KATEDRA
INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI

Paradigmata programování 1

2 Uživatelské funkce a prostředí

Michal Krupka

Obsah

Definice funkce

```
(defun triangle-area (b h)
  (* 1/2 b h))
```

Definice funkce

```
(defun triangle-area (b h)
  (* 1/2 b h))
```

s dokumentací

Definice funkce

```
(defun triangle-area (b h)
  (* 1/2 b h))
```

s dokumentací

```
(defun triangle-area (b h)
  "Returns the area of a triangle with base B and height H."
  (* 1/2 b h))
```

Definice funkce

název funkce — *parametry* funkce

```
(defun triangle-area (b h)
  "Returns the area of a triangle with base B and height H."
  (* 1/2 b h))
```

tělo funkce

nepovinný *dokumentační řetězec*

Definice funkce

název funkce *parametry* funkce

```
(defun triangle-area (b h)
  "Returns the area of a triangle with base B and height H."
  (* 1/2 b h))
```

tělo funkce

nepovinný *dokumentační řetězec*

Ke každé funkci musí být známy následující informace:

- parametry,
- tělo
- a ještě něco (co to je, zjistíme později).

Uživatelsky definované funkce v našem Lispu

- vestavěné funkce nestačí
- budeme si sami definovat další

Aplikace uživatelsky definované funkce

Aplikace uživatelsky definované funkce

Vyhodnocujeme například

```
> (triangle-area (+ 5 7) (- 4 3))
```

Aplikace uživatelsky definované funkce

Vyhodnocujeme například

```
> (triangle-area (+ 5 7) (- 4 3))
```

Co se bude dít při aplikaci funkce `triangle-area` na čísla 12 a 1?

Aplikace uživatelsky definované funkce

Vyhodnocujeme například

```
> (triangle-area (+ 5 7) (- 4 3))
```

Co se bude dít při aplikaci funkce `triangle-area` na čísla 12 a 1?

Připomenutí:

```
(defun triangle-area (b h)
  "Returns the area of a triangle with base B and height H."
  (* 1/2 b h))
```

Aplikace uživatelsky definované funkce

Vyhodnocujeme například

```
> (triangle-area (+ 5 7) (- 4 3))
```

Co se bude dít při aplikaci funkce `triangle-area` na čísla 12 a 1?

Připomenutí:

```
(defun triangle-area (b h)
  "Returns the area of a triangle with base B and height H."
  (* 1/2 b h))
```

1. Zařídí se, aby symbol `b` měl hodnotu 12 a symbol `h` měl hodnotu 1.

Aplikace uživatelsky definované funkce

Vyhodnocujeme například

```
> (triangle-area (+ 5 7) (- 4 3))
```

Co se bude dít při aplikaci funkce `triangle-area` na čísla 12 a 1?

Připomenutí:

```
(defun triangle-area (b h)
  "Returns the area of a triangle with base B and height H."
  (* 1/2 b h))
```

1. Zařídí se, aby symbol `b` měl hodnotu 12 a symbol `h` měl hodnotu 1.
2. Vyhodnotí se tělo funkce.

Aplikace uživatelsky definované funkce

Vyhodnocujeme například

```
> (triangle-area (+ 5 7) (- 4 3))
```

Co se bude dít při aplikaci funkce `triangle-area` na čísla 12 a 1?

Připomenutí:

```
(defun triangle-area (b h)
  "Returns the area of a triangle with base B and height H."
  (* 1/2 b h))
```

1. Zařídí se, aby symbol `b` měl hodnotu 12 a symbol `h` měl hodnotu 1.
2. Vyhodnotí se tělo funkce.
3. Výsledkem aplikace bude výsledek tohoto vyhodnocení.

Obsah

Programová abstrakce

Nástroj (např. funkce), který je možné použít bez znalosti technických detailů jeho práce. (Víme *co* dělá, nemusíme vědět, *jak*.)

Programová abstrakce

Nástroj (např. funkce), který je možné použít bez znalosti technických detailů jeho práce. (Víme *co* dělá, nemusíme vědět, *jak*.)

