TEKNIK KOMPILASI

Muhamad Nursalman

Ilmu Komputer

FPMIPA - UPI

Daftar Isi

Bab I Pendahuluan

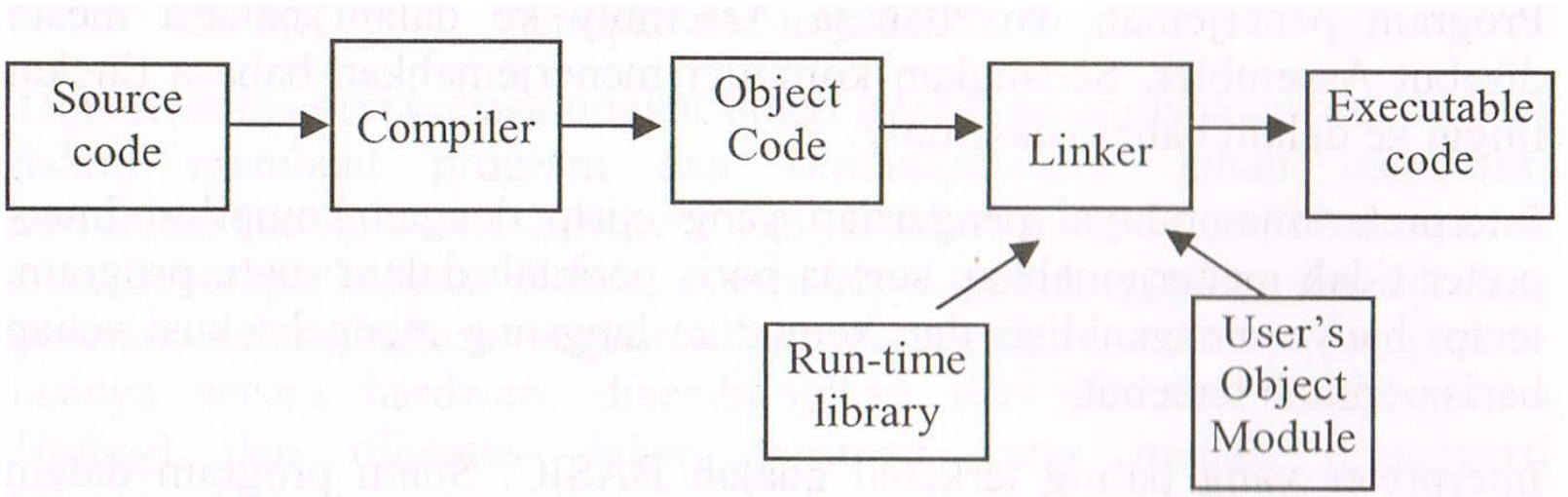
- Kompiler (compiler) adalah program yang menerjemahkan bahasa seperti Pascal, C, PL/I, FORTRAN atau COBOL ke dalam bahasa mesin.
- Program penerjemah dari bahasa Assembly ke dalam bahasa mesin disebut Assembler. Sedangkan kompiler menerjemahkan bahasa tingkat tinggi ke dalam bahasa assembly.
- Interpreter mempunyai pengertian yang mirip dengan kompiler. Inerpreter tidak menerjemahkan semua baris perintah dalam suatu program, tetapi hanya menganalisis dan kemudian langsung mengeksekusi setiap baris perintah tersebut.

Bab I Pendahluluan (2)

- Keuntungan interpreter adalah user dapat cepat memperoleh tanggapan. Dengan menulis satu baris perintah, lalu menulis RUN, pemakai bisa langsung mengetahui hasilnya. Sedangkan kerugiannya adalah lambat.
- Berbeda dengan kompiler yang tidak menerjemahkan semua perintah program sumber menjadi object code, tetapi kompiler akan menyediakan subroutine khusus dan subroutine tsb hanya akan digunakan pada saat program hasil kompilasi dijalankan. Kumpulan subroutine tsb dinamakan *run-time library*.

Bab I Pendahuluan (3)

- Skematis Proses Kompilasi



Bab I Pendahuluan (4)

- Tahap-tahap kompilasi:
  - Analisis Leksikal
  - Analisis Sintaktik/Semantik
  - Intermediate Code Generation
  - Optimization
  - Object Code Generation

Bab II Analisis Leksikal

- Tugas utama penganalisis leksikal adalah memecah tiap baris source menjadi token-token.
- Pekerjaan yang dikerjakannya antara lain:
  - Membuang komentar
  - Menyeragamkan huruf kapital menjadi huruf kecil atau sebaliknya
  - Membuang white space
  - Menginterpretasikan kompiler directive
  - Berkomunikasi dg symbol table
  - Mebuat listing

