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

9. Funkcionální uživatelské rozhraní

verze z 3. února 2025

1 Lexikální uzávěry

Prostředí udává hodnoty proměnných. Například:

`x`	`1`
`y`	`2`

je prostředí, kde hodnota proměnné `x` je číslo jedna a proměnné `y` číslo dva.

Prostředí může mít **rodiče**, kterým je opět prostředí. Jediné prostředí, které nazýváme **globální prostředí**, nemá žádného rodiče.

Vykonávání kódu probíhá vždy v nějakém prostředí, které nazýváme **aktuální**. Při spuštění programu se kód programu vykoná v globálním prostředí. Pokud neřekneme jinak, myslíme vykonáním programu jeho vykonání v globálním prostředí.

Zjišťování hodnoty proměnné ***`variable`*** v prostředí E probíhá následovně:

1. Pokud prostředí E udává hodnotu V proměnné ***`variable`***, pak je V výsledkem.
2. Pokud prostředí E neudává hodnotu proměnné ***`variable`***, ale má rodiče E_P, pak je výsledek hodnota proměnné ***`variable`*** v prostředí E_P.
3. Jinak proměnná ***`variable`*** nemá hodnotu v prostředí E.

Například pokud by prostředí G:

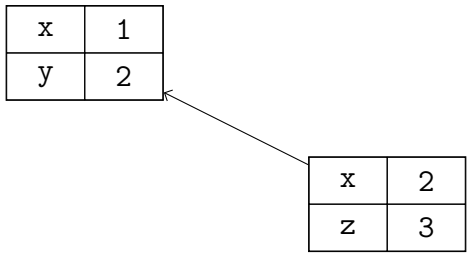
`x`	`1`
`y`	`2`

bylo globální a prostředí E:

`x`	`2`
`z`	`3`

mělo za rodiče prostředí G, pak by v prostředí E měla proměnná `x` hodnotu 2, `y` hodnotu 2, `z` hodnotu 3 a proměnná `a` by hodnotu v prostředí E neměla.

Šipkou můžeme vyznačit rodiče prostředí:



Vyhodnocování výrazů v prostředí probíhá podle obvyklých pravidel. Například vyhodnocením výrazu:

```
x + y
```

povede v prostředí:

x	1
y	2

k hodnotě 3.

Vykonání příkazu:

```
variable = value
```

v prostředí E proběhne tak, že se v prostředí E vytvoří proměnná ***variable*** s hodnotou ***value***. Připomeňme si, že neumožňujeme změnu hodnoty proměnných.

Například po vykonání:

```
y = x + 1
```

v prostředí:

x	2

se prostředí změní na:

x	2
y	3

Každá funkce je určena:

1. parametry,
2. tělem
3. a **prostředím vzniku**.

Příkaz definující funkci:

```
def function(param1, ..., paramn):
    body
```

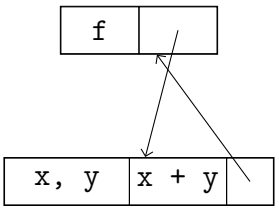
vytvoří funkci s parametry *param*1, ..., *param*n a tělem *body*, kde prostředím vzniku bude aktuální prostředí, a v aktuálním prostředí vytvoří proměnnou *function*, kde hodnotou bude právě vytvořená funkce.

Například program:

```
def f(x, y):
    return x + y
```

přidá do globálního prostředí proměnnou `f`, která za hodnotu bude mít funkci s parametry `x` a `y`, tělem `return x + y` a prostředím vzniku rovným globálnímu prostředí.

Funkci můžeme znázornit řádkem s třemi buňkami, kde první buňka udává její parametry, druhá tělo a třetí šipkou odkazuje na prostředí vzniku. Pokud je tělo funkce tvořeno pouze příkazem návratu, vynecháváme klíčové slovo `return`. Příklad zobrazení funkce `f`:



Volání funkce probíhá tak, že

1. se nejprve vytvoří prostředí E, kde parametry funkce budou mít postupně hodnoty argumentů. Rodičem prostředí E bude prostředí vzniku funkce.
2. Poté se vykoná tělo funkce v prostředí E.