Tedy když například místo

```
(> (* 1/2 11 4) (* 1/2 14 3))
```

napíšeme

```
(> (triangle-area 11 4) (triangle-area 14 3))
```

Programová abstrakce

Nástroj (např. funkce), který je možné použít bez znalosti technických detailů jeho práce. (Víme *co* dělá, nemusíme vědět, *jak*.)

Tedy když například místo

```
(> (* 1/2 11 4) (* 1/2 14 3))
```

napíšeme

```
(> (triangle-area 11 4) (triangle-area 14 3))
```

Výhody programové abstrakce

- v daný moment řešíme méně problémů, na které se tedy lépe soustředíme

Programová abstrakce

Nástroj (např. funkce), který je možné použít bez znalosti technických detailů jeho práce. (Víme *co* dělá, nemusíme vědět, *jak*.)

Tedy když například místo

```
(> (* 1/2 11 4) (* 1/2 14 3))
```

napíšeme

```
(> (triangle-area 11 4) (triangle-area 14 3))
```

Výhody programové abstrakce

- v daný moment řešíme méně problémů, na které se tedy lépe soustředíme
- zvyšuje se srozumitelnost

Programová abstrakce

Nástroj (např. funkce), který je možné použít bez znalosti technických detailů jeho práce. (Víme *co* dělá, nemusíme vědět, *jak*.)

Tedy když například místo

```
(> (* 1/2 11 4) (* 1/2 14 3))
```

napíšeme

```
(> (triangle-area 11 4) (triangle-area 14 3))
```

Výhody programové abstrakce

- v daný moment řešíme méně problémů, na které se tedy lépe soustředíme
- zvyšuje se srozumitelnost
- snadněji se dělají změny

Programová abstrakce

Nástroj (např. funkce), který je možné použít bez znalosti technických detailů jeho práce. (Víme *co* dělá, nemusíme vědět, *jak*.)

Tedy když například místo

```
(> (* 1/2 11 4) (* 1/2 14 3))
```

napíšeme

```
(> (triangle-area 11 4) (triangle-area 14 3))
```

Výhody programové abstrakce

- v daný moment řešíme méně problémů, na které se tedy lépe soustředíme
- zvyšuje se srozumitelnost
- snadněji se dělají změny
- snižuje se opakování v kódu a tím chybovost

Programová abstrakce

Nástroj (např. funkce), který je možné použít bez znalosti technických detailů jeho práce. (Víme *co* dělá, nemusíme vědět, *jak*.)

Tedy když například místo

```
(> (* 1/2 11 4) (* 1/2 14 3))
```

napíšeme

```
(> (triangle-area 11 4) (triangle-area 14 3))
```

Výhody programové abstrakce

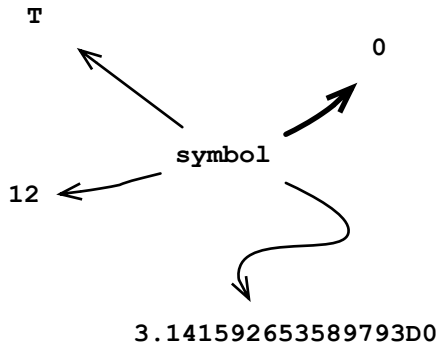
- v daný moment řešíme méně problémů, na které se tedy lépe soustředíme
- zvyšuje se srozumitelnost
- snadněji se dělají změny
- snižuje se opakování v kódu a tím chybovost
- zvyšuje se možnost znovupoužitelnosti programu

Obsah

Zajímavá otázka

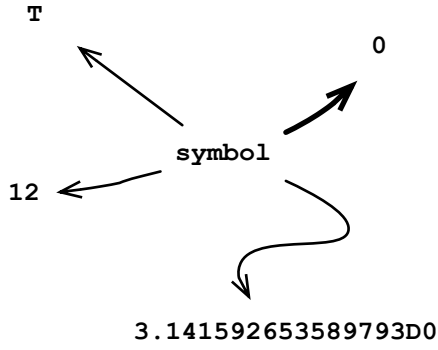
```
(defun circle-area (r)
  (* pi r r))
```

