
KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Jan Laštovička
https://www.inf.upol.cz/lide/jan-lastovicka
jan.lastovicka@upol.cz
17. listopadu 12, 771 46 Olomouc

Paradigmata programování 4 ◇ poznámky k přednášce

2. Paralelní Scheme

verze z 19. února 2025

1 Interpret Scheme

Nejprve si připomeneme a rozšíříme interpret jazyka Scheme představený v Paradigmatech programování 2 (PP2) a napsaný v zásobníkovém stroji. Soubor `02_stacks_4.lisp` obsahuje část jeho původní verze, kterou budeme potřebovat.

Podprogramu budeme říkat **procedura**, pokud splňuje následující podmínky:

1. Podprogram odebere z datového zásobníku dopředu stanovený počet prvků (*argumentů*).
2. Po skončení výkonu podprogramu bude na datovém zásobníku jedna nová hodnota (*návratová hodnota*).
3. Po skončení výkonu podprogramu bude zásobník vazeb ve stejném stavu jako před spuštěním.

Výraz ***`expr`*** vyhodnotí procedura `eval`:

```
expr eval :val :clsub
```

Například:

```
(+ 1 1) eval :val :clsub
```

V Lispu používáme funkci `evaluate`. Například:

```
(evaluate '(+ 1 1))
```

Interpret používá **dynamické proměnné**. Základní procedury zodpovědné za interpretaci jsou `eval` (vyhodnocení výrazu), `eval-comp` (vyhodnocení složeného výrazu) a `eval-app` (vyhodnocení aplikace).

Například `eval`:

```
(:dup :pair? :if :dup :symbol?
    :if :else                    ; konstanta
        :val :then               ; proměnná
    :else
        eval-comp :val :clsub    ; složený výraz
    :then
:ret)
```

Interpretace aplikace

Aplikace má tvar:

```
(op arg1 ... argn)
```

Vyhodnocení probíhá následovně:

1. Vyhodnotí se *arg1*, ..., *argn* na *arg1-val*, ..., *argn-val*
2. Vyhodnotí se *op* na *op-val*
3. Aplikuje se procedura *op-val* na *arg1-val*, ..., *argn-val*

Příklad:

```
(+ (* 2 2) 3)
```

Speciální operátory

Seznam speciálních operátorů se nachází v proměnné `specials`. Na speciální operátor je navázána procedura očekávající seznam nevyhodnocených argumentů zodpovědná za jeho vykonání. Například vyhodnocení speciálního operátoru `quote` vede k vykonání následujícího kódu. Na vrcholu zásobníku hodnot je jednoprvkový seznam nevyhodnocených argumentů operátoru.

```
(:split :drop :ret)
```

Použití:

```
(quote a)
```

Přehled speciálních operátorů

Scheme používá následující speciální operátory.

1. `(quote` *`expr`*`)`
2. `(if` *`test yes no`*`)`
3. `(lambda (`*`x1 ... xn`*`)` *`body1 ... bodym`*`)`
4. `(define` *`var val`*`)`

   Vytvoří vazbu *`var`* na vyhodnocený *`val`*.
5. `(begin` *`body1 ... bodyn`*`)`

   Podobně jako `progn` v Lispu, vyhodnotí postupně výrazy *`body1`*, ..., *`bodyn`* a vrátí hodnotu posledního výrazu. Přijímá aspoň jeden výraz.

Nastavení hodnoty proměnné

Přidáme speciální operátor `set!`:

```
set! specials :val :cons specials :set!
(:split :split :drop eval :val :clsub :swap :set! :ret)
set! :bind
```

Výraz `(set!` *`var val`*`)` nastaví hodnotu proměnné *`var`* na vyhodnocený *`val`*. Test:

```
1 a :bind (a (+ 1 1)) set! :val :clsub a :val :unbind
```

To samé ve Scheme (`let` definujeme za chvíli):

```
(let ((a 1))
  (set! a (+ 1 1))
  a)
```

Makra

Seznam maker se nachází v proměnné `macros`. Pro přidání maker je nutné změnit vyhodnocování složených výrazů procedurou `eval-comp`. Hodnotou makra je jeho expanzní procedura očekávající seznam nevyhodnocených argumentů. Například přidáme makro `list`. Operátor `list` musí být makrem, protože chceme, aby přijímal libovolný počet argumentů. Makro by mělo expandovat následovně.

