Kapitola VIII. Optimalizace a generování cílového programu

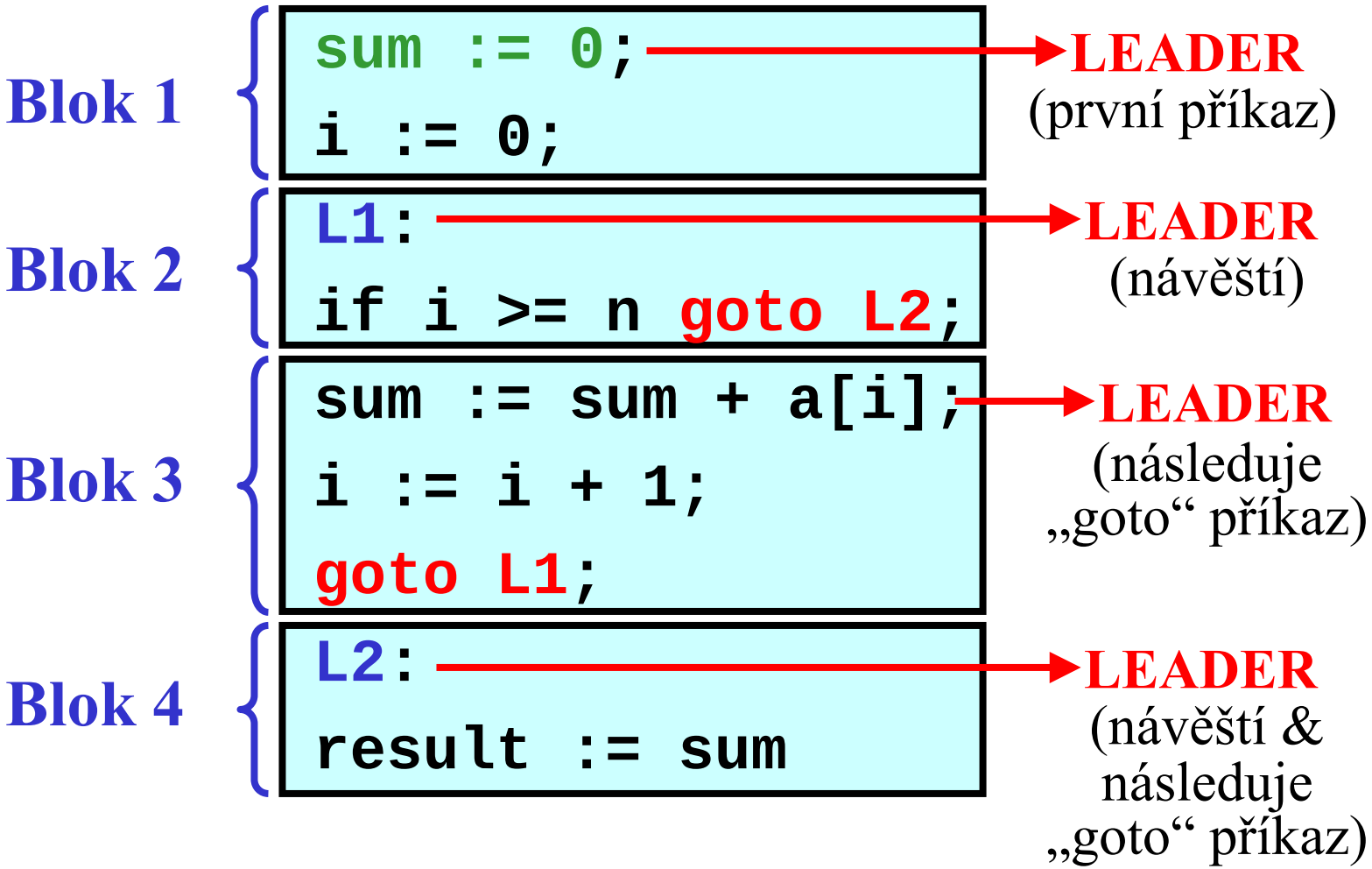
Základní bloky

• ***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:

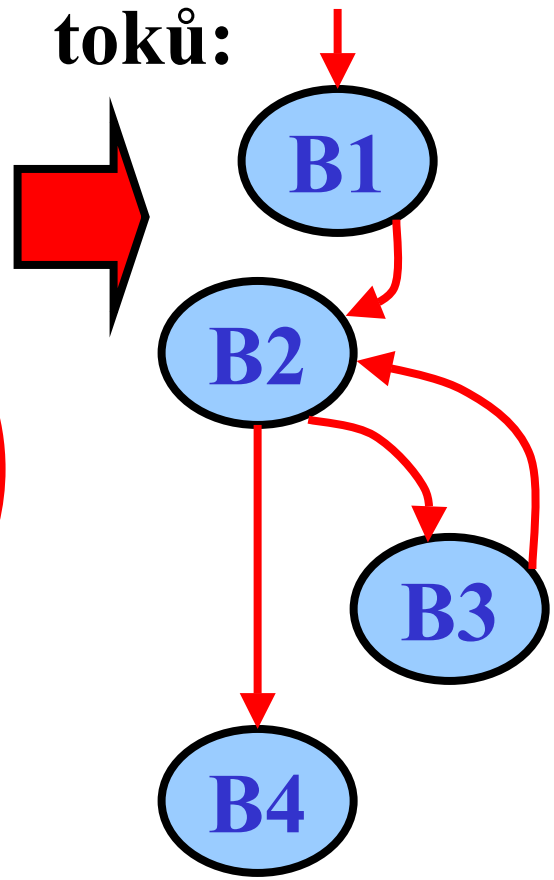
B1
```
sum := 0;
i := 0;
```

B2
```
L1:
if i >= n goto L2;
```

B3
```
sum := sum + a[i];
i := i + 1;
goto L1;
```

B4
```
L2:
result := sum
```

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

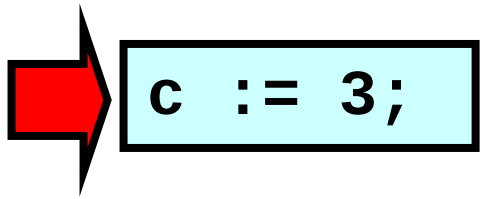
Optimalizační metody:

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

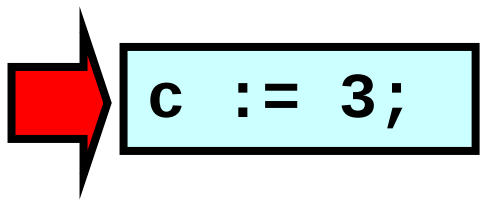
```
a := 1;
b := 2;
c := a + b;
```



```
c := 3;
```

2) Šíření konstanty

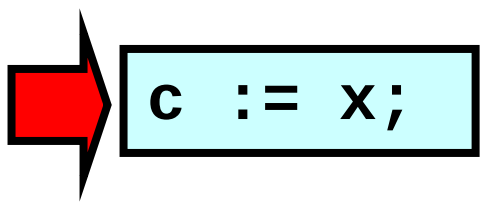
```
a := 3;
b := a;
c := b;
```



```
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

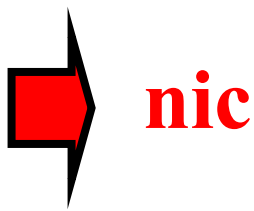
6) Eliminace mrtvého kódu

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

---

ad **a)**

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

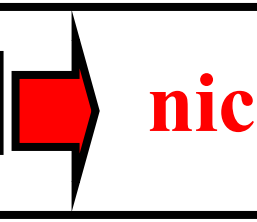


nic

---

ad **b)**