Vazby



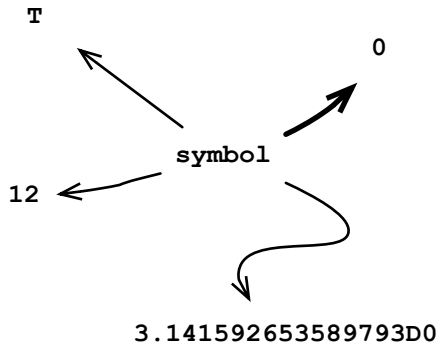
- Každý symbol může mít více *vazeb*.

Vazby



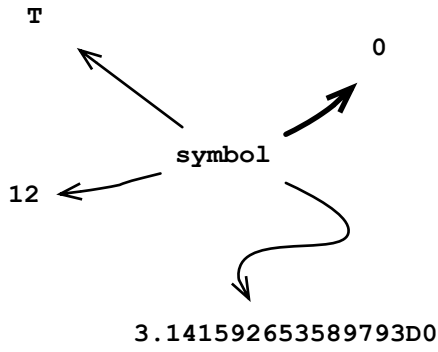
- Každý symbol může mít více *vazeb*.
- Jedna vazba může být *aktuální*. Ta *zastiňuje* ostatní (na obr. tučně).

Vazby



- Každý symbol může mít více *vazeb*.
- Jedna vazba může být *aktuální*. Ta *zastiňuje* ostatní (na obr. tučně).
- Každá vazba má *hodnotu*.

Vazby

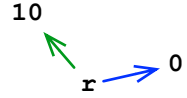


- Každý symbol může mít více *vazeb*.
- Jedna vazba může být *aktuální*. Ta *zastiňuje* ostatní (na obr. tučně).
- Každá vazba má *hodnotu*.
- Hodnotou symbolu je vždy hodnota jeho aktuální vazby.

Vazby

Vazby

Symbol `r` má dvě vazby:



V jazyce Lisp je zařízeno, že v těle funkce `circle-area` je aktuální první vazba,
v Listeneru druhá.

Vazby

Při aplikaci funkce `circle-area`:

1. se vytvoří nová vazba na symbol `r`,

Vazby

Při aplikaci funkce `circle-area`:

1. se vytvoří nová vazba na symbol `r`,
2. učiní se aktuální (tím zastíní původní vazbu),

Vazby

Při aplikaci funkce `circle-area`:

1. se vytvoří nová vazba na symbol `r`,
2. učiní se aktuální (tím zastíní původní vazbu),
3. nastaví se jí hodnota 10.

Vazby

Při aplikaci funkce `circle-area`:

1. se vytvoří nová vazba na symbol `r`,
2. učiní se aktuální (tím zastíní původní vazbu),
3. nastaví se jí hodnota 10.
4. Vyhodnotí se tělo funkce.

Vazby

Při aplikaci funkce `circle-area`:

1. se vytvoří nová vazba na symbol `r`,
2. učiní se aktuální (tím zastíní původní vazbu),
3. nastaví se jí hodnota 10.
4. Vyhodnotí se tělo funkce.
5. Po vyhodnocení vazba přestává být aktuální.

Vazby

Při aplikaci funkce `circle-area`:

1. se vytvoří nová vazba na symbol `r`,
2. učiní se aktuální (tím zastíní původní vazbu),
3. nastaví se jí hodnota 10.
4. Vyhodnotí se tělo funkce.
5. Po vyhodnocení vazba přestává být aktuální.

Vazby

Při aplikaci funkce `circle-area`:

1. se vytvoří nová vazba na symbol `r`,
2. učiní se aktuální (tím zastíní původní vazbu),
3. nastaví se jí hodnota 10.
4. Vyhodnotí se tělo funkce.
5. Po vyhodnocení vazba přestává být aktuální.

