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 2 ◇ poznámky k přednášce

1. Opakování Common Lispu (a něco navíc)

verze z 19. května 2020

1 Úvod

V minulém semestru jsme se seznámili s některými základními principy programování (zejména funkcionálního). Neuškodí je tady zopakovat: vyhodnocování výrazů, prostředí, vazby a proměnné, funkce pojmenované a anonymní, lexikální uzávěry, rekurzivní funkce a rekurzivní výpočetní proces, abstraktní datové struktury, páry a struktury z nich vytvořené, jako jsou seznamy a stromy, symbolická data. Základy jsme pak implementovali v interpretu jednoduchého programovacího jazyka.

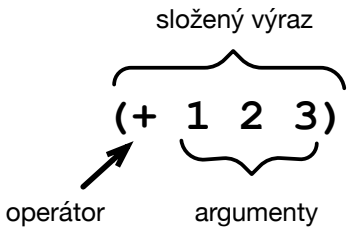
Průběžně jsme se také seznamovali se základy jazyka Common Lisp, ve kterém jsme principy demonstrovali a ve kterém jsme programovali příklady. Účelem této přednášky je zopakovat a shrnout je na jednom místě. Také přidáme něco nového, co se nám v budoucnu bude hodit.

2 Vyhodnocování výrazů

Program v Lispu zapisujeme formou *výrazů*. Výrazy mohou být

- symboly
- čísla, textové řetězce a další
- složené výrazy

Prvním dvěma typům výrazů říkáme též *atomy* nebo *jednoduché výrazy*. Složeným výrazům říkáme i *seznamy*. (I když je to trochu nepřesné, protože prázdný seznam není složený výraz, ale symbol.)



Slovo „argument“ se používá také v jiném významu, např. ve větě „Parametry funkce se navážou na argumenty“.

Běh programu v Lispu lze chápat jako postupné **vyhodnocování výrazů** v rámci **vyhodnocovacího procesu**. Výrazy se vždy vyhodnocují v nějakém prostředí (o prostředích podrobněji za chvíli) podle následujících pravidel:

Vyhodnocení výrazu E **v prostředí** *env*

Je-li E **symbol,** výsledkem je hodnota symbolu E v prostředí *env*.

Je-li E **jiný atom než symbol,** výsledkem je E.

Je-li E **seznam s operátorem** o **a argumenty** $a_1 \dots a_n$**,** pak

> **Jestliže** o **je speciální operátor nebo makro,** seznam se vyhodnotí podle pravidel tohoto operátoru.
>
> **Jinak** operátor o musí být symbol. Pak
>
> 1. se zjistí hodnota f funkční vazby symbolu o v prostředí *env* (je to funkce),
> 2. zjistí se hodnoty $v_1 \dots v_n$ argumentů $a_1 \dots a_n$ v prostředí *env* (opět vyhodnocovacím procesem).
> 3. Výsledkem je výsledek aplikace funkce f na hodnoty $v_1 \dots v_n$. Aplikace funkce může mít též vedlejší efekt. Funkce mohou vracet více hodnot.

V rámci vyhodnocování výrazu E se prostředí *env* říká také **aktuální prostředí**. Proto by např. bod 2. mohl znít takto: „zjistí se hodnoty $v_1 \dots v_n$ argumentů $a_1 \dots a_n$ v aktuálním prostředí.“

Pojem hodnota, použitý před chvílí, obecně označuje cokoli, co může být výsledkem aplikace funkce. **Hodnotu** (též **prvek jazyka**; v Lispu se také říká **objekt**) lze použít jako:

- hodnotu vazby symbolu (tj. uložit do proměnné),
- hodnotu v datové struktuře,
- argument funkce,
- návratovou hodnotu funkce.

Čísla, symboly, páry, funkce jsou hodnoty. Prostředí a vazby nejsou hodnoty.

O **výrazech** hovoříme v souvislosti se zdrojovým kódem programu. O **hodnotách** mluvíme při popisování, co se děje za běhu programu.