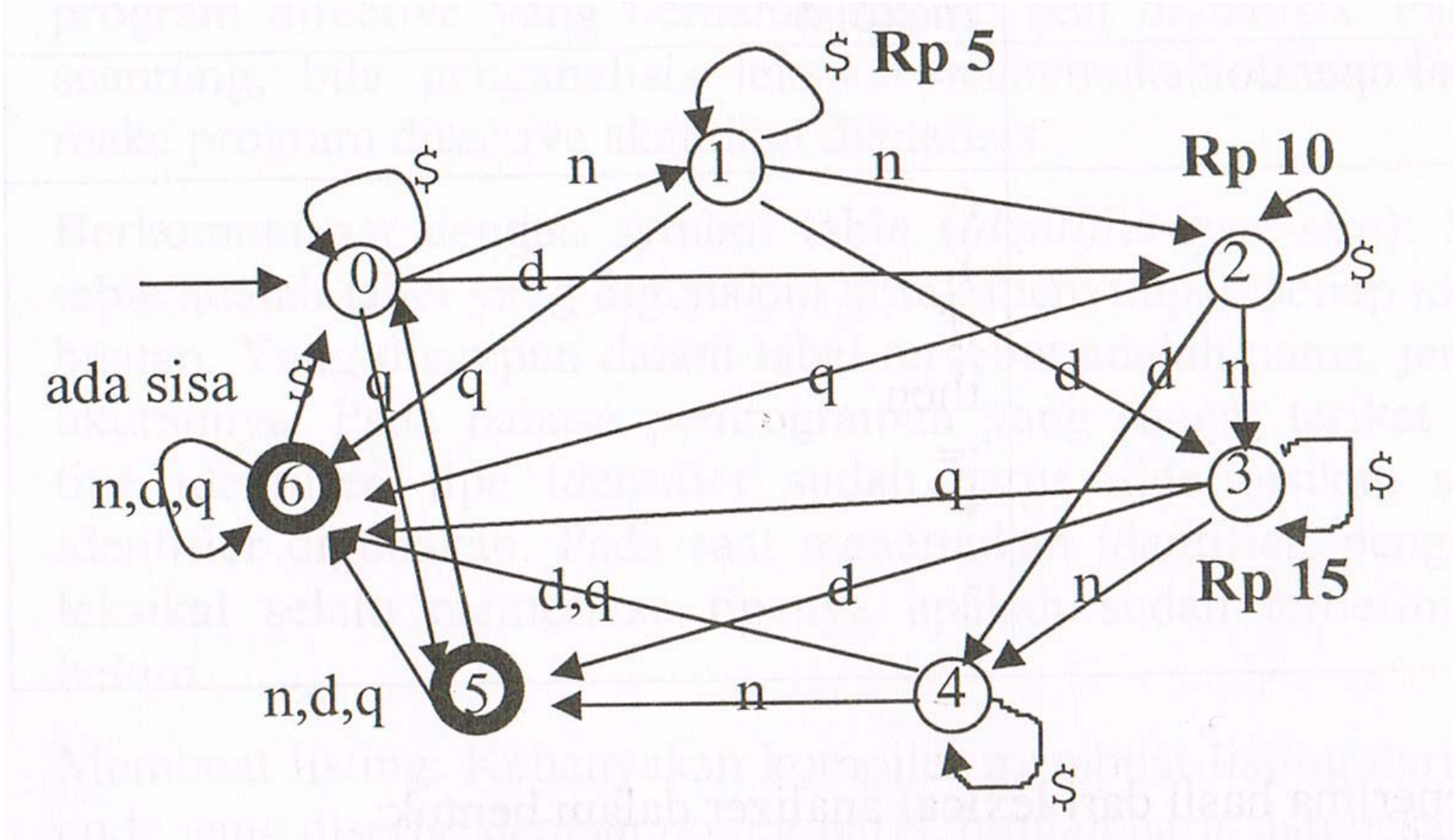
Bab II Analisis Leksikal (2)

- Analisis leksikal lebih mudah diimplementasikan pada Finite State Machine atau Finite State Automata. Materi ini mempelajari sehimpunan state beserta dg aturan-aturan perpindahan dari satu state ke state yang lainnya. Sehimpunan state tersebut menyatakan satu proses dan aturan-aturannya menyatakan kemungkinan2 yang terjadi dalam menyelesaikan proses tersebut.

Bab II Analisis Leksikal (3)

- State Diagram dan State Table
- Contoh1: Ada mesin penjual permen yang memuat aturan-aturan sebagai berikut: harga permen Rp 25. Mesin tersebut dapat dimasuki 3 jenis koin, Rp 5 (n), Rp 10 (d), Rp 25 (q). $ = tombol untuk mengeluarkan permen. Kemungkinan2 yang terjadi dalam proses tersebut digambarkan dalam state diagram berikut ini:

Bab II Analisis Leksikal (4)

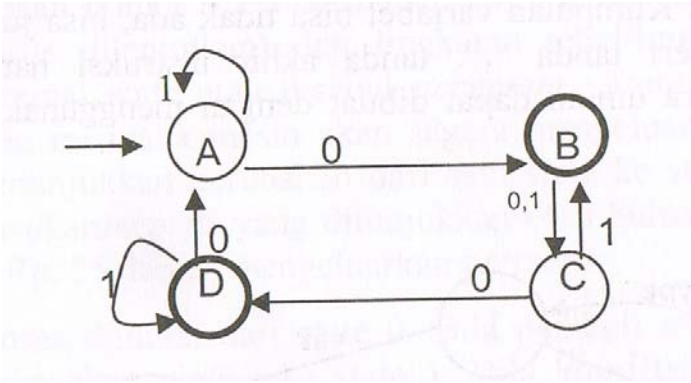


Bab II Analisis Leksikal (5)

Current state	input Rp.5 (n)	Rp.10 (d)	Rp. 25 (q)	tekan tombol ($)
0	1	2	5	0
1	2	3	6	1
2	3	4	6	2
3	4	5	6	3
4	5	6	6	4
5	6	6	6	0*
6	6	6	6	0*
CURRENT	NEXT STATE			

Bab II Analisis Leksikal (6)

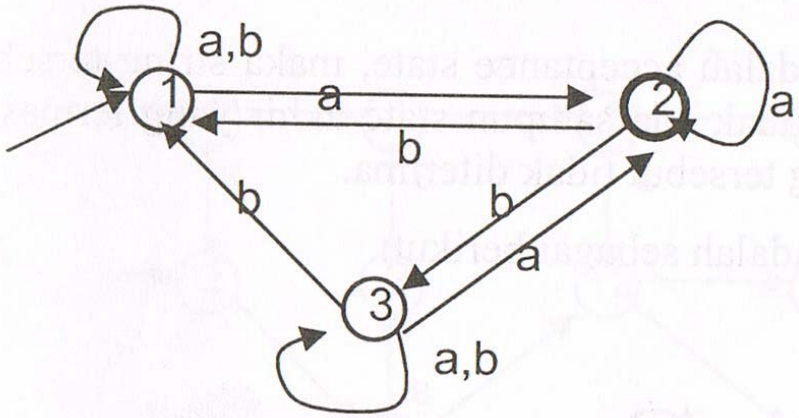
- Finite State Automata



Current state	INPUT 0	1
A	B	A
B	C	C
C	D	B
D	A	D

Bab II Analisis Leksikal (7)

- Non-Deterministic FSA (NFA)

