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

Úvod do programovacích stylů ◇ poznámky k přednášce

8. Funkcionální datové struktury

verze z 13. listopadu 2024

Minulou přednášku jsme zavedli omezení, že hodnoty proměnných nebudeme měnit. Nyní k němu přidáme omezení, že ani hodnoty měnit nebudeme. Nemůžeme například nastavit prvek pole:

```
>>> a = [1, 2, 3]
>>> a[0] = 4
```

Funkcionální datové struktury jsou datové struktury, kde není dovoleno měnit jejich položky. To především znamená, že nemají mutátory. Funkcionální datové struktura je tedy dána konstruktory a selektory. Můžeme mít případně i další funkce jako typový predikát.

Pokud program píšeme ve funkcionálním stylu, nemusí konstruktor začínat předponou `make` a selektor předponou `get`. Například:

```
def point(x, y):
    return [x, y]

def point_x(point1):
     return point1[0]

def point_y(point1):
     return point1[1]
```

1 Spojové seznamy

Zavedeme abstraktní datovou strukturu (**spojového**) **seznamu**. Seznam je buď hodnota `empty` nebo výsledek volání `cons(`*`val`*`, `*`lst`*`)`, kde *`val`* je libovolná hodnota a *`lst`* je seznam. Seznam `empty` se nazývá **prázdný seznam**. Seznam `cons(`*`val`*`, `*`lst`*`)` je **neprázdný**, *`val`* je **první hodnota v seznamu** a *`lst`* je **seznam bez první hodnoty** také nazývaný **zbytek seznamu**.

Definujeme si prázdný seznam:

```
empty = []
```

konstruktor neprázdného seznamu:

```
def cons(val, lst):
    return [val, lst]
```

a jeho selektory:

```
def first(lst):
    return lst[0]

def rest(lst):
    return lst[1]
```

Ekvivalentní úpravy výrazů můžeme přirozeně provádět i s funkcionálními datovými strukturami. Například pro libovolnou hodnotu `val` a seznam `lst` platí:

```
    first(cons(val, lst))
=== first([val, lst])
=== [val, lst][0]
=== val

    rest(cons(val, lst))
=== rest([val, lst])
=== [val, lst][1]
=== lst
```

Testovací seznam:

```
l1 = cons(1, cons(2, cons(3, empty)))
```

Například máme:

```
    first(l1)
=== 1

    rest(l1)
=== cons(2, cons(3, empty))
```

Definujeme si predikát rozhodující prázdnost seznamu:

```
def is_empty(lst):
    return lst == empty
```

Jistě:

```
    is_empty(empty)
=== True
```

a pro libovolný neprázdný seznam `cons(val, lst)`:

```
    is_empty(cons(val, lst))
=== False
```

Druhý prvek seznamu můžeme získat funkcí:

```
def second(lst):
    return first(rest(lst))
```

Například dostáváme:

```
    second(l1)
=== second(cons(1, cons(2, cons(3, empty))))
=== first(rest(cons(1, cons(2, cons(3, empty)))
=== first(cons(2, cons(3, empty)))
=== 2
```

Připomeňme si skládání funkcí z minulé přednášky:

```
def comp(f, g):
    return lambda x: f(g(x))
```

Druhý prvek seznamu můžeme zavést i takto:

```
second2 = comp(first, rest)
```

Snadno ověříme, že:

```
    second2(l1)
=== 2
```

2 Rekurzivní funkce na seznamech

Délku seznamu definujeme následovně. Prázdný seznam má délku nula. Délka neprázdného seznamu je následník jeho délky bez prvního prvku. Pro funkci `length` vracející délku seznamu chceme, aby platily následující rovnosti:

```
    length(empty)
=== 0

    length(cons(val, lst))
=== 1 + length(lst)
```

Funkci `length` můžeme nyní snadno definovat jako rekurzivní funkci:

```
def length(lst):
    return (0
            if is_empty(lst)
            else 1 + length(rest(lst))
```