Další pojem použitý při popisu vyhodnocovacího procesu je pojem prostředí. Zopakujeme, co to je (a ještě dnes rozšíříme).

Prostředí	se skládá ze **seznamu vazeb** symbolů a **předka**.
Vazba symbolu	je spojení symbolu a hodnoty. Vazby mohou být **hodnotové** a **funkční**.
Předek prostředí	je jiné prostředí. Prostředí předka mít nemusí.

Během vyhodnocování výrazů se také zjišťují hodnoty symbolů v daném prostředí. To se dělá ve dvou krocích: nejprve se zjistí vazba symbolu v prostředí a pak hodnota symbolu v této vazbě.

Hledání vazby symbolu v daném prostředí

(platí pro hodnotovou i funkční vazbu)

1. vazba se hledá v prostředí
2. pokud není nalezena, hledá se rekurzivně v předkovi prostředí
3. pokud předek neexistuje, je to chyba

Hodnota symbolu v prostředí

- je to hodnota nalezené vazby symbolu

Zvláštním typem symbolu jsou tzv. **klíče**. Jsou to symboly, jejichž zápis začíná dvojtečkou. Hodnotou klíče je vždy tentýž klíč:

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

CL-USER 2 > :a-key-with-a-long-name
:A-KEY-WITH-A-LONG-NAME
```

3 Funkce

Známe dva typy funkcí:

- vestavěné (primitiva)
- uživatelské (uživatelsky definované)

Uživatelské funkce obsahují

- seznam parametrů neboli λ-*seznam*

- tělo
- prostředí (tzv. **prostředí vzniku**)
- nepovinně dokumentaci

λ-seznam: seznam symbolů zvaných **parametry funkce**, před posledním může být klíčové slovo `&rest`. (Za chvíli rozšíříme o další možnosti.)

Tělo: seznam výrazů.

Prostředí: prostředí. Proto jsou funkce v Lispu **lexikální uzávěry**.

Dokumentace: textový řetězec.

Při aplikaci uživatelské funkce se vykonají následující kroky:

1. Vytvoří se prostředí, ve kterém se argumenty aplikace navážou na parametry funkce (podrobnosti dále).
2. Předkem prostředí se učiní prostředí vzniku funkce.
3. V novém prostředí se postupně (od prvního po poslední) vyhodnotí výrazy těla, hodnota posledního je návratovou hodnotou funkce.

Nové funkce lze vytvořit pomocí maker `lambda`, `defun`, `labels`:

`lambda` novou funkci vrací jako výsledek. Prostředí vzniku je aktuální prostředí.

`defun` funkci vytvoří a zadanému symbolu nastaví globální funkční vazbu na ni. Prostředí vzniku je aktuální prostředí (obvykle globální).

`labels` nová funkce je tzv. **lokální**. Prostředí vzniku i prostředí, ve kterém je vytvořena nová funkční vazba, je nové prostředí, jehož předkem je aktuální prostředí.

Zajímavost: více hodnot lze z funkce vrátit funkcí `values`. Navázat více hodnot vrácených vyhodnocením výrazu lze pomocí speciálního operátoru `multiple-value-bind`.

Zajímavost: dokumentace funkce

V definici funkce ji uvádíme jako řetězec za λ-seznam:

```
(defun add-zero (x)
  "Přičítá nulu k argumentu."
  (+ x 0))
```

Test:

