



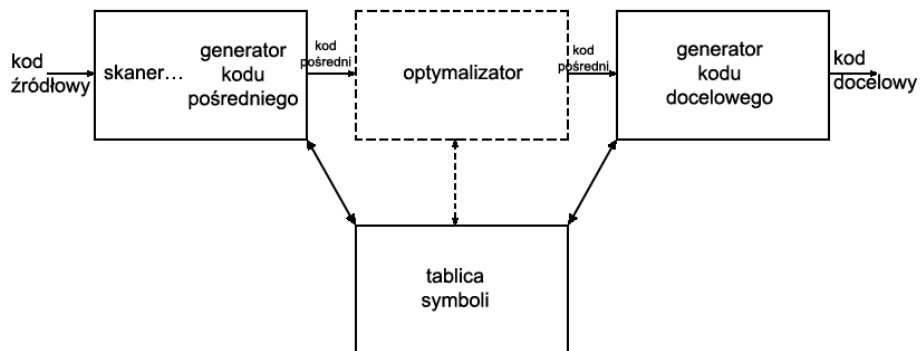
AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Generacja kodu ostatecznego

Dr inż. Janusz Majewski
Języki formalne i automaty



Generacja kodu ostatecznego





Generator kodu docelowego

We : - kod pośredni *niezależny sprzętowo*

Wy : - kod assemblera
- kod relokowalny
- kod absolutny } *zależny sprzętowo*



Rozkazy maszynowe

- Przyjmujemy następujący format rozkazu maszynowego:

operacja docelowy, źródłowy

Przykład:

add R0, R1 /* dodaj zawartość rejestru R1 do rejestru R0, wynik pozostaw w R0 */

- Rozkazy maszynowe są na ogół instrukcjami dwuadresowymi

a := a + 5

add a, 5



Rozkazy maszynowe

- Na ogół co najwyżej jeden argument rozkazu może być argumentem ulokowanym w pamięci

$a := a + b$

```
mov R0, b    /* załaduj b do rejestru R0 */  
add a, R0    /* dodaj zawartość R0 do a */
```

$a := b + c$

```
mov R0, b    /* załaduj b do rejestru R0 */  
add R0, c    /* dodaj c do zawartości R0 */  
mov a, R0    /* zapamiętaj zawartość R0 w a */
```



Generacja optymalnego kodu

- Problem generacji optymalnego kodu zależnego sprzętowo jest matematycznie nierozwiązywalny bądź też praktycznie niemożliwy do rozwiązania. Dlatego też w praktyce zadowolamy się technikami heurystycznymi dającymi dobry kod, ale nie zawsze optymalny.



Problemy:

– wybór rozkazów, minimalizacja kosztów (czasowych)

Gdybyśmy tłumaczyli każdą instrukcję trójadresową o postaci: $x := y + z$, gdzie x , y i z mają statycznie przydzieloną pamięć, na następujący kod:

```
mov R0, y      /* ładuj y do rejestru R0 */
add R0, z      /* dodaj z do R0 */
mov x, R0      /* zapamiętaj R0 w x */
```



Problemy:

... to następujące instrukcje:

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

zostałyby przetłumaczone na:

```
mov R0, b      /* ładuj b do rejestru R0 */
add R0, c      /* dodaj c do R0 */
mov a, R0      /* zapamiętaj R0 w a */
...
```



Problemy:

... to następujące instrukcje:

a := b + c

d := a + e

zostałyby przetłumaczone na:

```
mov R0, b      /* ładuj b do rejestru R0 */
add R0, c      /* dodaj c do R0 */
mov a, R0      /* zapamiętaj R0 w a */
mov R0, a      /* ładuj a do rejestru R0 */
add R0, e      /* dodaj e do R0 */
mov d, R0      /* zapamiętaj R0 w d */
```



Problemy:

... to następujące instrukcje:

a := b + c

d := a + e

zostałyby przetłumaczone na:

```
mov R0, b      /* ładuj b do rejestru R0 */
add R0, c      /* dodaj c do R0 */
mov a, R0      /* zapamiętaj R0 w a */
mov R0, a      /* ładuj a do rejestru R0 */ niepotrzebne
add R0, e      /* dodaj e do R0 */
mov d, R0      /* zapamiętaj R0 w d */
```



Problemy:

... to następujące instrukcje:

a := b + c

d := a + e

zostałyby przetłumaczone na:

mov R0, b /* ładuj b do rejestru R0 */

add R0, c /* dodaj c do R0 */

~~**mov a, R0** /* zapamiętaj R0 w a */~~ **niepotrzebne, jeśli a nie jest używane nigdzie dalej**

~~**mov R0, a** /* ładuj a do rejestru R0 */~~ **niepotrzebne**

add R0, e /* dodaj e do R0 */

mov d, R0 /* zapamiętaj R0 w d */



Problemy:

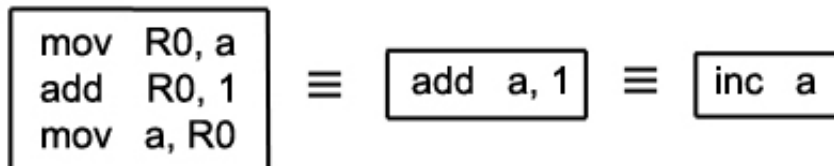
... widać więc, że jeśli nie interesujemy się efektywnością kodu wynikowego, to wybór rozkazów jest bardzo prosty. Jednakże naiwne tłumaczenie może prowadzić do poprawnego, acz niedopuszczalnie nieefektywnego kodu wynikowego.



Problemy:

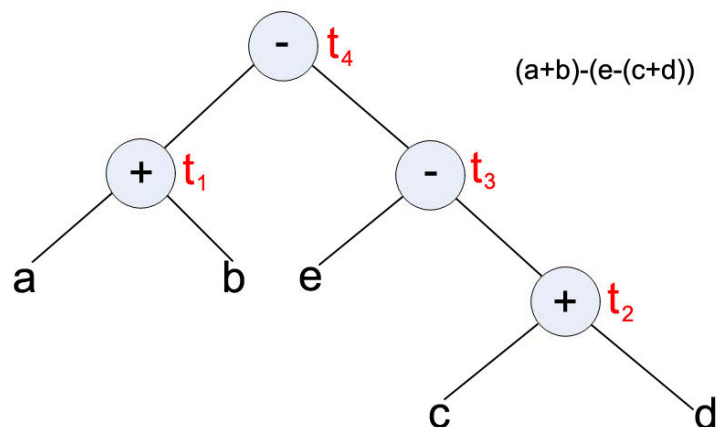
- wybór rozkazów, minimalizacja kosztów (czasowych)
- Maszyna docelowa z bogatym zbiorem rozkazów może dostarczać wielu metod implementacji danej operacji. Ponieważ różnice kosztu między implementacjami mogą być znaczące, więc naiwne tłumaczenie może prowadzić do nieefektywnego kodu wynikowego.

Przykład: $a := a + 1$



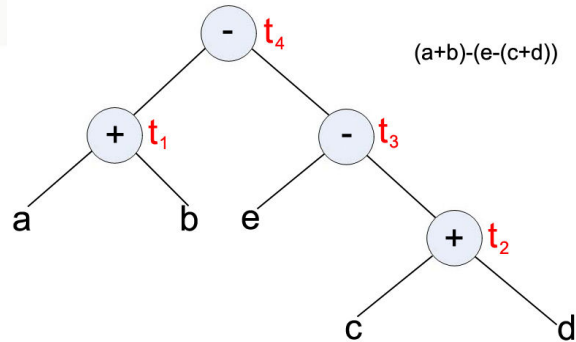
Problemy:

- wybór kolejności obliczeń





Wybór kolejności obliczeń



Kolejność naturalna:

$t_1 := a + b$
 $t_2 := c + d$
 $t_3 := e - t_2$
 $t_4 := t_1 - t_3$

Kolejność zmieniona:

$t_2 := c + d$
 $t_3 := e - t_2$
 $t_1 := a + b$
 $t_4 := t_1 - t_3$



Wybór kolejności obliczeń

Kolejność naturalna:

$t_1 := a + b$
 $t_2 := c + d$
 $t_3 := e - t_2$
 $t_4 := t_1 - t_3$

Kolejność zmieniona:

$t_2 := c + d$
 $t_3 := e - t_2$
 $t_1 := a + b$
 $t_4 := t_1 - t_3$

Założenie: dostępne są tylko dwa rejestry: R0 i R1

```
MOV R0, a
ADD R0, b /* w R0 jest t1 */
MOV R1, c
ADD R1, d /* w R1 jest t2 */
MOV t1, R0 /* odłożenie t1 do pamięci */
MOV R0, e
SUB R0, R1 /* w R0 jest t3 */
MOV R1, t1 /* zabranie t1 z pamięci */
SUB R1, R0
MOV t4, R1
```

```
MOV R0, c
ADD R0, d /* w R0 jest t2 */
MOV R1, e
SUB R1, R0 /* w R1 jest t3 */
MOV R0, a
ADD R0, b /* w R0 jest t1 */
SUB R0, R1
MOV t4, R0
zaoszczędzenie dwóch
rozkazów
```




Problemy

– „łatanie wstecz” dla skoków

Kod pośredni:

100: a := a+5

...

120: goto 100

Kod docelowy:

adres: add a, 5

...

jmp adres

Kod pośredni:

100: goto 120

...

120: a := a+5

Kod docelowy:

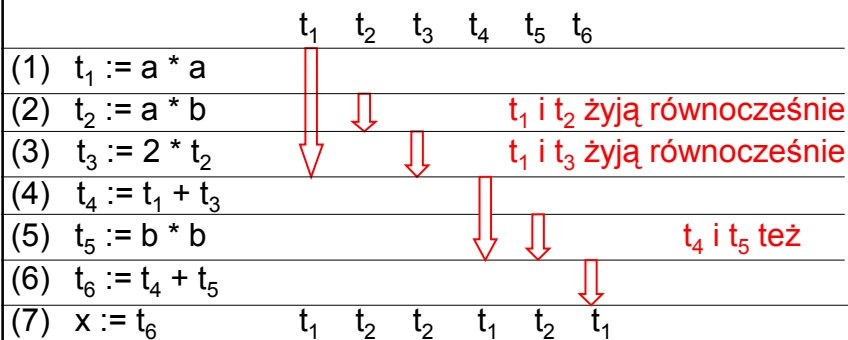
jmp

adres: add a, 5

← łatanie wstecz



„Życie” zmiennych tymczasowych



Na podstawie informacji o „życiu” można dokonać lokalizacji zmiennych tymczasowych według generalnej zasady, że dwie zmienne mogą zajmować tę samą lokalizację, gdy nie są równocześnie „żywe”



Redukcja zmiennych tymczasowych

Przed:

- (1) $t_1 := a * a$
- (2) $t_2 := a * b$
- (3) $t_3 := 2 * t_2$
- (4) $t_4 := t_1 + t_3$
- (5) $t_5 := b * b$
- (6) $t_6 := t_4 + t_5$
- (7) $x := t_6$

Po:

- (1) $t_1 := a * a$
- (2) $t_2 := a * b$
- (3) $t_2 := 2 * t_2$
- (4) $t_1 := t_1 + t_2$
- (5) $t_2 := b * b$
- (6) $t_1 := t_1 + t_2$
- (7) $x := t_1$