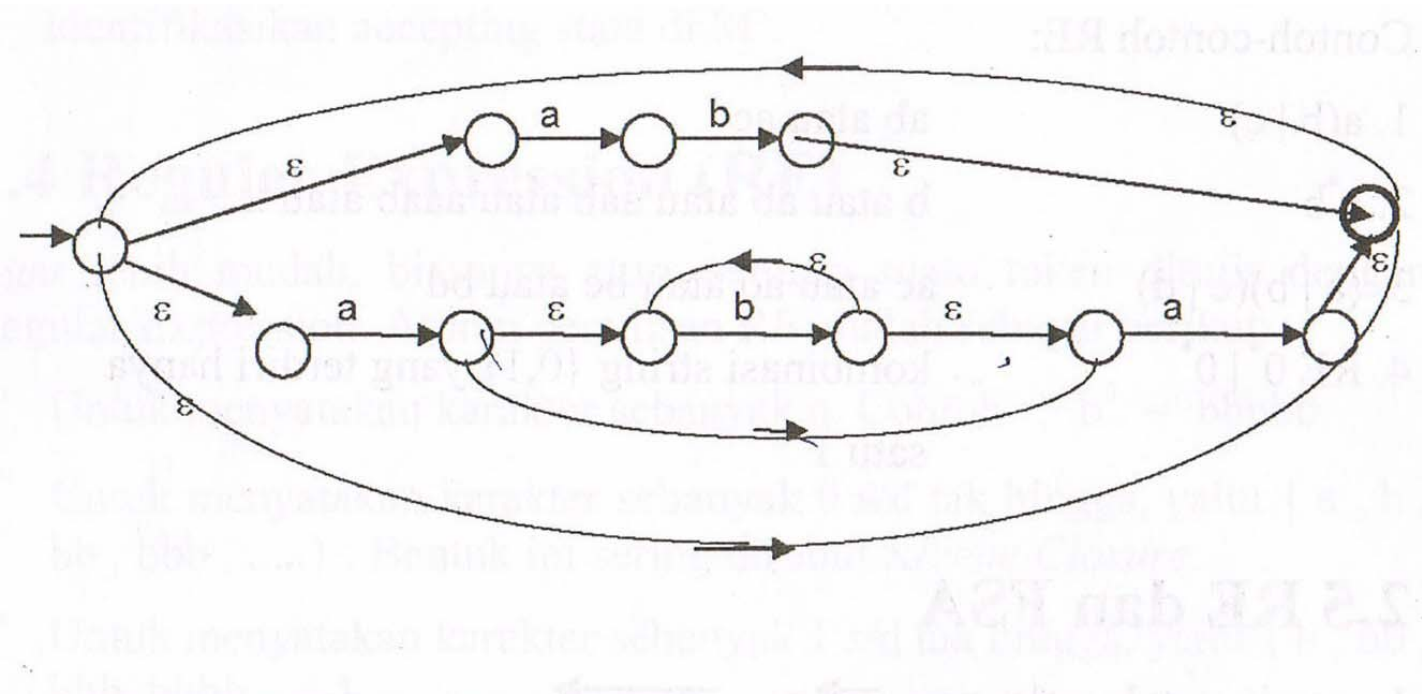


State table

Current state	INPUT	
	a	b
1	{1,2}	{1}
2	{2}	{1,3}
3	{2,3}	{1,3}

Bab II Analisis Leksikal (8)

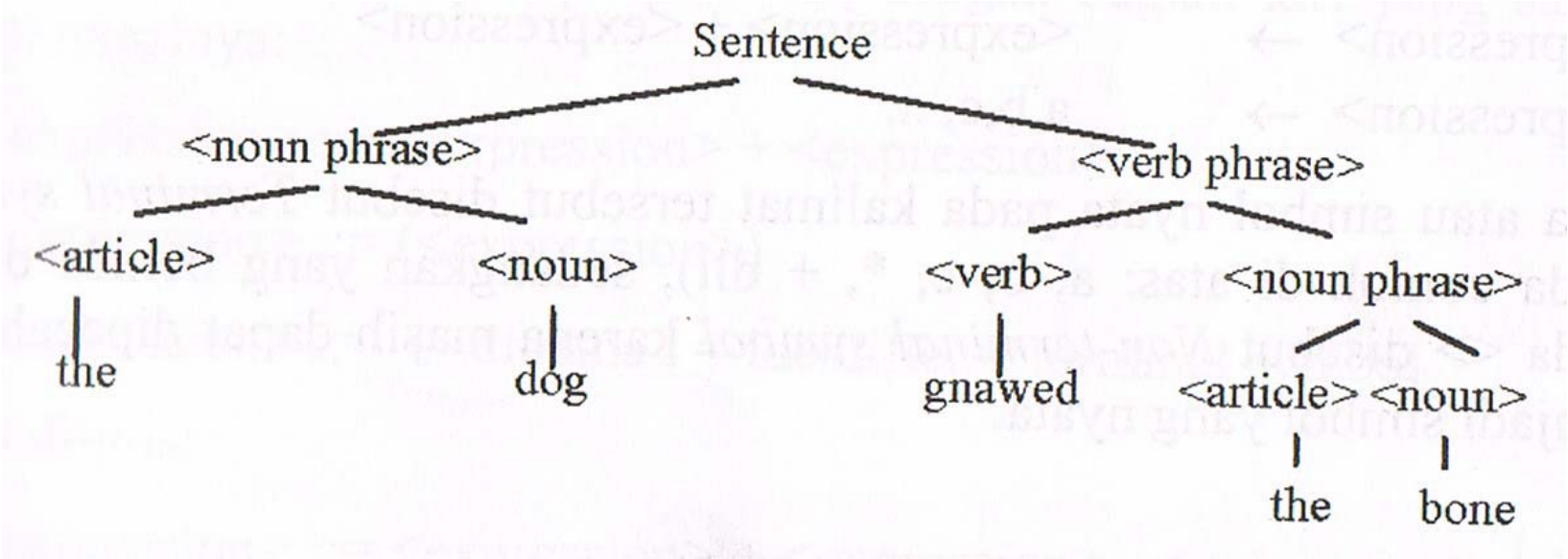
- Reguler Expression (RE) & FSA untuk (ab|ab*a)* dinyatakan dengan



Bab III Analisis Sintaktik

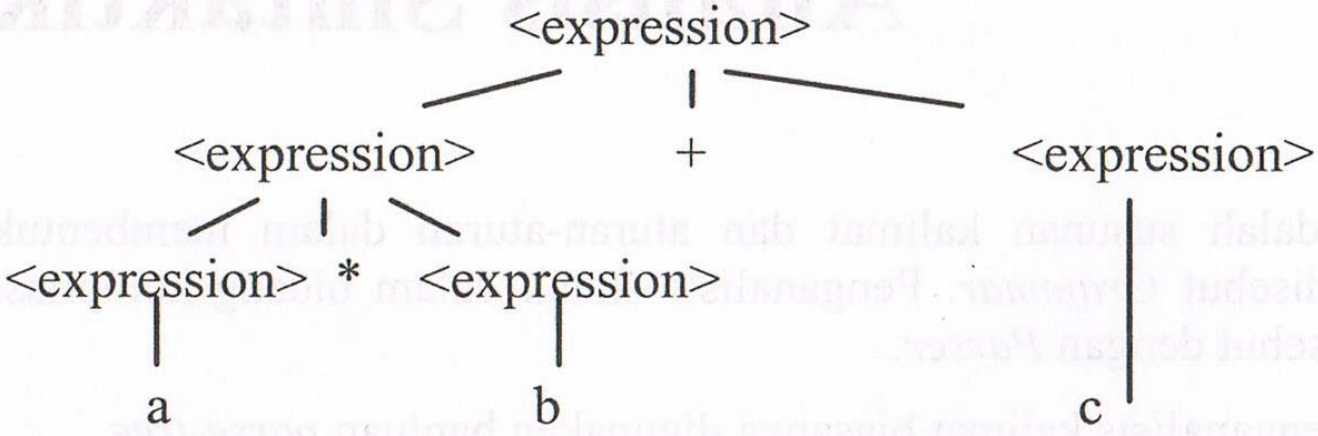
- Sintak adalah susunan kalimat dan aturan-aturan dalam membentuk kalimat disebut grammar. Penganalisis intak dalam bidang kompilasi sering disebut dengan *Parser*.
- Untuk menganalisis kalimat biasanya digunakan bantuan *parse-tree*.
- Contoh: Perhatikan kalimat berikut ini, “The dog gnawed the bone”. Kalimat tersebut disusun dalam bentuk parse-tree berikut ini:

Bab III Analisis Sintaktik (1)



Bab III Analisis Sintaktik (2)

- a*b+c disusun dalam bentuk parse-tree:



Dalam bentuk Production rule:

<expression> → <expression> * <expression>
<expression> → <expression> + <expression>
<expression> → a,b,c, ...

Bab III Analisis Sintaktik (3)

- Definisi Formal dari Grammar (T,N,S,R):
  - T: Himpunan simbol terminal
  - N: Himpunan simbol non terminal
  - S: Simbol awal yang unik, SeN
  - R: Himpunan produksi dengan bentuk

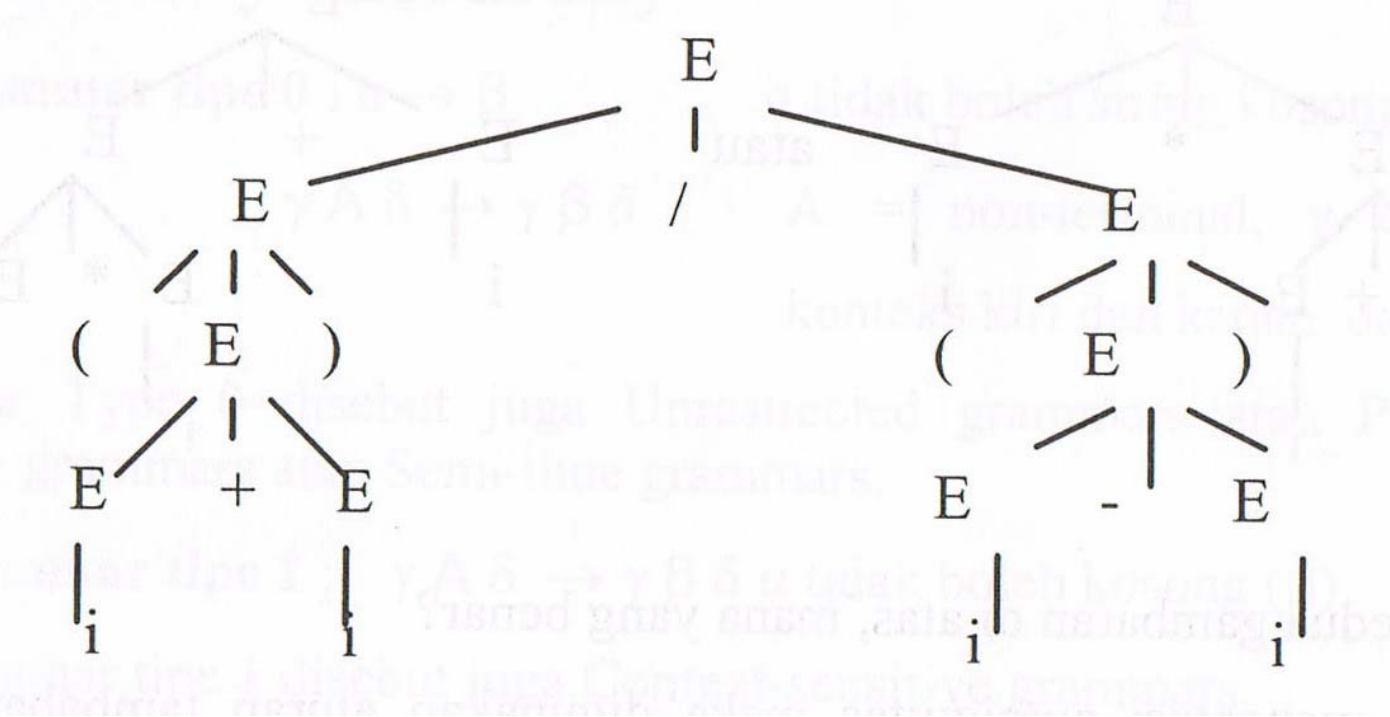
    alpha->betha, alpha dan betha adalah kumpulan simbol non terminal dan terminal.

