

# **Kapitola XII.**

# **Optimalizace a**

# **generování cílového**

# **programu**

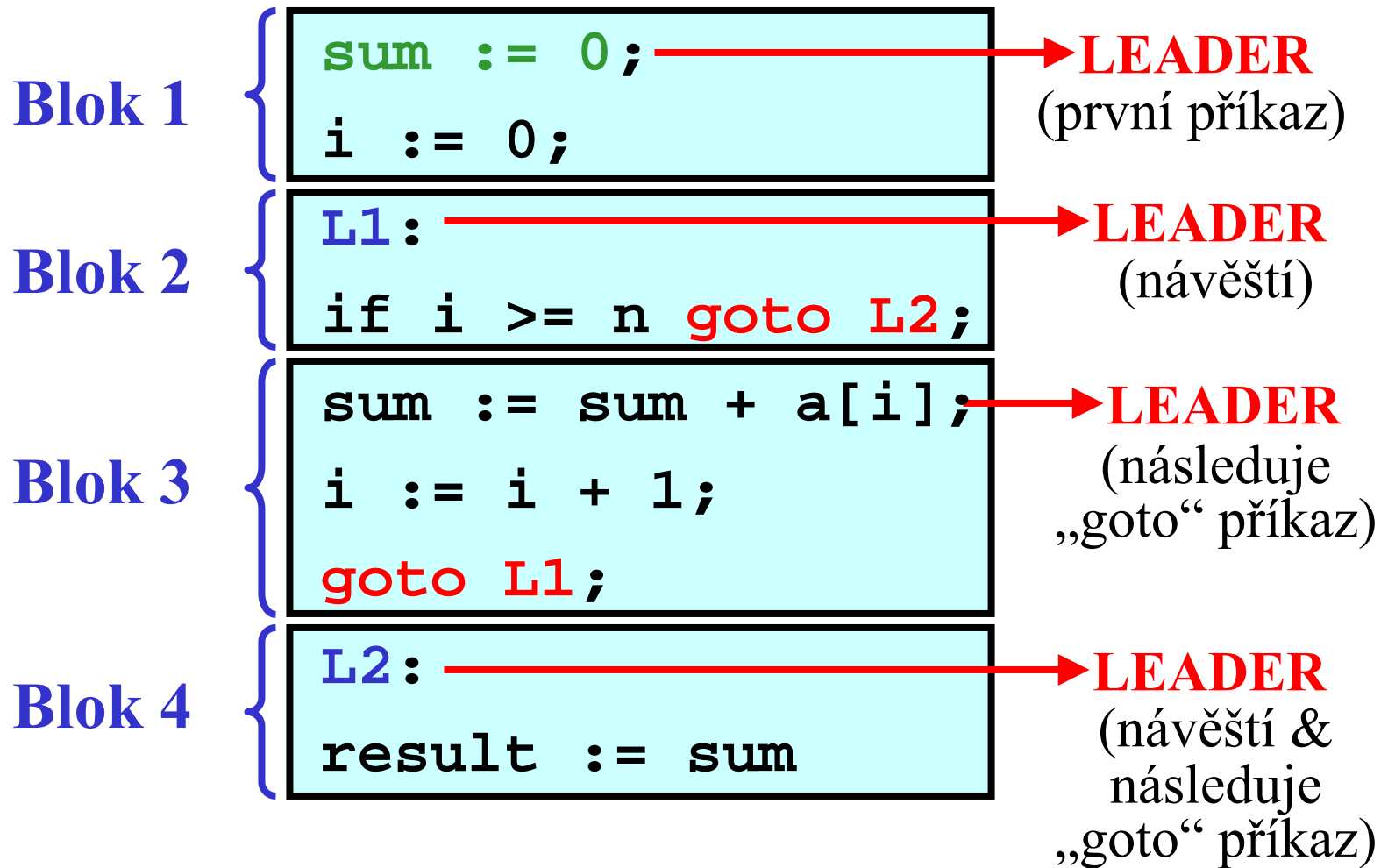
# Základní bloky

- *Základní blok* je sekvence příkazů, jejíž příkazy jsou v tomto pořadí vždy **všechny** provedeny (neexistuje skok dovnitř ani naopak zevnitř této sekvence)
- 

