LABORATOR NR. 8

GENERAREA ŞI OPTIMIZAREA CODULUI INTERMEDIAR

Programul rezultat după analiza sintactică şi semantică este o succesiune de operaţii (aritmetice, atribuiri, teste, salturi) împreună cu operanzii asociaţi. Codul intermediar se deosebeşte de limbajele de asamblare prin aceea că operanzii nu sunt regiştrii sau cuvinte de memorie, ci referinţe la intrări în tabela de simboluri. Structura secvenţei operaţiilor şi modul lor de reprezentare depind de soluţia adoptată pentru codul intermediar: forma poloneză, arbori sintactici, triplete, cuadruple.

Forma poloneză. O expresie în formă poloneză se defineşte astfel:

a-pentru orice operand *a*, *a* este expresie în formă poloneză

b-dacă $e_1, e_2, ..., e_n$ sunt în formă poloneză şi $*$ este o operaţie n-ară, atunci $e_1, e_2, ..., e_n *$ este expresie în formă poloneză

c-orice expresie formată altfel decât la *a* şi *b* nu este expresie în formă poloneză

Algoritm pentru aducerea unei expresii aritmetice la forma poloneză:
Intrare: o expresie aritmetică
Ieşire: forma poloneză (postfixată) a expresiei aritmetice
Metoda: se definesc priorităţi pentru operatori:

$ şi (au prioritatea 0
+ şi – au prioritatea 1
$*$ şi / au prioritatea 2

$ este un simbol special, iar pentru) nu este nevoie de prioritate. Algoritmul lucrează cu triplete (α, β, γ) unde α este şirul de intrare, β este stiva de lucru folosită pentru depozitarea operatorilor, iar γ este şirul de ieşire. Configuraţia iniţială este $(\alpha, \$, \varepsilon)$ şi cea finală este $(\varepsilon, \$, \overline{\alpha})$

P1. $(a\alpha, b\beta, \gamma) \vdash (\alpha, b\beta, \gamma a)$ dacă *a* este operand

$\vdash (\alpha, ab\beta, \gamma)$ dacă *a* este operator şi $P(a) > P(b)$ sau $a = ($

$\vdash (a\alpha, \beta, \gamma b)$ dacă *a* este operator şi $P(a) \le P(b)$

P2. $()\alpha, b\beta, \gamma) \vdash (\alpha, \beta, \gamma)$ dacă $b = ($

$\vdash ()\alpha, \beta, \gamma b)$ dacă $b \neq ($

P3. $(\varepsilon, b\beta, \gamma) \vdash (\varepsilon, \beta, \gamma b)$ dacă $b \neq \$$

Se notează $P(x)$ prioritatea operatorului *x*.

Algoritm de evaluare a unei expresii în formă poloneză

Intrare: o expresie în formă poloneză
Ieşire: valoarea expresiei
Metoda:

-se poziționează citirea pe primul simbol din şir $(i \leftarrow 1)$
-while nu s-a epuizat şirul do
Begin
-citeşte simbolul curent din şir, fie el $s(i)$
-if $s(i)$ este operator
then
begin
-fie *n* aritatea lui $s(i)$
-extrage ultimele n simboluri din; fie $t_n, t_{n-1}, ..., t_1$
-execută operația $s(i)$ $(t_n, t_{n-1}, ..., t_1)$ şi depune rezultatul în stivă
end
else
-semnalează eroare
$-i \leftarrow i+1$ (se citeşte următorul caracter din şir)

2. Arbori sintactici. Reprezentarea programului intermediar sub formă de arbore este mai apropiată de structura sintactică şi din această cauză reprezentarea nu aduce elemente noi. Totuşi ea este utilă în faza de optimizare. Nodurile interioare sunt etichetate cu operatori iar frunzele cu operanzi.Operațiile nu sunt numai aritmetice ci pot fi selectări, comparații, atribuiri, salturi, etc. Aritatea operatotului dă numărul de descendenți ai nodului etichetat.

3. Cod intermediar cu trei adrese – este văzut ca o secvență de instrucțiuni de tipul $A \leftarrow B \; op \; C$ unde A, B, C sunt identificatori, constante sau variabile temporare şi *op* este o operație aritmetică sau logică. Codul cu trei adrese poate fi implementat sub formă de triplete, triplete indirecte sau cuadruple.

1. **Triplete** – sunt reprezentate prin structuri cu trei câmpuri conținând operatorul şi cei doi operanzi. Cele trei câmpuri sunt pointeri către tabela de simboluri sau către structura tripletelor. Pointerii către structura tripletelor se vor reprezenta prin numere între paranteze rotunde. În faza de optimizare a codului au loc frecvent operații de suprimare sau deplasare a instrucțiunilor cu trei adrese. În caz de deplasare trebuie modificați toți pointerii către aceste instrucțiuni. Trebuie parcursă lista tripletelor şi

modificate toate cele care utilizează variabila temporară asociată tripletului deplasat. Pentru a economisi timp se preferă folosirea tripletelor indirecte. În acest caz structurii tripletelor i se asociază o listăde pointeri, care dau ordinea de execuţie a tripletelor. Când au loc modificări în ordinea de execuţie a tripletelor, este suficient să se reordoneze numai lista pointerilor.

2. **Cuadruple**.- sunt structuri cu patru câmpuri conţinând operatorul, cei doi operanzi şi rezultatul. Câmpurile corespunzătoare operanzilor şi rezultatului pointează către tabela de simboluri.
   Instrucţiunile cu trei adrese utilizate sunt:
   - instrucţiuni de atribuire:
     $A \leftarrow B\ op\ C$ cu *op* operator aritmetic sau logic binar;
     $A \leftarrow op\ B$ cu *op* operator aritmetic sau logic binar;
     $A \leftarrow B$
   - instrucţiuni de salt necondiţional: *goto q;*
   - instrucţiuni de salt condiţional: *if A oprel B goto q;* dacă relaţia este satisfăcută se execută instrucţiunea cu trei adrese etichetată cu *q*
   - apel de subprograme:
     -Param Pa – specifică faptul că Pa este parametru;
     -Call Pr,Np – specifică apelul procedurii Pr cu Np Parametri
   - atribuirea indexată: - $A \leftarrow B[i]\ si\ A[i] \leftarrow B$
   - atribuirea prin pointeri şi adrese: - $A \leftarrow *B \quad si \quad *A \leftarrow B$
     - $A \leftarrow Adr\ B$

Optimizarea codului- este o fază opţională şi are drept scop rearanjarea codului intermediar în vederea obţinerii unui program mai eficient din punct de vedere al memoriei dar mai ales al timpului de execuţie. Metodele folosite pentru optimizarea codului obiect sunt puternic dependente de maşină.

Optimizări simple :

-aplatizarea care constă în înlocuirea expresiilor ce pot fi evaluate în timpul compilării prin valorile lor, se poate ţine seama de unele identităţi algebrice ($X+0=X; 0+X=X; X*1=X; 0/X=0; X-0=X$), proprietăţi de asociativitate şi comutativitate ale unor operatori ($5+A+B+7$ se înlocuieşte cu $12+A+B$).

Exemplu: Instrucţiunea $A \leftarrow 2*3+5-3+B*C$ poate fi înlocuită cu $A \leftarrow 8+B*C$.

-propagarea constantelor constă în înlocuirea variabilelor prin valorile lor, dacă aceste valori sunt cunoscute la compilare. Înlocuirea variabilelor prin valorile lor trebuie făcută după o analiză globală a programului, pentru că altfel pot apare erori.

Optimizări globale. Cea mai mare parte a timpului este destinată optimizării buclelor. Pentru a determina diferitele bucle ale unui program se utilizează graful de flux. Graful de flux este un graf orientat ce defineşte relaţiile dintre blocurile de bază. Un bloc de bază este format din instrucţiuni consecutive ale programului şi sunt executate una după alta exact în ordinea în care apar în program. Două blocuri de bază nu pot avea instrucţiuni comune.
Determinarea primelor instrucţiuni se face astfel:

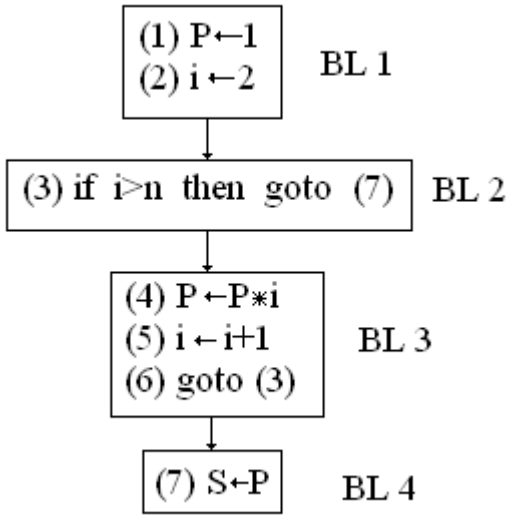
-prima instrucţiune din program este primă instrucţiune a unui bloc
-o instrucţiune ce urmează unei instrucţiuni de transfer este o primă instrucţiune
-o instrucţiune la care trimite o instrucţiune de transfer este o primă instrucţiune

Exemplu:

Se consideră următoarea secvenţă de program:

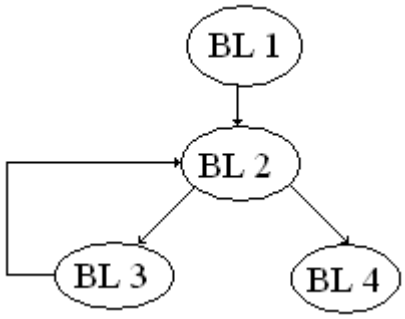
(1) $P \leftarrow 1$
(2) $i \leftarrow 2$
(3) *if* $i > 2$ *then goto* (7)
(4) $P \leftarrow P * i$
(5) $i \leftarrow i + 1$
(6) *goto* (3)
(7) $S \leftarrow P$

Blocurile de bază sunt prezentate în figura următoare:



0

Graful de flux asociat programului este următorul:



Optimizarea buclelor constă în:

- determinarea invarianților(variabile care nu-şi modifică valoarea pe parcursul execuției) din bucle şi scoaterea lor în afară.
- eliminarea variabilelor induse (sunt variabilele ale căror valori formează pe parcursul execuției repetate a ciclului o progresie aritmetică.

Optimizări locale – constau în eliminarea instrucțiunilor inutile, date de existența a două sau mai multe subexpresii comune, care produc acelaşi rezultat.

Teme propuse:

1. Implementați algoritmul de aducere a unei expresii artmetice la forma poloneză şi algoritmul de evaluare a formei poloneze în urma căreia se obține codul intermediar.
2. Implementați algoritmul care construieşte graful de flux asociat unui program