol ((expr *-expr))
  '*)

(defmethod bin-expr-symbol ((expr /-expr))
  '/')

```

A metodu representation teď stačí napsat pro třídu binary-expression:

```

(defmethod representation ((expr binary-expression))
  `(,(bin-expr-symbol expr)
    ,(representation (slot-value expr 'expr-1))
    ,(representation (slot-value expr 'expr-2))))

```

8.4. Zpráva simplify a volání metody předka

Nyní se stručně zmíníme o zjednodušování algebraických výrazů. Definujeme zprávu simplify tak, aby volání

```
(simplify expr)
```

vrátilo jako výsledek algebraický výraz ekvivalentní výrazu expr, ale jednodušší.

Metoda simplify pro obecný algebraický výraz nemá ke zjednodušení výrazu dost informací. Proto nebude dělat nic a vrátí zadaný výraz jako výsledek. Podobně je třeba se zachovat u konstanty a proměnné, které se nijak zjednodušit nedají. Proto metodu pro třídu expression definujeme takto:

```

(defmethod simplify ((expr expression))
  expr)

```

a pro třídy const a var ji nebudeme přepisovat.

Poznámka 8.12. Tato metoda je typickým příkladem metody, která nedělá nic, ale kterou není vhodné definovat jako abstraktní.

U třídy binary-expression by zjednodušení mohlo vypadat takto:

```

(defmethod simplify ((expr binary-expression))
  (parse `(,(bin-expr-symbol expr)
            ,(simplify (slot-value expr 'expr-1))
            ,(simplify (slot-value expr 'expr-2)))))

```

Binární výraz tedy zjednodušíme tak, že vytvoříme binární výraz stejného typu, ale se zjednodušenými oběma podvýrazy.

U binárních výrazů tím ale ještě možnosti zjednodušování nekončí; u konkrétních typů binárních výrazů lze navíc využít jejich speciálních vlastností, například toho, že součet libovolného výrazu a nuly je tentýž výraz.

Metoda `simplify` u třídy `+ -expr` by tedy mohla dělat následující:

1. zjednodušit výraz jako obecný binární výraz (metodou třídy `binary-expression`),
2. aplikovat další zjednodušení vztahující se konkrétně ke třídě `+ -expr`.

Pro vhodné zkombinování obecné metody pro třídu `binary-expression` a metody konkrétní třídy lze použít funkci `call-next-method`. Například tedy u třídy `+ -expr`:

```
(defmethod zero-const-p ((expr expression))
  nil)

(defmethod zero-const-p ((expr const))
  (zerop (slot-value expr 'value)))

(defmethod simplify ((expr + -expr))
  (let* ((result (call-next-method))
        (expr-1 (slot-value result 'expr-1))
        (expr-2 (slot-value result 'expr-2)))
    (cond ((zero-const-p expr-1) expr-2)
          ((zero-const-p expr-2) expr-1)
          (t result))))
```

Podobně lze postupovat u dalších tří typů binárních výrazů.

Příklad 8.13. Definujte metody `simplify` i pro třídy `--expr`, `*-expr`, `/-expr` tak, aby uměly zjednodušovat alespoň následující výrazy:

- součin nuly a libovolného výrazu (zleva i zprava),
- součin jedničky a libovolného výrazu (zleva i zprava),
- rozdíl libovolného výrazu a nuly,
- podíl libovolného výrazu a jedničky.

Příklad 8.14. Jak byste obecně pro třídu `binary-expression` tuto metodu doplnili tak, aby binární výraz, který obsahuje dvě konstanty, zjednodušila na konstantu? Příklad:

```
(representation (simplify (parse '(+ 1 2)))) => 3
```

Příklad 8.15. Vyzkoušejte rozšiřitelnost systému symbolických výrazů tím, že do něj postupně doplníte unární výrazy: třídu `unary-expression`, která bude potomkem třídy `expression` a několik jejích potomků, například pro unární mínus, goniometrické funkce nebo přirozený logaritmus. Bude nutno přepracovat něco z předchozího kódu? Všimněte si rozdílu mezi přepracováváním objektové a neobjektové (funkce `parse`) části kódu.

Příklad 8.16. Šlo by derivování unárních výrazů nějak zjednodušit pomocí vzorce pro derivaci složené funkce?

Příklad 8.17. Nešlo by derivaci zavést jako binární výraz, který by měl jako jeden podvýraz derivovaný výraz a jako druhý proměnnou, podle které se derivuje? Zpráva `deriv` by se pak stala zbytečnou (nebo by zůstala jako pomocná) a tyto výrazy by se zderivovaly v metodě `simplify`. Bylo by nutno zásadně zasahovat do existujícího kódu? Příklad:

```
(representation (simplify (parse '(d (* x x) x)))) => (* 2 x)
```

Je vhodné, aby se při zjednodušování nahlásila chyba, pokud není druhý podvýraz výrazu pro derivaci proměnná.

Příklad 8.18. V metodách `expr-subst` a `deriv` třídy `constant` je shodný kód. Nešlo by jej nahradit voláním nějaké obecné metody na porovnávání výrazů? Představa je taková: metoda `equal-to-p` by k výrazu zjistila, zda je totožný se zadaným výrazem, takže například

```
(equal-to-p (parse 0) (parse 0))  
=> pravda  
  
(equal-to-p (parse 0) (parse x))  
=> nepravda  
  
(equal-to-p (parse '(+ x 1)) (parse '(+ x 1)))  
=> pravda  
  
(equal-to-p (parse '(+ 1 x)) (parse '(+ x 1)))  
=> pravda nebo nepravda?
```

8.5. Posloupnosti

Jako další příklad si ukážeme vlastní implementaci funkcí pro práci s posloupnostmi (lineárními datovými strukturami). Definujeme některé typy posloupností (jednorozměrná pole a seznamy) jako třídy a napíšeme obdobu některých funkcí Common Lispu, které pracují se seznamy, (například funkce `elt`, `length`, `find`) jako zprávy. Tím umožníme uživatelům definovat vlastní nové typy posloupností, se kterými půjdou tyto zprávy také používat, pokud pro ně napíší příslušné metody.

Nejprve definujme předka všech tříd posloupností, třídu `my-sequence`:

```
(defclass my-sequence ()  
  ())
```

Poznámka 8.19. Pojem posloupnosti je v Common Lispu již použit; proto jsou mnohé názvy, které by se nám hodily, obsazeny. V takových případech použijeme u našich názvů předponu „my-“.

Zprávy, které budeme posílat instancím třídy `my-sequence`, lze rozdělit do dvou skupin, podle toho, zda jejich metody musí být informovány o datové reprezentaci posloupností, nebo zda budou k přístupu k nim používat výhradně jiných zpráv. Zprávy prvního typu budou pracovat s posloupnostmi na fyzické úrovni (tj. budou závislé na datové reprezentaci jednotlivých typů posloupností), metody zpráv druhého typu

bude možné napsat bez znalosti implementačních detailů. Tyto metody pak budou použité i u dalších nově vzniklých typů posloupností a jejich autor je nebude muset vytvářet znovu.

Zprávám prvního typu budeme říkat *zprávy fyzické úrovně*, zprávám druhého typu *zprávy logické úrovně*.

Naším cílem je určit co nejmenší množinu zpráv fyzické úrovně, které budou s posloupnostmi pracovat přímo a které vytvoří abstraktní bariéru pro všechny ostatní zprávy.

Poznámka 8.20. Čím méně bude zpráv fyzické úrovně, tím snadnější bude definice nových tříd posloupností s veškerou funkcí, kterou budou poskytovat zprávy logické úrovně.

Metody zpráv logické úrovně pak definujeme ve třídě `my-sequence` jako abstraktní.

8.6. Zprávy fyzické úrovně

Zprávy fyzické úrovně budou využívat objekt *pozice*, který bude reprezentovat (způsobem, který si implementátor konkrétních typů posloupností může sám stanovit) ukazovátka na konkrétní pozici v posloupnosti. Pomocí tohoto objektu bude možno získávat a nastavovat hodnotu posloupnosti na dané pozici.

Poznámka 8.21. Objekt *pozice* jsme zavedli kvůli skrytí způsobu určení umístění prvků v posloupnosti. Pro každý typ posloupnosti bude možno určovat polohu prvků jiným, pro příslušný typ nejvhodnějším způsobem: víme, že u jednorozměrných polí se k prvkům přistupuje zcela jinak, než u seznamů. Bez této abstrakce by také nebylo možné obecně definovat níže uvedené metody logické úrovně.

Vzhledem ke způsobu procházení posloupností (použitým například v níže uvedených metodách `my-length` a `my-find`) vyžadujeme, aby bylo pomocí objektu *pozice* specifikovat i jednu pozici před prvním a za posledním prvkem posloupnosti.

V následujícím textu ukážeme, jak objekt *pozice* definujeme pro pole a seznamy.

Metody pro zprávy fyzické úrovně ve třídě `my-sequence`:

Zprávy `after-end-p` resp. `before-start-p` rozhodují, zda je daná pozice za koncem, resp. před začátkem posloupnosti. Tyto zprávy budeme používat k testování ukončení iterace přes prvky posloupnosti. Abstraktní metody pro třídu `my-sequence`:

```
(defmethod after-end-p ((sequence my-sequence) position)
  (error "Method after-end-p has to be rewritten. "))

(defmethod before-start-p ((sequence my-sequence) position)
  (error "Method before-start-p has to be rewritten. "))
```

Následující zprávy vrátí první, resp. poslední pozici posloupnosti.

```
(defmethod first-position ((sequence my-sequence))
  (error "Method first-position has to be rewritten. "))

(defmethod last-position ((sequence my-sequence))
  (error "Method last-position has to be rewritten. "))
```

Zprávy `next-position` a `prev-position` vracejí k dané posloupnosti a pozici následující a předchozí pozici.

```
(defmethod next-position ((sequence my-sequence) position)
  (error "Method next-position has to be rewritten.))

(defmethod prev-position ((sequence my-sequence) position)
  (error "Method prev-position has to be rewritten.))
```

Jak jsme uvedli dříve, implementace pozic musí umožňovat i specifikaci jedné pozice před začátkem a jedné pozice za koncem posloupnosti. Proto následující dvě volání nesmí pro danou neprázdnou posloupnost `seq` skončit chybou a musí vrátit hodnotu *pravda*:

```
(before-start-p
  seq (prev-position seq (first-position seq))) => pravda

(after-end-p
  seq (next-position seq (last-position seq))) => pravda
```

Poslední dvojice zpráv, `position-item` a `set-position-item` slouží ke čtení a zápisu hodnoty posloupnosti na dané pozici:

```
(defmethod position-item ((seq my-sequence) pos)
  (error "Method position-item has to be rewritten.))

(defmethod set-position-item ((seq my-sequence) pos item)
  (error "Method set-position-item has to be rewritten.))
```

Tyto dvě zprávy by měly vyvolat chybu, pokud se pokoušíme číst nebo nastavit hodnotu mimo rozsah posloupnosti.

8.7. Zprávy logické úrovně

Uvedených osm zpráv fyzické úrovně lze použít k definici metod mnoha různých dalších zpráv pro práci s posloupnostmi. Jako příklad uvedeme několik základních (abychom zabránili kolizi se standardními názvy, opět používáme předponu `my`).

Zprávy `my-elt` a `my-set-elt` slouží ke čtení a nastavování hodnoty prvku posloupnosti daného číselného indexu (`elt` je zkratka slova *element*). Metody těchto zpráv lze napsat bez znalosti struktury konkrétní posloupnosti čistě pomocí výše uvedených zpráv fyzické vrstvy. Nejprve napíšeme pomocnou metodu `nth-pos`, která vrátí objekt pozice pro zadaný index tak, že projde postupně všechny pozice od začátku:

```
(defmethod nth-pos ((seq my-sequence) index)
  (labels ((iter (position i)
    (if (= i 0)
        position
        (iter (next-position seq position)
              (- i 1)))))
    (iter (first-position seq) index)))
```

Metody `my-elt` a `my-set-elt` jsou nyní již jednoduché:

```
(defmethod my-elt ((seq my-sequence) index)
  (position-item seq (nth-pos seq index)))

(defmethod my-set-elt ((seq my-sequence) index value)
  (set-position-item seq (nth-pos seq index) value))
```

Metoda `my-length` vrací délku (počet prvků) posloupnosti:

```
(defmethod my-length (seq)
  (labels ((iter (index pos)
            (if (after-end-p seq pos)
                index
                (iter (+ index 1)
                      (next-position seq pos)))))
    (iter 0 (first-position seq))))
```

Metoda `my-find` je obdobou známé funkce `find`. Prochází posloupnost zleva doprava a vrací první prvek, který je roven (pomocí funkce `eq1`) danému. Pokud metoda v hledání neuspěje, vrací `nil`.

```
(defmethod my-find ((seq my-sequence) elem)
  (labels ((iter (pos)
            (unless (after-end-p seq pos)
              (if (eq1 elem (position-item seq pos))
                  elem
                  (iter (next-position seq pos)))))
    (iter (first-position seq))))
```

Všechny uvedené metody přistupují k datům posloupností pomocí výše uvedených zpráv fyzické vrstvy. Proto budou fungovat pro všechny třídy posloupností, které budou mít definovány metody zpráv fyzické vrstvy.

Příklad 8.22. Definujte metodu `my-find-if` pro třídu `my-sequence` tak, aby se chovala stejně jako funkce `find-if` pro seznamy: metoda `my-find-if` by tedy měla mít jeden parametr, predikát, který by aplikovala postupně na všechny prvky posloupnosti, dokud by predikát nevrátil hodnotu *pravda*. Nalezený prvek by měla vrátit. V případě neúspěchu by měla vrátit `nil`.

Příklad 8.23. Definujte metodu `my-find-if-from-right`, která bude pracovat stejně jako metoda `my-find-if`, ale bude posloupnost procházet zprava doleva.

Příklad 8.24. Definujte metodu `my-find-if-from`, která vybere a zašle zprávu `my-find-if` nebo `my-find-if-from-right` podle dalšího parametru, který může nabývat hodnot `:left` nebo `:right`.

8.8. Jednorozměrná pole

Definujme nyní třídu `my-vector`, jejíž instance budou reprezentovat jednorozměrná pole — vektory. Vnitřně budeme jednorozměrná pole reprezentovat pomocí standardních jednorozměrných polí Common Lispu. Uveďme nejprve stručnou informaci o nich:

Podobně jako v jiných jazycích je v Common Lispu k dispozici typ vícerozměrného indexovaného pole. Tato pole jsou obecně heterogenní (tj. mohou obsahovat objekty různých typů) a mohou mít libovolný počet rozměrů počínaje nulovým. Všechny rozměry pole jsou indexovány celými čísly od nuly. K vytvoření pole slouží funkce `make-array`. Zde uvedeme její zjednodušenou syntax:

```
(make-array dimensions) => new-array
```

Funkce `make-array` vytváří a vrací nové pole podle hodnot parametrů. Parametr *dimensions* je seznam nezáporných celých čísel, který určuje velikost jednotlivých rozměrů vytvářeného pole. Pokud vytváříme jednorozměrné pole, je možno místo seznamu v parametru *dimensions* zadat přímo délku vytvářeného pole jako číslo (to bude náš případ v této kapitole).

Počáteční obsah takto vytvořeného pole není specifikován, pole je po vytvoření třeba naplnit. Příklady:

```
CL-USER 33 > (make-array 5)
#(NIL NIL NIL NIL NIL)

CL-USER 34 > (make-array '(4 6))
#2A((NIL NIL NIL NIL NIL NIL) (NIL NIL NIL NIL NIL NIL)
     (NIL NIL NIL NIL NIL NIL) (NIL NIL NIL NIL NIL NIL))
```

Poznámka 8.25. Uvedené příklady naznačují, že nová pole lze také zadávat přímo do zdrojového kódu pomocí notace `#(...)` u jednorozměrných a `#nA(...)` u vícerozměrných polí. Tento postup může být někdy výhodnější, zejména při testování. Takto zadaná pole je ovšem zakázáno měnit, protože by to znamenalo modifikaci zdrojového kódu za chodu programu, která je v Common Lispu zakázána.

Ke čtení prvků pole je definována funkce `aref`, kterou lze také v kombinaci s makrem `setf` použít k zápisu do pole:

```
CL-USER 35 > (setf a (make-array 5))
#(NIL NIL NIL NIL NIL)

CL-USER 36 > (aref a 1)
NIL
```

(Na výsledek se v tomto případě ale nelze spolehnout; řekli jsme, že počáteční obsah pole není specifikován. Je třeba jej nejprve nastavit.)

```
CL-USER 37 > (setf (aref a 1) "a")
"a"

CL-USER 38 > a
#(NIL "a" NIL NIL NIL)
```

Textový řetězec je také jednorozměrné pole (tedy vektor), jehož prvky jsou znaky. K zápisu znaků se používá dvojice znaků `#\` následována buď příslušným znakem, nebo (pro větší čitelnost u speciálních znaků) jeho textovým popisem.

Příklad 8.26. Několik příkladů zápisu znaků: `#\a`, `#\A`, `#\2`, `#\Newline`, `#\Space`.

Vytvářet nové řetězce lze pomocí funkce `make-string`, jejíž zjednodušená syntax je tato:

```
(make-string size) => new-string
```

Funkce `make-string` vytváří řetězec délky `size` s nedefinovaným obsahem a vrací jej jako svůj výsledek.

Nové řetězce vznikají i staticky, pokud je uvedeme přímo v kódu ohraničené uvozovkami. Takto vytvořené řetězce ale není povoleno modifikovat.

Stejně jako u ostatních polí lze i k prvkům řetězců přistupovat pomocí funkce `aref`.

Délku (počet prvků) jednorozměrných polí lze podobně jako u seznamů zjišťovat funkcí `length`.

Nová jednorozměrná pole a textové řetězce, které je možno modifikovat, lze také vytvořit pomocí funkce `copy-seq`. Tato funkce akceptuje jako parametr libovolnou posloupnost (tj. jednorozměrné pole nebo seznam) a vrací její kopii. Nový vektor nebo řetězec s inicializovaným obsahem tedy můžeme vytvořit tak, že pomocí této funkce vytvoříme kopii vektoru nebo objektu, zadaného přímo do zdrojového kódu:

```
CL-USER 41 > (copy-seq "ahoj")
"ahoj"

CL-USER 42 > (copy-seq #(1 2 3))
#(1 2 3)
```

Oba takto vytvořené vektory je nyní povoleno modifikovat.

Nyní zpět k definici třídy `my-vector`. Třidu definujeme tak aby její instance obsahovaly ve slotu `representation` jednorozměrné pole.

```
(defclass my-vector (my-sequence)
  ((representation :initform "")))

(defmethod representation ((vector my-vector))
  (copy-seq (slot-value vector 'representation)))

(defmethod set-representation ((vector my-vector) value)
  (setf (slot-value vector 'representation)
        (copy-seq value))
  vector)
```

Ukažme nyní implementaci metod funkcí fyzické vrstvy pro tuto třídu. Jako ukazatel na pozici ve vektoru použijeme její index:

```
(defmethod after-end-p ((vec my-vector) position)
  (>= position (my-length vec)))

(defmethod before-start-p ((vec my-vector) position)
  (<= position 0))

(defmethod first-position ((vec my-vector))
```

```

0)

(defmethod last-position ((vec my-vector))
  (- (my-length vec) 1))

(defmethod next-position ((vec my-vector) position)
  (+ position 1))

(defmethod prev-position ((vec my-vector) position)
  (- position 1))

(defmethod position-item ((vec my-vector) position)
  (aref (slot-value vec 'representation)
        position))

(defmethod set-position-item ((vec my-vector) position value)
  (setf (aref (slot-value vec 'representation)
              position)
        value)
  vec)

```

Než tuto implementaci vyzkoušíme, provedeme ještě jednu úpravu. Naše obecná metoda funkce `my-length` počítá délku posloupnosti tak, že postupně projde všechny její prvky. U jednorozměrných polí je ale efektivnější způsob jak zjistit délku, například pomocí funkce `length`. Proto obecnou metodu přepíšeme metodou specializovanou na třídu `my-vector`:

```

(defmethod my-length ((vec my-vector))
  (length (representation vec)))

```

Nyní můžeme implementaci vyzkoušet:

```

CL-USER 12 > (setf v (make-instance 'my-vector))
#<MY-VECTOR 2009A507>

CL-USER 13 > (set-representation v "Jaro je tu.")
#<MY-VECTOR 2009A507>

CL-USER 14 > (representation v)
"Jaro je tu."

CL-USER 15 > (my-find v #\a)
#\a

CL-USER 16 > (my-find v #\i)
NIL

CL-USER 17 > (my-elt v 6)
#\e

CL-USER 18 > (progn
              (my-set-elt v 5 #\t)
              (my-set-elt v 6 #\u))

```

```

(my-set-elt v 8 #\j)
(my-set-elt v 9 #\e))
#<MY-VECTOR 2009A507>

CL-USER 19 > (representation v)
"Jaro tu je."

```

Příklad 8.27. Jako další optimalizaci je možno zařídit, aby metody `my-elt` a `my-set-elt` pro třídu `my-vector` neprocházely celou posloupnost ale využily přímého přístupu k prvkům pole. Navrhněte jak.

8.9. Seznamy

Jako druhý příklad potomka třídy `my-sequence` uvedeme třídu `my-list`, jejíž instance budou reprezentovat spojové seznamy. Při jejich implementaci budeme kopírovat způsob zavedený v Lispu. Seznamy tedy budou dvou typů: prázdné seznamy (instance třídy `my-empty-list`) a páry (instance třídy `my-cons`), které budou obsahovat jednak libovolný prvek (první prvek seznamu) a jednak jinou instanci třídy `my-list`. Základní definice tedy budou následující:

```

(defclass my-list (my-sequence)
  ())

(defclass my-empty-list (my-list)
  ())

(defclass my-cons (my-list)
  ((my-car :initform nil)
   (my-cdr :initform (make-instance 'my-empty-list))))

(defmethod my-car ((cons my-cons))
  (slot-value cons 'my-car))

(defmethod my-cdr ((cons my-cons))
  (slot-value cons 'my-cdr))

```

Poznámka 8.28. Instance třídy `my-cons` je tedy při vytvoření inicializována na jednoprvkový seznam obsahující `nil`.

Kvůli snazší práci s našimi seznamy si definujeme funkci a zprávu, které budou převádět obyčejné seznamy na naše a zpět:

```

(defun list-to-my-list (list)
  (if (null list)
      (make-instance 'my-empty-list)
      (let ((new (make-instance 'my-cons)))
        (setf (slot-value new 'my-car)
              (car list)
              (slot-value new 'my-cdr)
              (list-to-my-list (cdr list)))
          new)))

```

```
(defmethod my-list-to-list ((list my-list))
  '())

(defmethod my-list-to-list ((list my-cons))
  (cons (my-car list)
        (my-list-to-list (my-cdr list))))
```

Test:

```
CL-USER 3 > (list-to-my-list '())
#<MY-EMPTY-LIST 200B925B>

CL-USER 4 > (list-to-my-list '(1 2 3))
#<MY-CONS 201048B3>

CL-USER 5 > (my-list-to-list *)
(1 2 3)
```

K určení pozice v seznamu budeme používat podseznam, který začíná prvkem, jehož pozici chceme specifikovat. Tak umožníme efektivní přístup k prvkům seznamu na dané pozici a průchod seznamem zleva doprava (tedy určení následující pozice k dané pozici).

Poznámka 8.29. Méně efektivní bude určení předchozí pozice k dané pozici, ale to vyplývá z konstrukce spojových seznamů.

K určení pozice za koncem a před začátkem seznamu použijeme prázdný seznam (instanci třídy `my-empty-list`).

Následuje implementace metod zpráv fyzické úrovně.

```
(defmethod my-empty-list-p ((list my-list))
  nil)

(defmethod my-empty-list-p ((list my-empty-list))
  t)

(defmethod after-end-p ((list my-list) position)
  (my-empty-list-p position))

(defmethod before-start-p ((list my-list) position)
  (my-empty-list-p position))

(defmethod first-position ((list my-list))
  list)

(defmethod last-position ((list my-empty-list))
  list)

(defmethod last-position ((list my-cons))
  (if (my-empty-list-p (my-cdr list))
      list
      (last-position (my-cdr list))))
```



```

(defmethod next-position ((list my-list) position)
  (my-cdr position))

(defmethod prev-position ((list my-list) position)
  (cond ((my-empty-list-p position)
        (error "There is no previous position"))
        ((eql position (my-cdr list))
         list)
        (t (prev-position (my-cdr list) position))))

(defmethod position-item ((list my-list) position)
  (my-car position))

(defmethod set-position-item ((list my-list) position value)
  (setf (slot-value position 'my-car) value)
  list)

```

Nyní by měly instance třídy `my-list` správně reagovat na všechny zprávy logické úrovně. Vyzkoušíme to:

```

CL-USER 1 > (setf l (list-to-my-list '(1 2 3 4 5)))
#<MY-CONS 200D85C7>

CL-USER 2 > (my-length l)
5

CL-USER 3 > (my-find l 3)
3

CL-USER 4 > (my-find l 6)
NIL

CL-USER 5 > (my-elt l 4)
5

CL-USER 6 > (my-elt l 5)

Error: No applicable methods for #<STANDARD-GENERIC-FUNCTION
MY-CAR 216BE74A> with args (#<MY-EMPTY-LIST 200D8487>)

```

Příklad 8.30. Otestujte také vaši zprávu `my-find-if` z příkladu 8.22

Příklad 8.31. Všechny zprávy, které jsme zatím použili v testech, procházejí posloupnosti zleva doprava. Metody `last-position`, `prev-position` a `before-start-p` tedy zatím nejsou otestovány. Napravte to tím, že zkusíte otestovat zprávy `my-find-if-from-right` a `my-find-if-from` z příkladů 8.23 a 8.24.

Příklad 8.32. Průchod seznamu zprava doleva není efektivní. Navrhněte pro třídu `my-list` efektivnější implementaci metod `my-find-if-from-right` a `my-find-if-from`. Pokud z vašeho návrhu vyplyne potřeba definovat nové zprávy fyzické vrstvy, definujte jejich metody i pro třídu `my-vector`.

8.10. Posloupnosti: další příklady

Příklad 8.33. Dalším příkladem posloupnosti je obousměrný spojový seznam. Podle vzoru jednosměrného seznamu navrhnete implementaci třídy obousměrných seznamů a definujte metody fyzické vrstvy. Vše otestujte.

Příklad 8.34. V matematice se setkáváme s číselnými posloupnostmi, jejichž prvky jsou určeny buď vzorcem pro n -tý člen nebo rekurentním vzorcem. Navrhnete třídu takových posloupností a vyzkoušejte ji například pro aritmetickou posloupnost. Můžete uvažovat posloupnosti konečné délky. Jak byste navrhli definici třídy nekonečných posloupností?

Poslední změny

- 22. dubna 2008 Přidány příklady na posloupnosti (podkapitoly 8.5 až 8.10).
- 5. dubna 2008 Přidány příklady 8.7, 8.13, 8.14, 8.15, 8.16, 8.17, 8.18.

9. Události

V objektových systémech (například grafických) se často kromě stromové hierarchie tříd založené na dědičnosti (tj. vztahu *potomek–předek*) setkáváme ještě s hierarchií jinou, danou tzv. *agregací*, neboli vztahem vlastnickým (tj. vztahem *prvek–kontejner*). Tento vztah se může vytvářet až za běhu aplikace, bývá méně formalizován a je také obvykle volnější (ne všechny objekty mu musí podléhat) a dynamičtější (za běhu aplikace může prvek změnit svého vlastníka a podobně).

Tento vztah se také využívá k *předávání zodpovědnosti* za některé činnosti z podřízených objektů na jejich kontejnery. U grafických aplikací totiž často dochází k situacím, kdy objekt není sám schopen rozhodnout, jaká akce se má vyvolat následkem nějakého uživatelského zásahu (třeba kliknutí myši), a proto by bylo nešikovné mu tuto akci definovat jako metodu. Definice nových metod by také znamenala nutnost definice nových tříd (například pro každé tlačítko v rámci okna), která by vedla k nepříjemnému rozrůstání zdrojového textu programu.

V objektových systémech proto bývá zpracován mechanismus, kterým objekty informují své nadřízené o nutnosti vykonání nějaké akce. Předávání této informace se děje pomocí *událostí*.

V této kapitole předvedeme fungování událostí na dvou základních příkladech: automatizace překreslování okna po změně jeho obsahu a reakce na vstup z myši. Mechanismus událostí předvedeme nejprve na jednoduchých grafických objektech, pak jej rozšíříme i na objekty složené.

9.1. Zpětná volání v knihovně *micro-graphics*

Při experimentování s okny knihovny *micro-graphics* jsme zjistili, že v nich nakreslené obrázky nikdy dlouho nevydrží. Je to tím, že když operační systém potřebuje okno znovu vykreslit, například poté, co bylo překryto jinými okny, neví, co do něj má nakreslit. Tato situace se v grafických systémech řeší pomocí zpětných volání: když je potřeba okno znovu překreslit, systém zavolá uživatelskou funkci, která to udělá.

Nejinak tomu je v knihovně *micro-graphics*, která umožňuje v každém okně zaregistrovat několik typů uživatelských funkcí (zpětných volání, callbacků), které pak knihovna zavolá, aby program pracující s oknem upozornila, že nastala situace, na kterou by mohl chtít reagovat. Tyto funkce jsou podobně jako grafické parametry okna zaregistrovány pod názvy.

K zaregistrování zpětného volání nebo k jeho zrušení slouží funkce `mg:set-callback`, k zjištění aktuální hodnoty funkce `mg:get-callback`. Syntax funkce `mg:set-callback` je následující:

```
(set-callback mg-window callback-name function) => nil
```

`mg-window`: hodnota vrácená funkcí `mg:display-window`

`callback-name`: symbol

`function`: funkce nebo `nil`

Parametr *callback-name* určuje název zpětného volání, které chceme nastavit nebo zrušit. Pokud je chceme nastavit, uvedeme v parametru *function* funkci, která se má při zpětném volání zavolat, pokud je chceme zrušit, uvedeme v tomto parametru hodnotu `nil`.

Syntax funkce `mg:get-callback`:

```
(get-callback mg-window callback-name) => result
```

mg-window: hodnota vrácená funkcí `mg:display-window`

callback-name: symbol

result: funkce nebo `nil`

Parametr *callback-name* určuje název zpětného volání, které chceme získat.

Zatím uvedeme dva typy zpětných volání, která lze v okně knihovny `micro-graphics` zaregistrovat. Tyto typy jsou určeny symboly, které lze použít jako hodnotu parametru *callback-type* ve výše uvedených funkcích. Jsou to symboly `:display` a `mouse-down`:

- `:display`** Označuje funkci, která se zavolá, kdykoliv okno potřebuje překreslit.
- `:mouse-down`** Tato funkce bude zavolána kdykoli uživatel klikne do okna myší.

Funkci zaregistrovanou jako zpětné volání s názvem `:display` knihovna `micro-graphics` volá s jedním argumentem, a to odkazem na příslušné okno (tj. hodnotou vrácenou funkcí `mg:display-window`).

Funkci zaregistrovanou pod názvem `:mouse-down` knihovna volá se čtyřmi argumenty: odkazem na okno, symbolem označujícím tlačítko, kterým uživatel klikl (v úvahu připadají symboly `:left`, `:center`, nebo `:right`) a souřadnicemi *x*, *y* pixelu v okně, na který uživatel klikl.

9.2. Jednoduché využití zpětných volání

V prvním kroku si ukážeme, jak jednoduchým způsobem využít callbacku `:display` k automatickému překreslení okna a callbacku `:mouse-down`, v dalším pak naše řešení zdokonalíme.

Upravíme třídu `window` tak, aby nově vytvořená okna registrovala jako zpětné volání `:display` funkci, která zajistí překreslení okna kdykoliv o ně operační systém požádá.

Definice třídy bude podobná jako dříve:

```
(defclass window (mg-object)
  ((mg-window :initform (mg:display-window))
   shape
   (background :initform :white)))
```

Pomocí zprávy `redraw` napíšeme metodu `install-callbacks`, která nastaví příslušné zpětné volání:

```
(defmethod install-callbacks ((w window))
  (mg:set-callback (slot-value w 'mg-window)
                   :display (lambda (mgw)
                              (declare (ignore mgw))
                              (redraw w))))

w)
```

Poznámka 9.1. Řádek `(declare (ignore mgw))` nemá význam pro běh programu. Pouze potlačí upozornění překladače o nepoužité proměnné `mgw`.

V metodě `initialize-instance` třídy `window` tuto metodu využijeme:

```
(defmethod initialize-instance ((w window) &key)
  (call-next-method)
  (set-shape w (make-instance 'empty-shape))
  (install-callbacks w)
  w)
```

Všechny dříve napsané příklady by měly fungovat i s novou definicí třídy `window`. Navíc by se měla nově vytvořená okna sama automaticky překreslovat podle potřeby operačního systému.

Příklad 9.2. Podívejme se na důsledky naší úpravy. Vytvořme například okno s evropskou vlajkou (kód z příkladů ke kapitole o dědičnosti):

```
(setf flag (make-instance 'eu-flag))
(setf w (make-instance 'window))
(set-shape w flag)
(redraw w)
```

Po vyhodnocení druhého výrazu se na obrazovce objeví prázdné okno. Třetí výraz nemá žádný viditelný efekt — jeho vyhodnocení obnáší pouze nastavení slotu v objektu `w`. Proto bylo potřeba ještě vykreslení okna vyvolat čtvrtým výrazem.

Zobrazené okno se nyní bude překreslovat, kdykoliv o to operační systém požádá (při změně velikosti, po přesunutí okna do popředí a podobně).

Poznámka 9.3. Za jiných okolností se ovšem okno automaticky nepřekresluje. To platí například pro volání `(set-shape w flag)`, `(set-black-and-white-p flag t)`, nebo třeba `(move flag 10 10)`.

Rozšířme ještě definici třídy `window` o reakci na zpětné volání `:mouse-down`. Uděláme to jednoduše. Definujeme novou metodu `window-mouse-down` třídy `window`, jejímž úkolem bude reagovat na uživatelské kliknutí do okna. Ve třídě `window` tato metoda nebude dělat nic. Metodu `install-callbacks` třídy `window` rozšíříme o instalaci zpětného volání, které zašle zprávu `mouse-down` oknu.

```
(defmethod send-mouse-down ((w window) button x y)
  (window-mouse-down w
                      button
                      (move (make-instance 'point) x y)))

(defmethod install-callbacks ((w window))
  (mg:set-callback (slot-value w 'mg-window)
                  :display (lambda (mgw)
                             (declare (ignore mgw))
                             (redraw w))))

(mg:set-callback
 (slot-value w 'mg-window)
 :mouse-down (lambda (mgw button x y)
```

```

        (declare (ignore mgw))
        (send-mouse-down w button x y)))

w)

(defmethod window-mouse-down ((w window) button position)
  w)

```

Příklad 9.4. Využití metody `window-mouse-down` demonstrujeme na okně s vlajkou EU. Definujeme třídu `eu-flag-window` jako potomka třídy `window` a v její metodě `initialize-instance` nastavíme nové instance jako grafický objekt vlajku EU (instanci třídy `eu-flag`). Metodu `window-mouse-down` přepíšeme tak, aby se po kliknutí myší vlajka změnila z barevné na černobílou a naopak.

```

(defclass eu-flag-window (window) ())

(defmethod initialize-instance ((w eu-flag-window) &key)
  (call-next-method)
  (set-shape w (make-instance 'eu-flag))
  w)

(defmethod window-mouse-down ((w eu-flag-window) butt pos)
  (set-black-and-white-p
   (shape w)
   (not (black-and-white-p (shape w))))
  (redraw w))

```

9.3. Vlastnické vztahy, delegování, události

V našem systému je vlastnický vztah objektů naznačen u objektů grafických: instance třídy `window` vlastní (také lze říci *obsahuje*) ve slotu `shape` potomka třídy `shape`, který sám je obvykle potomkem třídy `picture`, a tedy vlastní (obsahuje) ve slotu `items` své prvky, které opět mohou být instancemi třídy `picture` nebo, obecněji, `compound-shape`. Tak se v každém okně vytváří stromová struktura objektů organizovaná podle principu vlastnění.

S vlastnickým vztahem objektů se do značné míry kryje (v některých systémech více, v jiných méně) mechanismus *předávání zodpovědnosti* (*delegování*). Některé činnosti objektu jsou z principu závislé ne na jeho vnitřním stavu, ale na stavu prostředí, ve kterém se objekt nachází. Proto bývá zaveden mechanismus, kterým objekt po obdržení některých zpráv informuje své okolí o vzniku situace, na niž může okolí reagovat.

Příklad 9.5. Pokud například chceme nastavit barvu nějakého grafického objektu, třeba kruhu, není nutné, aby tuto činnost za daný kruh prováděl někdo jiný; ve třídě `shape` máme definovanou metodu, která příslušnou datovou položku nastaví. Přesto může být vhodné, aby se o této změně ještě nějaký jiný objekt dozvěděl — například okno, které kruh obsahuje, aby mohlo nechat překreslit svůj obsah, nebo třeba textové pole, ve kterém má být barva kruhu napsána. Proto v následujícím odstavci zavedeme způsob, jakým kruh po změně své barvy, případně jakéhokoli jiného parametru (poloměru, polohy, tloušťky pera) zašle jinému objektu obecnou zprávu o změně svého obsahu. Cílový objekt pak může podle charakteru změny rozhodnout o další činnosti.

Tak dosáhneme toho, že okolí kruhu bude vždy informováno (pokud o to bude mít zájem) o každé jeho změně, aniž by bylo nutno tomu samotný kruh přizpůsobovat a vytvářet mu nové metody.

V mnoha případech lze ale volit jiné řešení. Kruh sám by například mohl zavolat metodu `redraw` okna, které by našel ve svém slotu `window`. Ve druhém případě (zapsání jména barvy do textového pole) si jistě také dovedeme představit jinou možnost: mohli bychom kruhu předdefinovat metodu `set-color` tak, aby textovému poli sám zprávu měnící jeho obsah po změně své barvy poslal. Tento způsob by ale znamenal nutnost definice nové třídy kruhu.

Mechanismus předávání zodpovědnosti nemusí být nutno komponovat přímo do objektového systému; uživatel si jej v jednoduché podobě může zavést sám. Na druhé straně, v objektových systémech a vývojových prostředích bývá obvykle nějaká jeho podpora kvůli rozšíření možností a standardizaci zavedena. Tak to postupně v dalších odstavcích uděláme i v našem objektovém systému.

Zavedeme pojem *delegáta objektu*. Bude to objekt, jemuž bude původní objekt zasílat různé zprávy o změně svého vnitřního stavu a podobně. Delegát objektu tedy bude zodpovědný za případné ošetření situací, které objekt sám neřeší. Každý objekt může a nemusí mít delegáta. Zprávám, které objekt svému delegátovi posílá, říkáme *události*. Abychom odlišili události od ostatních zpráv, zavedeme konvenci, že název každé události musí začínat předponou „ev-“.

Příklad 9.6. Například zpráva, kterou budou objekty informovat svého delegáta o změně svého stavu (třeba barvy), se bude standardně nazývat `ev-change`, zpráva informující o kliknutí myši `ev-mouse-down`.

Pojem delegáta objektu přirozeně koresponduje s pojmem vlastníka, uvedeném dříve. Proto bude delegát objektu většinou jeho vlastníkem. Například u instancí třídy `compound-shape` zařídíme, aby se všem jejich prvkům nastavil delegát na jejich vlastníka — tj. tuto instanci.

Delegáta objektu budeme ukládat do slotu `delegate`. Vzhledem k tomu, že by bylo nešikovné, kdyby každý delegát musel obsluhovat všechny zprávy, které mu objekt může posílat, budeme do slotu `events` objektu ukládat seznam všech událostí, kterým jako delegát rozumí. Jelikož budeme také chtít umožnit, aby různé objekty, které mají téhož delegáta, mohly posílat události téhož typu pod různými názvy (kvůli jejich rozlišení), budeme do seznamu `events` události zadávat včetně informace o případném překladu jejich názvu. K tomu všemu se ale podrobněji dostaneme v dalších odstavcích.

9.4. Implementace událostí u jednoduchých grafických objektů

Vzhledem k tomu, že novou funkčnost chceme přidat do instancí třídy `shape` i `window`, definujeme nejprve společného předka obou těchto tříd: třídu `mg-object`. Jak bylo popsáno, všechny instance této třídy budou obsahovat slot pro delegáta a seznam událostí:

```
(defclass mg-object ()
  ((delegate :initform nil)
   (events :initform '())))

(defmethod delegate ((obj mg-object))
  (slot-value obj 'delegate))

(defmethod set-delegate ((obj mg-object) delegate)
  (setf (slot-value obj 'delegate) delegate))

(defmethod events ((obj mg-object))
  (slot-value obj 'events))
```

(objekty nemusí mít delegáta povinně, proto je počáteční hodnota této položky nil; podobně počáteční hodnotou slotu `events` je prázdný seznam)

Do slotu `events` ukládáme seznam událostí obsluhovaných delegátem objektu. Každá událost je reprezentována svým jménem, jehož název začíná předponou “ev-”. Jelikož chceme, aby se událost dala posílat také pod jiným jménem než standardním, uvedeme ji vždy jako seznam tvaru (jméno překlad).

Příklad 9.7. Pokud bychom tedy například chtěli, aby událost `ev-change`, kterou objekty posílají po změně svého stavu, byla odeslána jako zpráva s názvem `ev-circle-1-change`, uvedeme v seznamu `events` prvek (`ev-change ev-circle-1-change`).

K nastavování seznamu událostí definujeme metodu `set-events`. Abychom uživatele nenutili zadávat událost dvouprvkovým seznamem se dvěma stejnými prvky, pokud ji nebude chtít překládat, použijeme pomocné funkce `canonicalize-event` a `canonicalize-events`:

```
(defun canonicalize-event (event)
  (if (typep event 'symbol)
      (list event event)
      event))

(defun canonicalize-events (events)
  (mapcar #'canonicalize-event events))

(defmethod set-events ((object mg-object) value)
  (setf (slot-value object 'events)
        (canonicalize-events value))
  object)
```

Kromě metody `set-events`, která objektu nastavuje celý seznam `events`, se budou hodit také metody `remove-event` a `add-event`, na odebrání a přidávání jednotlivých zpráv:

```
(defmethod remove-event ((obj mg-object) event)
  (setf (slot-value obj 'events)
        (remove-if (lambda (l)
                      (eql event (first l)))
                    (slot-value obj 'events))))

(defmethod add-event ((obj mg-object) event)
  (remove-event obj event)
  (setf (slot-value obj 'events)
        (cons (canonicalize-event event)
                (events obj))))
```

Metoda `send-event` slouží ke zjednodušení posílání událostí. Pokud budeme chtít, aby objekt zaslal svému delegátovi událost s názvem `event` a seznamem argumentů `event-args`, zavoláme jeho metodu `send-event` s těmito daty jako argumenty.

Metoda, kterou definujeme pro třídu `mg-object`, zjistí, zda má objekt delegáta a (s použitím pomocné zprávy `find-event`) zda se událost nachází v seznamu `events`. Pokud ano, najde v tomto seznamu její překlad a pod novým jménem ji pošle. Jako první

argument za jménem události použije objekt samotný. Každá událost tak bude mít jako první parametr po přijímajícím objektu objekt, který událost posílá.

U zprávy `send-event` se poprvé setkáváme s klíčovým slovem `&rest` v λ -seznamu funkce nebo metody. Toto klíčové slovo se používá v případě, že chceme definovat funkci nebo metodu s proměnným počtem parametrů. Symbol, který stojí za klíčovým slovem `&rest` slouží jako proměnná, do které se při volání funkce (metody) dosadí seznam všech zbývajících argumentů, které se nedostaly do předchozích parametrů.

Příklad 9.8. Definujme funkci `test`:

```
(defun test (a &rest b)
  (list a b))
```

A provedme několik pokusů:

```
CL-USER 4 > (test 1)
(1 NIL)
```

```
CL-USER 5 > (test 1 2)
(1 (2))
```

```
CL-USER 6 > (test 1 2 3 4 5)
(1 (2 3 4 5))
```

```
CL-USER 7 > (test)
```

```
Error: The call (#<Function TEST 216D6792>) does not match
definition (#<Function TEST 216D6792> A &REST B).
```

```
1 (continue) Return a value from the call to #<Function TEST
216D6792>.
```

```
2 Try calling #<Function TEST 216D6792> again.
```

```
3 Try calling another function instead of #<Function TEST
216D6792> with the same arguments.
```

```
4 Try calling #<Function TEST 216D6792> with a new argument
list.
```

```
5 (abort) Return to level 0.
```

```
6 Return to top loop level 0.
```

```
Type :b for backtrace, :c <option number> to proceed, or :?
for other options
```

Zpět k metodě `send-event`:

```
(defmethod find-event ((obj mg-object) event)
  (find-if (lambda (ev)
             (eql event (car ev)))
           (events obj)))

(defmethod send-event ((object mg-object) event
                      &rest event-args)
  (when (delegate object)
    (let ((ev (second (find-event object event))))
      (when ev
        (apply ev (delegate object) object event-args))))))
```

Nyní předvedeme implementaci tří událostí, které budou v našem objektovém systému standardní, události `ev-change`, `ev-changing` a události `ev-mouse-down`, pro případ jednoduchých grafických objektů. Je zřejmé, že standardních událostí by mohlo být více (například `ev-mouse-up`); také je možné, že při řešení konkrétních problémů si uživatel bude chtít implementovat své vlastní speciální události. Samozřejmě také může kdykoli změnit zde uvedené chování grafických objektů tím, že přepíše jejich metody.

Syntax těchto událostí bude následující:

```
ev-change receiver sender message &rest message-args
ev-changing receiver sender message &rest message-args
ev-mouse-down receiver sender origin button point
```

receiver je objekt přijímající zprávu (delegát), *sender* je vždy objekt, který událost svému delegátovi zasílá.

U událostí `ev-change` a `ev-changing` se v dalších argumentech zasílá informace o okolnostech, které ke změně objektu vedly — zpráva, která změnu vyvolala a argumenty, se kterými byla zaslána.

Dalšími argumenty události `ev-mouse-down` jsou objekt, na který se kliklo (argument *origin* — jak uvidíme u složených objektů, tento objekt se může od odesílatele zprávy lišit), a tlačítko myši s bodem, určujícím souřadnice místa kliknutí.

9.5. Reakce na změny u jednoduchých objektů

Události `ev-change` a `ev-changing` budeme posílat pomocí metod `change` a `changing`, které volají metodu `send-event` se správnými parametry:

```
(defmethod change ((object mg-object) message &rest args)
  (apply #'send-event object 'ev-change message args))

(defmethod changing ((object mg-object) message &rest args)
  (apply #'send-event object 'ev-changing message args))
```

Nyní je třeba upravit všechny metody, které mění vnitřní stav objektu tak, aby tyto dvě metody volaly. Například:

```
(defmethod set-color ((shape shape) value)
  (changing shape 'set-color value)
  (setf (slot-value shape 'color) value)
  (change shape 'set-color value)
  shape)

(defmethod set-thickness ((shape shape) value)
  (changing shape 'set-thickness value)
  (setf (slot-value shape 'thickness) value)
  (change shape 'set-thickness value)
  shape)

(defmethod set-x ((point point) value)
  (changing point 'set-x value)
```

```
(setf (slot-value point 'x) value)
(change shape 'set-x value)
shape)
```

Vidíme, že situace je zralá na úpravu, pomocí bychom se zbavili nutnosti psát tentýž kód na více míst programu. Definujeme zprávu `call-with-change`, po jejímž zaslání se zavolá zadaná funkce, ale před tímto voláním se zašle zpráva `changing` a po něm zpráva `change`:

```
(defmethod call-with-change
  ((object mg-object) function message &rest args)
  (apply #'changing object message args)
  (funcall function)
  (apply #'change object message args)
  object)
```

Předchozí tři definice nyní můžeme pomocí zprávy `call-with-change` přepsat. Ukažme si to na metodě `set-color`:

```
(defmethod set-color ((shape shape) value)
  (call-with-change shape
    (lambda ()
      (setf (slot-value shape 'color) value))
    'set-color
    value))
```

Nová definice metody `set-color` sice je jednodušší, než definice předchozí, ale pomocí makra ji lze ještě zjednodušit. Definujeme makro `with-change`, které budeme v podobných situacích používat. Makro definujeme tak, aby bylo mezi volání funkcí `changing` a `change` možno umístit libovolný kód (parametr `body`).

```
(defmacro with-change ((object message &rest msg-args)
  &body body)
  `(call-with-change ,object
    (lambda () ,@body)
    ,message
    ,@msg-args))
```

Pomocí makra `with-change` nyní můžeme uvedené tři definice přepsat takto:

```
(defmethod set-color ((shape shape) value)
  (with-change (shape 'set-color value)
    (setf (slot-value shape 'color) value)))

(defmethod set-thickness ((shape shape) value)
  (with-change (shape 'set-thickness value)
    (setf (slot-value shape 'thickness) value)))

(defmethod set-x ((point point) value)
  (with-change (point 'set-x value)
    (setf (slot-value point 'x) value)))
```

Pro porozumění dalšímu textu není v žádném případě nutné, aby čtenář definici makra `with-change`, kterou zde uvádíme jen pro úplnost, pochopil. Totéž platí i o makru `without-changes`, které ukážeme později.

Stejným způsobem upravíme i metody `set-filledp`, `set-y`, `set-r`, `set-phi`, `set-r-phi`, `set-radius`, `set-items`, `set-closedp`. Metody `move`, `scale` a `rotate` zatím upravovat nebudeme, podíváme se na ně později.

Příklad 9.9. Úpravu nyní můžeme vyzkoušet: vytvoříme pomocnou třídu `test-delegate`, jejíž instance budou přijímat zprávy o změně od jiného objektu a tisknout o nich informaci:

```
(defclass test-delegate (mg-object) ())

(defmethod ev-change ((d test-delegate) sender message
                     &rest args)
  (format t
          "~%~%Změna:~%Objekt ~s~%Metoda: ~s~%Argumenty: ~s"
          sender message args))
```

Dále vytvoříme instanci této třídy:

```
(setf delegate (make-instance 'test-delegate))
```

A nakonec vytvoříme bod, nastavíme mu jako delegáta objekt `delegate` a zařídíme, aby mu posílal událost `ev-change`:

```
(setf pt (make-instance 'point))
(setf (delegate pt) delegate)
(setf (events pt) '(ev-change))
```

Nyní by objekt `delegate` měl při každé změně objektu `pt` vytisknout zprávu. Například po zavolání `(set-x pt 5)` takovou:

```
Změna:
Objekt #<POINT 2166EBBF>
Metoda: SET-X
Argumenty: (5)
```

Po zaslání zprávy, která sama vede k zaslání dalších zpráv, se ale může změna signalizovat vícekrát. Například po zavolání `(set-r pt 2)` se vytiskne

```
Změna:
Objekt #<POINT 2166EBBF>
Metoda: SET-R-PHI
Argumenty: (2 0.0)

Změna:
Objekt #<POINT 2166EBBF>
Metoda: SET-R
Argumenty: (2)
```

Je to tím, že metoda `set-r` volá metodu `set-r-phi`, která rovněž vnitřní stav bodu mění a tuto změnu signalizuje.

Správné řešení tohoto problému není snadné: pokud zrušíme zasílání události `ev-change` po obsloužení zprávy `set-r`, ztratí se informace o tom, že změna byla vyvolána metodou `set-r`. Pokud v metodě `set-r` nebudeme volat metodu `set-r-phi`, prohřešíme se pravidlu zapouzdření a vymstí se nám to jindy. (Tento příklad by byl ještě markantnější u složitějších objektů, jako například obrázků.) Vyhovující řešení tohoto problému navrhneme až později.

Jak upravit metody `move`, `scale` a `rotate`? Snadno nahlédneme, že problém těchto metod je v tom, že jsou v potomcích třídy `shape` přepisovány.

Například metoda `move` je ve třídě `shape` zatím definována tak, že nic nedělá:

```
(defmethod move ((shape shape) dx dy)
  shape)
```

a ve třídě `point` přepsána tak, aby vykonala konkrétní akci vhodnou pro body:

```
(defmethod move ((pt point) dx dy)
  (set-x pt (+ (x pt) dx))
  (set-y pt (+ (y pt) dy))
  pt)
```

Nyní by se nám hodilo definovat tyto metody tak, aby ve třídě `shape` pomocí makra `with-change` signalizovaly změnu grafického objektu a v jeho těle provedly akci specifickou příslušné podtřídy.

Poznámka 9.10. Člověka by napadlo, že by se v této situaci hodilo místo funkce `call-next-method` použít nějakou jinou, která by nevolala metodu implementovanou u předka, ale naopak u potomka třídy `shape`:

```
(defmethod move ((shape shape) dx dy)
  (with-change (shape 'move dx dy)
    (call-previous-method)))

(defmethod move ((pt point) dx dy)
  (set-x pt (+ (x pt) dx))
  (set-y pt (+ (y pt) dy))
  pt)
```

Jak již bylo řečeno, tento způsob volání metod v opačném pořadí (od nejobecnější třídy k potomkům) je použit například v programovacím jazyce Beta, ale v obecně rozšířených objektových jazycích se nepoužívá. V Common Lispu lze do programu vlastní způsob volání metod přidat definicí tzv. *kombinace metod*. Druhou možností řešení tohoto problému by bylo použít tzv. `:around` metody. Populární objektové jazyky žádnou z těchto dvou možností neposkytují, proto je zmiňujeme pouze jako zajímavost a nebudeme se jimi dále zabývat.

Rozumným řešením tohoto problému, které lze použít v běžných objektových jazycích, je definovat pomocné metody nového názvu, které budou metodami původními ve třídě `shape` volány:

```

(defmethod do-move ((shape shape) dx dy)
  shape)

(defmethod move ((shape shape) dx dy)
  (with-change (shape 'move dx dy)
    (do-move shape dx dy)))

(defmethod do-move ((pt point) dx dy)
  (set-x pt (+ (x pt) dx))
  (set-y pt (+ (y pt) dy))
  pt)

```

Toto řešení použijeme pro všechny tři uvedené metody, takže kromě metody `do-move` definujeme pomocné metody `do-scale` a `do-rotate`.

Poznámka 9.11. Je zřejmé, že metody `move` a `do-move` hrají rozdílnou úlohu. Zatímco u metody `move` očekáváme, že ji uživatel bude volat, ale asi ji nebude v potomcích třídy `shape` přepisovat, u metody `do-move` to bude přesně obráceně: uživatel ji nebude nikdy volat (na to je tady metoda `move`), ale v případě, že bude definovat vlastního potomka třídy `shape`, bude ji možná potřebovat přepsat. Proto by se v běžném objektovém jazyce použila pro metodu `move` ochrana typu *public* a pro metodu `do-move` typu *protected*.

Nyní dotáhneme tento příklad do konce a ukážeme, jak zařídit, aby se změny nakreslených objektů projevily v okně. Grafický objekt okna, umístěný ve slotu `shape`, bude mít za delegáta své okno. Jemu bude zasílat události `ev-changing` a `ev-change`, kdykoli se změní jeho vnitřní stav. Okno se v reakci na tuto událost překreslí. Nastavení provedeme v metodě `set-shape` třídy `window`, kde také zařídíme, aby se okno po zavolání této metody rovněž překreslilo.

Překreslení okna vyvoláme funkcí `mg:invalidate` knihovny `micro-graphics`. Tato funkce zaznamená, že okno potřebuje překreslit, ale samotné překreslení neprovede. Jedná se o obvyklý postup, který grafické knihovny používají, aby se vyhnuly zbytečnému několikanásobnému překreslování okna po každé jeho změně. Funkci `mg:invalidate` lze bez obav volat několikrát za sebou, překreslení okna vyvolá knihovna sama až poté, co naše práce s oknem skončí, a zajistí, aby se provedlo pouze jednou.

Poznámka 9.12. Víme už z minulé části, jakým způsobem se okno překresluje: knihovna `micro-graphics` zavolá funkci, kterou jsme uložili v callbacku `:display-callback`. Vidíme, jak snadno a rychle se situace i v tak jednoduchém grafickém systému, jako je ten náš, komplikuje. S tím se dá dělat pouze to, o co se snažíme i my: vytvořit podmínky pro uživatele grafického systému co nejjednodušší a od vnitřních implementačních komplikací ho pokud možno izolovat.

```

(defmethod invalidate ((w window))
  (mg:invalidate (mg-window w))
  w)

(defmethod set-shape ((w window) shape)
  (with-change (w 'set-shape shape)
    (setf (slot-value w 'shape) shape)
    (set-window shape w)
    (set-delegate shape w)
    (set-events shape '(ev-change))
    (invalidate w)))

```

```
(defmethod ev-change ((w window) sender message &rest args)
  (invalidate w))
```

Všimněme si, že v metodě `set-shape` používáme makro `with-change`, abychom signalizovali změnu okna případnému jeho delegátovi.

Podobně upravíme i metodu `set-background`:

```
(defmethod set-background ((w window) color)
  (with-change (w 'set-background color)
    (setf (slot-value w 'background) color)
    (invalidate w)))
```

Příklad 9.13. Nyní můžeme vyzkoušet, jak každá změna jednoduchého objektu umístěného v okně vede k překreslení. Vytvoříme si nové okno a umístíme do něj kruh:

```
(setf w (make-instance 'window))
(setf circle (make-instance 'circle))
(set-shape w circle)
```

Vidíme, že po vyhodnocení posledního výrazu se kruh v okně okamžitě objeví. Při postupném vyhodnocování následujících výrazů se také každá změna hned v okně projeví:

```
(set-radius circle 40)
(set-filledp circle t)
(set-color circle :red)
(move circle 100 100)
(set-background w :blue)
```

Pokud ale zavoláme nějakou metodu středu kruhu, například takto:

```
(move (center circle) 0 -50)
```

žádná změna v okně nenastane. Je to tím, že střed události zprávu o své změně zatím nikomu neposílá.

9.6. Klikání myši u jednoduchých objektů

Nyní se věnujme události `ev-mouse-down`. Při klikání do okna je především třeba umět zjistit, do které části okna uživatel klikl. K tomu se hodí zpráva `contains-point-p`, která slouží k tzv *hit-testingu*, neboli testování, zda daný bod leží uvnitř grafického objektu, který je příjemcem zprávy. Implementace metod této zprávy je poměrně jednoduchá u většiny grafických objektů. Výjimku tvoří třída `polygon`, u níž je naopak implementace poměrně složitá. Ve zdrojovém kódu k této kapitole lze nalézt úplné implementace všech metod této zprávy.

Z předchozího už víme, že po kliknutí myši do okna (instance třídy `window`) se zavolá jeho metoda `windows-mouse-down`. Tuto metodu jsme využili na začátku této kapitoly při naivní implementaci okna `eu-flag-window`. Nyní tuto metodu implementujeme tak, aby zjistila, zda místo kliknutí je uvnitř grafického objektu kresleného do okna, a pokud ano, tak aby tomuto objektu poslala zprávu `mouse-down`:


```
(defmethod window-mouse-down ((w window) button position)
  (when (contains-point-p (shape w) position)
    (mouse-down (shape w) button position)))
```

Metodu `mouse-down` u grafických objektů zatím definujeme pouze jednoduše pro třídu `shape` tak, že pouze vyvolá událost `ev-mouse-down`:

```
(defmethod mouse-down ((shape shape) button position)
  (send-event shape 'ev-mouse-down shape button position))
```

Okno tak dostává dvě příležitosti k reakci na kliknutí. Obvyklý způsob (který ovšem může být v každém vývojovém prostředí jiný) je ten, že uživatel první možnosti reakce nevyužívá, metodu `mouse-down` u okna nepřepisuje a o kliknutí se zajímá buď v okamžiku, kdy je grafickému objektu zavolána metoda `mouse-down`, nebo když odešle oknu událost `ev-mouse-down`. První možnost znamená přepsání metody `mouse-down` grafického objektu, druhá obsluhu zprávy `ev-mouse-down` oknem.

Předvedme si obě tyto možnosti na příkladu kruhu, který má náhodně změnit barvu, když se na něj klikne.

Příklad 9.14. Začneme druhou možností: Pokud bychom chtěli barvu měnit v obsluze události `ev-mouse-down` v okně, vypadalo by řešení například následovně. Definice třídy:

```
(defun random-color ()
  (let ((colors (color:get-all-color-names)))
    (nth (random (length colors)) colors)))

(defclass my-window (window) ())

(defmethod ev-mouse-down ((w my-window) sender
                          origin button position)
  (set-color sender (random-color)))
```

a test:

```
(setf w (make-instance 'my-window))
(setf circle (make-instance 'circle))
(move circle 100 100)
(set-radius circle 40)
(set-shape w circle)
(add-event circle 'ev-mouse-down)
```

Příklad 9.15. První možnost znamená definovat potomka třídy `circle` s tou vlastností, že v metodě `mouse-down` vždy změní svou barvu. Překreslení okna se bude dít pomocí události `ev-change` mechanismem, který už je naprogramován:

```
(defclass click-circle (circle) ())

(defmethod mouse-down ((circ click-circle) button position)
  (set-color circ (random-color))
  (call-next-method))
```


Test:

```
(setf w (make-instance 'window))
(set-shape w (make-instance 'click-circle))
```

V metodě `mouse-down` třídy `click-circle` voláme zděděnou metodu, která zasílá delegátovi kruhu o kliknutí zprávu, protože chceme, aby mělo okno stále možnost na kliknutí reagovat. Toto řešení je výhodnější v tom, že instance třídy `click-circle` můžeme umísťovat na různá místa a stále si zachovají schopnost měnit po kliknutí barvu.

Předchozí řešení může být výhodnější zase v jiných situacích, například kdybychom měli v okně více objektů různých typů (kruhy, polygony atd.) a po všech vyžadovali totéž chování.

9.7. Reakce na změny u složených objektů

Nyní budeme postupovat tak, jak jsme již uvedli. Zařídíme, aby složené objekty byly delegáty svých prvků. U kruhů je přirozené toto nastavení provést v okamžiku jejich vzniku:

```
(defmethod initialize-instance ((c circle) &rest args)
  (call-next-method)
  (let ((center (center c)))
    (set-events center '(ev-changing ev-change))
    (set-delegate center c))
  c)
```

U instancí třídy `compound-shape` v momentě, kdy jsou jim jejich prvky přiřazovány — v metodě `do-set-items`:

```
(defmethod do-set-items ((shape compound-shape) value)
  (setf (slot-value shape 'items) (copy-list value))
  (send-to-items shape #'set-window (window shape))
  (send-to-items shape #'set-delegate shape)
  (send-to-items
   shape #'set-events '(ev-changing ev-change)))
```

Nyní ještě zařídíme, aby složené objekty události `ev-changing` a `ev-change` správně přeposílaly svým delegátům:

```
(defmethod ev-changing ((c circle) sender message
                        &rest message-args)
  (changing c 'ev-changing sender message message-args))

(defmethod ev-change ((c circle) sender message
                     &rest message-args)
  (change c 'ev-change sender message message-args))

(defmethod ev-changing ((cs compound-shape) sender message
                      &rest message-args)
```

```

(changing cs 'ev-changing sender message message-args))

(defmethod ev-change ((cs compound-shape) sender message
                      &rest message-args)
  (change cs 'ev-change sender message message-args))

```

Tím jsme nastavili jako základní přirozenou hierarchii zodpovědnosti objektů, o které jsme již hovořili. Ve speciálních případech je ji samozřejmě možno změnit.

Příklad 9.16. Vraťme se teď k problému, o kterém jsme již hovořili, a zkusme provést malý experiment. Nejprve definujeme pomocnou třídu `test-delegate` a metody `ev-change` a `ev-changing`:

```

(defclass test-delegate (mg-object)
  ())

(defmethod ev-change ((d test-delegate) sender message
                     &rest args)
  (format
   t
   "~%~%Proběhla změna:~%Objekt ~s~%Metoda: ~s~%Argumenty: ~s"
   sender message args))

(defmethod ev-changing ((d test-delegate) sender
                       message &rest args)
  (format
   t
   "~%~%Proběhne změna:~%Objekt ~s~%Metoda: ~s~%Argumenty: ~s"
   sender message args))

```

pak zkusíme následující:

```

CL-USER 19 > (setf d (make-instance 'test-delegate))
#<TEST-DELEGATE 2187EBFB>

CL-USER 20 > (setf c (make-instance 'circle))
#<CIRCLE 200D84DF>

CL-USER 21 > (set-delegate c d)
#<TEST-DELEGATE 2187EBFB>

CL-USER 22 > (set-events c '(ev-change ev-changing))
((EV-CHANGE EV-CHANGE) (EV-CHANGING EV-CHANGING))

CL-USER 23 > (move c 10 10)

Proběhne změna:
Objekt #<CIRCLE 200D84DF>
Metoda: MOVE
Argumenty: (10 10)

Proběhne změna:

```

```
Objekt #<CIRCLE 200D84DF>
Metoda: EV-CHANGING
Argumenty: (#<POINT 200D84AB> MOVE (10 10))
```

```
Proběhne změna:
Objekt #<CIRCLE 200D84DF>
Metoda: EV-CHANGING
Argumenty: (#<POINT 200D84AB> SET-X (10))
```

```
Proběhla změna:
Objekt #<CIRCLE 200D84DF>
Metoda: EV-CHANGE
Argumenty: (#<POINT 200D84AB> SET-X (10))
```

```
Proběhne změna:
Objekt #<CIRCLE 200D84DF>
Metoda: EV-CHANGING
Argumenty: (#<POINT 200D84AB> SET-Y (10))
```

```
Proběhla změna:
Objekt #<CIRCLE 200D84DF>
Metoda: EV-CHANGE
Argumenty: (#<POINT 200D84AB> SET-Y (10))
```

```
Proběhla změna:
Objekt #<CIRCLE 200D84DF>
Metoda: EV-CHANGE
Argumenty: (#<POINT 200D84AB> MOVE (10 10))
```

```
Proběhla změna:
Objekt #<CIRCLE 200D84DF>
Metoda: MOVE
Argumenty: (10 10)
#<CIRCLE 200D84DF>
```

Jedna zaslaná zpráva může vyvolat několik událostí `ev-changing` a `ev-change`, a to i u jednoduchých objektů (dříve jsme uváděli příklad s metodou `set-r`). Tento problém vyřešíme tím, že definujeme metodu `call-without-changes`, která vytvoří dynamické prostředí, v němž události `ev-changing` a `ev-change` nebudou zasílány. K tomu zavedeme slot `change-level` ve třídě `mg-object`:

```
(defclass mg-object ()
  ((delegate :initform nil)
   (events :initform '())
   (change-level :initform 0)))
```

Tento slot bude indikovat, zda má objekt povoleno události `ev-change` a `ev-changing` zasílat. Pokud bude jeho hodnota nulová, zasílání bude povoleno, pokud bude kladná, bude blokováno.

Funkce `call-without-changes` bude na začátku hodnotu inkrementovat a na konci dekrementovat, takže v kódu mezi nimi bude zasílání těchto událostí blokováno:

```
(defmethod call-without-changes ((object mg-object) function)
  (setf (slot-value object 'change-level)
        (+ (slot-value object 'change-level) 1))
  (unwind-protect
    (funcall function)
    (setf (slot-value object 'change-level)
          (- (slot-value object 'change-level) 1))))
object)
```

V metodě `call-without-changes` jsme použili speciální operátor `unwind-protect`, se kterým jsme se zatím nesetkali. Tento speciální operátor zajistí, že se dekrementace správně provede i pokud dojde v průběhu volání funkce `function` k chybě. Objekt tak zůstane v konzistentním stavu, i když v průběhu práce s ním došlo k chybě.

Poznámka 9.17. Uvědomme si, že pokud nebude inkrementace hodnoty slotu `change-level` vždy (tedy i v případě, že dojde k chybě) vyvážena její dekrementací, nebude již možné s objektem pracovat. Pokud by se něco takového stalo v aplikaci, kterou bude používat uživatel, je možné, že už s ní nebude moci pracovat a bude ji muset ukončit.

Příklad 9.18. Funkci speciálního operátoru `unwind-protect` ozřejmíme na příkladě:

```
CL-USER 7 > (progn (error "Testovací chyba")
                  (print "Testovací tisk 1")
                  (print "Testovací tisk 2"))

Error: Testovací chyba
1 (abort) Return to level 0.
2 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed, or :?
for other options

CL-USER 8 : 1 > :a

CL-USER 9 >
```

Vidíme, že poté co došlo k chybě, vyvolané výrazem `(error "Testovací chyba")`, a poté, co jsme přerušili výpočet (napsáním `:a`), se následující výrazy již nevyhodnotily a nic se nevytisklo. Srovnajme to s dalším testem:

```
CL-USER 9 > (unwind-protect (progn (error "Testovací chyba")
                                   (print "Testovací tisk 1"))
                  (print "Testovací tisk 2"))

Error: Testovací chyba
1 (abort) Return to level 0.
2 Return to top loop level 0.

Type :b for backtrace, :c <option number> to proceed, or :?
for other options

CL-USER 10 : 1 > :a

"Testovací tisk 2"

CL-USER 11 >
```

Zde se neprovedl první tisk, protože před ním došlo k chybě. Vykonání druhého tisku i v případě chyby je ale zaručeno operátorem `unwind-protect`. Proto se poté, co byl výkon programu v důsledku chyby přerušen, druhý tisk provedl.

Podobně jako u makra `with-change` ještě definujeme makro, které použití funkce `call-without-changes` usnadní:

```
(defmacro without-changes (object &body body)
  `(call-without-changes ,object (lambda () ,@body)))
```

Typické použití makra `without-changes` je následující:

```
(without-changes object
  form-1
  form-2
  ...
  form-n)
```

V průběhu vyhodnocení výrazů `form-1` až `form-n` nebude objekt `object` žádné zprávy o změně zasílat. To je ale ještě třeba zařídit úpravou metod `changing` a `change`, které se musí na hodnotu slotu `change-level` podívat a podle toho se zachovat:

```
(defmethod changing ((object mg-object) msg &rest args)
  (unless (> (slot-value object 'change-level) 0)
    (apply #'send-event object 'ev-changing msg args)))

(defmethod change ((object mg-object) msg &rest args)
  (unless (> (slot-value object 'change-level) 0)
    (apply #'send-event object 'ev-change msg args)))
```

Jako poslední je třeba změnit metodu `call-with-change` tak, aby ignorovala všechny zprávy o změnách zasílané v průběhu činnosti kódu v jejím těle. Použijeme na správném místě makro `without-changes`:

```
(defmethod call-with-change
  ((object mg-object) function message &rest args)
  (apply #'changing object message args)
  (without-changes object
    (funcall function))
  (apply #'change object message args)
  object)
```

Příklad 9.19. Nyní můžeme vyzkoušet průběh signalizace změn například zopakováním experimentu. Definujeme třídu `test-delegate` jako dříve a pak vyhodnotíme následující výrazy (pokud jsme od předchozího pokusu neměnili obsah proměnné `c`, stačí vyhodnotit poslední řádek):

```
CL-USER 24 > (setf d (make-instance 'test-delegate))
#<TEST-DELEGATE 20092FE7>

CL-USER 25 > (setf c (make-instance 'circle))
```

```
#<CIRCLE 200EEEAB>

CL-USER 26 > (set-delegate c d)
#<TEST-DELEGATE 20092FE7>

CL-USER 27 > (set-events c '(ev-change ev-changing))
((EV-CHANGE EV-CHANGE) (EV-CHANGING EV-CHANGING))

CL-USER 28 > (move c 10 10)

Proběhne změna:
Objekt #<CIRCLE 200EEEAB>
Metoda: MOVE
Argumenty: (10 10)

Proběhla změna:
Objekt #<CIRCLE 200EEEAB>
Metoda: MOVE
Argumenty: (10 10)
#<CIRCLE 200EEEAB>
```

9.8. Klikání myší u složených objektů

Nyní se pojďme podívat, jak v grafických objektech definujeme metodu `mouse-down`. Princip naší implementace, kterou může uživatel kdykoli změnit přepsáním příslušných metod, bude následující: instance třídy `picture` (tedy obrázky) se v metodě `mouse-down` podívají, na který jejich prvek se kliklo, a zavolají jeho metodu `mouse-down`. Tak se zpráva o kliknutí dostane hierarchií kontejnerů a jejich prvků až k jednoduchému grafickému útvaru, na který se kliklo.

Ten zase naopak odešle svému delegátovi, kterým je v základní implementaci nadřazený obrázek, událost `ev-mouse-down`. Její obsluhu v instancích třídy `picture` (tj. v obrázcích) implementujeme tak, že neudělá nic, jen událost přepoše delegátovi o úroveň výše. Tak se zpráva o kliknutí dostane formou události postupně zase až na nejvyšší úroveň — grafický objekt okna.

Seznam parametrů události `ev-mouse-down` bude (`delegate sender origin button pt`), kde `delegate` bude příslušný delegát, pro nějž se metoda implementuje, `sender` bude objekt, který událost posílá (víme, že tento parametr se nastavuje u každé události v základní implementaci zprávy `send-event`), a `origin` objekt na nejnižší úrovni, který zprávu `mouse-down` dostal.

Uživatel může do tohoto řetězce volání kdekoli vstoupit tím, že buď přepíše metodu `mouse-down`, čímž změní přeposílání zpráv směrem shora dolů, nebo přepíše metodu `ev-mouse-down` a změní přeposílání událostí zdola nahoru. Také může změnit základní hierarchii delegátů definovanou ve třídě `compound-shape` a učinit jiný objekt než složený grafický objekt delegátem jeho prvků.

Poznámka 9.20. Obvykle není vhodné řetězec volání událostí `ev-mouse-down` směrem zdola nahoru přerušovat. Podřízený objekt nikdy nemůže vědět, zda se jeho delegát nepotřebuje o kliknutí myší dozvědět. Proto metody `ev-mouse-down` instancí třídy `picture` vždy na závěr funkcí `call-next-method` volají metodu definovanou níže, která událost `ev-mouse-down` přeposílá. Naproti tomu volání metody `mouse-down` směrem dolů se objekt může rozhodnout zastavit, pokud ji chce obsloužit sám a nepřeposílat ji podřízeným objektům (v takovém případě ovšem také sám pošle událost `ev-mouse-down` směrem nahoru).

Metoda `mouse-down` u třídy `picture` je definována následovně:

```
(defmethod mouse-down ((p picture) button position)
  (let ((item (find-if (lambda (it)
                        (contains-point-p it position))
                        (items p))))
    (when item
      (mouse-down item button position))))
```

Aby obrázky událost `ev-mouse-down` od podřízených objektů správně přijímaly, musí definovat její obsluhu. Ta jak víme má pouze přeposílat událost delegátovi:

```
(defmethod ev-mouse-down
  ((p picture) sender origin button position)
  (send-event p 'ev-mouse-down origin button position))
```

Kromě toho je třeba nastavit všem podřízeným objektům obrázku slot `events` tak, aby věděly, že událost mohou posílat:

```
(defmethod do-set-items ((p picture) items)
  (call-next-method)
  (when (propagate-color-p p)
    (send-to-items-set-color p (color p)))
  (send-to-items p #'add-event 'ev-mouse-down)
  items)
```

Poznámka 9.21. Tato implementace volání metody `mouse-down` shora dolů má vzhledem k hloubce vnoření grafických objektů kubickou složitost. Algoritmus by se u složitějších obrázků rychle zpomalil. Na optimalizaci je ale zatím času dost.

9.9. Příklady

Kód k této kapitole obsahuje několik jednodušších příkladů použití zde implementované knihovny. Doporučujeme si tyto příklady podrobně projít. Prvním krokem k pochopení obsahu kapitoly je naučit se zde implementovanou knihovnu používat.

A. Slovníček

Většina prvků jazyka Scheme má přesný nebo velmi podobný protipól v Common Lispu. Tento slovníček uvádí překlad všech symbolů ze standardu R⁵RS do Common Lispu jako první pomoc uživatelům jazyka Scheme při programování v Common Lispu (čtenáři–studenti se nemusejí obávat, že musí všechny uvedené výrazy ovládat; slovníček je opravdu míněn pouze jako pomoc).

Při používání slovníčku je třeba být seznámen se základními rozdíly mezi Common Lispem a Scheme uvedenými v kapitole 2. Pro další práci je samozřejmě nutné pokračovat ve studiu Common Lispu, například pomocí dalších částí tohoto textu. Je-li vysvětlení ve slovníčku příliš stručné, je třeba sáhnout po definici ve standardu.

A.1. Základní výrazy

Scheme	Common Lisp	Poznámka pro CL
(define a x)	(defvar a x) (defparameter a x)	Pouze je-li define na nejvyšší úrovni. Lokální define v CL obdobu nemá (je třeba použít jiný nástroj, např. let).
(quote x), 'x (lambda (x1...xn) ...) (lambda (x1...xn . y) ...) (lambda y ...) (define f (lambda (x1...xn) ...)) (define f (lambda (x1...xn . y) ...)) (define f (lambda y ...)) if	(quote x), 'x (lambda (x1...xn) ...) (lambda (x1...xn &rest y) ...) (lambda (&rest y) ...) (defun f (x1...xn) ...) (defun f (x1...xn &rest y) ...) (defun f (&rest y) ...) if	Nastavuje ale funkční vazbu. Nastavuje ale funkční vazbu. Nastavuje ale funkční vazbu. Je-li hodnota a v (if a b) nil, vrací nil (ve Scheme nedefinováno). (setf a x) vrací hodnotu x.
(set! a x)	(setf a x)	

A.2. Odvozené výrazy

Scheme	Common Lisp	Poznámka pro CL
cond	cond	Základy stejné, místo else psát t, další drobné rozdíly. Drobné rozdíly.
case and, or let, let*	case and, or let, let*	Inicializace nemusí být uvedena, pak navazuje na nil, např. (let (a (b) (c nil)) ...) navazuje a, b i c na nil. Tzv. pojmenované let neexistuje. Varianta neexistuje, viz ale labels.
letrec begin do delay quasiquote, unquote, unquote-splicing `(x ,y ,@z)	progn do `(x ,y ,@z)	Neexistuje. Neexistují. Nutno používat zkratky “~”, “,” a “,”@”.

A.3. Makra

Scheme
 let-syntax, letrec-syntax,
 syntax-rules

Common Lisp

Poznámka pro CL
 V CL nejsou hygienická makra.
 Obvyčejná makra pomocí defmacro

A.4. Standardní procedury — predikáty ekvivalence

Scheme
 eqv?
 eq?
 equal?

Common Lisp
 eql
 eq
 equal, equalp

Poznámka pro CL
 Přibližně
 Přibližně
 Přibližně

A.5. Standardní procedury — čísla

Scheme
 (number? x)

 (complex? x)

 (real? x)
 (rational? x)

 (integer? x)

exact?, inexact?
 =, <, >, <=, >=
 zero?, positive?, nega-
 tive?, odd?, even?
 max, min
 +, -, *, /

abs
 quotient
 remainder, modulo
 gcd, lcm
 numerator, denominator
 floor, truncate, ceiling,
 round

rationalize

exp, log, sin, cos, tan,
 asin, acos, atan
 sqrt
 expt
 make-rectangular
 make-polar
 real-part, imag-part
 magnitude, angle
 exact->inexact,
 inexact->exact
 number->string

string->number

Common Lisp
 (numberp x),
 (typep x 'number)
 (or (complexp x)
 (rationalp x))
 (typep x
 '(or complex rational))
 (realp x), (typep x 'real)
 (rationalp x),
 (typep x 'rational)
 (integerp x),
 (typep x 'integer)

=, <, >, <=, >=
 zerop, plusp, minusp, oddp,
 evenp
 max, min
 +, -, *, /

abs
 truncate
 rem, mod
 gcd, lcm
 numerator, denominator
 floor, truncate, ceiling,
 round

rationalize

exp, log, sin, cos, tan,
 asin, acos, atan
 sqrt
 expt
 complex

real-part, imag-part
 abs, phase

prin1-to-string

parse-integer,
 read-from-string

Poznámka pro CL

complexp vrací nil pro racionální
 čísla

CL nezná pojem přesného čísla

Vždy s libovolným počtem argu-
 mentů (u + a * i nulovým)

Jako druhou hodnotu vrací zbytek.

Je možno zadat i dělitel (default 1),
 vrací podíl (zaokrouhlený) a zby-
 tek.

Nemá druhý argument; počítá
 vždy co nejpřesněji.
 log připouští druhý parametr —
 základ (default e)

Není, je třeba použít abs a cis.

CL nezná pojem přesného čísla.

Jedna z obecných funkcí pro tisk
 do řetězce; netýká se pouze čísel.
 Soustava se nastavuje dynamickou
 proměnnou *print-base*.
 Přibližně

A.6. Standardní procedury — logické typy

Scheme	Common Lisp	Poznámka pro CL
not (boolean? x)	not (typep x 'boolean)	

A.7. Standardní procedury — páry a seznamy

Scheme	Common Lisp	Poznámka pro CL
Scheme (pair? x) cons, car, cdr (set-car! x y), (set-cdr! x y) caar, caddr, ... (null? x) (list? x)	Common Lisp (consp x), (typep x 'cons) cons, car, cdr (setf (car x) y) (setf (cdr x) y) caar, caddr, ... (null x), (typep x 'null) (listp x), (typep x 'list)	Poznámka pro CL Poznámka pro CL
list length	list list-length	Vrací y. Viz též endp. Nezkoumá ale, zda není seznam kruhový a zda poslední cdr je ().
append reverse (list-tail list n) (list-ref list n)	append reverse (nthcdr n list) (nth n list), (elt list n)	length je rovněž použitelné, dělá ale něco mírně jiného.
memq, memv, member	member	Přibližně. nth je přesný překlad, elt je obecnější.
assq, assv, assoc	assoc	Je obecnější, porovnávací funkce se zadává parametrem. Je obecnější, porovnávací funkce se zadává parametrem.

A.8. Standardní procedury — symboly

Scheme	Common Lisp	Poznámka pro CL
(symbol? x)	(symbolp x), (typep x 'symbol)	
symbol->string string->symbol	symbol-name intern	Přibližně. Standardně je třeba používat velká písmena.

A.9. Standardní procedury — znaky

Scheme	Common Lisp	Poznámka pro CL
(char? x)	(characterp x), (typep x 'character)	
char=?, char<?, char>?, char<=?, char>=?	char=, char<, char>, char<=, char>=	Všechny akceptují lib. počet parametrů.
char-ci=?, char-ci<?, char-ci>?, char-ci<=?, char-ci>=?	char-equal, char-lessp, char-greaterp, char-not-greaterp, char-not-lessp	Všechny akceptují lib. počet parametrů.
char-alphabetic? char-numeric?	alpha-char-p digit-char-p	Volitelně je možno zadat soustavu, jako true vrací příslušnou číselnou hodnotu.
char-whitespace?		Není. Souvisí to s tím, že v CL lze měnit syntax.
char-upper-case? char-lower-case? char->integer integer->char	upper-case-p lower-case-p char-code code-char	

Scheme	Common Lisp	Poznámka pro CL
char-upcase, char-downcase	char-upcase, char-downcase	

A.10. Standardní procedury — řetězce

Scheme	Common Lisp	Poznámka pro CL
(string? x)	(stringp x), (typep x 'string)	
(make-string k),	(make-string k),	
(make-string k fill)	(make-sequence 'string k k :initial-element fill) (make-sequence 'string k :initial-element fill)	
string-length	length	Pracuje nejen s řetězcí.
(string-ref str k)	(char str k), (elt str k), (aref str k)	Přesná obdoba je char, lépe je používat elt nebo aref.
(string-set! str k char)	(setf (char str k) char), (setf (elt str k) char), (setf (aref str k) char)	Vrací char.
string=?, string<?, string>?, string<=?, string>=?	string=, string<, string>, string<=, string>=	Všechny akceptují lib. počet parametrů.
string-ci=?, string-ci<?, string-ci>?, string-ci<=?, string-ci>=?	string-equal, string-lessp, string-greaterp, string-not-greaterp, string-not-lessp	Všechny akceptují lib. počet parametrů.
substring	subseq	Pracuje nejen s řetězcí, poslední parametr je volitelný.
(string-append str1 ...)	(concatenate 'string str1 ...)	Pracuje nejen s řetězcí; aby byl výsledek řetězec, je třeba použít 'string jako první parametr.
(string->list str), (list->string lst)	(coerce str 'list), (coerce lst 'string)	
string-copy	copy-seq	Pracuje nejen s řetězcí.
string-fill!	fill	Pracuje nejen s řetězcí, volitelně lze určit rozmezí.

A.11. Standardní procedury — vektory

Všechny zde uvedené funkce lze použít i na řetězce, protože v Common Lispu jsou řetězce vektory.

Scheme	Common Lisp	Poznámka pro CL
(vector? x)	(vectorp x), (typep x 'vector)	
(make-vector k),	(make-vector k),	
(make-vector k fill)	(make-sequence 'vector k k :initial-element fill) (make-sequence 'vector k :initial-element fill)	
vector	vector	
vector-length	length	
(vector-ref vec k)	(elt vec k), (aref vec k)	
(vector-set! vec k x)	(setf (elt vec k) x), (setf (aref vec k) x)	
(vector->list vec), (list->vector lst)	(coerce vec 'list), (coerce lst 'vector)	

Scheme
vector-fill!

Common Lisp
fill

Poznámka pro CL
Pracuje nejen s vektory, volitelně lze určit rozmezí.

A.12. Standardní procedury — řízení běhu

Scheme
(procedure? x)

apply
map

for-each
force
call-with-current-continuation
values
call-with-values
dynamic-wind

Common Lisp
(functionp x),
(typep x 'function)
apply
mapcar

mapc

values
multiple-value-call
unwind-protect

Poznámka pro CL

Mírnější podmínky. Pozor, funkce map dělá něco jiného.

Neexistuje.
Neexistuje.

Mírnější podmínky.
Přibližně.
Velmi přibližně; unwind-protect je speciální operátor a je jednodušší, protože není call-with-current-continuation.

A.13. Standardní procedury — eval

Ve Scheme přijímá eval druhý parametr — prostředí, v němž se vyhodnocení provede. Tento parametr může nabývat pouze hodnot vracených procedurami scheme-report-environment, null-environment a interaction-environment. V CL druhý parametr chybí a vyhodnocení se provádí v aktuálním dynamickém prostředí.

Scheme
eval
scheme-report-environment,
null-environment,
interaction-environment

Common Lisp
eval

Poznámka pro CL
Rozdíly viz výše.
Neexistují, viz výše.

A.14. Standardní procedury — vstup a výstup

Scheme
call-with-input-file,
call-with-output-file
input-port?, output-port?

current-input-port,
current-output-port

with-input-from-file,
with-output-to-file
open-input-file,
open-output-file
close-input-port,
close-output-port
read
read-char
peek-char
eof-object?

char-ready?

Common Lisp
with-open-file

input-stream-p,
output-stream-p
debug-io, *error-output*,
query-io,
standard-input,
standard-output,
trace-output
with-open-file

open

close

read
read-char
peek-char

Poznámka pro CL
Přibližně. Makro s více možnostmi.

Dynamické proměnné, výběr závisí na účelu.

Přibližně. Makro s více možnostmi.

Více možností, zadávají se parametry.

Neexistuje, konec souboru se zjišťuje jinak.
Neexistuje, používat read-char-no-hang

Scheme`write``display``newline``write-char`**Common Lisp**`prinl``princ``terpri``write-char`**Poznámka pro CL**

Jedna z mnoha funkcí pro zápis objektu. Neplést s funkcí `write`.

Jedna z mnoha funkcí pro zápis objektu.

Viz též `fresh-line`.

A.15. Standardní procedury — systémové rozhraní

Scheme`load``transcript-on,``transcript-off`**Common Lisp**`load`**Poznámka pro CL**

Více možností

Neexistuje, dělá se jinak.

B. Knihovna micro-graphics: seznam použitelných barev

:ALICEBLUE		:ANTIQUWHITE		:ANTIQUWHITE1	
:ANTIQUWHITE2		:ANTIQUWHITE3		:ANTIQUWHITE4	
:AQUAMARINE		:AQUAMARINE1		:AQUAMARINE2	
:AQUAMARINE3		:AQUAMARINE4		:AZURE	
:AZURE1		:AZURE2		:AZURE3	
:AZURE4		:BEIGE		:BISQUE	
:BISQUE1		:BISQUE2		:BISQUE3	
:BISQUE4		:BLACK		:BLANCHEDALMOND	
:BLUE		:BLUE1		:BLUE2	
:BLUE3		:BLUE4		:BLUEVIOLET	
:BROWN		:BROWN1		:BROWN2	
:BROWN3		:BROWN4		:BURLYWOOD	
:BURLYWOOD1		:BURLYWOOD2		:BURLYWOOD3	
:BURLYWOOD4		:CADETBBLUE		:CADETBBLUE1	
:CADETBBLUE2		:CADETBBLUE3		:CADETBBLUE4	
:CHARTREUSE		:CHARTREUSE1		:CHARTREUSE2	
:CHARTREUSE3		:CHARTREUSE4		:CHOCOLATE	
:CHOCOLATE1		:CHOCOLATE2		:CHOCOLATE3	
:CHOCOLATE4		:CONFIRMER-BACKGROUND		:CORAL	
:CORAL1		:CORAL2		:CORAL3	
:CORAL4		:CORNFLOWERBLUE		:CORNSILK	
:CORNSILK1		:CORNSILK2		:CORNSILK3	
:CORNSILK4		:CYAN		:CYAN1	
:CYAN2		:CYAN3		:CYAN4	
:DARK-BLUE		:DARKGOLDENROD		:DARKGOLDENROD1	
:DARKGOLDENROD2		:DARKGOLDENROD3		:DARKGOLDENROD4	
:DARKGREEN		:DARKKHAKI		:DARKLIVEGREEN	
:DARKLIVEGREEN1		:DARKLIVEGREEN2		:DARKLIVEGREEN3	
:DARKLIVEGREEN4		:DARKORANGE		:DARKORANGE1	
:DARKORANGE2		:DARKORANGE3		:DARKORANGE4	
:DARKORCHID		:DARKORCHID1		:DARKORCHID2	
:DARKORCHID3		:DARKORCHID4		:DARKSALMON	
:DARKSEAGREEN		:DARKSEAGREEN1		:DARKSEAGREEN2	
:DARKSEAGREEN3		:DARKSEAGREEN4		:DARKSLATEBLUE	
:DARKSLATEGRAY		:DARKSLATEGRAY1		:DARKSLATEGRAY2	
:DARKSLATEGRAY3		:DARKSLATEGRAY4		:DARKSLATEGREY	
:DARKTURQUOISE		:DARKVIOLET		:DEEPPINK	
:DEEPPINK1		:DEEPPINK2		:DEEPPINK3	
:DEEPPINK4		:DEEPSKYBLUE		:DEEPSKYBLUE1	
:DEEPSKYBLUE2		:DEEPSKYBLUE3		:DEEPSKYBLUE4	
:DIMGRAY		:DIMGREY		:DODGERBLUE	
:DODGERBLUE1		:DODGERBLUE2		:DODGERBLUE3	
:DODGERBLUE4		:FIREBRICK		:FIREBRICK1	
:FIREBRICK2		:FIREBRICK3		:FIREBRICK4	
:FLORALWHITE		:FORESTGREEN		:GAINSBORO	
:GHOSTWHITE		:GOLD		:GOLD1	
:GOLD2		:GOLD3		:GOLD4	
:GOLDENROD		:GOLDENROD1		:GOLDENROD2	
:GOLDENROD3		:GOLDENROD4		:GRAY	
:GRAY-BLUE		:GRAY0		:GRAY1	
:GRAY10		:GRAY100		:GRAY11	
:GRAY12		:GRAY13		:GRAY14	
:GRAY15		:GRAY16		:GRAY17	
:GRAY18		:GRAY19		:GRAY2	
:GRAY20		:GRAY21		:GRAY22	
:GRAY23		:GRAY24		:GRAY25	
:GRAY26		:GRAY27		:GRAY28	
:GRAY29		:GRAY3		:GRAY30	
:GRAY31		:GRAY32		:GRAY33	
:GRAY34		:GRAY35		:GRAY36	
:GRAY37		:GRAY38		:GRAY39	
:GRAY4		:GRAY40		:GRAY41	
:GRAY42		:GRAY43		:GRAY44	
:GRAY45		:GRAY46		:GRAY47	
:GRAY48		:GRAY49		:GRAY5	

:GRAY50		:GRAY51		:GRAY52	
:GRAY53		:GRAY54		:GRAY55	
:GRAY56		:GRAY57		:GRAY58	
:GRAY59		:GRAY6		:GRAY60	
:GRAY61		:GRAY62		:GRAY63	
:GRAY64		:GRAY65		:GRAY66	
:GRAY67		:GRAY68		:GRAY69	
:GRAY7		:GRAY70		:GRAY71	
:GRAY72		:GRAY73		:GRAY74	
:GRAY75		:GRAY76		:GRAY77	
:GRAY78		:GRAY79		:GRAY8	
:GRAY80		:GRAY81		:GRAY82	
:GRAY83		:GRAY84		:GRAY85	
:GRAY86		:GRAY87		:GRAY88	
:GRAY89		:GRAY9		:GRAY90	
:GRAY91		:GRAY92		:GRAY93	
:GRAY94		:GRAY95		:GRAY96	
:GRAY97		:GRAY98		:GRAY99	
:GREEN		:GREEN1		:GREEN2	
:GREEN3		:GREEN4		:GREENYELLOW	
:GREY		:GREY0		:GREY1	
:GREY10		:GREY100		:GREY11	
:GREY12		:GREY13		:GREY14	
:GREY15		:GREY16		:GREY17	
:GREY18		:GREY19		:GREY2	
:GREY20		:GREY21		:GREY22	
:GREY23		:GREY24		:GREY25	
:GREY26		:GREY27		:GREY28	
:GREY29		:GREY3		:GREY30	
:GREY31		:GREY32		:GREY33	
:GREY34		:GREY35		:GREY36	
:GREY37		:GREY38		:GREY39	
:GREY4		:GREY40		:GREY41	
:GREY42		:GREY43		:GREY44	
:GREY45		:GREY46		:GREY47	
:GREY48		:GREY49		:GREY5	
:GREY50		:GREY51		:GREY52	
:GREY53		:GREY54		:GREY55	
:GREY56		:GREY57		:GREY58	
:GREY59		:GREY6		:GREY60	
:GREY61		:GREY62		:GREY63	
:GREY64		:GREY65		:GREY66	
:GREY67		:GREY68		:GREY69	
:GREY7		:GREY70		:GREY71	
:GREY72		:GREY73		:GREY74	
:GREY75		:GREY76		:GREY77	
:GREY78		:GREY79		:GREY8	
:GREY80		:GREY81		:GREY82	
:GREY83		:GREY84		:GREY85	
:GREY86		:GREY87		:GREY88	
:GREY89		:GREY9		:GREY90	
:GREY91		:GREY92		:GREY93	
:GREY94		:GREY95		:GREY96	
:GREY97		:GREY98		:GREY99	
:HIGHLIGHT-RED		:HONEYDEW		:HONEYDEW1	
:HONEYDEW2		:HONEYDEW3		:HONEYDEW4	
:HOTPINK		:HOTPINK1		:HOTPINK2	
:HOTPINK3		:HOTPINK4		:INDIANRED	
:INDIANRED1		:INDIANRED2		:INDIANRED3	
:INDIANRED4		:IVORY		:IVORY1	
:IVORY2		:IVORY3		:IVORY4	
:KHAKI		:KHAKI1		:KHAKI2	
:KHAKI3		:KHAKI4		:LAVENDER	
:LAVENDERBLUSH		:LAVENDERBLUSH1		:LAVENDERBLUSH2	
:LAVENDERBLUSH3		:LAVENDERBLUSH4		:LAWNGREEN	
:LEMONCHIFFON		:LEMONCHIFFON1		:LEMONCHIFFON2	
:LEMONCHIFFON3		:LEMONCHIFFON4		:LIGHT-BLUE	
:LIGHT-BROWN		:LIGHT-RED		:LIGHTBLUE	
:LIGHTBLUE1		:LIGHTBLUE2		:LIGHTBLUE3	
:LIGHTBLUE4		:LIGHTCORAL		:LIGHTCYAN	

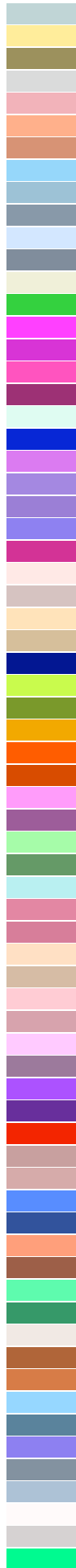
:LIGHTCYAN1
:LIGHTCYAN4
:LIGHTGOLDENROD2
:LIGHTGOLDENRODYELLOW
:LIGHTPINK
:LIGHTPINK3
:LIGHTSALMON1
:LIGHTSALMON4
:LIGHTSKYBLUE1
:LIGHTSKYBLUE4
:LIGHTSLATEGREY
:LIGHTSTEELBLUE2
:LIGHTYELLOW
:LIGHTYELLOW3
:LINEN
:MAGENTA1
:MAGENTA4
:MAROON2
:MEDIUM-BLUE
:MEDIUM-YELLOW
:MEDIUMORCHID
:MEDIUMORCHID3
:MEDIUMPURPLE1
:MEDIUMPURPLE4
:MEDIUMSPRINGGREEN
:MIDNIGHTBLUE
:MISTYROSE1
:MISTYROSE4
:NAVAJOWHITE1
:NAVAJOWHITE4
:OLDLACE
:OLIVEDRAB2
:ORANGE
:ORANGE3
:ORANGERED1
:ORANGERED4
:ORCHID2
:PALEGOLDENROD
:PALEGREEN2
:PALETURQUOISE
:PALETURQUOISE3
:PALEVIOLETTRED1
:PALEVIOLETTRED4
:PEACHPUFF1
:PEACHPUFF4
:PINK1
:PINK4
:PLUM2
:POWDERBLUE
:PURPLE2
:RED
:RED3
:ROSYBROWN1
:ROSYBROWN4
:ROYALBLUE2
:SADDLEBROWN
:SALMON2
:SANDYBROWN
:SEAGREEN2
:SEASHELL
:SEASHELL3
:SIENNA1
:SIENNA4
:SKYBLUE2
:SLATEBLUE
:SLATEBLUE3
:SLATEGRAY1
:SLATEGRAY4
:SNOW1
:SNOW4



































:LIGHTCYAN2
:LIGHTGOLDENROD
:LIGHTGOLDENROD3
:LIGHTGRAY
:LIGHTPINK1
:LIGHTPINK4
:LIGHTSALMON2
:LIGHTSEAGREEN
:LIGHTSKYBLUE2
:LIGHTSLATEBLUE
:LIGHTSTEELBLUE
:LIGHTSTEELBLUE3
:LIGHTYELLOW1
:LIGHTYELLOW4
:LISPWORKS-BLUE
:MAGENTA2
:MAROON
:MAROON3
:MEDIUM-BROWN
:MEDIUMAQUAMARINE
:MEDIUMORCHID1
:MEDIUMORCHID4
:MEDIUMPURPLE2
:MEDIUMSEAGREEN
:MEDIUMTURQUOISE
:MINTCREAM
:MISTYROSE2
:MOCCASIN
:NAVAJOWHITE2
:NAVY
:OLIVEDRAB
:OLIVEDRAB3
:ORANGE1
:ORANGE4
:ORANGERED2
:ORCHID
:ORCHID3
:PALEGREEN
:PALEGREEN3
:PALETURQUOISE1
:PALETURQUOISE4
:PALEVIOLETTRED2
:PAPAYAWHIP
:PEACHPUFF2
:PERU
:PINK2
:PLUM
:PLUM3
:PURPLE
:PURPLE3
:RED1
:RED4
:ROSYBROWN2
:ROYALBLUE
:ROYALBLUE3
:SALMON
:SALMON3
:SEAGREEN
:SEAGREEN3
:SEASHELL1
:SEASHELL4
:SIENNA2
:SKYBLUE
:SKYBLUE3
:SLATEBLUE1
:SLATEBLUE4
:SLATEGRAY2
:SLATEGREY
:SNOW2
:SPRINGGREEN


















:LIGHTCYAN3
:LIGHTGOLDENROD1
:LIGHTGOLDENROD4
:LIGHTGREY
:LIGHTPINK2
:LIGHTSALMON
:LIGHTSALMON3
:LIGHTSKYBLUE
:LIGHTSKYBLUE3
:LIGHTSLATEGREY
:LIGHTSTEELBLUE1
:LIGHTSTEELBLUE4
:LIGHTYELLOW2
:LIMEGREEN
:MAGENTA
:MAGENTA3
:MAROON1
:MAROON4
:MEDIUM-GREEN
:MEDIUMBLUE
:MEDIUMORCHID2
:MEDIUMPURPLE
:MEDIUMPURPLE3
:MEDIUMSLATEBLUE
:MEDIUMVIOLETTRED
:MISTYROSE
:MISTYROSE3
:NAVAJOWHITE
:NAVAJOWHITE3
:NAVYBLUE
:OLIVEDRAB1
:OLIVEDRAB4
:ORANGE2
:ORANGERED
:ORANGERED3
:ORCHID1
:ORCHID4
:PALEGREEN1
:PALEGREEN4
:PALETURQUOISE2
:PALEVIOLETTRED
:PALEVIOLETTRED3
:PEACHPUFF
:PEACHPUFF3
:PINK
:PINK3
:PLUM1
:PLUM4
:PURPLE1
:PURPLE4
:RED2
:ROSYBROWN
:ROSYBROWN3
:ROYALBLUE1
:ROYALBLUE4
:SALMON1
:SALMON4
:SEAGREEN1
:SEAGREEN4
:SEASHELL2
:SIENNA
:SIENNA3
:SKYBLUE1
:SKYBLUE4
:SLATEBLUE2
:SLATEGREY
:SLATEGRAY3
:SNOW
:SNOW3
:SPRINGGREEN1



:SPRINGGREEN2	
:STEELBLUE	
:STEELBLUE3	
:TAN1	
:TAN4	
:THISTLE2	
:TOMATO	
:TOMATO3	
:TURQUOISE	
:TURQUOISE3	
:VIOLETRED	
:VIOLETRED3	
:WHEAT1	
:WHEAT4	
:YELLOW	
:YELLOW3	

:SPRINGGREEN3	
:STEELBLUE1	
:STEELBLUE4	
:TAN2	
:THISTLE	
:THISTLE3	
:TOMATO1	
:TOMATO4	
:TURQUOISE1	
:TURQUOISE4	
:VIOLETRED1	
:VIOLETRED4	
:WHEAT2	
:WHITE	
:YELLOW1	
:YELLOW4	

:SPRINGGREEN4	
:STEELBLUE2	
:TAN	
:TAN3	
:THISTLE1	
:THISTLE4	
:TOMATO2	
:TRANSPARENT	
:TURQUOISE2	
:VIOLET	
:VIOLETRED2	
:WHEAT	
:WHEAT3	
:WHITESMOKE	
:YELLOW2	
:YELLOWGREEN	