KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

CVIČENÍ Z PARADIGMAT PROGRAMOVÁNÍ I

Lekce 3: Lokální vazby a definice

Učební materiál k přednášce 19. října 2006
(pracovní verze textu určená pro studenty)

JAN KONEČNÝ, VILÉM VYCHODIL

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

Olomouc 2006

Lekce 3: Lokální vazby a definice

Obsah lekce: V této kapitole objasníme důvody, proč potřebujeme lokální vazby, obeznámíme se se dvěma novými speciálními formami pro vytváření lokálních prostředí a vazeb v nich. Dále vysvětlíme pojem abstrakční bariéra a seznámíme se se dvěma důležitými styly vytváření programů: top-down a bottom-up. Dále rozšíříme λ-výrazy tak, abychom v jejich těle mohli používat speciální formu `define`, modifikovat tak lokální prostředí procedur a vytvářet interní definice.

Klíčová slova: abstrakční bariéry, blackbox, interní definice, `let*`-blok, `let`-blok, lokální vazba, rozsah platnosti, styl bottom-up, styl top-down, top-level definice

3.1 Vytváření lokálních vazeb

V prvních dvou lekcích jsme se seznámili s několika způsoby, jakými může být v jazyku Scheme hodnota navázána na symbol. Víme už, že v počátečním prostředí interpretu jazyka Scheme jsou na některé symboly navázány primitivní procedury, zabudované speciální formy a některé další elementy. Také již umíme zavádět nové vazby do globálního prostředí (respektive měnit stávající vazby) použitím speciální formy `define`. Dále víme, že při aplikaci uživatelské procedury vznikají vazby mezi formálními argumenty a hodnotami, na které je procedura aplikována. Tyto vazby přitom vznikají v lokálním prostředí aplikace této procedury, nikoli v globálním prostředí, jako v předchozích dvou případech. Vazby tohoto typu jsou tedy platné jen „v těle procedury“. Takovéto vazby nazýváme *lokální*. Právě lokálním vazbám se budeme věnovat v této lekci.

K čemu vlastně jsou lokální vazby? Důvody jsme si už vlastně řekli v lekci 1, kde jsme hovořili o abstrakcích vytvářených pojmenováním hodnot:

- Pojmenovaní hodnot může vést ke odstranění redundance kódu a *zvýšit tak jeho efektivitu*.
- Názvy symbolů nám mohou říkat, jakou roli jejich hodnota hraje. Tím se kód stává *čitelnějším*.

Demonstrujme si předchozí dva body na příkladech: Předpokládejme, že chceme napsat proceduru, která počítá povrch válce S podle jeho objemu V a výšky h. Ze vzorce pro výpočet objemu válce

$$V = \pi r^2 h$$

vyjádříme poloměr rotačního válce

$$r = \sqrt{\frac{V}{\pi h}}.$$

Poté takto stanovený poloměr dosadíme do vzorce na výpočet povrchu $S = 2\pi r(r + h)$. Všimněte si, že ve vzorci na výpočet povrchu se vyskytuje hodnota poloměru dvakrát. Jedním z řešení by bylo naprogramovat proceduru pro výpočet povrchu tak, že bychom vyšli ze vzorce pro výpočet S a nahradili v něm oba výskyty r vzorcem pro výpočet hodnoty r z objemu válce a výšky:

$$S = 2\pi\sqrt{\frac{V}{\pi h}}\left(\sqrt{\frac{V}{\pi h}} + h\right).$$

To ale není z programátorského hlediska příliš čisté řešení. Proceduru bychom mohli napsat také takto:

```
(define povrch-valce
  (lambda (V h)
    ((lambda (r)
       (* 2 r pi (+ r h)))
     (sqrt (/ V pi h)))))
```

V těle procedury `povrch-valce` tedy vytváříme novou proceduru vyhodnocením výrazu

`(lambda (r) (* 2 r pi (+ r h)))`.

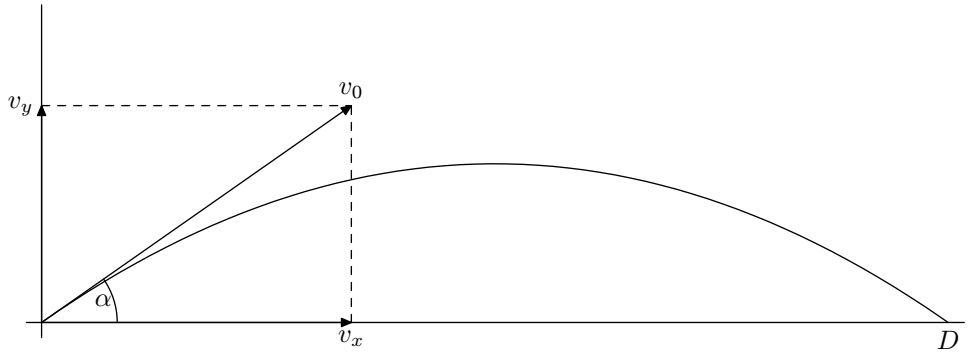
Tu hned aplikujeme na vyhodnocení výrazu na výpočet poloměru r. Nejde nám ani tak o proceduru samotnou, ale o lokální vazbu symbolu `r` na poloměr r, která vznikne její jednorázovou aplikací. V těle procedury

vzniklé vyhodnocením vnitřního λ-výrazu však můžeme zacházet s poloměrem jako s pojmenovanou hodnotou a neuvádět tak vícekrát stejnou část kódu (na výpočet poloměru).

Jiný příklad, kde se lokální vazba používá k tomuto účelu, je úkol č. 8 z předchozí lekce.

A teď druhý příklad. Tentokrát využijeme lokálních vazeb k pojmenování jednotlivých hodnot podle jejich rolí tak, abychom zlepšili čitelnost programu. Chtěli bychom proceduru, ve které počítáme délku šikmého vrhu (ve vakuu). Vstupními hodnotami pro nás bude úhel α, pod kterým je těleso vrženo a počáteční rychlost v_0. V těle této procedury přitom chceme používat symboly, které odpovídají proměnným v následujícím odvození: Okamžitá rychlost v je dána vektorovým součtem svislé $v_y = v\sin(\alpha)$ a vodorovné

Obrázek 3.1. Šikmý vrh ve vakuu



rychlosti $v_x = v\cos(\alpha)$. Vodorovná rychlost v_x je přitom stále stejná. Dostřel je tedy dán vodorovnou rychlostí v_x a časem T_d, po který těleso letí: $D = v_x T_d$. Let tělesa má dvě fáze. V první fázi letí těleso nahoru než dosáhne nulové svisle rychlosti. Pak přechází do druhé fáze a začíná padat. První fáze trvá po dobu $t_1 = \frac{v_y}{g}$, a těleso při ní dosáhne výšky $\frac{{v_y}^2}{2g}$. Doba, po kterou bude těleso z této výšky padat, je $t_2 = \frac{v_y}{g}$. Tedy $T_d = t_1 + t_2 = \frac{2v_y}{g}$. Přímočarým přepisem předchozího odvození a s využitím pomocné procedury pro vytvoření lokálních vazeb, naprogramujeme proceduru pro výpočet dostřelu takto:

```
(define dostrel
  (lambda (v0 alfa g)
    ((lambda (vx vy)
       (* 2 vx vy (/ g)))
     (* v0 (cos alfa))
     (* v0 (sin alfa)))))
```

Vidíme, že ačkoli bylo naším záměrem zpřehlednění kódu, dosáhli jsme efektu spíše opačného. A to jsme ještě nevytvářeli lokální vazbu na symbol `Td`, který by odpovídal proměnné T_d. Přehlednější, a přitom ekvivalentní, kód se dá napsat s použitím *speciální formy* `let`. Takto by vypadala procedura `dostrel` napsaná pomocí této speciální formy:

```
(define dostrel
  (lambda (v0 alfa g)
    (let ((vx (* v0 (cos alfa)))
          (vy (* v0 (sin alfa))))
      (* 2 vx vy (/ g)))))
```

Všimněte si, v čem vlastně spočívalo zpřehlednění programu. Jde především o to, že v kódu máme výrazy a symboly, na které budou vyhodnocení těchto výrazů lokálně navázány, uvedeny hned vedle sebe. Kdežto při použití λ-výrazu (jako v předchozím případě) jsou symboly v seznamu formálních argumentů, a jejich hodnoty jsou pak až za λ-výrazem.

Nyní se podrobně podíváme na použití a aplikaci speciální formy `let`.

Definice 3.1. Speciální forma `let` se používá ve tvaru:

```
(let ((⟨symbol_1⟩ ⟨hodnota_1⟩)
      (⟨symbol_2⟩ ⟨hodnota_2⟩)
       ⋮
      (⟨symbol_n⟩ ⟨hodnota_n⟩))
 ⟨tělo⟩),
```

kde n je nezáporné celé číslo, $\langle symbol_1\rangle$, $\langle symbol_2\rangle$, ..., $\langle symbol_n\rangle$ jsou vzájemně různé symboly, $\langle hodnota_1\rangle$, $\langle hodnota_2\rangle$, ..., $\langle hodnota_n\rangle$ a $\langle tělo\rangle$ jsou libovolné S-výrazy. Tento výraz se nazývá `let`-blok (někdy též `let`-výraz). Vyhodnocení `let`-bloku v tomto tvaru je ekvivalentní vyhodnocení výrazu

$$((\texttt{lambda}\ (\langle symbol_1\rangle\ \langle symbol_2\rangle \cdots \langle symbol_n\rangle)\ \langle tělo\rangle)\ \langle hodnota_1\rangle\ \langle hodnota_2\rangle \cdots \langle hodnota_n\rangle) \qquad \blacksquare$$

V definici 3.1 jsme zavedli sémantiku `let`-bloku tím, že jsme uvedli symbolický výraz, který se vyhodnotí stejně. Aplikaci speciální formy `let` v prostředí $\mathcal{P}$ bychom mohli však mohli popsat nezávisle takto:

(1) S-výrazy $\langle hodnota_1\rangle$, $\langle hodnota_2\rangle$ ... $\langle hodnota_n\rangle$ jsou vyhodnoceny *v aktuálním prostředí* $\mathcal{P}$. Pořadí jejich vyhodnocování přitom *není specifikováno*. Výsledky jejich vyhodnocení označme E_i, kde $i = 1, \ldots, n$.

(2) Vytvoří se *nové prázdné prostředí* $\mathcal{P}_l$, Tabulka vazeb prostředí $\mathcal{P}_l$ je v tomto okamžiku prázdná (neobsahuje žádné vazby), předek prostředí $\mathcal{P}_l$ není nastaven.

(3) *Nastavíme předka prostředí* $\mathcal{P}_l$ *na hodnotu* $\mathcal{P}$ (předkem prostředí $\mathcal{P}_l$ je *aktuální prostředí*).

(4) V prostředí $\mathcal{P}_l$ se zavedou vazby $\langle symbol_i\rangle \mapsto E_i$ pro $i = 1, \ldots, n$.

(5) Výsledek vyhodnocení je pak roven $\mathrm{Eval}[\langle tělo\rangle, \mathcal{P}_l]$.

Poznámka 3.2. (a) Zřejmě každý `let`-blok je symbolický výraz, protože je to, obdobně jako v případě λ-výrazů, seznam ve speciálně vyžadovaném tvaru. Příklady `let`-bloků jsou třeba

```
(let ((x 10) (y 20)) (+ x y 1))
(let ((x 10)) (* x x))
(let ((a 1) (b 2) (c 3)) 0)
  ⋮
```

Kvůli čitelnosti píšeme obvykle `let`-bloky do více řádků, každou vazbu na nový řádek a tělo rovněž zvlášť. Viz následující příklad přepisu prvního z výše uvedených `let`-bloků.

```
(let ((x 10)
      (y 20))
   (+ x y 1))
```

(b) Všimněte si, že definice 3.1 připouští i nulový počet dvojic `(`$\langle symbol_i\rangle$ $\langle hodnota_i\rangle$`)`, takže například symbolický výraz ve tvaru `(let () 10)` je také `let`-blok.

(c) Vyhodnocením `let`-bloku je provedena aplikace speciální formy navázané na symbol `let`. Je zřejmé, že element navázaný na symbol `let` nemůže být procedura. V takovém případě by vyhodnocení `let`-bloku ve tvaru `(let ((x 10)) (* x x))` skončilo chybou v kroku (B.e), protože symbol `x` nemá vazbu.

(d) Vidíme, že speciální forma `let` je nadbytečná, protože každý `let`-blok můžeme nahradit ekvivalentním kódem pomocí λ-výrazu. Umožňuje však přehlednější zápis kódu – a to především díky tomu, že symboly vázané `let`-blokem jsou uvedeny hned vedle hodnot, na které jsou navázány.

I když je forma `let` nadbytečná ve smyslu, že vše co lze pomocí ní v programu vyjádřit, bychom mohli vyjádřit i bez ní, v žádném případě to ale neznamená, že je snad „nepraktická“, právě naopak. Všechny soudobé programovací jazyky obsahují spoustu konstrukcí, které jsou tímto způsobem nadbytečné, ale výrazně ulehčují programátorům práci. Z trochou nadsázky můžeme říct, že kvalita a propracovanost programovacího jazyka se pozná právě pomocí množství a potenciálu (nadbytečných) konstrukcí, které jsou k dispozici programátorovi. Strohé programovací jazyky obsahující pouze nezbytně nutné minimum konstrukcí se v praxi nepoužívají a většinou slouží pouze jako teoretické výpočetní formalismy.

Příklad 3.3. (a) Výsledkem vyhodnocení `let`-bloku `(let ((a (+ 5 5))) (+ a 2))` je číslo 12. Průběh jeho vyhodnocování je následující. V aktuálním prostředí – v tomto případě v globálním prostředí – se vyhodnotí výraz `(+ 5 5)` na číslo `10`. Je vytvořeno nové prázdné prostředí $\mathcal{P}$, do něj je přidána vazba `a` $\mapsto_{\mathcal{P}}$ `10`, jako předek je nastaveno aktuální (globální) prostředí. V prostředí $\mathcal{P}$ pak bude vyhodnoceno tělo `(+ a 2)`. Vazba symbolu `+` na proceduru sčítání je nalezena v lexikálním předkovi prostředí $\mathcal{P}$, to jest v globálním prostředí. Vazbu `a` $\mapsto_{\mathcal{P}}$ `10` jsme přidali do lokálního prostředí. Výsledkem vyhodnocení je tedy 12.

(b) `let`-blok `(let ((+ (+ 10 20))) +)` se vyhodnotí tímto způsobem: v globálním prostředí se vyhodnotí výraz `(+ 10 20)` na číslo `30`. Je vytvořeno nové prázdné prostředí s vazbou `+` $\mapsto$ `30`, jako předek bude nastaveno aktuální (globální) prostředí. V novém prostředí pak bude vyhodnoceno tělo `+` a jelikož je v tomto prostředí `+` navázáno na hodnotu `30`, je číslo `30` výsledkem vyhodnocení celého výrazu. Vazba symbolu `+` na proceduru sčítání v globálním prostředí v tomto případě při vyhodnocení těla nehraje žádnou roli. Upozorněme na fakt, že vazba `+` v těle `key`-bloku je skutečně lokální, tedy globální definice `+` na primitivní proceduru sčítání čísel se vně `let`-bloku nijak nezmění.

(c) Uvažujme vyhodnocení následujících dvou výrazů z nichž druhý je `let`-blok.

```
(define x 10)
(let ((x (+ x 1))
      (y (+ x 2)))
  y)  ⟹  12
```

Nejprve jsou oba výrazy `(+ x 1)` a `(+ x 2)` vyhodnoceny v aktuálním (tedy globálním) prostředí. Jelikož je v tomto prostředí `x` navázáno na číslo `10`, budou výsledkem vyhodnocení čísla `11` a `12`. Bude vytvořeno nové prázdné prostředí $\mathcal{P}$ a do něj budou přidány vazby `x` $\mapsto_{\mathcal{P}}$ `11` a `y` $\mapsto_{\mathcal{P}}$ `12`. Tělo `let`-bloku `y` se pak vyhodnotí na svou aktuální vazbu, tedy číslo `12`.

(d) Vyhodnocování výrazu `(let ((x 1) (x 2)) (* x 10))` skončí chybou „CHYBA: Vázáné symboly musí být vzájemně různé“.

Zopakujme, že výrazy $\langle hodnota_1 \rangle$, $\langle hodnota_2 \rangle$, ..., $\langle hodnota_n \rangle$ se vyhodnotí na elementy E_1, E_2, ..., E_n v *aktuálním prostředí*, až poté vznikají v novém prostředí vazby $\langle symbol_i \rangle \mapsto E_i$. To mimo jiné znamená, že tyto vazby nemohou ovlivnit vyhodnocovaní výrazů $\langle hodnota_1 \rangle$, ..., $\langle hodnota_n \rangle$. Za prvé, všechna vyhodnocení proběhnou dříve než vzniknou vazby. Za druhé, tato vyhodnocení proběhnou v prostředí, ze kterého na tyto vazby „nelze vidět“. Například v následujícím kódu

```
(define x 100)
(let ((x 10)
      (y (* 2 x)))
  (+ x y))  ⟹  210
```

máme `let`-výraz, ve kterém vážeme vyhodnocení výrazu `10` na symbol `x` a vyhodnocení výrazu `(* 2 x)` na symbol `y`. Přitom se výraz `(* 2 x)` vyhodnotí v globálním prostředí $\mathcal{P}_G$, kde je na `x` navázáno číslo `100` (viz obrázek 3.2), a výsledkem tohoto vyhodnocení bude číslo `100`. Proto je výsledkem vyhodnocení celého `let`-výrazu číslo `210`, nikoli číslo `30`.

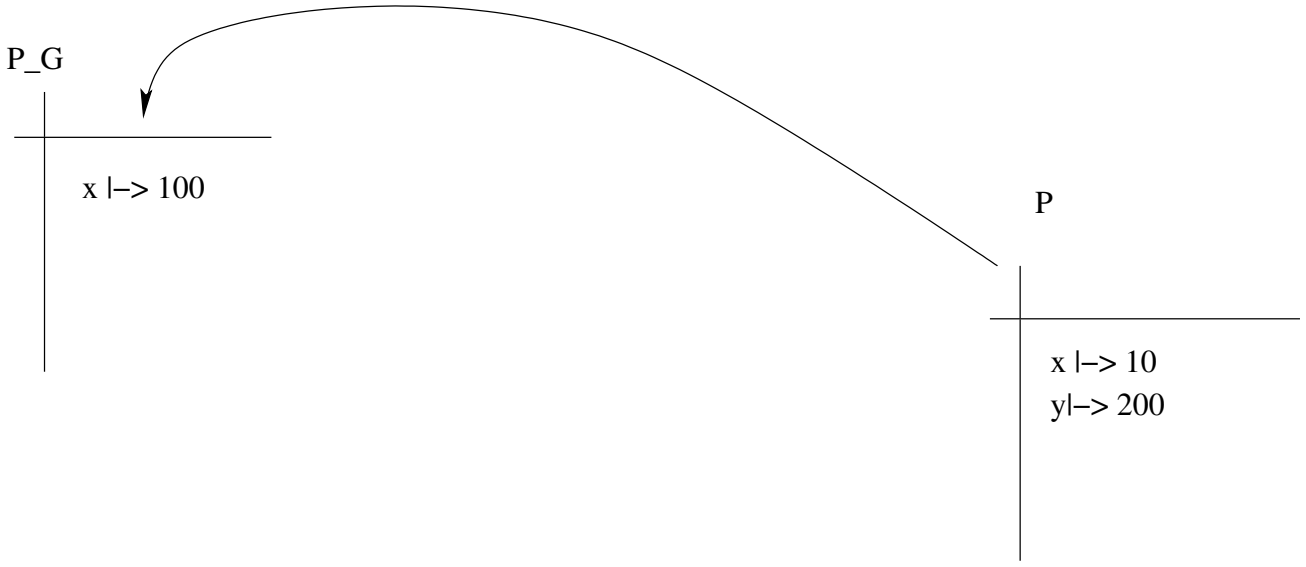
Ke stejnému závěru dospějeme po rozepsání `let`-bloku pomocí λ-výrazu:

```
(define x 100)
((lambda (x y) (+ x y)) 10 (* 2 x))
```

Příklad 3.4. Další ukázka představuje použití speciální formy `let` k vytvoření prostředí, kde jsou zaměněny vazby symbolů `+` a `*`. Jelikož jsou oba tyto symboly vyhodnoceny v globálním prostředí a až poté vznikají vazby, nemohou se tato vyhodnocení vzájemně ovlivnit.

```
(let ((+ *)
      (* +))
  (+ (* 10 10) 20))  ⟹  400
```

Obrázek 3.2. Vznik prostředí během vyhodnocení programu (NACRT OBRAZKU)



Všimněte si, že kdybychom k podobné záměně chtěli dospět pomocí speciální formy `define` museli bychom použít nějaký pomocný symbol pro uchování jedné z vazeb. Například takto:

```
(define plus +)
(define + *)
(define * plus)
```

Předchozí sekvence používající `define` ale *neprovádí souběžnou záměnu vazeb* symbolů `+` a `*`.

Vraťme se nyní k sekci 2.6 – tam jsme si vysvětlili rozdíl mezi *lexikálním rozsahem platnosti symbolů* a *dynamickým rozsahem platnosti symbolů*. Také jsme uvedli jednoduchý program, který při vyhodnocení při lexikálním typu platnosti symbolů dával jiný výsledek než při dynamickém a na kterém tak byla vidět odlišnost obou typů rozsahů platnosti, viz příklad 2.18 na straně 61. Teď demonstrujme tuto odlišnost na jiném programu. Uvažujme, následující definici v globálním prostředí:

```
(define na2 (lambda (x) (* x x)))
```

a podívejme se na vyhodnocení následujícího výrazu:

```
(let ((+ *)
      (* +))
  (na2 10))
```

`let`-blok nám vytváří lokální prostředí, ve kterém jsou vzájemně zaměněny vazby symbolů `+` a `*` stejně jako v předchozím příkladu. V těle `let`-bloku `(na2 10)` se však ani jeden z těchto symbolů nevyskytuje. Při lexikálním rozsahu platnosti dostáváme výsledek `100`. To je velice přirozené, protože uvedené vazby se skutečně (z důvodu absence vázaných symbolů v těle) jeví jako zbytečné.

Nyní se podívejme na tento program při dynamickém rozsahu platnosti. V tomto případě dojde k nepříjemnému efektu, který byl popsán už v příkladě 2.18 na straně 61. A to k závislosti chování procedury na prostředí, ve kterém je aplikovaná. Při aplikaci procedury `na2` vyhodnocujeme její tělo v lokálním prostředí, jehož dynamickým předchůdcem je právě lokální prostředí vytvořené `let`-blokem. Vazba na symbol `*` není v prostředí procedury `na2` nalezena a je tedy hledána v tomto předchůdci. Tam je `*` navázán na proceduru sčítání čísel. Proto v případě dynamického rozsahu platnosti dostáváme výsledek `20`. Dostali jsme tedy opětně jiný výsledek než při lexikálním rozsahu platnosti.

Příklad 3.5. Další příklad, který si ukážeme, představuje použití speciální formy `let` k zapamatování si lexikální vazbu symbolu, která bude posléze globálně předefinována. V tomto příkladu chceme navázat na

symbol `<=` proceduru porovnávající absolutní hodnoty dvou čísel. Absolutní hodnoty ale chceme porovnávat pomocí procedury, která je v globálním prostředí navázána právě na symbol `<=`. Řešení využívající `let`-blok vypadá takto:

```
(define <=
  (let ((<= <=))
    (lambda (x y)
      (<= (abs x) (abs y)))))
```

Pomocí speciální formy `let` vytvoříme nové lokální prostředí $\mathcal{P}$. V tomto novém prostředí bude na symbol `<=` navázáno vyhodnocení symbolu `<=` v aktuálním (tedy globálním) prostředí $\mathcal{P}_G$. Tam má symbol `<=` vazbu na proceduru porovnávání čísel. Jinými slovy, v lokálním prostředí vznikne stejná vazba na symbol `<=` jako je v globálním prostředí. Tělem tohoto `let`-bloku je λ-výraz `(lambda (x y) (<= (abs x) (abs y)))`. Ten bude vyhodnocen v tomto novém prostředí $\mathcal{P}$ a tedy $\mathcal{P}$ se stane prostředím vzniku procedury

$$\langle \texttt{(x y)}, \texttt{(<= (abs x) (abs y))}, \mathcal{P} \rangle.$$

Ta pak bude navázána v globálním prostředí aplikací speciální formy `define` na symbol `<=`. Situace je schématicky zobrazena v obrázku 3.3. Rozdíl oproti původní proceduře porovnávání čísel je zřejmý:

```
(<= 10 20)     ⟹  #t        (<= 20 10)     ⟹  #f
(<= -10 20)    ⟹  #t        (<= -20 10)    ⟹  #f
(<= 10 -20)    ⟹  #t        (<= 20 -10)    ⟹  #f
(<= -10 -20)   ⟹  #t        (<= -20 -10)   ⟹  #f
```

Při aplikaci naší nové procedury je vytvořeno nové lokální prostředí $\mathcal{P}_l$, jsou v něm vytvořeny vazby na symboly `x` a `y`. A v něm bude vyhodnoceno tělo `(<= (abs x) (abs y))`. Vazba symbolu `<=` nebude nalezena v lokálním prostředí, takže bude hledána v prostředí předka, jímž je prostředí $\mathcal{P}$, jenž bylo vytvořeno během vyhodnocování `let`-bloku. V tomto prostředí má symbol `<=` vazbu na primitivní proceduru porovnávání čísel. To jest při vyhodnocení těla `(<= (abs x) (abs y))` skutečně dojde k porovnání dvou absolutních hodnot. Téhož efektu bychom dosáhli následujícím kódem, který je možná přehlednější, protože jsem v něm použili nový symbol jiného jména (odlišného od `<=`). Ale, jak jsme si už ukázali, zavádet nový symbol není nutné.

```
(define <=
  (let ((mensi-nez <=))
    (lambda (x y)
      (mensi-nez (abs x) (abs y)))))
```

Na závěr tohoto příkladu si ještě ukážeme, že následující program by k požadovanému cíli nevedl.

```
(define <=
  (lambda (x y)
    (<= (abs x) (abs y))))
```

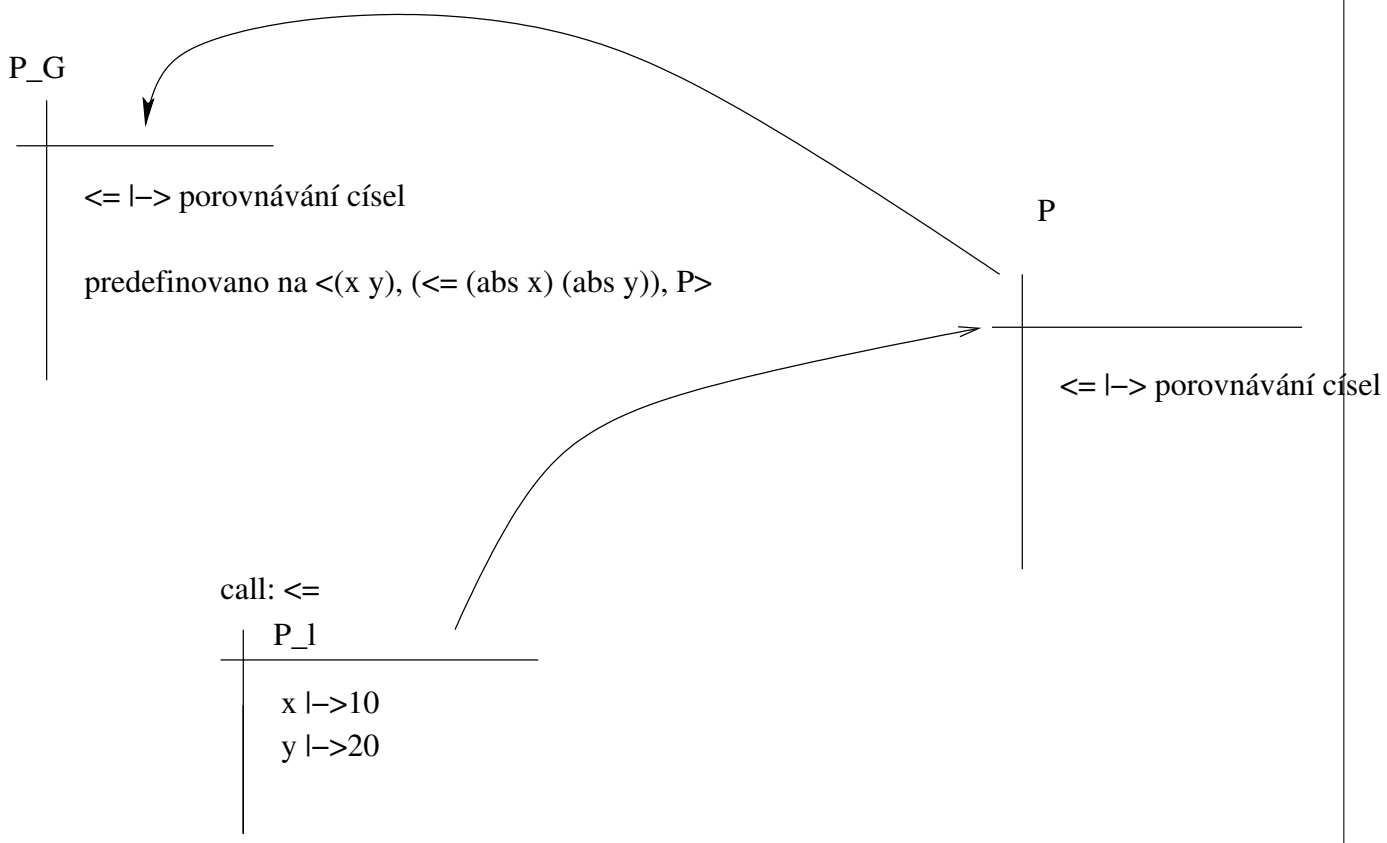
Při aplikaci takto nadefinované procedury vniká nové lokální prostředí, jehož lexikální předchůdce je prostředí jejího vzniku – tedy globální prostředí $\mathcal{P}_G$. Při vyhodnocování těla procedury dojde k vyhodnocení symbolu `<=`. Vazba na něj je nalezena v $\mathcal{P}_G$ a při vyhodocování těla je tedy opět aplikována stejná procedura. Dochází tedy k nekonečící sérii aplikací procedury `<=` a výpočet tak nikdy neskončí.

Jak už jsme několikrát řekli, vyhodnocování jednotlivých výrazů $\langle hodnota_i \rangle$ určujících hodnoty vazeb v `let`-bloku se navzájem neovlivňují. Někdy ale nastávají situace, kdy jistý typ ovlivnění požadujeme. Můžeme toho dosáhnout třeba postupným vnořováním `let`-bloků:

```
(let ((x 10))
  (let ((y (* x x)))
    (let ((z (- y x)))
      (/ z y))))  ⟹  9/10
```

Nebo můžeme použít další speciální formu `let*`:

Obrázek 3.3. Vznik prostředí během vyhodnocení programu z příkladu 3.5 (NACRT OBRAZKU)




```
(let* ((x 10)
       (y (* x x))
       (z (- y x)))
  (/ z y))  ⟹  9/10
```

Aplikaci této speciální formy si můžeme představit takto. Forma `let*` se chová analogicky jako `let`, ale při jejím použití se výrazy $\langle hodnota_1 \rangle, \langle hodnota_2 \rangle, \ldots, \langle hodnota_n \rangle$ nevyhodnocují v nespecifikovaném pořadí a vazby se neprovádějí „současně“. Naopak vyhodnocování výrazů (i vznik vazeb) se provádí „postupně“ v pořadí v jakém jsou uvedeny. Přitom vyhodnocení každého výrazu $\langle hodnota_i \rangle$ probíhá v takovém okamžiku a v takovém prostředí, že vazby symbolů $\langle symbol_j \rangle$ na vyhodnocení $\langle hodnota_j \rangle$ pro $j < i$ jsou již „viditelné“.

Definice 3.6. Syntaxe speciální formy `let*` je stejná, jako u speciální formy `let` (až na jména symbolů):

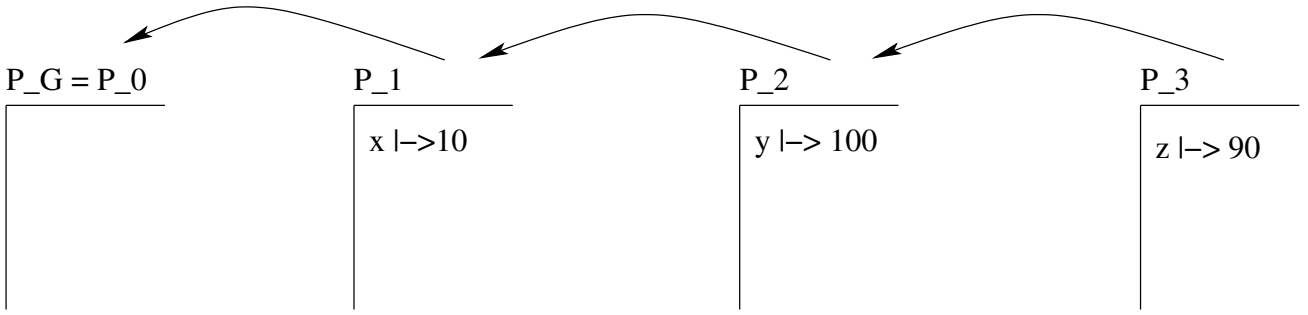
```
(let* ((⟨symbol_1⟩ ⟨hodnota_1⟩)
       (⟨symbol_2⟩ ⟨hodnota_2⟩)
           ⋮
       (⟨symbol_n⟩ ⟨hodnota_n⟩))
  ⟨tělo⟩),
```

kde n je nezáporné celé číslo, $\langle symbol_1 \rangle, \ldots, \langle symbol_n \rangle$ jsou symboly a $\langle hodnota_1 \rangle, \ldots, \langle hodnota_n \rangle$, $\langle tělo \rangle$ jsou libovolné výrazy. Tento výraz budeme nazývat `let*`-*blokem*.

Aplikaci speciální formy `let*` zavedeme pomocí speciální formy `let`.

- Jestliže $n = 0$ nebo $n = 1$, je vyhodnocení stejné, jako u speciální formy `let`.
- Jinak je vyhodnocení stejné jako vyhodnocení následujícího výrazu:

Obrázek 3.4. Hierarchie prostředí (NACRT OBRAZKU)




```
(let ((⟨symbol_1⟩ ⟨hodnota_1⟩))
  (let* ((⟨symbol_2⟩ ⟨hodnota_2⟩))
              ⋮
        ((⟨symbol_n⟩ ⟨hodnota_n⟩)))
    ⟨tělo⟩))
```

■

Během vyhodnocování `let*`-bloku tedy nevzniká jen jedno prostředí, jako v případě `let`-bloku, nýbrž celá hierarchie prostředí. Je postupně vytvářeno nové prostředí $\mathcal{P}_i$ pro každou dvojici `(`$\langle symbol_i \rangle$ $\langle hodnota_i \rangle$`)`, kde $i = 1, \ldots, n$. Označme aktuální prostředí jako $\mathcal{P}_0$. Předchůdcem prostředí $\mathcal{P}_i$ je prostředí $\mathcal{P}_{i-1}$, pro $i = 1, \ldots, n$. Při vzniku každého z prostředí $\mathcal{P}_i$ je do něj vložena nová vazba symbol $\langle symbol_i \rangle$ na vyhodnocení výrazu $\langle hodnota_i \rangle$ v prostředí $\mathcal{P}_{i-1}$. Poté je nastaven jeho lexikální předek na $\mathcal{P}_{i-1}$. Například prostředí vytvořená vyhodnocením následujícího `let*`-bloku budou vypadat tak, jak je to znázorněno na obrázku 3.4.

```
(let* ((x 10)
       (y (* x x))
       (z (- y x)))
  (/ z y))  ⟹  9/10
```

Hierarchii prostředí si také dobře uvědomíme, pokud předchozí `let*`-blok rozepíšeme pomocí `let`-bloků a ty dále rozepíšeme pomocí λ-výrazů:

```
((lambda (x)
   ((lambda (y)
      ((lambda (z)
         (/ z y))
       (- y x)))
    (* x x)))
 10)  ⟹  9/10
```

Poznámka 3.7. (a) Všimněte si, že narozdíl od `let`-bloku, nemusejí být symboly $\langle symbol_i \rangle$ vzájemně různé. Třeba výraz ve tvaru `(let* ((x 10) (x 20)) x)` je `let*`-blok.

(b) Sémantika `let*`-bloku by samozřejmě šla popsat, podobně jako u `let`-bloku, v bodech popisujících provádění jednotlivých vazeb:

(0) Označme aktuální prostředí $\mathcal{P}_0$, nastavme i na 1.

(1) Pokud $n = 0$, tedy seznam dvojic $(\langle symbol_i \rangle\ \langle vazba_i \rangle)$ je prázdný, je vytvořeno nové prázdné prostředí, jako jeho předchůdce je nastaveno aktuální prostředí $\mathcal{P}_0$ a v tomto prostředí je vyhodnoceno tělo `let*`-bloku. Tím aplikace speciální formy `let*` končí. V opačném případě $n > 0$ pokračujeme bodem (2).

(2) Pokud platí $i > n$ (to jest, už jsme prošli všechny dvojice `(`$\langle symbol_i \rangle$ $\langle hodnota_i \rangle$`)`), je v prostředí $\mathcal{P}_i$ vyhodnoceno tělo `let*`-bloku. Tím aplikace `let*` končí. V opačném případě ($i \leq n$) pokračujeme bodem (3).

(3) Vyhodnotí se výraz $\langle hodnota_i \rangle$ v prostředí $\mathcal{P}_{i-1}$, výsledek vyhodnocení označme E_i. Dále pokračujeme bodem (4).

(4) Vytvoří se nové prázdné prostředí $\mathcal{P}_i$, je do něj vložena vazba $\langle symbol_i \rangle \mapsto E_i$, předek prostředí $\mathcal{P}_i$ je nastaven na $\mathcal{P}_{i-1}$. Pokračujeme bodem (5).

(5) Zvýšíme i o 1 a pokračujeme bodem (2).

(c) Při aplikaci speciální formy `let*` vzniká tolik prostředí, kolik je dvojic $(\langle symbol_i \rangle\ \langle vazba_i \rangle)$. Zvláštním případem je, pokud je tento počet nulový. V takovém případě vzniká jedno prostředí.

Jak vyplývá z definic 3.1 a 3.6 jsou speciální formy představené `let` a `let*` v této sekci „nadbytečné“. Jejich sémantiku jsme totiž schopni vyjádřit pomocí speciální formy `lambda`. Použití forem `let` a `let*` ale umožňuje přehlednější zápis kódu. Větší přehlednosti je dosaženo především tím, že symboly vázané `let`-blokem (popřípadě `let*`-blokem) jsou uvedeny hned vedle hodnot, na které jsou navázány.

Příklad 3.8. (a) Uvažujme následující `let*`-bloky a jejich vyhodnocení:

```
(let* ((a (+ 5 5))) (+ a 2))   ⟹  12
(let* ((+ (+ 10 20))) +)       ⟹  30
```

To jest výsledkem vyhodnocení `let*`-bloků jsou čísla `12` a `30` (v tomto pořadí). Jejich vyhodnocení je tedy stejné jako v případě speciální formy `let` – to bylo podrobně rozepsáno v příkladě 3.3 v bodech (a) a (b).

(b) Vyhodnocení následujícího `let*` výrazu už bude odlišné.

```
(let* ((x 10)
       (y (* 2 x)))
  (+ x y))  ⟹  30
```

Zatímco aplikace speciální formy `let` by skončila chybou „CHYBA: Symbol `x` nemá vazbu“, aplikace formy `let*` proběhne následovně: v globálním prostředí je vyhodnocen výraz `10` na hodnotu `10`. Pak je vytvořeno nové prázdné prostředí $\mathcal{P}_1$, do něhož je přidána vazba na tento element, tedy vazba $\texttt{x} \mapsto_{\mathcal{P}_1} \texttt{10}$. Jako lexikální předek tohoto prostředí je nastaveno globální prostředí. V tomto prostředí je vyhodnocen výraz `(* 2 x)` na hodnotu `20`. Je vytvořeno další prázdné prostředí $\mathcal{P}_2$ a do něj je vložena vazba $\texttt{y} \mapsto_{\mathcal{P}_2} \texttt{20}$. Předkem tohoto prostředí bude $\mathcal{P}_1$. V prostředí $\mathcal{P}_2$ je konečně vyhodnoceno tělo `let*`-výrazu `(+ x y)`, na výslednou hodnotu `30`. K témuž bychom došli po rozepsání `let*`-bloku podle definice 3.6:

```
(let ((x 10))
  (let ((y (* 2 x)))
    (+ x y)))  ⟹  30
```

a k témuž dojdeme i při nahrazení `let`-bloků λ-výrazy:

```
((lambda (x)
   ((lambda (y)
      (+ x y))
    (* 2 x)))
 10)  ⟹  30
```

(c) Při použití speciální formy `let` by následující výraz nebyl `let`-blokem, protože vázané symboly použité v bloku nejsou vzájemně různé.

```
(let* ((x 10)
       (x (+ x 20)))
  (+ x 1))     ⟹  31
```

Definice `let*`-bloku ale připouští i stejné symboly, a proto je tento výraz `let*`-blokem.

3.2 Rozšíření λ-výrazů a lokální definice

Doteď jsme se zabývali jen tím jak vytvořit nové lokální prostředí, ne však tím, jak vytvářet nové vazby v již existujících lokálních prostředích ani tím, jak měnit vazby v existujících lokálních prostředích.

Ze sekce 1.6 již známe speciální formu `define`, jejíž aplikace má *vedlejší efekt*. Tímto vedlejším efektem je modifikace prostředí. Přesněji vyhodnocením `(define name` $\langle$*výraz*$\rangle$`)` v prostředí $\mathcal{P}$ je do prostředí $\mathcal{P}$ vložena vazba symbolu `name` na výsledek vyhodnocení argumentu $\langle$*výraz*$\rangle$: `name` $\mapsto$ Eval($\langle$*výraz*$\rangle, \mathcal{P}$) Dosud jsme ale tuto speciální formu používali jen k modifikaci globálního prostředí. Nyní si ukážeme, jak modifikovat i lokální prostředí. Abychom toho dosáhli, potřebujeme aplikovat speciální formu `define` v tom prostředí, které chceme modifikovat. Jinými slovy, potřebujeme vyhodnotit definici v těle procedury. Dále musíme zajistit, aby tato define byla vyhodnocena dříve, než část kódu, ve které chceme vzniklou vazbu využívat. Tento druhý bod je tím, co dělá lokální definice netriviální. Uvědomme si totiž, že nám principiálně nic nebrání vytvořit například proceduru vzniklou vyhodnocením následujícího λ-výrazu:

```
(lambda (x) (define y 20))
```

Při aplikaci této procedury vznikne lokální prostředí $\mathcal{P}$ v němž bude formální argument `x` navázaný na předanou hodnotu. Při vyhodnocení těla procedury v tomto lokálním prostředí dojte k aplikaci `define` a zavedení nové vazby `y` $\mapsto_{\mathcal{P}}$ `20` v lokálním prostředí. Tato vazba ale není nijak užitečná, protože výsledkem vyhodnocení těla (to jeste speciální formy `define`) je *nedefinovaná hodnota*, která je vrácena jako výsledek aplikace celé procedury. Musíme tedy zajistit vytvoření lokální definice a možnost jejího dalšího netriviálního použití.

K tomuto účelu můžeme například použít speciální formu `lambda`. Viz následující program:

```
(lambda ()
  ((lambda (nepouzity-symbol)
     (+ 10 x))
   (define x 20)))
```

Vyhodnocením tohoto λ-výrazu, vznikne procedura. Při aplikaci této procedury je v jejím lokálním prostředí $\mathcal{P}$ vyhodnoceno tělo, tedy výraz `((lambda (ingorovany-symbol) (+ x 10)) (define x 20))`. Vyhodnocením prvního prvku tohoto seznamu dostáváme proceduru, která ignoruje svůj jediný argument a vrací součet čísla `10` a hodnoty navázané na symbol `x`. Prvním prvkem seznamu (těla aplikované procedury) je tedy procedura, vyhodnotí se tedy zbývající prvek seznamu: seznam `(define x 20)`. Jako vedlejší efekt vyhodnocení tohoto výrazu je vytvoření vazby symbolu `x` $\mapsto_{\mathcal{P}}$ `20`. V okamžiku vyhodnocení těla `(+ 10 x)` vnitřního λ-výrazu je tedy vazba na `x` už zavedena.

Jiná možnost je využití aplikace speciální formy `and`:

```
(lambda ()
  (and (define x 10)
       (define y (- x 2))
       (define z (/ x y))
       (+ x y z)))
```

Speciální forma `and`, jak bylo popsáno v definici 2.22, vyhodnocuje své argumenty v pořadí, v jakém jsou uvedeny – zleva doprava. Jelikož výsledkem aplikace speciální formy `define` je *nedefinovaná hodnota*, tedy element různý od `#f`, jsou postupně vyhodnoceny všechny tři definice i výraz `(+ x y z)`. A jelikož se jedná o poslední argument formy `and`, je výsledek vyhodnocení výrazu `(+ x y z)` také výsledkem aplikace formy `and`.

Například proceduru `dostrel` bychom mohli napsat tak, jak je uvedeno v programu 3.1. Ve skutečnosti nemusíme použití definic v těle λ-výrazu řešit takovou oklikou, jako je využití jiné speciální formy. Stačí když uděláme následující změny: Rozšíříme λ-výraz tak, aby se jeho tělo mohlo skládat z více než jednoho výrazu a stejným způsobem upravíme definici procedury. Dále upravíme popis aplikace speciální formy `lambda` a aplikaci uživatelsky definovatelné procedury. Pak můžeme proceduru `dostrel` definovat tak, jak je ukázáno v programu 3.2.

Program 3.1. Procedura `dostrel` s lokálními vazbami vytvářenými s využitím `and`.

```
(define dostrel
  (lambda (v0 alfa g)
    (and (define vx (* v0 (cos alfa)))
         (define vy (* v0 (sin alfa)))
         (define Td (* 2 vy (/ g)))
         (* vx Td))))
```

Program 3.2. Procedura `dostrel` s lokálními vazbami vytvářenými pomocí speciální formy `define`.

```
(define dostrel
  (lambda (v0 alfa g)
    (define vx (* v0 (cos alfa)))
    (define vy (* v0 (sin alfa)))
    (define Td (* 2 vy (/ g)))
    (* vx Td)))
```

Nyní rozebereme uvedené změny podrobněji. V definici 2.1 jsme zavedli tělo λ-výrazu jako jeden libovolný S-výraz. Od tohoto okamžiku budeme uvažovat, že se tělo λ-výrazu bude skládat z libovolného nenulového počtu symbolických výrazů:

Definice 3.9 (λ-výraz). Každý seznam ve tvaru

$$\texttt{(lambda (}\langle param_1\rangle\ \langle param_2\rangle \cdots \langle param_n\rangle\texttt{)}\ \ \langle výraz_1\rangle\ \ \langle výraz_2\rangle \cdots \langle výraz_m\rangle\texttt{)},$$

kde n je nezáporné číslo, m je kladné číslo, $\langle param_1\rangle, \langle param_2\rangle, \ldots, \langle param_n\rangle$ jsou vzájemně různé symboly a $\langle výraz_1\rangle, \langle výraz_2\rangle, \ldots, \langle výraz_m\rangle$ jsou symbolické výrazy, tvořící *tělo*, se nazývá *λ-výraz* (*lambda výraz*). Symboly $\langle param_1\rangle, \ldots, \langle param_n\rangle$ se nazývají *formální argumenty* (někdy též *parametry*). Číslo n nazýváme *počet formálních argumentů* (*parametrů*). ■

Vyhodnocením v prostředí $\mathcal{P}$ tohoto výrazu vznikne uživatelsky definovaná procedura reprezentovaná trojicí $\langle(\langle param_1\rangle\ \langle param_2\rangle \cdots \langle param_n\rangle); \langle výraz_1\rangle, \langle výraz_2\rangle, \ldots, \langle výraz_m\rangle; \mathcal{P}\rangle$ stejně tak, jak je popsáno v předchozí lekci. Až na ten detail, že tělo zde není interní reprezentace jednoho S-výrazu, ale jednoho nebo více S-výrazů $\langle výraz_i\rangle$.

Aplikace takové procedury je totožná s aplikací popsanou v definici 2.12, a to až na bod 5. Zde se vyhodnotí všechny S-výrazy, z nichž se skládá tělo procedury, v prostředí $\mathcal{P}_l$. Vyhodnocují se postupně v tom pořadí, v jakém jsou v těle procedury uvedeny. Výsledkem je pak vyhodnocení *posledního z nich*.

Všimněte si, že výsledky vyhodnocení S-výrazů v těle – až na poslední z nich – jsou „zapomenuty“. Předpokládá se, že při vyhodnocení těchto S-výrazů dojde k nějakému vedlejšímu efektu, jako je například modifikace aktuálního prostředí při aplikaci speciální formy `define`. Takto užité definice nazýváme *interní* (též *vnitřní*) *definice*. Definice v globálním prostředí nazýváme pro odlišení *globálními* (též *top-level*) *definicemi*.

Příklad 3.10. Nyní si to, co jsme v této sekci řekli, ukážeme na vyhodnocení následujícího kódu:

```
(define proc
  (lambda (x)
    (define y 20)
    (+ x y)
    (* x y)))

(proc 10)  ⟹  200
```

Jedná se o aplikaci procedury, jejíž tělo je tvořeno třemi výrazy: `(define y 20)`, `(+ x y)` a `(* x y)`. Vytvoří se tedy nové lokální prostředí $\mathcal{P}$, v němž bude vytvořena vazba `x` $\mapsto$ `10` a jehož předchůdce bude nastaven na globální prostředí $\mathcal{P}_G$. V tomto lokálním prostředí se budou vyhodnocovat všechny výrazy z těla. Vyhodnocují zleva doprava, tak jak jsou uvedeny – nejdříve se tedy vyhodnotí interní definice a v lokálním prostředí $\mathcal{P}$ procedury bude provedena vazba `y` $\mapsto$ `20`. Dále se vyhodnocuje výraz `(+ x y)`, jeho výsledkem bude číslo `30`, ale protože se nejedná o poslední výraz z těla je tento výsledek ignorován a pokračujeme dalším – posledním výrazem `(* x y)`. Výsledkem vyhodnocení tohoto výrazu je číslo `200`, a protože se jedná o poslední výraz těla procedury, je toto číslo i výsledkem Eval[`(proc 10)`, $\mathcal{P}_G$]. Všimněte si, že výraz `(+ x y)` nemá žádný význam. Jeho výsledek je zapomenut a přitom nemá ani žádný vedlejší efekt. Vazby vytvořené interními definicemi samozřejmě nejsou viditelné zvenčí. Kdybychom se pokusili vyhodnotit v globálním prostředí třeba výraz `(+ y 1)`, skončili bychom chybou „CHYBA: Symbol `y` nemá vazbu“.

Poznámka 3.11. (a) Týmž způsobem, jakým jsme právě rozšířili definici λ-výrazu, rozšiřujeme i definici `let`-bloku a `let*`-bloku. Tedy ⟨*tělo*⟩ `let`-bloku i `let*`-bloku se může skládat z více než jednoho S-výrazu. Rozšíření `let`-bloku a `let*`-bloku je vlastně automatické, protože jejich sémantiku jsme popisovali přes λ--výrazy a aplikaci procedur vzniklých jejich vyhodnocováním.

(b) Specifikace R^{5}RS jazyka Scheme – viz [R5RS] – říká, že interní definice se mohou objevit na začátku těla λ-výrazu (popř. `let`-bloku, let*-výrazu.) To znamená, že například vyhodnocování λ-výrazu

```
(lambda (x) (+ x 1) (define y 2) (+ x y))
```

v některých interpretech může skončit chybou „CHYBA: Speciální forma `define` je použita ve špatném kontextu“. V našem abstratním interpretru se tímto nebudeme omezovat a budeme připouštět i interní definice na jiném místě v těle λ-výrazu. Stejně tak je připouštějí například interprety Elk a Bigloo. Nepovoluje je třeba MIT Scheme.

(c) Některé interpretry dokonce mimo omezení popsané v (b) neumožňují v jednom těle definovat vazbu na symbol a pak ji pomocí další interní definice změnit. Příklady takových interpretů jsou MIT Scheme a Guile. Třeba vyhodnocování výrazu

```
(lambda () (define x 10) (define y 10) (define x 20) y)
```

by skončilo chybou.

Příklad 3.12. (a) Podívejme se na program 3.2 – tedy implementaci procedury `dostrel` s použitím interních definic. Vyhodnocení tohoto kódu bude mít za efekt navázání nové procedury na symbol `dostrel`. Při vyvolání této procedury vznikne nové prostředí a v něm vazby na formální argumenty `v0`, `alfa` a `g`. V tomto prostředí se postupně vyhodnocují všechny výrazy v těle. Vedlejším efektem jsou postupně do lokálního prostředí přidány vazby na symboly `vx`, `vy` a `Td`. Důležité je, že třetí vnitřní definice se vyhodnocuje až po vyhodnocení druhé. V té době už tedy existuje vazba na symbol `vy`. Výsledek vyhodnocení posledního výrazu v těle `(* vx Td)` je pak výsledkem aplikace procedury.

(b) Následující `let`-blok se vyhodnotí na číslo `30`:

```
(let ()
  (define x (lambda () y))
  (define y 10)
  (x))   ⟹   30
```

Do prázdného prostředí vytvořeného při aplikaci `let`-bloku, je vyhodnocením vnitřní definice

```
(define x (lambda () y))
```

přidána vazba `x` na proceduru. Tato procedura nebere žádný argument a vrací vždy vyhodnocení symbolu `y` a prostředím jejího vzniku je právě toto prostředí. Poté je do tohoto prostředí přidána vazba na symbol `y`. Při vyvolání procedury `x` v posledním výrazu v těle už je tedy symbol `y` navázán. Výsledkem je číslo `30`.

3.3 Příklady na použití lokálních vazeb a interních definic

Nyní ukážeme několik příkladů, v nichž je vhodné použít lokální vazby a definice. A to na jednoduché finanční aritmetice: Střádání – na počátku každého úrokovacího období se pravidelně ukládá částka a a na konci období se k úsporám připisuje úrok ve výši $p\%$ úspor. Po n obdobích vzroste vklad na částku a_n danou

$$a_n = ar\frac{r^n - 1}{r - 1}, \text{ kde } r = \left(1 + \frac{p}{100}\right).$$

Následující program vypočítá hodnotu a_n s využitím lokální vazby vytovřené v `let`-bloku:

```
(define sporeni
  (lambda (a n p)
    (let ((r (+ 1 (/ p 100.0))))
      (/ (* a r (- (expt r n) 1)) (- r 1)))))
```

Při aplikaci procedury se vytvoří lokální prostředí s vazbami na formální argumenty `a`, `n` a `p`. V uvedeném vzorci se nám ale vyskytuje několikrát r, které musíme dopočítat. Místo toho, abychom zbytečně psali na každém místě místo r výraz `(+ 1 (/ p 100.0))`, vytvořili jsme vazbu symbolu `r` na vyhodnocení tohoto výrazu. Tím jsme odstranili redundanci (stejný výraz by se nám zde vyskytoval třikrát). Vazba na symbol `r` vzniká v novém lokálním prostředí vyhodnocením `let`-bloku. A v tomto prostředí je také vyhodnoceno tělo procedury.

Lokální vazbu bychom mohli vytvořit i pomocí interní definice:

```
(define sporeni
  (lambda (a n p)
    (define r (+ 1 (/ p 100.)))
    (/ (* a r (- (expt r n) 1)) (- r 1))))
```

Toto řešení je také správné. V tomto případě ale nevzniká nové prostředí. Aplikací formy `define` Je modifikováno aktuální prostředí – tedy prostředí procedury `sporeni`. K vazbám vytvořených navázáním argumentů tak přibude nová vazba na symbol `r`.

Proceduru `sporeni` je vhodné upravit na proceduru vyššího řádu. Takto upravenou procedurou je pak možné vytvářet procedury na výpočet úspor při různém úročení.

```
(define sporeni
  (lambda (p)
    (let ((r (+ 1 (/ p 100))))
      (lambda (a n)
        (/ (* a r (- (expt r n) 1)) (- r 1))))))
```

Pomocí lokálních definic můžeme také upravit program 2.7 na straně 61, kde jsme implementovali proceduru `derivace`, která jako výsledek vracela přibližnou derivaci. Tato procedura používala pomocnou proceduru `smernice`. Tento program můžeme upravit následujícím způsobem. Z definice procedury `smernice` uděláme interní definici.

```
(define derivace
  (lambda (f delta)

    (define smernice
      (lambda (f a b)
        (/ (- (f b) (f a))
           (- b a))))

    (lambda (x)
      (smernice f x (+ x delta)))))
```

Kód můžeme ještě začistit, a to následujícím způsobem: parametr `f`, nemusíme předávat proceduře `smernice`. Ta totiž vzniká v prostředí, ve kterém je symbol `f` už navázán, to jest v prostředí procedury `derivace`, viz program 3.3.

Program 3.3. Procedura `derivace` s použitím interní definice.

```
(define derivace
  (lambda (f delta)

    (define smernice
      (lambda (a b)
        (/ (- (f b) (f a))
           (- b a))))

    (lambda (x)
      (smernice x (+ x delta)))))
```

Pro úplnost uvádíme variantu pomocí `let`. Viz program 3.4

Program 3.4. Procedura `derivace` s použitím speciální formy `let`.

```
(define derivace
  (lambda (f delta)
    (let ((smernice
           (lambda (a b)
             (/ (- (f b) (f a))
                (- b a)))))
      (lambda (x)
        (smernice x (+ x delta))))))
```

V mnoha případech se vyplatí vytvářet pomocné procedury v těle hlavních procedur. Často tak lze snížit celkový počet předávaných argumentů. To díky tomu, že elementy, s kterými pracuje pomocná procedura, mohou být navázány symbol v prostředí vzniku této procedury. Pomocná procedura tedy může některé symboly vázat ve svém prostředí a některé brát z prostředí nadřazeného, to jest prostředí hlavní procedury. Snížením počtu předávaných argumentů dochází k zpřehlednění kódu.

Další ukázkou bude procedura na výpočet nákladů vydaných za určitý počet kusů nějakého výrobku (`pocet-kusu`), který má nějakou cenu (`cena-kus`). V nejprimitivnější formě vypadá implementace takto:

```
(define naklady
  (lambda (cena-kus pocet-kusu)
    (* cena-kus pocet-kusu)))
```

Nyní zahrneme nový fakt. A to, že při zakoupení daného množství kusů (`sleva-kus`) dostaneme množstevní slevu (`sleva-%`):

```
(define naklady
  (lambda (cena-kus pocet-kusu sleva-kus sleva-%)
    (- (* cena-kus pocet-kusu)
       (if (>= pocet-kusu sleva-kus)
           (* cena-kus pocet-kusu (/ sleva-% 100))
           0))))
```

Tedy od původní ceny, která je `(* cena-kus pocet-kusu)`, odečteme buďto nulu, nebo slevu o velikosti `(* cena-kus pocet-kusu (/ sleva-% 100))`. To v závislosti na tom, jestli počet kupovaných kusů překročil či nepřekročil množství potřebné na slevu: `(>= pocet-kusu sleva-kus)`.

Ačkoli se vlastně jedná o jednoduchou proceduru, její kód je docela nepřehledný. Navíc zde dvakrát počítáme součin `(* cena-kus pocet-kusu)`. Proto jej nahradíme následujícím kódem:

```
(define naklady
  (lambda (cena-kus pocet-kusu sleva-kus sleva-%)
    (let* ((bez-slevy (* cena-kus pocet-kusu))
           (sleva-mult (/ sleva-% 100))
           (sleva (* sleva-mult bez-slevy)))
      (if (>= pocet-kusu sleva-kus)
          (- bez-slevy sleva)
          bez-slevy))))
```

Použitím lokálních vazeb jsme kód zpřehlednili tím, že jsme hodnoty pojmenovali podle rolí, které mají. Také jsme odstranili redundanci v kódu.

3.4 Abstrakční bariéry založené na procedurách

V této sekci se nebudeme zabývat jazykem Scheme, ale zamyslíme se nad problémy souvisejícími s vytvářením velkých programů. Při vytváření větších programů je potřeba programové celky vhodně strukturovat a organizovat, aby se v nich programátoři vyznali. Při psaní jednoduchých programů, se kterými jsme se zatím setkali, tato potřeba není příliš markantní. Při programování velkých programů však rychle vzrůstá jejich složitost a hrozí postupné snižování čitelnosti programu a následné zanášení nechtěných chyb do programu vlivem neznalosti nebo nepochopení některých jeho částí.

Předchozího jevu si programátoři všimli záhy po vytvoření vyšších programovacích jazyků a po jejich nasazení do programátorské praxe. Velmi brzy se proto programátoři začali zamýšlet nad metodami strukturování programů do menších celků, které by byly snadno pochopitelné a bylo by je možné (do jisté míry) programovat a ladit samostatně.

Snad historicky nejstarší metoda granularizace větších programových celků se nazývá *top-down*. Tento styl vytváření programů vychází z toho, že si velký program nejprve představíme jako „celek“ a navrhneme jej rozdělit do několika samostatných částí (podprogramů). Každou z těchto částí opět prohlédneme a opět navrhneme její rozdělení. Takto pokračujeme dokud nedosáhneme požadovaného stupně „rozbití programu“ na mají celky. Tyto malé celky se potom obvykle programují samostatně (na jejich vytvoření může pracovat několik kolektivů programátorů). Pro dokončení všech celků se provede jejich spojení do výsledného programu.

Ve funkcionálních jazycích a zejména v dialektech LISPu, což jsou jedny z nejflexibilnějších programovacích jazyků, se však obvykle používá jiný přístup, který se nazývá *bottom-up*. V tomto případě je myšlenkový postup jakoby „obrácený“. Programátoři vytvářejí program po vrstvách. Nejnižší vrstvou programu je vždy samotný programovací jazyk (v našem případ jazyk Scheme). Nad ním je druhá vrstva, která obsahuje nově definované procedury řešící jistou třídu problémů. Tuto vrstvu si lze v podstatě představit jako rozšíření jazyka Scheme. Další vrstva bude tvořena jinými procedurami, které budou řešit další problémy a které již budou používat nejen primitivní procedury jazyka Scheme, ale i procedury vytvořené v předchozí vrstvě. Při programování ve stylu *bottom-up* je tedy zvykem postupně *obohacovat samotný programovací jazyk* o nové schopnosti a postupně tak dospět k dostatečně bohatému jazyku, ve kterém bude již snadné naprogramovat zamýšlený program. Pokud jsou procedury v jednotlivých vrstvách navrženy dostatečně abstraktně, změna jejich kódu nebo reimplementace celé jedné vrstvy programu (třeba kvůli efektivitě nebo kvůli změně zadání od uživatele) nečiní příliš velký problém.

S vytvářením programů metodou *bottom-up* souvisí dva důležité pojmy, z nichž první jsme již slyšeli.

černá skříňka Jakmile máme vytvořeny procedury v jedné vrstvě programu a začneme vytvářet další (vyšší) vrstvu, na procedury v nižší vrstvě bychom se správně měli dávat jako na „černé skříňky“. To jest,

neměli bychom se zabývat tím, jak jsou naprogramované, ale měli bychom je pouze používat na základě toho, jaké argumenty je jim možné předávat a na základě znalosti výsledků jejich aplikací. Tím, že odhlédneme od implementace procedur na nižší úrovni, budeme vytvářet kvalitnější procedury na vyšší úrovni, které nebudou zasahovat do nižší vrstvy. Když potom provedeme reimplementaci procedur na nižší vrstvě při zachování jejich rozhraní (argumentů a výstupů), pak se funkčnost procedur na vyšší vrstvě nezmění. Připomeňme, že pohled na procedury jako na černé skříňky není nic „umělého“. Při programování ve Scheme jsme se doposud dívali na všechny primitivní procedury (to jest na procedury na nejnižší vrstvě), jako na černé skříňky.

abstrakční bariéra je pomyslný mezník mezi dvěma vrstvami programu. Pokud vytváříme procedury na vyšší vrstvě programu, procedury nižších vrstev jsou pro nás „za abstrakční bariérou.“ Otázkou je, na jakých místech v programu je vhodné tyto bariéry „vytvářet“. Jinými slovy, otázkou je, kde od sebe oddělovat jednotlivé vrstvy programu. Odpověď na tuto otázku není jednoduchá a závisí na konkrétním problému, zkušenostech programátora a jeho představivosti. Dobrou zprávou ale je, že vrstvy v programu se mohou během života programu vyvíjet. Obzvlášť pro programování ve funkcionálních jazycích je typické, že se začne vytvářet program, který je pouze částečně navržený a během jeho vývoje se postupně segregují vrstvy tak, jak je to zrovna potřeba nebo podle toho, když se zjistí, že je to „praktické“. Taková situace může například nastat v momentě, když si programátoři všimnou, že několik procedur „vypadá podobně“. V takovém případě je vhodné zamyslet se nad jejich zobecněním, při kterém je možné například použít procedury vyšších řádů.

V tomto kursu se budeme zabývat základy programovacích stylů a až na výjimky budeme vše demonstrovat na malých programech, to jest programech majících maximálně desítky řádků. Výhody metody *bottom-up* bychom ocenili až při vytváření větších programů. Ve dvanácté lekci ukážeme implementaci interpretu funkcionální podmnožiny jazyka Scheme, která bude mít několik set řádků. To je z hlediska velikosti programů opět pouze malý program (za „velké programy“ se obvykle považují programy mající minimálně stovky tisíc řádků; za největší program současnosti je považován program pro řízení letu raketoplánu, která má údajně přes 150 milionů řádků). Na druhou stranu, tento program už bude „natolik velký“, že v něm budeme moci rozlišit několik přirozených vrstev a abstrakčních bariér.

Shrnutí

V této kapitole jsme se zabývali lokálními vazbami. Uvedli jsme důvody, proč je potřebujeme. Zopakovali jsme, jak vznikají při aplikaci uživatelsky definovaných procedur. Ukázali jsme si dvě speciální formy – `let` a `let*`. Uvedli jsme příklady použití těchto forem a také jsme ukázali, že obě tyto nové speciální formy jsou nahraditelné použitím speciální formy `lambda`. Dále jsme rozšířili λ-výrazy, tím že jsme umožnili jejich tělo sestávat z libovolného nenulového počtu výrazů. Ukázali jsme použití interních definic, tedy speciální formy `define` pro modifikaci lokálních prostředí.

Pojmy k zapamatování

- lokální vazba
- interní definice, top-level definice
- `let`-blok, `let*`-blok

Nově představené prvky jazyka Scheme

- Speciální formy `let` a `let*`

Kontrolní otázky

1. *Co jsou lokální vazby? K čemu jsou dobré?*
2. *Proč nemohou být* `let` *a* `let*` *procedury?*

3. *Jak se* `let`*-bloky přepisují na λ-výrazy?*
4. *Jak se na ně přepisují* `let*`*-bloky?*
5. *Jak probíhá aplikace formy* `let`*?*
6. *Jak probíhá aplikace formy* `let*`*?*
7. *Jak lze přepsat* `let*`*-blok pomocí formy* `let`*?*
8. *Jak jsme rozšířili λ-výrazy?*
9. *Co jsou interní definice, jak se liší od top-level definic?*

Cvičení

1. Bez použití interpretu určete výsledky vyhodnocení následujících výrazů:

```
(let () 10)                                                    ⟼
(let () x)                                                     ⟼
(let () (define x 5))                                          ⟼
(let () (define x 5) (* x x))                                  ⟼
(let ((x (+ 5 5))) (* x x))                                    ⟼
(let* ((x (+ 5 5))) (* x x))                                   ⟼
(let ((x (+ 5 5))) (* x x) (+ x x))                            ⟼
(let ((x 10)) (define x 5) (* x x))                            ⟼
(let* ((x 10)) (define x (+ x 1)) (* x x))                     ⟼
(let ((x 10)) (define x x) (* x x))                            ⟼

(let ((x 10) (y 20)) (+ x y))                                  ⟼
(let* ((x 10) (y 20)) (+ x y))                                 ⟼
(let ((x 10) (y x)) (* x y))                                   ⟼
(let* ((x 10) (y x)) (* x y))                                  ⟼
(let ((x 10) (y +)) (* x y))                                   ⟼
(let ((x 10) (y +)) (define y x) (* x y))                      ⟼
(let* ((x 10) (y +)) (define y x) (* x y))                     ⟼
(let* ((x 10) (y x)) (define y 3) (* x y))                     ⟼
(let ((x 10)) (define y 3) (* x y))                            ⟼
(let ((x 10) (y (lambda () x))) (+ x (y)))                     ⟼

(let* ((x 10) (y (lambda () x))) (+ x (y)))                    ⟼
(let* ((x 10) (y (lambda (x) x))) (+ x (y x)))                 ⟼
(let* ((x 10) (y (lambda () x)) (x 5)) (+ x (y)))              ⟼
(let* ((x 10) (y (lambda (x) x)) (x 5)) (+ x (y x)))           ⟼
(let ((x 10)) (define y (lambda () x)) (define x 5) (+ x (y))) ⟼
(let ((x 10)) (define y (lambda (x) x)) (define x 5) (+ x (y x))) ⟼
(let ((x 10)) (+ (let ((y (+ x 1))) (* y 2)) x))               ⟼
(let ((x 10) (x 20)) (+ x x))                                  ⟼
(let* ((x 10) (x 20)) (+ x x))                                 ⟼
(if #f (let ((x 10) (x 20)) (+ x x)) 10)                       ⟼
```

2. Přepište následující výrazy na ekvivalentní výrazy bez použití speciální forem `let` a `let*`.

```
• (let* ((x (+ 10 y))
         (y (let ((x 20)
                  (y 30))
              (* 2 (+ x y)))))
    (/ x y z))

• (let ((a 3)
        (b 4)
        (c (let* ((a 10)
                  (b 20))
```

```
                (+ a b 10))))
      (+ a b c))
```

- ```
  (let* ((x (let ((x 10))
              (define x 2)
              (+ x x)))
        (y (+ x x)))
    (+ x y))
  ```

3. Napište proceduru, která vypočte, v jaké výšce dosáhne těleso vržené vzhůru počáteční rychlostí v_0 rychlosti v (ve vakuu). Vstupními parametry budou v_0 a v. Použijte vzorce:

   $$t = \frac{v_0 - v}{g}, \quad s = v_0 t - \frac{1}{2} g t^2.$$

   Lokální vazbu pro t vytvořte jednou pomocí interní definice, podruhé pak pomocí speciální formy `let`. Tíhové zrychlení g definujte globálně.
4. Napište proceduru řešící tento typ slovních úloh: Objekt se pohybuje rovnoměrně zrychleným pohybem, s počáteční rychlostí v_0 se zrychlením a. Vypočtěte dráhu, po které dosáhne rychlosti v. Použijte vzorce:

   $$t = \frac{v - vo}{a}, \quad s = v_0 t + \frac{1}{2} a t^2.$$

   Lokální vazbu pro t vytvořte jednou pomocí interní definice, podruhé pomocí speciální formy `let`.
5. Předefinujte proceduru navázanou na symbol `+` na proceduru sčítání druhých mocnin dvou čísel.
6. Napište λ-výraz, který má ve svém těle více než jeden výraz a jehož přímočarý přepis na λ-výraz obsahující ve svém těle jeden výraz s požitím speciální formy `and` (viz například programy 3.1 a 3.2) se vyhodnotí na jinou proceduru. Jinou procedurou v tomto případě myslíme to, že aplikací na nějaký argument bude vracet jiný výsledek.

Úkoly k textu

1. Uvažujme speciální formu `let+`, která se používá ve stejném tvaru jako speciální forma `let*`:

   ```
   (let+ ((⟨symbol_1⟩ ⟨hodnota_1⟩)
          (⟨symbol_2⟩ ⟨hodnota_2⟩)
          ⋮
          (⟨symbol_n⟩ ⟨hodnota_n⟩))
     ⟨tělo⟩)
   ```

   ale přepisuje se na:

   ```
   (let ()
     (define ⟨symbol_1⟩ ⟨hodnota_1⟩)
     (define ⟨symbol_2⟩ ⟨hodnota_2⟩)
     ⋮
     (define ⟨symbol_n⟩ ⟨hodnota_n⟩)
    ⟨tělo⟩)
   ```

   Zamyslete se nad rozdílem oproti speciální formě `let*`. Napište výraz, který bude mít různé výsledky vyhodnocení při použití `let+` a `let*`.

2. Popište jak se dá nahradit `let*`-blok pomocí λ-výrazů.

3. Na začátku sekce 3.2 jsme uvedli, jak použít definice v těle λ-výrazů, kdybychom nezměnili definici λ-výrazu. Použili jsme přitom speciálních forem `lambda` a `and`. Popište, jak by se k tomu dalo využít jiných speciálních forem, např. `if` nebo `cond`.
   ```

Řešení ke cvičením

1. 10, chyba, nespecifikovaná hodnota, 25, 100, 100, 20, 25, 36, 100, 30, 30, chyba, 100, chyba, 100, 100, 30, 30, chyba, 20, 20, 15, 10, 10, 10, 32, chyba, 40, 10

2.
```
((lambda (x)
   ((lambda (y)
      (/ x y z))

    ((lambda (x y)
       (* 2 (+ x y)))
     20 30)))

   (+ 10 y))
```

- ```
  ((lambda (a b c)
     (+ a b c))
   3 4
   ((lambda (a)
      ((lambda (b)
         (+ a b 10))
       20))
    10))
  ```

- ```
  ((lambda (x)
     ((lambda (y)
        (+ x y))
      (+ x x)))

    ((lambda (x)
       (define x 2)
       (+ x x))
     10))
  ```

3. ```
   (define g 9.80665)

   (define vrh-vzhuru
     (lambda (v0 v)
       (let ((t (/ (- v0 v) g)))
         (- (* v0 t) (* 1/2 g t t)))))
   ```

   ```
   (define vrh-vzhuru
     (lambda (v0 v)
       (define t (/ (- v0 v) g))
       (- (* v0 t) (* 1/2 g t t))))
   ```

4. ```
   (define pohyb-rovnomerne-zrychleny
     (lambda (v0 v a)
       (let ((t (/ (- v0 v) a)))
         (+ (* v0 t) (* 1/2 a t t)))))
   ```

   ```
   (define pohyb-rovnomerne-zrychleny
     (lambda (v0 v a)
       (define t (/ (- v0 v) a))
       (+ (* v0 t) (* 1/2 a t t))))
   ```

5. ```
   (define +
     (let ((+ +))
       (lambda (x y)
         (+ (* x x) (* y y)))))
   ```

6. Například: `(lambda () #f 1)`
   ```