Vazba na symbol `pi` se nevytváří a nenastavuje, aktuální zůstává původní vazba.

Obsah

Prostředí

Vazby jsou organizovány v *prostředí*. To chápeme jako tabulku, ze které Lisp zjišťuje vazby symbolů a jejich hodnoty. Řádky v tabulce jsou vazby. Například:

Prostředí

Vazby jsou organizovány v *prostředí*. To chápeme jako tabulku, ze které Lisp zjišťuje vazby symbolů a jejich hodnoty. Řádky v tabulce jsou vazby. Například:

Globální prostředí

symbol	**hodnota**
`pi`	`3.141592653589793D0`
`t`	`t`
`nil`	`nil`
⋮	⋮

Prostředí

Prostředí
funkce `circle-area`

symbol	hodnota
`r`	10

To nestačí. Proto: předek prostředí.

Prostředí

Globální prostředí

symbol	hodnota
pi	3.141592653589793D0
⋮	⋮

Prostředí
funkce `circle-area`

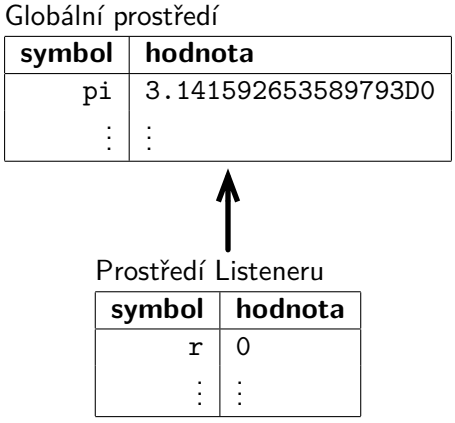
symbol	hodnota
r	10

Globální prostředí je předkem prostředí funkce `circle-area`. Každé prostředí kromě globálního má předka.

Prostředí Listeneru

Po vyhodnocení

```
> (bind r 0)
0
```



Celkově

Globální prostředí

symbol	hodnota
pi	3.141592653589793D0
⋮	⋮

Prostředí Listeneru

symbol	hodnota
r	0
⋮	⋮

Prostředí
funkce `circle-area`

symbol	hodnota
r	10

Speciální operátor `bind` (znovu)

Vyhodnocení výrazu `(bind` *a* *b*`)` v Listeneru

1. Vyhodnotí se *b*.
2. Pokud v prostředí Listeneru neexistuje vazba na symbol *a*, vytvoří se.
3. Získaná hodnota výrazu *b* se učiní hodnotou vazby symbolu *a* v prostředí Listeneru.

Znovu o vyhodnocování

Výrazy se vždy vyhodnocují v prostředí.

Znovu o vyhodnocování

Výrazy se vždy vyhodnocují v prostředí.
Aktuální prostředí je prostředí, ve kterém je výraz vyhodnocován.

Znovu o vyhodnocování

Výrazy se vždy vyhodnocují v prostředí.
Aktuální prostředí je prostředí, ve kterém je výraz vyhodnocován.

Prvky složeného výrazu se (až na stanovené výjimky) vyhodnocují ve stejném prostředí jako původní složený výraz.

Znovu o vyhodnocování

Výrazy se vždy vyhodnocují v prostředí.
Aktuální prostředí je prostředí, ve kterém je výraz vyhodnocován.

Prvky složeného výrazu se (až na stanovené výjimky) vyhodnocují ve stejném prostředí jako původní složený výraz.

Vyhodnocování symbolu

Při vyhodnocování symbolu se nejprve hledá jeho vazba v aktuálním prostředí. Pokud je nalezena, hodnotou symbolu je hodnota vazby. Pokud není nalezena, vazba se hledá v předkovi aktuálního prostředí. Tak se pokračuje tak dlouho, dokud se neskončí v globálním prostředí. Pokud ani tam není vazba nalezena, dojde k chybě.

