Paradigmata programování 1

Přednáška 12. Interpret Scheme

Michal Krupka

KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

Obsah

Interpret a kompilátor

Interpret a kompilátor

Interpret (*interpretr*) programovacího jazyka: vykonává program krok po kroku bez překladu do jiného jazyka.

Interpret a kompilátor

Interpret (*interpretr*) programovacího jazyka: vykonává program krok po kroku bez překladu do jiného jazyka.

Kompilátor (*překladač*) programovacího jazyka: převádí program z jednoho jazyka do druhého.

Zásady

1. Při interpretaci programu budeme využívat datové struktury jazyka Scheme (páry, symboly, čísla ...).

Zásady

1. Při interpretaci programu budeme využívat datové struktury jazyka Scheme (páry, symboly, čísla ...).
2. Dále budeme využívat primitiva jazyka Scheme (např. proceduru na sčítání čísel).

Zásady

1. Při interpretaci programu budeme využívat datové struktury jazyka Scheme (páry, symboly, čísla ...).
2. Dále budeme využívat primitiva jazyka Scheme (např. proceduru na sčítání čísel).
3. Program k interpretaci budeme zadávat jako datovou strukturu jazyka Scheme (seznam), nikoliv v textové podobě, jak je obvyklé. Tím odpadne potřeba analyzovat textový zápis programu.

Procedura `eval`: příklady

```
> (eval '(+ 1 2))
```

Procedura `eval`: příklady

```
> (eval '(+ 1 2))
3
```

Procedura `eval`: příklady

```
> (eval '(+ 1 2))
3
> (eval '((lambda (x y) x) 1 2))
```

Procedura `eval`: příklady

```
> (eval '(+ 1 2))
3
> (eval '((lambda (x y) x) 1 2))
1
```

Procedura `eval`: příklady

```
> (eval '(+ 1 2))
3
> (eval '((lambda (x y) x) 1 2))
1
> (eval '(let ((x 1) (y 2))
           (+ x y)))
```

Procedura `eval`: příklady

```
> (eval '(+ 1 2))
3
> (eval '((lambda (x y) x) 1 2))
1
> (eval '(let ((x 1) (y 2))
           (+ x y)))
3
```

Procedura eval: příklady

```
> (eval '(+ 1 2))
3
> (eval '((lambda (x y) x) 1 2))
1
> (eval '(let ((x 1) (y 2))
           (+ x y)))
3
> (eval '(define proc (lambda (x y) (+ x y))))
```

Procedura `eval`: příklady

```
> (eval '(+ 1 2))
3
> (eval '((lambda (x y) x) 1 2))
1
> (eval '(let ((x 1) (y 2))
           (+ x y)))
3
> (eval '(define proc (lambda (x y) (+ x y))))
> (eval '(proc 1 2))
```

Procedura eval: příklady

```
> (eval '(+ 1 2))
3
> (eval '((lambda (x y) x) 1 2))
1
> (eval '(let ((x 1) (y 2))
           (+ x y)))
3
> (eval '(define proc (lambda (x y) (+ x y))))
> (eval '(proc 1 2))
3
```

Procedura `eval`: příklady

```
> (eval '(+ x y) env)
3
> (eval '((lambda (x) (* x y)) 5) env)
10
```

Procedura `eval`: příklady

```
> (eval '(+ x y) env)
3
> (eval '((lambda (x) (* x y)) 5) env)
10
```

Napíšeme ji přesně podle pravidel vyhodnocovacího procesu Scheme.

Způsob práce

1. Dělit složitější problémy na jednodušší.

Způsob práce

1. Dělit složitější problémy na jednodušší.
2. Stále udržovat funkční program.

Obsah

Procedura `eval`

Procedura `eval`

Výchozí prostředí

```
(define initial-env ())
```

Procedura `eval`

Výchozí prostředí

```
(define initial-env ())
```

Procedura `eval`

Procedura `eval`

Výchozí prostředí

```
(define initial-env ())
```

Procedura `eval`

```
(define (eval expr . env-lst)
```

Procedura eval

Výchozí prostředí

```
(define initial-env ())
```

Procedura eval

```
(define (eval expr . env-lst)
  (let ((env (if (null? env-lst) initial-env (car env-lst))))
    (if (pair? expr)
        (eval-pair expr env)
        (eval-atom expr env))))
```

Vylepšená procedura eval

```
(define (optional-arg-val arg-lst default)
  (if (null? arg-lst) default (car arg-lst)))

(define (eval expr . env-lst)
  (let ((env (optional-arg-val env-lst initial-env)))
    (if (pair? expr)
        (eval-pair expr env)
        (eval-atom expr env))))
```

