Paradigmata programování 4

Přednáška 2. Paralelní Scheme

verze z 18. února 2025

Jan Laštovička

KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

Obsah

Procedury

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.

Interpret Scheme

Interpret je napsaný v zásobníkovém jazyce.

Výraz *expr* vyhodnotí procedura `eval`:

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

Například:

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

Interpret Scheme

Interpret je napsaný v zásobníkovém jazyce.

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 Scheme

Interpret je napsaný v zásobníkovém jazyce.

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.

Interpret Scheme

Procedury zodpovědné za interpretaci:

- `eval` ... vyhodnocení výrazu
- `eval-comp` ... vyhodnocení složeného výrazu
- `eval-app` ... vyhodnocení aplikace

Interpret Scheme

Procedury zodpovědné za interpretaci:

- `eval` ... vyhodnocení výrazu
- `eval-comp` ... vyhodnocení složeného výrazu
- `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

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

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

V proměnné `specials`.

Na speciální operátor je navázána procedura očekávající seznam nevyhodnocených argumentů zodpovědná za jeho vykonání.

Speciální operátory

V proměnné `specials`.

Na speciální operátor je navázána procedura očekávající seznam nevyhodnocených argumentů zodpovědná za jeho vykonání. Například `quote`:

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

Použití:

```
(quote a)
```

Přehled speciálních operátorů

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

Nastavení hodnoty proměnné

Přidáme speciální operátor `set!`:

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

Operátor nevyhodnocuje první argument.

Nastavení hodnoty proměnné

Přidáme speciální operátor `set!`:

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

Operátor nevyhodnocuje první argument.

Test:

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

Nastavení hodnoty proměnné

Přidáme speciální operátor `set!`:

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

Operátor nevyhodnocuje první argument.

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

- v proměnné `macros`
- nutné změnit `eval-comp`
- Hodnotou makra je jeho expanzní procedura očekávající seznam argumentů.

Makra

- v proměnné `macros`
- nutné změnit `eval-comp`
- Hodnotou makra je jeho expanzní procedura očekávající seznam argumentů.

Například:

- `(list)` $\Rightarrow$ `nil`
- `(list` *`expr1 expr2 ...exprn`*`)` $\Rightarrow$ `(cons` *`expr1`* `(list` *`expr2 ...exprn`*`))`

Makra

- v proměnné `macros`
- nutné změnit `eval-comp`
- Hodnotou makra je jeho expanzní procedura očekávající seznam argumentů.

Například:

- `(list)` $\Rightarrow$ `nil`
- `(list` *`expr1 expr2 ...exprn`*`)` $\Rightarrow$ `(cons` *`expr1`* `(list` *`expr2 ...exprn`*`))`

Test expanze:

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

Makra

- v proměnné `macros`
- nutné změnit `eval-comp`
- Hodnotou makra je jeho expanzní procedura očekávající seznam argumentů.

Například:

- `(list)` $\Rightarrow$ `nil`
- `(list` *`expr1 expr2 ...exprn`*`)` $\Rightarrow$ `(cons` *`expr1`* `(list` *`expr2 ...exprn`*`))`

Test expanze:

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

Makro na definici maker:

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

Makro `let`

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

Makro `let`

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

$\Rightarrow$

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

Makro let

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

$\Rightarrow$

```
((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

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

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

Speciální operátor do-while

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

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`

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

$\Rightarrow$

Makro `while`

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

$\Rightarrow$

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

Makro dotimes

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

$\Rightarrow$

Makro dotimes

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

⇒

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

n... generovaný symbol

Obsah

Jak vyjádřit následující kód ve Scheme?

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

Jak vyjádřit následující kód ve Scheme?

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

Procedura process-run-procedure

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

Procedura process-run-procedure

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

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

Procedura process-run-procedure

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

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

Procedura process-run-procedure

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

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

Speciální operátor `await`

```
(await condition)
```

vede k:

Speciální operátor `await`

```
(await condition)
```

vede k:

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

Speciální operátor `await`

```
(await condition)
```

vede k:

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

Definice:

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

Použití await

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

Použití await

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

Použití await

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

Použití let

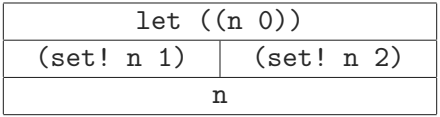
```
(let ((end? nil))
  (process-run-procedure (lambda (v)
                           (print v)
                           (set! end? t)) 1)
  (await end?)) eval :val :clsub
```

Použití `let`

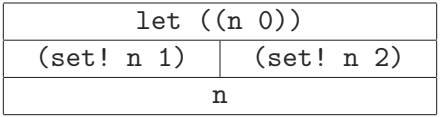
```
(let ((end? nil))
  (process-run-procedure (lambda (v)
                           (print v)
                           (set! end? t)) 1)
  (await end?))
```

Už jsme v Scheme.

Jak přepsat pseudokód do Scheme?

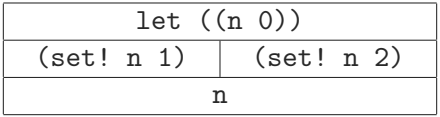


Jak přepsat pseudokód do Scheme?