Bab III Analisis Sintaktik (4)

- Parse Tree dan Penurunannya
- Contoh: G=(T,N,S,R) di mana T = {i,+,-,*,/,(,)},
  N = {E}
  S = E
  R = { E->E+E,
  E->E-E,
  E->E*E,
  E->E/E,
  E->(E),
  E->i };

Bab III Analisis Sintaktik (5)

- (a+b)/(a-b)



Bab III Analisis Sintaktik (6)

- **Grammar Noam Chomsky:**

1. **Kelas 0** Unrestricted grammar (aturan produksinya tak dibatasi)
2. **Kelas 1** Context sensitive grammar, di mana $\alpha \rightarrow \beta$ dengan $|\alpha| \leq |\beta|$
3. **Kelas 2** Context free grammar, di mana $\alpha \rightarrow \beta$ $\alpha \in V_N$ dan β adalah $(V_N \cup V_T)^*$
4. **Kelas 3** Regular grammar di mana $\alpha \rightarrow \beta$ dengan $|\alpha| \leq |\beta|$, $\alpha \in V_N$ dan β berbentuk aB atau a, dengan $a \in V_T^*$ dan $B \in V_N$.

Bab IV Analisis Semantik

- memeriksa token dan ekspresi dari batasan-batasan yang ditetapkan. Batasan-batasan tersebut misalnya :
  - panjang maksimum token *identifier* adalah 8 karakter,
  - panjang maksimum ekspresi tunggal adalah 80 karakter,
  - nilai bilangan bulat adalah -32768 s/d 32767,
  - operasi aritmatika harus melibatkan operan-operan yang bertipe sama

Bab IV Analisis Semantik (2)

Contoh : A := (A+B) * (C+D)

- *Parser* hanya akan mengenali simbol-simbol ‘:=‘, ‘+’, dan ‘*’, parser tidak mengetahui makna dari simbol-simbol tersebut
- Untuk mengenali makna dari simbol-simbol tersebut, Compiler memanggil routin semantics

Bab IV Analisis Semantik (3)

Untuk mengetahui makna, maka routin ini akan memeriksa:

- Apakah variabel yang ada telah didefinisikan sebelumnya
- Apakah variabel-variabel tersebut tipenya sama
- Apakah operand yang akan dioperasikan tersebut ada nilainya, dan seterusnya
- Menggunakan tabel simbol
- Pemeriksaan bisa dilakukan pada tabel *identifier*, tabel *display*, dan tabel *block*

Bab IV Analisis Semantik (4)

Pengecekan yang dilakukan dapat berupa:

- Memeriksa penggunaan nama-nama (keberlakuannya)
  - **Duplikasi**

    Apakah sebuah nama terjadi pendefinisian lebih dari dua kali. Pengecekan dilakukan pada bagian pengelolaan block
  - **Terdefinisi**

    Apakah nama yang dipakai pada program sudah terdefinisi atau belum. Pengecekan dilakukan pada semua tempat kecuali block
- Memeriksa tipe

  Melakukan pemeriksaan terhadap kesesuaian tipe dalam *statement - statement* yang ada, Misalnya bila terdapat suatu operasi, diperiksa tipe operand nya

Bab IV Analisis Semantik (5)

Contohnya;

- expresi yang mengikut **IF** berarti tipenya boolean, akan diperiksa tipe *identifier* dan tipe ekspresinya
- Bila ada operasi antara dua operand maka *tipe operand* pertama harus bisa dioperasikan dengan *operand* yang kedua

Analisa Semantic sering juga digabungkan dengan *intermediate code* yang akan menghasilkan *output intermediate code*.

Intermediate code ini nantinya akan digunakan pada proses kompilasi berikutnya (pada bagian *back end compilation*)

Bab V Intermediate Code

- membangkitkan kode antara (*intermediate code*) berdasar-kan pohon parsing. Pohon parse selanjutnya diterjemahkan oleh suatu penerjemah yang dinamakan *penerjemah berdasarkan sintak* (*syntax-directed translator*). Hasil penerjemahan ini biasanya merupakan *perintah tiga alamat* (*three-address code*) yang merupakan representasi program untuk suatu *mesin abstrak*. Perintah tiga alamat bisa berbentuk *quadruples* (*op*, *arg1*, *arg2*, *result*), *tripels* (*op*, *arg1*, *arg2*). Ekspresi dengan satu argumen dinyatakan dengan menetapkan *arg2* dengan - (strip, *dash*)

Bab V Intermediate Code (2)

- 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

Bab V Intermediate Code (3)

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>

Bab V Intermediate Code (4)

- 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

Bab V Intermediate Code (5)