Například:

```
f(1, 2)
```

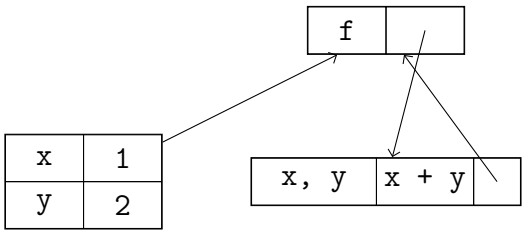
způsobí vytvoření prostředí E:

`x`	`1`
`y`	`2`

Rodičem prostředí E je globální prostředí (prostředí vzniku funkce `f`). Poté se v prostředí E vykoná tělo funkce:

```
return x + y
```

To povede na návratovou hodnotu 3. Obrázek s přidaným prostředím E:



Definujme si funkci:

```
def g(y):
    def h(x):
        return x + y
    return h
```

Zavoláním:

```
h1 = g(2)
```

vznikne prostředí E_1:

y	2

v kterém se vykoná:

```
def h(x):
   return x + y
return h
```

Funkce `h` bude mít prostředí vzniku E_1. Zavoláním:

```
h1(3)
```

vznikne prostředí E_2:

x	3

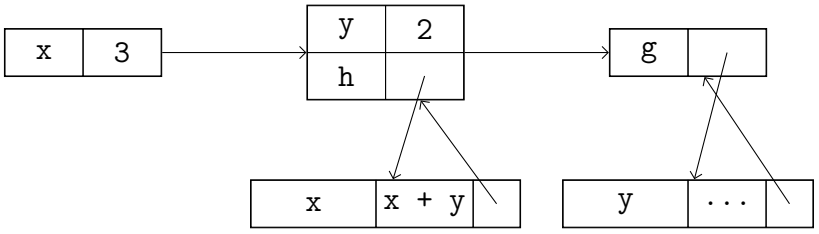
Rodičem E_2 bude prostředí vzniku funkce `h1` tedy prostředí E_1.

V prostředí E_2 se vykoná:

```
return x + y
```

Hodnota proměnné `x` se nalezne v prostředí E_2 a hodnota proměnné `y` v prostředí E_1 (rodič E_2). Výsledkem volání tedy bude 3.

Zobrazení všech funkcí a prostředí vzniklých při vykonávání programu:



Vyhodnocením lambda výrazu:

```
lambda param1, ..., paramn: body
```

v prostředí E vznikne funkce s parametry *param*1, ..., *param*n, tělem `return` *body* a prostředím vzniku E.

2 Čisté funkce

Zavolání funkce může mít vedlejší efekt. **Vedlejším efektem** se myslí změna, kterou volání funkce způsobilo. Typicky se jedná o změnu proměnné, mutaci datové struktury, zápis do souboru nebo tisk na výstup.

Například funkce:

```
def succ_print(x):
    print(x)
    return x + 1
```

jako vedlejší efekt tiskne svůj argument.

Čistá funkce je funkce, která nemá vedlejší efekt a kde návratová hodnota závisí pouze na argumentech.

Například funkce

```
def succ(x):
    return x + 1
```

je čistá, ale funkce

```
y = [0]

def succ(x):
    y[0] = 1
    return x + 1
```

a ani funkce

```
y = 1

def succ(x):
    return x + y
```

čisté nejsou. Druhá funkce by byla čistá za předpokladu, že proměnnou `y` nebudeme měnit.

Funkce je naprogramována ve **funkcionálním stylu**, pokud je čistá a nedopouští se změny proměnných ani hodnot.

Například funkce:

```
def succ(x):
    return x + 1
```

je naprogramována ve funkcionálním stylu, ale funkce

```
def succ(x):
    x = x + 1
    return x
```

není naprogramována ve funkcionálním stylu, přestože je čistá.

3 Funkcionální uživatelské rozhraní

Pro vytváření uživatelského rozhraní ve funkcionálním stylu budeme potřebovat knihovnu Functional Micro Widget (`fmw`). Manuál ke knihovně se nalézá v souboru `fmw.pdf`.