## Výpočet množiny vedoucích příkazů (leaderů):

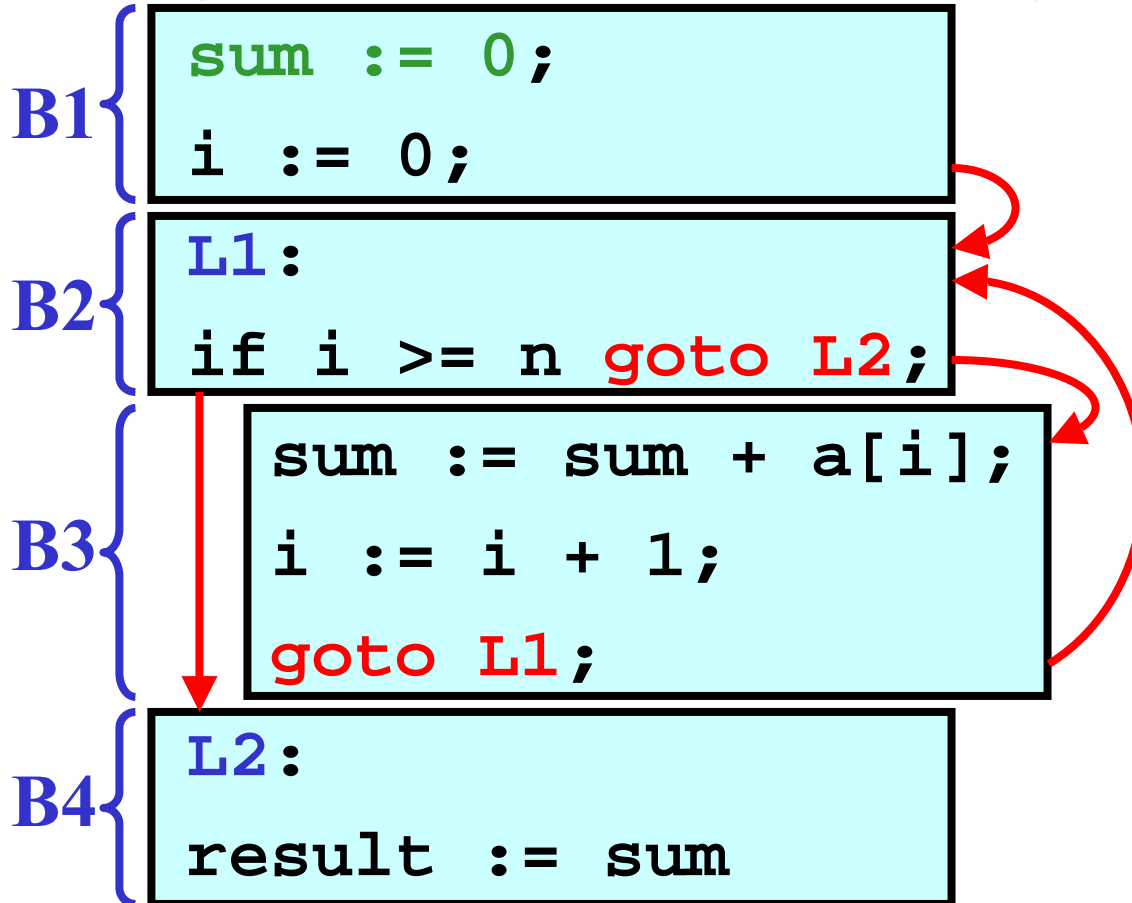
- Hned **první příkaz** je vedoucí - **leader**
- Každý příkaz, který je **návěštím** pro příkazy skoku, je vedoucí - **leader**
- Každý příkaz, který **následuje** za **goto** příkazem, je vedoucí - **leader**

# Základní bloky: Příklad

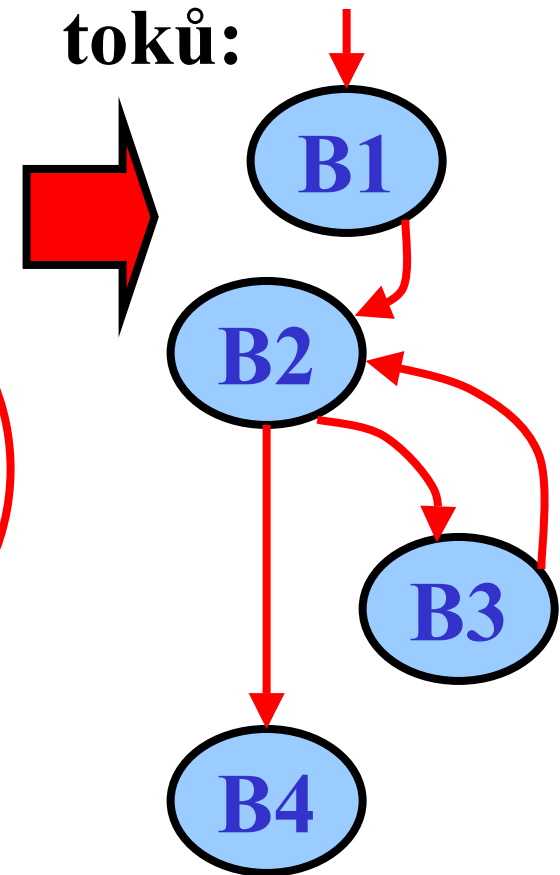


# Graf řídicích toků nad bloky

Program se základními bloky:



Graf řídicích toků:



**Pozn.:** Izolovaný blok ve grafu = **mrtvý kód**

# Optimalizace: Úvod

**Myšlenka: *Optimalizátor* dělá efektivnější verzi vnitřního kódu**

---

## Druhy optimalizací:

- 1) **Lokální optimalizace** × **Globální optimalizace**
    - **Lokální optimalizace** – v rámci základního bloku
    - **Globální optimalizace** – v rámci několika bloků
  - 2) **Optimalizace rychlosti** × **Optimalizace velikosti**
- 

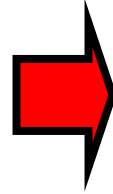
## Optimalizační metody:

- 1) Zabalení konstanty
- 2) Šíření konstanty
- 3) Kopírování proměnné
- 4) Výrazové invarianty v cyklu
- 5) Rozbalení cyklu
- 6) Eliminace mrtvého kódu

# Optimalizační metody 1/3

## 1) Zabalení konstanty

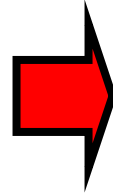
```
a := 1;  
b := 2;  
c := a + b;
```



```
c := 3;
```

## 2) Šíření konstanty

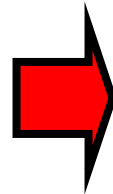
```
a := 3;  
b := a;  
c := b;
```



```
c := 3;
```

## 3) Kopírování proměnné

```
a := x;  
b := a;  
c := b;
```

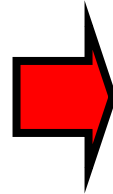


```
c := x;
```

# Optimalizační metody 2/3

## 4) Výrazové invarianty v cyklu

```
for i := 1 to 100 do  
  a[i] := p*q/r + i
```



```
x := p*q/r  
for i := 1 to 100 do  
  a[i] := x + i
```

## 5) Rozbalení cyklu

```
for i := 1 to 100 do  
begin  
  for j := 1 to 2 do  
    write(x[i, j]);  
end;
```



```
for i := 1 to 100 do  
begin  
  write(x[i, 1]);  
  write(x[i, 2]);  
end;
```

# Optimalizační metody 3/3

## 6) Eliminace mrtvého kódu

- **Mrtvý kód:** a) Nikdy nevykonán  
b) Nedělá nic užitečného

ad a)

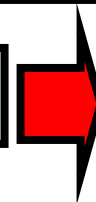
```
trace := false;  
if trace then begin  
    writeln(...);  
    ...  
end;
```



**nic**

ad b)

```
x := x;
```



**nic**



# Optimalizace velikosti

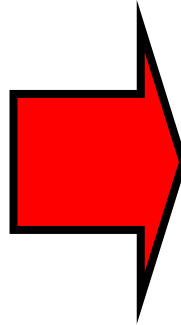
- Tato optimalizace jen vytvoří kratší program

## Příklad:

```

case p of
  1: u := a*b * c;
  2: v := a*b + c;
  3: x := d - a*b;
  4: y := d / a*b;
  5: z := 2 * a*b;
end;

```



```

T := a*b;
case p of
  1: u := T * c;
  2: v := T + c;
  3: x := d - T;
  4: y := d / T;
  5: z := 2 * T;
end;

```

- **Pozn.:** Výraz (**a\*b**) je v programu několikrát (vždy je proveden ale jen jednou)

# Generování cílového kódu: Úvod

## Typy generování cílového kódu:

- **Slepé generování** × **Kontextové generování**
- 

### 1) **Slepé generování**

- Pro každou 3AK instrukci existuje procedura, která generuje příslušný cílový kód

### Hlavní nevýhoda:

- Každá 3AK instrukce je mimo kontext ostatních instrukcí bloku, dochází tedy k přebytečným načítáním a ukládáním proměnných
- 

### 2) **Kontextové generování**

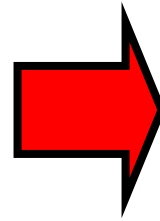
- Zredukování nepotřebných načítání a ukládání proměnných

# Slepé generování: Příklad

**3AK:**

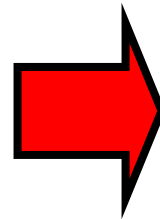
**Vygenerovaný kód:**

(+ , a , b , r)



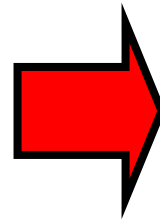
```
load  ri, a
add   ri, b
store ri, r
```

(\* , a , b , r)



```
load  ri, a
mul   ri, b
store ri, r
```

(:= , a , , r)



```
load  ri, a
store ri, r
```

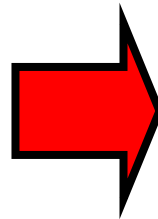
# Slepé generování

**Příklad:**

3AK:

Vygenerovaný cílový kód:

```
( + , a , b , c )  
( * , c , d , e )
```



```
load  r1, a  
add   r1, b  
store r1, c  
load  r1, c  
mul   r1, d  
store r1, e
```

**Přebytečná  
instrukce**

# Kontextové generování (KG)

- Minimalizace počtu načítání a ukládání mezi registry a paměti:
- **Obecné pravidlo:** Jestliže je hodnota proměnné v registru a bude „*brzy*“ použita, ponech ji v registru

## Potřebné informace:

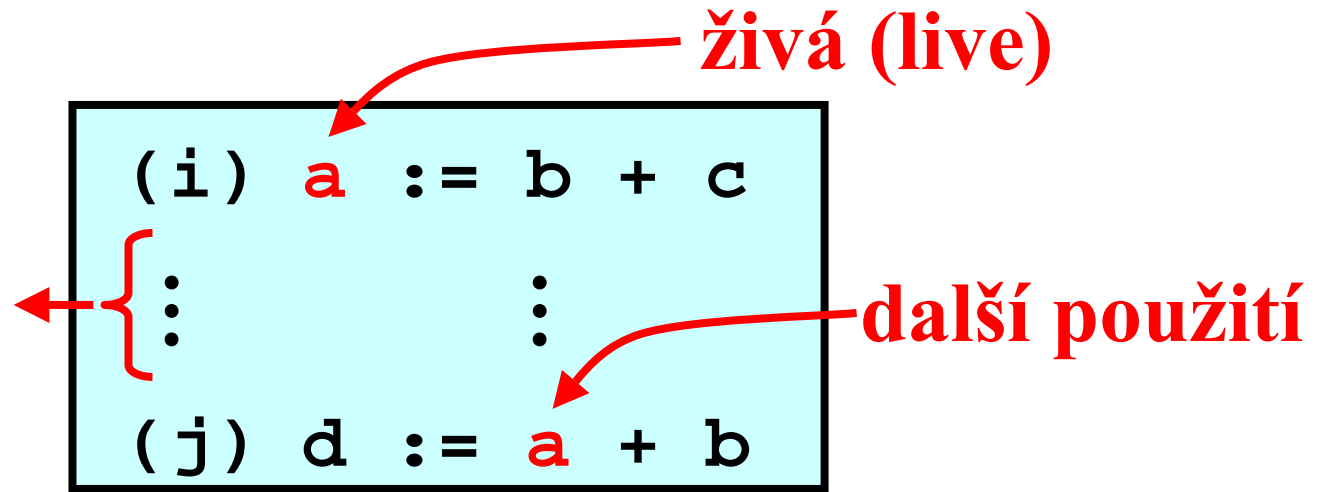
- 1) **Otázka:** Které proměnné jsou později potřeba a kde?  
**Odpověď:** v *tabulce základního bloku (TZB)*
- 2) **Otázka:** Které registry jsou použity a co v nich je?  
**Odpověď:** v *tabulce registrů (TR)*
- 3) **Otázka:** Kde je uložena aktuální hodnota dané proměnné?  
**Odpověď:** v *tabulce adres (TA)*

# KG: Analýza v základním bloku

- Proměnná je *živá*, pokud je použita v bloku později

## Příklad:

Není žádný  
výskyt  
proměnné “a”




**Otázka:** Jak detekovat efektivně živé proměnné?

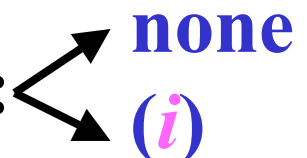
**Answer:** Aplikací *zpětného algoritmu*—to znamená, že instrukce se čtou od konce bloku směrem k začátku

# Tabluka symbolů (TS)

## Rozšíření TS:

<i>proměnná</i>	<i>stav</i>	<i>další použití</i>
a	live	(10)
b	live	(20)
pos	dead	none
⋮	⋮	⋮

*Stav:*  live  
dead

*Další použití:*  none  
(*i*)

• *i* = číslo řádku

## Inicializace:

- Programátorské proměnné: *Stav:* **live**
- Pomocné proměnné: *Stav:* **dead**
- Všechny proměnné: *Další použití:* **none**

# Tabulka základního bloku (TZB)

Struktura tabulky základního bloku:

<i>řádek</i>	<i>instrukce</i>	<i>stav</i>	<i>další použití</i>
⋮	⋮		
( <i>i</i> )	$a := b + c$		
⋮	⋮		

↑ zpětné procházení

## • Metoda:

Předpokládejme, že (*i*) je aktuální instrukce:

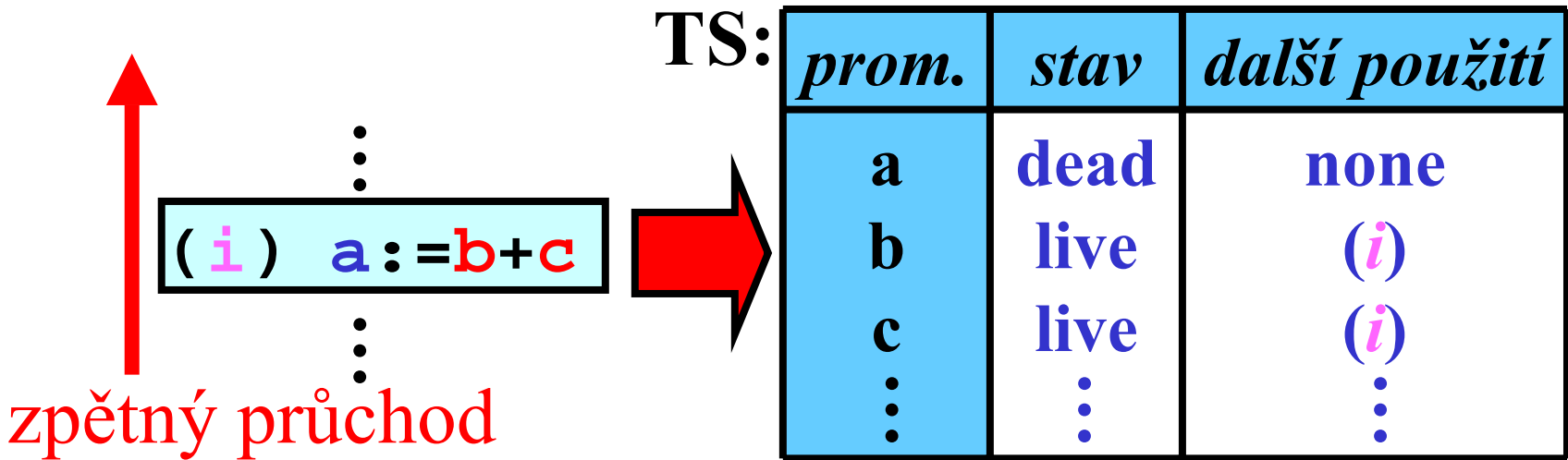
- 1) Zkopíruj *stav* a *další použití* *a*, *b*, *c* z TS do TZB
- 2) V TS proved' následující změny:

Pro proměnnou *a*: *Stav*: **dead** *Další použití*: **none**

Pro proměnné *b*, *c*: *Stav*: **live** *Další použití*: (*i*)



# Změny v TS: Ilustrace



- $a$  je mrtvá, protože  $a := b + c$  „usmrtí“ předchozí hodnotu proměnné  $a$
- $b, c$  jsou živé na řádku  $(i)$ . Tuto informaci je potřeba šířit k příkazům použitým dříve v programu

# Vyplňování TZB: Příklad 1/8

$$d := \underbrace{(a-b)}_u + \underbrace{(c-a)}_v - \underbrace{(d+b)}_x * \underbrace{(c+1)}_y$$

$\underbrace{\hspace{150px}}_w$ 
 $\underbrace{\hspace{150px}}_z$

**TZB:**

<i>řádek</i>	<i>instrukce</i>	<i>stav</i>	<i>další použití</i>
(1)	$u := a - b$		
(2)	$v := c - a$		
(3)	$w := u + v$		
(4)	$x := d + b$		
(5)	$y := c + 1$		
(6)	$z := x * y$		
(7)	$d := w - z$		

# Vyplňování TZB: Příklad 2/8

TS - řádek (7):

<i>prom.</i>	<i>stav</i>	<i>další použití</i>
a	L	N
b	L	N
c	L	N
d	L <sup>[1]</sup>	N <sup>[3]</sup>
u	D	N
v	D	N
w	D <sup>[2]</sup>	N <sup>[3]</sup>
x	D	N
y	D	N
z	D <sup>[2]</sup>	N <sup>[3]</sup>

programátorské  
proměnné

pomocné  
proměnné

L – live  
(živá)  
D – dead  
(mrtvá)  
N – none  
(nikde)

# Vyplňování TZB: Příklad 3/8

**TZB:**

<i>řádek</i>	<i>instrukce</i>	<i>stav</i>	<i>další použití</i>
(1)	$u := a - b$		
(2)	$v := c - a$		
(3)	$w := u + v$		
(4)	$x := d + b$		
(5)	$y := c + 1$		
(6)	$z := x * y$		
(7)	$d := w - z$	$d:L^{[1]}; w,z:D^{[2]}$	$d,w,z:N^{[3]}$

# Vyplňování TZB: Příklad 4/8

TS - řádek (6):

<i>var</i>	<i>stav</i>	<i>další použití</i>
a	L	N
b	L	N
c	L	N
d	D	N
u	D	N
v	D	N
w	L	(7)
x	D [1]	N [3]
y	D [1]	N [3]
z	L [2]	(7) [4]

# Vyplňování TZB: Příklad 5/8

**TZB:**

<i>řádek</i>	<i>instrukce</i>	<i>stav</i>	<i>další použití</i>
(1)	$u := a - b$		
(2)	$v := c - a$		
(3)	$w := u + v$		
(4)	$x := d + b$		
(5)	$y := c + 1$		
(6)	$z := x * y$	$z:L^{[2]}; x,y:D^{[1]}$	$z:7^{[4]}; x,y:N^{[3]}$
(7)	$d := w - z$	$d:L; w,z:D$	$d,w,z:N$

# Vyplňování TZB: Příklad 6/8

TS - řádek (5):

<i>var</i>	<i>stav</i>	<i>další použití</i>
<b>a</b>	<b>L</b>	<b>N</b>
<b>b</b>	<b>L</b>	<b>N</b>
<b>c</b>	<b>L<sup>[1]</sup></b>	<b>N<sup>[2]</sup></b>
<b>d</b>	<b>D</b>	<b>N</b>
<b>u</b>	<b>D</b>	<b>N</b>
<b>v</b>	<b>D</b>	<b>N</b>
<b>w</b>	<b>L</b>	<b>(7)</b>
<b>x</b>	<b>L</b>	<b>(6)</b>
<b>y</b>	<b>L<sup>[1]</sup></b>	<b>(6)<sup>[3]</sup></b>
<b>z</b>	<b>D</b>	<b>N</b>

# Vyplňování TZB: Příklad 7/8

**TZB:**

<i>řádek</i>	<i>instrukce</i>	<i>stav</i>	<i>další použití</i>
(1)	$u := a - b$		
(2)	$v := c - a$		
(3)	$w := u + v$		
(4)	$x := d + b$		
(5)	$y := c + 1$	$y, c: L$ <sup>[1]</sup>	$y: 6$ <sup>[3]</sup> ; $c: N$ <sup>[2]</sup>
(6)	$z := x * y$	$z: L; x, y: D$	$z: 7; x, y: N$
(7)	$d := w - z$	$d: L; w, z: D$	$d, w, z: N$

- Zbytek vyplnit analogicky



# Vyplňování TZB: Příklad 8/8

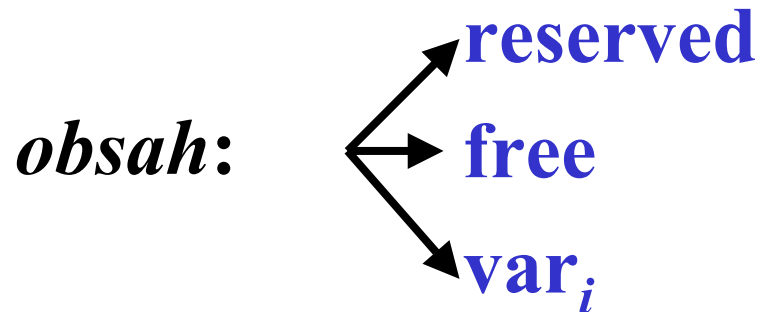
## Výsledná TZB:

<i>řádek</i>	<i>instrukce</i>	<i>stav</i>	<i>další použití</i>
(1)	$u := a - b$	$u, a, b: L$	$u: 3; a: 2; b: 4$
(2)	$v := c - a$	$v, c, a: L$	$v: 3; c: 5; a: N$
(3)	$w := u + v$	$w: L; u, v: D$	$w: 7; u, v: N$
(4)	$x := d + b$	$x, b: L; d: D$	$x: 6; d, b: N$
(5)	$y := c + 1$	$y, c: L$	$y: 6; c: N$
(6)	$z := x * y$	$z: L; x, y: D$	$z: 7; x, y: N$
(7)	$d := w - z$	$d: L; w, z: D$	$d, w, z: N$

# Tabulka registrů (TR)

## Struktura TR:

<i>reg.</i>	<i>obsah</i>
0	reserved
1	reserved
2	free
3	a
4	free
5	b
⋮	⋮



- **reserved** – registr je rezervován pro potřeby OS
- **free** – registr je volný
- **var<sub>i</sub>** – název proměnné

- Při každém použití registru je třeba modifikovat TR
- TR uchovává informace o stavech registrů

# Tabulka adres (TA)

Struktura tabulky adres:

<i>proměnná</i>	<i>adresa</i>
a	in memory
b	reg. 5
c	nowhere
⋮	⋮

*Adresa:*

- **in memory**  
(v paměti)
- **reg. *i***  
(v registru *i*)
- **nowhere**  
(nikde)

• *i* = číslo registru

- Tabulka adres ukazuje, odkud můžeme načíst aktuální hodnotu dané proměnné

## Funkce *GetReg*

- *GetReg* vrátí optimální registr pro načtení proměnné **b** např. z výrazu  $a := b + c$

*GetReg*:

begin

if **b** je v registru R and **b** je „*dead*“ and  
**b** má další použití nastaveny na „*none*“

then return R

else

if existuje volný registr R then return R

else begin

- vyber register R obsahující proměnnou, která je použita „co nejpozději“
- ulož obsah R do paměti a modifikuj TR & TA
- return R

end;

end;



# *GetReg* a *GenCode*: Příklad 1/10

**TZB:**

<i>řádek</i>	<i>instrukce</i>	<i>stav</i>	<i>další použití</i>
(1)	<b>u := a - b</b>	u,a,b:L	u:3; a:2; b:4
(2)	<b>v := c - a</b>	v,c,a:L	v:3; c:5; a:N
(3)	<b>w := u + v</b>	w:L; u,v:D	w:7; u,v:N
(4)	<b>x := d + b</b>	x,b:L; d:D	x:6; d,b:N
(5)	<b>y := c + 1</b>	y,c:L	y:6; c:N
(6)	<b>z := x * y</b>	z:L; x,y:D	z:7; x,y:N
(7)	<b>d := w - z</b>	d:L; w,z:D	d,w,z:N

**TR:**

<i>reg.</i>	<i>obsah</i>
0,1	reserved
2-11	free
12-15	reserved

**TA:**

<i>prom.</i>	<i>adresa</i>
a-d	in memory
u-z	nowhere

# *GetReg* a *GenCode*: Příklad 2/10

**Instrukce:** (1)  $u := a - b$

**Vlastnosti:**  $u, a, b$ : live

***GetReg:*** R2

***GenCode:***  
 load R2, a  
 sub R2, b

**TR:**

<i>reg.</i>	<i>obsah</i>
0,1	reserved
2-11	free
12-15	reserved

**TA:**

<i>prom.</i>	<i>adresa</i>
a-d	in memory
u-z	nowhere

# *GetReg* a *GenCode*: Příklad 3/10

**Instrukce:** (2)  $v := c - a$

**Vlastnosti:** v, c, a: live

***GetReg:*** R3

***GenCode:*** load R3, c  
sub R3, a

**TR:**

<i>reg.</i>	<i>obsah</i>
0,1	reserved
2	u
3-11	free
12-15	reserved

**TA:**

<i>prom.</i>	<i>adresa</i>
a-d	in memory
u	2
v-z	nowhere



# *GetReg* a *GenCode*: Příklad 4/10

**Instrukce:** (3)  $w := u + v$   
**Vlastnosti:** w: live; u, v: dead

---

***GetReg:*** R2  
***GenCode:*** add R2, R3

---

**TR:**

<i>reg.</i>	<i>obsah</i>
0,1	reserved
2	u
3	v
3-11	free
12-15	Reserved

**TA:**

<i>prom.</i>	<i>adresa</i>
a-d	in memory
u	2
v	3
w-z	nowhere

# *GetReg* a *GenCode*: Příklad 5/10

**Instrukce:**           (4)  $x := d + b$   
**Vlastnosti:**         **x, b: live; d: dead**

---

***GetReg:***             **R3**  
***GenCode:***         **load R3, d**  
                           **add R3, b**

---

**TR:**

<i>reg.</i>	<i>obsah</i>
0,1	reserved
2	w
3-11	free
12-15	reserved

**TA:**

<i>prom.</i>	<i>adresa</i>
a-d	in memory
u, v	nowhere
w	2
x-z	nowhere

# *GetReg* a *GenCode*: Příklad 6/10

**Instrukce:** (5)  $y := c + 1$   
**Vlastnosti:**  $y, c$ : live

---

***GetReg*:** R4  
***GenCode*:** load R4, c  
 add R4, #1

---

**TR:**

<i>reg.</i>	<i>obsah</i>
0,1	reserved
2	w
3	x
4-11	free
12-15	reserved

**TA:**

<i>prom.</i>	<i>adresa</i>
a-d	in memory
u, v	nowhere
w	2
x	3
y, z	nowhere

# *GetReg* a *GenCode*: Příklad 7/10

**Instrukce:**             $(6) \quad z := x * y$   
**Vlastnosti:**             $z: \text{live}; x, y: \text{dead}$

---

***GetReg:***                **R3**  
***GenCode:***            **mul R3, R4**

---

**TR:**

<i>reg.</i>	<i>obsah</i>
0,1	reserved
2	w
3	x
4	y
5-11	free
12-15	reserved

**TA:**

<i>prom.</i>	<i>adresa</i>
a-d	in memory
u, v	nowhere
w	2
x	3
y	4
z	nowhere

# *GetReg* a *GenCode*: Příklad 8/10

**Instrukce:** (7)  $d := w - z$   
**Vlastnosti:** d: live; w, z: dead

---

***GetReg:*** R2  
***GenCode:*** sub R2, R3

---

**TR:**

<i>reg.</i>	<i>obsah</i>
0,1	reserved
2	w
3	z
4-11	free
12-15	reserved

**TA:**

<i>prom.</i>	<i>adresa</i>
a-d	in memory
u, v	nowhere
w	2
x, y	nowhere
z	3

# *GetReg* a *GenCode*: Příklad 9/10

**Instrukce:** *konec bloku*  
**Vlastnosti:** *d: live;*

---

***GetReg:*** -

***GenCode:*** *store R2,d*  
 (uložit všechny živé proměnné!)

---

**TR:**

<i>reg.</i>	<i>obsah</i>
0,1	reserved
2	d
3-11	free
12-15	reserved

**TA:**

<i>prom.</i>	<i>adresa</i>
a-c	in memory
d	2
u-z	nowhere

# *GetReg* a *GenCode*: Příklad 10/10

- Výsledný kód: 12 instrukcí místo  $7 \cdot 3 = 21$ !

<i>řádek</i>	<i>3AC</i>	<i>generated code</i>
(1)	<code>u := a - b</code>	<code>load R2, a</code> <code>sub R2, b</code>
(2)	<code>v := c - a</code>	<code>load R3, c</code> <code>sub R3, a</code>
(3)	<code>w := u + v</code>	<code>add R2, R3</code>
(4)	<code>x := d + b</code>	<code>load R3, d</code> <code>add R3, b</code>
(5)	<code>y := c + 1</code>	<code>load R4, c</code> <code>add R4, #1</code>
(6)	<code>z := x * y</code>	<code>mul R3, R4</code>
(7)	<code>d := w - z</code>	<code>sub R2, R3</code> <code>store R2, d</code>

# Paralelní kompilátory: Úvod

- *Lexikální analyzátor* přeloží **celý** zdrojový program na tokeny

---

- **Příprava syntaktické analýzy pro paralelní překlad:**
  - Vyjmutí nějakých částí programů (podřetězců tokenů jako např. výrazy, podmínky, ...). Pro tyto podřetězce a pro zbytek (nazývaný **kostra**) je prováděna syntaktická analýza paralelně
  - V kostře jsou tyto vyjmuté podřetězce nahrazeny tzv. **pseudotokeny**, na jejich pozice je později vložen jejich kód.

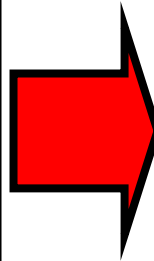


# Paralelní kompilátory: Vyjmutí podmínek

```

:
:
if cond1 then ...
:
:
while cond2 do ...
:
:
repeat ... until cond3
:
:

```



```

:
:
if [cond, 1] then ...
:
:
while [cond, 2] do ...
:
:
repeat ... until [cond, 3]
:
:

```

- Tabulka podmínek:

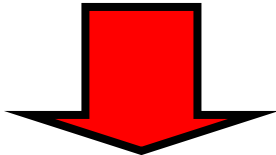
1	<i>cond</i> <sub>1</sub>
2	<i>cond</i> <sub>2</sub>
3	<i>cond</i> <sub>3</sub>

# Paralelní kompilátory: Víceúrovňové vyjmutí

```

:
:
if  $a + b > c * d$  and  $a - b = c + d$  then ...
:
:

```



```

:
:
if [cond, 1] then ...
:
:

```

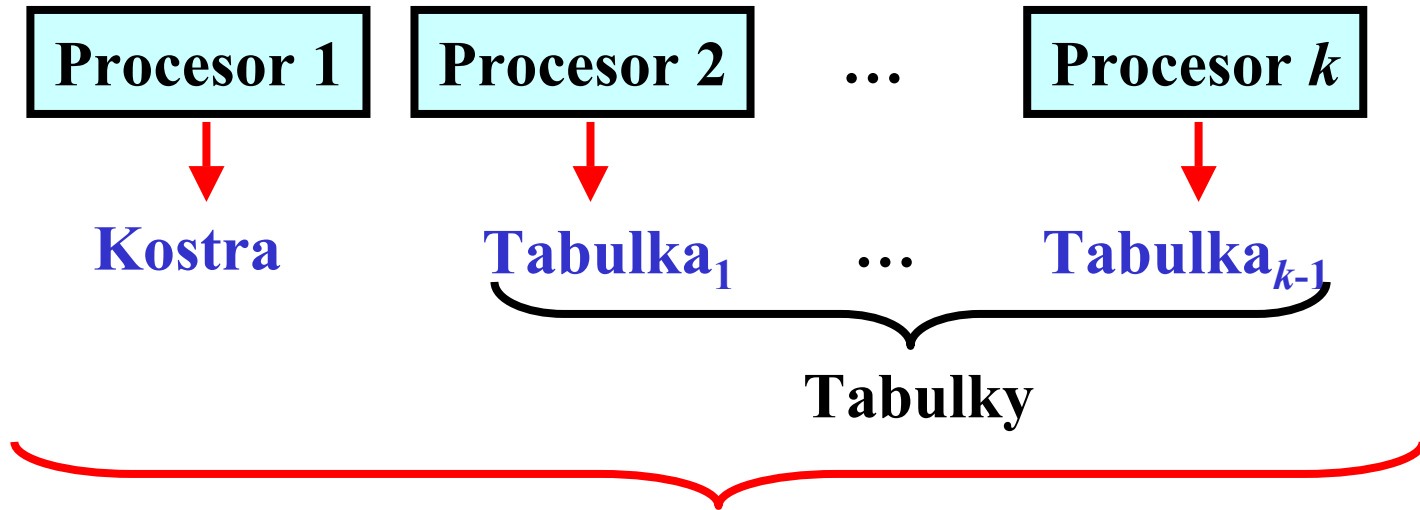
• Tabulka podmínek:

1	$[expr, 1] > [expr, 2]$ and $[expr, 3] = [expr, 4]$
2	...

• Tabulka výrazů:

1	$a + b$
2	$c * d$
3	$a - b$
4	$c + d$

# Paralelní kompilátory: Synt. analýza



## Paralelní syntaktická analýza

- mohou být rozdílné metody 1 –  $k$
- mohou být rozdílné vnitřní kódy