- `(list)` ⇒ `nil`
- `(list` *`expr1 expr2 ...exprn`*`)` ⇒ `(cons` *`expr1`* `(list` *`expr2 ...exprn`*`))`

Expanzní proceduru můžeme zavolat přímo:

```
stacks > (1 2 3) list :val :clsub
(CONS 1 (LIST 2 3))
```

Přidáme si makro `defmac` zjednodušující definici maker:

```
(defmac macro expansion-procedure)
```

Makro `let`

Makro výraz:

```
(let ((p1 expr1) ... (pn exprn)) body1 ... bodym)
```

by měl expandovat na:

```
((lambda (p1 ... pn) body1 ... bodym)
  expr1 ... exprn)
```

Definice makra:

```
(defmac let (lambda (args)
              (cons
                (cons 'lambda
                      (cons
                        (mapcar first (first args))
                        (rest args)))
                (mapcar second (first args)))))
```

Speciální operátor `do-while`

Pro definování pokročilejších cyklů, si nejprve definujeme základní cyklus `do-while` jako speciální operátor. Výraz:

```
(do-while
  expr1
  ⋮
  exprn
  condition)
```

se vyhodnotí následovně.

1. Vyhodnotí výrazy *expr1*, ..., *exprn*, *condition*
2. Pokud je hodnota *condition* *Pravda*, pokračuj prvním bodem.

Příklad:

```
(let ((a 5))
  (do-while
    (print a)
    (set! a (- a 1))
    (> a 0)))
```

Makro `while`

Nyní již můžeme cyklus `while`:

```
(while condition
  expr1
  ⋮
  exprn)
```

definovat jako makro, které expanduje na:

```
(if condition
  (do-while
    expr1
    ⋮
    exprn
    condition)
  nil)
```

Makro `dotimes`

Pomocí makra `while` můžeme již snadno napsat makro `dotimes` tak, že výraz:

```
(dotimes (var count)
  expr1
  ⋮
  exprn)
```

expanduje na:

```
(let ((var 0)
      (n count))
  (while (< var n)
    expr1
    ⋮
    exprn
    (set! var (+ var 1))))
```

Aby nevznikl problém zabrání symbolu, musí být `n` nově vytvořený symbol.

Podobně jako u zásobníkového stroje, i zde máme REPL Scheme, který se spouští zavoláním Lispové funkce `scheme-repl`:

```
CL-USER 1 > (scheme-repl)
scheme > (+ 1 1)
2
scheme >
```

Rozšíření Scheme oproti verzi z PP2 se nalézají v souboru `02_stacks_5.lisp`.

Přehled procedur Scheme

Následuje přehled procedur, které máme v našem Scheme definovány.

Páry:

- `(pair? val)` ⇒ rozhodne, zda `val` je pár
- `(cons val1 val2)` ⇒ pár, kde *car* je `val1` a *cdr* je `val2`
- `(car pair)` ⇒ *car* páru `pair`
- `(cdr pair)` ⇒ *cdr* páru `pair`

Seznamy:

- `(first lst)` ⇒ první prvek seznamu
- `(second lst)` ⇒ druhý prvek seznamu
- `(third lst)` ⇒ třetí prvek seznamu
- `(fourth lst)` ⇒ čtvrtý prvek seznamu
- `(fifth lst)` ⇒ pátý prvek seznamu
- `(fourth lst)` ⇒ čtvrtý prvek seznamu
- `(rest lst)` ⇒ seznam *lst* bez prvního prvku
- `(find val lst)` ⇒ rozhodne, zda se hodnota *val* nachází v seznamu *lst*
- `(append lst1 lst2)` ⇒ spojení seznamů *lst1* a *lst2*
- `(mapcar procedure lst)` ⇒ mapování procedury *procedure* přes seznam *lst*

Porovnávání:

- `(= num1 num2)` ⇒ rozhodne, zda se čísla *num1* a *num2* rovnají
- `(<= num1 num2)` ⇒ rozhodne, zda je číslo *num1* menší nebo rovno *num2*
- `(>= num1 num2)` ⇒ rozhodne, zda je číslo *num1* větší nebo rovno *num2*
- `(< num1 num2)` ⇒ rozhodne, zda je číslo *num1* menší než *num2*
- `(> num1 num2)` ⇒ rozhodne, zda je číslo *num1* větší než *num2*

Další:

- `(print val)` ⇒ vytiskne a vrátí *val*
- `(gensym str)` ⇒ nový symbol v názvu obsahující řetězec *str*

2 Paralelní Scheme

Níže uvedený kód se budeme snažit převést do Scheme.

```
nil end? :bind
1 (:print t end? :set! :quit) :prsub
(end? :val :ret) :await
:unbind
```

Začneme modře vyznačenou částí.

Procedura `process-run-procedure`

Napíšeme si proceduru očekávající proceduru jednoho parametru a hodnotu, která zavolá obdrženou proceduru s obdrženou hodnotou jako argumentem v novém procesu. Všimněte si, že procedura se sama stará o ukončení procesu slovem `:quit`.

```
(:swap (:clsub :quit) :cons :prsub nil :ret)  process-run-procedure
:bind
```

Výše označený modrý kód nahradíme voláním právě vytvořené procedury:

```
nil end? :bind
(process-run-procedure (lambda (v)
                         (print v)
                         (set! end? t)) 1) eval :val :clsub
(end? :val :ret) :await
:unbind
```

Přejdeme k modře označenému aktivnímu čekání.

Speciální operátor `await`

Aktivní čekání umožníme přes speciální operátor `await` tak, aby vyhodnocení výrazu:

```
(await condition)
```

vedlo k vykonání kódu:

```
(condition eval :val :clsub :ret) :await nil
```

Takto vypadá definice operátoru:

```
(eval :val :clsub :ret) append :val :clsub :await nil :ret)
```

Použití `await`

Použijeme speciální operátor `await` pro aktivní čekání:

```
nil end? :bind
(process-run-procedure (lambda (v)
                         (print v)
                         (set! end? t)) 1) eval :val :clsub
(await end?) eval :val :clsub
:unbind
```

Zbývá nahradit modře vyznačený vznik vazby. K tomu můžeme použít existující makro `let`:

```
(let ((end? nil))
  (process-run-procedure (lambda (v)
                           (print v)
                           (set! end? t)) 1)
  (await end?)) eval :val :clsub
```

To už ale máme výraz Scheme:

```
(let ((end? nil))
  (process-run-procedure (lambda (v)
                           (print v)
                           (set! end? t)) 1)
  (await end?))
```

Jak přepsat pseudokód do Scheme?

Paralelní programy budeme zapisovat do tabulky tak, že souběžně vykonávaný kód bude v tabulce uveden v buňkách vedle sebe. Řádky tabulky se vykonávají postupně. Například program s jednou sdílenou proměnnou `n` s hodnotou nula, kde souběžně dva procesy nastaví `n` na hodnotu jedna a dva a poté vrátí hodnotu proměnné `n` zapíšeme tabulkou:

`let ((n 0))`	
`(set! n 1)`	`(set! n 2)`
`n`	

Právě ukázaný program bychom do Scheme přepsali následovně.

```
(let ((n 0)
      (end1? nil)
      (end2? nil))
  (process-run-procedure (lambda (arg)
                           (set! n 1)
                           (set! end1? t)) nil)
  (process-run-procedure (lambda (arg)
                           (set! n 2)
                           (set! end2? t)) nil)
  (await end1?)
  (await end2?)
  n)
```

Kód není elegantní, protože obsahuje povinné části, které se jen starají o čekání na skončení vedlejších procesů. Zmiňované části by bylo vhodnější doplnit automaticky. Tím by se kód pročistil a programátor by se o čekání nemusel starat. Zavedeme si za tímto účelem vhodné makro.

Makro `co`

Název makra je odvozen od anglického slova *concurrency*. Zápis:

```
(co body1 ... bodyn)
```

Vyhodnocení makra probíhá následovně.

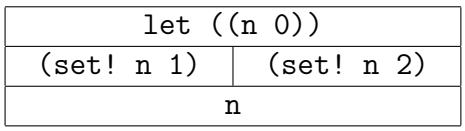
1. Vyhodnotí se výrazy ***body1***, ..., ***bodyn*** souběžně.
2. Čeká se, až se všechny výrazy vyhodnotí.
3. Vrátí se `nil`.

Makro expanduje následovně.

1. `(co)` ⇒ `nil`
2. `(co body1 body2 ... bodyn)` ⇒

```
(let ((end1? nil))
  (process-run-procedure (lambda (arg)
                           body1
                           (set! end1? t)) nil)
  (co body2 ... bodyn)
  (await end1?))
```

Musíme ošetřit zabrání symbolů `end1?`, ..., `endn?` a `arg`. S použitím makra `co` už můžeme program:



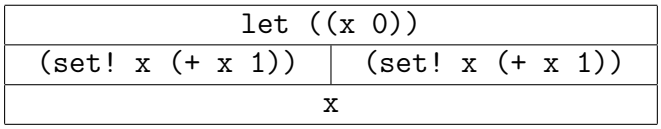
přepsat přímočaře do Scheme:

```
(let ((n 0))
  (co
    (set! n 1)
    (set! n 2))
  n)
```

Dále už nemusíme rozlišovat mezi programem Scheme a programem zapsaným tabulkou. Operátory pro paralelní Scheme jsou definované v souboru `02_co.lisp`.

3 Atomické příkazy

Zamysleme se, co může být výsledkem následujícího programu.



Zkušenosti se zásobníkovým strojem by nás měly dovést k přesvědčení, že hodnotami výrazu můžou být čísla dva a jedna. Číslo jedna obdržíme tak, že se nejprve v obou procesech získá hodnota proměnné `x` a poté dojde k nastavení `x` v obou případech na číslo jedna.

Uvažování o souběhu při vykonávání jednotlivých příkazů programu je značně náročné a proto si dále ukážeme jaké příkazy můžeme používat, aby k souběhu nedocházelo.

Jednoduchým výrazem rozumíme

1. atom (symbol, číslo, řetězec, ...) nebo
2. složený výraz:

```
(procedure expr1 ... exprn)
```

   kde *expr1* ... *exprn* jsou jednoduché výrazy a *procedure* je název procedury bez vedlejšího efektu.

Příklady:

- `x`, `1`
- `(+ x 2)`, `(+ x x)`, `(= (+ x 1) y)`

Jednoduchý výraz se **vyhodnotí atomicky** pokud

1. obsahuje nejvýše jednu proměnnou, která může být změněna v jiném procesu,
2. a obsahuje ji nejvýše jednou.

Abychom se přiblížili imperativnímu stylu programování, zavedeme si následující **příkazy**. Příkaz přiřazení:

```
(set! var expr)
```

Mění hodnotu proměnné *var*. Příkaz aktivního čekání:

```
(await condition)
```

Příkaz tisku:

```
(print expr)
```

Uvažujeme jen jednoduché výrazy v příkazech.

Atomické vykonání příkazů

Příkaz

- `(set! var expr)`

se vykoná **atomicky**, pokud

1. výraz *expr* se vyhodnotí atomicky a proměnná *var* není čtena ani měněna v jiném procesu
2. nebo *expr* neobsahuje žádnou proměnnou, která může být změněna v jiném procesu.

Příkazy

- `(await condition)`
- `(print expr)`

se vykonají **atomicky**, pokud se výrazy *condition* a *expr* vyhodnotí atomicky. V paralelním programu budeme používat pouze příkazy, které se vykonají atomicky.

Příkazy v následujícím programu se nevykonají atomicky.

let ((x 0))	
(set! x (+ x 1))	(set! x (+ x 1))
x	

Neatomické příkazy přiřazení vždy můžeme rozlomit na několik příkazů, které se již vykonají atomicky. Stačí k tomu zavést pomocné proměnné. Například rozlomením příkazů dostáváme ekvivalentní program s atomickými příkazy:

let ((x 0))	
(let ((tmp nil)) (set! tmp (+ x 1)) (set! x tmp))	(let ((tmp nil)) (set! tmp (+ x 1)) (set! x tmp))
x	

K souběhu procesů většinou dochází zřídka. K zvýšení pravděpodobnosti souběhu necháme problematický kód několikrát zopakovat. Například u následujícího programu už často dochází k souběhu u souběžného zvyšování hodnoty proměnné.

```
(let ((n 0))
  (co
   (let ((tmp nil))
     (dotimes (i 100)
       (set! tmp (+ n 1))
       (set! n tmp)))
   (let ((tmp nil))
     (dotimes (i 100)
       (set! tmp (+ n 1))
       (set! n tmp))))
  n)
```

4 Historie

Pro zjednodušení uvažování nad paralelními programy bývá často užitečné chápat paralelní program jako množinu procesů, kde každý postupně vykonává atomické příkazy. **Stav paralelního programu** je pak tvořen hodnotami všech proměnných a ukazateli na následující atomický příkaz všech procesů.

Například uvažujme program:

`let ((x 0))`	
p	q
`(let ((tmp1 nil))`	`(let ((tmp2 nil))`
1 : `(set! tmp1 (+ x 1))`	1 : `(set! tmp2 (+ x 1))`
2 : `(set! x tmp1))`	2 : `(set! x tmp2))`

Pro lepší orientaci jsme si v kódu pojmenovali oba procesy písmeny p a q a příkazy očíslovali čísli 1 a 2. Stavy programu můžeme zachytit tabulkou o dvou sloupcích, kde první sloupec udává názvy proměnných a procesy a druhý hodnoty proměnných a následující atomický příkaz procesů. V případě, že proces již skončil, příslušné políčko proškrtneme. Příklady stavů předchozího programu:

x	0	x	0	x	1	x	1
tmp1	nil	tmp1	1	tmp1	1	tmp1	1
tmp2	nil	tmp2	nil	tmp2	nil	tmp2	1
p	1	p	2	p	/	p	/
q	1	q	1	q	1	q	/

Počátečním stavem paralelního programu nazveme stav, ve kterém začíná vykonávání programu. Stav paralelního programu nazveme **koncovým stavem**, pokud v něm může skončit vykonávání paralelního programu.

Například:

`let ((x 0))`	
p	q
`1 : (set! x 1)`	`1 : (set! x 2)`

má počáteční stav:

`x`	`0`
`p`	`1`
`q`	`1`

a dva koncové stavy:

`x`	`1`		`x`	`2`	
`p`	`/`	,	`p`	`/`	.
`q`	`/`		`q`	`/`	

Historie paralelního programu je posloupnost stavů programu zachycující jeden možný běh programu.

Například:

`let ((x 0))`	
p	q

`(let ((tmp1 nil))`
`1 : (set! tmp1 (+ x 1))`
`2 : (set! x tmp1))`
	`(let ((tmp2 nil))`
	`1 : (set! tmp2 (+ x 1))`
	`2 : (set! x tmp2))`

Jedna z možných historií:

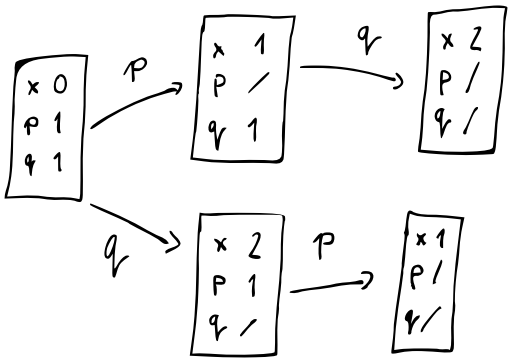
`x`	`0`	`0`	`1`	`1`	`2`
`tmp1`	`nil`	`1`	`1`	`1`	`1`
`tmp2`	`nil`	`nil`	`nil`	`2`	`2`
`p`	`1`	`2`	`/`	`/`	`/`
`q`	`1`	`1`	`1`	`2`	`/`

Graf paralelního programu je orientovaný graf, kde vrcholy grafu jsou stavy programu a platí, že z vrcholu S_1 vede hrana do vrcholu S_2, právě když lze ze stavu S_1 přejít do stavu S_2 vykonáním atomického příkazu nějakého procesu. Pro přehlednost můžeme hrany označit procesem, jehož příkaz jsme vykonali. Konečné historie paralelního programu odpovídají sledům v jeho grafu od počátečního vrcholu ke koncovému. Nekonečné historie odpovídají nekonečným sledům z počátečního vrcholu.

Například program:

let ((x 0))	
p	q
1 : (set! x 1)	1 : (set! x 2)

má graf:



Otázky a úkoly na cvičení

1. Rozhodněte, zda u následujících programů se všechny příkazy vykonají atomicky. Pokud tomu tak není, rozlomte neatomické příkazy na atomické.

   (a)

let ((x 0) (y 1))	
(set! x 1)	(set! y (+ x y))
y	

   (b)

let ((x 0) (y 0))	
(set! x 1)	(set! y (+ x x))
y	

   (c)

let ((x 1) (y 2))	
(set! x y)	(set! y x)
(list x y)	

   (d)

let ((x 0) (y 1) (z 1))	
(set! z (+ x y))	(set! x (+ y y))
(list x y z)	

2. Nakreslete graf stavů následujícího programu.

`let ((x 1) (y 2) (a nil) (b nil))`	
p	q
1 : `(set! a y)`	1 : `(set! b x)`
2 : `(set! x a)`	2 : `(set! y b)`

3. Naprogramujte předchozí program a zkuste docílit všech možných koncových stavů. K zvýšení pravděpodobnosti některých historií může být výhodné použít následující proceduru.

```
(define sleep (lambda (n)
                (dotimes (i n)
                  nil)))
```

4. Synchronizujte procesy v předchozím programu tak, aby vždy došlo k prohození hodnot proměnných `x` a `y`.

5. Nakreslete graf stavů pro program z předchozího úkolu.

6. Jaké jsou možné historie následujícího programu? Příkaz 1 v procesu p chápeme jako příkaz k vyhodnocení podmínky cyklu.

`let ((x t) (y 0))`	
p	q
1 : `(while x`	1 : `(set! x nil)`
2 : `  (set! y 1))`	