
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. Od Scheme k Lispu

verze z 17. března 2019

V předmětu Paradigmata programování 2 přejdeme od Scheme k jazyku Common Lisp a vývojovému nástroji LispWorks. Na přednášce jsme probrali základní rozdíly mezi Schemem a Common Lispem. Ve zvláštním dokumentu máte k dispozici slovníček s překladem operátorů Scheme do Lispu.

Kromě zvládnutí základů jazyka je třeba, abyste se dobře obeznámili s vývojovým prostředím LispWorks, kterému se tady nevěnuji. Uživatelské příručky lze nalézt buď na webu LispWorks, nebo přímo v aplikaci.

1 Logické hodnoty

V jazyce Scheme slouží k reprezentaci logických hodnot hodnoty `#t` (*pravda*) a `#f` (*nepravda*). V Common Lispu tuto úlohu hrají symboly `t` a `nil`. Na rozdíl od Scheme se tedy nejedná o hodnoty speciálního typu, ale o symboly. Jejich zvláštností je, že se vyhodnocují samy na sebe, podobně jako například čísla:

```
CL-USER 7 > nil
NIL

CL-USER 8 > t
T
```

Zařídit, aby hodnotou symbolu byl tento symbol samotný je ovšem snadné; ve Scheme by se dalo napsat například `(define a 'a)`, v Common Lispu například `(defvar a 'a)`.

V Common Lispu se (stejně jako v některých variantách Scheme) používají tzv. *zobecněné logické hodnoty*: hodnotu *pravda* může reprezentovat libovolná hodnota kromě symbolu `nil`, hodnotu *nepravda* pak symbol `nil`. V literatuře o Common Lispu i v tomto textu je zvykem psát slovo *pravda* místo konkrétnější hodnoty v případě, že je tato hodnota podstatná pouze jako logická hodnota. V tomto kontextu se také používá slovo *nepravda* jako synonymum pro symbol `nil`.

Jako argumenty logických operací je tedy možno používat jakékoliv hodnoty a samotné logické operace mohou jako hodnotu *pravda* místo symbolu `t` vracet jiný, v dané situaci užitečnější objekt.

Příklad

Výraz `(and a b c)` vrací *pravdu*, pokud hodnoty `a`, `b` i `c` jsou *pravda*. Díky použití zobecněných logických hodnot je možno, aby proměnné `a`, `b`, `c` nabývaly i jiných hodnot než jenom `t` nebo `nil`, a hodnotou celého výrazu může být i něco jiného, než jen symboly `t` a `nil`. Makro `and` je například definováno tak, že pokud jsou všechny jeho argumenty *pravda* (rozumějme: různé od `nil`), vrací hodnotu posledního z nich.

Výraz, který vrátí číslo `3`, pokud je `x` rovno `1` a `y` rovno `2`, a jinak vrátí `nil`, se dá napsat takto:

```
(if (and (= x 1) (= x 2)) 3 nil)
```

nebo takto:

```
(when (and (= x 1) (= x 2)) 3)
```

(s makrem `when` se ještě setkáme), nebo jednoduše takto:

```
(and (= x 1) (= y 2) 3)
```

Pokud bychom navíc třeba chtěli, aby v případě negativního výsledku výraz nevracel `nil`, ale nulu, můžeme použít makro `or`, které také pracuje se zobecněnými logickými hodnotami, a napsat jej takto:

```
(or (and (= x 1) (= y 2) 3) 0)
```

2 Prázdný seznam

V Common Lispu není žádná zvláštní hodnota reprezentující prázdný seznam. Roli prázdného seznamu hraje symbol `nil`:

```
CL-USER 9 > '()
NIL

CL-USER 10 > (cons 1 nil)
(1)
```

Stejně jako v jazyce Scheme se funkcím, které vrací hodnotu *pravda* nebo *nepravda*, říká *predikáty*. V jazyce Scheme je obvyklé ukončovat názvy predikátů otazníkem, v Common Lispu koncovkou „`p`“ nebo „`-p`“ (i když se často setkáme s výjimkami).

3 Rozdíly ve vyhodnocovacím procesu

V Common Lispu může symbol sloužit současně jako název funkce i jako název proměnné. Tyto dvě role symbolů spolu nekolidují. V terminologii Common Lispu se říká, že každý symbol má *dvě vazby*: *hodnotovou* a *funkční*. Pomocí každé z těchto vazeb může být symbol svázán s nějakou hodnotou. V případě vazby hodnotové s libovolnou hodnotou, v případě vazby funkční s funkcí.

Je-li symbol svázán funkční vazbou s nějakou funkcí, říkáme, že je *názvem* této funkce. V případě hodnoty svázané se symbolem hodnotovou vazbou hovoříme prostě o *hodnotě symbolu* (či, volněji, *hodnotě proměnné*).

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

```
(defun encap-car (list)
  (list (car list)))
```

(pomocí makra `defun` se v Common Lispu definují funkce; viz níže uvedený doslovný překlad do Scheme). Funkce `encap-car` vrací jednoprvkový seznam obsahující první prvek seznamu `list`:

```
CL-USER 12 > (encap-car '(1 2 3))
(1)
```

Ve Scheme by doslovná analogie definice této funkce vypadala takto:

```
(define (encap-car list)                                    Scheme
  (list (car list)))                                        nesprávně
```

a volání `(encap-car '(1 2))` by skončilo chybou (proč?).

Existence dvojí vazby symbolů vede k několika komplikacím. Jde zejména o dva případy: když chceme zavolat funkci uloženou v proměnné a když chceme zjistit funkci podle jejího názvu.

K ilustraci těchto problémů nejprve uveďme následující definici procedury na kompozici dvou procedur v jazyce Scheme:

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

Analogická definice v Common Lispu by vypadala takto:

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

a vedla by k nepředvídatelným důsledkům, protože ve výrazu `(a (b x))` by se nepoužily aktuální *hodnoty* proměnných `a` a `b`, ale došlo by k pokusu aplikovat *funkce se jmény* `a` a `b`, o kterých není jasné, jestli by existovaly a pokud ano, co by dělaly.

V naší definici potřebujeme volat nikoli funkce, jejichž jména jsou `a` a `b`, ale funkce, které jsou hodnotami proměnných `a` a `b`, tedy jsou se symboly `a`, `b` svázány hodnotovou, nikoliv funkční vazbou. Pro podobné situace je v Common Lispu připravena funkce `funcall`, která volá funkci, již najde ve svém prvním argumentu. Správná definice funkce `comp` v Common Lispu je tedy následující:

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

Ke druhému problému: pomocí funkce `comp` a funkcí `car` a `cdr` lze snadno vyjádřit funkci, která vrací druhý prvek daného seznamu (tedy funkci `cadr`). Ve Scheme by takovou funkci (proceduru) vracel výraz

```
(comp car cdr)                                             Scheme
```

V Common Lispu by vyhodnocení tohoto výrazu opět vedlo k nepředvídatelným důsledkům, protože by se v něm nepoužily funkční, ale hodnotové vazby symbolů `car` a `cdr`.

Abychom získali funkce `car` a `cdr`, musíme použít speciální operátor `function`. Funkce `car` a `cdr` získáme vyhodnocením výrazů `(function car)` a `(function cdr)`. Analogický výraz v Common Lispu by tedy správně vypadal takto:

```
(comp (function car) (function cdr))
```

Pro výraz `(function name)` se používá zkratka `#'name`, takže uvedený výraz je možno napsat stručněji:

```
(comp #'car #'cdr)
```

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

```
(comp 'car 'cdr)
```

Dvojí vazby symbolů představují jediný významný rozdíl mezi vyhodnocovacím procesem Scheme a Common Lispu.

Pro zopakování: Chceme-li zavolat funkci uloženou v proměnné `a` s argumenty `p1`, ..., `pn`, nestačí jako ve Scheme napsat

```
(a p1 ... pn)                                                    Scheme
```

ale

```
(funcall a p1 ... pn)
```

Chceme-li získat funkci s názvem `f`, musíme místo prostého

```
f                                                                Scheme
```

uvést

```
#'f
```

což je zkratka pro

```
(function f)
```

Kvůli zopakování uvádíme na obrázku zjednodušený popis vyhodnocovacího procesu v Common Lispu. Tento vyhodnocovací proces je stejně jako ve Scheme rekurzivní; kdekoli se v popisu na obrázku hovoří o vyhodnocení nějakého výrazu, znamená to spuštění celého vyhodnocovacího procesu od začátku na tento výraz.

Zjednodušený vyhodnocovací proces v Lispu:

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 $a_1, \ldots, a_n$, pak

> **Jestliže o je speciální operátor,** seznam se vyhodnotí podle pravidel tohoto speciálního operátoru.

Jinak o musí být název funkce.

1. Vyhodnocením `(function` o`)` (neboli `#'`o) v prostředí *env* se zjistí funkce f,
2. zjistí se hodnoty $v_1, \ldots, v_n$ argumentů $a_1, \ldots, a_n$ v prostředí *env* (opět vyhodnocovacím procesem).
3. Výsledkem vyhodnocení je výsledek aplikace funkce f na hodnoty v_1, $\ldots$, v_n.

4 Obyčejné λ-seznamy

λ-seznam je seznam specifikující parametry funkce. U operátoru `lambda` je to první jeho argument, u makra `defun` druhý. λ-seznamům funkcí se říká *obyčejné λ-seznamy*, aby se odlišily od λ-seznamů s jiným účelem.

Nejjednodušší λ-seznam je seznam povinných parametrů. S tím jsme se už setkali.

Pokud chceme λ-seznamem stanovit, že funkce bude mít i nepovinné parametry, uděláme to pomocí klíčových slov. Základní z nich jsou `&rest`, `&optional` a `&key`.

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

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 `&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.

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 `&key`

uvádí nepovinné parametry dané jménem. Jako jméno slouží symboly začínající dvojtečkou. Ty se vyhodnocují samy na sebe.

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. Příklad:

```
(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.

Klíčová slova λ-seznamů lze i kombinovat. To zatím nebudeme potřebovat.

Otázky a úkoly na cvičení

Úlohy se týkají interpretu Scheme z konce minulého semestru přepsaného do Lispu. Interpret máte k dispozici jako zdrojový kód k této přednášce.

1. Vyřešte problém interpretu se zacyklením při pokusu o vyhodnocení symbolu bez vazby (viz jeden z testů na konci zdrojového souboru). Na hlášení chyby použijte funkci `error`: `(error "Chybová hláška")`.
2. Doplňte do interpretu speciální operátory `begin`, `let`, `let*`.
3. Vylepšete funkci `eval-pair` tak, aby ji nebylo nutné předělávat, kdykoliv chceme do interpretu přidat nový speciální operátor. Můžete například vytvořit seznam speciálních operátorů a názvů funkcí, které se spouštějí při jejich vyhodnocování a uložit jej do globální proměnné:

```
(defvar *special-operators*)
(setf *special-operators*
  '((quote . eval-quote) (if . eval-if) (define . eval-define)
    (lambda . eval-lambda)))
```