```
CL-USER 1 > (documentation 'add-zero 'function)
"Přičítá nulu k argumentu."
```

(Ve vývojovém prostředí LispWorks také myší nebo `Meta+Ctrl+Shift+A`)

λ-seznamy funkcí

Základní klíčová slova uvozující parametry daného typu:

`&optional` nepovinné parametry dané pozičně

`&rest` parametr pro seznam zbylých argumentů

`&key` nepovinné parametry dané názvem

Povinné pořadí při použití více klíčových slov: `&optional`, `&rest`, `&key`. Více klíčových slov v λ-seznamu současně ale obvykle nebudeme používat.

Parametry před klíčovými slovy jsou **povinné**. Nejjednodušší λ-seznam je seznam povinných parametrů. S tím jsme se setkávali většinu minulého semestru.

Klíčové slovo `&optional`

uvádí nepovinné parametry dané pozičně. Defaultní hodnota parametrů je `nil`. To lze změnit, pokud místo parametru uvedeme dvouprvkový seznam, jehož prvním prvkem je parametr a druhým výraz, který se v momentě aplikace funkce vyhodnotí na defaultní hodnotu.

`&optional`-parametry jsou navazovány na argumenty, dokud jsou argumenty k dispozici. Pak se použije defaultní hodnota (defaultní defaultní hodnota je `nil`).

λ-seznam	funkce aplikovatelná na
`(a b)`	dva argumenty
`(a b &optional c)`	dva nebo tři argumenty default třetího je `nil`
`(a b &optional c d)`	dva až čtyři argumenty default třetího a čtvrtého je `nil`
`(a b &optional (c 0) (d c))`	dva až čtyři argumenty default třetího je `0`, čtvrtého hodnota parametru `c`

Možná podoba λ-seznamu funkce `log` je tedy tato:

```
(number &optional (base (exp 1)))
```

(funkce defaultně počítá logaritmus o základu e, lze ovšem nepovinně zadat jiný základ). Volání této funkce s méně než jedním a více než dvěma argumenty vede k chybě.

Ukázka. Definujme funkci g takto:

```
(defun g (a &optional b (c 0) (d (list a b c)))
  (print a)
  (print b)
  (print c)
  (print d)
  nil)
```

Test:

```
CL-USER 7 > (g 1)

1
NIL
0
(1 NIL 0)
NIL

CL-USER 8 > (g 1 2)

1
2
0
(1 2 0)
NIL

CL-USER 9 > (g 1 2 3)

1
2
3
(1 2 3)
NIL

CL-USER 10 > (g 1 2 3 4)

1
2
3
4
NIL
```

Klíčové slovo `&rest`
uvádí parametr, který bude při aplikaci funkce obsahovat seznam všech zbylých argumentů. Umožňuje tedy definovat funkce akceptující neomezený (teoreticky) počet argumentů. Naváže se na seznam všech argumentů nepoužitých předchozími parametry.

Funkce `+` a `*` akceptují libovolný (i nulový) počet argumentů, mohly by tedy mít tento λ-seznam:

```
(&rest numbers)
```

Funkce `-` a `/` mají jeden parametr povinný, jejich λ-seznam by mohl být

```
(number &rest numbers)
```

Ukázka. Definujme funkci `f` takto:

```
(defun f (a b &rest rest)
  (print a)
  (print b)
  (print rest)
  nil)
```

Test:

```
CL-USER 1 > (f 1 2)

1
2
NIL
NIL

CL-USER 2 > (f 1 2 3)

1
2
(3)
NIL

CL-USER 3 > (f 1 2 3 4)

1
2
(3 4)
NIL
```

Klíčové slovo `&key`
uvádí nepovinné parametry dané jménem. Jako jméno slouží symboly začínající dvojtečkou. //Ty se vyhodnocují samy na sebe.

Jsou navazovány na argumenty podle **jména**. To se zadává při aplikaci pomocí **klíčů**. Ve funkci se definují a používají stejně jako `&optional`-parametry.

λ-seznam	seznam argumentů	a	x	y	z
(a &key x y z)	(1 :x 2)	1	2	nil	nil
(a &key x y z)	(1 :y 3)	1	nil	3	nil
(a &key x y z)	(1 :x 2 :z 4)	1	2	nil	4
(a &key x (y 0) (z 0))	(1 :x 2 :z 4)	1	2	0	4

Na přednášce jsme si ukazovali, jak s `&key`-parametry pracuje funkce `find`.

Ukázka. Definujme funkci `h` takto:

```
(defun h (a &key b (c 0) (d (list a b c)))
  (print a)
  (print b)
  (print c)
  (print d)
  nil)
```

Test:

```
CL-USER 12 > (h 1)

1
NIL
0
(1 NIL 0)
NIL

CL-USER 13 > (h 1 :c 3)

1
NIL
3
(1 NIL 3)
NIL

CL-USER 14 > (h 1 :c 3 :d 4)

1
NIL
```

```
3
4
NIL
```

Zjištění použití nepovinných parametrů

Někdy se u nepovinných parametrů hodí možnost poznat, zda uživatel příslušný argument vůbec použil, nebo ne. Pokud je například λ-seznam funkce `fun (&optional x)`, pak nepoznáme, jestli uživatel funkci volal takto: `(f)`, nebo takto: `(f nil)`. K rozlišení těchto možností můžeme použít tříprvkový seznam, jehož první a druhý prvek jsou jako dříve parametr a defaultní hodnota a třetí prvek je další parametr, do nějž se při volání funkce uloží informace, zda byl nepovinný parametr použit, nebo ne.

Ukázka:

```
(defun fun (&optional (x nil usedp))
  (print (if usedp "Parametr použit" "Parametr nepoužit"))
  x)
```

Test:

```
CL-USER 9 > (fun 1)

"Parametr použit"
1

CL-USER 10 > (fun nil)

"Parametr použit"
NIL

CL-USER 11 > (fun)

"Parametr nepoužit"
NIL
```

Tuto metodu lze použít i u `&key`-parametrů.

4 REPL

Co se děje v Listeneru při vyhodnocování výrazů:

1. Zadaný text se **přečte**,

2. přečtený výraz se **vyhodnotí**,
3. výsledek se **vytiskne**.
4. Postup se **opakuje** od bodu 1.

(V Listeneru se dějí i další méně podstatné věci: uchovává se historie, aktualizují proměnné `*`, `**`, `***` ...)

Čtení: `Read`, vyhodnocení: `Eval`, tisk: `Print`, opakování: `Loop`.

Reader

Čtení: převod textu na hodnotu.

Reader: sada funkcí starajících se o čtení.

Základní funkce readeru:

- `read`
- `read-from-string`
- `read-line`, `read-char`

Evaluátor

Funkce `eval`. Vyhodnocuje zadané výrazy v globálním prostředí.

Printer

Tisk: převod hodnoty na text.

Printer: sada funkcí starajících se o tisk.

Základní funkce printeru:

- `print`,
- `prin1`, `princ`.
- `format`

Loop

Makro `loop` v jednoduché verzi spustí nekonečný cyklus. Další makra na cyklus:

- `dotimes`
- `dolist`

Jednoduchý REPL spustíme vyhodnocením výrazu `(loop (print (eval (read))))`

5 Viditelnost a životnost, speciální proměnné

Proměnné a vazby, které v Lispu používáme, mají tu vlastnost, že jejich přístupnost je určena nějakou oblastí ve zdrojovém kódu. Říká se také, že je určena *lexikálně*.

Například vazba symbolu vytvořená operátorem `let` je platná v jeho těle, vazba parametru funkce na argument je platná v těle funkce. Jen výrazy uvedené v těle (operátoru `let` nebo funkce) mohou vazbu použít.

U proměnných v Lispu platí, že kromě omezení daného lexikálně, o kterém jsme mluvili teď, už nijak jinak omezeny nejsou. Konkrétněji, nejsou omezeny *časově*. Jednou vytvořená vazba proměnné na hodnotu z pohledu programátora nikdy nezaniká (proto je v Lispu možné používat lexikální uzávěry).

Přístupnost vazeb symbolů je tedy dána dvěma podmínkami: za jakých okolností jsou vidět a po jaký čas od svého vzniku existují.

Těmto dvěma rysům vazeb symbolů (a dalších věcí) říkáme **viditelnost** (*scope*) a **životnost** (*extent*). Ty tedy

- Používají se k popisu vazeb symbolů (a dalších věcí).
- Určují, za jakých okolností (např. z jakých částí programu) jsou vidět a po jaký čas existují.

Základní typy viditelnosti jsou dva:

Lexikální viditelnost: vazba symbolu je vidět pouze v určité oblasti zdrojového kódu. V případě parametrů funkcí je to všude v těle funkce, v případě vazeb vytvořených operátorem `let` všude v jeho těle. V obou případech pokud není vazba překryta novou vazbou téhož symbolu.

Neomezenou viditelnost mají vazby, které jsou použitelné kdekoli (na libovolném místě) zdrojového kódu. V Lispu lze definovat proměnné s neomezenou viditelností, jak uvidíme později.

V jiných jazycích se můžeme setkat ještě s dalšími typy viditelnosti vazeb, například viditelnost v rámci jednoho zdrojového souboru (modulu).

Základní typy životnosti jsou také dva.

Omezenou životnost má vazba, jestliže existuje moment v čase, kdy vazba zanikne.

Neomezená životnost vazby způsobuje, že vazba po svém vzniku (z pohledu programátora) nikdy nezanikne.

Typy proměnných a vazeb jsou dány tím, jakou mají viditelnost a životnost.

Lexikální vazby. Mají lexikální viditelnost a neomezenou životnost. Jsou to vazby, které dosud známe z Lispu.

Dynamické vazby. Mají neomezenou viditelnost a omezenou životnost. Je to starší typ, ukážeme je za chvíli.

Dále například **lokální proměnné v C** mají lexikální viditelnost a omezenou životnost. **Globální proměnné** ve většině programovacích jazyků mají neomezenou viditelnost a neomezenou životnost.

V Lispu mohou mít symboly jak lexikální, tak dynamické vazby. Za normálních okolností jsou všechny vazby lexikální. Výjimky jsou tyto:

1. Symbol lze na globální úrovni definovat jako **speciální** (**dynamickou**) proměnnou, pak jsou všechny jeho nové vazby dynamické.
2. Novou vazbu symbolu můžeme učinit dynamickou pomocí vhodné **deklarace**.

Vysvětlíme obě tyto možnosti.

Speciální proměnná (nepřesně též **dynamická proměnná**) je proměnná, jejíž vazby jsou vždy a za každých okolností dynamické. Takovou je například proměnná `*print-length*`, která určuje délku tištěného seznamu:

```
CL-USER 1 > *print-length*
NIL

CL-USER 2 > (setf list (list 1 2 3 4 5 6 7 8 9 10))
(1 2 3 4 5 6 7 8 9 10)

CL-USER 3 > (setf *print-length* 8)
8

CL-USER 4 > list
(1 2 3 4 5 6 7 8 ...)

CL-USER 5 > (let ((*print-length* 5))
              (print list))

(1 2 3 4 5 ...)
(1 2 3 4 5 6 7 8 ...)
```

Poslední vyhodnocení používá techniku, která by nešla použít, kdyby proměnná `*print-length*` byla lexikální. Je to z toho důvodu, že její vazba na hodnotu `5` by pak byla vidět jen v těle `let`-výrazu, nikoliv v těle funkce `print` (které pochopitelně někde existuje, i když nevíme, jak vypadá).

Jen připomeňme, že druhý vypsaný seznam je vždy `(1 2 3 4 5 6 7 8 ...)`, protože jde o vytisknutou návratovou hodnotu funkci `print`. Ta se tiskla v době, kdy vnitřní vazba symbolu `*print-length*` už neexistovala.

V Lispu jsou i další speciální proměnné, které ovlivňují tisk. Jednou z mnoha je proměnná `*print-base*`, která určuje soustavu, ve které se tisknou čísla:

```
CL-USER 10 > (setf *print-base* 8)
10

CL-USER 11 > (let ((*print-base* 16))
               (print 140))

8C
214
```

Všimněte si, jak se v Listeneru jednotlivé hodnoty vytiskly. Na prvním řádku se vytisklo `10`, protože je to číslo 8 v osmičkové soustavě a v momentě tisku je už proměnná `*print-base*` nastavena na `8`. Na druhém řádku se vytisklo `8C` a `214`. První text vytiskla funkce `print` v našem kódu. V momentě, kdy tiskla, byla proměnná `*print-base*` nastavena na `16` (je to dáno **viditelností** vazby). `8C` je skutečně zápis čísla `140` v šestnáctkové soustavě. Funkce `print` pak vrátila hodnotu `140`. Ta se vytiskla v osmičkové soustavě, protože v ten moment už vazba proměnné `*print-base*` na číslo `16` neexistovala (má omezenou **životnost**).

V Lispu můžeme definovat vlastní speciální proměnné na globální úrovni makrem `defvar`:

```
(defvar var)

var:  symbol
```

Speciální proměnné je třeba vždy jasně odlišit, aby je nebylo možné zaměnit za proměnné lexikální. V Lispu je zavedena konvence, že **všechny proměnné definované makrem `defvar` mají název ohraničený hvězdičkami**.

Makro `defvar` má další dva nepovinné parametry: počáteční hodnotu proměnné a dokumentaci proměnné (kterou lze pak zjistit výrazem `(documentation` *`var`* `'variable)`. Počáteční hodnota se ovšem nastavuje jen při **prvním** vyhodnocení `defvar`-výrazu.

Pomocí tzv. *deklarace* lze stanovit pro jednotlivou vazbu symbolu, že má jít o dynamickou vazbu. Deklarace má tvar

```
(declare (special sym1 ... symn))
```

kde *sym1 ... symn* jsou názvy proměnných, jejichž vazby mají být dynamické, a uvádí se na začátek těla funkce, `let`-výrazu nebo dalších výrazů, které umožňují deklarace (např. `let*`).

Kdybychom například chtěli vyzkoušet funkci `adder` s dynamickou vazbou parametru `x`, udělali bychom to takto:

```
(defun adder (x)
  (declare (special x))
  (lambda (y)
    (+ y x)))
```

Test:

```
CL-USER 11 > (funcall (adder 5) 2)

Error: The variable X is unbound.

CL-USER 13 > (setf x 98)
98

CL-USER 14 > (funcall (adder 5) 2)
100
```

V tomto příkladu (na řádku 13) jsme nastavili globální hodnotu symbolu `x` na `98`. Jelikož jsme nepoužili makro `defvar`, nevytvořili jsme dynamickou proměnnou. Další vazby tohoto symbolu mohou i nadále být lexikální.

Speciální proměnné a dynamické vazby jsou potenciálně nebezpečné. Proto je dobré je příliš nepoužívat a pokud musíme, tak vždy s dodržením uvedené dohody pro jejich názvy.

Na závěr jen stručně poznamenejme, že **funkční vazby symbolů**, tedy vazby symbolů na funkce vytvořené například pomocí makra `defun`, **jsou vždy lexikální**. Lexikální viditelnost a neomezenou životnost mají tedy všechny názvy funkcí (i lokálních).

Otázky a úlohy na cvičení

1. Co bude hodnotou výrazu

```
(let ((x 'y))
  (funcall (let ((x 'z))
             (lambda () x))))
```

v případě, že vzniklé vazby na symbol `x` budou lexikální, a co v případě, kdy budou dynamické?

2. Máme tyto dvě funkce:

```
(defun my-mapcar (a b)
  (if (null b)
      '()
    (cons (funcall a (car b)) (my-mapcar a (cdr b)))))

(defun test (a b)
  (my-mapcar (lambda (c) (+ c b)) a))
```

Co bude hodnotou výrazu `(test '(1 2 3) 1)` v Common Lispu a co v Lispu, ve kterém by všechny symboly měly dynamické vazby?