Prvky uživatelského rozhraní definujeme pomocí příslušných funkcí z knihovny. Například popisek s textem `"Ahoj"`:

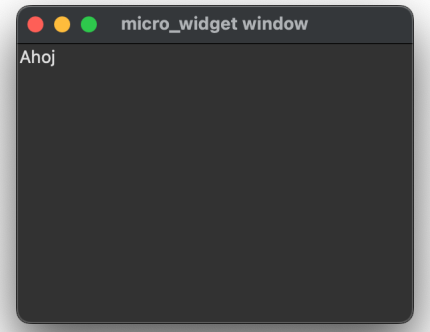
```
>>> label("Ahoj")
label('Ahoj', 0, 0)
```

Funkce `display_window` zavolaná na prvek uživatelského rozhraní zobrazí okno se zadaným prvkem.

Například příkaz:

```
>>> display_window(label("Ahoj"))
```

zobrazí okno:



Souřadnice prvku můžeme zadat přímo:

```
>>> label("Ahoj", 10, 50)
label('Ahoj', 10, 50)
```

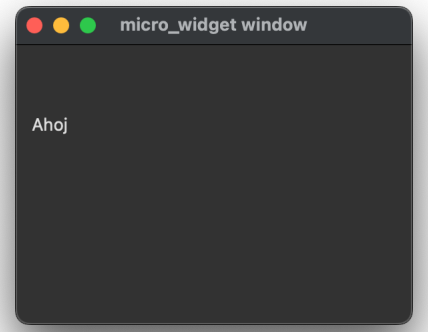
nebo posunutím:

```
>>> moved(label("Ahoj"), 10, 50)
label('Ahoj', 10, 50)
```

V obou případech dostaneme stejný prvek, který můžeme dát do okna:

```
>>> display_window(moved(label("Ahoj"), 10, 50))
```

uvidíme:



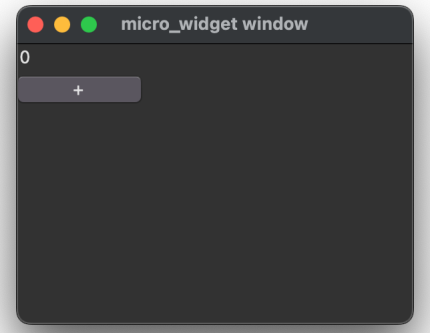
Složený prvek zadáme funkcí `group` očekávající dva prvky, které chceme složit:

```
>>> group(label("0"), moved(button("+"), 0, 20))
group(label('0', 0, 0), button('+', None, 0, 20))
```

Příkaz:

```
>>> display_window(group(label("0"), moved(button("+"), 0, 20)))
```

zobrazí okno:



Obsah okna často závisí na datech, která chceme v okně zobrazit. V takovém případě je výhodné zadat obsah okna funkcí. Například zobrazení počítadla závisí na jeho hodnotě:

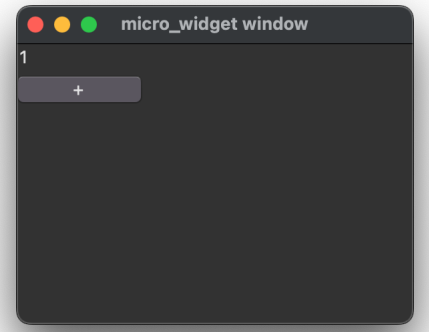
```
def counter(value):
    return group(label(str(value)),
                 moved(button("+"), 0, 20))
```

Obsah předchozího okna můžeme zadat i takto:

```
>>> counter(0)
group(label('0', 0, 0), button('+', None, 0, 20))
```

Zadáme jinou hodnotu počítadla:

```
>>> display_window(counter(1))
```



Funkci `display_window` můžeme zavolat i následovně.

```
display_window(content, initial_state)
```

Argument *content* musí být funkce jednoho parametru, která vrací ovládací prvek, a argument *initial_state* je libovolná hodnota. Nejprve se zavolá funkce *content* na *initial_state*. Tím se obdrží ovládací prvek, který bude obsahem zobrazeného okna.

Naposledy zobrazené okno získáme i příkazem:

```
>>> display_window(counter, 1)
```

Zobrazené okno má **stav**, kterým je na začátku hodnota *initial_state*. Druhý argument tlačítka udává stav, do kterého okno přejde při jeho stisku:

```
button(text, next_state)
```

Upravíme definici funkce `counter` tak, aby další stav po stisku tlačítka byl o jedna větší než aktuální stav:

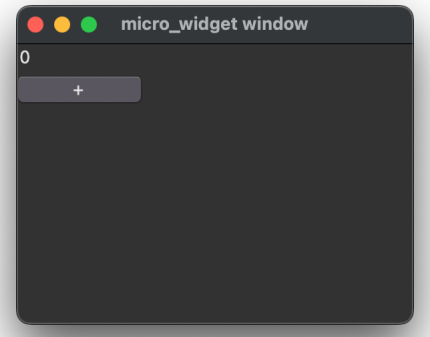
```
def counter(value):
    return group(label(str(value)),
                 moved(button("+", value + 1), 0, 20))
```

Pokud by například byl aktuální stav nula, pak další stav je číslo jedna:

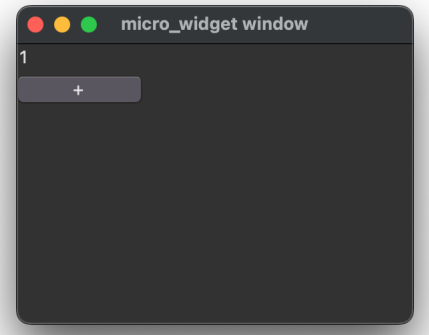
```
>>> counter(0)
group(label('0', 0, 0), button('+', 1, 0, 20))
```

Zobrazíme počítadlo:

```
>>> display_window(counter, 0)
```



Po kliku na tlačítko se aktuální stav okna změní na číslo jedna a v důsledku toho se hodnota počítadla zvětší:



Klikáním na tlačítko zvyšujeme hodnotu počítadla.

Další varianta volání funkce `display_window` je následující.

```
display_window(content, initial_state, next_state)
```

Argument ***next_state*** musí být funkce dvou parametrů. Funkce se volá v případě, že dojde k **vyvolání akce**. V takové situaci se funkce ***next_state*** zavolá na aktuální stav a vyvolanou akci a vrátí následující stav okna.

Druhý argument tlačítka se také nazývá **akce**. Při stisku tlačítka se vyvolá jeho akce.

Upravíme funkci `counter`, tak aby tlačítko mělo akci číslo jedna:

```
def counter(value):
    return group(label(str(value)),
                 moved(button("+", 1), 0, 20))
```

Ověříme:

```
>>> counter(5)
group(label('5', 0, 0), button('+', 1, 0, 20))
```

Přidáme funkci na zpracování vyvolaných akcí:

```
def add(value, increment):
    return value + increment
```

Pokud by aktuální stav bylo číslo pět a akce číslo jedna, pak funkce určí následující stav:

```
>>> add(5, 1)
6
```

Počítadlo zobrazíme příkazem:

```
display_window(counter, 0, add)
```

Klikáním na tlačítko se zvětšuje stav aplikace, kterým je hodnota počítadla.

Druhý argument textového pole je akce:

```
entry(text, action)
```

Při změně textu v poli se vyvolá akce:

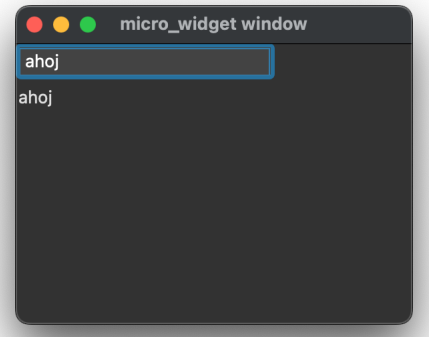
```
[action, text]
```

kde *text* je změněný text.

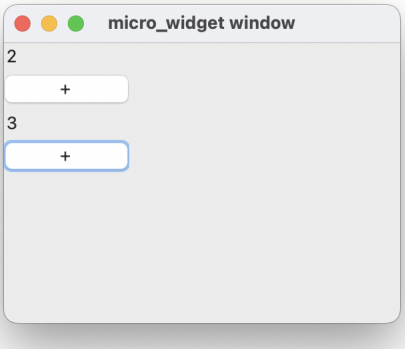
Například tento jednoduchý příklad zopakuje pod textovým polem v něm zadaný text:

```
def content(string):
    return group(entry(string, True),
                 moved(label(string), 0, 30))


def next_string(string, action):
    return action[1]

display_window(content, "", next_string)
```

Po napsání `ahoj` do textového pole obdržíme:



Představme si, že chceme v okně zobrazit dvě nezávislá počítadla:



a rádi bychom použili výše definované počítadlo. Stav aplikace bude dvojice `[`*`value1`*`, `*`value2`*`]` udávající hodnoty počítadel. Výchozím stavem bude dvojice `[0, 0]`. Jak ale při kliku na tlačítko poznat, které počítadlo máme navýšit? K rozlišení akcí se používá funkce:

```
action_changed(widget, id)
```

která změní každou akci *`action`* ve *`widget`* na `[`*`action`*`, `*`id`*`]`. Nyní již umíme rozlišit akce obou počítadel:

```
>>> action_changed(counter(0), 0)
group(label('0', 0, 0), button('+', [0, 1], 0, 20))
>>> action_changed(counter(0), 1)
group(label('0', 0, 0), button('+', [1, 1], 0, 20))
```

Obsah okna je dán funkcí:

```
def counters(values):
    counter1 = action_changed(counter(values[0]), 0)
    counter2 = action_changed(counter(values[1]), 1)
    return group(counter1, moved(counter2, 0, 50))
```

Pro změnu hodnoty jednoho počítadla si napíšeme pomocnou funkci:

```
def values_increment(values, index, increment):
    return values[:index] + [add(values[index], increment)] + values[index + 1:]
```

Například máme:

```
>>> values_increment([0, 0], 1, 1)
[0, 1]
```

Protože akce je nyní dvojice `[index, increment]`, můžeme zpracování akce zařídit funkcí:

```
def counters_increment(values, action):
    index = action[0]
    increment = action[1]
    return values_increment(values, index, increment)
```

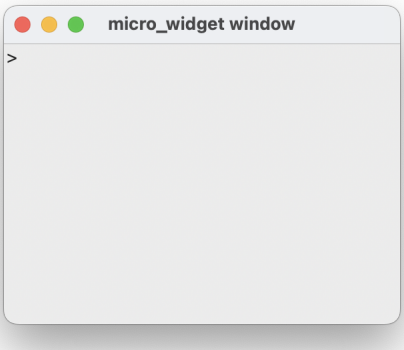
Zbývá otevřít okno:

```
display_window(counters, [0, 0], counters_increment)
```

Ke komunikaci vnějšího prostředí s aplikací slouží funkce jednoho parametru *emit_action*, kterou vrací funkce `display_window`. Její aplikování na libovolnou hodnotu *action* způsobí, že se v okně vyvolá akce *action*. Uvažujme například následující program:

```
def next_state(state, action):
    return state + " " + action

emit_action = display_window(label, ">", next_state)
```

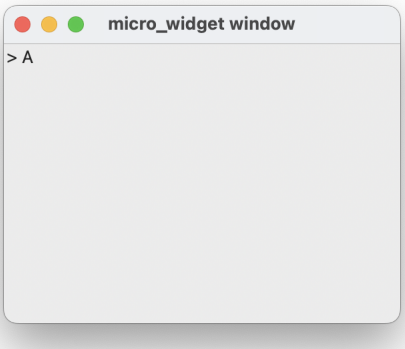
Spuštěním se otevře okno:



Každá aplikace funkce `emit_action` na řetězec způsobí jeho přidání do okna. Například po zavolání:

```
emit_action("A")
```

se písmeno A zobrazí v okně:



Celé uživatelské rozhraní je definované ve funkcionálním stylu. Pro komunikaci aplikace s vnějším prostředím je potřeba napsat funkci jednoho parametru *process_effect*, která většinou bude mít vedlejší efekt. Funkci pak můžeme zadat při otevření okna:

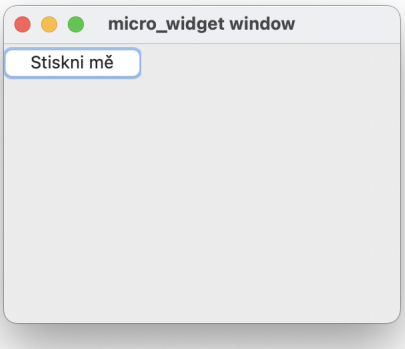
```
display_window(content,
               initial_state,
               next_state,
               process_effect)
```

Funkce `process_effect` očekává jediný argument nazývaný **efekt**. Funkce se volá v případě, že funkce *next_state* vrátí hodnotu `with_effect(state, effect)`. Pak je *state* další stav aplikace a funkce `process_effect` se aplikuje na *effect*.

Například program:

```
def next_state(state, action):
    state2 = state + action
    return with_effect(state2, f"Stisknuto {state2}-krát.")

display_window(button("Stiskni mě", 1), 0, next_state, print)
```

zobrazí okno:



Každý stisk tlačítka vytiskne kolikrát bylo tlačítko stisknuto. Například sedmeronásobné stisknutí vytiskne:

```
Stisknuto 1-krát.
Stisknuto 2-krát.
Stisknuto 3-krát.
Stisknuto 4-krát.
Stisknuto 5-krát.
Stisknuto 6-krát.
Stisknuto 7-krát.
```

Otázky a úkoly na cvičení

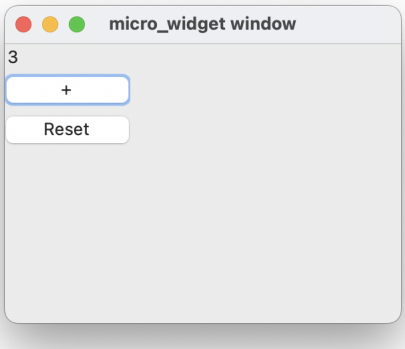
1. Uvažujme program:

   ```
   def succ(x):
       return x + 1

   def square(x):
       return x ** 2

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

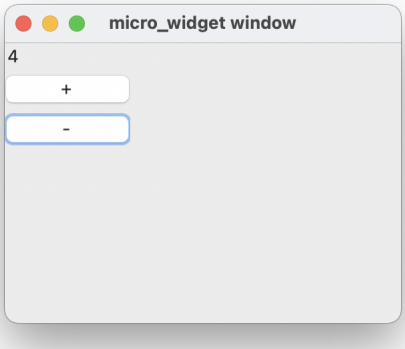
   Zakreslete všechna prostředí a funkce, které vzniknou vykonáním programu a vyhodnocením:

   ```
   >>> comp(succ, square)(3)
   10
   ```

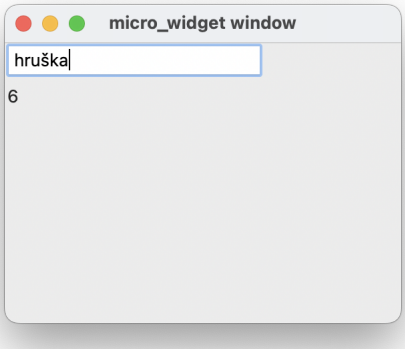
2. Vytvořte počítadlo s tlačítkem pro reset (nastaví hodnotu počítadla na nulu). Napište dvě řešení. V prvním použijte přímou změnu stavu a v druhém akce. Ukázka:



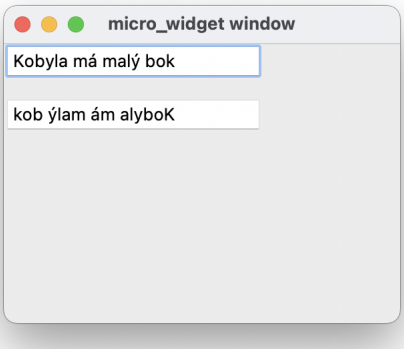
3. Vytvořte počítadlo s tlačítky pro inkrementaci i dekrementaci a zařiďte, aby hodnota počítadla byla vždy nezáporné číslo. Ukázka:



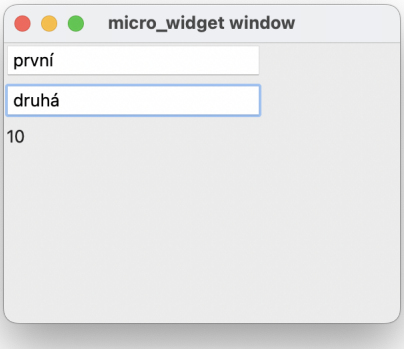
4. Vytvořte textové pole, kde vedle bude zobrazena délka řetězce, které do něj uživatel zadal. Ukázka:



5. Vytvořte dvě textová pole, kde obsah jednoho je vždy převrácený obsah druhého. Ukázka:



6. Vytvořte dvě textová pole a popisek, který bude udávat součet délek obsahu obou polí. Ukázka:



7. Vytvořte dvě textová pole a tlačítko, které prohodí jejich obsah. Ukázka: