

KATEDRA INFORMATIKY, PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO, OLOMOUČ

PARADIGMATA PROGRAMOVÁNÍ 2A

VEDLEJŠÍ EFEKT

Slajdy vytvořil Vilém Vychodil

Co je vedlejší efekt?

Imperativní rysy při programování

- **funkcionální programování** = založené na postupné aplikaci procedur (funkcí)
LISP, Scheme, ML, Haskell, ...
- **procedurální programování** = založená na postupném vykonávání příkazů, které mají vedlejší efekt
C, PASCAL, FORTRAN, ...

Trend: objektivě orientované programování

- funkcionálně objektivě = založené na generických procedurách
CLOS (Common LISP Object System), GOOPS (objektový Scheme)
- procedurálně objektivě = založené na zasílání signálů
PERL, PYTHON, Java, C++, C#, ... většina „soudobých PJ“

Jaké typy vedlejších efektů rozeznáváme

hodně typů

- vstupně / výstupní operace
(některé FJ umí i bez vedlejšího efektu, třeba Haskell)
- změnou vazby symbolu
tím se budeme zabývat dnes
- destruktivní změnou datové struktury
tím se budeme zabývat příště

Připomínáme:

FJ je *čistý*, pokud programátorovi neumožňuje vytvořit vedlejší efekt

Scheme není čistý (máme `define`), Haskell je čistý

Náš interpret z Lekce 12 *byl čistý* (ačkoliv byl napsán „nečistě“).

vedlejší efekt není ve FJ na škodu, musí se ale umět používat

Čím se budeme zabývat

- obohatíme Scheme o několik ryze imperativních konstruktů
- demonstrujeme na tom rysy imperativních jazyků

Sekvencování příkazů

Speciální forma `begin`:

postupné vyhodnocení výrazů a vrácení výsledku vyhodnocení posledního z nich

```
(begin
  (+ 1 2)
  (* 3 4))
```

zde už máme vedlejší efekt: vytištění čísla na obrazovku

```
(begin
  (display (+ 1 2))
  (newline)
  (* 3 4))
```

Pozor: `(display "Ahoj")` vytiskne na obrazovku, řetězec nelze chápat jako „výsledek vyhodnocení“, tím je nedefinovaná hodnota.

Sekvencování příkazů

Chování `begin` vzhledem k prostředí

vyhodnocování výrazů probíhá v aktuálním prostředí

```
(define x 100)
```

```
(begin  
  (define x 10)  
  (* 2 x))
```

$x \implies 10$

následující je použití `begin` v novém prostředí

```
((lambda ()  
  (begin  
    (define x 10)  
    (* 2 x))))
```

`x` \implies Chyba: `x` nemá vazbu

přepis předchozího

```
(let ()  
  (begin  
    (define x 10)  
    (* 2 x)))
```

`x` \implies Chyba: `x` nemá vazbu

Těla procedur obsahují „implicitní begin“

opět se můžeme vrátit ke konceptu „jednoho výrazu v těle procedury“

```
(define f
  (lambda (x)
    (define local
      (lambda (y)
        (+ x y)))
    (local 20)))
```

```
(define f
  (lambda (x)
    (begin
      (define local
        (lambda (y)
          (+ x y)))
      (local 20))))
```

`(begin)` \implies nedefinovaná hodnota

Použití při ladění

```
(define depth-map
  (lambda (f l)
    (cond ((null? l) '())
          ((not (list? (car l)))
           (cons (f (car l)) (depth-map f (cdr l))))
          (else (cons (depth-map f (car l))
                      (depth-map f (cdr l)))))))
```

předchozí kód lze obohatit o sekvence tisknoucích argumenty sledováním hodnot na výstupu je možné odladit proceduru

Provedení příkazů pro každý prvek seznamu

```
(for-each  
  (lambda (x)  
    (display "Cislo: ")  
    (display x)  
    (newline))  
  '(10 20 30 40)) ⇒ nedefinovaná hodnota
```

```
(for-each (lambda (x) x) '(1 2 3)) ⇒ nedefinovaná hodnota  
(map (lambda (x) x) '(1 2 3)) ⇒ (1 2 3)
```

```
(for-each  
  (lambda (x y z)  
    (newline)  
    (display (list x y z))  
    (newline))  
  '(10 20) '(#t #f) '(a b))
```

Proceduru for-each snadno uděláme

for-each lze jednoduše naprogramovat (verze pro 1 seznam):
všimněte si použití if bez alternativního výrazu

```
(define for-each
  (lambda (f l)
    (if (not (null? l))
        (begin
          (f (car l))
          (for-each f (cdr l)))))))
```

a obecná verze je taky jednoduchá:

```
(define for-each
  (lambda (f . lists)
    (if (not (null? (car lists)))
        (begin
          (apply f (map car lists))
          (apply for-each f (map cdr lists)))))))
```