Například dostáváme:

```
>>> length(empty)
0
>>> length(l1)
3
>>> length(rest(l1))
2
```

Hodnota je prvkem seznamu, jestliže je seznam neprázdný a platí, že hodnota je jeho prvním prvkem nebo je prvkem zbytku seznamu. Definici můžeme přímočaře přepsat do rekurzivní funkce:

```
def is_member(val, lst):
    return (not is_empty(lst)
            and (first(lst) == val
                 or is_member(val, rest(lst))))
```

3 Funkce vyšších řádů pro seznamy

Mapováním funkce jednoho parametru na vstupní seznam obdržíme seznam stejné délky, kde každý prvek je výsledek aplikace funkce na pozičně odpovídající prvek vstupního seznamu. Například mapováním funkce `square` počítající čtverec zadaného čísla na seznam `cons(1, cons(2, empty))` obdržíme seznam `cons(1, cons(4, empty))`.

Mapování bude provádět funkce `list_map` očekávající funkci jednoho parametru a seznam, přes který se má funkce mapovat. Mapování můžeme ekvivalentně definovat i rozborem případů. Mapováním funkce na prázdný seznam obdržíme prázdný seznam, což můžeme zapsat rovností:

```
    list_map(fun, empty)
=== empty
```

Mapováním funkce na neprázdný seznam `lst` obdržíme neprázdný seznam, kde první prvek je výsledek aplikace funkce na první prvek seznamu `lst` a zbytek je mapování funkce na zbytek seznamu `lst`. Tedy pokud `lst` je neprázdný seznam, pak:

```
    list_map(fun, lst)
=== cons(fun(first(lst)),
         list_map(val, rest(lst)))
```

Tato definice je vhodná k přepsání na rekurzivní funkci:

```
def list_map(fun, lst):
    return (empty
            if is_empty(lst)
            else cons(fun(first(lst)),
                      list_map(fun, rest(lst))))
```

Například:

```
    list_map(square, cons(1, cons(2, empty)))
=== cons(1, cons(4, empty))
```

Filtrováním vstupního seznamu podle predikátu jednoho parametru obdržíme seznam právě těch prvků vstupního seznamu, které splňují predikát. Pořadí prvků se filtrováním nezmění.

Například filtrováním seznamu `cons(1, cons(2, empty))` podle predikátu rozhodujícího sudost čísel, obdržíme seznam `cons(2, empty)`.

Filtrování bude provádět funkce `list_filter`, která očekává predikát jednoho parametru a seznam, který chceme filtrovat. Filtrování také můžeme definovat rozborem případů. Filtrováním prázdného seznamu obdržíme prázdný seznam:

```
    list_filter(predicate, empty)
=== empty
```

Filtrováním neprázdného seznamu `lst` začínajícího prvkem `val`, který vyhovuje predikátu, obdržíme neprázdný seznam začínající `val`, kde zbytek vznikne filtrováním zbytku seznamu `lst`. Tedy pokud `lst` je neprázdný seznam `predicate(first(lst))` je pravda, pak

```
    list_filter(predicate, lst)
=== cons(first(lst),
         list_filter(predicate, rest(lst)))
```

Nakonec filtrování neprázdného seznamu začínajícího prvkem, jenž predikát nesplňuje, je rovno filtrování jeho zbytku. Tedy pro neprázdný seznam `lst` takový, že `predicate(first(lst))` je nepravda, platí

```
    list_filter(predicate, lst)
=== list_filter(predicate, rest(lst))
```

Získané rovnice již snadno přepíšeme na rekurzivní funkci:

```
def list_filter(predicate, lst):
    return (empty
            if is_empty(lst)
            else (cons(first(lst), list_filter(predicate, rest(lst)))
                  if predicate(first(lst))
                  else list_filter(predicate, rest(lst))))
```

Pro predikát `is_even` rozhodujícím sudost například dostáváme:

```
    list_filter(is_even, cons(1, cons(2, empty)))
=== cons(2, empty)
```

Redukcí seznamu funkcí dvou parametrů s počáteční hodnotou obdržíme hodnotu, která vznikne z počáteční hodnoty tak, že od konce seznamu aplikujeme funkci na každý prvek seznamu a aktuální hodnotu.

