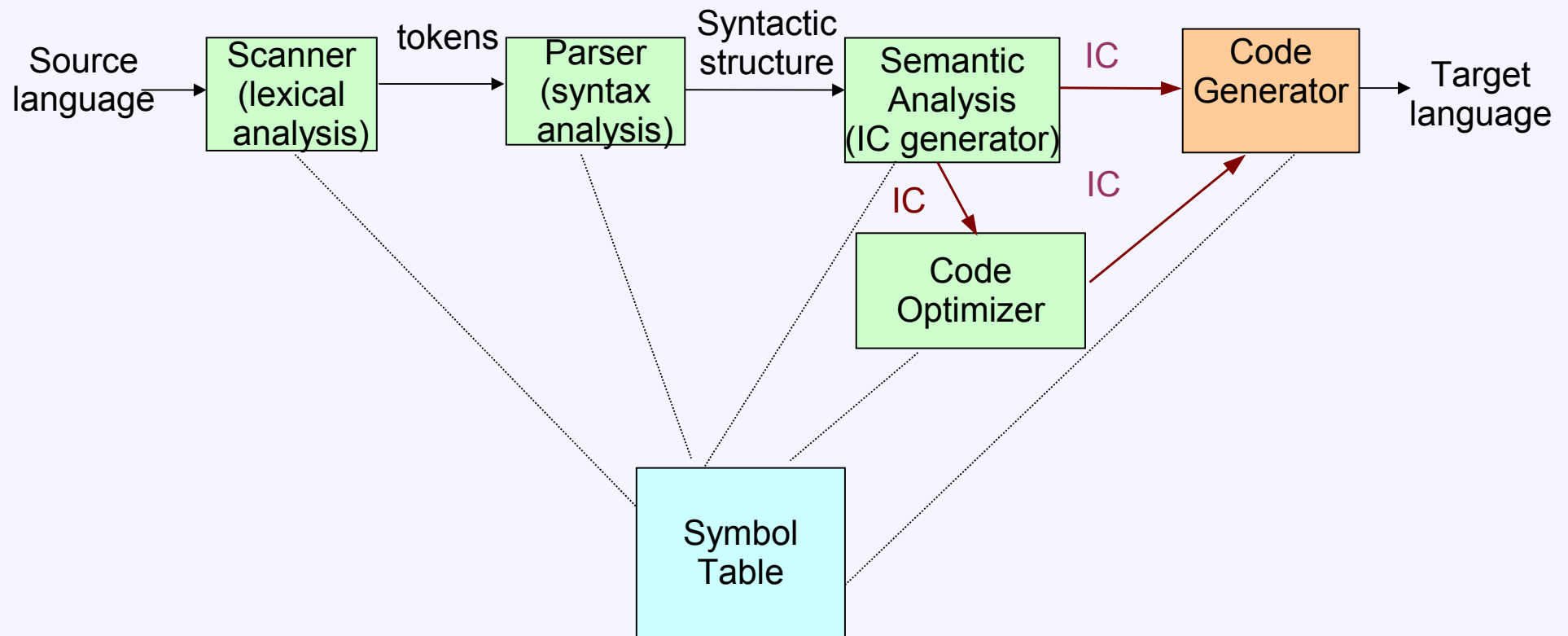


Generovanie kódu a lokálna optimalizácia

Ján Šturc

Podľa fialového draka.

Generátor kódu v kompilátore



Cieľové architektúry

- RISC (reduced instruction set computer)
 - trojadresové inštrukcie
 - veľa registrov
 - jednoduchý pamäťový cyklus (málo adresných módov)
- SIC (single instruction computer)
 - risc ad absurdum (Ari Tabak)
- CISC (complex instruction set computer)
 - dvojadresové inštrukcie
 - menej registrov (dnes často procesorová cache)
 - komplikovaný pamäťový cyklus (viac adresných módov)
- SBM (stack base machine)
 - 1963 Burroughs B5000, boli aj ruské pokusy Zebra
 - neosvečilo sa to, mnoho zbytočných swapov
 - trochu to oživuje JVM (java virtual machine)

Úryvok z nového vydania „dračej knihy“.

In the 1990s, stack-based architectures were revived with the introduction of the Java Virtual Machine (JVM). The JVM is a software interpreter for Java bytecodes, an intermediate language produced by Java compilers. The interpreter provides software compatibility across multiple platforms, a major factor in the success of Java.

To overcome the high performance penalty of interpretation, which can be on the order of a factor of 10, *just-in-time* (JIT) Java compilers have been created. These JIT compilers translate bytecodes during run time to the native hardware instruction set of the target machine. Another approach to improving Java performance is to build a compiler that compiles directly into the machine instructions of the target machine, bypassing the Java bytecodes entirely.

Čo generovať ?

- **Strojový kód**
 - Relatívne najjednoduchšie: definujeme si konštanty – symbolické mená inštrukcií.
 - Generujeme priamo do pamäti a inštrukcie obsahujú absolútne adresy
- **Binárny relokatívny kód (BRC)**
 - Inštrukcie sa generuju rovnako ako v predošlom prípade
 - Adresy sú také, že program môže byť umiestnený kdekoľvek v pamäti.
 - Loader (linker) to zariadí.
 - Umožňuje separátnu kompiláciu modulov
- **Assembler**
 - Generujeme symbolické mená inštrukcií a symbolické adresy
 - Prekladá sa assemblerom
 - Zbytočná robota, vyskytuje sa len v študentských projektoch.

Mapovanie medzijazyka do strojového kódu

- Úlohou je mapovať medzijazyk do strojového kódu cieľového stroja. Zložitosť tejto úlohy je určená tromi faktormi
 1. Úroveň medzijazyka
 2. Množina inštrukcií a jej štruktúra (instruction set architecture)
 3. Požadovaná kvalita výsledného kódu
 4. Rýchlosť vykonania inštrukcií a „machine idioms“

Ak nemáme vysoké nároky na efektívnosť výsledného kódu, môžeme každú inštrukciu medzijazyka naprogramovať ako makro alebo šablónu programu a výsledný program skladať z takýchto „makier“.

Ak cieľová architektúra je nehomogénna s veľa obmedzeniami premietne sa to do zložitosti mapovania. Napr. pre operácie s reálnymi číslami sú vyhradené špeciálne registre.

Ak úroveň medzijazyka je príliš vysoká, je celkom možné, že nevystačíme s makrami, ale budeme musieť niektoré inštrukcie interpretovať s využitím špecializovaných dátových štruktúr.

Príklady

```
x:= y + z;      LD  R0, y      // R0:= y
                 ADD R0, R0, z  // R0:= R0 + z
                 ST  x, R0      // x:= R0
```

```
a:= b + c;      LD  R0, b      // R0:= b
d:= a + e;      ADD R0, R0, c  // R0:= R0 + c
                 ST  a, R0      // a:= R0
                 LD  R0, a      // R0:= a
                 ADD R0, R0, e  // R0:= R0 + e
                 ST  d, R0      // d:= R0
```

```
a:= a + 1;      LD  R0, a      // R0:= b
                 ADD R0, R0, #1 // R0:= R0 + 1
                 ST  a, R0      // a:= R0
```

Machine idiom:
INC a, #1

Model cieľového počítača

Presuny:

LD $\langle r_i \rangle$, $\langle \text{address} \rangle$

ST $\langle r_i \rangle$, $\langle \text{address} \rangle$

MV r_i , r_j

Prečo nie univerzálna inštrukcia:

MV $\langle \text{destination} \rangle$ $\langle \text{source} \rangle$?

Výpočtové operácie:

OP $\langle r_i \rangle$, $\langle r_j \rangle$, $\langle r_k \rangle$ // $r_i := r_j \text{ op } r_k$

Skoky:

JMP $\langle \text{address} \rangle$

BR $\langle \text{cond} \rangle$, $\langle r_i \rangle$, $\langle \text{address} \rangle$

Spôsoby adresácie:

#a operand je adresná konštanta

a bez modifikácie operand = $*(a)$

* inderektná adresa operand = $*(*(a))$

modifikácia registrom $*(a + r)$, $*(a + *r)$ alebo aj $*(a + r)$?

autorelatívne adresy

Prehľad spôsobov adresácie

MODE	FORM	ADDRESS	ADDED COST
absolute	M	M	1
register	R	R	0
indexed	c(R)	c + contents(R)	1
indirect register	*R	contents(R)	0
indirect indexed	*c(R)	contents(c+contents(R))	1
literal	#c	c	1
stack	SP	SP	0
indexed stack	c(SP)	c + contents(SP)	1

Dĺžka a cena inštrukcií

1. Počet inštrukcií	16 – 256
2. Počet registrov	4 – 64
3. Priamo adresovateľná pamäť	256 – 4G
4. Počet spôsobov adresácie	1 – 8

3 typy inštrukcií

0. typ: OP r MOD a	48 bit
1. typ: OP r_i, r_j	16 bit
2. typ: OP r_i, r_j, r_k	24 – 32 bit

Významnú úlohu hrajú aj:

- šírka dátovej zbernice
- šírka inštrukčnej zbernice

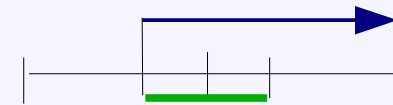
Cena inštrukcie = počet pamäťových cyklov

Pamäťový cyklus: prečítanie a obnovenie.

Adresný cyklus: výpočet adresy (obvykle jeden pamäťový cyklus za každú indirekciu).

Inštrukčný cyklus:

- Načítanie inštrukcie, dekódovanie inštrukcie
- Výpočet adresy
- Načítanie operandu
- Vykonanie operácie

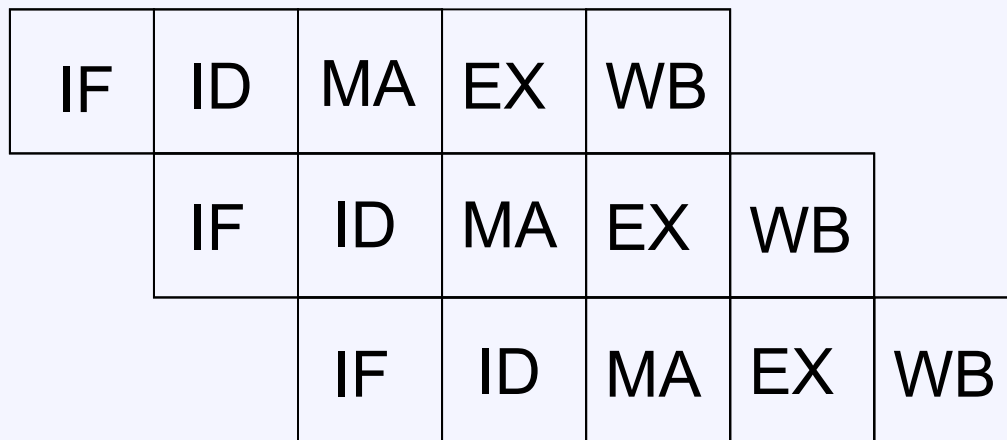


Špecializované a externé zariadenia

- Prúdove spracovanie (SIMD, pole procesorov, pipe)
 - Napr. násobičky, sčítačky
 - Predspracovanie (potrebná administrácia), inicializácia
 - Oneskorenie (prvý výsledok)
 - Perióda čas medzi dvomi po sebe idúcimi výsledkami
- Cache (často 2 – 3 úrovne)
 - inštrukčná
 - dátová
- Externé DMA blokové prenosy
 - Štartovacia inštrukcia, potom už pracuje paralelne
 - Dvojbufrová administrácia
 - RAID

Podľa „hardwaristov“ sú tieto veci pre užívateľa transparentné. Skutočnosť: program generovaný kompilátorom, ktorý pozorne nezohľadňuje možnosti týchto zariadení, dosahuje len zlomok ich teoretickej výkonnosti.

Pipeline – prúdové spracovanie



IF – Instruction Fetch
ID – Instruction Decode
EX – Execute
MA – Memory access
WB – Write back

Načítanie inštrukcie
Dekódovanie inštrukcie
Vykonanie inštrukcie
Načítanie operandu
Obnovenie pamäti

Konflikty – hazards

Štrukturálne:
Ohraničenia na súčasné použitie zdrojov počítača.

Dátové:
Potrebujeme výsledok ešte neskončenej inštrukcie.

Riadenia:
Podmienené skoky

Riešenie konfliktov:

Dátové a štrukturálne konflikty sa dajú riešiť pozdržaním (zariadením prázdneho taktu).

Konflikty spôsobené skokmi vyžadujú zrušenie časti výpočtu a začať znovu. Rafinované stratégie a odhady.


Pridelovanie registrov

- Dve úlohy
 - **Register allocation:** Pre každý bod programu určíme množinu premenných, ktorá bude v danom bode v registroch
 - **Register assignment:** Vyberáme konkrétny register pre danú premennú.
- Register assignment je matematicky (NP) ťažká úloha
 - Ďalšou komplikáciou je skutočnosť, že niektoré operácie vyžadujú konkrétne dvojice registrov. (napr. MQ pre IBM370).
 - Párny + nepárny register pre operácie s dvojitou dĺžkou slova.

Príklad: $t := a + b$ LD R₀, #0
 $t := t * c$ LD R₁, a
 $t := t / d$ ADD R₁, b
MUL R₀ c
DIV R₀, d
ST R₁, t

$t := a + b$ LD R₀, a
 $t := t + c$ ADD R₀, b
 $t := t / d$ ADD R₀, c
SHDAR R₀, #32
DIV R₀, d
ST R₁, t

Shift double
arithmetic right



Správa pamäti počas behu

- Pre každú procedúru tri oblasti (areas)
 - Oblasť kódu (do tej sa generuje kód, je statická)
 - Oblasť statických dát
 - Oblasť dát na zásobníku
- Halda je spoločná a riadí sa explicitnými príkazmi alokácie a dealokácie.
- Volajúci musí:
 - Alokovat' na zásobníku priestor pre volací rekord volaného.
 - Uložit' doň potrebné údaje.
 - Vyhradiť priestor pre vrátené hodnoty.
 - Ovzdat' riadenie volanému (skočiť na jeho vstupnú adresu v oblasti kódu).
- Volaný musí
 - Vykonať prácu
 - Upratať po sebe na zásobníku
 - Vrátiť riadenie volanému

