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. Funkce a prostředí

verze z 9. října 2019

1 Uživatelsky definované funkce

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

```
CL-USER 23 > (* 1/2 4 3)
6
```

Obsah trojúhelníka o základně 6 a výšce 7 vypočítáme takto:

```
CL-USER 24 > (* 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:

```
CL-USER 24 > (* 1/2 b h)
```

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

To je použití Listeneru jako kalkulačky: zadáváme mu pouze čísla a základní operace, které s nimi má provádět. Význam výpočtu 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 Lisp dělal co nejvíc práce za nás. Aby stačilo říct mu, ať vypočítá obsah trojúhelníka se zadanými délkami základny a výšky:

```
CL-USER 25 > (triangle-area 4 3)
6
```

Chceme definovat naši vlastní funkci.

Definice funkce

K definování funkce můžeme použít makro `defun`:

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

Testy:

```
CL-USER 26 > (triangle-area 4 3)
6

CL-USER 27 > (triangle-area 12 1)
6
```

Důležité pojmy:

```
        název funkce       parametry funkce
(defun triangle-area (b h)
  (* 1/2 b h))
   tělo funkce
```

V makru `defun` tedy stanovujeme název, parametry a tělo nové funkce.

Název funkce je libovolný symbol. **Parametry funkce** jsou symboly, **tělo funkce** je libovolný výraz.

Je třeba správně pochopit, čím se parametry funkce liší od hodnot, na které se funkce aplikuje. V jiných programovacích jazycích se parametry nazývají *formální* parametry a pro hodnoty, na které je funkce aplikována se používá název *aktuální parametry*.

Ke každé funkci musí být známy následující informace:

- parametry,
- tělo
- a ještě něco (co to je, zjistíme později).

Dále je třeba, aby k danému symbolu byla známa funkce, jíž je symbol jménem (naopak, tj. k funkci daných parametrů a těla jméno znát nepotřebujeme).

Definice funkcí píšeme do souborů, které můžeme ukládat a příště zase otevírat. Listener se používá obvykle jen na experimentování a testování. Jak se v LispWorks se soubory pracuje, se dozvíte na cvičení.

Funkce, které jsme sami definovali, se někdy nazývají *uživatelsky definované funkce*, na rozdíl od funkcí *primitivních*, které už v Lispu jsou (např. funkce `sqrt`).

Aplikace uživatelsky definované funkce

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

Vyhodnocujeme například

```
CL-USER 28 > (triangle-area (+ 5 7) (- 4 3))
```

Předpokládejme, že vyhodnocovací proces už došel do momentu, kdy aplikuje funkci `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 funkce.
3. Výsledkem aplikace bude výsledek tohoto vyhodnocení.

2 Funkce jako abstrakce

Programová abstrakce

Nástroj (např. funkce), 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 funkci `triangle-area`)

- neřešíme problémy, které zrovna nejsou podstatné

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

- zvyšuje se možnost znovupoužitelnosti programu

  (funkce na výpočet obsahu trojúhelníka bude častěji použitelná, než funkce 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) 2.0)`, uděláme změnu jen na jednom místě)

- snižuje se chybovost

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

3 Vazby

Zajímavá otázka

Definujeme funkci `circle-area`:

```
(defun circle-area (r)
  (* pi r r))
```

Co bude nyní výsledkem posledního vyhodnocení?

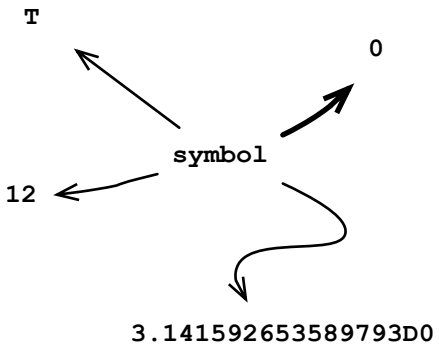
```
CL-USER 4 > (setf r 0)
0

CL-USER 5 > (circle-area 10)
314.1592653589793D0

CL-USER 6 > r
???
```

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

Naštěstí hodnota opravdu bude `0` a teď si řekneme, jak je to zařízeno.

Vazby

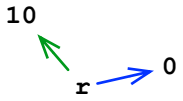
- Každý symbol může mít více *vazeb*.
- Jedna vazba může být *aktuální*. Ta *zastiňuje* ostatní (na obr. tučně).
- Každá vazba má *hodnotu*.
- Hodnotou symbolu je vždy **hodnota jeho aktuální vazby**.



Zpět k zajímavé otázce:

```
CL-USER 4 > (setf r 0)
0

CL-USER 5 > (circle-area 10)
314.1592653589793D0

CL-USER 6 > r
???
```

Symbol `r` má dvě vazby:



V jazyce Lisp je zařízeno, že v těle funkce `circle-area` je aktuální první vazba, v Listeneru druhá. Proto bude poslední hodnota opravdu `0`.

Při aplikaci funkce `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 funkce.
5. Po vyhodnocení vazba přestává být aktuální.

Vazba na symbol `pi` se nevytváří a nenastavuje, aktuální zůstává původní vazba.

4 Prostředí

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

Vazby jsou organizovány v *prostředí*. To chápeme jako tabulku, ze které Lisp zjišťuje vazby symbolů a jejich hodnoty. Řádky v tabulce jsou vazby. Například takto si můžeme představit *globální prostředí*, které obsahuje vazby aktuální v Listeneru:

Globální prostředí

symbol	**hodnota**
`pi`	`3.141592653589793D0`
⋮	⋮

Vyhodnocením

```
CL-USER 4 > (setf r 0)
0
```

se do globálního prostředí přidá nová vazba:

Globální prostředí

symbol	**hodnota**
`r`	`0`
`pi`	`3.141592653589793D0`
⋮	⋮

Makro `setf` (znovu)

Teď si můžeme přesněji popsat, jak pracuje makro `setf`. Výraz s makrem `setf` musí mít dva argumenty.

Vyhodnocení výrazu `(setf` *a b*`)`

1. Vyhodnotí se *b*.
2. Pokud v globálním prostředí neexistuje vazba na symbol *a*, vytvoří se.
3. Získaná hodnota výrazu *b* se učiní hodnotou této vazby.

Ačkoli to Lisp umožňuje, **budeme makro `setf` používat pouze v Listeneru** (a nikdy v těle funkcí).

Vraťme se k prostředím a ukažme si, co se s nimi děje při aplikaci funkce. Při aplikaci funkce `circle-area` na hodnotu `10` se vytvoří nové prostředí s vazbou na symbol `r`:

Prostředí
funkce `circle-area`

symbol	hodnota
`r`	`10`

To ale nestačí, protože v těle funkce se potřebuje i vazba z globálního prostředí (symbol `pi`). Proto se zavádí *předek prostředí*.

Globální prostředí

symbol	hodnota
`r`	`0`
`pi`	`3.141592653589793D0`
⋮	⋮

Prostředí
funkce `circle-area`

symbol	hodnota
`r`	`10`

Globální prostředí je **předkem** prostředí funkce `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. speciálního operátoru `let`, který si ukážeme za chvíli.)

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 funkce:

Aplikace uživatelské funkce

Při aplikaci uživatelské funkce 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 funkce. Nové prostředí se vytváří při každé aplikaci znovu.

5 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.

6 Závěr

Probrané pojmy

Primitivní a uživatelsky definovaná funkce, parametry a tělo funkce, 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`.

Otázky a úkoly na cvičení

Definice všech funkcí ukládejte do souboru.

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

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

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

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

```
CL-USER 9 > (power2 5)
25

CL-USER 10 > (power2 -6)
36
```

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

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

```
CL-USER 11 > (hypotenuse 3 4)
5
```

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

```
CL-USER 12 > (absolute-value 2)
2

CL-USER 13 > (absolute-value -3)
3
```

(V Lispu už taková funkce je a jmenuje se `abs`.)

6. Definujte funkci `signum`, která vrátí `1`, když je zadané číslo kladné, `0`, když je nulové, a jinak vrátí `-1`. (V Lispu už je funkce `sgn`, která to dělá.)

7. Napište funkci `minimum` (resp. `maximum`), které ze zadaných čísel vrátí to menší (resp. větší). (V Lispu už jsou funkce `min` a `max`.)

8. Napište funkci `my-positive-p`, která vrátí logickou hodnotu „zadané číslo je kladné“:

```
CL-USER 14 > (my-positive-p 5)
T

CL-USER 15 > (my-positive-p 1.5)
T

CL-USER 16 > (my-positive-p (- pi))
NIL

CL-USER 17 > (my-positive-p 0)
NIL
```

   Napište analogickou funkci `my-negative-p`. (V Lispu už jsou funkce `plusp` a `minusp`.)

9. 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. (Lisp 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.)

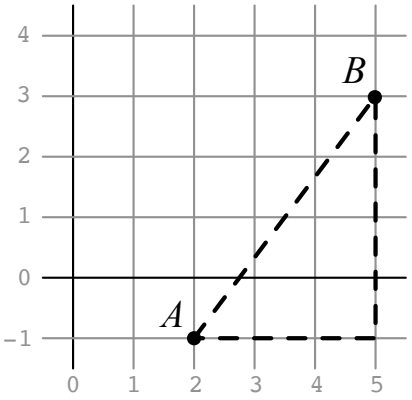
   Víme, že vzdálenost bodů v rovině můžeme vypočítat pomocí Pythagorovy věty. Následující funkce to dělá:

```
(defun point-distance (A-x A-y B-x B-y)
  (let ((x-leg (- A-x B-x))
        (y-leg (- A-y B-y)))
    (sqrt (+ (* x-leg x-leg) (* y-leg 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 $[5, 3]$:

```
CL-USER 18 > (point-distance 2 -1 5 3)
5.0
```

Na obrázku:



Vadou této funkce ovšem je, že nepoužívá už napsanou funkci `hypotenuse`. Napravte to.

10. 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 funkci `trianglep`, která vrátí logickou hodnotu „zadaná tři čísla mohou být délkami stran trojúhelníka“:

```
CL-USER 19 > (trianglep 1 1 1)
T

CL-USER 20 > (trianglep 3 2 1)
NIL

CL-USER 21 > (trianglep 2 3 4)
T
```

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

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

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

```
CL-USER 22 > (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`.)