Obsah

Vyhodnocení atomu

Vyhodnocení atomu

```
(define (eval-atom atom env)
```

Vyhodnocení atomu

```
(define (eval-atom atom env)
  (if (symbol? atom)
      (symbol-value atom env)
      atom))
```

Vyhodnocení atomu

```
(define (eval-atom atom env)
  (if (symbol? atom)
      (symbol-value atom env)
      atom))
```

Test:

Vyhodnocení atomu

```
(define (eval-atom atom env)
  (if (symbol? atom)
      (symbol-value atom env)
      atom))
```

Test:

```
> (eval 1)
1
> (eval #t)
#t
> (eval ())
()
```

Obsah

Reprezentace prostředí

Víme, že prostředí obsahuje:

Reprezentace prostředí

Víme, že prostředí obsahuje:

1. Svého předka,

Reprezentace prostředí

Víme, že prostředí obsahuje:

1. Svého předka,
2. tabulku vazeb symbolů.

Reprezentace prostředí

Víme, že prostředí obsahuje:

1. Svého předka,
2. tabulku vazeb symbolů.

Reprezentace prostředí

Víme, že prostředí obsahuje:

1. Svého předka,
2. tabulku vazeb symbolů.

Prostředí

```
(env bindings parent)
```

bindings: seznam vazeb jako párů (*symbol* . *value*)
parent: předek prostředí (jiné prostředí) nebo #f

Konstruktor a selektory prostředí

Konstruktor a selektory prostředí

```
(define (make-env bindings . parent-lst)
  (list 'env bindings (optional-arg-val parent-lst #f)))
```

Konstruktor a selektory prostředí

```
(define (make-env bindings . parent-lst)
  (list 'env bindings (optional-arg-val parent-lst #f)))

(define (env-bindings env)
```

Konstruktor a selektory prostředí

```
(define (make-env bindings . parent-lst)
  (list 'env bindings (optional-arg-val parent-lst #f)))

(define (env-bindings env)
  (cadr env))
```

Konstruktor a selektory prostředí

```
(define (make-env bindings . parent-lst)
  (list 'env bindings (optional-arg-val parent-lst #f)))

(define (env-bindings env)
  (cadr env))

(define (env-parent env)
```

Konstruktor a selektory prostředí

```
(define (make-env bindings . parent-lst)
  (list 'env bindings (optional-arg-val parent-lst #f)))

(define (env-bindings env)
  (cadr env))

(define (env-parent env)
  (caddr env))
```

Procedura `symbol-value`

Procedura `symbol-value`

má zjistit hodnotu vazby symbolu v daném prostředí. To obnáší:

Procedura `symbol-value`

má zjistit hodnotu vazby symbolu v daném prostředí. To obnáší:

1. Zjistit, zda má symbol vazbu v daném prostředí.

Procedura `symbol-value`

má zjistit hodnotu vazby symbolu v daném prostředí. To obnáší:

1. Zjistit, zda má symbol vazbu v daném prostředí.
2. Pokud nemá, hledat vazbu rekurzivně v předkovi prostředí.

Procedura `symbol-value`

má zjistit hodnotu vazby symbolu v daném prostředí. To obnáší:

1. Zjistit, zda má symbol vazbu v daném prostředí.
2. Pokud nemá, hledat vazbu rekurzivně v předkovi prostředí.
3. Vrátit hodnotu nalezené vazby.

Procedura `symbol-value`

Procedura symbol-value

```
(define (symbol-value env symbol)
```

Procedura symbol-value

```
(define (symbol-value env symbol)
  (cdr (symbol-binding env symbol)))
```

Procedura symbol-value

```
(define (symbol-value env symbol)
  (cdr (symbol-binding env symbol)))
```

```
(define (symbol-binding env symbol)
```

Procedura symbol-value

```
(define (symbol-value env symbol)
  (cdr (symbol-binding env symbol)))
```

```
(define (symbol-binding env symbol)
  (let ((bnd (symbol-binding-1 env symbol)))
    (if (pair? bnd)
        bnd
        (symbol-binding (env-parent env) symbol))))
```

Testy

```
> (define env-1 (make-env '((a . 1))))
> env-1
(env ((a . 1)) #f)
> (define env-2 (make-env '((b . 2)) env-1))
> env-2
(env ((b . 2)) (env ((a . 1)) #f))
> (symbol-binding-1 'a env-1)
(a . 1)
> (symbol-binding-1 'a env-2)
#f
> (symbol-binding 'a env-2)
(a . 1)
> (symbol-value 'a env-2)
1
> (symbol-value 'b env-2)
2
```

Testy

```
> (eval 'a env-2)
1
> (eval 'b env-2)
2
```

Výchozí prostředí

Výchozí prostředí

```
(define initial-env (make-env (list (cons 'zero 0)
                                    (cons '+ +)
                                    (cons '* *)
                                    (cons '> >)
                                    (cons 'cons cons))))
```