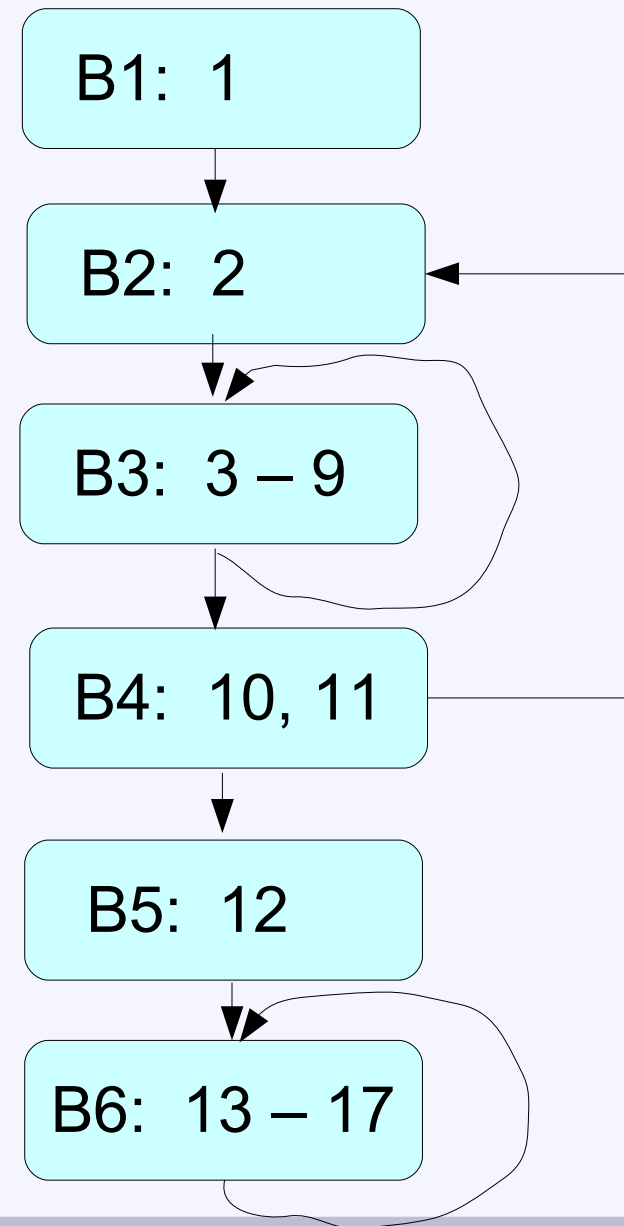
Rozdelenie programu na bloky

- Blok (interval) je postupnosť za sebou idúcich príkazov programu, ktorá, keď sa začne vykonávať, sa vždy vykoná celá.
- Blok sa skladá z hlavičky (leader) a ostatných príkazov bloku.
- Algoritmus rozdelenia na bloky spočíva v určení hlavičiek.
 - Prvý príkaz programu je hlavička bloku.
 - Každý cieľ skoku je hlavička bloku.
 - Príkaz bezprostredne nasledujúci za príkazom skoku je hlavička bloku.
 - Neexistujú iné hlavičky blokov.
- Blok je úsek programu začínajúci hlavičkou po nasledujúcu hlavičku.
- Blok obsahuje najviac jeden skok (na svoju hlavičku).

Príklad

```
for (i=1, i <= 10, i++)  
  for (j=1, i <= 10, j++)  
    a[i, j] = 0.0;  
for (i=1, i <= 10, i++)  
  a[i, i] = 1.0;
```

1. $j := 1$
2. $i := 1$
3. $t_1 := 10 * i$
7. $t_2 := t_1 + j$
8. $t_3 := 8 * t_2$
9. $t_4 := 88 - t_3$
10. $a[t_4] := 0.0$
11. $j := j + 1$
12. if $j \leq 10$ go to (3)
19. $i := i + 1$
21. if $i \leq 10$ go to (2)
23. $i := 1$
24. $t_5 := i - 1$
27. $t_6 := 88 * t_5$
28. $a[t_6] := 1.0$
29. $i := i + 1$
30. if $i \leq 10$ go to (13)



Next use information (ďalšie použitie)

- Definícia p. $x := \dots$ má ďalšie použitie, ak existuje príkaz s, používajúci x a cesta v grafe toku riadenia z p do s taká, že hodnota x sa počas nej nezmení.
- Premenná, ktorá má ďalšie použitie je živá.
- Premenná, ktorá nie je živá, je mŕtva.
- „Živost“ je o tom, či sa musíme ešte zaujímať o hodnotu danej premennej.

Algoritmus pre výpočet next use v bloku

Vstup: základný blok B, tabuľka symbolov

Výstup: pre každý priradovací príkaz i . $x := \text{op}(x_1, \dots, x_k)$
v bloku B informácia o živosti premenných.

Metóda:

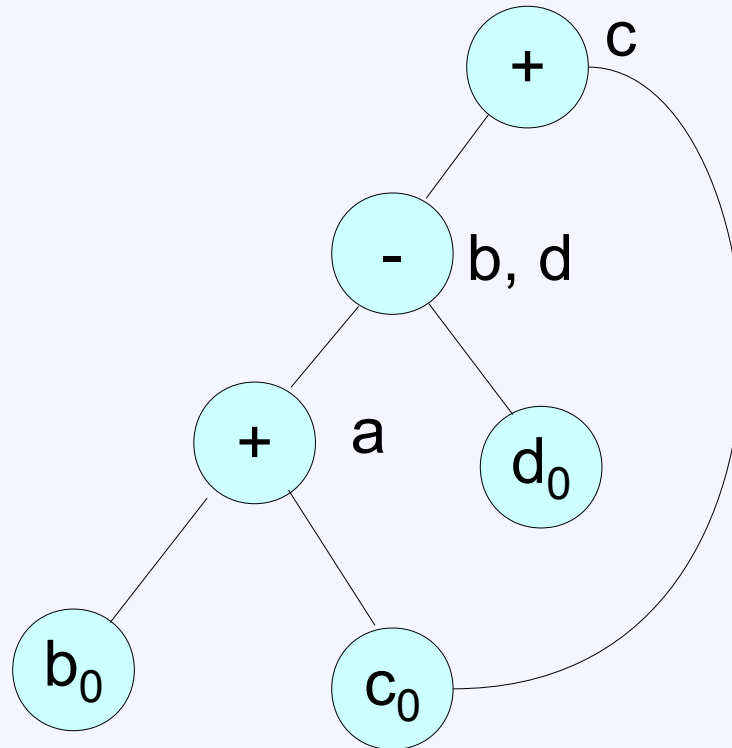
- Všetky nie dočasné premenné (nontemporary) označíme na výstupe z bloku B ako živé.
- for(i = last statement of B, $i \geq$ first statement of B, $i--$)
{ Ak i je tvaru $x := \text{op}(x_1, \dots, x_k)$:
Pre i zober next use x, x_1, \dots, x_k , z tabuľky symbolov;
V tabuľke symbolov polož next use a live x false;
V tabuľke symbolov polož next use a live x_1, \dots, x_k , true
}

Dag základného bloku

- Pravidlá vytvorenia dagu:
 1. Pre každú pôvodnú hodnotu premennej v bloku vytvoríme uzol - list.
 2. Pre každý príkaz s bloku B vytvoríme uzol N . Synovia uzlu N sú vrcholy zodpovedajúce poslednej definícii operandov N predchádzajúcej príkaz s .
 3. Uzol N označíme operáciou príkazu s a zoznamom premenných pre ktoré je poslednou definíciou v bloku.
 4. Niektoré uzly označíme ako výstupné uzly sú to tie uzly, ktorých premenné sú živé po výstupe z bloku. Výpočet živých premenných je predmetom **globálnej analýzy toku dát**.
- Použitie dagu
 - Eliminácia lokálnych spoločných podvýrazov
 - Eliminácia mŕtveho kódu
 - Preusporiadanie nezávislých príkazov
 - Využitie algebraických zákonov – preusporiadanie operandov

Príklad – eliminácia spoločných podvýrazov

$a := b + c$
 $b := a - d$
 $c := b + c$
 $d := a - d$

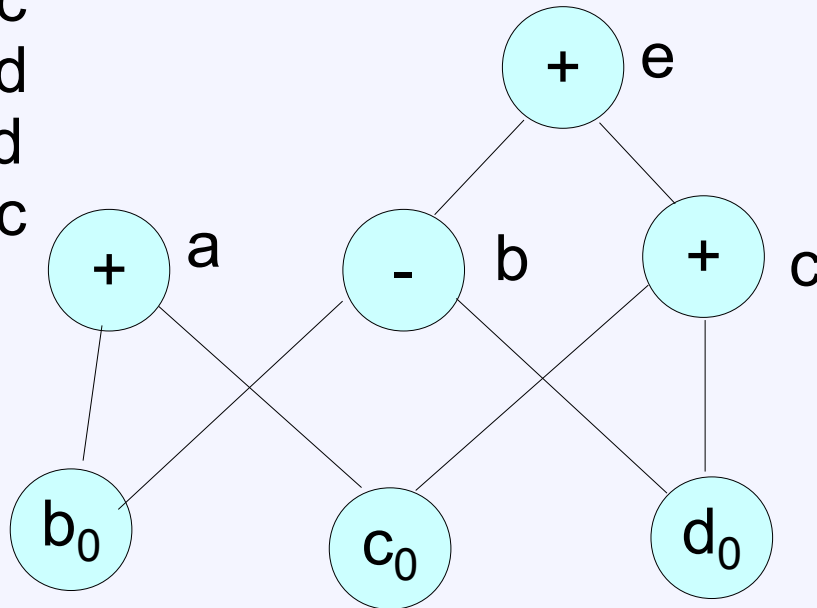


$a := b + c$
 $b := a - d$
 $d := b$
 $c := b + c$

Až globálna analýza toku dát zistí, či premenná d je naozaj zbytočná.

Príklad – eliminácia mŕtveho kódu

a := b + c
b := b - d
c := c + d
e := b + c



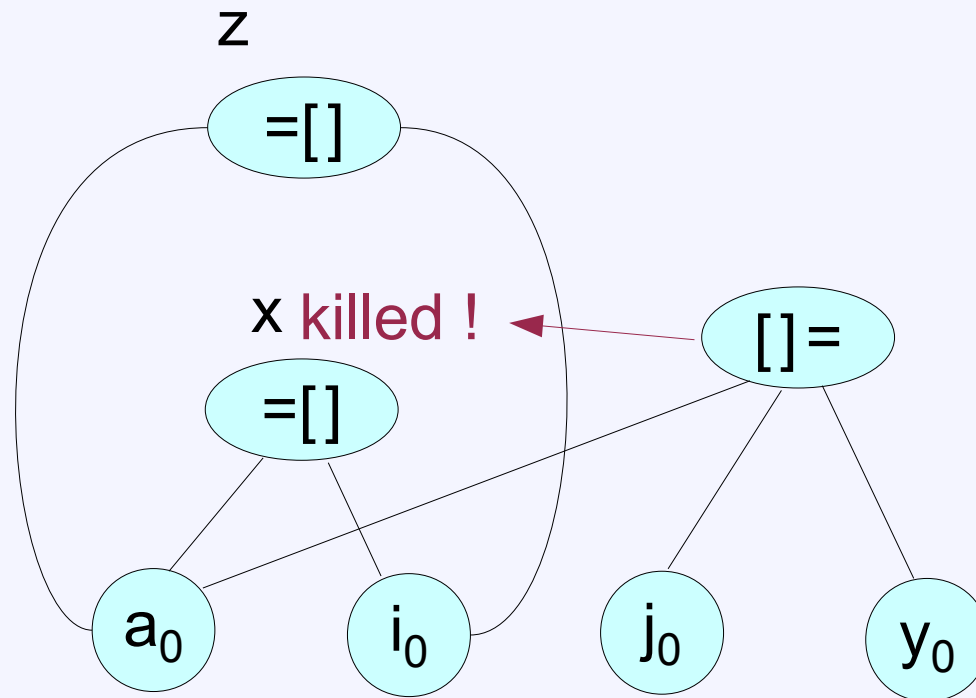
Pozn.: $e = a$
Posledné priradenie je zbytočné, lebo
 $b + c = (b - d) + (c + d)$.
Tento fakt eliminácia spoločných podvýrazov neodhalí.
Využitie algebraických zákonov ?

Ak a , b sú živé a c , e nemajú následné použitie, je ich výpočet zbytočný. Po eliminácii mŕtveho kódu:

a := b + c
b := b - d

Polia vyžadujú opatrnosť

```
x := a[i]  
a[j] := y  
z := a[i]
```



Nemôžeme využiť x pre výpočet z , lebo nevieme, či náhodou nie je $i = j$.

Podobne: smerníky. Špeciálne $*x$ = „vraždí“ všetky pred týmto operátorom urobené priradenia.

Poradie výpočtu

- Poradie výpočtu musí rešpektovať usporiadanie v dagu. Vyhodnotenie synov musí predchádzať vyhodnotenie otca.
- Polia smerniky volania procedúr musia byť vyhodnotené v takom poradí, ako v pôvodnom neoptimalizovanom bloku.

Algebraické zákony

- Identity
 - $x + 0 = 0 + x = x$; $x - 0 = x$
 - $1 * x = x * 1 = x$; $x / 1 = x$
- Redukcia v sile
 - $x^2 = x * x$
 - $2 * x = x + x$
 - $x / 2 = 0.5 * x$
- Skladanie konštánt (constant folding) počas kompilácie.
- Komutatívne zákony
 - Zdanlivo nič neoptimalizujú, ale keď jeden operand je už v registri môžeme inštrukciu LD ušetriť.

Význam registrov

- Pre väčšinu architektúr aspoň jeden operand (niekedy aj všetky operandy) musia byť v registroch, aby sa operácia dala vykonať.
- Registre sú vhodné miesto pre uchovanie medzivýsledkov a dočasných premenných v rámci bloku.
- Registre sú vhodné miesto pre niektoré globálne premenné (premenná cyklu, indexy do polí) používané vo viacerých blokoch.
- Registre sa používajú na údržbu dátových štruktúr pre podporu počas behu. Napr. smerník na vrchol zásobníka a rôzne pomocné smerníky.

Dátové štruktúry pre prácu s registrami

- Register descriptor
 - Pre každý (dostupný) register eviduje premennú, ktorej hodnota je v registri.
 - Môžeme predpokladať (pokiaľ evidujeme len registre) pre daný základný blok, že na počiatku je register descriptor prázdny.
- Address descriptor
 - Pre každú premennú programu udržiava miesto (miestá), kde sa okamžitá hodnota danej premennej nachádza.
 - Miesto môže byť register, adresa v pamäti, miesto v zásobníku, v halde a pod.
 - Vhodné miesto pre udržiavanie address deskriptoru je príslušná položka tabuľky symbolov.
- Proces generovania kódu je sprevádzaný údržbou týchto štruktúr

Algoritmus generovania kódu

- Predpokladáme, že máme postupnosť inštrukcií v medzijazyku (pregenerovaných pomocou dagu).

```
for (i=leaderB; i < nextleader; i++)
```

```
  { LR = GetReg(instruction[i]); // vráti registre Rx, Ry, Rz  
    CodeGen(instruction[i], LR); // vygeneruje strojový kód  
    Update(register_descriptor);  
    Update(address_descriptor)  
  }
```

- Procedúry GetReG a CodeGen využívajú informácie z register descriptoru a address descriptoru. Sú navrhnuté tak, aby negenerovali zbytočné presuny medzi registrami.

Príklad

$t := a - b$
 $u := a - c$
 $v := t + u$
 $a := d$
 $d := v + u$

3-ADDRESS INSTRUCTION	MACHINE CODE	REGISTER DESCRIPTOR	ADDRESS DESCRIPTOR
$t = a - b$	LD R1, a LD R2, b SUB R2, R1, R2	$R1$ has $\{a\}$ $R2$ has $\{b\}$ $R2$ has $\{t\}$	a in $\{a, R1\}$ a in $\{b, R2\}$ b in $\{b\}$ t in $\{R2\}$
$u = a - c$	LD R3, c SUB R1, R1, R3	$R3$ has $\{c\}$ $R1$ has $\{u\}$	c in $\{c, R3\}$ a in $\{a\}$ u in $\{R1\}$
$v = t + u$	ADD R3, R2, R1	$R3$ has $\{v\}$	c in $\{c\}$ v in $\{R3\}$
$a = d$	LD R2, d	$R2$ has $\{d\}$ $R2$ has $\{a, d\}$	d in $\{d, R2\}$ a in $\{R2\}$
$d = v + u$	ADD R1, R3, R1	$R1$ has $\{d\}$	d in $\{R1\}$
end	ST a, R2 ST d, R1		a in $\{a, R2\}$ d in $\{d, R1\}$

Stratégia pre GetReg($x := y \text{ op } z$) 1

- Najprv potrebujeme registre pre operandy y , z
 1. Ak y je už v registri vráť tento register ako R_y .
 2. y nie je v registri, ale existuje register, ktorý obsahuje mŕtvu premennú vráť tento register.
 3. Nie (1) ani (2), ale je k dispozícii prázdny register, vráť tento register.
 4. Ani prázdny register nie je k dispozícii. Ak x je v registri a x nie je druhý operand vráť tento register.
 5. Nič z predošlých bodov, najdi vhodného kandidáta register R . Register R obsahuje premennú v .
 - Ak address descriptor pre v hovorí, že premenná v je uložená aj inde, môžeš vrátiť register R .
 - R je jediné miesto pre premennú v potom „spill“ generuj inštrukciu $ST R v$, updatuj address descriptor a vráť register R .
- Stratégia pre druhý operand je rovnaká.

Stratégia pre GetReg($x := y \text{ op } z$) 2

- Register pre výsledok R_x
 1. Pretože sa počíta nová hodnota x register obsahujúci x je najvhodnejší.
 2. Ak neexistuje a aspoň jeden z operandov je mŕtvy alebo nemá v bloku next use a je uložený aj inde je register operandu vhodný.
 3. Ďalej môžeme skúsiť mŕtvú premennú a prázdny register
 4. Pokračujeme ako pri hľadaní registrov pre operandy.
- Stratégia pre GetReg($x := y$)
 - Najdeme register R_y pre y ako v predošlom prípade.
 - Register pre výsledok bude vždy ten istý $R_x = R_y$.

Globalizácia problému registrov

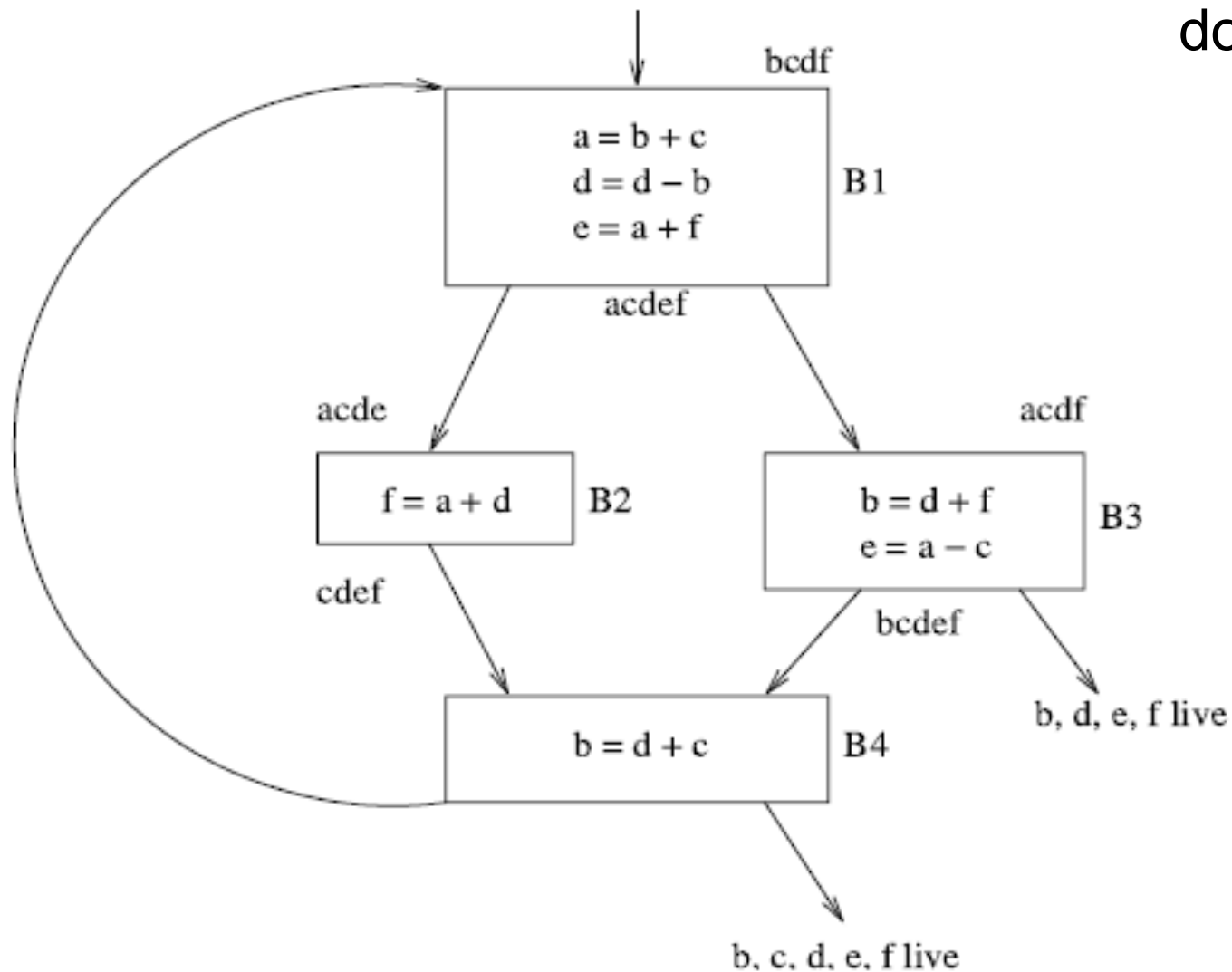
- Pri lokálnom pridelovaní registrov sa na konci bloku premenné zapamätajú do pamäti.
- Mnohé premenné a cykly prechádzajú cez viac blokov.
- Odhad prínosu premennej v registri počas vnútorného cyklu
- Predpokladáme, že ak premenná x je počas cyklu L v registri, pri každom jej použití ušetríme jeden pamäťový cyklus.
- Celkový zisk pre prípad, že premenná x je daný vzorcom:

$$\sum_{B \in L} (use(x, B) + 2 * live(x, B))$$

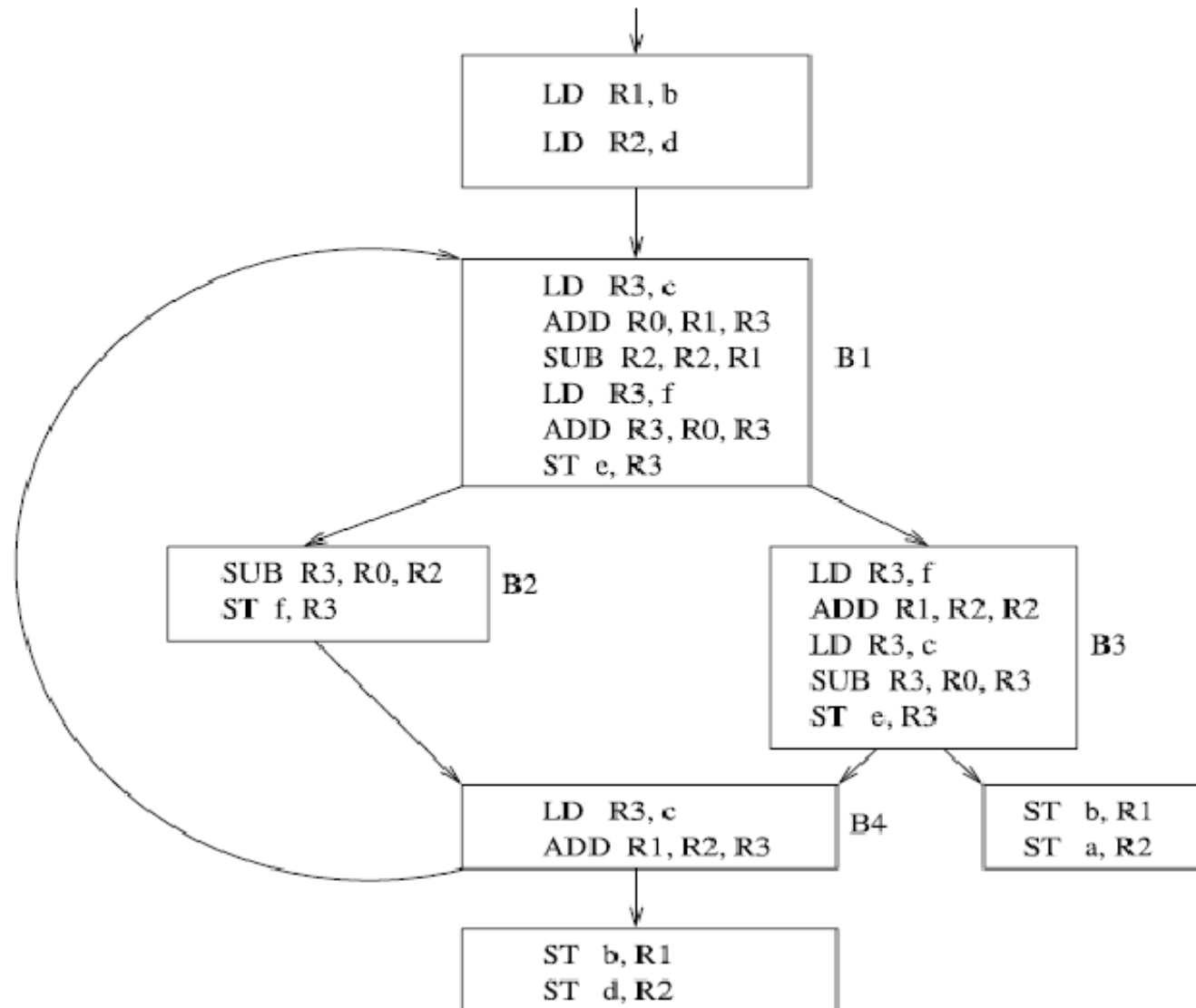
- Je to približné vyjadrenie, lebo nie všetky bloky sa vykonávajú v cykle rovnaký počet krát.

Príklad – medzijazyk

Vyberieme a, b
do registrov



Príklad generovaný kód



Register inference graph – RIG

- RIG je graf (inak, variable inference graph)
 - ktorého uzly sú registre.
 - Hrana medzi dvoma uzlami, ak premenné v oboch uzloch sú súčasne živé.
 - Hranu konštruujeme, keď v čase definície premennej druhá je živá.
- Vlastnosti RIG
 - Chromatické číslo RIG určuje minimálny počet registrov pre implementáciu bez operácie „spill“.
 - Skoro reálna úloha: Daný je graf RIG a číslo k (počet registrov) zistiť, či sa RIG dá zafarbiť k farbami (vrcholové farbenie).
 - Obe úlohy sú NP-úplné.

Heuristika

- Skoro reálnu úlohu môžeme riešiť nasledovnou heuristikou
 1. V grafe G nájdí vrchol n , ktorý má menej ako k susedov.
 2. Ak takýto vrchol neexistuje, graf sa nedá zafarbiť k farbami.
 3. Ak existuje, odstráň vrchol n a hrany s ním incidentné. Dostaneš graf G' . Zafarbenie G' k farbami sa dá triviálne rozšíriť na G .
 4. $G := G'$, ak G nie je prázdny graf pokračuj bodom 1.
 5. Ak si sa dostal sem, a G je prázdne, graf sa dá zafarbiť k farbami.
 6. Postupným pripájaním vrcholov v opačnom poradí ako boli odstraňované zafarbíš pôvodný graf.
- Ak sa graf nedá zafarbiť treba spill
 - Všeobecné pravidlo vyhýbať sa spillu v najvnútornejších cykloch.
- Realizmus:
 - Ak sa zafarbenie k farbami podarí. O.K.
 - Ak nie, ja najlepšie viem, ktoré premenné majú byť v registroch.

Jeršovové čísla – iné stromy výrazov

- Strom výrazu obsahuje len uzly pre premenné
 - výsledok (a operácia) je otec
 - operandy sú synovia
- Jeršovové číslovanie uzlov stromu výrazu
 - Listy majú číslo 1.
 - Vnútorňý uzol, ktorý má len jedneho syna, má také isté číslo ako jeho syn.
 - Vnútorňý uzol s dvoma synami má za číslo, väčšie z čísiel jeho synov.
 - Ak obaja synovia majú rovnaké číslo, potom má číslo od tohto čísla o 1 väčšie.
- Jeršovové číslo koreňa určuje počet registrov potrebný na vyhodnotenie výrazu bez zapamätania medzivýsledkov pri zachovaní štruktúry zatvoriek.

Príklad

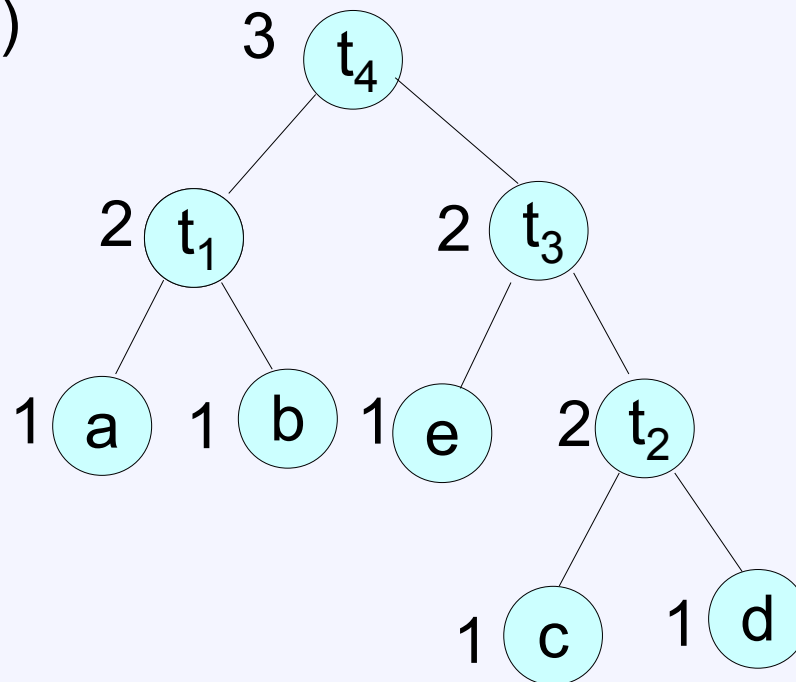
$(a - b) + e*(c + d)$

$t_1 := a - b$

$t_2 := c + d$

$t_3 := e * t_2$

$t_4 := t_1 + t_3$



```
LD R3, d
LD R2, c
ADD R3, R2, R3
LD R2, e
MUL R3, R2, R3
LD R2, b
LD R1, a
SUB R2, R1, R2
ADD R3, R2, R3
```

Generuje to vlastne dvojadresový kód.

„Dračí“ algoritmus generovania kódu je prekomplikovaný.

Jeden register sa dá ušetriť, ak sa poruší pravidlo o zachovaní zátvoriek.

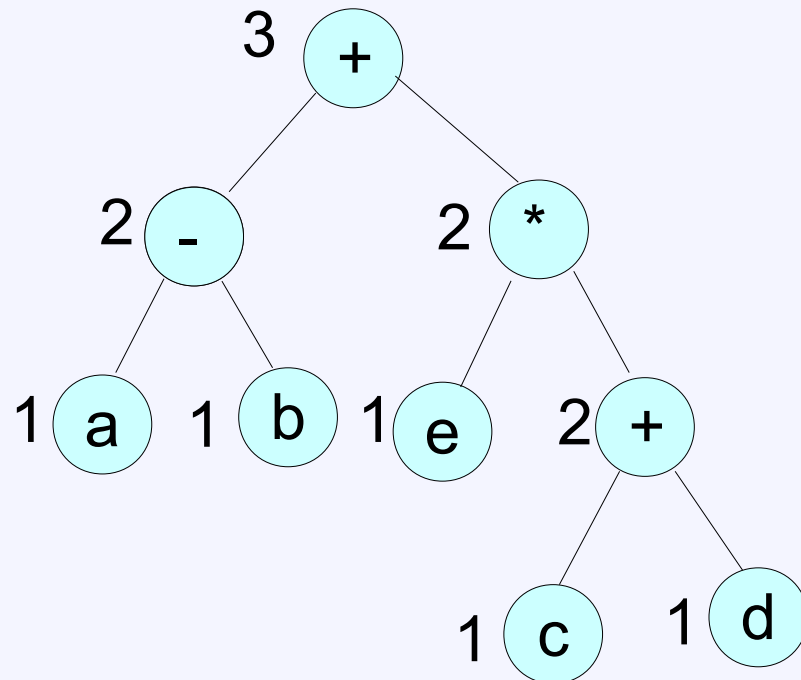
Algoritmus generovania kódu

Jeršovovými číslami riadený postorder traversing.

Ak sú čísla synov rovnaké, traverzuj v poradí ľavý syn, pravý syn.

Inak traverzuj najprv syna s väčším číslom.

Po vyhodnotení uzla si ponecháme výsledok v jednom registri.



```
LD R1, a
LD R2, b
SUB R1, R1, R2
LD R2, c
LD R3, d
ADD R2, R2, R3
LD R3, e
MUL R2, R3, R2
ADD R1, R1, R2
```

Cez klúčovú dierku, peephole optimization

- Princíp
 - malé okienko, vidí len malý úsek kódu, lokálne optimalizácie
- Čo optimalizujeme
 - Eliminácia nedodostupného kódu (blok bezprostredne za príkazom skoku na hlavičku, ktorého nevedie žiaden skok)
 - Optimalizácia toku riadenia
 - Skok na skok
 - Nešikovné podmienky
 - Zbytočné LD a ST inštrukcie (klasika)
 - Algebraické zákony (často je lepšie aplikovať ich už na medzijazyku)
 - Využitie „machine idioms“

Príklady

```
    if debug == 1 goto L1
    goto L2
L1:  printf(debug information)
L2:
```

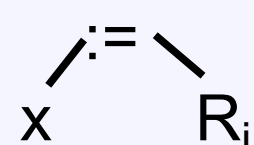
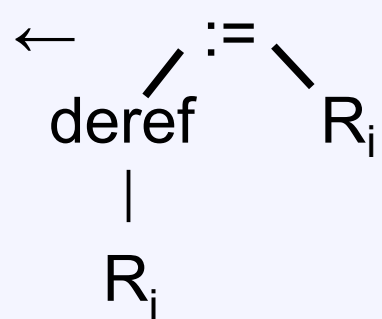
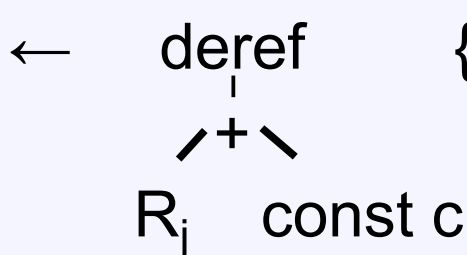
Po optimalizácii

```
    if debug != 1 goto L2
    printf(debug information)
L2:
```


Generovanie kódu pokrývaním – tiling

- Myšlienka
 - daný je program v medzijazyku „ozdobený syntaktický strom“ alebo strom výrazu.
 - a dané sú „kachličky“ – inštrukcie resp. skupiny inštrukcií.
 - kachličky majú svoju cenu
- Skupina inštrukcií (inštrukcia) pokrýva nejaký vzor (súvislú časť) kódu v medzijazyku.
- Cieľom je (najlacnejšie) pokrytie programu v medzijazyku
- Metóda
 - opakujeme nasledujúce tri kroky:
 - Nájdi vzor
 - nahraď ho výsledkom (obvykle register obsahujúci výsledok)
 - umiestni kachličku a zväčš cenu o cenu kachličky
 - pokiaľ celý strom nie je pokrytý

Niektoré vzory a kachličky 1

výsledok	vzor	kachlička	štvorice
R_i	$\leftarrow \text{const } a$	$\{ \text{LD } R_i \#a \}$	$\langle :=, \text{const } a, \text{nil}, R_i \rangle$
R_i	$\leftarrow x$	$\{ \text{LD } R_i \text{ref } x \}$	$\langle :=, x, \text{nil}, R_i \rangle$
M	\leftarrow 	$\{ \text{ST } R_i \text{ref } x \}$	$\langle :=, R_i, \text{nil}, x \rangle$
M	\leftarrow 	$\{ \text{ST } R_i (R_j) \}$	$\{ \langle :=, R_j, \text{nil}, t \rangle, \langle :=, R_i, \text{nil}, 0[t] \rangle \}$
R_i	\leftarrow 	$\{ \text{ST } R_i (R_j + \#c) \}$	$\{ \langle +, R_j, \text{const } c, t \rangle, \langle :=, t, \text{nil}, R_i \rangle \}$

Niektoré vzory a kachličky 2

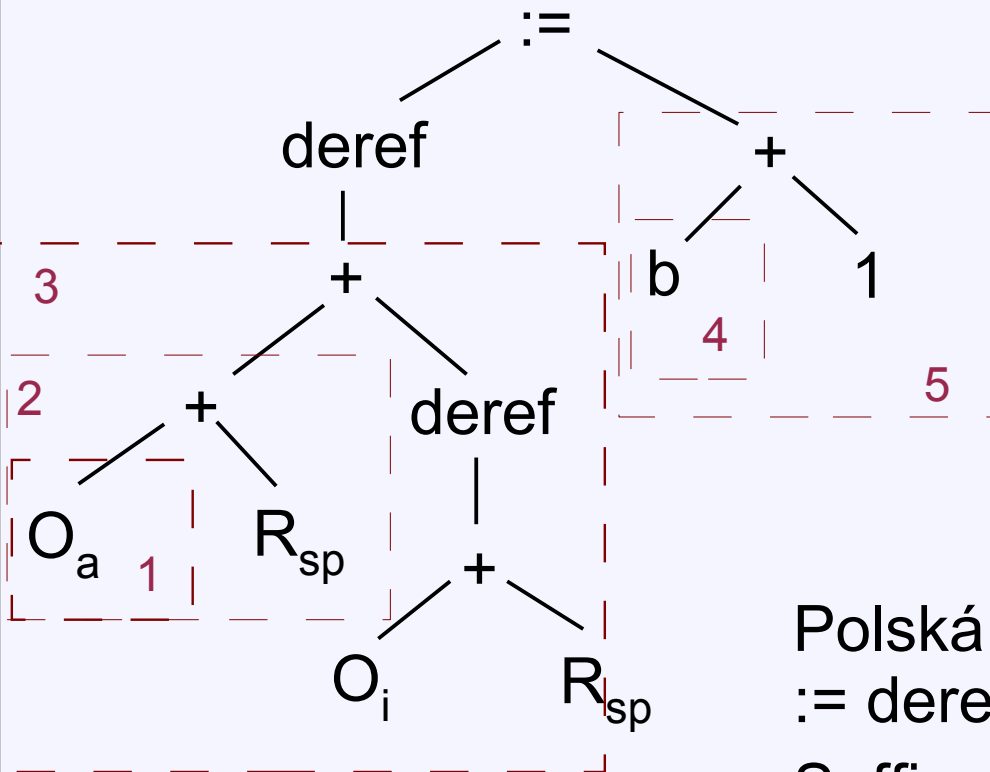
výsledok	vzor	kachlička	štvorice
R_i	$\begin{array}{c} \swarrow \quad + \quad \searrow \\ R_i \quad \quad R_j \end{array}$	$\{\text{ADD } R_i \ R_j \}$	$\langle +, R_i, R_j, R_i \rangle$
R_i	$\begin{array}{c} \swarrow \quad + \quad \searrow \\ R_i \quad \text{cons } 1 \end{array}$	$\{\text{INC } R_i \}$	$\langle +, R_i, \text{cons } 1, R_i \rangle$
R_i	$\begin{array}{c} \swarrow \quad + \quad \searrow \\ R_i \quad \text{deref} \\ \\ \begin{array}{c} \swarrow \quad + \quad \searrow \\ R_j \quad \text{cons } a \end{array} \end{array}$	$\{\text{ADD } R_i \ (R_j + a) \}$	$\{ \langle +, R_j, \text{cons } a, R_j \rangle, \\ \langle :=, 0[R_j], \text{nil}, t \rangle, \\ \langle +, R_i, t, R_i \rangle \}$

Pozn.: Veľké a zložité vzory sú dôsledkom zložitých inštrukcií a adresných módov CISC architektúry.

Príklad

$a[i] := b + 1$

1. LD R_0 O_a
2. ADD R_0 R_{sp}
3. ADD R_0 ($R_{sp} + O_i$)
4. LD R_1 b
5. INC R_1
6. ST R_1 (R_0)



Polská prefixová forma:

$:=$ deref + + O_a R_{sp} deref + O_i R_{sp} + b 1

Suffixová forma:

O_a R_{sp} + O_i R_{sp} + deref + deref b 1 + $:=$

Iný príklad

Source Code:

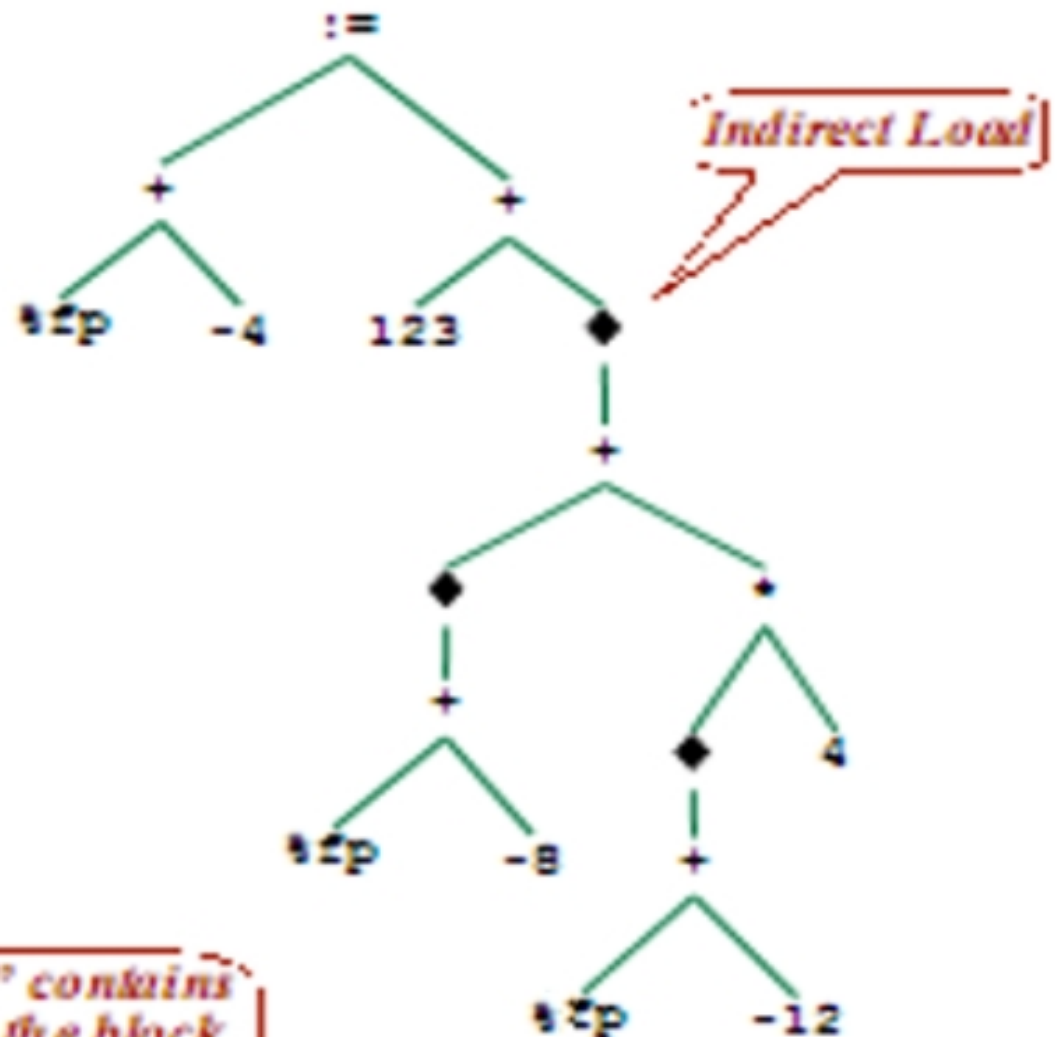
```
x := 123 * a[1];
```

IR Code:









```
t1 := %fp + -4  
t2 := %fp + -8  
t3 := *t2  
t4 := %fp + -12  
t5 := *t4  
t6 := t5 + 4  
t7 := t3 + t6  
t8 := *t7  
t9 := 123 * t8  
*t1 := t9
```

x	%fp-4
a	%fp-8
l	%fp-12

Variable "a" contains a pointer to the block of elements; Each element is 4 bytes



Vzory pokrývania

Pattern	Replacement	Code Template
<pre> + / \ reg num </pre>	 reg	ADD reg, num, reg
<pre> + / \ reg reg </pre>	 reg	ADD reg, reg, reg
<pre> * / \ reg reg </pre>	 reg	MUL reg, reg, reg
<pre> * + / \ reg num </pre>	 reg	LOAD [reg+num], reg
<pre> * reg </pre>	 reg	LOAD [reg], reg
<pre> num </pre>	 reg	SET num, reg
<pre> := / \ reg reg </pre>	 done	ST reg, [reg]
<pre> := / \ + reg / \ reg num </pre>	 done	ST reg, [reg+num]

Tiling the tree with maximal munch

- The simplest technique for tiling the tree is a greedy, top-down algorithm known as maximal munch.
- Tile the root with the biggest tile possible and then recurse on the children.
- This requires that it is possible to tile every type of node (otherwise, some tiles may be illegal).
 - For example, suppose that we can tile $OP(MEM(*), MEM(*))$, $OP(*, *)$, and $MEM(R(*))$, but not $R(*)$. (Okay, this is unlikely, but ...).
 - Then we don't want to use the biggest tile for the root of $OP(MEM(R(1)), MEM(R(2)))$.

Pokrývanie syntaxou riadeným prekladom

- Kachličkovanie nahradíme syntaxou riadeným prekladom
- Vstup lineárny zápis medzijazyka
- Gramatika lineárny zápis pravidiel nahradzovania vzoru
- „Drak“ doporučuje prefixovú notáciu a LALR metódu syntaktickej analýzy.
- Možno použiť aj suffixovú formu, ale potom sa uprednostňujú pravé operandy. Pri použití kontext senzitivnej analýzy, problém vyrieší vhodný kontext.

Príklad

- 1) $R_i \rightarrow x$ /* left context not := */ { LD R_i x }
 - 2) $M \rightarrow := x R_i$ { ST R_i x }
 - 3) $M \rightarrow := \text{deref } R_j R_i$ { ST R_i (R_j) }
 - 4) $R_i \rightarrow \text{deref } + R_j \text{ cons } a$ { LD R_i ($R_j + a$) }
 - 5) $R_i \rightarrow + R_i \text{ deref } + R_j \text{ cons } a$ { ADD R_i ($R_j + a$) }
 - 6) $R_i \rightarrow + R_i R_j$ { ADD R_i R_j }
 - 7) $R_i \rightarrow \text{cons } a$ { LD R_i $\#a$ } emit " ... " !
 - 8) $R_i \rightarrow + R_i \text{ cons } 1$ { INC R_i }
 - 9) $R_{sp} \rightarrow sp$
 - 10) $M \rightarrow x$ /* left context := */
- Gramatika je nejednoznačná, riešenie konfliktov.

Algoritmus najlepšieho pokrytia

- A top-down algorithm:
for each tile that covers the root
 compute the optimum tiling of each subtree of that tile
 compute the cost using those tilings and the current tile
pick the minimum of those tiles (Generuje kód v opačnom poradí.)
- Does this algorithm work? Yes. It tries all the tiles, which covers the root. This ensures that it puts the one that leads to the lowest total cost for the tree, too.
- Is this algorithm efficient? No. It tends to do repeated work, and can be exponential.
- Can it be done more efficient? Yes.
- **Dynamic programming.**

Optimum tiling – dynamic programming

- The dynamic programming algorithm proceeds bottom-up in four phases
 1. Compute bottom-up for each node N of the adorned parse T an array C of possible coverings and their costs. In the case of expressions $C[i]$ is cost of a minimal covering using i registers ($1 \leq i \leq r$ total number of registers).
 2. Traverse T , using the cost vectors to determine, which subtree of T must be stored into memory.
 3. Traverse the subtrees to be computed into memory and associated instructions using cost vectors to generate target code.
 4. Do the same as in step 3 for the main tree T to generate final target code.

Zložitosť

- Každá fáza sa dá realizovať v lineárnom čase.
- Optimálne pokrytie stromu je teda lineárny problém vzhľadom na veľkosť stromu
- Neeliminuje spoločné podvýrazy
- Algoritmus pokrývania a dynamického programovania sa dá realizovať aj na dagu. Zložitosť môže byť rovnaká ako na strome, ak cena zapamätania spoločnej časti je väčšia ako jej opätovný výpočet, alebo ak v optimálnom programe treba spoločnú časť počítat dvomi rôznymi spôsobmi.
- Strom môže byť až exponenciálne väčší ako dag.
- V praxi rozdiely nie sú až tak dramatické.