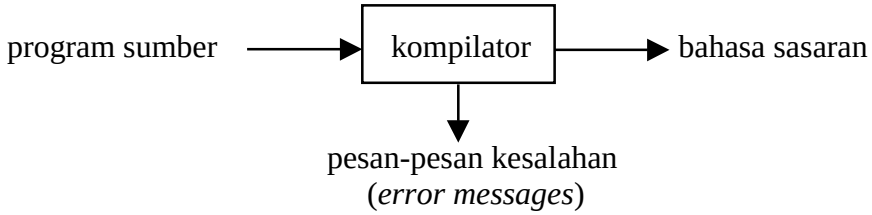
CATATAN KULIAH
TEKNIK KOMPILASI
Oleh: Asep Juarna

KOMPILASI, Catatan ke-1 : Pendahuluan

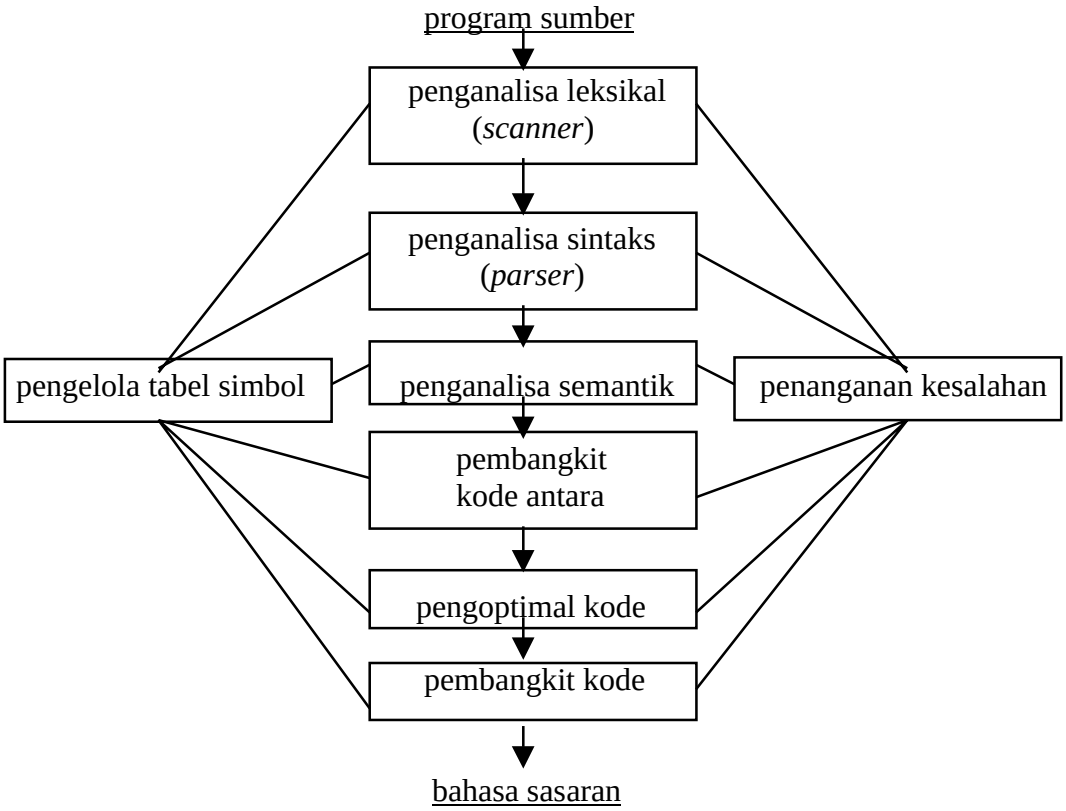
Definisi : Kompilator (*compiler*) adalah sebuah *program* yang membaca suatu program yang ditulis dalam suatu *bahasa sumber* (*source language*) dan menterjemahkannya ke dalam suatu *bahasa sasaran* (*target language*).

Proses kompilasi dapat digambarkan melalui sebuah kotak hitam (*black box*) berikut :



Proses kompilasi dikelompokkan ke dalam dua kelompok besar :
1. *analisa* : program sumber dipecah-pecah dan dibentuk menjadi bentuk antara (*intermediate representation*)
2. *sintesa* : membangun program sasaran yang diinginkan dari bentuk antara

Fase-fase proses sebuah kompilasi adalah sebagai berikut :



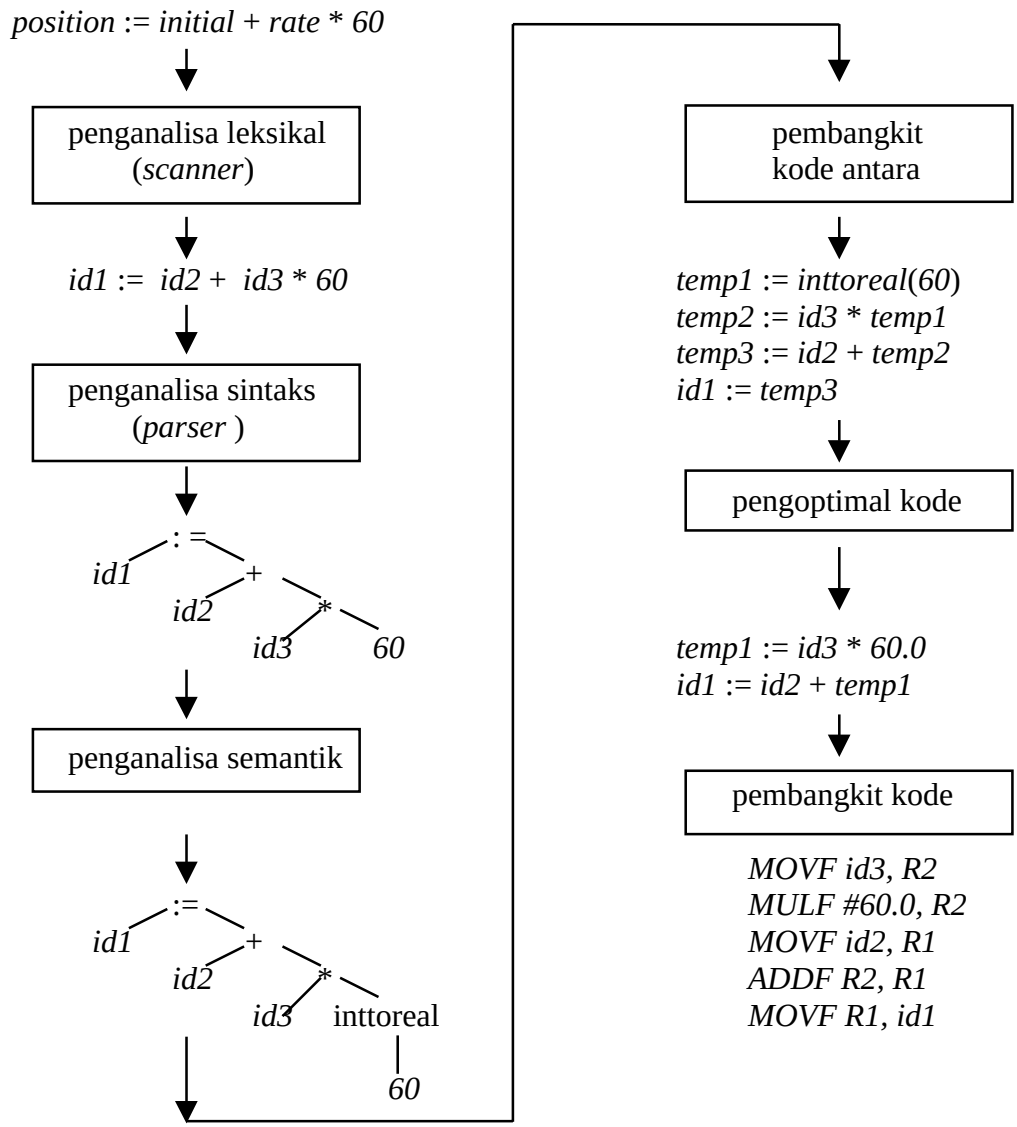
Program sumber merupakan rangkaian karakter. Berikut ini hal-hal yang dilakukan oleh setiap fase pada proses kompilasi terhadap program sumber tersebut :

1. Penganalisa leksikal : membaca program sumber, karakter demi karakter. Sederetan (satu atau lebih) karakter dikelompokkan menjadi satu kesatuan mengacu kepada *pola kesatuan kelompok karakter* (*token*) yang ditentukan dalam *bahasa sumber*. Kelompok karakter yang membentuk sebuah token dinamakan *lexeme* untuk token tersebut. Setiap token yang dihasilkan disimpan di dalam *tabel simbol*. Sederetan karakter yang tidak mengikuti pola token akan dilaporkan sebagai *token tak dikenal* (*unidentified token*).

   Contoh : Misalnya pola token untuk *identifier* I adalah : I = *huruf*(*huruf*|*angka*)*. Lexeme *ab2c* dikenali sebagai token sementara lexeme *2abc* atau *abC* tidak dikenal.
2. Penganalisa sintaks : memeriksa kesesuaian *pola deretan token* dengan aturan sintaks yang ditentukan dalam *bahasa sumber*. Sederetan token yang tidak mengikuti aturan sintaks akan dilaporkan sebagai *kesalahan sintaks* (*sintax error*). Secara logika deretan token yang bersesuaian dengan sintaks tertentu akan dinyatakan sebagai pohon parsing (*parse tree*).

   Contoh : Misalnya sintaks untuk ekspresi *if-then* E adalah : E $\rightarrow$ *if* L *then*, L $\rightarrow$ IOA, I = *huruf*(*huruf*|*angka*)*, O $\rightarrow$ <|=|>|<=|>=, A $\rightarrow$ 0|1|...|9. Ekspresi *if a2 < 9 then* adalah ekspresi sesuai sintaks; sementara ekspresi *if a2 < 9 do* atau *if then a2B < 9* tidak sesuai. Perhatikan bahwa contoh ekspresi terakhir juga mengandung token yang tidak dikenal.
3. Penganalisa semantik : memeriksa token dan ekspresi dari batasan-batasan yang ditetapkan. Batasan-batasan tersebut misalnya :
   a. panjang maksimum token *identifier* adalah 8 karakter,
   b. panjang maksimum ekspresi tunggal adalah 80 karakter,
   c. nilai bilangan bulat adalah -32768 s/d 32767,
   d. operasi aritmatika harus melibatkan operan-operan yang bertipe sama.
4. Pembangkit kode antara : membangkitkan kode antara (*intermediate code*) berdasarkan 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*)
5. Pengoptimal kode : melakukan optimasi (penghematan *space* dan *waktu komputasi*), jika mungkin, terhadap kode antara.
6. Pembangkit kode : membangkitkan kode dalam bahasa target tertentu (misalnya bahasa mesin).

Berikut ini akan diberikan sebuah contoh skema penerjemahan suatu ekspresi dalam bahasa sumber, yaitu : *position* := *initial* + *rate* * *60*.



Keterangan :

- id adalah token untuk *identifier*. Tiga *lexeme* untuk token ini adalah position, initial, dan rate.
- penganalisa semantik secara logika membangkitkan pohon parse.
- penganalisa semantik mendeteksi *mismatch type*. Perbaikan dilakukan dengan memanggil *procedure* inttoreal yang mengkonversi integer ke real.
- quadruples dari : *temp2* := *id3* * *temp1* adalah (*,*id3*,*temp1*,*temp2*), *id1* := *temp3* adalah (*assign*,*temp3*,-,*id1*), *temp1* := *inttoreal*(*60*) adalah (*inttoreal*,*60*,-,*temp1*).
- pembangkit kode dalam contoh ini menghasilkan kode dalam bahasa mesin.

KOMPILASI, Catatan ke-2 : Merancang Bahasa Sumber dan Scanner

1. Bahasa Sumber

Bahasa adalah kumpulan kalimat. Kalimat adalah rangkaian kata. Kata adalah unit terkecil komponen bahasa yang tidak bisa dipisah-pisahkan lagi. Kalimat-kalimat : '*Seekor kucing memakan seekor tikus.*' dan '*Budi menendang sebuah bola.*' adalah dua contoh kalimat lengkap Bahasa Indonesia. '*A cat eats a mouse*' dan '*Budi kick a ball.*' adalah dua contoh kalimat lengkap Bahasa Inggeris. '*if a2 < 9.0 then b2 := a2+a3;*' dan '*for i := start to finish do A[i] := B[i]*sin(i*pi/16.0).*' adalah dua contoh kalimat lengkap dalam Bahasa Pemrograman Pascal. Dalam bahasa pemrograman *kalimat* lebih dikenal sebagai *ekspresi* sedangkan *kata* sebagai *token*.

Perancangan sebuah bahasa harus memperhatikan tiga aspek berikut :

1. *spesifikasi leksikal*, misalnya setiap kata harus tersusun atas huruf mati dan huruf hidup yang disusun bergantian, atau setiap token harus dimulai dengan huruf dan selanjutnya boleh diikuti oleh huruf atau angka,
2. *spesifikasi sintaks*, misalnya setiap kalimat mengikuti pola *subyek-predikat-obyek* atau ekspresi *for_do* mengikuti pola *for*-identifier-*:*=-identifier-*to*-identifier-*do*-ekspresi.
3. *aturan-aturan semantik,* misalnya kata yang mendahului kata kerja haruslah kata benda yang menggambarkan sesuatu yang hidup dan berkaki, atau operasi perkalian hanya bisa dilakukan antara dua operan dengan tipe yang sama.

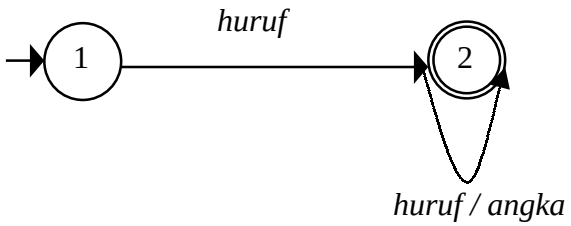
Berdasarkan rancangan bahasa di atas, perhatikan hal-hal berikut. Kita tidak bisa mengganti kata *Budi* dengan *2udi* sebagaimana kita tidak bisa mengganti token *start* dengan *?tart*. Kita juga tidak bisa merubah susunan kata-kata menjadi *Budi sebuah menendang bola* sebagaimana kita tidak boleh merubah susunan token-token menjadi *9.0 if < a2 then b2:= a2*. Demikian pula kita tidak boleh mengganti kata *Budi* dengan *lemari* sebagaimana kita tidak boleh mengganti *B[i]*sin(i*pi/16.0)* dengan *B*sin(i*pi/16.0)*.

Dalam spesifikasi leksikal biasanya digunakan *grammar regular* (GR) dalam bentuk *ekspresi regular* (ER). Sebagai contoh pola token *identifier* ditentukan oleh *grammar regular* berikut :
I→ *aA*|*bA*|...|*zA*|*a*|*b*|...|*z*, A→*aA*|*bA*|...|*zA*|*0A*|*1A*|...|*9A*|*a*|*b*|...|*z*|*0*|*1*|...|*9*
yang ekuivalen dengan *ekspresi regular* berikut :
I = (*a*|*b*|...|*z*)(*a*|*b*|...|*z*|*0*|*1*|...|*9*)* = *huruf*(*huruf*|*angka*)*

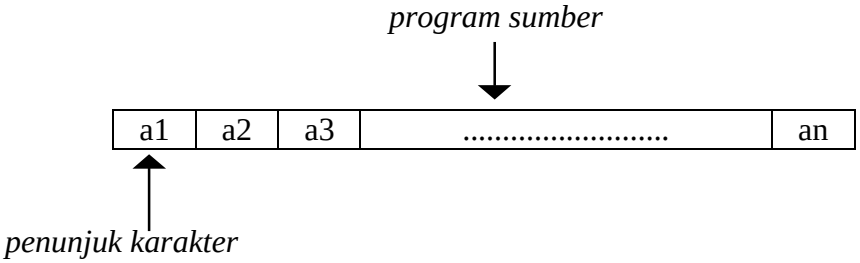
Dalam spesifikasi sintaks biasanya digunakan *context free grammar* (CFG). Sebagai contoh ekspresi *if-then* E adalah :
E → *if* L *then*, L → IOA, I = *huruf*(*huruf*|*angka*)*, O → <|=|>|<=|>=, A→ *0*|*1*|...|*9*.

2. Scanner

Scanner diimplementasikan dengan *Automata Hingga Deterministik* (AHD). Pada kuliah *Teori Bahasa dan Automata* (atau *Pengantar Automata, Bahasa Formal, dan Kompilasi*) telah dipelajari siklus transformasi : GR → ER → AHN → AHD → GR. Sebagai contoh, scanner (yaitu AHD) untuk mengenali identifier adalah :



2.1. Membaca program sumber




```
type Text_Pos = record {posisi penunjuk karakter}
                    Row_Numb : word; {baris ke-, bisa ribuan baris/program_sumber}
                    Char_Numb : byte; {karakter ke-, maksimum 255 karakter/baris}
                  end;
var Now_Pos : Text_Pos; {posisi sekarang}
    Line : string; {baris yang sedang diproses}
    End_of_line : byte; {posisi akhir baris yang sedang diproses}
procedure Next_Character(var Ft : text); {baca karakter berikut pada program_sumber}
begin
    with Now_Pos do {coba tebak, apa itu perintah with ... do ?}
    begin
       if Char_Numb = End_of_line then
          begin
             List_Line; {menampilkan kembali baris yang telah dibaca, beserta errornya}
             Next_Line(Ft); {membaca baris berikutnya}
             Row_Numb := Row_Numb + 1;
             Char_Numb := 1
          end
       else
          Char_Numb := Char_Numb + 1;
          character := Line[Char_Numb]
    end
end;
```

```
procedure List_Line;
begin
    write{Now_Pos.Row_Numb : 3, ‘     ‘);
    writeln(Line);
    List_Error; {menampilkan kesalahan-kesalahan yang terjadi pada suatu baris}
end

procedure Next_Line(Ft : text);
begin
    readln(Ft, Line);
    End_of_line := length(Line) + 1:
    Line := Line + #32; {karakter spasi}
end;
```

2.2. Aturan Translasi

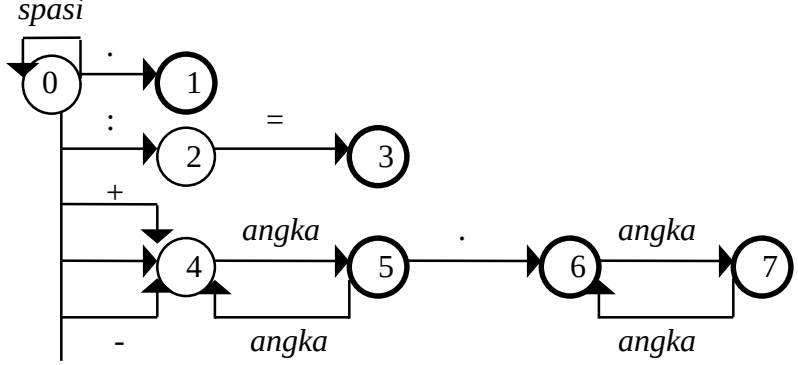
Berikut ini adalah contoh aturan translasi (*translation rule*) untuk beberapa *ekspresi regular* (ER) atau token.

token	Aturan translasi	token	Aturan translasi
.	{Current_Token(1,1)}=		{Current_Token(15,1)}
,	{Current_Token(2,1)}<>		{Current_Token(15,2)}
;	{Current_Token(3,1)}<		{Current_Token(15,3)}
:	{Current_Token(4,1)}<=		{Current_Token(15,4)}
:=	{Current_Token(12,1)}	>	{Current_Token(15,5)}
+	{Current_Token(13,1)}	>=	{Current_Token(15,6)}
-	{Current_Token(13,2)}	identifier	{Current_Token(27,Id)}
*	{Current_Token(14,1)}	$(+\mid-\mid\varepsilon)$angka$^+$	{Current_Token(28,IN)}
/	{Current_Token(14,2)}	$(+\mid-\mid\varepsilon)$angka$^+$ .angka$^+$	{Current_Token(29,RN)}

Current_Token(tipe,nilai) adalah *procedure* yang memberikan spesifikasi kepada sebuah token yang baru saja ditemukan. Argumen *tipe* adalah *kelompok token* sedangkan sedangkan argumen *nilai* merupakan nilai dari token tersebut. Tipe = 0 ditetapkan bagi *token yang tidak dikenal*.

2.3. DFA dari ER dan Tabel Transisi

Contoh DFA untuk beberapa ER di atas adalah :



DFA di atas mempunyai *tabel transisi* T(stata,karakter) sebagai berikut :

stata	k a r a k t e r						
	spasi	.	:	=	+	-	angka
0	0	1	x	x	4	4	x
1	x	x	x	x	x	x	x
2	x	x	x	3	x	x	x
3	x	x	x	x	x	x	x
4	x	x	x	x	x	x	5
5	x	6	x	x	x	x	4
6	x	x	x	x	x	x	7
7	x	x	x	x	x	x	6

Formula tabel tersebut adalah : T(i,a) = { j, jika ada transisi berlabel a dari i ke j ; x, jika tidak ada transisi berlabel a dari i }

Jika T(i,a) = x, maka ada 2 kemungkinan :
- jika stata i merupakan stata akhir berarti pada stata i telah ditemukan sebuah token
- jika stata i bukan stata akhir berarti pada stata i telah terdeteksi *token tidak dikenal*

2.4. Simulasi DFA

Simulasi DFA dimaksudkan untuk mengenali token.

```
type Token_Kind = record
                    tipe : byte;
                    nilai : byte
                  end;
var Token : array[0..Max_State] of Token_Kind;
    Found_Token : Token_Kind; {token yang ditemukan}
    Tok_Pos : Text_Pos; {posisi token dalam program sumber}
procedure Next_Token(var Ft : text); {digunakan untuk mengenali sebuah token}
var state1, state2 : shortint;
begin
    state1 := 0;
    Tok_Pos := Now_Pos;
    repeat
        state2 := Next_State(state1, character);
        if state2 <> -1 then {-1 bersesuaian dengan x pada tabel transisi}
        begin
            state1 := state2;
            Next_Character(Ft); {baca karakter berikut pada program_sumber}
            {di antaranya menghasilkan nilai baru untuk Now_Pos}
        end;
    until state2 = -1;
    Act_for_Token(state1);
end;
```

```
procedure Act_for_Token(state : shortint);
var Tok_Length : byte;
      Err : integer;
begin
     Current_Token(Token[state].tipe, Token[state].nilai);
     Tok_Length := Now_Pos.Char_Numb - Tok_Pos.Char_Numb;
     case Token[state].tipe of
          0    : Error('Token tidak dikenal!', Tok_Pos);
          27  : Id := copy(Line, Tok_Pos.Char_Num, Tok_Length);
          28  : val(copy(Line, Tok_Pos.Char_Num, Tok_Length), IN, Err);
          29  : val(copy(Line, Tok_Pos.Char_Num, Tok_Length), RN, Err);
     end
end;
```

catatan :

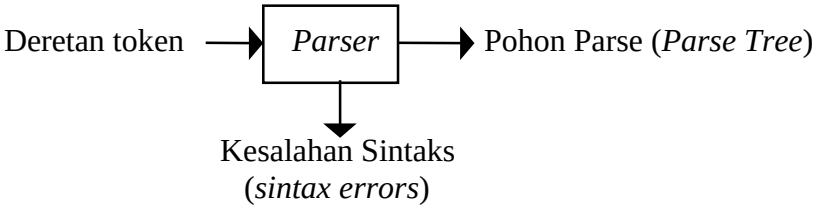
- copy(string, start, length) mengembalikan substring
- val(string_value, number_variable, error_flag) :
  jika string_value = '137' maka number_variable = 137 dan error_flag = 0
  jika string_value = 'string' maka number_variable = 137 dan error_flag $\neq$ 0
- Token.tipe $\in$ {1, 2, 3, ..., 26} dimisalkan bernilai pasti, sehingga tidak perlu penanganan lebih lanjut

```
procedure Current_Token(tipe, nilai : byte);
begin
     Found_Token.tipe := tipe;
     Found_Token.nilai := nilai;
end;
```

KOMPILASI, Catatan ke-3 : Penganalisa Sintaks (*Parser*)

1. Posisi Parser dalam Kompilator

Posisi Penganalisa Sintaks (*Parser*) dalam proses kompilasi adalah sebagai berikut :



- Deretan token : dihasilkan oleh Penganalisa Leksikal (*Scanner*)
- Pohon parse : suatu pohon dimana akarnya (*root*) adalah *simbol awal grammar* (*starting symbol*), setiap node dalam (*inner node*) adalah simbol nonterminal, dan daunnya (*leaf*) dibaca dari kiri ke kanan adalah *deretan token masukan*. Pohon parse ini dibentuk berdasarkan aturan grammar yang ditetapkan untuk *parser*.
- Kesalahan sintaks : terjadi jika pola deretan token tidak memenuhi ketentuan pola yang telah ditentukan grammar untuk *parser*.

Grammar yang dipilih untuk *scanner* adalah *Regular Grammar* (*RG*) sedangkan untuk *parser* adalah *Grammar Context Free* (*CFG*). Penting diketahui perbedaan *cara pandang RG* dengan *CFG* terhadap sebuah token yang mengalir antara *scanner* dan *parser*. Bagi RG (*scanner*) sebuah token (kecuali *reserve word*) adalah *sebuah kalimat* dimana setiap karakter pembentuk token tersebut adalah *simbol terminal*. Sebaliknya bagi CFG (*parser*) sebuah token adalah *sebuah simbol terminal* dimana sederetan tertentu token akan membentuk *sebuah kalimat*.

Latihan 1 : Berikan contoh kalimat_versi_*parser* dimana setiap simbol terminalnya adalah sebuah kalimat_versi_*scanner*. Tunjukkan secara gramatikal bahwa setiap kalimat_versi_*scanner* tersebut memang merupakan simbol-simbol terminal bagi kalimat_versi_*parser*.

2. Review Hal-hal Penting dari CFG

a. Pola umum CFG : $A \rightarrow \alpha$, $A \in V_N$, $\alpha \in (V_N \mid V_T)^*$

b. Sifat *ambigu* (*ambiguity*) :

Sebuah kalimat adalah *ambigu* jika terdapat lebih dari satu pohon sintaks yang dapat dibentuk oleh kalimat tersebut.

Secara gramatikal kalimat ambigu dihasilkan oleh *grammar ambigu* yaitu grammar yang mengandung *beberapa* produksi dengan *ruas kiri yang* sama sedangkan dua atau lebih ruas kanan-nya mempunyai *string terkiri* (*prefix*) yang sama. Contoh :

$S \rightarrow$ if E then S | if E then S else S,

dengan S : *statement* dan E : *expression*, mempunya dua produksi dengan ruas kiri sama (S) sedangkan kedua ruas kanannya dimulai dengan string sama (if E then S).

Grammar ambigu dapat diperbaiki dengan metoda *faktorisasi kiri* (*left factorization*). *Prefix* dari produksi di atas adalah *sentensial* if E then S sehingga faktorisasi akan menghasilkan :

$S \rightarrow$ if E then S T, $T \rightarrow \varepsilon$ | else S {ε : simbol hampa}

c. Sifat rekursi kiri (*left recursion*) :

Sebuah grammar dikatakan bersifat rekursi kiri jika untuk sebuah simbol nonterminal A terdapat *derivasi non hampa* $A \Rightarrow \ldots \Rightarrow A\alpha$. Produksi berbentuk $A \rightarrow A\alpha$ disebut produksi yang bersifat *immediate left recursion.*

Rekursi kiri dapat dieliminir dengan transformasi berikut :

$A \rightarrow A\alpha \mid \beta$ transformasi menjadi : $A \rightarrow \beta R,\ R \rightarrow \alpha R \mid \varepsilon$

Transformasi ini dapat diperluas sehingga :

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

bertransformasi menjadi :

$$A \rightarrow \beta_1 R \mid \beta_2 R \mid \ldots \mid \beta_n R, \quad R \rightarrow \alpha_1 R \mid \alpha_2 R \mid \ldots \mid \alpha_n R \mid \varepsilon$$

Contoh 1 : Diketahui : $E \rightarrow E + T \mid T,\ T \rightarrow T * F \mid F,\ F \rightarrow (E) \mid I$

yang jelas mengandung *immediate left recursion* untuk simbol E dan T. Transformasi menghasilkan :

$E \rightarrow TR_E,\ R_E \rightarrow +TR_E \mid \varepsilon,\quad T \rightarrow FR_T,\ R_T \rightarrow *FR_T \mid \varepsilon,\quad F \rightarrow (E) \mid I$

Prosedur transformasi di atas dituangkan dalam algoritma berikut :

Algoritma Rekursi_Kiri

1. *Rename* semua nonterminal menjadi $A_1,\ A_2, \ldots, A_n$
2. for i = 1 to n do begin

2.a. for j = 1 to i-1 do begin

ganti setiap produksi berbentuk $A_i \rightarrow A_j \gamma$

dengan produksi-produksi : $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$

dimana : $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ adalah produksi-A_j saat iterasi ini

end

2.b. eliminasi semua *immediate left recursion* produksi-A_i

end

Contoh 2 : Diketahui : $S \rightarrow Aa \mid b,\ A \rightarrow Ac \mid S\,d \mid \varepsilon$

Algoritma Rekursi_Kiri akan digunakan terhadap himpunan produksi ini.

Langkah 1 : $A_1 := S,\ A_2 := A$ sehingga produksi menjadi

$$A_1 \rightarrow A_2 a \mid b,\ A_2 \rightarrow A_2 c \mid A_1 d \mid \varepsilon$$

Saat i = 1 *inner loop* tidak dipenuhi karena (j =1) > (i-1= 0), maka program masuk ke (2.b) untuk A_1. Tetapi A_1 tidak bersifat *immediate left recursion*. Jadi saat i = 1 program tidak melakukan apapun.

Saat i = 2,

(2.a) j = 1 : ganti $A_2 \rightarrow A_1 d$ dengan $A_2 \rightarrow A_2 ad \mid bd$

(2.b) $A_2 \rightarrow A_2 c \mid A_2 ad \mid bd \mid \varepsilon$ adalah *immediate left recursion*, sehingga diperoleh transformasinya :