Výchozí prostředí

```
(define initial-env (make-env (list (cons 'zero 0)
                                    (cons '+ +)
                                    (cons '* *)
                                    (cons '> >)
                                    (cons 'cons cons))))
```

Test:

Výchozí prostředí

```
(define initial-env (make-env (list (cons 'zero 0)
                                    (cons '+ +)
                                    (cons '* *)
                                    (cons '> >)
                                    (cons 'cons cons))))
```

Test:

```
> (eval 'zero)
0
> (eval '+)
#<procedure:+>
> (eval 'cons)
#<procedure:cons>
```

Obsah

Vyhodnocení (neprázdného) seznamu

Vyhodnocení (neprázdného) seznamu

```
(define (eval-pair pair env)
```

Vyhodnocení (neprázdného) seznamu

```
(define (eval-pair pair env)
  (let ((op (car pair))
        (args (cdr pair)))
    (cond ((eq? op 'quote) (eval-quote args env))
          ((eq? op 'if) (eval-if args env))
          ((eq? op 'define) (eval-define args env))
          ((eq? op 'lambda) (eval-lambda args env))
          (#t (eval-application op args env)))))
```

Vyhodnocení aplikace

Vyhodnocení aplikace

```
(define (eval-application op args env)
```

Vyhodnocení aplikace

```
(define (eval-application op args env)
  (apply-proc (eval op env)
              (map (lambda (arg) (eval arg env))
                   args)))
```

Vyhodnocení aplikace

```
(define (eval-application op args env)
  (apply-proc (eval op env)
              (map (lambda (arg) (eval arg env))
                   args)))
```

Lepší verze:

```
(define (eval-list-elements lst env)
```

Vyhodnocení aplikace

```
(define (eval-application op args env)
  (apply-proc (eval op env)
              (map (lambda (arg) (eval arg env))
                   args)))
```

Lepší verze:

```
(define (eval-list-elements lst env)
  (map (lambda (el) (eval el env))
       lst))
```

Vyhodnocení aplikace

```
(define (eval-application op args env)
  (apply-proc (eval op env)
              (map (lambda (arg) (eval arg env))
                   args)))
```

Lepší verze:

```
(define (eval-list-elements lst env)
  (map (lambda (el) (eval el env))
       lst))

(define (eval-application op args env)
```

Vyhodnocení aplikace

```
(define (eval-application op args env)
  (apply-proc (eval op env)
              (map (lambda (arg) (eval arg env))
                   args)))
```

Lepší verze:

```
(define (eval-list-elements lst env)
  (map (lambda (el) (eval el env))
       lst))

(define (eval-application op args env)
  (apply-proc (eval op env)
              (eval-list-elements args env)))
```

Vyhodnocení aplikace

Vyhodnocení aplikace

```
(define (primitive? proc)
```

Vyhodnocení aplikace

```
(define (primitive? proc)
  (procedure? proc))
```

Vyhodnocení aplikace

```
(define (primitive? proc)
  (procedure? proc))
```

```
(define (apply-proc proc args)
```

Vyhodnocení aplikace

```
(define (primitive? proc)
  (procedure? proc))
```

```
(define (apply-proc proc args)
  (if (primitive? proc)
      (apply-primitive proc args)
      (apply-user-proc proc args)))
```

Aplikace primitiva

Aplikace primitiva

```
(define (apply-primitive proc args)
```

Aplikace primitiva

```
(define (apply-primitive proc args)
  (apply proc args))
```

Aplikace primitiva

```
(define (apply-primitive proc args)
  (apply proc args))
```

Test:

Aplikace primitiva

```
(define (apply-primitive proc args)
  (apply proc args))
```

Test:

```
> (eval '(+ 1 2))
3
> (eval '(cons (+ 1 2) (* 3 4)))
(3 . 12)
```

Aplikace primitiva

```
(define (apply-primitive proc args)
  (apply proc args))
```

Test:

```
> (eval '(+ 1 2))
3
> (eval '(cons (+ 1 2) (* 3 4)))
(3 . 12)
```

Obsah

quote

quote

```
(define (eval-quote args env)
```

quote

```
(define (eval-quote args env)
  (car args))
```

quote

```
(define (eval-quote args env)
  (car args))
```

Test:

quote

```
(define (eval-quote args env)
  (car args))
```

Test:

```
> (eval '(quote a))
a
> (eval '(quote (a b c)))
(a b c)
> (eval ''x)
x
```

if

if

```
(define (eval-if args env)
```

if

```
(define (eval-if args env)
  (if (eval (car args) env)
      (eval (cadr args) env)
      (eval (caddr args) env)))
```

if

```
(define (eval-if args env)
  (if (eval (car args) env)
      (eval (cadr args) env)
      (eval (caddr args) env)))
```

