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

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

Paradigmata programování 1 ◇ poznámky k přednášce

2. Procedury a prostředí

1 Abstrakce pomocí procedur

Takto můžeme vypočítat obsah trojúhelníka o základně 4 a výšce 3:

```
> (* 1/2 4 3)
6
```

Obsah trojúhelníka o základně 6 a výšce 8 vypočítáme takto:

```
> (* 1/2 6 7)
21
```

Obecně, pokud v proměnných `b` a `h` máme základnu a výšku (*base* a *height*) trojúhelníka, pak jeho obsah můžeme vypočítat takto:

```
> (* 1/2 b h)
```

(Výsledek vyjde v závislosti na hodnotách proměnných `b` a `h`.)

To je použití Scheme jako kalkulačky: zadáváme mu pouze čísla a základní operace, které s nimi má provádět. Význam výpočtů známe my a my také musíme znát jeho správný postup (v tomto případě vzorec na výpočet obsahu trojúhelníka pomocí délek základny a výšky).

Teď bychom chtěli, aby Scheme dělal co nejvíc práce za nás. Aby stačilo říct mu, ať vypočítá obsah trojúhelníka se zadanými délkami základny a výšky:

```
> (triangle-area 4 3)
6
```

Chceme vytvořit *abstrakci*.

Abstrakce

Jako činnost (proces)

Myšlenkový proces, kterým předmět zbavujeme zvláštností a odlišností a uvažujeme jen jeho obecné a (pro daný problém) podstatné vlastnosti.

(Ve hmotném světě žádný trojúhelník neexistuje. Existují jen (víceméně) trojúhelníkové předměty. Procesem abstrakce se zbavíme všech jejich vlastností, které s jejich „trojúhelníkovitostí“ nesouvisejí (materiál, hmotnost, barva, účel...). Necháme si jen ty pro onu „trojúhelníkovitost“ podstatné: vrcholy, strany, jejich délky apod.)

Jako pojem

Výsledek procesu abstrakce, tzv. *abstraktní pojem*. Je dán souhrnem vlastností nějaké množiny objektů a má *název*. (Každý pojem je více nebo méně abstraktní.)

(Například pojem *trojúhelník*, který zahrnuje vlastnosti trojúhelníkových předmětů. V programování pracujeme s abstraktními pojmy: procedura `triangle-area` by měla počítat obsah všech trojúhelníků bez ohledu na jejich další vlastnosti. Můžeme ji pak použít na všechny trojúhelníkové objekty. Například pomocí obsahu trojúhelníka můžeme zjistit hmotnost trojúhelníkové desky, spotřebu materiálu apod.; v počítačové grafice můžeme zjistit vlastnosti trojúhelníka bez ohledu na jeho barvu; koneckonců už to, že při zadání délek neuvádíme jednotky, je taky abstrakce.)

Programová abstrakce

Vytvoření nástroje (např. procedury), který je možné použít bez znalosti technických detailů jeho práce. (Víme *co* dělá, nemusíme vědět, *jak*.)

Výhody programové abstrakce

- v daný moment řešíme pouze omezené množství problémů

  (nemusíme řešit, *jak* se počítá obsah trojúhelníka, stačí použít proceduru `triangle-area`)

- neřešíme problémy, které zrovna nejsou podstatné

  (*jak* se počítá obsah trojúhelníka, nemusí být zrovna podstatné)

- zvyšuje se možnost znovupoužitelnosti programu

  (procedura na výpočet obsahu trojúhelníka bude častěji použitelná, než procedura na výpočet obsahu dřevěné trojúhelníkové desky)

- zvyšuje se čitelnost

  (`(triangle-area 5 6)` je čitelnější než `(* 1/2 5 6)`, snadněji pochopíme smysl)

- snadněji se dělají změny

  (pokud se rozhodneme počítat obsah trojúhelníka jiným způsobem, třeba takto: `(/ (* b h) #i2)`, uděláme změnu jen na jednom místě)

- snižuje se chybovost

  (když nemusíme myslet na mnoho věcí současně, děláme méně chyb)