Příkaz přiřazení

Obecné schéma

$$LVALUE := RVALUE.$$

LVALUE ... paměťové místo (adresa)

RVALUE ... hodnota

Ve Scheme: reprezentován speciální formou `set!`

(`set!` *<symbol>* *<výraz>*)

sémantika

- 1 v hierarchii prostředí vyhledej symbol *<symbol>*, začni aktuálním (lokálním) prostředím a pokračuj přes jeho rodiče směrem ke globálnímu;
- 2 pokud *<symbol>* nemá vazbu v žádném prostředí, zahlas chybu;
- 3 v opačném případě nahraď vazbu symbolu *<symbol>* hodnotou vzniklou vyhodnocením *<výraz>*.

srovnej:

```
(define x 100)
```

```
(let ()  
  (define x 10)  
  (+ x 1))
```

$x \implies 100$

versus:

```
(define x 100)
```

```
(let ()  
  (set! x 10)  
  (+ x 1))
```

$x \implies 10$

versus:

```
(define x 100)
```

```
(let ((x 10))  
  (set! x (+ x 1))  
  (+ x 1))
```

x \implies 100

Programování s příkazem přiřazení

čisté řešení ve funkcionálním stylu:

```
(define fib
  (lambda (n)
    (let iter ((a 1)
              (b 1)
              (i n))
      (if (<= i 1)
          a
          (iter b (+ a b) (- i 1))))))
```

V jazycích jako je C, PASCAL, FORTRAN a spol. – cyklus

Programování s příkazem přiřazení

```
int
fib (n)
  int n;
{
  int a = 1, b = 1, c, i;
  for (i = n; i > 1; i --) {
    c = b;
    b = a + b;
    a = c;
  }
  return a;
}
```

můžeme otrocky přepsat i ve Scheme, není ale pěkné, ani efektivní:


```
(define fib
  (lambda (n)
    (define a 1)
    (define b 1)
    (define c 'pomocna-promenna)
    (define i n)
    (define loop
      (lambda ()
        (if (> i 1)
            (begin
              (set! c b)
              (set! b (+ a b))
              (set! a c)
              (set! i (- i 1))
              (loop))))))
    (loop)
    a))
```

na druhou stranu v C lze programovat bez vedlejšího efektu:

```
int
fib_iter (a, b, i)
    int a, b, i;
{
    if (i <= 1)
        return a;
    else
        return fib_iter (b, a + b, i - 1);
}
```

```
int
fib_wse (n)
    int n;
{
    return fib_iter (1, 1, n);
}
```

Varování

Scheme má vše k dispozici k tomu, abychom se na něj mohli dívat jako na plnohodnotný imperativní jazyk, při praktickém programování ve Scheme by ale měly převládat funkcionální rysy.

Důvod: vedlejší efekt – obrovský zdroj chyb v programech

Někdy se vedlejší efekty hodí.

```
(let ((i 0))      nastav i na 0
  (for-each
   (lambda (x)
     (display "Index: ")
     (display i)
     (display " , prvek: ")
     (display x)
     (newline)
     (set! i (+ i 1)))      zvedni i o 1
   '(a b c d e f)))
```

Procedury pracující s globálními symboly

globálně zavedený symbol

```
(define value 0)
```

procedura pracující s globálním symbolem

```
(define inc  
  (lambda (x)  
    (set! value (+ value x))  
    value))
```

```
(inc 10)  $\implies$  10
```

```
(inc 10)  $\implies$  20
```

```
(inc 10)  $\implies$  30
```

```
value  $\implies$  30
```

Dočasné překrytí lexikální vazby symbolu

Speciální forma fluid-let

```
(fluid-let ((value 200))  
  (display (inc 10))      zobrazí 210  
  (newline)  
  (display (inc 10))      zobrazí 220  
  (newline))
```

(inc 10) ... vrátí se k původní vazbě

```
(let ((value 200))  
  (display (inc 10))      pracuje s globální vazbou  
  (newline)  
  (display (inc 10))      pracuje s globální vazbou  
  (newline))
```

Procedury drží si vnitřní stav

místo globálního symbolu měníme lokální vazbu symbolu, na kterou není vidět zvenčí

```
(define inc
  (let ((value 0))
    (lambda (x)
      (set! value (+ value x))
      value))))
```

`(inc 5)` \implies 5

`(inc 5)` \implies 10

`(inc 5)` \implies 15

`value` \implies „CHYBA: symbol value nemá vazbu“

Stav (vazba symbolu `value`) je držen v prostředí vzniku procedury.

Procedury držící si vnitřní stav

Pozor, následující nefunguje!

všimněte si záměny `lambda` a `let`

```
(define inc
  (lambda (x)
    (let ((value 0))
      (set! value (+ value x))
      value)))
```

Při používání procedur s vedlejším efektem se podstatně hůř ladí už nestačí pouze testovat vstupy a výstupy procedury, musíme vždy počítat s tím, jaký má procedura zrovna vnitřní stav!