POSTFIX

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

Dalam bentuk Postfix

11	a	19	
12	b	20	25
13	>	21	BR
14	22		22 c
15	BZ		23 e
16	c	24	:=
17	d	25	
18	:=		

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

Bab V Intermediate Code (6)

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

Bab V Intermediate Code (7)

Triple 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

Bab V Intermediate Code (8)

TRIPLE 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)

Bab V Intermediate Code (9)

TRIPLE 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

Bab V Intermediate Code (10)

QUARDRUPLE 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

Bab V Intermediate Code (11)

QUARDRUPLE 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:

Bab VI Code Optimization

- melakukan optimasi (penghematan *space* dan *waktu komputasi*), jika mungkin, terhadap kode antara

Bab VI Code Optimization (2)

- **Dependensi Optimasi**
- **Optimasi Lokal**
- **Optimasi Global**

- ***Dependensi Optimasi***

  bertujuan untuk menghasilkan kode program yang berukuran lebih kecil dan lebih cepat

  - Machine Dependent Optimizer
  - Machine Independent Optimizer (Optimasi lokal dan Optimasi global)

Bab VII Code Optimization (3)

Optimasi Lokal : adalah optimasi yang dilakukan hanya pada suatu blok dari source code, dengan cara:

- ✓ ***Folding***

  menganti konstata atau ekpresi yang bisa dievaluasi pada saat *compile time* dengan nilai komputasinya. Misalnya:

  A := 2 + 3 + B bisa diganti dengan A:= 5 + B

  5 dapat mengantikan ekspresi 2 + 3

- ✓ ***Redundant-Subexpression Elimination***

  hasilnya digunakan lagi dari pada dilakukan computasi ulang, contoh:

  A:= B + C

  X := Y + B + C

Bab V Code Optimization (4)

- ✓ **Optimasi dalam sebuah Iterasi**
  - **_Loop Unrrolling_:Menganti suatu _loop_ dengan menulis statement yang ada dalam loop ditulis beberapa kali**
  - **Karena sebuah iterasi pada implemnetasi ke level rendah, memerlukan :**
    - **Inisialisasi nilai awal, pada loop dilakukan sekali pada saat permulaan eksekusi loop**
    - **Penge-test-an, apakah variabel loop telah mencapai kondisi terminasi**
    - **Adjustment yaitu: penambahan atau pengurangan nilai pada variabel loop dengan jumlah tertentu**
    - **Operasi yang terjadi pada tubuh perulangan (loop body)**

Bab V Code Optimization (5)

- Contoh :

  FOR I := 1 to 2 DO
  A[I] := 0;

  dapat dioptimasikan menjadi

  A[1] := 0;
  A[2] := 0;

- Frequency Reduction: Pemindahan statement ke tempat yang lebih jarang dieksekusi, contoh

```
FOR I:= 1 to 10 DO
BEGIN
  X := 5
  A := A + 1
END:
```

```
X := 5
FOR I:= 1 to 10 DO
BEGIN
   A := A + 1
END:
```

Bab V Code Optimization (6)

- ✓ Strength Reduction
  - Penggantian suatu operasi dengan operasi lain yang lebih cepat dieksekusi
  - misalnya: pada komputer operasi perkalian memerlukan waktu eksekusi lebih banyak dari pada operasi penjumlahan
  - contoh lain

    A:= A + 1
  - dapat digantikan dengan

    INC(A)

Bab V Code Optimization (7)

Optimasi global biasanya dilakukan dengan suatu graph terarah yang menunjukkan jalur yang mungkin selama eksekusi program

ada dua kegunaan yaitu bagi programmer dan compiler itu sendiri

- ✓ **Bagi Programmer**
  - **_Unreachable/dead code_: Kode yang tidak pernah dieksekusi**
  - **misalnya :**

```
X :=  5;
IF X = 0 THEN
    A := A + 1
```

Instruksi

A := A + 1 tidak pernah dikerjakan

Bab V Code Optimization (8)

- ***Unused parameter*** : parameter yang tidak pernah digunakan dalam procedure
  - Misalnya :

```
procedure penjumlahan(a,b,c ; Integer);
        var x : integer;
        begin
          x := a + b;
        end
```

Parameter **c** tidak pernah digunakan sehingga tidak perlu diikut sertakan

Bab V Code Optimization (9)

- ***Unsused Variabel :*** variabel yang yang tidak pernah dipergunakan

```
Program pendek;
var a, b: integer
begin
    a := 5;
end;
```

- B tidak pernah digunakan

Bab V Code Optimization (10)

- ***Variabel :*** variabel yang dipakai tanpa nilai awal. Contoh

```
Program Awal;
var a, b: integer
begin
    a := 5
   a := a + b;
end;
```

  - variabel b digunakan tetapi tidak memiliki harga awal

✓ ***Bagi Compiler***

- Meningkatkan efisiensi eksekusi program
- Menghilangkan useless code/kode yang tidak terpakai