Test:

if

```
(define (eval-if args env)
  (if (eval (car args) env)
      (eval (cadr args) env)
      (eval (caddr args) env)))
```

Test:

```
> (eval '(if #t 1 2))
1
> (eval '(if #f 1 2))
2
> (eval '(if (> 1 2) (+ 1 2) (* 3 4)))
12
```

define

define

Procedura set-car!:

define

Procedura set-car!:

```
> (define c (cons 1 2))
> c
(1 . 2)
> (set-car! c 3)
> c
(3 . 2)
```

define:

define

Procedura `set-car!`:

```
> (define c (cons 1 2))
> c
(1 . 2)
> (set-car! c 3)
> c
(3 . 2)
```

`define`:

```
(define (add-binding! env symbol value)
  (set-car! (cdr env)
            (cons (cons symbol value)
                  (env-bindings env))))

(define (eval-define args env)
  (add-binding! initial-env (car args) (eval (cadr args))))
```

Obsah

Reprezentace uživatelských procedur

Reprezentace uživatelských procedur

Jak víme, uživatelské procedury obsahují tři údaje:

Reprezentace uživatelských procedur

Jak víme, uživatelské procedury obsahují tři údaje:

1. seznam parametrů,

Reprezentace uživatelských procedur

Jak víme, uživatelské procedury obsahují tři údaje:

1. seznam parametrů,
2. tělo,

Reprezentace uživatelských procedur

Jak víme, uživatelské procedury obsahují tři údaje:

1. seznam parametrů,
2. tělo,
3. prostředí vzniku.

Reprezentace uživatelských procedur

Jak víme, uživatelské procedury obsahují tři údaje:

1. seznam parametrů,
2. tělo,
3. prostředí vzniku.

Reprezentace uživatelských procedur

Jak víme, uživatelské procedury obsahují tři údaje:

1. seznam parametrů,
2. tělo,
3. prostředí vzniku.

Reprezentace:

Reprezentace uživatelských procedur

Jak víme, uživatelské procedury obsahují tři údaje:

1. seznam parametrů,
2. tělo,
3. prostředí vzniku.

Reprezentace:

```
(proc params body env)
```

Konstruktor a selektory

Konstruktor a selektory

```
(define (make-proc params body env)
```

Konstruktor a selektory

```
(define (make-proc params body env)
  (list 'proc params body env))
```

Konstruktor a selektory

```
(define (make-proc params body env)
  (list 'proc params body env))

(define (proc-params proc)
```

Konstruktor a selektory

```
(define (make-proc params body env)
  (list 'proc params body env))

(define (proc-params proc)
  (cadr proc))
```

Konstruktor a selektory

```
(define (make-proc params body env)
  (list 'proc params body env))

(define (proc-params proc)
  (cadr proc))

(define (proc-body proc)
```

Konstruktor a selektory

```
(define (make-proc params body env)
  (list 'proc params body env))

(define (proc-params proc)
  (cadr proc))

(define (proc-body proc)
  (caddr proc))
```

Konstruktor a selektory

```
(define (make-proc params body env)
  (list 'proc params body env))

(define (proc-params proc)
  (cadr proc))

(define (proc-body proc)
  (caddr proc))

(define (proc-env proc)
```

Konstruktor a selektory

```
(define (make-proc params body env)
  (list 'proc params body env))

(define (proc-params proc)
  (cadr proc))

(define (proc-body proc)
  (caddr proc))

(define (proc-env proc)
  (cadddr proc))
```

Speciální operátor `lambda`

Speciální operátor `lambda`

```
(define (eval-lambda args env)
```

Speciální operátor `lambda`

```
(define (eval-lambda args env)
  (make-proc (car args) (cadr args) env))
```

Aplikace uživatelské procedury

Aplikace uživatelské procedury

Nový konstruktor prostředí:

Aplikace uživatelské procedury

Nový konstruktor prostředí:

```
(define (make-env-sv symbols values . parent-lst)
  (make-env (map cons symbols values)
            (optional-arg-val parent-lst #f)))
```

procedura `apply-user-proc`:

Aplikace uživatelské procedury

Nový konstruktor prostředí:

```
(define (make-env-sv symbols values . parent-lst)
  (make-env (map cons symbols values)
            (optional-arg-val parent-lst #f)))
```

procedura `apply-user-proc`:

```
(define (apply-user-proc proc args)


```

Aplikace uživatelské procedury

Nový konstruktor prostředí:

```
(define (make-env-sv symbols values . parent-lst)
  (make-env (map cons symbols values)
            (optional-arg-val parent-lst #f)))
```

procedura `apply-user-proc`:

```
(define (apply-user-proc proc args)
  (eval (proc-body proc)
        (make-env-sv (proc-params proc) args (proc-env proc))))
```