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

6. Dynamické proměnné

verze z 19. května 2019

Dynamické proměnné jsou nejstarší a nejjednodušší druh proměnných v programovacích jazycích. Ukážeme si, jak se chovají a jak je možné je implementovat v interpretu programovacího jazyka. Řekneme si o jejich výhodách a (zejména) nevýhodách. Čtenáři to snad umožní lépe pochopit a ocenit proměnné lexikální, se kterými jsme pracovali od začátku minulého semestru doteď.

1 Procedurální programování

Téma této přednášky se úzce váže k programovacímu stylu zvanému *procedurální* (někdy též *strukturované*) *programování*. Jde o jeden z nejstarších a nejjednodušších programovacích stylů.

V *procedurálním programovacím jazyce* je program rozdělen na části zvané *procedury*. Jejich podstatným rysem je, že mají vlastní parametry a lokální proměnné a že mohou být volány rekurzivně (jedna procedura druhou i jedna procedura sama sebe). Parametry procedur ani jejich lokální proměnné spolu nekolidují, i když mají stejný název.

Procedurální program se musí tedy postarat o vytváření a zpětné uvolňování prostoru pro parametry a lokální proměnné jednotlivých procedur. Jak uvidíme, k tomu je vhodné použít datovou strukturu *zásobníku*.

Před procedurálním programováním existovaly pouze programovací jazyky více méně čistě *imperativní*. Byly to strojové kódy konkrétních počítačů a první vyšší programovací jazyky, které kromě příkazů umožňovaly i zápis výpočtu formou vzorce. Jelikož tyto jazyky neobsahovaly procedury, všechny proměnné (nebo v případě asemblerů adresy konkrétních míst v paměti) měly globální rozsah. Z našeho pohledu řečeno, bylo k dispozici pouze jedno, a to globální prostředí.

Nejstarší procedurální jazyky vznikají ve druhé polovině 50. a v 60. letech minulého století: Fortran, ALGOL, COBOL. Lisp (tehdy psaný LISP) vznikl v roce 1958 a lze jej považovat za procedurální v tom smyslu, že obsahuje procedury (zvané funkce) a lokální proměnné. Kromě toho ale obsahuje i další prvky, díky kterým jej lze považovat za první *funkcionální programovací jazyk*.

2 Datový zásobník

Představme si, že chceme implementovat interpret Scheme a uvažujeme, jak zařídit, aby procedury mohly mít parametry a lokální proměnné. Zapomeneme přitom, že jeden chytrý způsob implementace (pomocí lexikálních prostředí) už známe z minulého semestru a zkusíme je naprogramovat znovu a co nejjednodušeji.

Nejprve si uvědomíme, že parametry a lokální proměnné se chovají velmi podobně. Koneckonců víme, že parametry procedur jsou lokální proměnné a také že lokální proměnné můžeme nahradit parametry. Například místo

```
(let ((x 1)
      (y 2))
  tělo)
```

můžeme použít

```
((lambda (x y)
   tělo)
 1
 2)
```

V principu se tedy stačí zabývat implementací parametrů procedur. To také zatím budeme dělat. Budeme tedy impementovat pouze prostředí obsahující vazby parametrů procedur na argumenty a samozřejmě také výchozí (globální) prostředí obsahující vazby globálních proměnných na jejich hodnoty.

Základní požadavky na parametry procedur v programovacím jazyce jsou zřejmě tyto:

1. Parametry by neměly být vidět z jiných míst programu.
2. Parametry různých procedur se mohou jmenovat stejně.
3. Týž parametr může dokonce být použit víckrát současně (při rekurzi).

Ukažme si příklad tří procedur v jazyce Scheme:

```
(define f               (define g              (define h
  (lambda (a b)           (lambda (b c)          (lambda (a)
    ...                     ...                    ...))
    (g (+ a b) b).          (h (* b c))
    ... )).                 ... ))
```

Popíšeme, jak budou vznikat a zanikat prostředí pro parametry procedur při aplikaci `(f 1 2)`.

Při aplikaci `(f 1 2)` nejprve vznikne prostředí, ve kterém budou parametry `a` a `b` navázány na hodnoty `1` a `2`.

Prostředí procedury `f`

symbol	hodnota
a	1
b	2

Toto prostředí by mělo existovat, dokud je vykonáváno tělo procedury. Během toho se ovšem zavolá procedura `g` s argumenty `3` a `2`.