```
x := x;
```



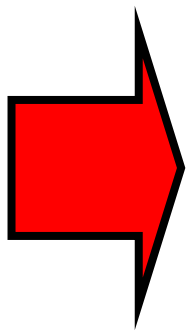
nic

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:

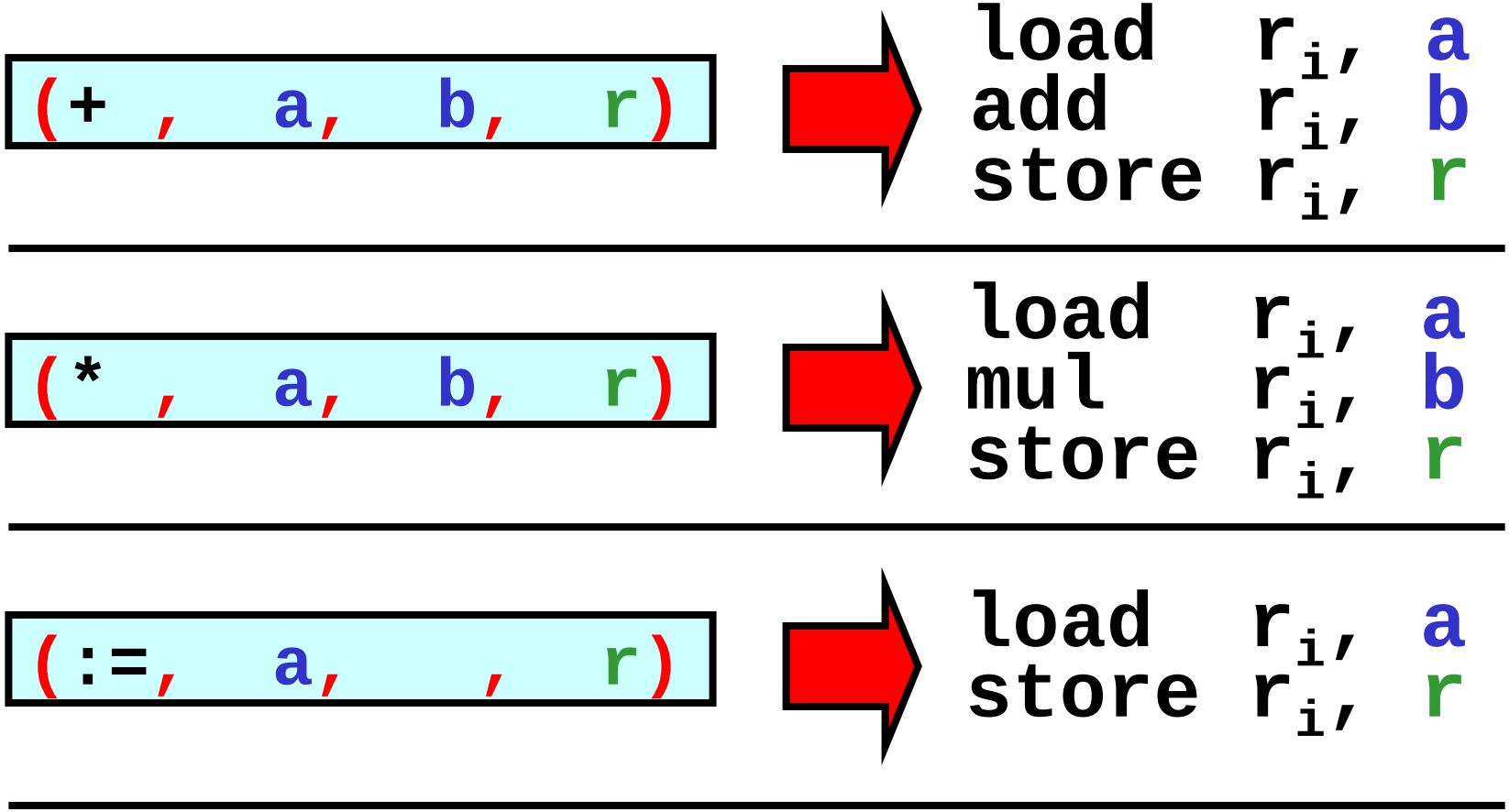
- Každá 3AK instrukce je mimo kontext ostatních instrukcí bloku, dochází tedy k přebytečným načítáním a ukládáním proměnných

2) Kontextové generování

- Zredukování nepotřebných načítaní a ukládání proměnných

Slepé generování: Příklad

3AK: **Vygenerovaný kód:**



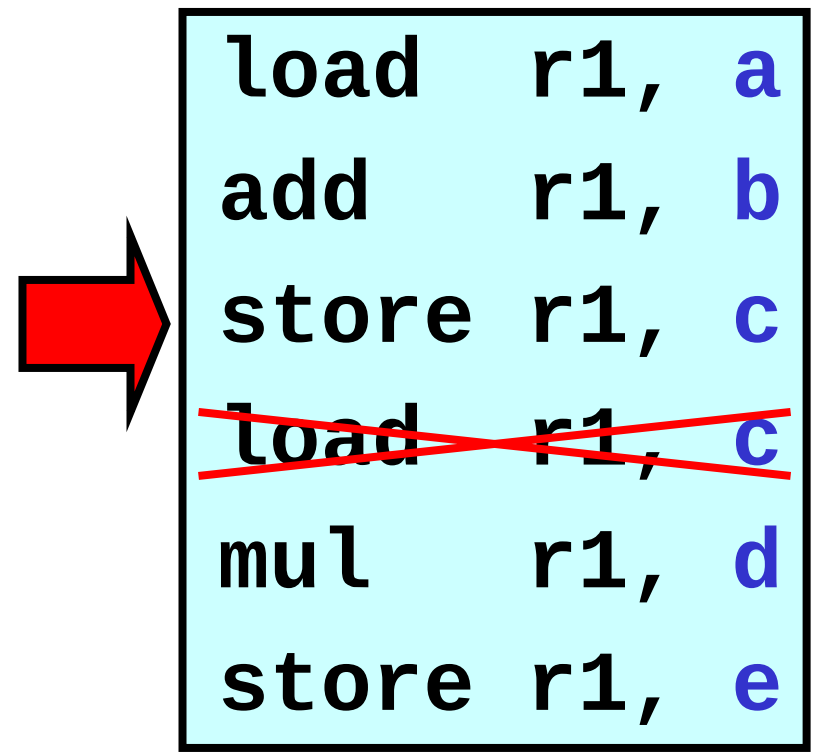
Slepé generování

Příklad:

3AK:

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

Vygenerovaný cílový kód:



Přebytečná instrukce

Kontextové generování (KG)

• Minimalizace počtu načítání a ukládání mezi registry a pamětí:

• **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**)

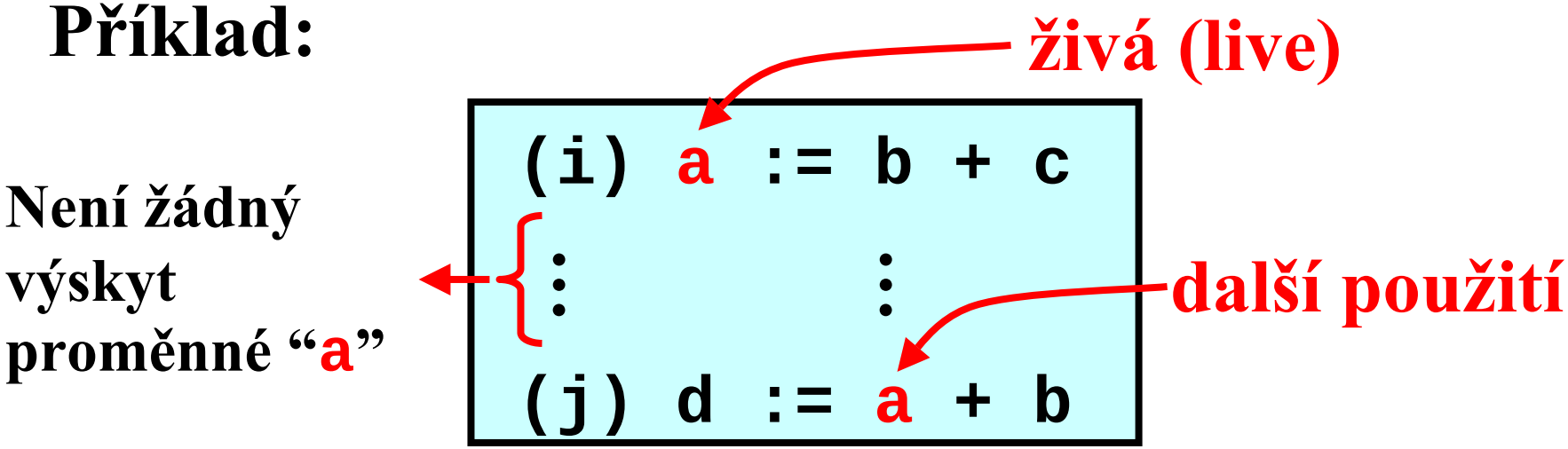
2) Otázka: Které registry jsou použity a co v nich je?
Odpověď: v ***tabulce registrů*** (**TR**)

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:



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:

proměnná	*stav*	*další použití*
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:

řádek	*instrukce*	*stav*	*další použití*
⋮	⋮		
(*i*)	**a:=b+c**		
⋮	⋮		

zpětné procházení

- **Metoda:**

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

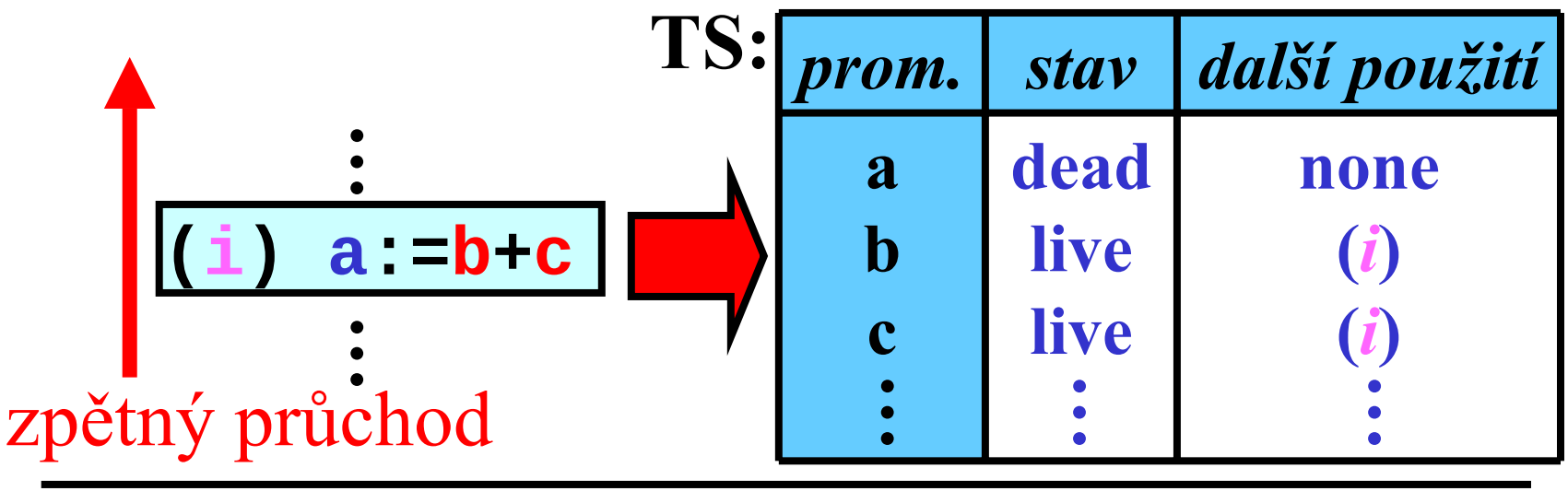
1) Zkopíruj ***stav*** a ***další použití*** ***a*** ,***b*** ,***c*** z **TS** do **TZB**

2) V **TS** proveď 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 := (a-b) + (c-a) - (d+b) * (c+1)
```

u v x y

w z

TZB:

řádek	*instrukce*	*stav*	*další použití*
(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):

prom.	*stav*	*další použití*
a	L	N
b	L	N
c	L	N
d	L [1]	N [3]
u	D	N
v	D	N
w	D [2]	N [3]
x	D	N
y	D	N
z	D [2]	N [3]

programátorské proměnné

pomocné proměnné

L – live (živá)

D – dead (mrtvá)

N – none (nikde)

Vyplňování TZB: Příklad 3/8

TZB:

řádek	*instrukce*	*stav*	*další použití*
(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):

var	*stav*	*další použití*
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:

řádek	*instrukce*	*stav*	*další použití*
(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):

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

Vyplňování TZB: Příklad 7/8

TZB:

řádek	*instrukce*	*stav*	*další použití*
(1)	`u:=a-b`		
(2)	`v:=c-a`		
(3)	`w:=u+v`		
(4)	`x:=d+b`		
(5)	`y:=c+1`	y,c:L [1]	y:6 [3]; c:N [2]
(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:

řádek	*instrukce*	*stav*	*další použití*
(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:

reg.	*obsah*
0	reserved
1	reserved
2	free
3	a
4	free
5	b
⋮	⋮

***obsah*:** **reserved**, **free**, **var**$_i$

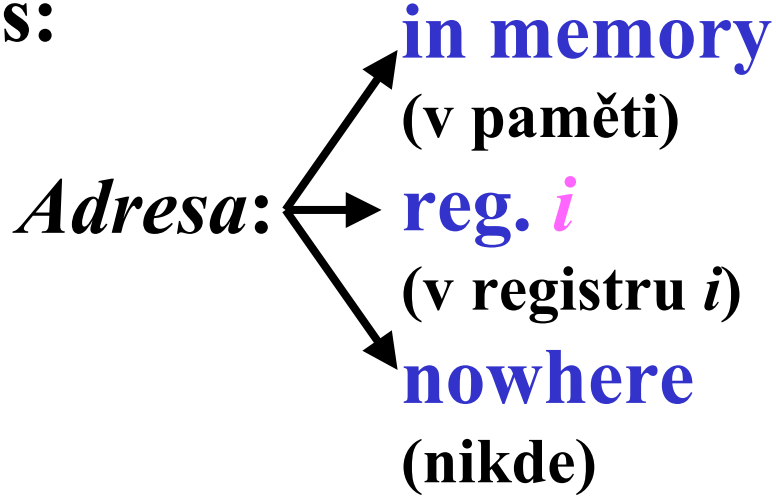
- **reversed** – registr je rezervován pro potřeby OS
- **free** – registr je volný
- **var**$_i$ – název proměnné

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

Tabulka adres (TA)

Struktura tabulky adres:

proměnná	*adresa*
a	in memory
b	reg. 5
c	nowhere
⋮	⋮



- *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;

Procedura *GenCode*

• ***GenCode*** generuje optimální kód např. pro výraz **a:=b+c**

***GenCode*:**
begin
- Zavolej ***GetReg*** pro výběr registru **R** pro **b**
- **if b** není v **R then** generate(**"load R,b"**)
- **if c** je v **S then** generate(**"add R,S"**)
  else generate(**"add R,c"**)
  // znamená, že c je v paměti
- Modifikuj TR & TA tak, aby určovaly, že současná hodnota proměnné **a** je v registru **R**
- **if c** je v **S and c** je „*dead*“ **and c** má další použití nastaveny na „*none*“ **then** nastav stav proměnné **S** v TR jako „*free*“

end;

GetReg a *GenCode*: Příklad 1/10

TZB:

řádek	*instrukce*	*stav*	*další použití*
(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

TR:

reg.	*obsah*
0,1	reserved
2-11	free
12-15	reserved

TA:

prom.	*adresa*
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:

reg.	*obsah*
0,1	reserved
2-11	free
12-15	reserved

TA:

prom.	*adresa*
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:

reg.	*obsah*
0,1	reserved
2	u
3-11	free
12-15	reserved

TA:

prom.	*adresa*
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:

reg.	*obsah*
0,1	reserved
2	u
3	v
3-11	free
12-15	Reserved

TA:

prom.	*adresa*
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:

reg.	*obsah*
0,1	reserved
2	w
3-11	free
12-15	reserved

TA:

prom.	*adresa*
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:

reg.	*obsah*
0,1	reserved
2	w
3	x
4-11	free
12-15	reserved

TA:

prom.	*adresa*
a-d	in memory
u, v	nowhere
w	2
x	3
y, z	nowhere

GetReg a *GenCode*: Příklad 7/10

Instrukce: `(6) z:=x*y`
Vlastnosti: **z: live; x, y: dead**

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

TR:

reg.	*obsah*
0,1	reserved
2	w
3	x
4	y
5-11	free
12-15	reserved

TA:

prom.	*adresa*
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:

reg.	*obsah*
0,1	reserved
2	w
3	z
4-11	free
12-15	reserved

TA:

prom.	*adresa*
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:

reg.	*obsah*
0,1	reserved
2	d
3-11	free
12-15	reserved

TA:

prom.	*adresa*
a-c	in memory
d	2
u-z	nowhere

GetReg a *GenCode*: Příklad 10/10

- **Výsledný kód: 12 instrukcí místo 7*3 = 21!**

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

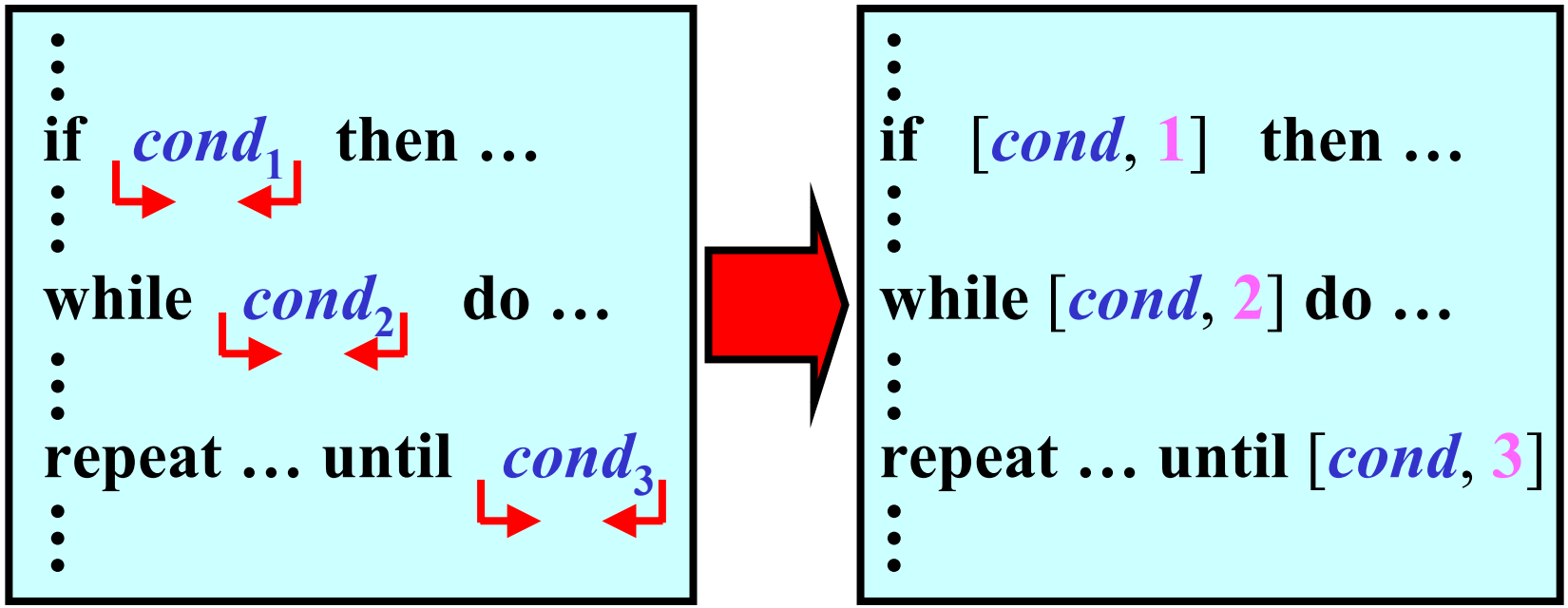
Paralelní kompilátory: Úvod

- ***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.

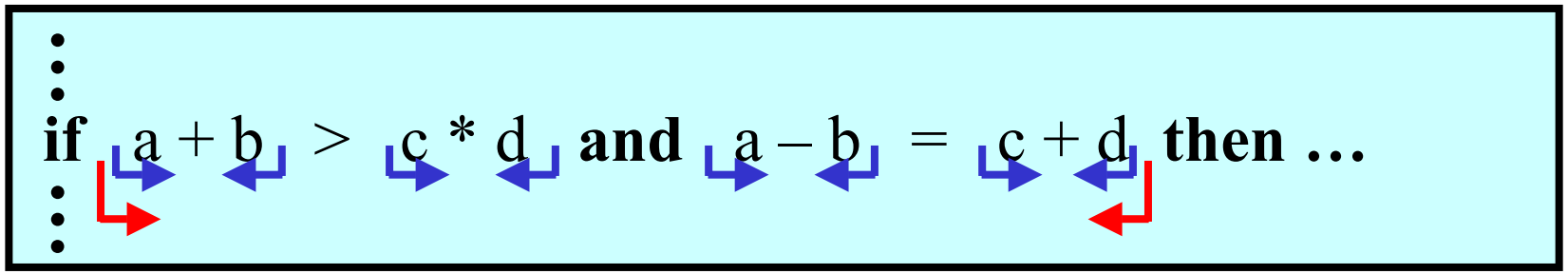
Paralelní kompilátory: Vyjmutí podmínek

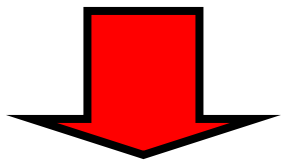


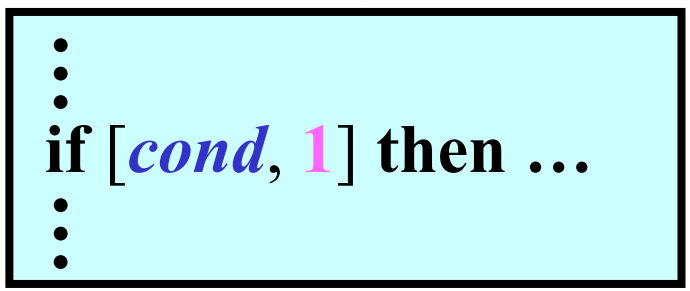
- **Tabulka podmínek:**

1	$cond_1$
2	$cond_2$
3	$cond_3$

Paralelní kompilátory: Víceúrovňové vyjmutí







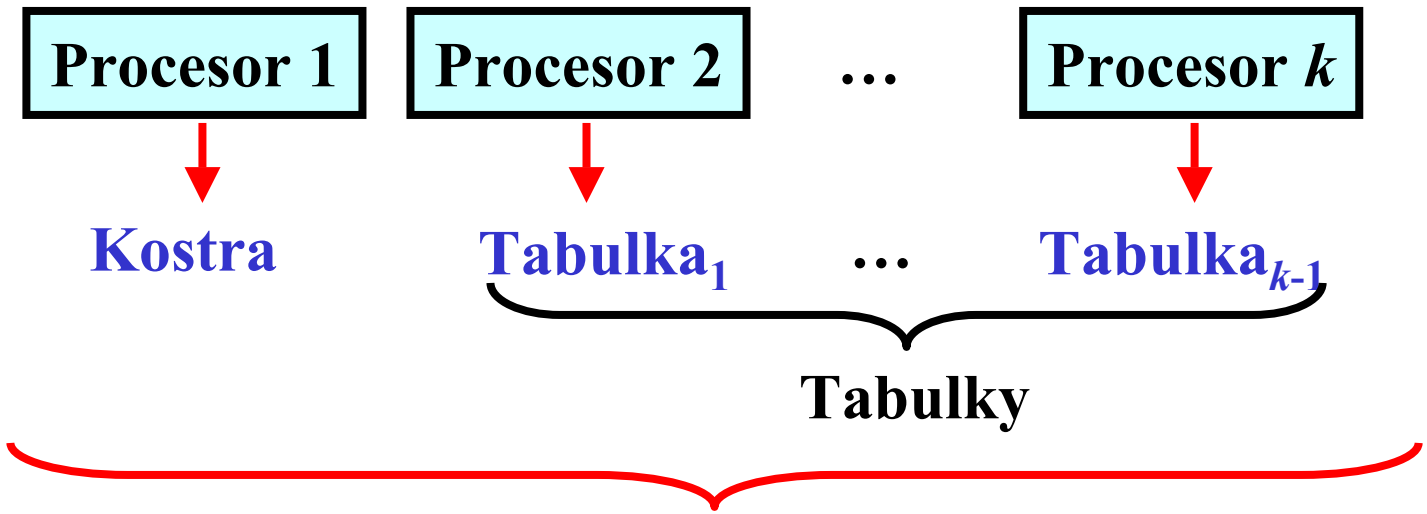
- **Tabulka výrazů:**

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

- **Tabulka podmínek:**

1	[*expr*, 1] > [*expr*, 2] and [*expr*, 3] = [*expr*, 4]
2	…

Paralelní kompilátory: Synt. analýza



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