Generarea de cod intermediar

Curs 13

Cod intermediar

- Compilatoarele – treceri multiple peste codul sursa
- Scopul:
  - Cod mai bun
  - Spatiu ocupat mai putin
  - Timp de executie mai mic
  - Generarea codului intermediar – interfata intre faza de analiza si sinteza a compilarii
  - Cod intermediar – reutilizabil – codul intermediar ar trebui sa fie cat mai generic
  - Calitatile unui limbaj intermediar:
    - Usor de transcris din arborele de sintaxa abstracta
    - Usor de translatat in cod masina

Limbaj intermediar

- Forme de reprezentare pentru limbajul intermediar:
  - Arbore atributat
  - Forma poloneza postfixata
  - **Cod cu trei adrese**
- Arbori atributati: cea mai intalnita reprezentare interna in compilatoare
- Avantaj:
  - arborii sunt construiti din faza de analiza sintactica
  - usor de parcurs
  - generarea de cod intermediar se suprapune cu faza de analiza semantica

Forma poloneza postfixata

- Expresia a+b*c
- F. p. infixata – operatorul este incadrat de cei doi operanzi
- F. p. prefixata – operatorul precede operanzii *+abc
- F. p. postfixata – operanzii preced operatorul abc*+
  - Reprezentare avantajoasa pentru limbajul intermediar:
    - modelul stivei – operanzii introdusi in stiva, operatiile se efectueaza in varful stivei (folosit de MSIL)
    - fpp nu contine paranteze si operatorii apar in odinea in care se executa (nu e nevoie de prioritatea operatorilor)
    - operatorii unari pot fi transcrisi prin operatori binari (-b=0-b)

Cod cu trei adrese

- **Codul cu trei adrese** este o secventa de instructiuni cu un format simplu, foarte apropiat de codul obiect, cu urmatoarea forma generala

  $\langle rezultat \rangle = \langle arg_1 \rangle \ \langle op \rangle \ \langle arg_2 \rangle$

unde:

- $\langle rezultat \rangle$, $\langle arg_1 \rangle$, $\langle arg_2 \rangle$ reprezinta **variabile** sau **constante** din program sau **nume temporare** create de compilator
- $\langle op \rangle$ este un **operator** binar

- Fiecare instructiune contine trei adrese: doua pentru operanzi (arg_1 si arg_2) si una pentru rezultat (rez)

Cod cu trei adrese

- Constituie o reprezentare liniara a arborelui de sintaxa abstracta
- Ordinea de executie a instructiunilor e mult mai clara
- Poate fi usor manipulat si rearanjat in faza de optimizare

Reguli de generare a codului cu trei adrese:

- Expresiile care contin operatori unari: tratate ca si cand primul operand lipseste

  <rezultat>=<op><arg2>

- Instructiunea de atribuire '=' joaca rolul atribuirii
  - a:=b se traduce prin *a=b* (lipsesc operatorul si al doilea operand)
- Salt neconditionat: instructiunea va avea forma ***goto L***, unde L este eticheta unei instructiuni din codul cu trei adrese
- Salt conditionat: instructiunea va avea forma *if c goto L* cu semnificatia: daca c este evaluat la true atunci se face salt neconditionat la eticheta L, altfel se executa instructiunea imediat urmatoare din codul cu trei adrese

Reguli de generare a codului cu trei adrese:

- **Apelul de procedura p(x_1,x_2,…,x_n)** va fi reprezentat ca:
  - Param x_1
  - Param x_2
  - …
  - Param x_n
  - Call p, n
  - cu semnificatia ca
    se evalueaza parametrii x_1,x_2,…,x_n prin instructiunea *param*, si apoi se apeleaza procedura p cu cei n parametri

- **Variabilele indexate**: <arg_1>, <arg_2>, <rezultat> pot fi elemente de tablou de forma a[i]

- e permisa o singura dimensiune
  - **a[i][j]** devine **a[i*(dim-1)+j]**, unde *dim* e prima dimensiune a tabloului bidimensional

Reprezentarea codului cu trei adrese

- Foloseste un **tablou de inregistrari**, fiecare inregistrare corespunde unei instructiuni
- Tipuri de inregistrari:
  - Cvadruple
  - Triplete
  - Triplete indirecte

Cvadruple

› Structura de inregistrare cu 4 campuri:

⟨op⟩ ⟨arg$_1$⟩ ⟨arg$_2$⟩ ⟨rezultat⟩

b*b-4*a*c

op	arg$_1$	arg$_2$	rez
*	b	b	T1
*	4	a	T2
*	T2	c	T3
-	T1	T3	T4

Cvadruple

› Pentru **salturile conditionate sau neconditionate eticheta de salt se trece la rezultat**

$$if\ (a < b)\ AND\ c\ then\ a := -1$$
$$else\ b := a + c$$

Nr.	op	arg1	arg2	rez
(1)	<	a	b	T1
(2)	AND	T1	c	T2
(3)	if	T2		(7)
(4)	+	a	c	T3
(5)	=	T3		b
(6)	goto			(9)
(7)	@	1		T4
(8)	=	T4		a
(9)				

Minus unar

ramura **else**

ramura **then**

Triplete

- Asemanatoare cvadruplelor, cu exceptia rezultatului – nu se reprezinta explicit ci se considera ca inregistrarea care calculeaza un rezultat il va si memora
- Structura cu trei campuri

  $\langle op \rangle \langle arg_1 \rangle \langle arg_2 \rangle$

Nr	op	arg_1	arg_2
(1)	*	b	b
(2)	*	4	a
(3)	*	(2)	c
(4)	-	(1)	(3)

Triplete

- Dezavantajul acestei reprezentari – instructiunile pot suferi modificari in faza de optimizare si acestea trebuie transmise si asupra referirilor la instructiunile respective – crearea unei liste suplimentare cu numarul de ordine al tripletelor in ordinea in care se executa – **triplete indirecte**

Comparatia reprezentarilor

	Cvadruple	Triplete
Avantaje	Referire explicita la rezultate si accesare imediata a TS; Algoritmi mai simpli in optimizarea codului	Dimensiune mai mica Evita supraincarcare in TS
Dezavantaje	Dimensiune mai mare Supraincarcare in TS	La optimizarea codului mutarea instructiunii implica modificarea referintelor la ea La generarea de cod e necesara scanarea tuturor tripletelor pentru a sti numarul de locatii temporare active

Optimizarea codului intermediar

- Colectie de transformari care se aplica in scopul obtinerii unui program mai eficient ca spatiu si timp
- Optimizarea codului intemediar este o etapa optionala, nu toate compilatoarele o implementeaza
- Operatiile de optimizare necesita timp – rezultatele optimizarii trebuie sa justifice timpul consumat
- Pasi:
  - Identificarea partilor din program care merita supuse optimizarii (in mod deosebit ciclurile)
  - Identificarea transformarilor care trebuie aplicate

Optimizarea codului intermediar

- **Principiul de baza**: orice transformare asupra codului intermediar trebuie sa ii conserve efectul, deci codul intermediar si cel optimizat trebuie sa fie semantic echivalente
- Componente in procesul de optimizare:
  - **Analiza fluxului de control**
  - Analiza fluxului de date
  - transformari

Graful de flux al unui program

- Graful de flux: nodurile sunt blocurile de baza, iar arcele indica transferul executiei de la un bloc la altul
- **Bloc de baza**: secventa de instructiuni consecutive in care fluxul de control intra in prima instructiune, iese la sfarsitul ultimei instructiuni, *fara posibilitate de salt sau ramificare*

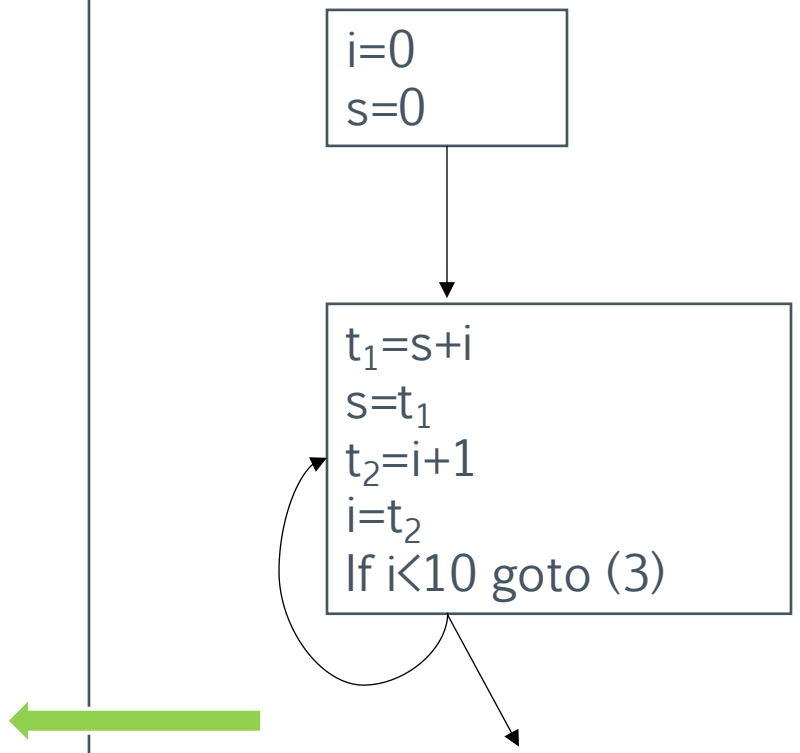
- Partitionarea in blocuri de baza – reguli:
  - Se determina **prima instructiune a unui bloc**:
    - Prima instructiune din program;
    - Orice instructiune care este rezultatul unui salt conditionat sau neconditionat;
    - Orice instructiune imediat urmatoare unui salt conditionat sau neconditionat

Graful de flux al unui program

- Pentru fiecare prima instructiune a unui bloc se construieste blocul de baza corespunzator care cuprinde: prima instructiune si toate instructiunile care urmeaza pana la urmatoarea prima instructiune (exclusiv aceasta)
- Arcele grafului se construiesc astfel: exista un arc intre blocurile B_i si B_j daca B_j urmeaza dupa B_i intr-o secventa de executie, adica:
  - Exista un salt conditionat sau neconditionat de la ultima instructiune a lui B_i la prima instructiune a lui B_j
  - B_j urmeaza imediat dupa B_i in ordinea secventiala din program si ultima instructiune a lui B_i nu este salt neconditionat

Exemplu

- (1) i=0
- (2) s=0
- (3) t_1=s+i
- (4) s=t_1
- (5) t_2=i+1
- (6) i=t_2
- (7) if i<10 goto(3)



Transformari

- Locale – la nivelul unui bloc
- Globale – restul
- Principalele actiuni de optimizare posibile sunt:
  - Realizarea unor calcule la compilare
  - Eliminarea calculelor redundante
  - Eliminarea codului inaccesibil
  - Optimizarea ciclurilor

Optimizari locale

- Realizarea unor calcule la compilare
- Expresii de tipul z=x+y, unde x si y sunt constante pot fi calculate la compilare
- Conversiile implicite apartin aceluiasi tip de optimizari

```
X=4+8;

Y=2*pi;
```

Optimizari locale

- Eliminarea calculelor redundante:
- Se bazeaza pe folosirea repetata a unor calcule din program
- D:=D+C*B
- A:=D+C*B
- C:=D+C*B

Optimizari locale

(1)	*	C	B
(2)	+	D	(1)
(3)	=	(2)	D
(4)	*	C	B
(5)	+	D	(4)
(6)	=	(5)	A
(7)	*	C	B
(8)	+	D	(7)
(9)	=	(8)	C

```
D:=D+C*B
A:=D+C*B
C:=D+C*B
```

Operatie redundanta

- A *i*-a operatie dintr-un bloc este **redundanta** daca exista o operatie anterioara identica (fie *j*) si daca niciunul din operanzii de care depinde operatia *i* nu se modifica de catre o a treia operatie cuprinsa intre *i* si *j*.

Optimizari locale

```
D:=D+C*B
A:=D+C*B
C:=D+C*B
```

(1)	*	C	B
(2)	+	D	(1)
(3)	=	(2)	D
(4)	*	C	B
(5)	+	D	(4)
(6)	=	(5)	A
(7)	*	C	B
(8)	+	D	(7)
(9)	=	(8)	C

!

Tripletele (4) si (7) pot fi inlocuite de (1)

Tripletele (5) si (8) nu pot fi inlocuite cu (2), deoarece intre (2) si (5) D isi schimba valoarea in tripletul (3)

Eliminarea codului inaccesibil

- Cod inaccesibil – secventa de program care nu se executa pe nici un drum din graful de control

- Astfel de cod poate sa apara in urma evaluarii unei conditii la o constanta si astfel una din ramuri nu se va executa niciodata

Optimizarea ciclurilor

- Actiunile principale de optimizare a ciclurilor sunt:
  - Factorizarea invariantilor de cicluri
  - Reducerea puterii operatiilor
- O instructiune sau o expresie este un **invariant de ciclu** daca fiecare operand este:
  - Constanta sau
  - Are toate definirile in afara ciclului sau
  - Are exact o definire si aceea este un invariant de ciclu

Exemplu

```
for (int i=0; i<n; i++){
        x=y+z;
        a[i]=i*x;
}
```

```
x=y+z;
for (int i=0; i<n; i++){
        a[i]=i*x;
}
```

Reducerea puterii operatiilor

- Are ca scop inlocuirea operatiilor costisitoare (inmultirea) cu operatii mai ieftine (adunarea) in definirea variabilelor de ciclare si a altor variabile calculate in cicluri
- Reducere puterii operatiilor pentru variabilele de ciclare se refera la cazul in care ele sunt indexate – transformarea se refera la calcularea relativa a elementului in sir
- Pentru celelalte calcule se tine cont de faptul ca valoarea calculata la pasul *i* depinde de valoarea calculata la pasul *i-1*

Exemplu

```
i =1;

while  (i<10)
       {
         y = i * 4;

         i=i+1;

        }
```

```
i = 1
t = 4
{
while( t<40) {
  y = t;
  t = t + 4;
}
```

Exemplu

```
for (i=k; i<=n; i++){
        t=i*v;
…}
```

```
t1=k*v;
for (i=k; i<=n; i++){
        t=t1;
        t1=t+v;
…}
```