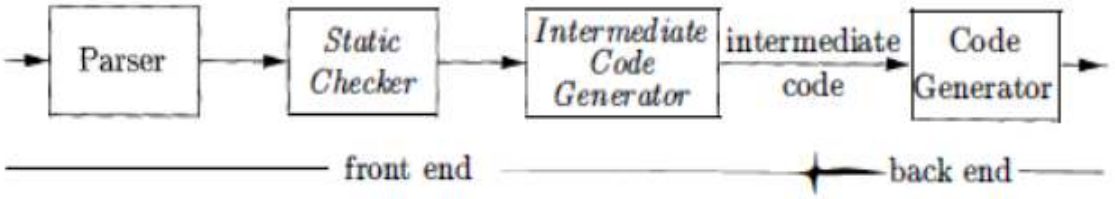
INTERMEDIATE-CODE GENERATION



Gambar struktur logika dari sebauh kompilator (frond end)

Intermediate Code

Memperkecil usaha dalam membuat compilator dari sejumlah bahasa ke sejumlah mesin

Lebih *Machine Independent*, hasil dari intermediate code dapat digunakan lagi pada mesin lainnya

Proses Optimasi lebih mudah. Lebih mudah dilakukan pada intermediate code dari pada *program sumber* (source program) atau pada kode *assembly* dan kode mesin

Intermediate code ini lebih mudah dipahami dari pada *kode assembly* atau kode mesin

Kerugiannya adalah melakukan 2 kali transisi, maka dibutuhkan waktu yang relatif lama

Intermediate Code

Ada dua macam intermediate code yaitu ***Notasi Postfix*** dan ***N-Tuple***

Notasi **POSTFIX**

<Operand> <Operand> < Operator>

Misalnya :

(a +b) * (c+d)

maka Notasi postfixnya

ab+ cd+ *

Semua instruksi kontrol program yang ada diubah menjadi notasi postfix, misalnya

IF <expr> **THEN** <stmt1> **ELSE** <stmt2>

POSTFIX

Diubah ke postfix menjadi ;

<expr> <label1> **BZ** <stmt1> <label2> **BR** < stmt2>

BZ : Branch if zero (salah)

BR: melompat tanpa harus ada kondisi yang ditest

Contoh : **IF** a > b **THEN** c := d **ELSE** c := e

POSTFIX

Contoh : **IF** a > b **THEN** c := d **ELSE** c := e

Dalam bentuk Postfix

11 a

12 b

13 >

14 22

15 BZ

16 c

17 d

18 :=

19

20 25

21 BR

22 c

23 e

24 :=

25

bila expresi (a>b) salah, maa loncat ke instruksi 22, Bila expresi (a>b) benar tidak ada loncatan, instruksi berlanjut ke 16-18 lalu loncat ke 25

POSTFIX

Contoh:

WHILE <expr> **DO** <stmt>

Diubah ke postfix menjadi ; <expr> <label1> **BZ** <stmt> <label2> **BR**

Instruksi : a:= 1

WHILE a < 5 DO

a := a + 1

Dalam bentuk Postfix

10 a	18 a
11 1	19 a
12 :=	20 1
13 a	21 +
14 5	22 :=
15 <	23 13
16 25	24 BR
17 BZ	25

TRIPLES NOTATION

Notasi pada triple dengan format

<operator> <operand> <operand>

Contoh:

A := D * C + B / E

Jika dibuat intermidiate code triple:

1. * , D, C
2. /, B, E
3. +, (1), (2)
4. :=, A, (3)

Perlu diperhatikan presedensi (hirarki) dari operator, operator perkalian dan pembagian mendapatkan prioritas lebih dahulu dari oada penjumlahan dan pengurangan

TRIPLES NOTATION

Contoh lain:

IF X > Y **THEN**

X := a - b

ELSE

X := a + b

Intermidiate code triple:

1. >, X, Y
2. BZ, (1), (6) bila kondisi 1 loncat ke lokasi 6
3. -, a, b
4. :=, X, (3)
5. BR, , (8)
6. +, a, b
7. :=, X, (6)

TRIPLES NOTATION

Kelemahan dari notasi ***triple*** adalah sulit pada saat melakukan optimasi, maka dikembangkan ***Indirect triples*** yang memiliki dua list; list instruksi dan list eksekusi. List Instruksi berisikan notasi triple, sedangkan list eksekusi mengatur eksekusinya; contoh

A := B + C * D / E

F := C * D

List Instruksi	***List Eksekusi***
1. *, C, D	1. 1
2. /, (1), E	2. 2
3. +, B, (2)	3. 3
4. :=, A , (3)	4. 4
5. :=, F, (1)	5. 1
	6. 5

Quardruples Notation

Format dari quardruples adalah

<operator> <operand> <operand> <result>

Result atau hasil adalah *temporary variable* yang dapat ditempatkan pada *memory* atau *register* . Problemnya adalah bagaimana mengelola temporary variable seminimal mungkin

Contoh:

A := D * C + B / E

Jika dibuat intermidiate codenya :

1. * , D, C, T1
2. / , B, E, T2
3. +, T1, T2, A

Quardruples Notation

Hasil dari tahapan anlisis diterima oleh code generator (pembangkit kode)

Intermediate code ditansfromasikan kedalam bahasa assembly atau mesin

Misalnya

(A+B)*(C+D) dan diterjemahkan kedalam bentuk quadruple:

1. +, A, B, T1
2. + , C, D, T2
3. *, T1, T2, T3

Dapat ditranslasikan kedalam bahasa assembly dengan accumulator tunggal:

Code Generator

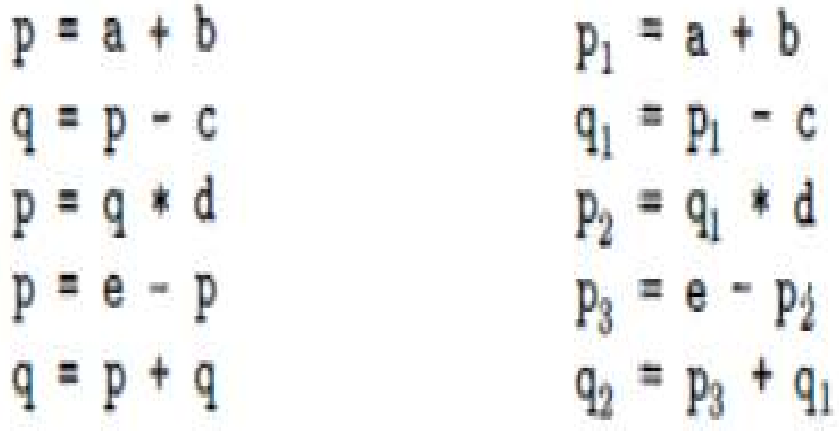
```
LDA   A    ( isi A ke dalam accumulator)
ADD   B    (isi accumulator dijumlahan dengan B)
STO   T1   ( Simpan isi Accumulator ke T1)
LDA   C
ADD   D
STO   T2
LDA   T1
MUL   T2
STO   T3
```

hasil dari *code generator* akan diterima oleh *code optimation* , Misalnya untuk kode assembly diatas dioptimasikan menjadi:

```
LDA   A
ADD   B
STO   T1
LDA   C
ADD   D
MUL   T1
STO   T2
```

Static single-assignment form/SSA

SSA merupakan representati intermediated yang dapat memfasilitasi optimasi kode tertentu



SYNTAX-DIRECTED TRANSLATION

- Kode antara (intermediate code) dibentuk dari sebuah kalimat x dalam Bahasa context free. Kalimat x ini adalah keluaran dari paser. Kalimat ini tentu saja dapat dinyatakan dalam representasi pohon parsing (parse tree). Syntax-directed translation adalah suatu urutan proses yang menstranslasikan parse tree menjadi kode-antara. Tahap pertama dari pembentukan kode antara lain adalah evaluasi atribut setiap token dalam kalimat x. yang dapat menjadi atribut setiap token adalah semua informasi yang disimpan di dalam table symbol. Evaluasi dimulai dari parse tree.

SYNTAX-DIRECTED TRANSLATION (Lanjutan)

Pandang sebuah mode n yang ditandai sebuah token X pada parse tree. Kita tuliskan X.a untuk menyatakan atribut a untuk token X pada mode n tersebut. Nilai X.a pada mode n tersebut dievaluasi dengan menggunakan aturan semantic (semantic rule) untuk atribut a. aturan semantic ini ditetapkan untuk setiap produksi dimana X adalah ruas kiri produksi. Sebuah parse tree yang menyertakan nilai-nilai atribut pada setiap nodenya dinamakan annonated parse tree kumpulan aturan yang menetapkan aturan-aturan semantic untuk setap sebuah kalimat yang dihasilkan oleh parser.

Diketahui :

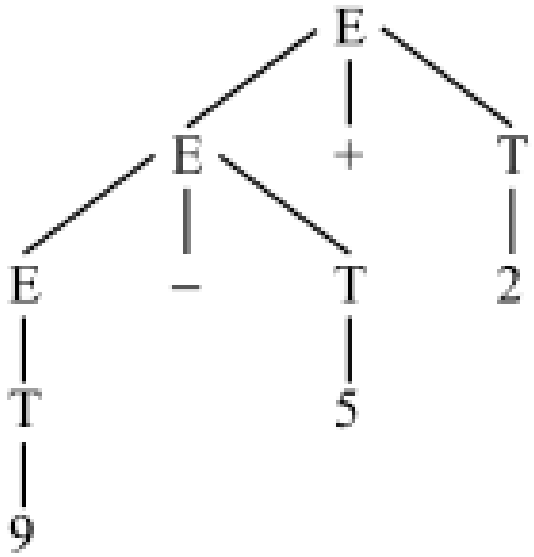
1. Kalimat x : 9-5+2
2. Grammar $Q = \{E \rightarrow E + T | E - T | T,\ T \rightarrow 0|1|3| \ldots |9\}$
3. Syntax-directed definition

Produksi	Aturan Semantik
$E \rightarrow E_1 + T$	E := E_1 .t ‖ T.t ‖ '+ '
$E \rightarrow E_1 - T$	E := E_1 .t‖ T.t ‖ '–'
$E \rightarrow T$	E := T.t
$E \rightarrow 0$	E := '0'
$E \rightarrow 1$	E := '1'
...	...
$E \rightarrow 9$	E := '9'

catatan : 1. Lambang ' ‖ ' menyatakan *concatenation*.
2. Mengingat catatan 1, aturan semantik kedua produksi pertama adalah *concate* dua operan diikuti sebuah operator.

Langkah-langkah translasi

1. pembentukan *parse tree* :



2. pembentukan *annonated parse tree* :