Aplikace uživatelské funkce

Aplikace uživatelské funkce

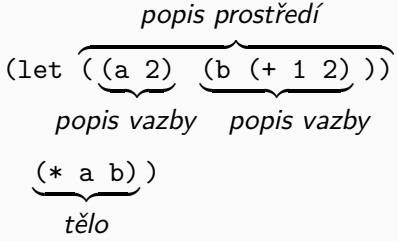
Při aplikaci uživatelské funkce se vytvoří nové prostředí s vazbami parametrů na argumenty. Předkem tohoto prostředí se stane globální prostředí (toto později zobecníme). V tomto prostředí se pak vyhodnotí tělo funkce. Nové prostředí se vytváří při každé aplikaci znovu.

Obsah

Speciální operátor `let`

```
> (let ((a 2)
        (b (+ 1 2)))
    (* a b))
6
```

`let`: terminologie

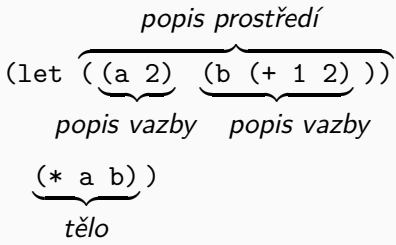


Popis prostředí Seznam libovolné délky. Jeho prvky jsou *popisy vazeb*.

Popis vazby Dvouprvkový seznam. Na prvním místě má symbol, na druhém libovolný výraz.

Tělo Libovolný výraz.

`let`: vyhodnocení

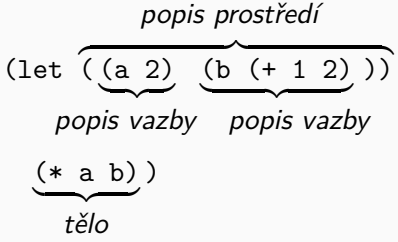


`let`: vyhodnocení

```
          popis prostředí
(let ((a 2)  (b (+ 1 2)))
   popis vazby  popis vazby
  (* a b))
    tělo
```

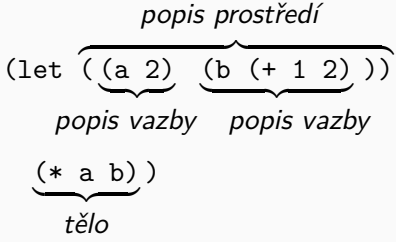
1. V aktuálním prostředí se vyhodnotí všechny druhé položky popisů vazeb.

`let`: vyhodnocení



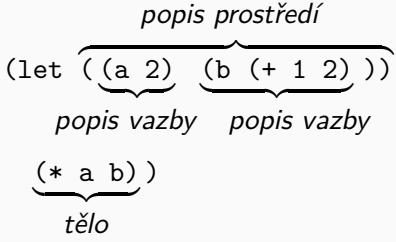
1. V aktuálním prostředí se vyhodnotí všechny druhé položky popisů vazeb.
2. Vytvoří se nové prostředí a v něm vazby tak, že každá první položka popisu vazby (která musí být symbolem) se naváže na hodnotu druhé položky.

`let`: vyhodnocení



1. V aktuálním prostředí se vyhodnotí všechny druhé položky popisů vazeb.
2. Vytvoří se nové prostředí a v něm vazby tak, že každá první položka popisu vazby (která musí být symbolem) se naváže na hodnotu druhé položky.
3. Předkem nového prostředí se učiní aktuální prostředí.

`let`: vyhodnocení



1. V aktuálním prostředí se vyhodnotí všechny druhé položky popisů vazeb.
2. Vytvoří se nové prostředí a v něm vazby tak, že každá první položka popisu vazby (která musí být symbolem) se naváže na hodnotu druhé položky.
3. Předkem nového prostředí se učiní aktuální prostředí.
4. Tělo se vyhodnotí v tomto novém prostředí. Výsledek se vrátí jako hodnota celého výrazu.

Příklady

```
(defun let-test (x)
  (let ((x 5))
    x))
```

```
> (let-test 5)
5

> (let-test 6)
5
```

```
> (let ((x 2))
    (* x (let ((x 3)) x)))
6
```