Bab VII Code Optimization (11)

```
var A, B, C, D, E, I, J, X, Y : integer;
begin
        B := 5;
      A := 10 - B / 4 * 3 + 2;
      C :=  A + B;
      Y := 10;
      D := A + B – E;
        for  I := 1 to 85 do
                begin
                    X := X + 1;
                    B := B – X;
                    Y := 7;
                end
        While Y< 5 do
             E := E – B;
end
```

Bab VII Code Generation

- membangkitkan kode dalam bahasa target tertentu (misalnya bahasa mesin)

Bab VII Code Generation (2)

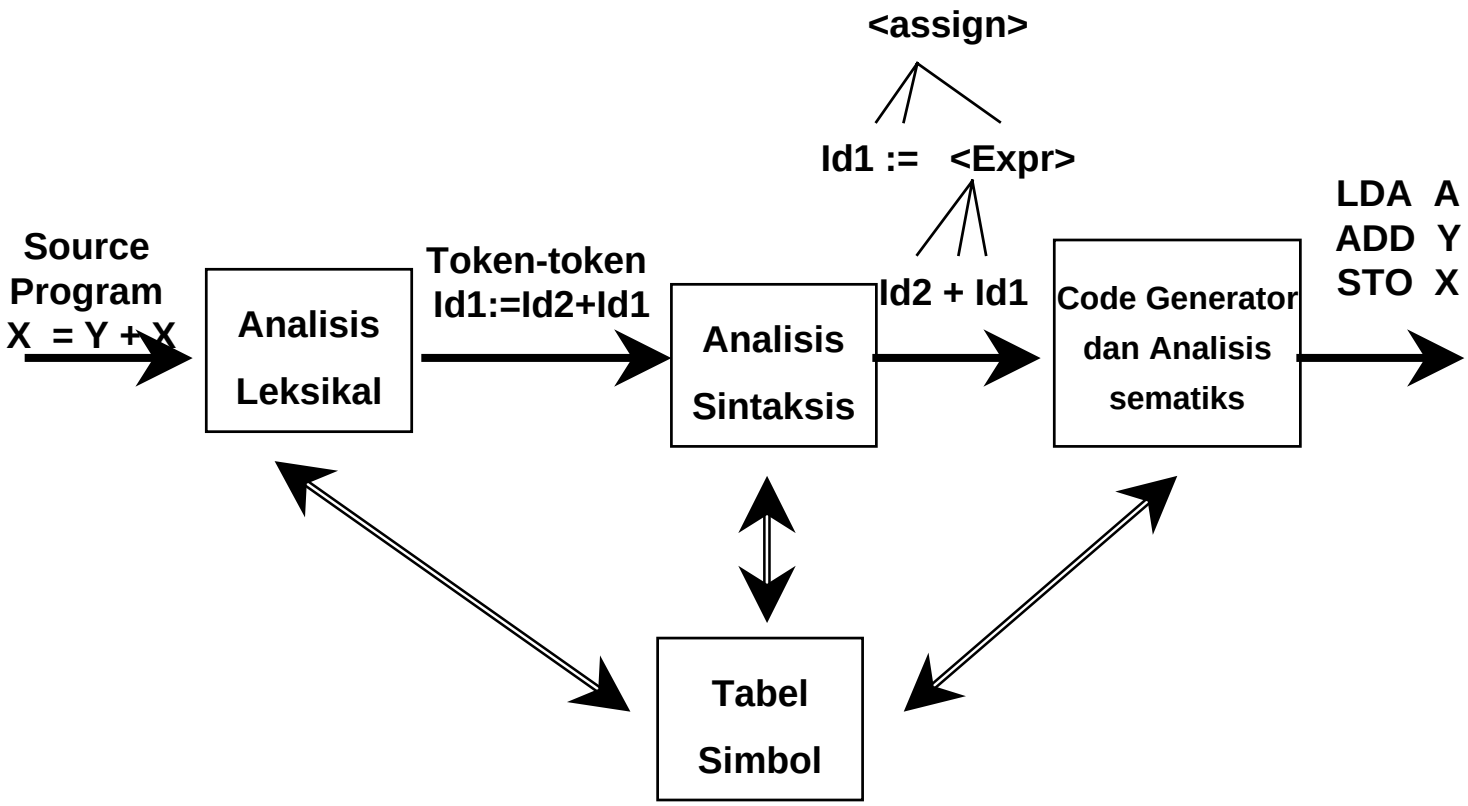
```
LDA     A       ( isi A ke dalam accumulator)
ADD     B       (isi accumulator dijumlahan dengan B)
STO     T1      ( Simpan isi Accumulator ke T1)
LDA     C
ADD     D
STO     T2
LDA     T1
MUL     T2
STO     T3
```

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
```

Bab VII Code Generation (3)

- Perjalanan Sebuah Instruksi



- Contoh Teknik Kompilasi: