

Objektové programování

Poznámky k přednášce

Michal Krupka

1. listopadu 2016

Obsah

1	Od Scheme k Lispu	5
1.1	Základní rozdíly mezi Schemem a Common Lispem	6
1.2	Common Lisp: základní výbava	12
2	Objekty a třídy	35
2.1	Základní pojmy	35
2.2	Třídy a instance v Common Lispu	38
2.3	Inicializace slotů nových instancí	47
	ÚLOHY	50
3	Zapouzdření a polymorfismus	53
3.1	Princip zapouzdření	53
3.2	Úprava tříd <code>point</code> a <code>circle</code>	59
3.3	Třída <code>picture</code>	62
3.4	Vlastnosti	65
3.5	Kreslení pomocí knihovny <code>micro-graphics</code>	66
3.6	Kreslení grafických objektů	70
3.7	Princip polymorfismu	79
3.8	Polygony	81
3.9	Geometrické transformace	85
	ÚLOHY	87
4	Dědičnost	89
4.1	Princip dědičnosti a pravidlo <i>is-a</i>	89
4.2	Určení předka v definici třídy	96
4.3	Poznámka o běžných jazycích	100
4.4	Přepisování metod	100
4.5	Volání zděděné metody	102
4.6	Inicializace instancí	108

4.7	Návrh stromu dědičnosti	109
	ÚLOHY	110
5	Překreslování oken a hlášení změn	113
5.1	Zpětná volání v knihovně <code>micro-graphics</code>	113
5.2	Překreslování oken	115
5.3	Překreslení při změně okna	116
5.4	Překreslování při změnách objektů	118
	ÚLOHY	122
A	Slovníček Scheme-Lisp	125
B	Klávesové zkratky v LW	133
B.1	Režim Windows	133
B.2	Režim Mac OS	134
C	Knihovna <code>micro-graphics</code>	137
D	Knihovna <code>micro-graphics</code>: seznam použitelných barev	143

Kapitola 1

Od Scheme k Lispu

Než začneme s objektovým programováním, je třeba se seznámit se základy programovacího jazyka, který budeme používat: s Common Lispem. K tomu slouží tato kapitola. Výklad vychází z toho, že čtenář je již z předchozích kurzů seznámen s příbuzným jazykem Scheme. Kromě zvládnutí základů jazyka, tak jak budou popsány v této kapitole, je vhodné, aby se čtenář dobře seznámil s vývojovým prostředím [LispWorks](#), kterému se v textu nevěnujeme. [Uživatelské příručky](#) lze nalézt buď na webu LispWorks, nebo přímo v aplikaci.

Účelem této části je poskytnout uživatelům jazyka Scheme informace potřebné k zahájení práce s Common Lispem. Čtenář se dozví o základních rozdílech mezi těmito jazyky, zejména terminologických a rozdílech ve vyhodnocovacím procesu. Dále pak uvádíme další informace o Common Lispu, 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 Příloze [A](#) je uveden slovníček, ve kterém čtenář najde ke každému symbolu definovanému ve standardu R⁵RS jazyka Scheme jeho ekvivalent v Common Lispu. Tento slovníček si projděte. Některé rozdíly mohou být z počátku matoucí (např. neexistence operátoru podobného schemovskému `define`).

1.1. Základní rozdíly mezi Schemem a Common Lispem

Rozdíly v terminologii

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

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 (obvyklé je vrácení jedné hodnoty). Kromě toho může funkce také vykonat vedlejší efekt.

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): 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*. Z praktického hlediska není nutné rozlišovat mezi speciálními operátory a symboly, pojmenovávajícími makra (kromě případů, kdy makra sami definujeme; speciální operátory definovat nejdou).

Uvedme si příklad. Seznam

```
(let ((a 1)
      (b 2))
  (or (+ a b) (- a b)))
```

je výraz, protože je součástí zdrojového kódu. Je taky formou, protože je určen k vyhodnocení. Je speciální formou, protože symbol `let` je speciální operátor.

Podvýraz `(a 1)` je výraz, ale není to forma. Symbol `or` je výraz, není forma, je operátor. Výraz `(or (+ a b) (- a b))` je makro forma, výraz `(+ a b)` je funkční forma.

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 se vyhodnocují samy na sebe, podobně jako například čísla:

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

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

Zařídit, aby hodnotou symbolu byl dotyčný 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 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 1.1.1 (užití `and` a `or` na větvení)

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)
```

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

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`).

Rozdíly ve vyhodnocovacím procesu

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

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 1.1.2 (tentýž symbol jako proměnná i funkce)

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


```
(defun encap-car (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?).

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.

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` (nebo použít klávesovou zkratku). Pokud jsme funkci `encap-car` definovali stejně jako zde, dozvíme se, že seznam parametrů této funkce je `(list)`. 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.

Existence dvojí vazby symbolů vede k několika komplikacím. 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 uveďme následující definici procedury na kompozici dvou procedur v jazyce Scheme:

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

neboli

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

Analogická definice v Common Lispu by vypadala takto:

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

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.

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) vracel 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`. 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)
```

U většiny funkcí v Common Lispu, které požadují jako argument funkci, je místo funkce možné zadat její název (tj. symbol). To platí i pro funkci `funcall`; proto je kompozici funkcí `car` a `cdr` možno vytvořit i takto:

```
(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 1.1 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.

1.2. Common Lisp: základní výbava

Abychom mohli při výkladu objektového programování Common Lisp používat, musíme si jej alespoň do určité míry osvojit. Tato podkapitola se věnuje vysvětlení části jazyka, kterou budeme v textu potřebovat. Popíšeme část z 978 symbolů, které tvoří názvy speciálních operátorů, maker, funkcí a dalších symbolů v Common Lispu. Tyto symboly budou tvořit naši základní výbavu pro práci v jazyce. Některé podrobně vysvětlíme, u jiných odkážeme čtenáře na definici ve standardu. U každého symbolu ze základní výbavy očekáváme, že o jeho existenci bude čtenář vědět a že jej bude schopen (s případným nahlédnutím do standardu) správně, rychle a účinně použít.

Podmíněné vyhodnocení, cykly

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.

V Common Lispu není rozdíl mezi *příkazem* a *výrazem*. Proto lze jak k podmíněnému vykonání příkazu, tak k podmíněnému získání hodnoty výrazu použít tentýž operátor *if*. 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))
```

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 1.1: Zjednodušený vyhodnocovací proces v Common Lispu

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
```

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

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

Proměnné a vazby

Hodnotové vazby symbolů 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 lexikální, 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.

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
```

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. 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)`.

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*.

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

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. 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 1.2.1 (makro *defvar* inicializuje jen poprvé)

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))`.

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.

Zopakujme, 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 naváží 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`*

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 1.2.2 (Makro `setf` a proměnné)

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
```

Při testování v příkazovém řádku je možné makrem `setf` nastavit i hodnotu proměnné, která nebyla předtím definována pomocí makra `defvar` (na mnoha místech tohoto textu to děláme). V principu to ovšem není správně a v programu je nutné každou proměnnou před použitím definovat.

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 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.

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`

Funkce

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

Příklad 1.2.3

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 hodnotu, se kterou je funkce volána, ale proměnnou, na kterou je v průběhu výkonu funkce některý z argumentů navázán.

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

Při definici nových funkcí (například pomocí operátorů `defun` a `lambda`, které popíšeme za chvíli, nebo `labels`, který známe ze Scheme), 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 1.2.4 (λ -seznamy funkcí `car`, `cdr` a `cons`)

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
lambda-list: obyčejný  $\lambda$ -seznam
form: forma
```

Makro `defun` vytváří novou globální funkci jménem *function-name* (nastavuje tedy globální 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 1.2.5 (makro `defun`)

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
```

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
```

lambda-list: obyčejný λ -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*. Na rozdíl 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.

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 1.2.6 (λ -výraz jako operátor)

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.

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`. Popis druhé lze najít ve standardu, v zásadě pracuje stejně jako stejnojmenná funkce ve Scheme.

Funkce vytvářené operátory `defun` a `lambda` jsou lexikální uzávěry, což je pojem, který známe už ze Scheme.

Symbolsy do základní výbavy: `defun`, `lambda`, `function`, `funcall`, `apply`

Logické operace

O logických hodnotách v Common Lispu jsme se už zmínili. Ve všech operátorech, které pracují s logickými hodnotami, symbol `nil` reprezentuje hodnotu *nepravda* 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.

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)))))
```

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

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

Porovnávání

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

Příklad 1.2.7 (shodnost seznamů)

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 1.2.8 (shodnost čísel)

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 `eq` a predikát `equalp`. Oba přijímají dva argumenty a oba vracejí hodnotu *pravda*, pokud si tyto argumenty jsou v jistém smyslu rovny.

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

Příklad 1.2.9 (funkce `eq`)

Například:

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

CL-USER 22 > (eq 1 1)
```



```
T
CL-USER 23 > (eql (list 1 2) (list 1 2))
NIL
CL-USER 24 > (eql 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 `eql`, jsou také určitě `equalp`.

Příklad 1.2.10 (funkce `equalp`)

```
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 `eql` a `equalp` je ve standardu.

Symbole do základní výbavy: `eql`, `equalp`

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).

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

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.

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`, `caaadr`, `caadar`, `caaddr`, `cadaar`, `cadadr`, `caddar`, `cadddr`, `cdaaar`, `cdaadr`, `cdadar`, `cdaddr`, `cdbaar`, `cddadr`, `cdddar`, `cddddr`. 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. *tečkové 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 obracení 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 1.2.11 (skalární součin pomocí funkce `mapcar`)

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 `eq`. 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.

Symboly do základní výbavy: cons, car, cdr, caar, cadr, cdar, cddr, caaar, caadr, caadar, caaddr, cadaar, cadadr, caddar, caddr, cdaaar, cdaadr, cdadar, cdaddr, cdbaar, cddadr, cdddar, cdddr, 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

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 vyjimečnému 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 1.2.12 (bezpečný faktoriál)

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 ar-
gument.
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 >
```

Symbol do základní výbavy: error

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 1.2.13 (funkce `format`)

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).

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.

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

Typy

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`.

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

Listener

V této podkapitole uvedeme pouze tři symboly užitečné při interaktivní práci s Common Lisem: *, **, ***. V příkazovém řádku (Listeneru) 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: *, **, ****

Poznámka k jazykovému a programovacímu stylu

Správný zdrojový kód programu splňuje základní stylové požadavky v jazyce. V Common Lispu například velbloudí notace, podtržítka v názvech

funkcí, závorky na samostatném řádku apod. nejsou povoleny; názvy predikátů musí končit na `p`, nikoli na otazník jako ve Scheme; řádky musí být správně odsazovány a nesmí být příliš dlouhé atd. Výběr vhodných názvů je důležitý. Dobrým vodítkem pro správný jazykový styl mohou být zdrojové kódy k tomuto textu.

Kromě jazykového stylu je třeba také dodržovat zásady stylu programátorského. Kromě zásad daných v objektovém programování jde i o zásady obecné, které pomáhají udržet pohromadě i program většího rozsahu. Zkušenost říká, že je dobré je dodržovat v malých i velkých programech.

Kapitola 2

Objekty a třídy

2.1. Základní pojmy

Objektové programování (objektově orientované programování, OOP) je založeno na přístupu, ve kterém je běžící program složen ze samostatných a nezávislých jednotek, zvaných *objekty*.

Prvním objektově orientovaným jazykem byl jazyk Simula 67, vytvořený v 60. letech minulého století v Norském výpočetním středisku v Oslu. Jak i název napovídá, jazyk byl navržen tak, aby byl vhodný pro simulace reálných procesů. Z reality je převzatá základní myšlenka programu jako souhrnu samostatných a vzájemně komunikujících objektů. Velmi důležitým objektovým programovacím jazykem je SmallTalk, který vznikl začátkem 70. let v Xerox Palo Alto Research Center. Objektový systém Common Lispu (Common Lisp Object System, CLOS), kterému se budeme věnovat v tomto textu, vznikl ze starších objektových systémů CommonLOOPS a MIT Flavors. Common Lisp je historicky prvním standardizovaným objektovým programovacím jazykem (ANSI standard).

Zopakujme znovu dvě základní charakteristiky objektů: *samostatnost* a *nezávislost*.

Samostatnost a nezávislost objektů

Samostatnost objektů znamená, že jim stačí říct, *co* mají dělat, není nutné specifikovat *jak*. Objekt by měl být pro programátora, který ho používá, co nejjednodušší k použití, měl by od něj vyžadovat co

nejmenší znalosti a odstínit ho od nepodstatných technických detailů řešení problému.

Nezávislost objektů umožňuje jejich *znovupoužitelnost*, neboli zachování funkčnosti po přenesení do jiného prostředí (programu). Objekt je tím nezávislejší, čím méně ke své práci vyžaduje existenci vnějších faktorů.

Tyto dvě charakteristiky si dobře zapamatujte. V budoucnu je budeme (a budete) používat k posuzování kvality napsaného programu.

Objekty během života programu vznikají, existují a zanikají. Tímto dynamickým charakterem se objekty odlišují od *modulů*, což jsou pouze části zdrojového kódu programu (obvykle se shodují se zdrojovými soubory, například v C).

Každý objekt je zodpovědný za určitou část činnosti programu. Koná ji na základě pokynů, které mu udělujeme zasíláním *zpráv*. Ty jsou identifikovány jménem. Součástí posílané zprávy mohou být i argumenty, podobně jako u volání funkce. Výsledkem zaslání zprávy může být (podobně jako u volání funkce) vrácená hodnota.

Na přijetí zprávy objekt reaguje tak, že spustí kód, který vykoná činnost, jež je po něm zprávou požadována. Tomuto kódu se říká *metoda*. Metoda je zvláštní druh funkce, má jméno a lze ji volat s argumenty. Každý objekt může obsahovat více metod, podle toho, jaké zprávy může přijímat. Po přijetí zprávy se spustí metoda stejného jména, jaké má přijímaná zpráva, a spustí se s těmi argumenty, se kterými byla zpráva poslána. Této metodě se říká *obsluha* zprávy. Procesu zavolání obsluhy zprávy se říká její *obsloužení*. Stejně jako funkce mají i metody návratové hodnoty. Ty jsou nakonec výsledkem zaslání zprávy.

Metody lze chápat jako kód, který se nachází *uvnitř objektu* (tuhle věc ještě za chvíli upřesníme). Při psaní programu je třeba důsledně odlišovat mezi autorem tohoto kódu a *uživatelé objektu*, tj. programátorem, který objekt ve svém programu používá. A to bez ohledu na to, že to může být tentýž člověk. Uživatel objektu nemusí znát informace o implementaci objektu a neměl by být nucen dělat činnosti, které by měl objekt zvládnout sám.

Objekty mají *vnitřní stav*. To je souhrn dat, která objekt obsahuje. V reakci na přijetí zprávy může objekt svůj vnitřní stav změnit. To je kromě vrácení hodnoty další efekt, který může zaslání zprávy objektu mít.

Příklad 2.1.1 (Naivní implementace objektů)

Základní principy objektového programování lze s výhodou používat i v neobjektových jazycích. Ukážeme si jednoduchý příklad objektu představujícího bod v rovině. Nejprve napíšeme funkci `make-point` na vytvoření nového bodu. Ten budeme reprezentovat párem, do jehož složek budeme ukládat souřadnice bodu.

```
(defun make-point ()  
  (cons 0 0))
```

Posílání zpráv bodu budeme simulovat voláním funkcí. Tyto funkce budou představovat metody objektu.

```
(defun point-x (point)  
  (car point))  
  
(defun point-y (point)  
  (cdr point))  
  
(defun set-point-x (point value)  
  (setf (car point) value)  
  point)  
  
(defun set-point-y (point value)  
  (setf (cdr point) value)  
  point)
```

Pravidlo, že s objekty komunikujeme posíláním zpráv, se zde projeví tím, že pro práci se souřadnicemi bodů budeme používat pouze funkce `point-x`, `point-y`, `set-point-x` a `set-point-y`. To nám umožní v budoucnu změnit implementaci bodu, aniž by se to navenek jakkoli projevilo.

Zamyslete se, jak jsme v tomto příkladě dodrželi princip samostatnosti a nezávislosti objektů.

Data v objektech jsou (podobně jako u struktur v jazyce C) rozdělena do pojmenovaných položek, kterým budeme říkat *sloty* (to je obvyklé řešení, uvedený příklad ale ukazuje, že to není nutné).

Podobně jako jiná data, i objekty mohou být různých typů. V objektovém programování se pro základní typy objektů používá pojem *třídy*. Na tomto místě uvedeme zjednodušenou definici třídy, kterou v dalších částech rozšíříme.

Aby dva objekty patřily téže třídě, musí splňovat tyto podmínky:

1. musí obsahovat stejnou sadu slotů, tedy stejný počet slotů stejných názvů (hodnoty těchto slotů však mohou být různé),
2. musí obsahovat stejné metody.

Definice třídy ve zdrojovém textu programu tyto dva údaje (kromě názvu třídy) uvádí. Třidu lze tedy chápat, jako popis objektu: obsahuje jednak seznam názvů jeho slotů a jednak definici všech jeho metod. Při běhu programu pak třída slouží jako předloha k vytváření nových objektů.

Objekt, který patří třídě, se nazývá její *instancí*. Chceme-li v programu vytvořit objekt, musíme pro něj nejdříve definovat třídu a pak jej vytvořit jako její instanci.

2.2. 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`, kterou lze v kombinaci s operátorem `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.

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 2.2.1 (třída `point`)

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 bez metod, k jejichž definici se dostaneme vzápětí), měli bychom se naučit vytvářet její 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 instanci třídy, jejíž jméno najde ve svém prvním parametru. Všechny sloty nově vytvořeného objektu jsou neinicializované a každý pokus získat jejich hodnotu skončí chybou (později si řekneme, jak se získávají hodnoty slotů a pak to budeme moci vyzkoušet).

Příklad 2.2.2 (vytváření a prohlížení instancí)

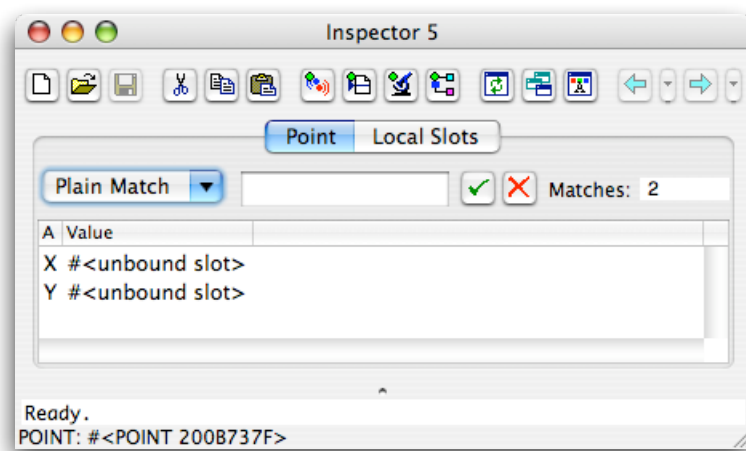
Vytvoření nové instance třídy `point` z předchozího příkladu:

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

Pokud není jasné, proč jsme ve výrazu `(make-instance 'point)` symbol `point` kvotovali, je třeba 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 2.1.

Text `#<unbound slot>` u názvů jednotlivých slotů znamená, že sloty jsou neinicializované. Můžeme jim ale pomocí prostředí LispWorks zkusit nastavit hodnotu.



Obrázek 2.1: Neinicializovaná instance třídy point v inspektoru

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 2.2 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:

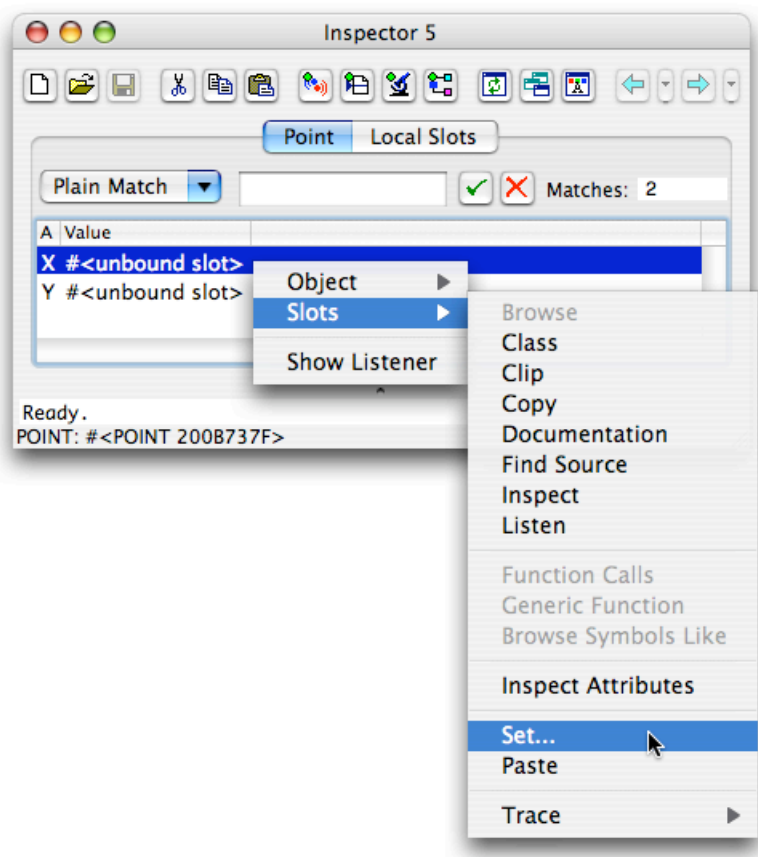
```
(slot-value object slot-name)
```

object: objekt
slot-name: symbol

Příklad 2.2.3 (práce s funkcí `slot-value`)

Vytvořme instanci třídy `point` a uložíme ji do proměnné `pt`:

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

Obrázek 2.2: Nastavování hodnoty slotu v inspektoru

Pozor, 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:

```
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
```

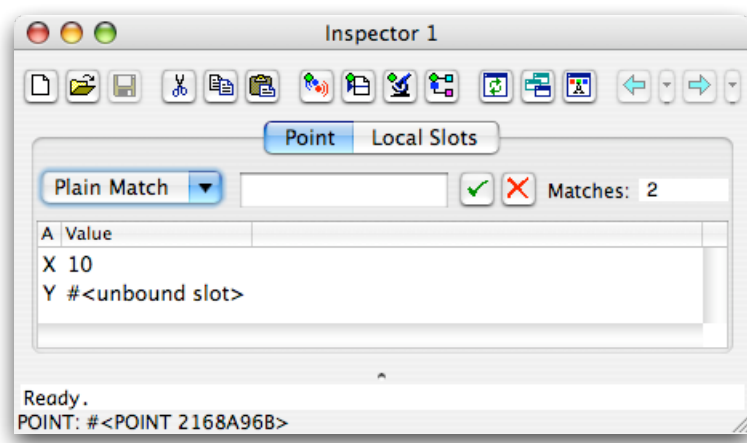
Nové hodnoty slotů lze také ověřit pomocí inspektoru. Získejme nejprve obsah proměnné `pt`:

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

a stiskneme tlačítko s mikroskopem. Výsledek vidíme na Obrázku 2.3.

K práci se sloty ještě dodejme, že používáním funkce `slot-value` ke čtení a nastavování hodnoty slotů vlastně porušujeme výše uvedený princip, že pro komunikaci s objekty se používají výhradně zprávy. K tomuto problému se dostaneme v další kapitole.

Zbývá vysvětlit, jak se v Common Lispu definují metody objektů. Jak jsme již řekli, všechny 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 touto výjimkou, že u prvního parametru je třeba specifikovat jeho třídu. Tím se metoda definuje pro všechny instance této třídy.



Obrázek 2.3: Instance třídy point po změně hodnoty slotu x

```
(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 zasílání 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 2.2.4 (polární souřadnice)

Řekněme, že potřebujeme zjišťovat polární souřadnice bodů. Správný objektový způsob řešení této úlohy je 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)))))
```

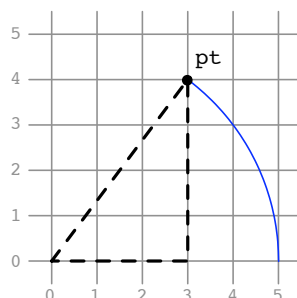
Pokud nechápete, co znamená *(point point)* v této definici, podívejte se znovu na syntax makra *defmethod*. Zjistíte, že první položkou tohoto seznamu je symbol, na nějž bude při vykonávání těla metody navázán příjemce zprávy, zatímco druhou položkou je název třídy, pro jejíž instance metodu definujeme. Že jsou oba tyto symboly 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 2.4.



Obrázek 2.4: Bod o souřadnicích (3, 4)

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
```

To správně, jelikož $\sqrt{3^2 + 4^2} = 5$.

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

```
(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 2.4):

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

Příklad 2.2.5 (nastavování polárních souřadnic)

Definujme ještě 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))
```

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 2.4).

2.3. Inicializace slotů nových instancí

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 2.3.1 (třída `point` s inicializací slotů)

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 2.3.2 (třída `circle`)

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.

V následujícím příkladu ukážeme ještě několik chybových hlášení, se kterými se můžeme ve vývojovém prostředí LispWorks setkat.

Příklad 2.3.3 (chybová hlášení)

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 posílali 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 > (fí (make-instance 'circle))

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

Type :b for backtrace or :c <option number> to proceed.
Type :bug-form "<subject>" for a bug report template or :?
for other options.
```

ÚLOHY KE KAPITOLE 2

2.1. Elipsu v rovině lze zadat pomocí dvou ohnisek a jedné z poloos. Druhou poloosu už lze vždy dopočítat. Definujte třídu `ellipse`, která bude obsahovat sloty `focal-point-1` a `focal-point-2`. Sloty slouží k uchovávání dvou bodů, které jsou ohnisky elipsy. Volitelně můžete napsat metody pro přístup k těmto slotům (v níže uvedeném výpisu takové metody používáme; od příští kapitoly budete psát přístupové metody ke slotům povinně). Dále můžete ve třídě definovat další slot (sloty) na uchovávání potřebných informací o elipse.

2.2. Definujte pro třídu `ellipse` metody `major-semiaxis` a `minor-semiaxis`, které vrátí délku hlavní (to je vždy ta delší) a vedlejší poloosy

2.3. Definujte pro třídu `ellipse` metody `set-major-semiaxis` a `set-minor-semiaxis` pro nastavení délky hlavní a vedlejší poloosy.

2.4. Definujte pro třídu `ellipse` metodu `current-center`, která vrátí novou instanci třídy `point` reprezentující střed elipsy.

2.5. *Výstřednost* elipsy je podíl vzdálenosti jejích ohnisek a délky hlavní osy (tj. dvojnásobku délky hlavní poloosy). Definujte pro třídu `ellipse` metodu `eccentricity`, která vrátí výstřednost elipsy.

Dosud definovaný kód by mělo jít otestovat takto:

```
CL-USER 1 > (setf e (make-instance 'ellipse))
#<ELLIPSE 200B809F>

CL-USER 2 > (setf (slot-value (focal-point-1 e) 'x) -1)
-1

CL-USER 3 > (setf (slot-value (focal-point-2 e) 'x) 1)
1

CL-USER 4 > (set-major-semiaxis e 2)
#<ELLIPSE 200B809F>

CL-USER 5 > (minor-semiaxis e)
1.7320508

CL-USER 6 > (* * *)
3.0

CL-USER 7 > (current-center e)
#<POINT 21CD10A3>

CL-USER 8 > (slot-value * 'x)
0

CL-USER 9 > (slot-value ** 'y)
0

CL-USER 10 > (eccentricity e)
0.5

CL-USER 11 > (set-minor-semiaxis e 1)
#<ELLIPSE 200B809F>

CL-USER 12 > (major-semiaxis e)
1.4142135

CL-USER 13 > (* * *)
1.9999999
```

2.6. Napište metodu `to-ellipse` třídy `circle`, která vrátí elipsu stejného tvaru, jako kruh, jenž je příjmcem zprávy.

Kapitola 3

Zapouzdření a polymorfismus

3.1. Princip zapouzdření

Začneme několika příklady.

Příklad 3.1.1 (problém s nastavováním slotu)

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`. (Než budete číst dál, zkuste odhadnout, co přesně se stane a proč.)

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

Příklad 3.1.2 (pátrání po chybě)

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 symbolem `nil`.

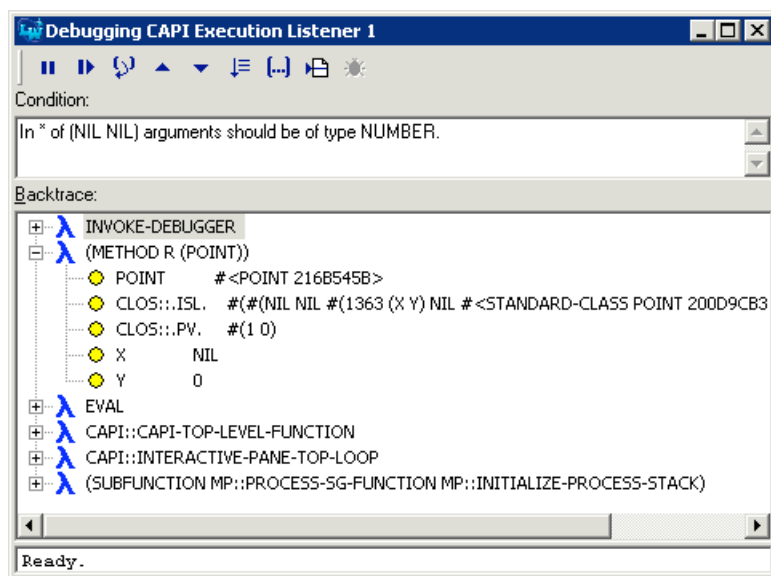
Nyní máme několik možností, co ve vzniklém chybovém stavu dělat. Především se z něj můžeme dostat napsáním :a. Tím se chybový stav ukončí a výkon programu definitivně zastaví. To by ale byla škoda, protože bychom opustili prostředí, které je nachystáno nám pomoci chybu v programu nalézt.

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. Máme totiž stále k dispozici zásobník pozastaveného procesu s informacemi o volaných funkcích, jejich argumentech a lokálních proměnných. Můžeme si jej prohlédnout v debuggeru (ladiči) po stisknutí tlačítka s beruškou () . Okno debuggeru je na Obrázku 3.1.

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 textu metody) opět s jejich hodnotami. Další lokální proměnné (v tomto případě

Obrázek 3.1: Výpis zásobníku v debuggeru s detailem metody `r`

`CLOS::ISL.` a `CLOS::PV.`) přidal do metody systém a my si jich nemusíme všimnout.

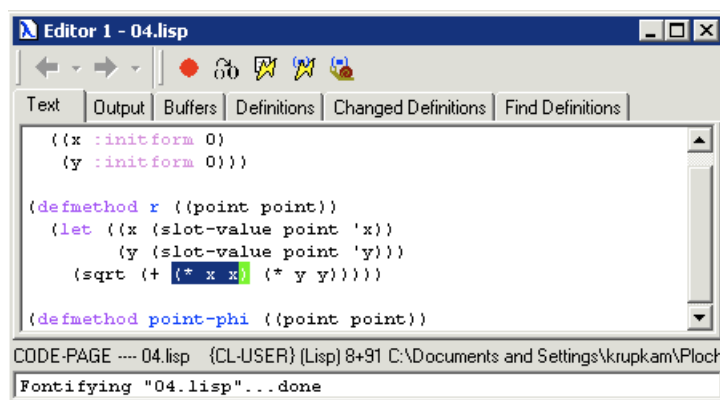
Vidíme, že lokální proměnná `x` má hodnotu `nil`, což způsobilo chybu. Také si můžeme všimnout, že hodnotou parametru `point` je nějaká instance třídy `point`. Pokud se chceme podívat podrobněji jaká, můžeme na řádek s parametrem `point` zaklikat a zobrazit si ji v inspektoru (ten už známe, proto tady 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 3.2.

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

Příklad 3.1.3 (polární souřadnice problém nemají)

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.

Obrázek 3.2: Místo v metodě `r`, kde došlo k chybě

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. 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 byly upozorněni na problém v momentě, kdy jsme jej způsobili, v předchozím proběhlo nastavení

nepřípustné hodnoty bez povšimnutí a upozornění jsme byli až na druhotnou chybu — tedy chybu způsobenou předchozí chybou.

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.

Poznamenejme, že ve 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` má smysl jako hodnoty slotu `radius` nastavovat pouze nezáporná čísla. Pokus uložit do tohoto slotu v průběhu programu vypočítané záporné číslo ale žádný kompilátor neodhalí.

Příklad 3.1.4 (problém se změnou implementace)

Předpokládejme, že změníme reprezentaci geometrických bodů tak, že místo kartézských souřadnic budeme ukládat souřadnice polární. Nová 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 dost velký) zdrojový kód programu a všude, kde to bude potřeba, provést příslušnou úpravu.

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

Příklad 3.1.5 (jednoduchost rozhraní)

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 mnohonásobily 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.

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. Uživatele nezajímají implementační detaily.

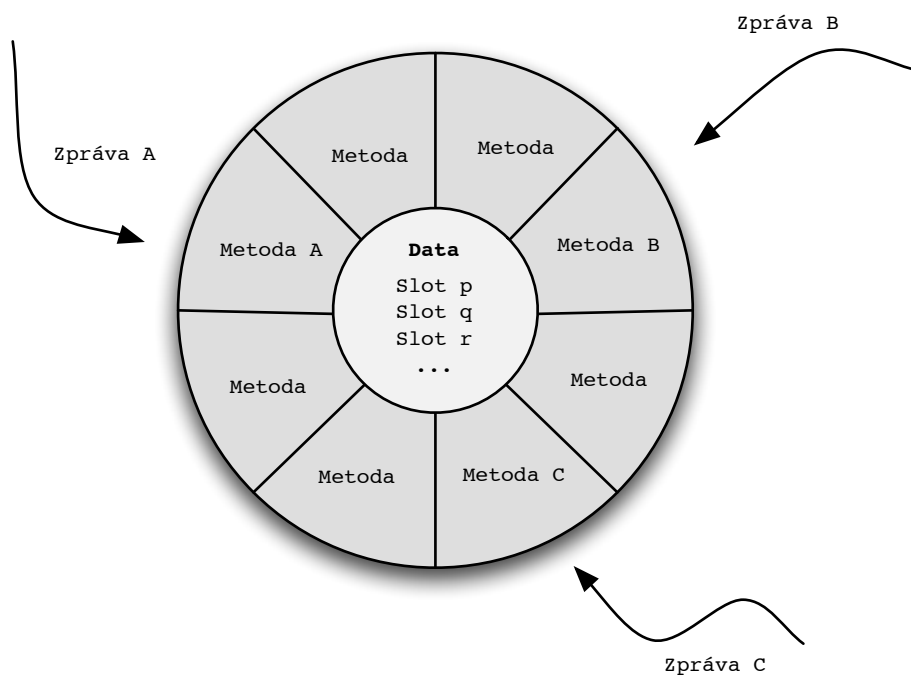
Jedná se o konkrétní realizaci obecnějšího principu, který je třeba dodržovat ve všech programovacích stylech, jakmile je program rozdělen na moduly, nikoliv pouze v objektovém programování. Tento princip bývá obecně nazýván principem *oddělení rozhraní a implementace* (v jiných souvislostech se také 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 3.3.

Nepřímý přístup k hodnotám slotů přes metody umožňuje kontrolovat správnost nastavované hodnoty, čímž se zabrání problémům ukázaným v Příkladě 3.1.1. Ukážeme si to v následujících příkladech. V dalších zdrojových textech už nebudeme kontrolu dělat důsledně, abychom je příliš nekomplikovali.

3.2. Úprava tříd *point a circle*

Upravme tedy definice našich tříd *point a circle*, aby byly v souladu s principem zapouzdření.



Obrázek 3.3: Objekt

Příklad 3.2.1 (třída `point` s přístupovými metodami)

Nová definice třídy `point` (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 3.2.2 (třída *circle* s přístupovými metodami)

Nová definice třídy *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* nedovolíme nastavování hodnoty a dáme mu pouze možnost čtení:

```
(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
```

3.3. Třída `picture`

Další využití zapouzďení ukážeme na příkladě třídy `picture`. Třída umožňuje spojit několik grafických objektů do jednoho a usnadnit tak operace prováděné na všech těchto objektech současně.

Příklad 3.3.1 (třída `picture` s typovou ochranou)

Instance třídy `picture` budou obsahovat seznam podřízených grafických objektů:

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

Metody ke čtení a nastavování hodnoty slotu `items`:

```
(defmethod items ((pic picture))  
  (slot-value pic 'items))  
  
(defmethod check-item ((pic picture) item)  
  (unless (or (typep item 'point)  
              (typep item 'circle)  
              (typep item 'picture))  
    (error "Invalid picture element type."))  
  pic)  
  
(defmethod check-items ((pic picture) items)  
  (dolist (item items)  
    (check-item pic item))  
  pic)  
  
(defmethod set-items ((pic picture) value)  
  (check-items pic value)  
  (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 makro `dolist` v metodě `check-items` — 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: Invalid picture element 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.

Snažíme se dodržovat osvědčenou programátorskou zásadu nepsat neúměrně dlouhé funkce (v našem případě metody). Proto jsme ucelenou část funkčnosti metody `set-items` izolovali a přemístili do jiné metody (se kterou jsme udělali totéž). Z tohoto kroku budeme těžit už v této kapitole a v příští kapitole se nám na nečekaném místě také vyplatí.

Všimněte si, že metodu `check-item` bude třeba přepracovat kdykoliv definujeme novou třídu grafických objektů. To je jistě nešikovné. Nápravu sjednáme později.

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. Jak? To si ukážeme v dalším příkladě.

Příklad 3.3.2 (problém třídy `picture`)

Pokračujme tedy s testováním třídy `picture`. 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>)
```

(Jste schopni vysvětlit, co se stalo? Než budete pokračovat dále, je to třeba pochopit.)
Obrázek `pic` nyní obsahuje nekonzistentní data.

Příklad 3.3.3 (úprava třídy `picture`)

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 zařídíme, 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)
  (check-items pic value)
  (setf (slot-value pic 'items) (copy-list value))
  pic)
```


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`.

3.4. Vlastnosti

V předchozích příkladech jsme uvedli několik zpráv sloužících ke čtení a zápisu dat objektů. V souladu s principem zapouzdření tyto zprávy nezávisí na tom, jakým způsobem jsou data v objektech uložena (např. souřadnice x a y u bodů jsou uložena ve slotech, zatímco souřadnice r a ϕ nikoliv).

Datům objektů, která můžeme zasíláním zpráv číst a nastavovat, říkáme *vlastnosti objektů* (anglicky *properties*, případně *attributes*). Každá vlastnost má své jméno, od kterého jsou odvozena jména příslušných zpráv. Zpráva pro čtení vlastnosti jménem *property* se jmenuje *property*, obvykle nemá žádný parametr a v reakci na ni by objekt měl vrátit hodnotu této vlastnosti. Jméno zprávy pro nastavení této vlastnosti je *set-property*. Tato zpráva má obvykle jeden parametr — novou hodnotu vlastnosti. Vrací modifikovaný objekt.

Některé vlastnosti není zvenčí povoleno měnit. U těchto vlastností není k dispozici zpráva pro jejich nastavování. Objekt samotný takovouto vlastnost měnit může, obvykle tak činí pomocí `slot-value`.

Příklad 3.4.1

Pro třídy `point`, `circle` a `picture` jsme zatím definovali následující vlastnosti: `x`, `y`, `r`, `phi` pro třídu `point`; `center` a `radius` pro třídu `circle` (první je jen ke čtení) a `items` pro třídu `picture`.

3.5. Kreslení pomocí knihovny `micro-graphics`

K vykreslování našich grafických objektů budeme využívat jednoduchou procedurální grafickou knihovnu `micro-graphics`, napsanou pro potřeby tohoto textu. V této kapitole budeme používat jen některé její základní funkce: `mg:display-window`, `mg:get-param`, `mg:set-param`, `mg:clear`, `mg:draw-circle` a `mg:draw-polygon`. Popis těchto funkcí spolu s popisem ostatních funkcí rozhraní knihovny najdete v Příloze C.

Příklad 3.5.1 (použití knihovny `micro-graphics`)

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).

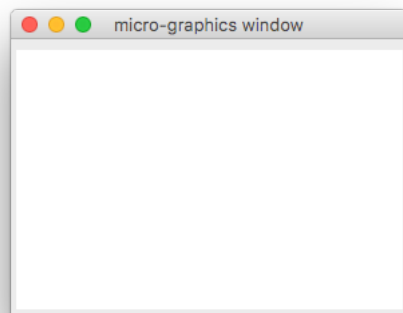
Nejprve zavoláme funkci `mg:display-window` a výsledek uložíme do proměnné:

```
CL-USER 40 > (setf w (mg:display-window))
#<MG-WINDOW 20099673>
```

Otevře se nově vytvořené okno knihovny `micro-graphics`, jak vidíme na Obrázku 3.4.

Výsledek tohoto volání (v našem případě zapisovaný prostředím jako `#<MG-WINDOW 20099673>`) 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 okna:

Obrázek 3.4: Prázdné okno knihovny *micro-graphics*

```
CL-USER 41 > (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))
```

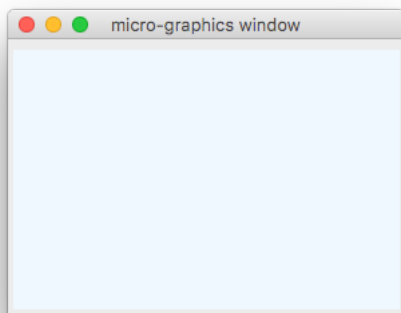
Kreslicí parametry můžeme nastavit funkcí `mg:set-param`. Zkusme tedy změnit barvu pozadí a potom pomocí funkce `mg:clear` okno vymazat:

```
CL-USER 42 > (mg:set-param w :background :aliceblue)
NIL

CL-USER 43 > (mg:clear w)
NIL
```

Pokud jsme to udělali dobře, pozadí okna se přebarví na světle modrou (Obrázek 3.5).

Poznámka: v Common Lispu se symboly začínající dvojtečkou vyhodnocují na sebe. Proto vyhodnocení předchozího výrazu neskončilo chybou. Můžeme si to vyzkoušet konkrétně:



Obrázek 3.5: Okno knihovny micro-graphics po změně barvy pozadí

```
CL-USER 1 > :background  
:BACKGROUND  
  
CL-USER 2 > :aliceblue  
:ALICEBLUE
```

Symbolům začínajícím dvojtečkou říkáme *klíče*.

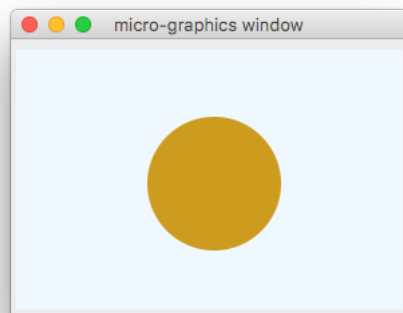
Jako další krok otestujeme funkci `mg:draw-circle`. Víme, že tato funkce vyžaduje jako parametry údaje o středu a poloměru vykreslovaného kruhu. Kromě toho její výsledek ovlivňují kreslicí parametry `:foreground`, `:thickness` a `:filledp`. Nastavme tedy nejprve například parametry `:foreground` a `:filledp`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

```
CL-USER 44 > (progn  
               (mg:set-param w :foreground :goldenrod3)  
               (mg:set-param w :filledp t)  
               (mg:draw-circle w 148 100 50))  
NIL
```

V okně se objeví vyplněný kruh barvy `:goldenrod3` (Obrázek 3.6).

Nyní ještě změníme parametr `:foreground` a zkusíme pomocí funkce `mg:draw-polygon` nakreslit čtverec:

```
CL-USER 45 > (progn  
               (mg:set-param w :foreground :aliceblue)
```

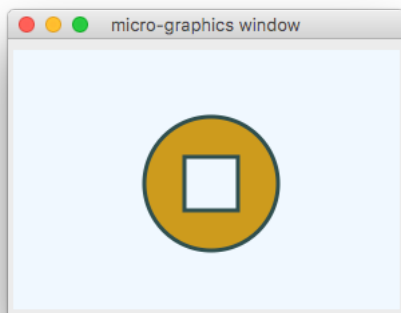
Obrázek 3.6: Okno knihovny *micro-graphics* po nakreslení kruhu

```
(mg:draw-polygon w '(128 80 168 80  
                    168 120 128 120)))  
NIL
```

Konečně, pro vylepšení efektu nastavíme novou barvu pera, parametry `:filledp`, `:thickness` a (kvůli uzavření čtverce) `:closedp` a znovu vykreslíme kruh a čtverec:

```
CL-USER 46 > (progn  
              (mg:set-param w :foreground :darkslategrey)  
              (mg:set-param w :thickness 3)  
              (mg:set-param w :filledp nil)  
              (mg:set-param w :closedp t)  
              (mg:draw-circle w 148 100 50)  
              (mg:draw-polygon w '(128 80 168 80  
                                  168 120 128 120)))  
NIL
```

Výslednou podobu okna vidíme na Obrázku 3.7.



Obrázek 3.7: Okno knihovny `micro-graphics` po nakreslení dvou kruhů a dvou polygonů

3.6. 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` není objektová. Výsledek kreslení je vždy závislý na hodnotách, které je nutno předem nastavit. V objektovém přístupu se snažíme dodržovat princip nezávislosti. Proto požadujeme, aby výsledky akcí prováděných s objekty byly pokud možno závislé pouze na vnitřním stavu objektů (a hodnotách parametrů zpráv objektům zasílaných). Proto budou informace o způsobu kreslení (barva, tloušťka pera a podobně) součástí vnitřního stavu objektů, stejně jako informace o okně, do něž se objekty mají kreslit.

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 3.6.1 (třída `window`)

Uvedme nejprve definici třídy `window`, jejíž instance budou obsahovat informace

o okně knihovny `micro-graphics`, do něhož lze kreslit naše grafické objekty (vlastnost `mg-window`), a další údaje. Mezi ně patří:

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

Definice třídy:

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

Je vidět, že při vytvoření instance této třídy se automaticky otevře nové okno.

Definice metod pro jednotlivé vlastnosti:

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

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

(defmethod set-shape ((w window) shape)
  (when 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)
```

Vlastnost `mg-window` je pouze ke čtení. Dále, při nastavování vlastnosti `shape` oknu posíláme objektu `shape` zprávu `set-window`. (Pokud místo něj nemáme `nil`, což by znamenalo, že grafický objekt chceme z okna jen vypustit.) Tuto zprávu jsme zatím nedefinovali, dostaneme se k ní později.

Chceme-li vykreslit obsah okna, pošleme mu zprávu `redraw`. Metoda `redraw` vykreslí obsah okna tak, že nejprve zjistí barvu pozadí (vlastnost `background`), touto barvou obsah okna vymaže (funkcí `mg:clear`) a nakonec pošle grafickému objektu ve slotu `shape` zprávu `draw`:

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

Zpráva draw má následující syntax:

```
(draw object)
```

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 3.6.2 (rozšíření třídy `circle`)

Nyní implementujeme kreslení pro třídu `circle`. Už víme, že je třeba definovat metodu pro zprávu `draw`. Po jejím obdržení by se měl kruh vykreslit. To bude vyžadovat přidání vlastností 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)
```

Význam vlastností `color`, `thickness` a `filledp` je jasný. Budeme je používat pro určování vzhledu vykresleného kruhu. Vlastnost `window` bude obsahovat instanci třídy `window` a bude nastavována v její metodě `set-shape`. 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))))
```

Příklad 3.6.3 (kreslení ve třídě `circle`)

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. (Podle obecného principu: rozdělit složitější úkol na více jednodušších. Přestože to teď netušíme, rozhodnutí vedené tímto obecným principem se nám v budoucnu vyplatí.) 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. Zde je metoda pro tuto zprávu:

```
(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))
```

Příklad 3.6.4 (test kreslení koleček)

Test kódu z předchozích příkladů:

```
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>

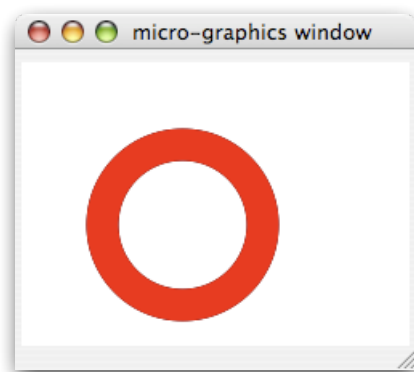
CL-USER 6 > (set-color circ :red)
#<CIRCLE 218359FB>

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>
```

Výsledek by měl odpovídat Obrázku 3.8. Kruh v okně můžeme kdykoli překreslit



Obrázek 3.8: Červené kolečko

zavoláním

```
(redraw w)
```

Příklad 3.6.5 (kreslení ve třídě `picture`)

Budeme pokračovat kreslením obrázků, tedy instancí třídy `picture`. Jak už víme, je třeba definovat metodu pro zprávu `draw`.

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

Metoda prochází všechny prvky obrázku `pic` od posledního k prvnímu (díky funkci `reverse`) a každému posílá 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.

Podobrázky ovšem zatím neví, do kterého okna se mají vykreslovat! To napravíme následující změnou třídy `picture`:

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

(defmethod window ((pic picture))
  (slot-value pic 'window))

(defmethod set-window ((pic picture) value)
  (dolist (item (items pic))
    (set-window item value))
  (setf (slot-value pic 'window) value)
  pic)
```

Tím jsme třídě `picture` definovali vlastnost `window`. Při jejím nastavování (v metodě `set-window`) se nastaví vlastnost `window` i u všech podobrázků.

To ale stále není všechno, co je třeba pro bezproblémové kreslení obrázků udělat. Je třeba myslet i na správné nastavení okna podobrázkům, které se do obrázku dostanou později, až když už je obrázek v okně umístěn. Proto upravíme metodu `set-items`.

```
(defmethod set-items ((pic picture) value)
  (check-items pic value)
  (setf (slot-value pic 'items) (copy-list value))
  (set-window pic (window pic))
  pic)
```

Tím, že obrázek sám sobě pošle zprávu `set-window`, se správné nastavení vlastnosti `window` všem novým podobrázkům zajistí.

Příklad 3.6.6 (test kreslení obrázků)

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)
  (let ((result (make-instance 'picture))
        (items '())
        (step (/ radius count))
        (blackp t)
        (circle))
    (dotimes (i count)
      (setf circle (set-filledp
                     (set-color
                      (set-radius (make-instance 'circle)
                                   (- radius (* i step)))
                      (if blackp :black :light-blue))
                     t))
      (set-y (set-x (center circle) x) y)
      (setf items (cons circle items)
              blackp (not blackp)))
    (set-items result 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:

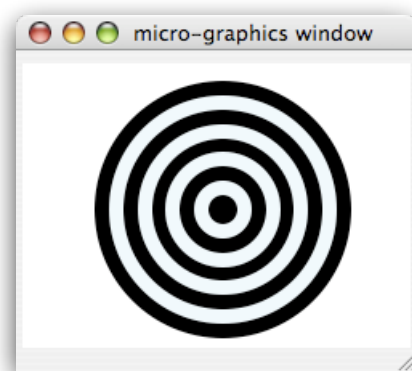
```
CL-USER 17 > (setf w (make-instance 'window))
#<WINDOW 21C94ADF>

CL-USER 18 > (setf eye (make-bulls-eye 125 90 80 9))
#<PICTURE 200E038F>

CL-USER 19 > (set-shape w eye)
#<WINDOW 21C94ADF>

CL-USER 20 > (redraw w)
#<WINDOW 21C94ADF>
```

Výsledné okno je na Obrázku 3.9. Kruhy v obrázku jsou vyplněné (mají nastaveno



Obrázek 3.9: Terč

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

Příklad 3.6.7 (druhý test kreslení obrázků)

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

```
CL-USER 26 > (setf w (make-instance 'window))
#<WINDOW 200E007F>

CL-USER 27 > (setf eye1 (make-bulls-eye 84 105 40 5))
#<PICTURE 200D248B>

CL-USER 28 > (setf eye2 (make-bulls-eye 213 105 40 5))
#<PICTURE 200BC66B>

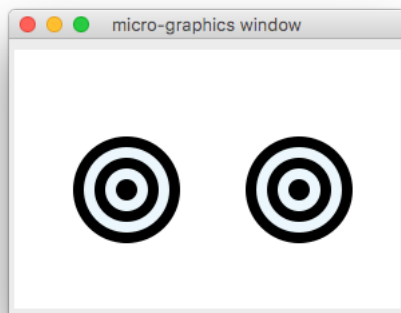
CL-USER 29 > (setf pic (make-instance 'picture))
#<PICTURE 200A1D4B>

CL-USER 30 > (set-items pic (list eye1 eye2))
#<PICTURE 200A1D4B>

CL-USER 31 > (set-shape w pic)
#<WINDOW 200E007F>
```

```
CL-USER 32 > (redraw w)
#<WINDOW 200E007F>
```

A výsledek je na Obrázku 3.10.



Obrázek 3.10: Dva terče vedle sebe

Příklady k této části obsahují vylepšenou funkci `make-bulls-eye`.

Příklad 3.6.8 (kreslení bodů)

Body kreslíme jako malá kolečka. Podrobnosti najdete ve zdrojovém kódu k této části.

3.7. Princip polymorfismu

Aniž bychom o tom hovořili, využili jsme v předchozích 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 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` v těchto třídách 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.

Příklad 3.7.1 (kreslení obrázků bez polymorfismu?)

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 'polygon) (polygon-draw item))
          ((typep item 'picture) (picture-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.

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.

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.

3.8. Polygony

Příklad 3.8.1 (třída `polygon`)

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, nebo plocha lomenou čarou omezená. Obdélníky, čtverce i trojúhelníky jsou polygony.

Základní definice třídy `polygon` je tedy velmi podobná definici třídy `picture`. Kromě vlastnosti `items` ovšem podobně jako u třídy `circle` definujeme další vlastnosti, 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 check-item ((poly polygon) item)
  (unless (typep item 'point)
    (error "Items of polygon should be points. "))
  poly)

(defmethod check-items ((poly polygon) items)
  (dolist (item items)
    (check-item poly item))
  poly)

(defmethod set-items ((poly polygon) value)
  (check-items poly value)
  (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)
        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 3.8.2 (zkouška polygonu)

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>

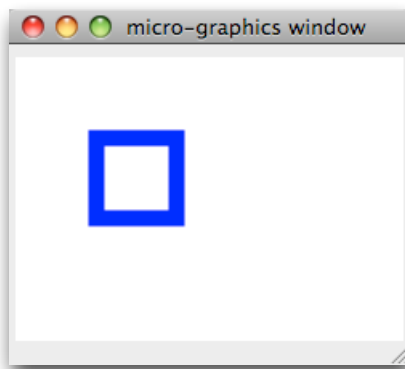
CL-USER 3 > (set-items
              p
              (list (move (make-instance 'point) 50 50)
                    (move (make-instance 'point) 100 50)
                    (move (make-instance 'point) 100 100)
                    (move (make-instance 'point) 50 100))))
#<POLYGON 216FBD3B>

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 3.11.



Obrázek 3.11: Modrý čtverec

Příklad 3.8.3 (úprava třídy `picture`)

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 `check-item` třídy `picture`:

```
(defmethod check-item ((pic picture) item)
  (unless (or (typep item 'point)
              (typep item 'circle)
              (typep item 'polygon)
              (typep item 'picture))
    (error "Invalid picture element type. "))
  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 totiž vyřešíme pomocí principu dědičnosti.

3.9. Geometrické transformace

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 měřítko.

Příklad 3.9.1 (posunutí)

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 (metoda třídy `polygon` je stejná jako u třídy `picture`):

```
(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 (item (items pic))
    (move item 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 3.9.2 (otočení)

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 (stejně u polygonů):

```
(defmethod rotate ((c circle) angle center)  
  (rotate (center c) angle center)  
  c)  
  
(defmethod rotate ((pic picture) angle center)  
  (dolist (item (items pic))  
    (rotate item angle center))  
  pic)
```

Příklad 3.9.3 (změna měřítka)

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

ÚLOHY KE KAPITOLE 3

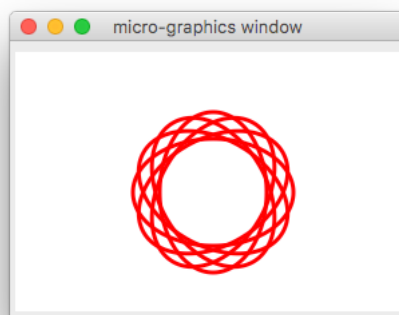
Hlavním cílem těchto úloh je doplnit třídu `ellipse` z minulé kapitoly, aby se dala používat stejně jako třídy definované v této kapitole.

3.1. Přesvědčte se, že definice třídy `ellipse` splňuje podmínku principu zapouzdření. Pokud ne, upravte ji. Třída by měla mít čtyři vlastnosti: `focal-point-1`, `focal-point-2`, `major-semiaxis` a `minor-semiaxis`. První dvě by měly být jen ke čtení, druhé dvě i k zápisu. Dodejte také testy správnosti zapisovaných hodnot.

3.2. Upravte také metodu `to-ellipse` třídy `circle`, aby vyhovovala principu zapouzdření.

3.3. Podobně jako v Příkladu 3.6.2 doplňte do třídy `ellipse` vlastnosti potřebné ke kreslení.

- 3.4. Podobně jako v Příkladu 3.6.3 definujte ve třídě `ellipse` metody pro kreslení a otestujte je.
- 3.5. Definujte metody `move`, `rotate` a `scale` pro třídu `ellipse`.
- 3.6. Napište funkci podobnou funkci z Příkladu 3.6.6 pro elipsy. Ke změně velikosti elipsy ale použijte zprávu `scale` a k posunutí do bodu `x y` zprávu `move`.
- 3.7. Napište funkci, která vrátí instanci třídy `picture`, která se vykreslí zhruba jako na Obrázku 3.12. K otočení elipsy použijte zprávu `rotate`. V případě potřeby třídu



Obrázek 3.12: Několik elips

`picture` upravte.

- 3.8. Definujte třídy `full-shape` a `empty-shape`. Instance třídy `full-shape` by měly představovat geometrické objekty, které vyplní celou rovinu. U třídy `empty-shape` to budou naopak objekty neobsahující žádný bod. Pro tyto třídy definujte všechny metody, které jsme v této kapitole definovali pro ostatní třídy, pokud to má smysl. Má například smysl definovat metodu pro třídu `empty-shape`? Jak tuto metodu definovat pro třídu `full-shape`?

Kapitola 4

Dědičnost

4.1. Princip dědičnosti a pravidlo is-a

V implementaci grafických objektů v předchozí kapitole se často na různých místech opakuje stejný kód. To porušuje základní programátorskou zásadu: pokud možno nikdy nenechávat v zdrojovém textu na více místech stejný nebo podobný kód. Dodržování této zásady má dobré důvody. Snižuje možnost vnesení chyb do kódu při jeho úpravách a vůbec přispívá k udržitelnosti a rozšiřitelnosti programu do budoucna.

Příklad 4.1.1 (příčina opakovaného kódu)

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

Podívejme se například na vlastnost `window` a související metody v dosud definovaných třídách. Vlastnost `window` má u všech grafických objektů, bez ohledu na jejich třídu, stejný účel: uchovávat odkaz na okno, do kterého se bude objekt vykreslovat. Zprávy, týkající se vlastnosti, jsou tři. Připomeňme implementaci příslušných metod 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` slouží (jako obvykle) k práci s vlastností `window`, 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`.

Podobně je to s dalšími zprávami; například metody zpráv `color`, `set-color`, `thickness`, `set-thickness`, `filledp`, `set-filledp` jsou u většiny tříd grafických objektů implementovány stejně — čtou nebo nastavují hodnotu příslušného slotu. Metoda `draw` je také u většiny tříd grafických objektů (kromě třídy `picture`) stejná, i když nepracuje se slotem, ale dělá něco méně triviálního. To se týká i výše uvedené metody `shape-mg-window`.

Všechny uvedené zprávy pracují se společnými rysy grafických objektů, zatímco další zprávy (například `x` a `r` u třídy `point` nebo třeba `radius` u třídy `circle`) pracují s vlastnostmi objektů, které jsou pro jednotlivé třídy specifické.

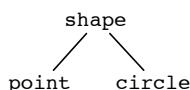
Fakt, že grafické objekty různých tříd mají některé rysy společné, není nijak překvapivý. Jde přece stále o grafické objekty, byť různých typů. V *realitě, kterou modelujeme, jsou body, kružnice, obrázky a polygony všechno grafické objekty*. Nevýhodou našeho řešení v minulé kapitole je, že na tuto skutečnost nebere ohled a tváří se, že instance různých tříd nemají nic společného.

Předměty reálného světa lze někdy účelně roztřídit podle jejich typu do skupin tak, že dvě různé skupiny jsou buď disjunktní (nemají žádný společný prvek), nebo je jedna úplně obsažena ve druhé. Takové třídění je v objektovém programování modelováno *principem dědičnosti*, ve kterém dvě různé třídy buď nemají žádnou společnou instanci, nebo je každá instance jedné z nich i instancí druhé.

Technicky řečeno, třídy lze uspořádat do stromové hierarchie, zvané *strom dědičnosti*, ve které instance níže ležících tříd jsou vždy také instancemi tříd ležících výše. Všechny instance dané třídy mají všechny sloty a metody v této třídě definované. Přesto se mohou v některých slotech lišit, pokud jsou současně instancemi různých tříd, ležících v hierarchii níže. Pomocí principu dědičnosti postupně vyřešíme výše uvedený problém s opakovaným kódem.

Příklad 4.1.2 (třída `shape`)

V našem případě definujeme obecnou třídu `shape`, jejímiž instancemi budou všechny grafické objekty a která bude definovat rysy společné všem grafickým objektům. U tříd konkrétnějších (například `point` nebo `circle`) pak již tyto společné rysy



Obrázek 4.1: Strom tříd grafických objektů, první verze

nebudeme znovu definovat. Část vznikajícího stromu dědičnosti našich tříd můžeme vidět na Obrázku 4.1.

O vztahu předchůdce–následník ve stromu dědičnosti (tedy o vztahu nadmnožina–podmnožina mezi třídami, pokud je chápeme jako množiny objektů) se hovoří jako o vztahu *předek–potomek* (*ancestor–descendant*, *predecessor–successor*), někdy také *nadtřída–podtřída* (*superclass–subclass*).

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. 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.

Objekty mohou být instancemi více tříd současně. Pokud je totiž objekt instancí nějaké třídy, je zároveň instancí i všech jejích předků. Ke každému objektu ale existuje jediná třída, jíž je objekt *přímou instancí*. Je to vždy ta třída, která byla uvedena při jeho vytváření (v Lispu funkcí `make-instance`). Objekt pak není instancí žádného jejího potomka.

Při návrhu stromu dědičnosti nesmíme zapomínat na jeho účel napodobit přirozený vztah podmnožina–nadmnožina mezi skupinami předmětů modelovanými třídami. Základní princip správně navrženého stromu dědičnosti lze popsat následujícím pravidlem:

Pravidlo is-a

Množina předmětů reálného světa modelovaná třídou je podmnožinou množiny předmětů modelovanou jejím předkem.

Příklad 4.1.3 (pravidlo is-a)

Důvod, proč se uvedenému pravidlu říká pravidlo *is-a* pochopíme na konkrétních příkladech. Každý bod *je* grafickým útvarem (*every point is a shape*), každý kruh *je* grafickým útvarem. Proto strom na Obrázku 4.1 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 podtřídu třídy vozidel a třídu psů jako podtřídu třídy savců.

Vztah výfuk–automobil, neodpovídá pravidlu *is-a*, proto nelze definovat třídu výfuků jako podtřídu třídy automobilů. Koneckonců, má výfuk kola? Totéž platí také třeba pro vztah čtverec–úsečka, úsečka–bod.

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.

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

Příklad 4.1.4 (definice třídy *shape*)

Přepracování návrhu našich tříd začneme třídou *shape*. Co se týká vlastností, které bude vhodné do třídy *shape* přesunout, jistě je vhodné kromě vlastnosti *window* zvážit i vlastnosti: *color*, *thickness*, *filledp*, *items*. O všech se totiž v principu dá říci, že jsou vlastnostmi všech grafických objektů bez ohledu na jejich typ.

Rozeberme si to u vlastnosti *color*. Tato vlastnost slouží k uložení barvy grafických objektů. Jistě má smysl, aby každý grafický objekt nesl informaci o barvě, kterou je vykreslován.

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. Pravidlo *is-a* musí dostat přednost před těmito drobnými výhradami.

Z podobných důvodů jako u informace o barvě přesuneme do třídy *shape* i vlastnosti *thickness* a *filledp*.

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

Metody pro vlastnosti:

```
(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))

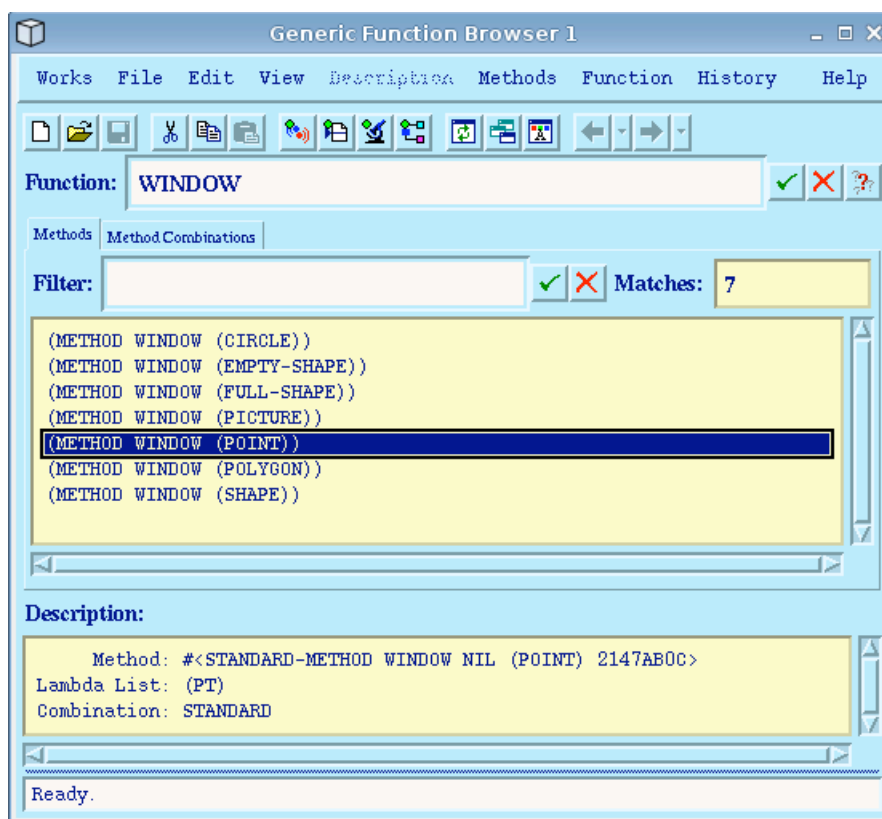
(defmethod set-thickness ((shape shape) value)
  (setf (slot-value shape 'thickness) value)
  shape)

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

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

Příklad 4.1.5

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 4.2).

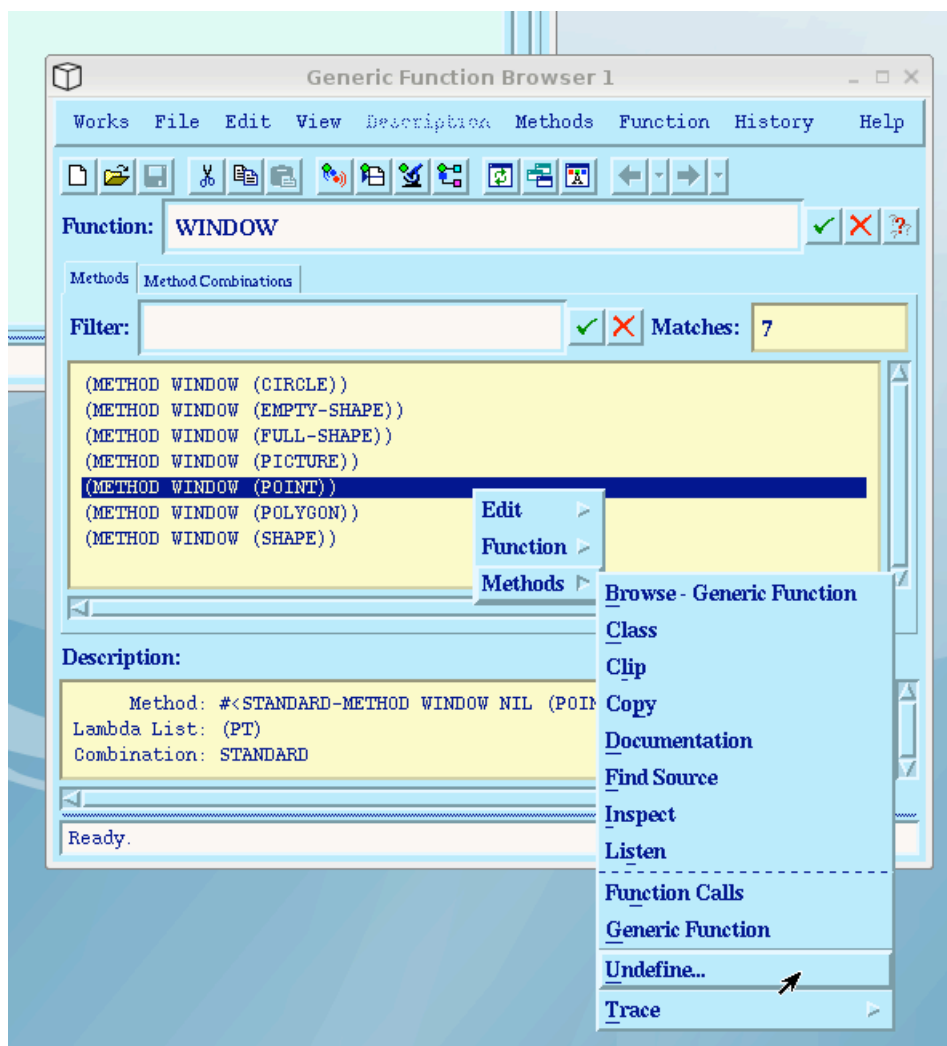


Obrázek 4.2: Prohlížeč generických funkcí

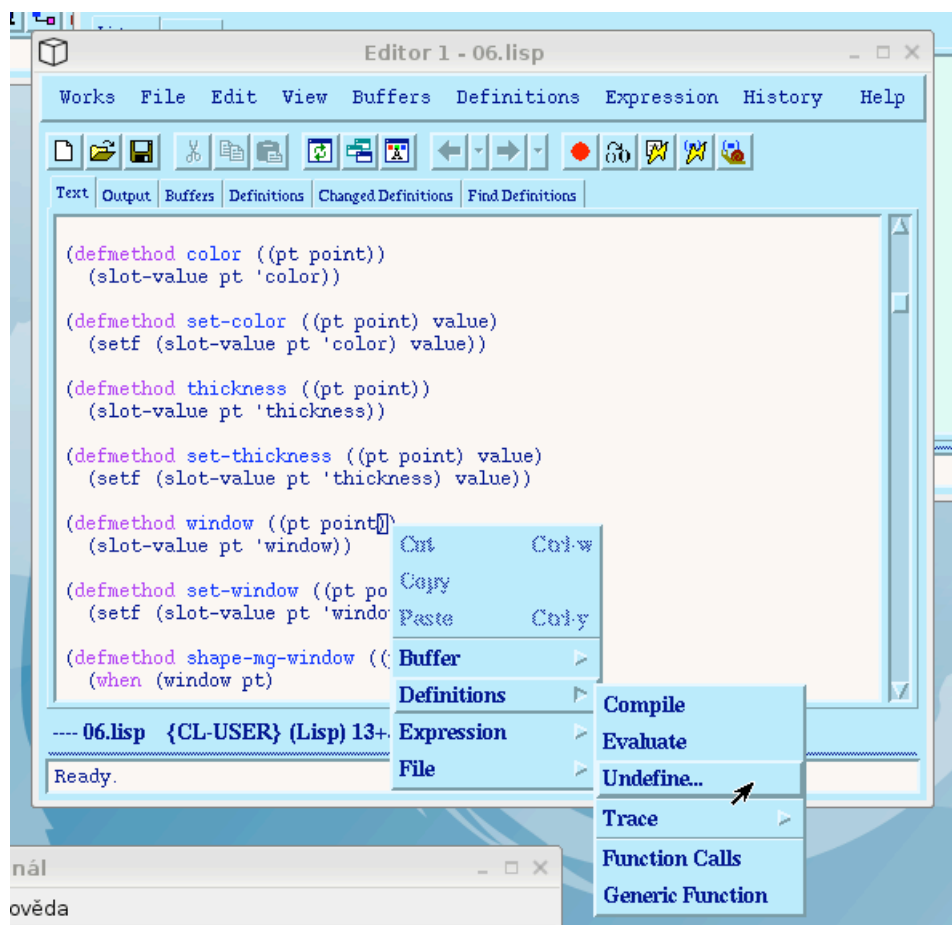
Pomocí tohoto nástroje nyní můžeme odstranit všechny metody, které potřebujeme. Obrázek 4.3 ukazuje, jak to lze udělat například s metodou `window` třídy `point`.

Pokud máme k dispozici zdrojový text metody, jejíž definici chceme zrušit, můžeme také použít jiný způsob: pomocí kontextové nabídky u definice samé (Obrázek 4.4), nebo odkazu na ni v seznamu definic (Obrázek 4.5).

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.



Obrázek 4.3: Odstranění metody v prohlížeči generických funkcí



Obrázek 4.4: Odstranění metody v editoru

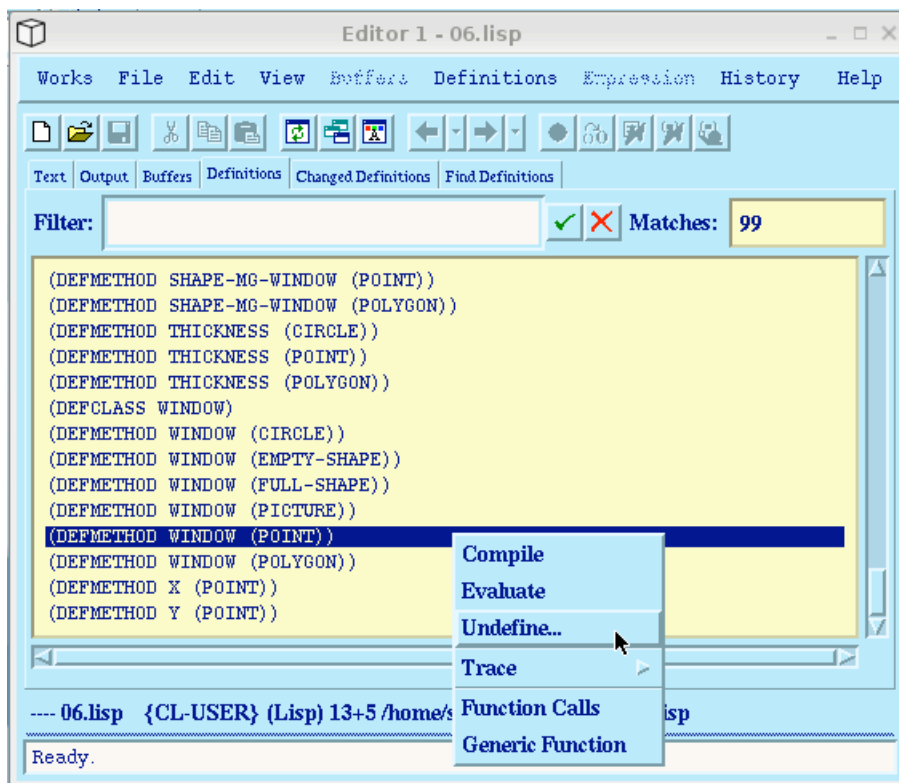
4.2. Určení předka v definici třídy

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 nebo jednoprvkový seznam



Obrázek 4.5: Odstranění metody v seznamu definic

```
slots: seznam
```

V druhém argumentu makra `defclass` (parametr *parents*) lze kromě prázdného seznamu uvést i seznam obsahující jeden symbol. Pokud této možnosti využijeme, musí tento symbol být názvem nějaké již existující třídy. Nově vytvářená třída se pak stane jejím přímým potomkem.

(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.)

Příklad 4.2.1 (třída `point` a `circle`)

Nyní budeme pokračovat předefinováním našich tříd grafických objektů. U vlastností a metod, které najdeme u více tříd, zvážíme přesunutí do třídy `shape`, abychom se vyhnuli opakovanému kódu. Provedené změny budou zpětně kompatibilní. Veškerý kód napsaný pro třídy grafických objektů z předchozí kapitoly bude fungovat i v nové verzi.

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.

Začneme u tříd `point` a `circle`. S těmito třídami nebude mnoho práce. Obě je třeba učinit bezprostředními potomky třídy `shape`. Dále vypustíme definice všech slotů a metod, které jsme do této třídy přesunuli.

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

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

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

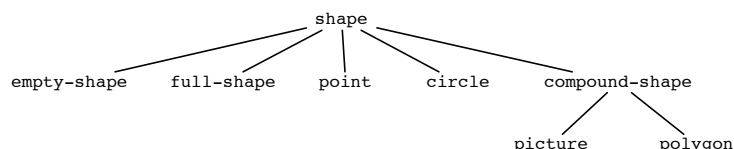
```
(defclass circle (shape)
  ((center :initform (make-instance 'point))
   (radius :initform 1)))
```

Příklad 4.2.2 (třída `compound-shape`)

Co provést s vlastností `items` u tříd `picture` a `polygon`? Tato vlastnost 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 tuto vlastnost 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 4.6 (obrázek obsahuje i třídy `empty-shape` a `full-shape`, o kterých se v textu občas zmiňujeme, ale které jsou především předmětem úloh).



Obrázek 4.6: 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.

Nově zavedená třída `compound-shape` obsahuje vlastnost `items`. Její 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)))
```

Nová definice třídy `picture`:

```
(defclass picture (compound-shape)
  ())
```

4.3. Poznámka o běžných jazycích

Abychom se přiblížili současným základním v praxi používaným objektovým jazykům a osvojili si způsob práce v nich, přijmeme několik omezení, 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, 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++, Java, C#. Toto pravidlo ale nebudeme dodržovat důsledně, občas bude účelné je porušit. Vznikne tím řešení v uvedených jazycích nerealizovatelné.

Metoda specializovaná na předka ostatních tříd nemusí dělat nic užitečného. V některých případech je dokonce vhodné zabránit jejímu volání. V CLOS 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 je nutno v podřízených třídách přepsat a jejichž spuštění vede k chybě.

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 obvykle metody třídy uvádět bezprostředně za její definicí.

4.4. 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, již 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.

Popsaným způsobem postupuje většina objektových jazyků. Existují ale výjimky; například v programovacím jazyce Beta se nehledají metody v hierarchii dědičnosti zespodu 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*.

Příklad 4.4.1 (přepisování metod grafických objektů)

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. 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)
```

4.5. 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.

Příklad 4.5.1

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 ((pic picture) value)
  (dolist (item (items pic))
    (set-window item value))
  (setf (slot-value pic 'window) value)
  pic)
```

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 přepisovanou 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 přepisujeme.

Příklad 4.5.2 (metoda `set-window` v `compound-shape`)

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 ((shape picture) value)
  (send-to-items shape 'set-window value)
  (call-next-method))
```

V tomto případě funkce `call-next-method` zavolá metodu `set-window` třídy `shape`, protože ve třídě `compound-shape` metodu tohoto jména nenalezne.

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.

Příklad 4.5.3 (kreslení ve třídě `shape`)

U většiny tříd jsme postupovali v první verzi kreslení grafických objektů podle stejného vzoru: nejprve jsme metodou `set-mg-params` nastavili potřebné grafické parametry okna knihovny `micro-graphics` a potom jsme objekt metodou `do-draw` vykreslili. 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 přepisovat 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 příslušných vlastností. Tento přístup zbaví některé třídy nutnosti metodu přepisovat:

```
(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)
```

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 určené 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. Toto rozdělení bude v další části textu ještě výraznější.

Metodám, které nemají být explicitně volány, ale jsou určeny pouze k tomu, aby byly v potomcích tříd (případně) přepsány, se v některých jazycích říká *chráněné metody* (*protected methods*).

Příklad 4.5.4 (kreslení u potomků třídy `shape`)

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` třídy `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` (všimněte si volání zděděné metody):

```
(defmethod set-mg-params ((poly polygon))
  (call-next-method)
  (mg:set-param (shape-mg-window poly)
                :closedp
                (closedp poly))
  poly)
```

Metoda `do-draw` je stejná jako dříve.

Kreslení instancí třídy `picture` můžeme nechat beze změny.

Příklad 4.5.5

Teď se podívejme na práci s vlastností `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)))
```

Metoda set-items:

```
(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)
```

Jak víme, úkolem metody `check-items` je otestovat, zda všechny prvky daného seznamu mají typ požadovaný pro prvky daného složeného grafického objektu (pro `polygons` 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 `polygons` 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 "Abstract method."))
```

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 ((poly polygon) item)
  (unless (typep item 'point)
    (error "Items of polygon should be points."))
  poly)
```

```
(defmethod check-item ((pic picture) item)
  (unless (typep item 'shape)
    (error "Invalid picture element type. "))
  pic)
```

4.6. 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. Zpráva `initialize-instance` patří ke zprávám, které objektům nezasíláme, ale pouze přepisujeme její 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 uvést, jinak dojde k chybě. Jeho význam pro nás ale zatím 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 funkci `call-next-method`.

Zprávu `initialize-instance` objektům nezasíláme, ale pouze přepisujeme její metody. V nich vždy voláme funkci `call-next-method`.

Příklady použití metody `initialize-instance` najdete v příkladech k této části textu.

4.7. Návrh stromu dědičnosti

Při návrhu stromu dědičnosti se programátor nachází v nezáviděníhodné pozici. Musí navrhnout strukturu použitelnou pro dnes neznámé účely uživatelem, který nebude mít možnost ji měnit. Jedinou šancí, jak se tohoto úkolu dobře zhostit, je (kromě schopnosti věštit) dodržovat osvědčené programátorské zásady.

Hlavní zásadou, kterou jsme už uvedli, je pravidlo *is-a*. Jeho dodržováním zajistíme, že námi navrhovaná struktura tříd bude (více méně věrně) kopírovat strukturu typů předmětů reálného světa, které se snažíme modelovat. (Ony typy předmětů reálného světa jsou ovšem také něco uměle definovaného; i zde je třeba být obezřetný.)

Příklad 4.7.1 (bod a úsečka)

Uvažme následující definici třídy `segment` (úsečka):

```
(defclass segment (point)
  ((x2 :initform 0)
   (y2 :initform 0)))
```

nesprávně

Vedla nás následující úvaha: Úsečka se skládá ze dvou bodů. Je tedy třeba definovat čtyři sloty, vždy dva pro dvě souřadnice jednoho bodu. Jednu takovou dvojici slotů už máme ve třídě `point`, je tedy třeba třídu rozšířit ještě o další dva. Použijeme dědičnost, sloty `x` a `y` zdědíme z třídy `point` a zbylé definujeme v naší třídě.

Nové sloty budou sloužit k uložení hodnot nových vlastností `x2` a `y2`, pro ně tedy samozřejmě napíšeme přístupové metody. Dále vhodně přepíšeme ostatní metody třídy `point`. Kde to bude účelné, zavoláme zděděnou metodu. Například:

```
(defmethod move ((seg segment) dx dy)
  (call-next-method)
  (set-x2 seg (+ (x2 seg) dx))
  (set-y2 seg (+ (y2 seg) dy))
  seg)
```

nesprávně

Při práci na třídě postupně zjistíme, že musíme přepsat v podstatě všechny metody třídy `point`. To, že jsme třídu `segment` definovali jako jejího potomka, nám nic užitečného nepřineslo, spíše komplikace. A kdyby autoři třídy `point` k ní v budoucnu připsali nové metody, museli bychom neustále naši třídu upravovat. Neustále by hrozilo, že se bude chovat nekorektně.

To platí například pro nové metody `left`, `top`, `right`, `bottom`, které máte za úkol doplnit k našim třídám v úlohách k této kapitole. Snadno zjistíte, že instance

třídy `segment` nebudou na nové zprávy reagovat správně, dokud třídu neupravíte. Tohle jistě není účelem principu dědičnosti.

Příčinu problémů najdeme v nedodržení principu *is-a*. *Úsečka* totiž *není bod*. V reálném světě není množina úseček podmnožinou množiny bodů. Proto není většina předpokladů o instancích třídy `point` splněna pro instance třídy `segment` a my musíme neustále přepisovat existující metody.

I když je pravidlo *is-a* užitečné, je formulováno poněkud neurčitě a v konkrétních situacích nemusí rozptýlit pochybnosti, které při navrhování stromu tříd mohou vzniknout. Závěr učiněný na konci předchozího příkladu nás vede k formulaci dalšího principu (ve formalizované podobě se nazývá principem Liskovové podle významné informatičky Barbary Liskov; my se opět spokojíme s mírně neurčitou formulací).

Substituční pravidlo

Instance třídy musí splňovat všechny podmínky kladené na instance jejího předka.

Neurčitost našeho substitučního pravidla je v tom, že jsme nevymezili přesně, co jsou „podmínky kladené na instance třídy“. Ty závisí na účelu třídy, který by její autor měl vždy dobře specifikovat. Některé programovací jazyky také dávají programátorům do ruky nástroje, které značně omezují možnosti přepisování vlastností a metod tříd.

Příklad 4.7.2 (metoda `left` pro úsečku?)

Jedna z přirozených podmínek kladených na instance třídy `point` se týká metody `left`. Pokud bychom neuvažovali tloušťku pera (vlastnost `thickness`), je zřejmé, že každý bod by měl v reakci na zprávu `left` vrátit hodnotu své x-ové souřadnice. Tuto podmínku ovšem instance naší **nesprávné** třídy `segment` nesplňují, takže třída porušuje substituční pravidlo.

ÚLOHY KE KAPITOLE 4

4.1. Upravte třídu `ellipse` podle poznatků z této kapitoly. Jaký by měl být vztah této třídy a třídy `circle`?

4.2. Totéž udělejte pro třídy `empty-shape` a `full-shape`.

4.3. Dodefinujte třídám grafických objektů nové metody `left`, `top`, `right`, `bottom` tak, aby po obdržení zprávy `left` (`top`, `right`, `bottom`) objekt vrátil souřadnici svého levého (horního, pravého, dolního) okraje. Nemusíte uvažovat tloušťku pera. Podle pravidla, které jsme si zavedli, je třeba metody definovat i pro třídu `shape`. U třídy `ellipse` výpočet vyžaduje určité geometrické znalosti, tuto třídu zatím můžete vynechat.

4.4. Navrhněte třídu `disc`, jejíž instance se budou skládat ze dvou různobarevných plných soustředných kruhů. Instance budou mít vlastnosti `inner-radius`, `outer-radius` (poloměr vnitřního a vnějšího kruhu) a `inner-color` a `outer-color` (barva vnitřního a vnějšího kruhu). Všechny vlastnosti budou jak ke čtení tak k zápisu. Je vhodné také napsat vlastnost `center` (jen ke čtení) obsahující střed kruhů.

Zvažte, zda je vhodné ukládat hodnoty těchto vlastností do slotů definovaných ve třídě `disc`. Nebo je lepší zvolit jiné řešení? Proč?

Třída `disc` by měla být potomkem třídy `picture`. K nastavení vlastnosti `items` na dva kruhy při vytváření nové instance použijte metodu `initialize-instance`. Zvažte také možnost definovat třídu jako potomka třídy `circle`.

4.5. Je nastavení vlastnosti `items` při vytváření instance třídy `disc` v souladu se substitučním pravidlem?

4.6. V ukázkovém souboru `04_bulls-eye.lisp` je definována třída `bulls-eye`, jejíž instance se zobrazují jako hranatý nebo kulatý terč s volitelným poloměrem a počtem pruhů. Upravte tuto třídu tak, aby její instance mohly mít i tvar elipsy (stačí elipsa jednoho konkrétního tvaru, např. s hlavní poloosou dvakrát delší než vedlejší poloosa). Udělejte to tak, že do třídy přidáte vlastnost `shape-type` na přepínání tvaru terče s možnými hodnotami `:circle`, `:square`, `:ellipse`.

4.7. Řekněme, že máte zakázáno měnit definici třídy `bulls-eye`. Bylo by možné vyřešit předchozí příklad vytvořením jejího potomka? Pokud ne, změňte definici třídy tak, aby byla přesně zachována její funkčnost a současně aby vytvořením potomka předchozí příklad vyřešit šlo.

4.8. Když jsme zavrhlí možnost definovat třídu `segment` jako potomka třídy `point`, zvažme ještě možnost opačnou. Dá se přece říci, že bod je speciálním případem úsečky; je to úsečka, jejíž dva krajní body splývají. Nebylo by tedy správné definovat třídu `point` jako potomka třídy `segment`?

4.9. Uvažme třídu `hideable-picture`, jejímiž instancemi by byly obrázky, jejichž některé prvky by bylo možné skrýt, takže by se nevykreslovaly. Třída by byla potomkem třídy `picture`. Která z následujících možností návrhu třídy je lepší a proč? Možnosti prodiskutujte a třídu naprogramujte. Případně navrhněte jiný, vlastní způsob.

1. Třída bude mít novou vlastnost `hide-items`. Ta bude obsahovat seznam stejné délky jako seznam `items`, který bude obsahovat hodnoty `t` nebo `nil`. Prvky

obrázku s příslušnou hodnotou t se budou vykreslovat, ostatní se skryjí. Vlastnost bude nastavitelná uživatelem.

2. Všechny prvky obrázku budou uloženy v nové vlastnosti `all-items`. Obrázek bude mít i vlastnost `hide-items` jako v předchozím případě, ta se ale nyní bude týkat vlastnosti `all-items`. Vlastnost `items` bude obsahovat pouze prvky obrázku, které nejsou skryté.

Kapitola 5

Překreslování oken a hlášení změn

5.1. Zpětná volání v knihovně `micro-graphics`

Při experimentování s okny knihovny `micro-graphics` jsme rychle zjistili, že v nich nakreslené obrázky nikdy dlouho nevydrží. Je to tím, že se zatím neumíme postarat, aby se okno překreslovalo, když je to potřeba. V knihovně `micro-graphics` se takové situace řeší pomocí tzv. *zpětných volání* (*callbacks*). Jedná se o uživatelské funkce, které knihovna zavolá, aby náš program informovala, že došlo k situaci, na kterou program může chtít reagovat. Knihovna rozlišuje několik typů takových situací, například že je třeba překreslit okno, nebo že uživatel do okna kliknul myší či stiskl klávesu, když bylo okno aktivní. Každý z těchto typů má své jméno a pro každý z nich můžeme pro okno zaregistrovat funkci, kterou knihovna zavolá, kdykoli k dané situaci dojde. Informace o různých typech zpětných voláních knihovny `micro-graphics` najdete v Příloze C. V této kapitole se budeme zabývat zpětnými voláními typu `:display`, který se týká funkcí, jež knihovna volá, když je třeba překreslit okno. Knihovna `micro-graphics` tyto funkce volá s jedním argumentem, a to odkazem na příslušné okno.

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.

Příklad 5.1.1 (test zpětného volání :redraw)

Tímto postupem zaregistrujeme do nového okna zpětné volání, které vytiskne daný text, kdykoliv je třeba okno překreslit:

```
CL-USER 3 > (setf mgw (mg:display-window))
#<MG-WINDOW 200CA85B>

CL-USER 4 > (mg:set-callback
             mgw
             :display
             (lambda (mgw)
               (format t "~%Překresli mě!")))
NIL
```

Nyní můžeme manipulovat s oknem a dívat se, co se tiskne do standardního výstupu. Tisk vypneme zrušením zpětného volání:

```
CL-USER 5 > (mg:set-callback mgw :display nil)
NIL
```

5.2. Překreslování oken

Nyní využijeme zpětného volání `:display` k automatickému překreslení okna v naší grafické knihovně.

Příklad 5.2.1

Upravíme třídu `window` tak, aby její instance měly instalovánu jako zpětné volání `:display` funkci, která zajistí překreslení okna kdykoliv o ně knihovna `micro-graphics` požádá. Instalaci provedeme při vytváření okna, tedy v metodě `initialize-instance`. Definice třídy bude stejná jako dříve:

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

Stejně zůstanou i definice metod `mg-window`, `shape` a `background` na zjišťování hodnot vlastností a metoda `redraw` na překreslování. Ostatní metody postupně dopíšeme.

Nyní napíšeme novou metodu `install-callbacks`, která nastaví oknu zpětné volání na překreslení:

```
(defmethod install-callbacks ((w window))
  (mg:set-callback (mg-window w)
                   :display (lambda (mgw)
                              (declare (ignore mgw))
                              (redraw w)))
  w)
```

(Řádek `(declare (ignore mgw))` nemá význam pro běh programu. Pouze potlačí upozornění překladače o nepoužité proměnné `mgw`.)

Tuto úpravu je možné si ihned vyzkoušet. Načtěte si zdrojový text k předchozí kapitole (soubor `04.lisp`) a vyhodnoťte novou definici metody `install-callbacks`. Následujícím kódem vytvoříme nové okno a vložíme do něj jednoduchý útvar (funkci `make-test-circle` si napíšeme bokem, protože ji budeme potřebovat v dalších příkladech):

```
(defun make-test-circle ()
  (move (set-radius (set-thickness (set-color
                                     (make-instance 'circle)
                                     :darkslategrey)
```

```

                    5)
                55)
            148
            100))

CL-USER 7 > (setf w (set-background
                    (set-shape (make-instance 'window)
                               (make-test-circle))
                    :ghostwhite))

#<WINDOW 20109733>

```

Jak už jsme zvyklí, okno se nevykreslí, a pokud ho vykreslíme ručně zasláním zprávy `redraw`, obrázek po manipulaci s oknem brzy zmizí. Pokud ale do okna nainstalujeme zpětné volání na vykreslení:

```

CL-USER 8 > (install-callbacks w)

#<WINDOW 20109733>

```

bude se už překreslovat automaticky. (Všimněme si, jak jsme hezky využili dynamičnosti Lispu.)

Instalaci zpětného volání nyní zautomatizujeme zavoláním metody `install-callbacks` při vytváření nové instance:

```

(defmethod initialize-instance ((w window) &key)
  (call-next-method)
  (install-callbacks w)
  w)

```

Nyní se již budou všechna nově vytvořená okna sama překreslovat. S novou definicí třídy `window` by měly fungovat i všechny dříve napsané příklady.

5.3. Překreslení při změně okna

V dalších příkladech zdokonalíme třídu `window` tak, aby se okna překreslovala i po změně barvy pozadí a nastavení grafického objektu (vlastnosti `shape`). Současně se naučíme jeden důležitý princip.

Změny vlastností `background` a `shape` okna zatím nevedou k jeho překreslení, protože nezpůsobují zpětné volání `:display`. V těchto případech se o překreslení musí okno postarat samo. Udělá to tak, že zavolá funkci

`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 pomocí zpětného volání `:display`, a to až poté, co naše práce s oknem skončí. Zajistí také, že se okno překreslí pouze jednou.

Překreslování okna knihovny `micro-graphics`.

Okna zásadně nepřekreslujeme přímo, ale pomocí funkce `mg:invalidate`.

Příklad 5.3.1 (Překreslení okna po změně `background` a `shape`)

Nejprve zapouzdříme volání funkce `mg:invalidate` do nové metody `invalidate`:

```
(defmethod invalidate ((w window))
  (mg:invalidate (mg-window w))
  w)
```

Tuto metodu pak zavoláme kdykoli je třeba:

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

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

Nyní se sami můžete podívat, že se okna automaticky překreslují jak při nastavení `background`, tak `shape`:

```
CL-USER 24 > (setf w (make-instance 'window))
#<WINDOW 21CC8377>
```

```
CL-USER 25 > (set-background w :khaki)
#<WINDOW 21CC8377>

CL-USER 26 > (set-shape w (make-test-circle))
#<WINDOW 21CC8377>
```

5.4. Překreslování při změnách objektů

Nakonec naučíme okna automatickému překreslení při jakékoliv změně obsažených objektů. Tento úkol bude poněkud náročnější, protože bude vyžadovat přepracování všech metod grafických objektů, po jejichž volání je třeba objekty překreslit. Stále se budeme držet pravidla, že objekty nepřekresluje hned po jejich změně, ale nepřímo pomocí funkce `mg:invalidate` (zapsouzdřené v metodě `invalidate` třídy `window`).

Příklad 5.4.1 (překreslení po změně barvy)

Začneme jednoduchým případem: metoda `set-color` třídy `shape` zatím pouze nastavuje příslušný slot:

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

To nám nyní již nestačí. Po nastavení slotu by objekt měl nějak nahlásit, že u něj došlo ke změně, která vyžaduje překreslení okna. Především napíšeme novou metodu `change`, kterou budou objekty volat, kdykoli se změní. Metoda provede všechno potřebné: podívá se, jestli objekt má nastavené okno, a pokud ano, zašle mu zprávu informující o jeho změně.