Na co je třeba dát pozor

build-list pracující zepředu

```
(define build-list
  (lambda (n f)
    (let iter ((i 0))
      (if (= i n)
          '()
          (cons (f i) (iter (+ i 1)))))))
```

build-list pracující zezadu

```
(define build-list
  (lambda (n f)
    (let iter ((i (- n 1))
              (aux '()))
      (if (< i 0)
          aux
          (iter (- i 1) (cons (f i) aux))))))
```


Na co je třeba dát pozor

Předchozí dvě verze `build-list` byly z funkcionálního hlediska nerozlišitelné. Po zavedení vedlejšího efektu již tak tomu ale není.

následující procedura ignoruje všechny argumenty a přičítá jedničku

```
(define incl
  (let ((value 0))
    (lambda args
      (set! value (+ value 1))
      value))))
```

z funkcionálního pohledu jsou obě verze `build-list` nerozlišitelné

```
(build-list 10 (lambda (x) x)) ; (0 1 2 3 4 5 6 7 8 9)
```

můžeme je ale odlišit pomocí vedlejšího efektu

```
(build-list 10 incl)  $\Rightarrow$  (1 2 3 4 5 6 7 8 9 10)
 $\Rightarrow$  (10 9 8 7 6 5 4 3 2 1)
```

Procedury generující procedury s vnitřním stavem

procedura (bez argumentu), která vrací proceduru jednoho argumentu reprezentující jeden čítač (jednu instanci čítače)

```
(define make-inc
  (lambda ()
    (let ((value 0))
      (lambda (x)
        (set! value (+ value x))
        value))))
```

```
(define i (make-inc))
(define j (make-inc))
(define k (make-inc))
```

`i` \implies #<procedure> první čítač

`j` \implies #<procedure> druhý čítač

`k` \implies #<procedure> třetí čítač

Procedury generující procedury s vnitřním stavem

(i 1) \Rightarrow 1
(i 1) \Rightarrow 2
(i 1) \Rightarrow 3
(j 5) \Rightarrow 5
(j 5) \Rightarrow 10
(j 5) \Rightarrow 15
(j 0) \Rightarrow 15
(k 10) \Rightarrow 10
(k 20) \Rightarrow 30
(k 0) \Rightarrow 30

nová verze `map`, který spolu s prvkem předává i jeho index
vyřešili jsme pomocí `map` a procedury využívající změnu stavu

```
(define map-index
  (lambda (f l)
    (let ((i -1))
      (map (lambda (x)
             (set! i (+ i 1))
             (f i x))
           l))))
```

```
(map-index cons '(a b c d e))
⇒ ((0 . a) (1 . b) (2 . c) (3 . d) (4 . e))
```

- v předchozí ukázce jsme tiše předpokládali, že `map` funguje „odpředu“
- pokud se změní samotný `map`, naše řešení přestane fungovat

ukázková verze `map`, která pracuje zezadu

```
(define map
  (lambda (f l)
    (let iter ((l (reverse l))
              (aux '()))
      (if (null? l)
          aux
          (iter (cdr l) (cons (f (car l)) aux))))))
```

```
(map-index cons '(a b c d e))
```

```
⇒ ((4 . a) (3 . b) (2 . c) (1 . d) (0 . e))
```

Reentrantní procedury

Procedura se nazývá *reentrantní*, pokud může být přerušena během jejího vykonávání (vyhodnocování těla v případě Scheme) a pak bezpečně vyvolána znovu předtím, než to předchozí vykonávání skončí. To přerušení může být způsobeno třeba skokem, nebo vyvoláním (aplikací v případě Scheme).

Pozn. Toto se běžně děje při použití rekurze.

Následující procedura není reentrantní:

```
(define length
  (let ((delka 0))
    (lambda (list)
      (set! delka 0)
      (for-each (lambda(x) (set! delka (+ delka 1)))) list)
      delka)))
```

Tu se nám ale zatím nepodaří vyvolat tak, aby se problém projevil.

Jiný příklad: reverse-map pro jeden seznam.

```
(define reverse-map1
  (let ((vysledek ()))
    (lambda (f l)
      (set! vysledek ())
      (for-each (lambda (x)
                  (set! vysledek (cons (f x) vysledek)))
                l)
      vysledek)))
```

Tady už stačí použít něco jako:

```
(reverse-map1 (lambda(x)
                (reverse-map1 - x))
              '((1) (2 3) (4)))
```

Tedy: nezneužívat vnitřní stav u procedur, které bychom rádi měli reentrantní.

