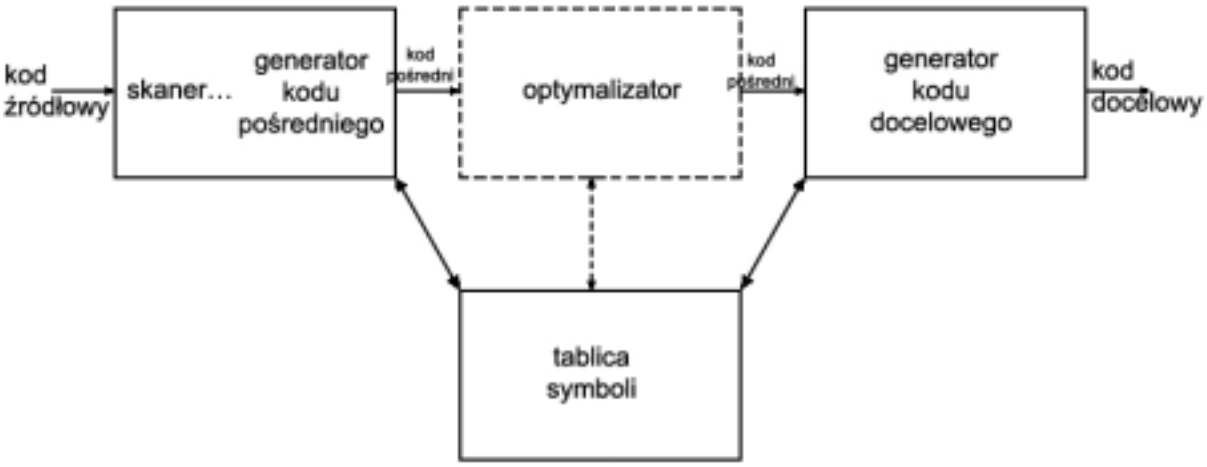
Generacja kodu :



Generator kodu docelowego:
We : kod pośredni *niezależny sprzętowo*

: kod assemblera
Wy : kod pośredni } *zależny sprzętowo*
: kod absolutny

Problem generacji optymalnego kodu zależnego sprzętowo jest matematycznie nierozwiązywalny bądź też praktycznie niemożliwy do rozwiązania. Dlatego też w praktyce zadowalamy się technikami heurystycznymi dającymi dobry kod, ale nie zawsze optymalny.
Problemy:
-przydział zasobów, efektywne wykorzystanie zasobów
-wybór instrukcji, minimalizacja kosztów (czasowych)

Przykład: a := a + 1

```
mov  R0, a
add  R0, 1   ≡   add  a, 1   ≡   inc  a
mov  a, R0
```

-zmiana kolejności obliczeń, itd.

Generowanie kodu dla bloków podstawowych

(a) Informacje nt. ”życia” i ”używania” zmiennych

(i)	x := y op z	(i) definiuje ”x”,
	: ┐ ”x” nie występuje	następne użycie ”x” w (j),
	: ┘	”x” żyje.
(j)	... := x op ...	(j) używa ”x” obliczonego w (i)
		odtąd ”x” może być martwy
		o ile nie będzie dalej używany

Badanie ”życia” i ”używania”:
Znajduje się koniec bloku podstawowego . Analiza grafu przepływu programu daje odpowiedź na pytanie: jaki jest stan zmiennych (”życie”) na końcu danego bloku . Informacje te zawarte są w tablicy symboli. Następnie przegląda się poszczególne zapisy w bloku podstawowym w kolejności odwrotnej (od końca do początku). Stosując poniższy algorytm można ustalić, czy w danym miejscu kodu trójadresowego zmienna żyje oraz gdzie będzie użyta. Jeżeli w trakcie przeszukiwania napotkano zapis:

(i) x := y op z

(1) Dla instrukcji (i) określamy stan zmiennych na podstawie aktualnej zawartości tablicy symboli.
(2) Wstawiamy do tablicy symboli dla ”x” (wynik) ”NIE ŻYJE” oraz ”BRAK NASTĘPNEGO UŻYCIA”
(3) Wstawiamy do tablicy symboli dla ”y” oraz ”z” (argumenty) „żyje” oraz „NASTĘPNE UŻYCIE w (i)”

Kolejność kroków (1) (2) (3) nie może być zmieniona , gdyż ta sama zmienna może być równocześnie argumentem oraz wynikiem.

Przykład:

	Wynik		Argument 1		Argument 2	
	życie	użycie	życie	użycie	życie	użycie
(1) t_1 := a * a	tak	(4)	tak	(2)	tak	(2)
(2) t_2 := a* b	tak	(3)	tak	⇩	tak	(5)
(3) t_3 := 2 * t_2	tak	(4)			nie	-
(4) t_4 := t_1 + t_3	tak	(6)	nie	-	nie	-
(5) t_5 := b * b	tak	(6)	tak	⇩	tak	⇩
(6) t_6 := t_4 + t_5	tak	(7)	nie	-	nie	-
(7) x := t_6	tak	⇩(dalej)	nie	-		

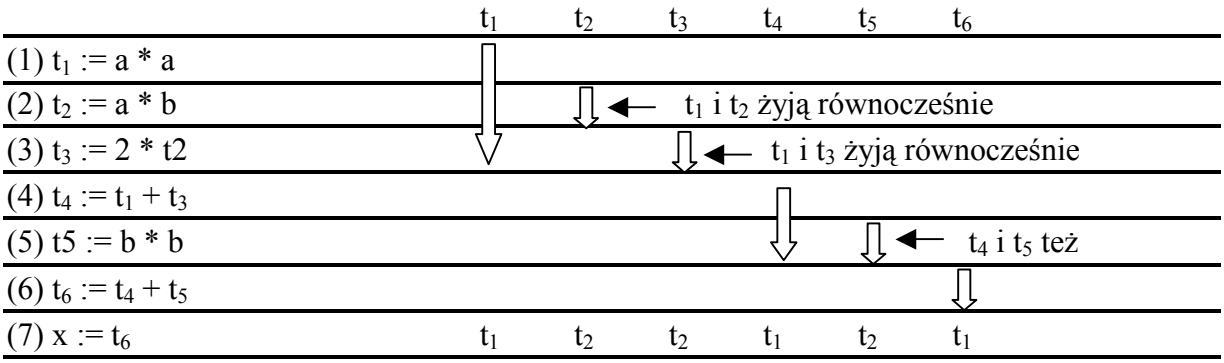
x := a * a + 2 * a * b + b * b

Tablica symboli

a ⇩		b ⇩		x ⇩		t_1		t_2		t_3		t_4		t_5		t6	
Ż	U	Ż	U	Ż	U	Ż	U	Ż	U	Ż	U	Ż	U	Ż	U	Ż	U
tak		tak		tak nie	-	nie	-	nie	-	nie	-	nie tak	- (6)	nie tak	- (6)	nie tak nie	- (7) -

0 stan początkowy
I ustalenie stanu zmiennych dla (7)
II zmiana tablicy symboli dla wyniku (7)
III zmiana tablicy symboli dla argumentu (7)
I ustalenie stanu zmiennych dla (6)
II zmiana tablicy symboli dla wyniku (6)
III zmiana tablicy symboli dla argumentów (6)
itd.

”Życie” zmiennych tymczasowych



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

Stąd:

(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) $x := t_1$

Przykładowy uproszczony algorytm generowania kodu: abdullahabdullah

Przyjmujemy generalne założenia :
- dla każdego operatora mamy odpowiadający mu rozkaz
  Format rozkazu :
  Operacja przeznaczenie , źródło
  Działanie :
  przeznaczenie ← przeznaczenie Operacja źródło
- rezultat obliczeń pozostaje w rejestrze tak długo, jak to możliwe
- rezultat jest przesyłany do pamięci, gdy
  (a) rejestr jest potrzebny dla innych obliczeń
  (b) przed każdym wywołaniem procedury, skokiem
  lub etykietowaną instrukcją, w szczególności
  przed końcem bloku podstawowego

Musimy mieć możliwość wykorzystywania następujących informacji:
- co jest aktualnie w każdym z rejestrów. Rejestr może być pusty , może też zawierać jedną lub więcej zmiennych (np.: po wykonaniu *x := y*) (deskryptor rejestru)
- gdzie jest aktualna wartość zmiennej. Wartość zmiennej może być równocześnie np. w pamięci i rejestrze (deskryptor adresu)

Algorytm: (szkic)

Dla każdej instrukcji kodu trójadresowego
x := y op z
wykonać:

1.Wywołać funkcję getreg dla określenia lokalizacji *L* wyniku *x*. Na ogół będzie to rejestr, ale może to być także pamięć.

2. Analizując deskryptor adresu wybrać lokalizację *y'* zmiennej *y*. Preferowany jest rejestr jeśli y jest równocześnie w rejestrze i pamięci . Jeśli wartość y nie jest w lokalizacji *L* przyszłego wyniku, to wygenerować
MOV L , y'

3.Analizując deskryptor adresu wybrać lokalizację *z'* zmiennej *z*. W razie istnienia wielu możliwości, wybrać rejestr.

4.Wygenerować *OP L , z'*

5.Zaktualizować deskryptor adresu dla *x* wskazując, że *x* znajduje się w *L*. Jeśli *L* jest rejestrem zaktualizować deskryptor tego rejestru.

6.Jeżeli *y* i/lub *z* nie mają następnego użycia i nie żyją na końcu bloku podstawowego, wówczas zaktualizować deskryptory odpowiednich rejestrów, tak aby wskazać, że rejestry te nie będą po wykonaniu *x:= y op z* zawierały *y* i/lub *z* odpowiednio.

Do tych zasad należy dołączyć dodatkowe ustalenia dla instrukcji z operacjami unarnymi, instrukcji kopiowania *x := y* (wtedy nie zawsze trzeba fizycznie przenosić dane, czasami wystarczy modyfikacja deskryptorów). Należy ponadto zachować w pamięci wartości tych zmiennych, które są żywe na końcu bloku podstawowego .

Funkcja getreg – zwraca lokalizację *L,* w której pozostawać ma wynik instrukcji *x := y* <u>*op*</u> *z*

1.Gdy :
- *y* jest w rejestrze i rejestr ten nie zawiera równocześnie innej wartości
- *y* nie jest żywe
- *y* nie jest używane dalej po wykonaniu *x:= y op z*

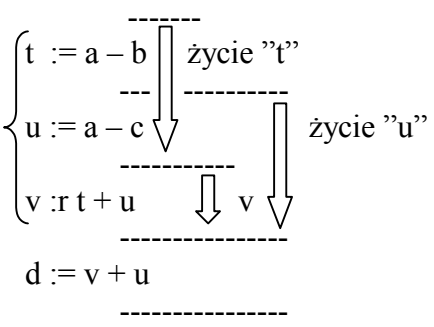
zwróć lokalizację *y*.

2. Gdy (1.) nie zachodzi zwróć pusty rejestr.
3. Gdy (2.) nie może być wykonane a *x* ma następne użycie w bloku, albo <u>*op*</u> jest operacją wymagającą rejestru – znajdź zajęty rejestr. Jeśli trzeba poprzednią zawartość rejestru odeślij do pamięci (jeśli zawartość ta jest tylko w rejestrze), zaktualizuj deskryptor adresu dla tej zawartości i zwróć wybrany rejestr.
4.Gdy *x* nie ma następnego użycia w bloku lub gdy nie da się znaleźć rejestru zajętego, zwróć lokalizację w pamięci.

Przykład:

d := (a - b) + (a - c) + (a – c)

kod trójadresowy (dla grafu)



t,u,v tymczasowe

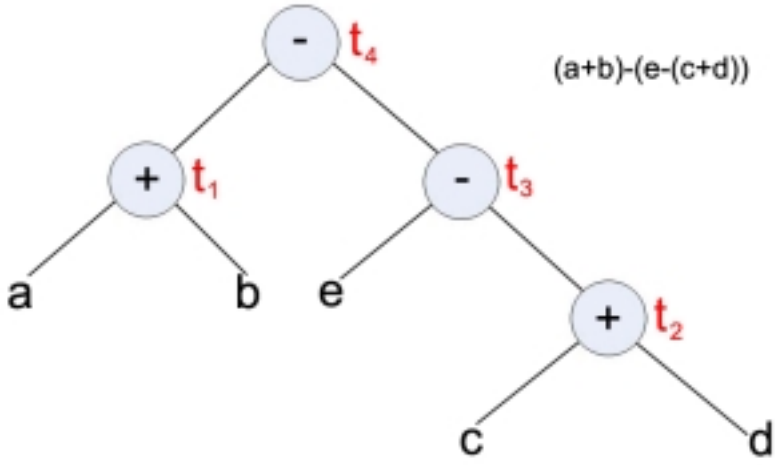
”d” żyje na końcu bloku

Założenie: dostępne dwa rejestry R0 i R1, oba puste na początku

Instrukcja	Kod	Deskryptor rejestru	Deskryptor adresu
		rejestry puste	a w pamięci b w pamięci c w pamięci
t := a – b	MOV R0 , a SUB R0 , b	R0 zawiera t	t w R0 a w pamięci b w pamięci c w pamięci

u := a – c	MOV R1 , a SUB R1,c	R1 zawiera u R0 zawiera t	u w R1 t w R0 a,b,c w pamięci
v := t + u	ADD R0 , R1	R1 zawiera u R0 zawiera v	u w R1 v w R0 a,b,c w pamięci
d := v + u	ADD R0,R1	R0 zawiera d R1 puste , bo ”u” tutaj martwe	d w R0 a,b,c w pamięci
koniec bloku	MOV d , R0 gdyż ”d” żyje	R0 zawiera d	d w R0 i w pamięci a,b,c w pamięci

Zmiana kolejności zapisów w bloku podstawowym



Kolejność naturalna:

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

Kolejność zmieniona :

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

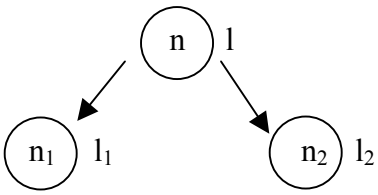
Kod wygenerowane według poprzedniego algorytmu, dostępne dwa rejestry R0 i R1

	MOV	R0 , a		MOV	R0 , c
	ADD	R0 , b		ADD	R0 , d
	MOV	R1, c		MOV	R1 , e
	ADD	R1 , d		SUB	R1 , R0
! →	MOV	t_1, R0 odłożenie t_1 do pamięci		MOV	R0 , a
	MOV	R0 , e		ADD	R0 , b
	SUB	R0 , R1		SUB	R0 , R1
! →	MOV	R1 , t_1 zabranie t_1 z pamięci		MOV	t_4 , R0
	SUB	R1 , R0		zaoszczędzenie dwóch	
	MOV	t_4 , R1		instrukcji	

Algorytm generacji kodu dla drzew syntaktycznych binarnych eliminujący niedogodność poprzedniego algorytmu.

(a) Etykietowanie drzewa (może być dokonywane równolegle z translacją boottom – up).

„Lewy wierzchołek” oznacza wierzchołek będący skrajnym lewym potomkiem przodka.

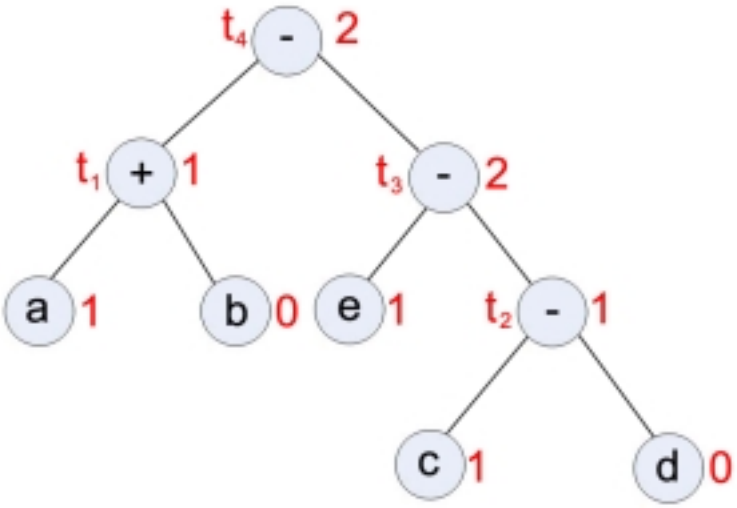


$$l = \underline{label}(n) = \begin{cases} \max(l_1, l_2) & \text{gdy } l_1 \neq l_2 \\ l_1 + 1 & \text{gdy } l_1 = l_2 \end{cases}$$

dla liści:
lewy skrajny liść poddrzewa $l \leftarrow 1$
pozostałe liście $l \leftarrow 0$

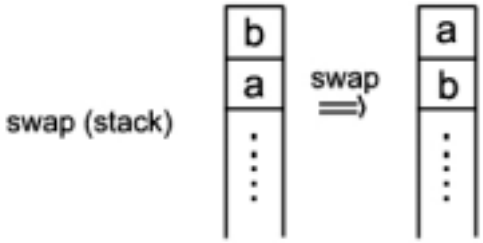
Algorytm etykietowania :
if n jest liściem then
if n jest skrajnym lewym potomkiem swojego przodka
then label (n) := 1
else label (n) := 0
else begin { węzeł wewnętrzny]
niech $n_1, n_2, ..., n_k$ będą potomstwem n uporządkowanym w/g etykiet , tzn. :
$label(n_1) \geq label(n_2) \geq ... \geq label(n_k)$;
$label(n) := \max(label(n_i) + i - 1)$;
$1 \leq i \leq k$
end;

Dla poprzedniego przykładu



Algorytm generacji:
We: etykietowane binarne drzewo syntaktyczne
Wy: kod maszynowy wyliczający wartość wyrażenia i pozostawiający ją w R0
Zał: drzewo jest binarne

Procedura rekursywna gencode(n) wywoływana dla korzenia. Wykorzystuje ona stos rstack dla alokacji rejestrów. Początkowo rstack zawiera rejestry R_0, R_1,...,R_{n-1} w takim porządku. Po wyjściu stos będzie zawierał rejestry w tym samym porządku.
Wykorzystuje się stos tstack dla alokacji w pamięci zmiennych tymczasowych. Początkowo stos zawiera t_0, t_1,...
Wykorzystuje się procedury i funkcje

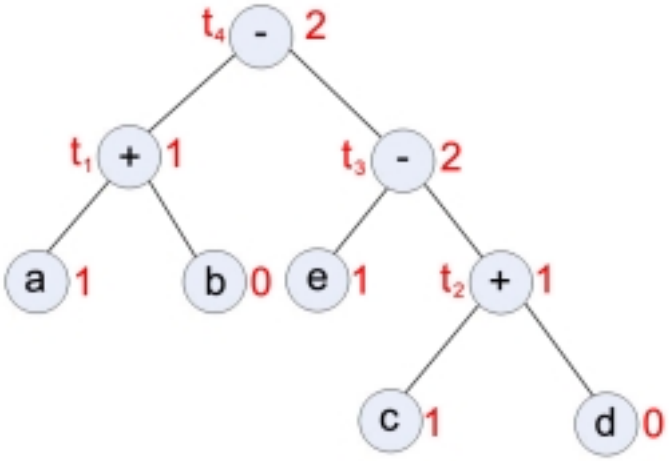


x := pop(stack); zdejmuje ze stosu
push (stack, x); składa na stosie
x := top(stack); zwraca element z wierzchołka stosu nie naruszając zawartości stosu

```
procedure gencode(n);
begin  /* przypadek 0 */
       if n jest lewym liściem reprezentującym operand ”name”
              and n jest skrajnym lewym potomkiem swego przodka
                     then print (‘MOV’ || top(rstack) || ’,’ || name)
       else if n jest wewnętrznym węzłem z operatorem op ,
              lewy potomek n1 , prawy n2 then
              /*  przypadek 1 */
           if label(n2) = 0 then begin
              ”name” jest operandem reprezentowanym przez n2;
              gencode(n1);
              print(OP || top(rstack) || ‘,’ || name);
              end
           /* przypadek 2 */
           else if 1 ≤ label (n1) < label (n2) and label (n1) < r
              then begin
                     swap(rstack);
                     gencode(n2);
                     R := pop(rstack); /* n2 było wyliczone w R */
                     gencode (n1)
                     print ( OP || top(rstack) || ‘,’ || R);
                     push (rstack , R);
                     swap(rstack);
              end
       / * przypadek 3 */
else if 1 ≤ label(n2) ≤ label(n1) and label(n2) < r
       then begin
              gencode(n1);
              R := pop(rstack); /* n1 zostało wyliczone w R */
              gencode (n2);
              print (OP || R || ‘,’ || top(rstack));
```

```
                push(rstack,R);
            end
        /* przypadek 4 */
else begin  /* obie etykiety ≥ r ( r – całkowita liczba rejestrów) */
        gencode(n2);
        T := pop(tstack);
        print ('MOV' || top(rstack) || ',' || T);
        gencode(n1);
        push(tstack,T);
        print(OP || top(rstack) || ',' || T);
    end
end;
```

Przykład: *(a + b) – (e – (c + d)*



	wierzchołek stosu	
gencode (t_4)	$[R_1\ R_0]$	/* 2 */
gencode(t_3)	$[R_0\ R_1]$	/* 3 */
gencode (e)	$[R_0\ R_1]$	/* 0 */
MOV R1 , e		
gencode (t_2)	$[R_0]$	/* 1 */
gencode(c)	$[R_0]$	/* 0 */
MOV R0 , c		
ADD R0 , d		
SUB R1 , R0		
gencode(t_1)	$[R_0]$	/* 1 */
gencode(a)	$[R_0]$	/* 0 */
MOV R0 , a		
ADD R0 , b		
SUB R0 , R1		

MOV t_1 , R2 (trzeba dołączyć)[T1]