Vznikne tedy další prostředí, ve kterém se vykonává tělo procedury `g`. Ta zavolá ještě proceduru `h` a vznikne ještě jedno prostředí navíc. Celkově budou prostředí vypadat takto:

Prostředí procedury `h`

symbol	hodnota
a	6

Prostředí procedury `g`

symbol	hodnota
b	3
c	2

Prostředí procedury `f`

symbol	hodnota
a	1
b	2

Prostředí jsou uložena na zásobníku, na vrcholu je poslední vzniklé prostředí, dole první. Při hledání vazby symbolu procedura postupuje od vrcholu zásobníku směrem dolů. Proto například procedura `h` vidí i vazby symbolů `b` a `c` a může je použít. Podobně procedura `g` vidí vazbu symbolu `a` v proceduře `f`.

V momentě, kdy procedura skončí a vykonávání kódu se vrátí do procedury, která ji volala, prostředí se z vrcholu zásobníku smaže.

Zdrojový kód k této přednášce obsahuje interpret Scheme s takto implementovanými parametry procedur. Interpret vznikl úpravou a podstatným zjednodušením původního interpretu s lexikálními proměnnými.

3 Viditelnost a životnost

K popisu vazeb symbolů používáme pojmy *viditelnost* (*scope*, *rozsah*) a *životnost* (*extent*). Viditelnost určuje oblast ve zdrojovém kódu, ve které je vazba symbolu platná. Může být *lexikální* nebo *neomezená*.

Lexikální viditelnost znamená, že vazba symbolu je vidět pouze v určité oblasti zdrojového kódu. V případě parametrů procedur (funkcí) je to všude v těle procedury, pokud není vazba překryta novou vazbou téhož symbolu.

Ve Scheme a v Lispu mají standardní parametry funkcí a lokální proměnné lexikální viditelnost. To platí i pro parametry a lokální proměnné většiny ostatních programovacích jazyků.

Neomezenou viditelnost mají vazby, které jsou použitelné na libovolném místě zdrojového kódu. V našem posledním interpretu Scheme mají neomezenou viditelnost globální proměnné a parametry procedur. V Lispu lze definovat proměnné s neomezenou viditelností, jak uvidíme později. Ve většině programovecích jazyků mají neomezenou viditelnost vazby globálních proměnných.

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

Životnost vazby určuje čas, po který vazba existuje. Může být omezená a neomezená.

Vazba má **omezenou životnost**, jestliže existuje moment v čase, kdy vazba zanikne. V našem posledním interpretu Scheme mají vazby parametrů procedur omezenou životnost, protože zanikají v momentě, kdy skončí vykonávání těla procedury.

Neomezená životnost vazby způsobuje, že vazba po svém vzniku nikdy nezanikne. Takto lze charakterizovat vazby, které mohou být součástí lexikálního uzávěru.

Podívejme se na tento příklad funkce v Lispu:

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

Díky tomu, že vazba parametru `x` má neomezenou životnost, funkce vrací lexikální uzávěr. Proto funguje tento test:

```
CL-USER 7 > (funcall (adder 5) 2)
7
```

Kdyby měl parametr životnost omezenou, vazba by v momentě, kdy se volá funkce vrácená funkcí `adder`, už neexistovala. Co by se stalo, by záleželo na tom, zda má symbol `x` globální vazbu. Pokud by ji neměl, došlo by k chybě. Kdyby ji měl, záležel by výsledek na hodnotě globální proměnné, nikoliv na argumentu, se kterým jsme funkci `adder` volali.

Je dobré si uvědomit, že zdrojový kód uzávěru vráceného funkcí `adder` se nachází uvnitř těla této funkce: tělem uzávěru je výraz `(+ y x)`, který je součástí těla funkce `adder`. Podle principu lexikální viditelnosti je v něm tedy vidět vazba parametru `x` vzniklá při aplikaci funkce `adder`.

Vazbám, které mají lexikální viditelnost a neomezenou životnost, se říká *lexikální vazby*. Proměnné s lexikální vazbou se nazývají *lexikální proměnné*.

Vazbám, které mají neomezenou viditelnost a omezenou životnost, se říká *dynamické vazby*. Proměnné s dynamickou vazbou se nazývají *dynamické proměnné* (v Lispu také *speciální proměnné*).

Jak jsme si už řekli, dynamické proměnné jsou starší typ lokálních proměnných, než proměnné lexikální. Na příkladě našeho interpretu jsme také viděli, že je snazší je implementovat. Mají však mnoho nevýhod, pro které se považují za zastaralé a nebezpečné. V drtivé většině současných programovacích jazyků už dynamické proměnné nenajdeme.

Ještě v nedávné době většina programovacích jazyků používala proměnné s lexikální viditelností a omezenou životností. S tím, jak se opět postupně do popředí dostává funkcionální programování, doplňuje se do programovacích jazyků možnost vytvářet lexikální uzávěry. Přechází se tedy na lexikální proměnné (s neomezenou životností).

Pojem lexikálního uzávěru je ovšem mnohem starší. Vznikl už v 60. letech minulého století a jeden z prvních jazyků, který ho začal úplně používat, byl Lisp (verze 1.5 z roku 1962).

4 Dynamické vazby v Lispu

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

1. Od momentu, kdy je symbol definován jako globální proměnná pomocí makra `defvar` (explicitně nebo implicitně např. pomocí `defparameter`), jsou všechny jeho nové vazby dynamické.
2. Vazbu symbolu můžeme učinit dynamickou pomocí vhodné *deklarace*.

Vysvětlíme obě tyto možnosti.

Proměnná definovaná makrem `defvar` je globálně označena jeko *speciální* (někdy se také poněkud nepřesně říká *dynamická*). Tak se říká proměnným, které mají vždy a za každých okolností dynamické vazby. Speciální (dynamické) jsou například už uvedené proměnné `*print-circle*` a `*print-length*`.

Následující technika by se nedala použít, kdyby proměnná `*print-length*` byla lexikální:

```
CL-USER 8 > (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 9 > (print list)
```

```
(1 2 3 4 5 6 7 8 9 10)
(1 2 3 4 5 6 7 8 9 10)

CL-USER 10 > (let ((*print-length* 5))
               (print list))

(1 2 3 4 5 ...)
(1 2 3 4 5 6 7 8 9 10)
```

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 9 10)`, protože jde o vytisknutou návratovou hodnotu funkci `print`. Ta se tiskla v době, kdy lokální 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**).

Dynamické 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.

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. (Proto bude fungovat i následující příklad.)

Hodnotu dynamických vazeb lze zjišťovat i u symbolů, které jsou zadány až za běhu programu a nejsou explicitně uvedeny ve zdrojovém kódu. Slouží k tomu funkce `symbol-value`.

Budeme pokračovat s proměnnou `x`, kterou jsme před chvílí nastavili na hodnotu `98`:

```
CL-USER 15 > x
98

CL-USER 16 > (symbol-value 'x)
98

CL-USER 17 > (defun symbol-x ()
               'x)
SYMBOL-X
```

```
CL-USER 18 > (symbol-value (symbol-x))
98
```

Funkce `symbol-value` určuje místo:

```
CL-USER 19 > (setf (symbol-value (symbol-x)) 'y)
Y

CL-USER 20 > (setf (symbol-value 'y) 'x)
X

CL-USER 21 > x
Y

CL-USER 22 > (symbol-value (symbol-value (symbol-value
(symbol-x))))
Y
```

Ještě jeden příklad:

```
(defun symbol-value-test-1 (a)
  (list a (symbol-value 'a)))
```

```
CL-USER 23 > (setf a 1)
1

CL-USER 24 > (symbol-value-test-1 2)
(2 1)
```

Funkce `symbol-value` zjiišťuje (a pomocí `setf` nastavuje) hodnotu aktuální dynamické vazby symbolu. Proto funguje následující příklad:

```
(defun symbol-value-test-2 (a)
  (declare (special a))
  (let ((a 1))
    (list (symbol-value 'a) a)))
```

```
CL-USER 25 > (symbol-value-test-2 2)
(2 1)
```

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 a maker. To se vztahuje i na lokální funkce, o kterých jsme zatím nemluvili a které lze definovat např. makrem `labels` (vzdálená obdoba schemovského `letrec`, podrobnosti si najděte ve standardu).

Otázky a úkoly na cvičení

1. Co bude hodnotou výrazu

```
(let ((x 1))
  ((let ((x 2))
     (lambda () x))))
```

   v klasickém Scheme a co ve Scheme s dynamickými vazbami?

2. Definujme ve Scheme procedury `map` a `add-to-all` takto:

```
(define map (p lst)                                    Scheme
  (if (null? lst)
    '()
    (cons (p (car lst))
          (map p (cdr lst)))))

(define add-to-all (p lst)
  (map (lambda (x) (+ p x)) lst))
```

   Co bude výsledkem aplikace `(add-to-all 1 '(1 2 3 4))` v klasickém Scheme a co ve Scheme s dynamickými vazbami?

3. Upravte interpret tak, aby pracoval s vazbami symbolů stejně jako C, tedy s lexikální viditelností a omezenou životností: z těla procedury je vidět jen na její parametry a na globální proměnné.

4. Vylepšete řešení předchozího příkladu tak, aby lexikální viditelnost fungovala i na funkce vytvořené v těle jiných funkcí, ale stále aby byla omezená životnost.