Procedury sdílející týž stav

vytvoření predikátu, který si pamatuje s jakými hodnotami byl volán

```
(define make-pred
  (lambda (pred?)
    (let ((called-with-values '()))
      '(,(lambda (x y)
           (set! called-with-values
                 (cons (cons x y) called-with-values))
           (pred? x y))
        ,(lambda ()
           (reverse called-with-values))))))
```

```
(define p (make-pred <=))
((car p) 10 20)  ⇒  #t
((car p) 20 30)  ⇒  #t
((car p) 30 20)  ⇒  #f
((cadr p))       ⇒  ((10 . 20) (20 . 30) (30 . 20))
```


Použití

lze vidět, jak `mergesort` postupně porovnává prvky:

```
(let* ((pred (make-pred <=))
      (new-pred? (car pred))
      (get-called (cadr pred)))
  (display (mergesort '(4 2 8 5 6 4 5 3 5 8) new-pred?))
  (newline)
  (get-called))
```

nebo `quicksort`:

```
(let* ((pred (make-pred <=))
      (new-pred? (car pred))
      (get-called (cadr pred)))
  (display (quicksort '(4 2 8 5 6 4 5 3 5 8) new-pred?))
  (newline)
  (get-called))
```

Jednoduchý objektový systém

Objektové paradigma (zabývá se jím celý třetí semestr PP)

- my se jím nebudeme přímo zabývat
- základní myšlenka: programátor spravuje systém elementů (objektů), které mají svůj stav;
- výpočetní proces v OO jazyku je složen ze vzájemné interakce objektů a změn jejich stavu (nejznámější objektové systémy jsou založeny na zasílání (emisi) signálů a jejich příjmu (pomoci slotů), tzv. *message-passing style of programming*)

Některé problémy se pomocí OO řeší pohodlně

např. „okna, tlačítka a jiné prvky GUI“ mají svůj přirozený stav, lze je chápat jako objekty v terminologii OO.

Ukážeme – jednoduchý objektový systém umožňující vytváření instancí – to jest elementů jednoho typu, které mají vnitřní stav a komunikují s vnějším světem pomocí signálů

```
(define stack
```

```
;; data (vnitřní stav objektu)
```

```
(let ((stack '())  
      (depth 0))
```

```
(define is-empty? (lambda () ...))
```

```
(define push (lambda (elem) ...))
```

```
(define pop (lambda () ...))
```

```
(define top (lambda () ...))
```

```
;; dispatcher: aktivuje jednotlivé procedury podle jmen signálů
```

```
(lambda (signal . args)
```

```
  (cond ((equal? signal 'empty?) (is-empty?))
```

```
        ((equal? signal 'push) (push (car args)))
```

```
        ((equal? signal 'pop) (pop))
```

```
        ((equal? signal 'top) (top))
```

```
        (else (error "unknown signal"))))))
```

příklad použití

```
(stack 'push 10)
```

```
(stack 'top)      ⇒ 10
```

```
(stack 'push 20)
```

```
(stack 'top)      ⇒ 20
```

```
(stack 'push 30)
```

```
(stack 'top)      ⇒ 30
```

```
(stack 'empty?)   ⇒ #f
```

```
(stack 'pop)
```

```
(stack 'pop)
```

```
(stack 'pop)
```

```
(stack 'empty?)   ⇒ #t
```

```
(stack 'pop)      ⇒ chyba, zásobník je prázdný
```

;; je zásobník prázdný?

```
(define is-empty?  
  (lambda ()  
    (= depth 0)))
```

;; přidej prvek na vrchol zásobníku

```
(define push  
  (lambda (elem)  
    (set! stack (cons elem stack))  
    (set! depth (+ depth 1))))
```

;; odstraň prvek z vrcholu zásobníku

```
(define pop
  (lambda ()
    (if (is-empty?)
        (error "cannot perform 'pop', stack is empty")
        (begin
          (set! stack (cdr stack))
          (set! depth (- depth 1)))))))
```

;; vrať prvek nacházející se na vrcholu zásobníku

```
(define top
  (lambda ()
    (if (is-empty?)
        (error "stack is empty")
        (car stack))))
```

```
(define make-stack
  (lambda ()
    (let ((stack '())
          (depth 0))

      (define is-empty? (lambda () ...))
      (define push (lambda (elem) ...))
      (define pop (lambda () ...))
      (define top (lambda () ...))

      (lambda (signal . args)
        (cond ((equal? signal 'empty?) (is-empty?))
              ((equal? signal 'push) (push (car args)))
              ((equal? signal 'pop) (pop))
              ((equal? signal 'top) (top))
              (else (error "unknown signal"))))))))
```

```
(define s1 (make-stack))  
(define s2 (make-stack))  
(s1 'push 10)  
(s2 'push 'a)  
(s1 'push 20)  
(s2 'push 'b)  
(s1 'push 30)  
(s1 'top)       $\implies$  30  
(s2 'top)       $\implies$  b
```