```
(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)
```

Jak přepsat pseudokód do Scheme?




```
(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)
```

Lze to zlepšit?

Makro co

Makro co

- název odvozen od: concurrency

Makro co

- název odvozen od: concurrency

Zápis:

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

Makro co

- název odvozen od: concurrency

Zápis:

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

1. Vyhodnotí se výrazy *body1*, ..., *bodyn* souběžně.

Makro co

- název odvozen od: concurrency

Zápis:

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

1. Vyhodnotí se výrazy *body1*, ..., *bodyn* souběžně.
2. Čeká se, až se všechny výrazy vyhodnotí.

Makro co

- název odvozen od: concurrency

Zápis:

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

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

Expanze makra co

```
1 (co)
```

Expanze makra co

```
1 (co) ⇒ nil
```

Expanze makra co

1. `(co)` $\Rightarrow$ `nil`
2. `(co` *`body1 body2 ... bodyn`*`)`

Expanze makra co

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?))
```

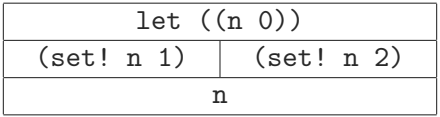
Expanze makra co

1. `(co)` $\Rightarrow$ `nil`
2. `(co` *`body1 body2 ... bodyn`*`)` $\Rightarrow$

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

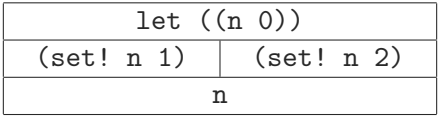
Musíme ošetřit zabrání symbolů `end1?`, ..., `endn?` a `arg`.

Použití makra co




```
(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)
```

Použití makra co

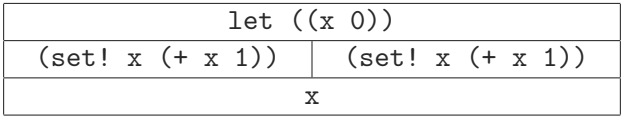



```
(let ((n 0))
  (co
    (set! n 1)
    (set! n 2))
  n)
```

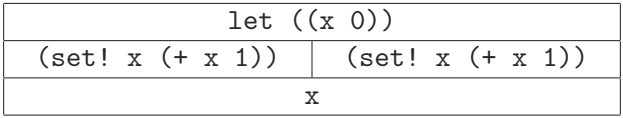
Obsah

Co může být výsledkem?

Co může být výsledkem?

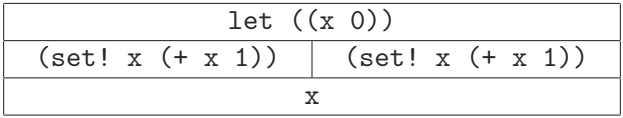


Co může být výsledkem?



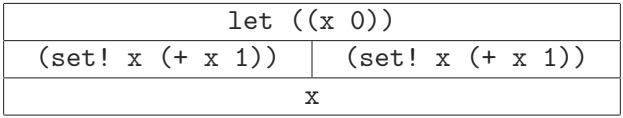
Možné hodnoty proměnné x:

Co může být výsledkem?



Možné hodnoty proměnné x: 2

Co může být výsledkem?



Možné hodnoty proměnné x: 2, 1

Jednoduchý výraz

Jednoduchý výraz

je

1. atom (symbol, číslo, řetězec, . . .)

Jednoduchý výraz

je

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.

Jednoduchý výraz

je

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

je

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

Jednoduchý výraz

je

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,

Jednoduchý výraz

je

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.

Příkazy

Příkazy

Příkaz přiřazení:

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

Mění hodnotu proměnné *var*.

Příkazy

Příkaz přiřazení:

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

Mění hodnotu proměnné *var*.

Příkaz aktivního čekání:

```
(await condition)
```

Příkazy

Příkaz přiřazení:

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

Mění hodnotu proměnné *var*.

Příkaz aktivního čekání:

```
(await condition)
```

Příkaz tisku:

```
(print expr)
```

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

Atomické vykonání příkazů

Příkaz

- `(set!` *var expr*`)`

se vykoná atomicky, pokud

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

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.

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.

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.

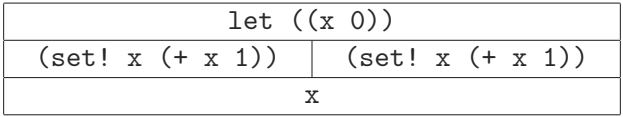
Příkazy

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

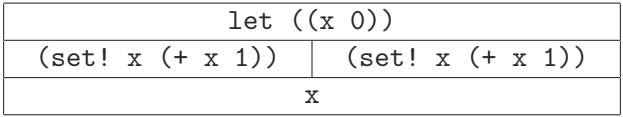
se vykonají atomicky, pokud se výrazy *condition* a *expr* vyhodnotí atomicky.

Budeme používat pouze příkazy, které se vykonají atomicky.

Vykonají se příkazy atomicky?

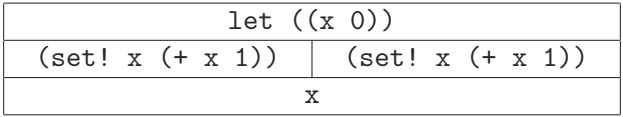


Vykonají se příkazy atomicky?



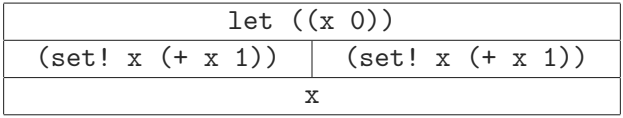
Ne.

Vykonají se příkazy atomicky?

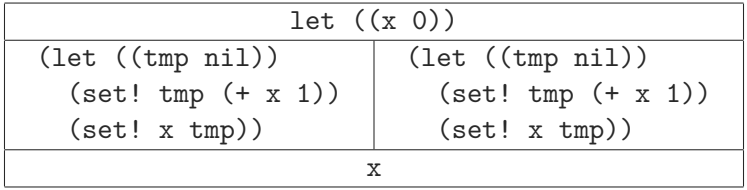


Ne. Rozlomíme na atomické:

Vykonají se příkazy atomicky?



Ne. Rozlomíme na atomické:



Pokročilejší test

```
(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)
```

Jaké jsou možné hodnoty?

Obsah

Stav paralelního programu

Stav paralelního programu

- je tvořen hodnotami všech proměnných
- a ukazateli na následující atomický příkaz všech procesů.

Stav paralelního programu

- je tvořen hodnotami všech proměnných
- a ukazateli na následující atomický příkaz všech procesů.

Například:

`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))`

Stav paralelního programu

- je tvořen hodnotami všech proměnných
- a ukazateli na následující atomický příkaz všech procesů.

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))`

Stavy:

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	/

Počáteční a koncový stav

Počáteční a koncový stav

počáteční stav paralelního programu

- je stav, ve kterém začíná vykonávání programu.

Počáteční a koncový stav

počáteční stav paralelního programu

- je stav, ve kterém začíná vykonávání programu.

koncový stav paralelního programu

- je stav, ve kterém může skončit vykonávání programu.

Počáteční a koncový stav

počáteční stav paralelního programu

- je stav, ve kterém začíná vykonávání programu.

koncový stav paralelního programu

- je stav, ve kterém může skončit vykonávání programu.

Například:

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

Počáteční a koncový stav

počáteční stav paralelního programu

- je stav, ve kterém začíná vykonávání programu.

koncový stav paralelního programu

- je stav, ve kterém může skončit vykonávání programu.

Například:

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

Počáteční stav:

x	0
p	1
q	1

Koncové stavy:

x	1
p	/
q	/

x	2
p	/
q	/

Historie paralelního programu

Historie paralelního programu

- je posloupnost stavů programu
- zachycující jeden možný běh programu.

Historie paralelního programu

- je posloupnost stavů programu
- zachycující jeden možný běh programu.

Například:

`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))`

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

Graf paralelního programu

- orientovaný graf
- vrcholy jsou stavy programu
- 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.

Graf paralelního programu

- orientovaný graf
- vrcholy jsou stavy programu
- 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 = sled od počátečního vrcholu ke koncovému
- nekonečná historie = nekonečný sled z počátečního vrcholu

Příklad

Příklad

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

Příklad

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