Například redukcí seznamu `cons(2, cons(3, empty))` funkcí součtu s počáteční hodnotou nula obdržíme číslo pět.

Chceme definovat funkci `list_reduce`, která očekává funkci dvou parametrů, počáteční hodnotu a seznam, který chceme redukovat. Redukci můžeme ekvivalentně definovat pomocí rovností. Zaprvé redukce prázdného seznamu je počáteční hodnota:

```
    list_reduce(function, init, empty)
=== init
```

a redukci neprázdného seznamu provedeme tak, že aplikujeme redukující funkci na první prvek seznamu a výsledek redukce zbytku seznamu. Tedy pro neprázdný seznam `lst`:

```
    list_reduce(function, init, lst)
=== function(first(lst),
             list_reduce(function,
                         init,
                         rest(lst)))
```

Z rovností již snadno vytvoříme rekurzivní funkci:

```
def list_reduce(function, init, lst):
    return (init
            if is_empty(lst)
            else function(first(lst),
                          list_reduce(function,
                                      init,
                                      rest(lst))))
```

Pro funkci `add` součtu čísel dostáváme:

```
    reduce(add, 0, cons(2, cons(3, empty)))
=== add(2, add(3, 0))
=== 5
```

Zajímavé je, že mapování a filtrování seznamu lze definovat pomocí redukce seznamu:

```
def list_map(function, lst):
    return list_reduce((lambda value, result:
                            cons(function(value),
                                 result)),
                       empty,
                       lst)

def list_filter(predicate, lst):
    return list_reduce((lambda value, result:
                            (cons(value, result)
                             if predicate(value)
                             else result)),
                       empty,
                       lst)
```

4 Objektová reprezentace seznamů

Ve funkcionálním přístupu můžeme používat objekty, ale nesmíme je měnit. Všechny vlastnosti objektů jsou tedy jen pro čtení. Změníme reprezentaci seznamů z polí na objekty.

Nejprve si zavedeme obecnou třídu pro seznamy.

```
class List:
    pass
```

Třída pro prázdný seznam bude jejím potomkem:

```
class Empty(List):
    def is_empty(self):
        return True

    def __repr__(self):
        return "empty"
```

Systém objektu zašle zprávu `__repr__` bez argumentů v případě, že potřebuje získat textovou reprezentaci objektu, například při tisku. Návratová hodnota zprávy musí být řetězec. Třída bude mít jedinou instanci:

```
empty = Empty()
```

Zařídili jsme, že se objekt tiskne výrazem, který se na něj vyhodnotí:

```
>>> empty
empty
```

Dalším potomkem třídy `List` je třída pro neprázdné seznamy:

```
class NonEmpty(List):
    def __init__(self, first, rest):
        self.first = first
        self.rest = rest

    def get_first(self):
        return self.first

    def get_rest(self):
        return self.rest

    def is_empty(self):
        return False

    def __repr__(self):
        return ("cons("
                + repr(self.get_first())
                + ", "
                + repr(self.get_rest())
                + ")")

    def __eq__(self, val):
        return (isinstance(val, NonEmpty)
                and self.get_first() == val.get_first()
                and self.get_rest() == val.get_rest())
```

Případné argumenty při instanciaci třídy se předávají jako argumenty jejího konstruktoru. Tedy instanciace třídy `NonEmpty` vyžaduje dvě hodnoty. Například:

```
>>> ne = NonEmpty(1, empty)
>>> ne.get_first()
1
>>> ne.get_rest()
empty
```

Zprávou `__eq__` systém rozhoduje, zda se příjemce rovná argumentu zprávy. Tedy dva neprázdné seznamy se rovnají, pokud se rovnají jejich první prvky a zbytky. Operátor `==` používá k rozhodování rovosti objektů zprávu `__eq__` . Například:

```
>>> NonEmpty(1, empty) == NonEmpty(1, empty)
True
```

Definujeme konstruktor seznamů:

```
def cons(val, lst):
    return NonEmpty(val, lst)
```

Neprázdný seznam se tiskne jako výraz, jehož vyhodnocením obdržíme jemu rovný seznam:

```
>>> cons(1, empty)
cons(1, empty)
```

Zavedeme selektory pro neprázdný seznam:

```
def first(lst):
    return lst.get_first()

def rest(lst):
    return lst.get_rest()
```

a test prázdnosti:

```
def is_empty(lst):
    return lst.is_empty()
```

Zachovali jsme abstraktní datovou strukturu neprázdného seznamu i konstantu `empty` pro prázdný seznam. Můžeme tedy použít libovolnou funkci napsanou dříve. Například pro mapování:

```
def list_map(function, lst):
    return (empty
            if is_empty(lst)
            else cons(function(first(lst)),
                      list_map(function, rest(lst))))
```

dostáváme:

```
>>> list_map(lambda x: x + 1, cons(1, cons(2, empty)))
cons(2, cons(3, empty))
```

Otázky a úkoly na cvičení

1. Napište funkci na získání prvku seznamu na indexu. Například:

   ```
   >>> l1 = cons(1, cons(2, cons(3, cons(4, empty))))
   >>> list_nth(l1, 2)
   3
   ```

2. Napište funkci `list_range`, která očekává dvě čísla m a n. Funkce vrátí seznam všech čísel větších nebo rovno než m a menších než n. Čísla se v seznamu nebudou opakovat a budou uspořádána vzestupně. Například:

```
>>> list_range(1, 5)
cons(1, cons(2, cons(3, cons(4, empty))))
```

3. Napište predikát `is_every`, který očekává predikát jednoho parametru a seznam a rozhodne, zda každý prvek seznamu splňuje zadaný predikát:

```
>>> is_every(lambda x: x > 2, cons(3, cons(5, empty)))
True
>>> is_every(lambda x: x > 2, cons(1, cons(5, empty)))
False
```

4. Napište funkci, která hledá první prvek splňující predikát. Funkce vrátí singleton (pole délky jedna) s nalezeným prvkem, jinak prázdné pole. Například:

```
>>> list_find(lambda x: x >= 2, l1)
[2]
>>> list_find(lambda x: x < 1, l1)
[]
```

5. Napište funkci, která pro seznam čísel vrátí seznam jejich absolutních hodnot.

6. Napište funkci, která ze seznamu čísel vybere kladná čísla.

7. Napište funkci, která pro seznam čísel vrátí součet jejich čtverců (druhých mocnin).

8. Napište funkci, která pro seznam čísel vrátí součin všech sudých čísel v seznamu.

9. Napište funkci, která vrátí seznam prvků na sudých indexech. Například:

```
>>> list_every_second(l1)
cons(1, cons(3, empty))
```

10. Napište funkci, která vrátí poslední prvek neprázdného seznamu. Například:

```
>>> list_last(l1)
4
```

11. Napište funkci, která spojí dva zadané seznamy. Například:

```
>>> list_append(cons(1, cons(2, empty)), cons(3, cons(4, empty)))
cons(1, cons(2, cons(3, cons(4, empty))))
```

12. Napište funkci, která otočí pořadí prvků v seznamu. Například:

```
>>> list_reverse(l1)
cons(4, cons(3, cons(2, cons(1, empty))))
```

13. Lze v předchozích příkladech použít redukci seznamu?