```
(defmethod change ((shape shape))
  (when (window shape)
    (ev-change (window shape) shape)
    shape)
```

Zprávu zasílanou objektem oknu jsme nazvali `ev-change`. Předpona „ev“ znamená *event* neboli *událost*. Události jsou důležitým prvkem našeho systému; budeme se jim

věnovat později. Zatím nám stačí vědět, že jde o zvláštní typ zprávy. Zpráva `ev-change` má jeden parametr, kterým je objekt, jenž zprávu zasílá. Okno bude tedy informováno nejen o tom, že se nějaká část jeho obsahu změnila, ale bude také vědět která.

Nyní je tedy třeba napsat metodu `ev-change` pro třídu `window`. Ta nemusí dělat nic jiného než zavolat metodu `invalidate`:

```
(defmethod ev-change ((w window) shape)
  (invalidate w))
```

Nyní nám tedy zbývá změnit metodu `set-color` třídy `shape`, aby po provedení změny změnu ohlásila zasláním zprávy `change`:

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

Napsaný kód bychom opět měli vyzkoušet (nezapomínejme na to!). Treba takto:

```
CL-USER 5 > (setf w (make-instance 'window))
#<WINDOW 2009F157>

CL-USER 6 > (set-shape w (make-test-circle))
#<WINDOW 217141CF>

CL-USER 7 > (set-color (shape w) :purple)
#<CIRCLE 20099E53>
```

Po posledním vyhodnocení by se mělo kolečko v okně přebarvit.

Za chvíli uvidíme, že zvolené řešení není zcela vyhovující, a ještě je upravíme.

Příklad 5.4.2 (hlášení, že změna nastane)

Někdy se oknu může hodit, aby se dozvědělo o změně objektu v momentě, kdy se tato změna teprve chystá (například pro zapamatování stavu objektu před změnou). Předchozí příklad jasně ukazuje, jak takovou věc naprogramovat například pro zprávu `set-color`. Jediný podstatný rozdíl bude, že okno v reakci na informaci o chystané změně neudělá nic. Třída `window` ale bude připravena na to, aby její potomci přepsáním příslušných metod nějakou netriviální reakci umožnili. Zde je tedy příslušná obecná úprava ve třídách `shape` a `window`:

```
(defmethod changing ((shape shape))
  (when (window shape)
    (ev-changing (window shape) shape)
    shape)

(defmethod ev-changing ((w window) shape)
  w)
```

A například metoda `set-color` třídy `shape` bude upravena takto:

```
(defmethod set-color ((shape shape) value)
  (changing shape)
  (setf (slot-value shape 'color) value)
  (change shape))
```

Stále ještě nejde o verzi, se kterou se spokojíme. Jak uvidíme v dalších příkladech, bude dobré metodu ještě změnit.

Příklad 5.4.3 (překreslení po posunutí)

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 zpráv `changing` a `change` signalizovaly změnu grafického objektu a mezitím provedly akci specifickou pro příslušnou podtřídu.

Člověka by napadlo, že by se v této situaci hodilo místo funkce `call-next-method` použít nějakou jinou, která by nezavolala metodu implementovanou u předka, ale naopak u potomka třídy `shape`:


```
(defmethod move ((shape shape) dx dy)
  (changing shape)
  (call-previous-method)
  (change shape))

(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, jež budou metodami původními ve třídě *shape* volány:

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

(defmethod move ((shape shape) dx dy)
  (changing shape)
  (do-move shape dx dy)
  (change shape))

(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*.

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*.

Příklad 5.4.4 (nastavení barvy naposledy)

Inspirováni předchozím příkladem uděláme poslední změnu metody `set-color`. Ta se totiž rovněž dělí na dvě části: hlášení o změně a vlastní nastavení barvy. Přestože v případě této metody se obě části dějí ve stejné třídě (narozdíl od předchozího příkladu), je účelné je rozdělit do dvou metod. Poslední verze metody `set-color` bude tedy vypadat takto:

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

(defmethod set-color ((shape shape) value)
  (changing shape)
  (do-set-color shape value)
  (change shape))
```

Jako v předchozím příkladě, zpráva `set-color` je určena k volání uživatelem, zpráva `do-set-color` by mu měla být utajena. Je nachystána na případné přepsání v potomcích třídy `shape`, pokud by se v nich barva objektu ukládala jinak než do slotu nebo pokud by se kromě uložení barvy do slotu měla provést i další akce.

Všimněme si, že z pohledu uživatele jsou všechny změny, které jsme v této kapitole v naší grafické knihovně udělali, zpětně kompatibilní. Všechny příklady, které jsme pro knihovnu dříve naprogramovali, by měly i nadále fungovat.

ÚLOHY KE KAPITOLE 5

- 5.1. Upravte třídu `ellipse` podle principů z této kapitoly.
- 5.2. Napište potomka třídy `window`, jehož instance budou mít následující vlastnost: po změně libovolného objektu v okně se pozadí okna přebarví na jeho barvu.
- 5.3. Napište potomka třídy `window`, jehož instance budou mít následující vlastnost: před změnou libovolného objektu v okně se objeví dotaz, zda se má změna opravdu provést. Pokud uživatel zvolí možnost NE, dojde k chybě. Dialog s dotazem můžete vytvořit voláním

```
(capi:confirm-yes-or-no "Umožnit změnu?")
```

5.4. Definujte třídu `extended-picture`, která bude potomkem třídy `picture` a navíc bude mít vlastnost `propagate-color-p`. Pokud bude hodnota této vlastosti *Pravda*, obrázek při přijetí zprávy `set-color` nastaví na tutéž barvu barvu všech svých prvků.

Příloha A

Slovníček Scheme–Lisp

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 1. Pro další práci je samozřejmě nutné pokračovat ve studiu Common Lispu, například pomocí dalších částí Kapitoly 1. Je-li vysvětlení ve slovníčku příliš stručné, je třeba sáhnout po definici ve standardu.

Základní výrazy

Scheme

```
(define a x)
```

```
(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 ...))
```

Common Lisp

```
(defvar a x)  
(defparameter a x)  
  
(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) ...)
```

Poznámka pro CL

Pouze je-li `define` na nejvyšší úrovni. Lokální `define` v CL obdobu nemá (je třeba použít jiný nástroj, např. `let`).

Nastavuje ale funkční vazbu.

Nastavuje ale funkční vazbu.

Nastavuje ale funkční vazbu.

Scheme

if

(set! a x)

Common Lisp

if

(setf a x)

Poznámka pro CL

Je-li hodnota a v (if a b) nil, vrací nil (ve Scheme nedefinováno).
(setf a x) vrací hodnotu x.

Odvozené výrazy**Scheme**

cond

case

and, or

let, let*

letrec

begin

do

delay

quasiquote, unquote,

unquote-splicing

(x ,y ,@z)

Common Lisp

cond

case

and, or

let, let*

progn

do

(x ,y ,@z)

Poznámka pro CL

Základy stejné, místo else psát t, další drobné rozdíly.
Drobné rozdíly.

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.

Neexistuje.

Neexistují. Nutno používat
zkratky „“, „“, „a“, „@“.

Makra**Scheme**

let-syntax, letrec-
syntax, syntax-rules

Common Lisp**Poznámka pro CL**

V CL nejsou hygienická makra.
Obvyčejná makra pomocí defmacro

Standardní procedury — predikáty ekvivalence**Scheme**

eqv?

eq?

equal?

Common Lisp

eql

eq

equal, equalp

Poznámka pro CL

Přibližně

Přibližně

Přibližně

Standardní procedury — čísla

Scheme	Common Lisp	Poznámka pro CL
(number? x)	(numberp x), (typep x 'number)	
(complex? x)	(or (complexp x) (rationalp x)) (typep x '(or complex rational))	complexp vrací nil pro racionální čísla
(real? x)	(realp x), (typep x 'real)	
(rational? x)	(rationalp x), (typep x 'rational)	
(integer? x)	(integerp x), (typep x 'integer)	
exact?, inexact? =, <, >, <=, >= zero?, positive?, nega- tive?, odd?, even? max, min +, -, *, /	=, <, >, <=, >= zerop, plusp, minusp, oddp, evenp max, min +, -, *, /	CL nezná pojem přesného čísla
abs quotient remainder, modulo gcd, lcm numerator, denominator floor, truncate, ceiling, round	abs truncate rem, mod gcd, lcm numerator, denominator floor, truncate, ceiling, round	Vždy s libovolným počtem argu- mentů (u + a * i nulovým)
rationalize	rationalize	Jako druhou hodnotu vrací zbytek.
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	exp, log, sin, cos, tan, asin, acos, atan sqrt expt complex real-part, imag-part abs, phase prin1-to-string	Je možno zadat i dělitel (default 1), vrací podíl (zaokrouhlený) a zby- tek. Nemá druhý argument; počítá vždy co nejpresněji. log připouští druhý parametr — základ (default e)
string->number	parse-integer, read-from-string	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

Standardní procedury — logické typy

Scheme	Common Lisp	Poznámka pro CL
not (boolean? x)	not (typep x 'boolean)	

Standardní procedury — páry a seznamy

Scheme	Common Lisp	Poznámka pro CL
Scheme (pair? x)	Common Lisp (consp x), (typep x 'cons)	Poznámka pro CL
cons, car, cdr (set-car! x y), (set-cdr! x y) caar, caddr, ... (null? x) (list? x)	cons, car, cdr (setf (car x) y) (setf (cdr x) y) caar, caddr, ... (null x), (typep x 'null) (listp x), (typep x 'list)	Vrací y.
list length	list list-length	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.

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.

Standardní procedury — znaky

Scheme

```
(char? x)

char=?, char<?, char>?,
char<=?, char>=?
char-ci=?, char-ci<?,
char-ci>?, char-ci<=?,
char-ci>=?

char-alphabetic?
char-numeric?

char-whitespace?

char-upper-case?
char-lower-case?
char->integer
integer->char
char-upcase, char-
downcase
```

Common Lisp

```
(characterp x),
(typep x 'character)
char=, char<, char>,
char<=, char>=
char-equal, char-lessp,
char-greaterp,
char-not-greaterp,
char-not-lessp
alpha-char-p
digit-char-p

upper-case-p
lower-case-p
char-code
code-char
char-upcase, char-
downcase
```

Poznámka pro CL

Všechny akceptují lib. počet parametrů.

Všechny akceptují lib. počet parametrů.

Volitelně je možno zadat soustavu, jako true vrací příslušnou číselnou hodnotu.

Není. Souvisí to s tím, že v CL lze měnit syntax.

Standardní procedury — řetězce**Scheme**

```
(string? x)

(make-string k),
(make-string k fill)

string-length
(string-ref str k)

(string-set! str k char)

string=?, string<?,
string>?, string<=?,
string>=?
string-ci=?, string-ci<?,
string-ci>?, string-
ci<=?, string-ci>=?

substring

(string-append str1 ...)
```

Common Lisp

```
(stringp x),
(typep x 'string)
(make-string k),
(make-sequence 'string k)
(make-string
k
:initial-element fill)
(make-sequence
'string
k
:initial-element fill)
length
(char str k), (elt str
k), (aref str k)
(setf (char str k) char),
(setf (elt str k) char),
(setf (aref str k) char)
string=, string<,
string>, string<=,
string>=
string-equal, string-
lessp, string-greaterp,
string-not-greaterp,
string-not-lessp
subseq

(concatenate 'string
str1
...)
```

Poznámka pro CL

Pracuje nejen s řetězci.

Přesná obdoba je char, lépe je používat elt nebo aref.

Vrací char.

Všechny akceptují lib. počet parametrů.

Všechny akceptují lib. počet parametrů.

Pracuje nejen s řetězci, poslední parametr je volitelný.

Pracuje nejen s řetězci; aby byl výsledek řetězec, je třeba použít 'string jako první parametr.

Scheme	Common Lisp	Poznámka pro CL
(string->list str), (list->string lst) string-copy string-fill!	(coerce str 'list), (coerce lst 'string) copy-seq fill	Pracuje nejen s řetězci. Pracuje nejen s řetězci, volitelně lze určit rozmezí.

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) (make-vector k), (make-vector k fill) vector vector-length (vector-ref vec k) (vector-set! vec k x) (vector->list vec), (list->vector lst) vector-fill!	(vectorp x), (typep x 'vector) (make-vector k), (make-sequence 'vector k) (make-vector k :initial-element fill) (make-sequence 'vector k :initial-element fill) vector length (elt vec k), (aref vec k) (setf (elt vec k) x), (setf (aref vec k) x) (coerce vec 'list), (coerce lst 'vector) fill	Pracuje nejen s vektory, volitelně lze určit rozmezí.

Standardní procedury — řízení běhu

Scheme	Common Lisp	Poznámka pro CL
(procedure? x) apply map for-each force call-with-current-continuation values	(functionp x), (typep x 'function) apply mapcar mapc values	Mírnější podmínky. Pozor, funkce map dělá něco jiného. Neexistuje. Neexistuje. Mírnější podmínky.

Scheme

call-with-values
dynamic-wind

Common Lisp

multiple-value-call
unwind-protect

Poznámka pro CL

Přibližně.

Velmi přibližně; unwind-protect je speciální operátor a je jednodušší, protože není call-with-current-continuation.

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.

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?

write

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

prinl

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

Jedna z mnoha funkcí pro zápis objektu. Neplést s funkcí write.

Scheme	Common Lisp	Poznámka pro CL
<code>display</code>	<code>princ</code>	Jedna z mnoha funkcí pro zápis objektu.
<code>newline</code>	<code>terpri</code>	Viz též <code>fresh-line</code> .
<code>write-char</code>	<code>write-char</code>	

Standardní procedury — systémové rozhraní

Scheme	Common Lisp	Poznámka pro CL
<code>load</code>	<code>load</code>	Více možností
<code>transcript-on</code> , <code>transcript-off</code>		Neexistuje, dělá se jinak.

Příloha B

Klávesové zkratky v LW

Editor v LispWorks funguje na základě příkazů, které lze zadávat názvem, nebo (pokud ji příkaz má) klávesovou zkratkou. Zde uvádíme několik základních zkratek pro režimy Windows a Mac OS. Mapování příkazů na zkratky a zpět lze zjistit z menu (Help/Editing). Dokumentace příkazů je v příručce „Editor User Guide“, která je k dispozici ve vývojovém prostředí a [na webu](#) (je třeba vybrat správnou platformu a verzi).

Pokročilejší uživatele bude zajímat, že mapování klávesových zkratek na příkazy editoru lze libovolně měnit a že lze také programovat nové příkazy. Obojí používá autor tohoto textu na přednáškách na automatizaci práce s LispWorks.

B.1. Režim Windows

Režim je k dispozici na počítačích s OS Windows, je třeba ho zapnout v předvolbách LispWorks, panel Environment, podpanel Emulation. Zkratky, které jsou v rámci Windows standardní, neuvádíme.

Klávesa Meta se emuluje stisknutím Ctrl-m.

Zkratky pro příkazový řádek (Listener)

Zkratka
Ctrl-Down
Ctrl-Up

Příkaz
History Next
History Previous

Poznámka

Globální zkratky

Zkratka	Příkaz	Poznámka
Tab	Indent Selection or Complete Symbol	Při psaní symbolu nabídne jeho doplnění, jinak správně zarovná řádky
F7	Compile Defun	Zkompiluje definici, na které je kurzor
Ctrl-F7	Compile Buffer	Zkompiluje celý obsah okna
Ctrl-Shift-F7	Compile Region	Zkompiluje označený text
F1	Help	Seznam klávesových zkratek Vypíše dole seznam parametrů právě psané funkce, makra, spec. operátoru
F1 b	Function Argument List	
Shift-F1		
F2	Function Documentation	
F3	Incremental Search	Zobrazí dokumentaci k symbolu
Meta-.	Find Source	Další stisk vyhledá další výskyt
Meta-,	Continue Tags Search	Najde zdrojový kód k symbolu (první výskyt)
Meta-x	Extended Command	Další výskyty zdrojového kódu
		Vyvolání libovolného příkazu napsáním jeho názvu

B.2. Režim Mac OS

Režim je k dispozici na počítačích s Mac OS, je třeba ho zapnout v předvolbách LispWorks, panel Environment, podpanel Emulation. Zkratky, které jsou v rámci Mac OS standardní, neuvádíme.

Klávesu Meta lze v základním nastavení emulovat stisknutím Ctrl-m (toto nastavení lze změnit v předvolbách, ale není to vhodné). Pro některé zkratky to nefunguje, pak je třeba použít klávesu Option (Alt). Je také vhodné vypnout v globálních systémových předvolbách zkratky Ctrl-Right a Ctrl-Left.

Zkratky pro příkazový řádek (Listener)

Zkratka	Příkaz	Poznámka
Ctrl-c Ctrl-n	History Next	
Ctrl-c Ctrl-p	History Previous	
Ctrl-c <	History First	
Ctrl-c >	History Last	

Globální zkratky

Zkratka	Příkaz	Poznámka
Ctrl-Left	Beginning of Line Cancelling Selection	
Ctrl-Right	End Of Line Cancelling Selection	
Meta-Ctrl-Left	Backward Form Cancelling Selection	Posun o výraz doleva
Meta-Ctrl-Right	Forward Form Cancelling Selection	Posun o výraz doprava
Meta-Ctrl-Prior	Beginning of Defun Cancelling Selection	Posun na začátek definice

Zkratka	Příkaz	Poznámka
Meta-Ctrl-Next	End of Defun Cancelling Selection	(Prior = Page Up) Posun na konec definice
Tab	Indent Selection or Complete Symbol	(Next = Page Down) Při psaní symbolu nabídne jeho doplnění, jinak správně zarovná řádky
F7	Compile Defun	Zkompiluje definici, na které je kurzor
Ctrl-F7	Compile Buffer	Zkompiluje celý obsah okna
Ctrl-Shift-F7	Compile Region	Zkompiluje označený text
Ctrl-s	Incremental Search	Další stisk vyhledá další výskyt
F1	Help	
F1 b		Seznam klávesových zkratk
Shift-F1	Function Argument List	Vypíše dole seznam parametrů právě psané funkce, makra, spec. operátoru
F2	Function Documentation	Zobrazí dokumentaci k symbolu
Meta-.	Find Source	Najde zdrojový kód k symbolu (první výskyt)
Meta-,	Continue Tags Search	Další výskyty zdrojového kódu
Meta-x	Extended Command	Vyvolání libovolného příkazu napsáním jeho názvu

Příloha C

Knihovna `micro-graphics`

Knihovna `micro-graphics` je jednoduchá nízkoúrovňová procedurální grafická knihovna pro prostředí LispWorks určená k praktickému procvičování objektového programování.

Knihovna se do prostředí LispWorks načítá tak, že se načte (funkcí `load` nebo z menu) soubor `load.lisp` (musí být ve stejném adresáři jako soubory `micro-graphics.lisp` a `package.lisp`). K dispozici dává následující funkce:

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

window: odkaz na okno

Vytvoří a zobrazí nové grafické okno. Jako výsledek vrátí 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`. Pomocí funkce `mg:get-callback` lze také zjišťovat a pomocí funkce `mg:set-callback` nastavovat zpětná volání. Souřadnice v okně se udávají v pixelech, jejich počátek je v levém horním rohu okna, hodnoty druhé souřadnice se zvětšují směrem dolů. Rozměry okna jsou 297 na 210 pixelů.

```
(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 na kreslení obrazců (např. <code>mg:draw-circle</code>), pokud není nastaven parametr <code>:filledp</code> . Počáteční hodnota: 1.
:foreground	Barva inkoustu. Ovlivňuje funkce na kreslení obrazců (např. <code>mg:draw-circle</code>). Počáteční hodnota: <code>:black</code> .
:background	Barva pozadí. Ovlivňuje funkci <code>mg:clear</code> . Počáteční hodnota: <code>:white</code> .
:filledp	Zda kreslit obrazce vyplněné. Ovlivňuje funkce na kreslení obrazců (např. <code>mg:draw-circle</code>). Počáteční hodnota: <code>nil</code> .
:closedp	Zda spojit poslední a první vrchol polygonu. Ovlivňuje funkci <code>mg:draw-polygon</code> , pokud není nastaven parametr <code>filledp</code> . Počáteční hodnota: <code>nil</code> .
:mask	Udává část okna, na kterou se omezí kreslení. Může být <code>nil</code> (bez omezení), nebo seznam definujících body polygonu (pro formát tohoto seznamu viz funkci <code>mg:draw-polygon</code>). Kreslení je pak omezeno na vnitřní body tohoto polygonu. Ovlivňuje všechny kreslicí funkce.

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 Přílohy D.

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 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-ellipse window x y rx ry phi) => nil

window: hodnota vrácená funkcí mg:display-window
x, y, rx, ry, phi: čísla
```

Funkce `mg:draw-ellipse` nakreslí do okna *window* elipsu se středem o souřadnicích *x*, *y* s poloosami *rx* a *ry*. Elipsa bude natočená (kolem středu) o úhel *phi*. Kreslí se barvou uloženou v kreslicím parametru `:foreground` okna *window*. Kreslicí parametr `:filledp` okna *window* udává, zda se bude elipsa kreslit vyplněná. Pokud není nastaven, bude se kreslit pouze obvod 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 *xové* a *yové* 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.

```
(mg:close-window window) => nil
```

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

Uvolní systémové prostředky alokované pro okno *window* a okno vymaže z obrazovky. Předtím zavolá zpětné volání `:destroy` (viz níže).

```
(mg:get-callback window callback-name) => callback
```

window: hodnota vrácená funkcí `mg:display-window`
callback-name: symbol, přípustné hodnoty viz níže

Vrací zpětné volání názvu *callback-name* instalované v okně *window*. *callback-name* musí být jeden ze symbolů `:display`, `:mouse-down`, `:mouse-up`, `:mouse-move`, `:double-click`, `:destroy`, `:resize`, `:character`, `:activate`. Zpětná volání lze také nastavit pomocí funkce `mg:set-callback`.

Popis jednotlivých typů zpětných volání:

`:display`: funkce s jedním parametrem. Knihovna tuto funkci volá kdykoliv je nutno okno překreslit. Jako parametr použije odkaz na okno (vrácený funkcí `mg:display-window`).

`:mouse-down`: funkce akceptující čtyři parametry: odkaz na okno (vrácený funkcí `mg:display-window`), označení stisknutého tlačítka myši (jeden ze symbolů `:left`, `:center`, `:right`) a vodorovnou a svislou souřadnici myši. Zpětné volání knihovna zavolá při stisku tlačítka myši v okně.

`:mouse-up`, `:mouse-move`, `:double-click`: stejný význam jako `:mouse-down`, liší se pouze situacemi, za nichž je knihovna volá. Zpětné volání `:mouse-move` může být navíc voláno s druhým parametrem rovným `nil`.

`:destroy`: funkce akceptující jeden parametr. Knihovna tuto funkci zavolá při zavření okna uživatelem nebo funkcí `mg:close-window`. Jako parametr použije odkaz na okno (vrácený funkcí `mg:display-window`).

`:activate`: funkce akceptující dva parametry. Knihovna tuto funkci zavolá při aktivaci resp. deaktivaci okna. O kterou z těchto možností jde, určuje druhý parametr (*Pravda*: aktivováno, *Nepravda*: deaktivováno). První parametr je okno.

`:resize`: funkce akceptující tři parametry. Knihovna tuto funkci volá, kdykoliv dojde ke změně rozměrů okna. Jako parametry použije okno (vrácené funkcí `mg:display-window`) a nové rozměry jeho obsahu (šířku a výšku v pixelech).

`:character`: funkce akceptující dva parametry. Knihovna tuto funkci volá, kdykoliv uživatel stiskne klávesu. Prvním parametrem je okno, druhým znak odpovídající klávese.

```
(mg:set-callback window callback-name callback) => nil
```

window: hodnota vrácená funkcí `mg:display-window`
callback-name: symbol, přípustné hodnoty viz výše

Nastavuje zpětné volání *callback-name* okna *window* na funkci *callback*. Popis zpětných volání najdete u funkce `mg:get-callback`.

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

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

Oznámí knihovně *micro-graphic*, že okno *window* má neaktuální obsah. Knihovna pak ve vhodný moment zavolá zpětné volání `:display`.

```
(mg:get-string-extent window string)
=> (left top right bottom)

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

Vrátí rozměry, které by měl řetězec *string* po vykreslení do okna *window*.

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

window: hodnota vrácená funkcí mg:display-window
string: řetězec
x, y: čísla
```

Vykreslí řetězec *string* do okna *window* na souřadnice *x y*.

```
(mg:colorp object) => spec

object: libovolná hodnota
```

Vrátí *Pravdu*, pokud hodnota *object* reprezentuje barvu. Výsledek *spec* je pak reprezentace barvy, použitelná jako vstup pro funkce grafického systému LispWorks pracujících s barvami (např. `color:color-hue`).

Příloha D

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		: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		:DARKOLIVEGREEN		:DARKOLIVEGREEN1	
:DARKOLIVEGREEN2		:DARKOLIVEGREEN3		:DARKOLIVEGREEN4	
:DARKORANGE		:DARKORANGE1		:DARKORANGE2	
:DARKORANGE3		:DARKORANGE4		:DARKORCHID	

PŘÍLOHA D. KNIHOVNA MICRO-GRAPHICS: SEZNAM POUŽITELNÝCH
144 BAREV

: 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	

PŘÍLOHA D. KNIHOVNA MICRO-GRAPHICS: SEZNAM POUŽITELNÝCH
146 BAREV

:LIGHTCORAL		:LIGHTCYAN		:LIGHTCYAN1	
:LIGHTCYAN2		:LIGHTCYAN3		:LIGHTCYAN4	
:LIGHTGOLDENROD		:LIGHTGOLDENROD1		:LIGHTGOLDENROD2	
:LIGHTGOLDENROD3		:LIGHTGOLDENROD4		:LIGHTGOLDENRODYELLOW	
:LIGHTGRAY		:LIGHTGREY		:LIGHTPINK	
:LIGHTPINK1		:LIGHTPINK2		:LIGHTPINK3	
:LIGHTPINK4		:LIGHTSALMON		:LIGHTSALMON1	
:LIGHTSALMON2		:LIGHTSALMON3		:LIGHTSALMON4	
:LIGHTSEAGREEN		:LIGHTSKYBLUE		:LIGHTSKYBLUE1	
:LIGHTSKYBLUE2		:LIGHTSKYBLUE3		:LIGHTSKYBLUE4	
:LIGHTSLATEBLUE		:LIGHTSLATEGRAY		:LIGHTSLATEGREY	
:LIGHTSTEELBLUE		:LIGHTSTEELBLUE1		:LIGHTSTEELBLUE2	
:LIGHTSTEELBLUE3		:LIGHTSTEELBLUE4		:LIGHTYELLOW	
:LIGHTYELLOW1		:LIGHTYELLOW2		:LIGHTYELLOW3	
:LIGHTYELLOW4		:LIMEGREEN		:LINEN	
:LISPWORKS-BLUE		:MAGENTA		:MAGENTA1	
:MAGENTA2		:MAGENTA3		:MAGENTA4	
:MAROON		:MAROON1		:MAROON2	
:MAROON3		:MAROON4		:MEDIUM-BLUE	
:MEDIUM-BROWN		:MEDIUM-GREEN		:MEDIUM-YELLOW	
:MEDIUMAQUAMARINE		:MEDIUMBLUE		:MEDIUMORCHID	
:MEDIUMORCHID1		:MEDIUMORCHID2		:MEDIUMORCHID3	
:MEDIUMORCHID4		:MEDIUMPURPLE		:MEDIUMPURPLE1	
:MEDIUMPURPLE2		:MEDIUMPURPLE3		:MEDIUMPURPLE4	
:MEDIUMSEAGREEN		:MEDIUMSLATEBLUE		:MEDIUMSPRINGGREEN	
:MEDIUMTURQUOISE		:MEDIUMVIOLETRED		:MIDNIGHTBLUE	
:MINTCREAM		:MISTYROSE		:MISTYROSE1	
:MISTYROSE2		:MISTYROSE3		:MISTYROSE4	
:MOCCASIN		:NAVAJOWHITE		:NAVAJOWHITE1	
:NAVAJOWHITE2		:NAVAJOWHITE3		:NAVAJOWHITE4	
:NAVY		:NAVYBLUE		:OLDLACE	
:OLIVEDRAB		:OLIVEDRAB1		:OLIVEDRAB2	
:OLIVEDRAB3		:OLIVEDRAB4		:ORANGE	
:ORANGE1		:ORANGE2		:ORANGE3	
:ORANGE4		:ORANGERED		:ORANGERED1	
:ORANGERED2		:ORANGERED3		:ORANGERED4	
:ORCHID		:ORCHID1		:ORCHID2	
:ORCHID3		:ORCHID4		:PALEGOLDENROD	
:PALEGREEN		:PALEGREEN1		:PALEGREEN2	
:PALEGREEN3		:PALEGREEN4		:PALETURQUOISE	
:PALETURQUOISE1		:PALETURQUOISE2		:PALETURQUOISE3	
:PALETURQUOISE4		:PALEVIOLETRED		:PALEVIOLETRED1	
:PALEVIOLETRED2		:PALEVIOLETRED3		:PALEVIOLETRED4	
:PAPAYAWHIP		:PEACHPUFF		:PEACHPUFF1	
:PEACHPUFF2		:PEACHPUFF3		:PEACHPUFF4	
:PERU		:PINK		:PINK1	
:PINK2		:PINK3		:PINK4	
:PLUM		:PLUM1		:PLUM2	
:PLUM3		:PLUM4		:POWDERBLUE	
:PURPLE		:PURPLE1		:PURPLE2	
:PURPLE3		:PURPLE4		:RED	
:RED1		:RED2		:RED3	

:RED4		:ROSYBROWN		:ROSYBROWN1	
:ROSYBROWN2		:ROSYBROWN3		:ROSYBROWN4	
:ROYALBLUE		:ROYALBLUE1		:ROYALBLUE2	
:ROYALBLUE3		:ROYALBLUE4		:SADDLEBROWN	
:SALMON		:SALMON1		:SALMON2	
:SALMON3		:SALMON4		:SANDYBROWN	
:SEAGREEN		:SEAGREEN1		:SEAGREEN2	
:SEAGREEN3		:SEAGREEN4		:SEASHELL	
:SEASHELL1		:SEASHELL2		:SEASHELL3	
:SEASHELL4		:SIENNA		:SIENNA1	
:SIENNA2		:SIENNA3		:SIENNA4	
:SKYBLUE		:SKYBLUE1		:SKYBLUE2	
:SKYBLUE3		:SKYBLUE4		:SLATEBLUE	
:SLATEBLUE1		:SLATEBLUE2		:SLATEBLUE3	
:SLATEBLUE4		:SLATEGRAY		:SLATEGRAY1	
:SLATEGRAY2		:SLATEGRAY3		:SLATEGRAY4	
:SLATEGREY		:SNOW		:SNOW1	
:SNOW2		:SNOW3		:SNOW4	
:SPRINGGREEN		:SPRINGGREEN1		:SPRINGGREEN2	
:SPRINGGREEN3		:SPRINGGREEN4		:STEELBLUE	
:STEELBLUE1		:STEELBLUE2		:STEELBLUE3	
:STEELBLUE4		:TAN		:TAN1	
:TAN2		:TAN3		:TAN4	
:THISTLE		:THISTLE1		:THISTLE2	
:THISTLE3		:THISTLE4		:TOMATO	
:TOMATO1		:TOMATO2		:TOMATO3	
:TOMATO4		:TRANSPARENT		:TURQUOISE	
:TURQUOISE1		:TURQUOISE2		:TURQUOISE3	
:TURQUOISE4		:VIOLET		:VIOLETRED	
:VIOLETRED1		:VIOLETRED2		:VIOLETRED3	
:VIOLETRED4		:WHEAT		:WHEAT1	
:WHEAT2		:WHEAT3		:WHEAT4	
:WHITE		:WHITESMOKE		:YELLOW	
:YELLOW1		:YELLOW2		:YELLOW3	
:YELLOW4		:YELLOWGREEN			