KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Jan Laštovička
jan.lastovicka@upol.cz
www.inf.upol.cz/lide/jan-lastovicka
17. listopadu 12, 771 46 Olomouc

Základy programování pro IT 1

3. Řízení vykonávání programu

verze z 8. října 2024

1 Podmíněné vykonání

Příkazy, které se skládají z dalších příkazů, se nazývají **složené**. Zbylé příkazy jsou **jednoduché**. Příkaz nastavení hodnoty proměnné a příkaz tisku jsou jednoduché příkazy.

Zavedeme si první složený příkaz. Vezměme podmínku *condition* a dva programy *program1* a *program2*, pak

```
if condition:
    program1
else:
    program2
```

je **příkaz podmíněného vykonání** nebo také **příkaz větvení**. Například:

```
if x >= 0:
    y = x
else:
    y = -x
```

Příkaz podmíněného vykonání se vykoná tak, že se nejprve vyhodnotí podmínka *condition*. Pokud je podmínka splněna (její hodnota je `True`), pak se vykoná program *program1*, jinak se vykoná program *program2*. Například vykonání příkazu:

```
if x >= 0:
    y = x
else:
    y = -x
```

v prostředí:

```
x | 5
```

vyhodnotí podmínku `x >= 0`. Protože podmínka je v prostředí splněna, vykoná se příkaz `y = x` a dostaneme se do prostředí:

```
x | 5
y | 5
```

Pokud ale příkaz vykonáme v prostředí:

```
x | -5
```

kde není splněna podmínka, vykoná se příkaz `y = -x` a to povede na prostředí:

```
x | -5
y | 5
```

2 Pravidlo větvení

Pro příkaz větvení si zavedeme nové pravidlo programové logiky. Vezměme podmínky *precondition* a *condition* a programy *program1* a *program2*, pak platí následující. Pokud

```
{precondition and condition}
program1
{postcondition}
```

a

```
{precondition and not condition}
program2
{postcondition}
```

pak

```
{precondition}
if condition:
    program1
else:
    program2
{postcondition}
```

Například z

```
{x >= 0}
y = x
{y >= 0}
```

a

```
{not (x >= 0)}
y = -x
{y >= 0}
```

plyne

```
{True}
if x >= 0:
    y = x
else:
    y = -x
{y >= 0}
```

3 Podmíněné opakování

Pro podmínku *condition* a program *program* se složený příkaz

```
while condition:
    program
```

nazývá **příkaz podmíněného opakování** nebo také **příkaz cyklu**. Podmínka *condition* se nazývá **podmínka cyklu** a program *program* **tělo cyklu**. Příklad podmíněného opakování:

```
while x > 0:
    x = x - 1
```

Příkaz se vykoná následovně:

1. Vyhodnotí se podmínka *condition*
2. Pokud je podmínka splněna, vykoná se *program* a pokračuje se prvním bodem.
3. Jinak vykonávání příkazu končí.

Například vykonání příkazu:

```
while x > 0:
    x = x - 1
```

v prostředí:

$$\texttt{x} \mid 2$$

nejprve vyhodnotí podmínku `x > 0`. Ta je splněna a proto se vykoná příkaz `x = x - 1`. Což povede na prostředí:

$$\texttt{x} \mid 1$$

Opět se vyhodnotí podmínka cyklu. Protože je opět splněna, vykoná se tělo cyklu a obdržíme:

$$\texttt{x} \mid 0$$

Nyní už podmínka cyklu není splněna a tak vykonávání cyklu končí.

Vykonávání příkazu nemusí nikdy skončit. Například příkaz

```
while x != 0:
    x = x - 1
```

spuštěný v prostředí

$$\texttt{x} \mid -1$$

povede k nekonečnému vykonávání.

4 Změna pravdivosti trojice

Protože máme programy, jejichž vykonávání nemusí nikdy skončit, musíme změnit definici pravdivosti programové trojice.

Programová trojice

$$\{\mathit{precondition}\}$$
$$\mathit{program}$$
$$\{\mathit{postcondition}\}$$

je **pravdivá**, jestliže pro každé prostředí *env* vhodného typu platí, že

1. pokud *env* splňuje prekondici *precondition*
2. a vykonání programu *program* v prostředí *env* skončí,
3. pak se dostaneme do prostředí splňující postkondici *postcondition*.

Například programová trojice

```
{True}
while True:
    x = x + 1
{x == 0}
```

je pravdivá, protože spuštění jejího programu v libovolném prostředí nikdy neskončí.

5 Pravidlo cyklu

Vezměme podmínky *invariant* a *condition* a program *program*, pak platí následující. Pokud

```
{invariant and condition}
program
{invariant}
```

pak

```
{invariant}
while condition:
    program
{invariant and not condition}
```

Podmínka *invariant* se nazývá **invariant cyklu**.

Například z pravidla přiřazení víme, že

```
{x > 0}
x = x - 1
{x >= 0}
```

což můžeme přepsat na

```
{x >= 0 and x > 0}
x = x - 1
{x >= 0}
```

Použijeme pravidlo cyklu a dostaneme:

```
{x >= 0}
while x > 0:
    x = x + 1
{x >= 0 and not x > 0}
```

Zjednodušíme postkondici:

```
{x >= 0}
while x > 0:
    x = x - 1
{x == 0}
```

Podmínka `x >= 0` je invariantem cyklu.

6 Prázdný příkaz

Příkaz `pass` nazýváme **prázdný příkaz**. Ten se vykoná tak, že nic neudělá. Vykonání prázdného příkazu v libovolném prostředí jej nezmění. Máme pro něj jednoduché **pravidlo nic nedělání**. Pro každou podmínku *condition* platí

```
{codition}
pass
{condition}
```

Například:

```
{x = 1}
pass
{x = 1}
```

7 Zkratky

Vezměme podmínku *condition* a program *program*, pak je příkaz:

```
if condition:
    program
```

zkratkou za:

```
if condition:
    program
else:
    pass
```

Například:

```
if x < 0:
    x = -x
```

je zkratkou za:

```
if x < 0:
    x = -x
else:
    pass
```

Vezměme podmínky *condition1*, ..., *conditionn* a programy *program1*, ..., *programn* a *program*, pak je příkaz:

```
if condition1:
    program1
elif condition2:
    program2
elif condition3:
    program3
⋮
elif conditionn:
    programn
else:
    program
```

zkratkou za:

```
if condition1:
    program1
else:
    if condition2:
        program2
    elif condition3:
        program3
    ⋮
    elif conditionn:
        programn
    else:
        program
```

Například:

```
if x < 0:
    y = -1
elif x > 0:
    y = 1
else:
    y = 0
```

je zkratkou za:

```
if x < 0:
    y = -1
else:
    if x > 0:
        y = 1
    else:
        y = 0
```

Vezměme proměnnou *var*, čísla *m* a *n* a program *program*, který nemění proměnnou *var*, pak je příkaz:

```
for var in range(m, n):
    program
```

zkratkou za:

```
var = m
while var < n:
    program
    var = var + 1
```

Například:

```
for i in range(2, 5):
    x = x + i
```

je zkratkou za:

```
i = 2
while i < 5:
    x = x + i
    i = i + 1
```

Příkaz:

```
for var in range(n):
    program
```

je zkratkou za:

```
for var in range(0, n):
    program
```

Například:

```
for i in range(10):
    x = x + i
```

je zkratkou za:

```
for i in range(0, 10):
    x = x + i
```

8 Přehled pravidel programové logiky

Programová logika používá následujících sedm pravidel.

1. pravidlo přiřazení
2. pravidlo zesílení prekondice
3. pravidlo oslabení postkondice
4. pravidlo skládání
5. pravidlo větvení
6. pravidlo cyklu
7. pravidlo nic nedělání

9 Příklady důkazů

9.1 První příklad

Chceme dokázat:

```
{x > 0}
if y > 0:
    z = x + y
else:
    z = x - y
{z > 0}
```

Kroky:

1. Program trojice je větvení. Bude tedy nutné použít pravidlo větvení. Abychom mohli pravidlo použít, musíme dokázat:

   ```
   {x > 0 and y > 0}
   z = x + y
   {z > 0}
   ```

2. a také:

   ```
   {x > 0 and not y > 0}
   z = x - y
   {z > 0}
   ```

3. Začneme důkazem první trojice. Program trojice je přiřazení a musíme tedy použít jeho pravidlo:

```
{x + y > 0}
z = x + y
{z > 0}
```

4. Podmínka `(x > 0 and y > 0)` je silnější než `(x + y > 0)`. Stačí tedy použít zesílení prekondice a dostaneme trojici z prvního kroku:

```
{x > 0 and y > 0}
z = x + y
{z > 0}
```

5. Nyní obrátíme pozornost k trojici z druhého kroku. Opět použijeme pravidlo přiřazení:

```
{x - y > 0}
z = x - y
{z > 0}
```

6. Podmínka `(x > 0 and y <= 0)` je silnější než `(x - y > 0)`. Zesílíme prekondici::

```
{x > 0 and y <= 0}
z = x - y
{z > 0}
```

7. Přeformulováním prekondice obdržíme trojici z drohého kroku:

```
{x > 0 and not y > 0}
z = x - y
{z > 0}
```

8. Nyní již stačí použít pravidlo větvení na trojici z čtvrtého a sedmého kroku a obdržíme trojici, jenž jsme chtěli dokázat:

```
{x > 0}
if y > 0:
    z = x + y
else:
    z = x - y
{z > 0}
```

9.2 Druhý příklad

Chceme dokázat:

```
{x <= 5}
while x < 5:
    x = x + 1
{x == 5}
```

Kroky:

1. Protože program trojice, kterou chceme dokázat, je cyklus, budeme muset použít pravidlo cyklu. Aby jej však bylo možné použít, musíme nejprve dokázat:

   ```
   {x <= 5 and x < 5}
   x = x + 1
   {x <= 5}
   ```

2. Z pravidla přiřazení dostáváme:

   ```
   {x + 1 <= 5}
   x = x + 1
   {x <= 5}
   ```

3. Prekondici můžeme zjednodušit:

   ```
   {x < 5}
   x = x + 1
   {x <= 5}
   ```

4. Přeformulováním prekondice obdržíme požadovanou trojici z prvního kroku.

   ```
   {x <= 5 and x < 5}
   x = x + 1
   {x <= 5}
   ```

5. Nyní již můžeme použít pravidlo cyklu:

   ```
   {x <= 5}
   while x < 5:
       x = x + 1
   {x <= 5 and not x < 5}
   ```

6. Zjednodušením postkondice obdržíme trojici, jenž jsme chtěli dokázat:

   ```
   {x <= 5}
   while x < 5:
       x = x + 1
   {x == 5}
   ```

9.3 Třetí příklad

Chceme dokázat:

```
{x >= 0}
y = 0
z = 0
while z < x:
    y = y + 2
    z = z + 1
{y == 2 * x}
```

Kroky:

1. Dokazujeme odspodu. Poslední příkaz programu je cyklus. Vidíme, že podmínka `y == 2 * z` platí před začátkem cyklu a i po každém vykonání těla cyklu. Musí tedy platit i po skončení cyklu. Musíme umět dokázat, že `z == x`. Víme, že po skončení cyklu, bude platná negace jeho podmínky: `neg z < x`, což je `z >= x`. Proto by invariantem cyklu měla být podmínka `y == 2 * z and z <= x`. Vychází nám, že musíme dokázat následující trojici, abychom na ni následně mohli použít pravidlo cyklu:

   ```
   {y == 2 * z and z <= x and z < x}
   y = y + 2
   z = z + 1
   {y == 2 * z and z <= x}
   ```

2. Dokazujeme odspodu. Použijeme pravidlo přiřazení:

   ```
   {y == 2 * (z + 1) and (z + 1) <= x}
   z = z + 1
   {y == 2 * z and z <= x}
   ```

3. Zjednodušíme prekondici:

   ```
   {y == 2 * z + 2 and z < x}
   z = z + 1
   {y == 2 * z and z <= x}
   ```

4. Použijeme pravidlo přiřazení na první příkaz v trojici z prvního kroku. Prekondice předchozí trojice musí být postkondicí následující trojice.

   ```
   {y + 2 == 2 * z + 2 and z < x}
   y = y + 2
   {y == 2 * z + 2 and z < x}
   ```

5. Zjednodušíme prekondici:

```
{y == 2 * z and z < x}
y = y + 2
{y == 2 * z + 2 and z < x}
```

6. Použijeme pravidlo skládání na trojice z pátého a třetího kroku:

```
{y == 2 * z and z < x}
y = y + 2
z = z + 1
{y == 2 * z and z <= x}
```

7. Přeformulováním prekondice obdržíme trojici, kterou jsme chtěli dokázat v prvním kroku:

```
{y == 2 * z and z <= x and z < x}
y = y + 2
z = z + 1
{y == 2 * z and z <= x}
```

8. Můžeme použít pravidlo cyklu a dostat:

```
{y == 2 * z and z <= x}
while z < x:
    y = y + 2
    z = z + 1
{y == 2 * z and z <= x and not z < x}
```

9. Zjednodušíme postkondici:

```
{y == 2 * z and z <= x}
while z < x:
    y = y + 2
    z = z + 1
{y == 2 * x and z == x}
```

10. Podmínka `(y == 2 * x)` je slabší než `y == 2 * x and z == x` a proto můžeme postkondici oslabit a získat:

```
{y == 2 * z and z <= x}
while z < x:
    y = y + 2
    z = z + 1
{y == 2 * x}
```

11. Vezmeme si prostřední příkaz z původní trojice. Víme, že jeho postkondice musí být `y == 2 * z and z <= x`. Použijeme pravidlo přiřazení:

```
{y == 2 * 0 and 0 <= x}
z = 0
{y == 2 * z and z <= x}
```

12. Prekondici zjednodušíme:

```
{y == 0 and 0 <= x}
z = 0
{y == 2 * z and z <= x}
```

13. Použijeme pravidlo skládání na trojice z kroku 12 a 10:

```
{y == 0 and 0 <= x}
z = 0
while z < x:
    y = y + 2
    z = z + 1
{y == 2 * x}
```

14. Zbývá nám první příkaz v původní trojici, jehož postkondice musí být `y == 0 and 0 <= x`. Pravidlem přiřazení získáme:

```
{0 == 0 and 0 <= x}
y = 0
{y == 0 and 0 <= x}
```

15. Prekondici lze zjednodušit:

```
{0 <= x}
y = 0
{y == 0 and 0 <= x}
```

16. Složením trojic z kroků 15 a 13 získáme trojici, jenž jsme chtěli původně dokázat:

```
{x >= 0}
y = 0
z = 0
while z < x:
    y = y + 2
    z = z + 1
{y == 2 * x}
```

Otázky a úkoly k semináři

1. U následujících trojic rozhodněte, zda jsou pravdivé. V kladném případě trojici dokažte, v záporném napište, v kterém prostředí je pravdivost porušena.

   (a)

   ```
   {True}
   if x > y:
       x = x + 1
   else:
       x = y
   {x >= y}
   ```

   (b)

   ```
   {True}
   if x == y:
       y = y + 1
   else:
       x = x - y
   {x < y}
   ```

   (c)

   ```
   {True}
   if (x % 2) == 0:
       y = x
   else:
       y = x + 1
   {(y % 2) == 0}
   ```

   (d)

   ```
   {x > 0}
   if y > 0:
       z = x + y
   else:
       z = x
   {z > 0}
   ```

   (e)

   ```
   {True}
   while x % 10 != 0:
       x = x + 1
   {x % 10 == 0}
   ```

   (f)

   ```
   {(x >= 0) and (z == 0)}
   while x == 0:
       z = z + 1
   {z == 0}
   ```

(g)

```
{(x >= 0) and (y >= 0) and (x == x0) and (y == y0)}
while x > 0:
    y = y + 1
    x = x - 1
{y == x0 + y0}
```

2. Napište program *program* tak, aby platilo:

```
{True}
program
{(z >= x) and (z >= y)}
```

a

```
{True}
program
{(z == x) or (z == y)}
```

V programu nesmíte měnit proměnné `x` a `y`. Program otestujte a dokažte jeho správnost.

3. Napište program *program* tak, aby platilo:

```
{(x >= 0) and (y >= 0) and (x == x0) and (y == y0)}
program
{y = x0 * y0}
```

V programu nesmíte měnit proměnné `x0` a `y0` a z aritmetických operátorů smíte použít pouze `+`. Program otestujte a dokažte jeho správnost.