Definice procedury

K definování procedury můžeme použít speciální operátor `define`:

```
(define (triangle-area b h)
  (* 1/2 b h))
```

Testy:

```
> (triangle-area 4 3)
6
> (triangle-area 12 1)
6
> triangle-area
#<procedure:triangle-area>
```

Důležité pojmy:

```
                (formální) parametry procedury
(define (triangle-area b h)
  (* 1/2 b h))
   tělo procedury
```

Parametry procedury jsou **symboly**. Je třeba správně pochopit, čím se liší od argumentů, na které se procedura aplikuje. Název *formální* parametry se používá v jiných programovacích jazycích, kde se argumentům říká *aktuální parametry*.

Každá procedura si pamatuje:

- své parametry,
- své tělo
- a ještě jednu informaci (zjistíme později).

Definice procedur píšeme v DrRacket obvykle do okna definic. Obsah tohoto okna lze ukládat do souboru, takže o něj nepřijdeme a příště ho můžeme zase použít. Okno interakcí se používá na experimentování a testování.

Procedury, které jsme sami definovali, se někdy nazývají *uživatelsky definované procedury*, na rozdíl od procedur *primitivních*, které už ve Scheme jsou (např. procedura `+`).

Aplikace uživatelsky definované procedury

V minulé přednášce jsme při popisu vyhodnocovacího procesu nerozebírali, co se děje při aplikaci procedury. Jen jsme řekli, že procedura provede nějaký výpočet a vrátí jeho výsledek. Tak to bude i nadále u procedur primitivních. U uživatelsky definovaných procedur je ale třeba přesně vědět, co se při jejich aplikaci stane.

Vyhodnocujeme například

```
> (triangle-area (+ 5 7) (- 4 3))
```

Předpokládejme, že vyhodnocovací proces už došel do momentu, kdy aplikuje proceduru `triangle-area` na čísla 12 a 1. Co se bude dít dál?

1. Zařídí se, aby symbol `b` měl hodnotu `12` a symbol `h` měl hodnotu `1`.
2. Vyhodnotí se tělo procedury.
3. Výsledkem aplikace bude výsledek tohoto vyhodnocení.

2 Vazby

Zajímavá otázka

Co bude výsledkem posledního vyhodnocení?

```
> (define r 0)
> (define (circle-area r)
    (* pi r r))
> (circle-area 10)
#i314.1592653589793
> r
???
```

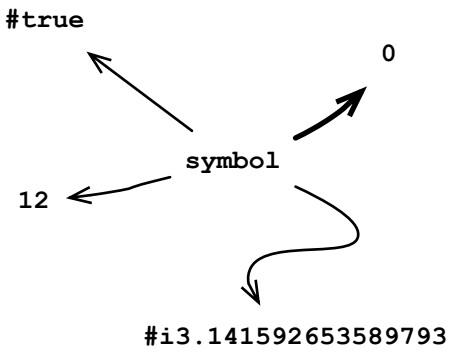
Samozřejmě by se nám líbilo, kdyby výsledkem bylo `0`. Uvědomte si ale, že dokud nevysvětlíme, jak přesně se při aplikaci procedury `circle-area` nastavuje symbol `r` na hodnotu argumentu, nemůžeme si být jisti, že to nebude `10`.

Naštěstí to tak opravdu bude a teď si řekneme, jak je to zařízeno.

Vazby

- Každý symbol může mít více *vazeb*.
- Jedna vazba může být *aktuální*. Ta *zastiňuje* ostatní (na obr. tučně).

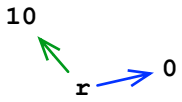
- Každá vazba má *hodnotu*.
- Hodnotou symbolu je vždy **hodnota jeho aktuální vazby**.



Zpět k zajímavé otázce:

```
> (define r 0)
> (define (circle-area r)
    (* pi r r))
> (circle-area 10)
#i314.1592653589793
> r
???
```

Symbol `r` má dvě vazby:



V jazyce Scheme je zařízeno, že v těle procedury `circle-area` je aktuální první vazba, v okně interakcí druhá. Proto bude poslední hodnota opravdu `0`.

Při aplikaci procedury `circle-area`:

1. se vytvoří nová vazba na symbol `r`,
2. učiní se aktuální (tím zastíní původní vazbu),
3. nastaví se jí hodnota `10`.
4. Vyhodnotí se tělo procedury.
5. Po vyhodnocení vazba přestává být aktuální.

Vazba na symbol `pi` se nevytváří a nenastavuje, aktuální zůstává původní vazba. Totéž platí ještě pro jiný symbol (který?).

3 Prostředí

Teď si povíme přesně, jak je dosaženo, že se vazby při aplikaci procedury chovají, jak bylo ukázáno na příkladech. K tomu budeme potřebovat nový pojem.

Vazby jsou organizovány v *prostředí*. To chápeme jako tabulku, ze které Scheme zjišťuje vazby symbolů a jejich hodnoty. Řádky v tabulce jsou vazby. Například takto si můžeme představit *globální* (*počáteční*) *prostředí*, které obsahuje vazby aktuální v okně interakcí:

Globální prostředí

symbol	hodnota
pi	#i3.141592653589793
+	#<procedure:+>
-	#<procedure:->
*	#<procedure:*>
/	#<procedure:/>
sqrt	#<procedure:sqrt>
⋮	⋮

Vyhodnocením

```
> (define r 0)
> (define (circle-area r)
    (* pi r r))
```

se do globálního (počátečního) prostředí přidají dvě nové vazby:

Globální prostředí

symbol	hodnota
circle-area	#<procedure:circle-area>
r	0
pi	#i3.141592653589793
+	#<procedure:+>
-	#<procedure:->
*	#<procedure:*>
/	#<procedure:/>
sqrt	#<procedure:sqrt>
⋮	⋮

Speciální operátor `define` (znovu)

Teď si můžeme přesně popsat, jak pracuje speciální operátor `define` při definici proměnné (nikoli procedury). Výraz se speciálním operátorem `define` musí v tom případě mít dva argumenty (při definici procedury jsou tři).

Vyhodnocení výrazu `(define` *a b*`)`

1. Vyhodnotí se b.
2. V globálním prostředí se vytvoří nová vazba na symbol a.
3. Získaná hodnota výrazu b se učiní hodnotou této vazby.

Vraťme se k prostředím. Při aplikaci procedury `circle-area` na hodnotu `10` se vytvoří nové prostředí s vazbou na symbol `r`:

Prostředí
procedury `circle-area`

symbol	hodnota
`r`	`10`

To ale nestačí, protože v těle procedury se potřebují i vazby z globálního prostředí (symboly `pi` a `*`). Proto se zavádí *předek prostředí*.

Globální prostředí

symbol	hodnota
`circle-area`	`#<procedure:circle-area>`
`pi`	`#i3.141592653589793`
`+`	`#<procedure:+>`
`-`	`#<procedure:->`
`*`	`#<procedure:*>`
⋮	⋮

Prostředí
procedury `circle-area`

symbol	hodnota
`r`	`10`

Globální prostředí je **předkem** prostředí procedury `circle-area`. Každé prostředí kromě globálního má předka.

Znovu o vyhodnocování

Je jasné, že výsledek vyhodnocení výrazu závisí na aktuálním prostředí. Proto musíme vylepšit pojetí vyhodnocovacího procesu, se kterým jsme se seznámili minule. Pokud se vyhodnocuje výraz, dělá se to vždy v nějakém prostředí. Správný pojem vyhodnocení je tedy **vyhodnocení výrazu v prostředí**.

Aktuální prostředí je prostředí, ve kterém je výraz vyhodnocován.

Vyhodnocování složeného výrazu v daném prostředí probíhá tak, jak jsme ho už popsali. Jediné upřesnění se týká vyhodnocování jeho podvýrazů: to probíhá vždy ve stejném prostředí, jako vyhodnocování původního složeného výrazu. **Pokud není uvedeno jinak**. (Poslední poznámka se týká např. speciálních operátorů `define` a `let`.)

Větší změna je u vyhodnocování symbolů:

Vyhodnocování symbolu

Při vyhodnocování symbolu se nejprve hledá jeho vazba v aktuálním prostředí. Pokud je nalezena, hodnotou symbolu je hodnota vazby. Pokud není nalezena, vazba se hledá v předkovi aktuálního prostředí. Tak se pokračuje tak dlouho, dokud se neskončí v globálním prostředí. Pokud ani tam není vazba nalezena, dojde k chybě.

Přesnější popis toho, co se děje při aplikaci uživatelské procedury:

Aplikace uživatelské procedury

Při aplikaci uživatelské procedury se vytvoří nové prostředí s vazbami parametrů na hodnoty argumentů. Předkem tohoto prostředí se stane globální prostředí (toto později zobecníme). V tomto prostředí se pak vyhodnotí tělo procedury. Nové prostředí se vytváří při každé aplikaci znovu.

4 Vytváření prostředí operátorem `let`

Pomocí speciálního operátoru `let` můžeme explicitně vytvářet nová prostředí. Jeho význam je intuitivně jasný z příkladu:

```
> (let ((a 2)
        (b (+ 1 2)))
    (* a b))
> 6
```

`let`: terminologie

```
          popis prostředí
(let ((a 2)  (b (+ 1 2)))
    popis vazby  popis vazby
  (* a b))
    tělo
```

Popis prostředí Seznam libovolné délky. Jeho prvky jsou *popisy vazeb*.

Popis vazby Dvouprvkový seznam. Na prvním místě má symbol, na druhém libovolný výraz.

Tělo Libovolný výraz.

`let`: vyhodnocení

1. V aktuálním prostředí se vyhodnotí všechny druhé položky popisů vazeb.
2. Vytvoří se nové prostředí a v něm vazby tak, že každá první položka popisu vazby (která musí být symbolem) se naváže na hodnotu druhé položky.
3. Předkem nového prostředí se učiní aktuální prostředí.
4. Tělo se vyhodnotí v tomto novém prostředí. Výsledek se vrátí jako hodnota celého výrazu.

5 Závěr

Pojmy k zapamatování

Primitivní a uživatelsky definovaná procedura, parametry a tělo procedury, vazba, aktuální vazba, zastínění vazby, hodnota vazby, prostředí, globální (počáteční) prostředí, aktuální prostředí, vyhodnocení výrazu v prostředí, předek prostředí; popis prostředí, popis vazby a tělo pro operátor `let`.

Kontrolní otázky

1. Kolik prvků může mít výraz se speciálním operátorem `define`?
2. Kolik prvků může mít výraz se speciálním operátorem `let`?

Otázky a úkoly na cvičení

Definice všech procedur pište do okna definic a ukládejte do souboru.

1. Uvažme proceduru `my-if` definovanou takto:

```
(define (my-if a b c)
  (if a b c))
```

   Je nějaký rozdíl mezi použitím této procedury a speciálního operátoru `if`?

2. Napište proceduru `power2` na výpočet druhé mocniny zadaného čísla:

```
> (power2 5)
25
> (power2 -6)
36
```

(Ve Scheme je na to funkce `sqr`.)

3. Napište procedury `power3`, `power4`, `power5` na další mocniny.

4. Napište proceduru `hypotenuse`, která z délek odvěsen pravoúhlého trojúhelníka vypočítá délku přepony:

```
> (hypotenuse 3 4)
5
```

5. Napište proceduru `absolute-value`, která vypočítá absolutní hodnotu zadaného čísla:

```
> (absolute-value 2)
2
> (absolute-value -3)
3
```

(Ve Scheme už taková procedura je a jmenuje se `abs`.)

6. Jaké výhody a nevýhody má následující řešení předchozího příkladu?

```
(define (absolute-value x)
  (sqrt (power2 x)))
```

7. Definujte proceduru `signum`, která vrátí `1`, když je zadané číslo kladné, `0`, když je nulové, a jinak vrátí `-1`. (Ve Scheme už je procedura `sgn`, která to dělá.)

8. Napište proceduru `minimum` (resp. `maximum`), které ze zadaných čísel vrátí to menší (resp. větší). (Ve Scheme už jsou procedury `min` a `max`.)

9. Napište proceduru `my-positive?`, která vrátí logickou hodnotu „zadané číslo je kladné“:

```
> (my-positive? 5)
#true
> (my-positive? 1.5)
#true
> (my-positive? (- pi))
#false
> (my-positive? 0)
#false
```

Napište analogickou proceduru `my-negative?`. (Ve Scheme už jsou procedury `positive?` a `negative?`.)

10. Bod v rovině můžeme zadat pomocí dvou kartézských souřadnic. V tomto a některých dalších příkladech budeme pro x-ovou a y-ovou souřadnici bodu používat symboly stejného názvu, jen odlišené koncovkami „`-x`“ a „`-y`“. Například symboly `A-x` a `A-y` by označovaly x-ovou a y-ovou souřadnici téhož bodu. (Scheme v našem nastavení nerozlišuje velká a malá písmena v symbolech, takže `a-x` a `a-y` označují totéž, ale méně srozumitelně, protože body je zvykem značit velkými písmeny.)

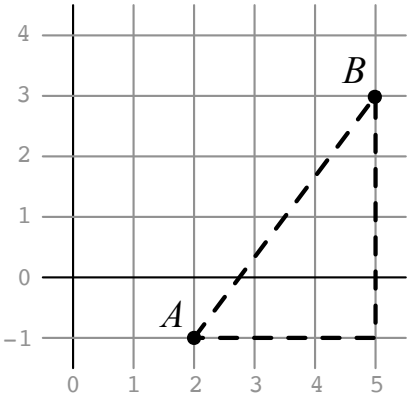
    Víme, že vzdálenost bodů v rovině můžeme vypočítat pomocí Pythagorovy věty. Následující procedura to dělá:

```
(define (point-distance A-x A-y B-x B-y)
  (let ((x-leg (- A-x B-x))
        (y-leg (- A-y B-y)))
    (sqrt (+ (sqr x-leg) (sqr y-leg)))))
```

    (Slovo *leg* se v angličtině používá pro odvěsnu.) Takto například vypočítáme vzdálenost bodu A o souřadnicích $[2, -1]$ a bodu B o souřadnicích $[5, 3]$:

```
> (point-distance 2 -1 5 3)
5
```

    Na obrázku:



    Vadou této procedury ovšem je, že nepoužívá už napsanou proceduru `hypotenuse`. Napravte to.

11. Pokud čísla a, b, c jsou délkami stran trojúhelníka, pak splňují *trojúhelníkové nerovnosti*

$$a + b > c,$$
$$b + c > a,$$
$$c + a > b.$$

    Naopak, pokud kladná čísla a, b, c splňují tyto nerovnosti, pak mohou být délkami stran trojúhelníka.

Napište proceduru `triangle?`, která vrátí logickou hodnotu „zadaná tři čísla mohou být délkami stran trojúhelníka“:

```
> (triangle? 1 1 1)
#true
> (triangle? 3 2 1)
#false
> (triangle? 2 3 4)
#true
```

Snažte se poceduru napsat co nejjednodušeji. Neváhejte použít operátor `let` nebo pomocné procedury, pokud se tím výsledek zjednoduší (například abyste se vyhnuli opakovanému psaní téhož). Také používejte procedury, které jste už dříve napsali. Tohle pravidlo platí obecně pro cokoliv, co budete kdy programovat. Určitě hned pro následující příklady:

12. Obsah trojúhelníka o stranách délek a, b, c lze vypočítat pomocí *Heronova vzorce*:

$$S = \sqrt{s(s-a)(s-b)(s-c)},$$

kde

$$s = \frac{a+b+c}{2}.$$

Napište proceduru `heron` se třemi parametry, která pomocí Heronova vzorce vypočítá obsah trojúhelníka zadaného délkami stran.

13. Napište proceduru `heron-cart`, která pomocí Heronova vzorce vypočítá obsah trojúhelníka zadaného body v kartézských souřadnicích. Například

```
> (heron-cart 2 -1 5 -1 5 3)
6
```

vypočítá obsah trojúhelníka z předchozího obrázku. (Parametry pojmenujte `A-x`, `A-y`, `B-x`, `B-y`, `C-x`, `C-y`.)