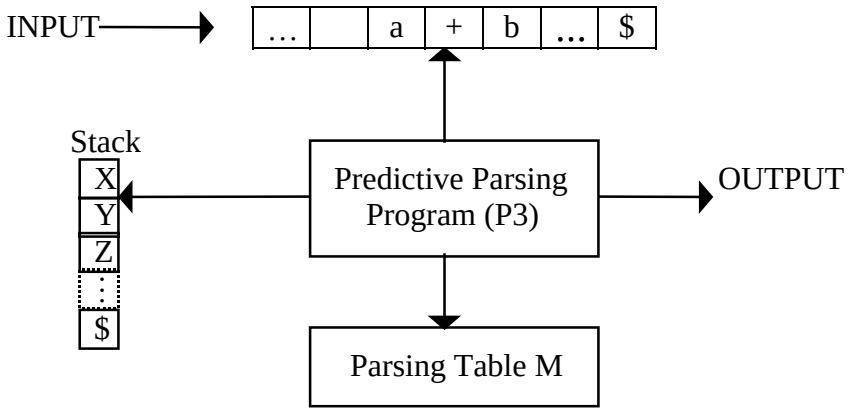
$$A_2 \rightarrow bdR_A \mid R_A,\ R_A \rightarrow c\,R_A \mid adR_A \mid \varepsilon$$

Hasilnya : $A_1 \rightarrow A_2 a \mid b,\ A_2 \rightarrow bdR_A \mid R_A,\ R_A \rightarrow c\,R_A \mid adR_A \mid \varepsilon$, atau :

$$S \rightarrow Aa \mid b,\ A \rightarrow bdR_A \mid R_A,\ R_A \rightarrow cR_A \mid adR_A \mid \varepsilon$$

3. Predictive Parser

Predictive Parser akan digunakan untuk mengimplementasikan Penganalisa Sintaks (*Parser*). Berikut ini adalah model dari *Predictive Parser*.



Input : rangkaian token dan diakhiri dengan tanda $.

Stack : berisi simbol grammar (V_N atau V_T). Pada keadaan awal stack hanya berisi $ dan S (simbol awal).

Parsing Table M : array 2 dimensi M(A,a), dimana A adalah simbol nonterminal dan a adalah simbol terminal (token) atau sibol $. Nilai M(A,a) adalah : sebuah produksi $A \rightarrow \alpha$ atau tanda-tanda kesalahan (keduanya akan dibahas kemudian)

Predictive Parsing Program (P3) : sebuah program yang mengendalikan *parser* berdasarkan nilai A dan a.

Sifat dan tanggapan P3 terhadap simbol A (pada *stack*) dan a (pada *input*) :

1. Jika A = a = $: *parser* berhenti dan memberitahukan bahwa kerja *parser* telah selesai tanpa ditemukan kesalahan sintaks.
2. Jika A = a ≠ $: *parser* akan mengeluarkan A dari *stack*, dan selanjutnya membaca token berikutnya.
3. Jika $A \in V_T$, A ≠ a : terjadi kesalahan sintaks, dan selanjutnya akan dipanggil *routine penanganan kesalahan* (*error handler*).
4. Jika $A \in V_N$: program akan membaca tabel M(A,a). Jika M(A,a) berisi produksi $A \rightarrow$ UVW maka *parser* akan mengganti A dengan WVU (yaitu dengan U berada di puncak *stack*). Jika M(A,a) berisi tanda-tanda kesalahan maka *parser* akan memanggil *Error_Handler routine*.

4. Parsing Table M

Parsing Table M dibentuk berdasarkan dua fungsi yang berhubungan dengan suatu tata bahasa. Kedua fungsi tersebut adalah *First*(X), $X \in (V_N \mid V_T)$ dan *Follow*(Y), $Y \in V_N$.

First(X) adalah himpunan *simbol terminal* yang merupakan simbol pertama dari X atau merupakan simbol pertama dari simbol-simbol yang dapat diturunkan dari X.

Follow(Y) adalah himpunan *simbol terminal* yang dapat muncul tepat di sebelah kanan Y melalui nol atau lebih derivasi.

Ketentuan selengkapnya tentang First(X) dan Follow(Y) adalah sebagai berikut.

a. First(X)

a1. Jika $X \in V_T$ maka First(X) = {X}

a2. Jika terdapat $X \rightarrow a\alpha$ maka $a \in$ First(X). Jika $X \rightarrow \varepsilon$ maka $\varepsilon \in$ First(X)

a3. Jika $X \rightarrow Y_1 Y_2 ... Y_k$ maka First(Y_1) $\subset$ First(X). Jika ternyata Y_1 dapat menderivasi ε (sehingga $\varepsilon \in$ First(Y_1)) maka First(Y_2) juga *subset* dari First(X). Jelaslah jika semua First(Y_i) mengandung ε , i = 1, 2, ..., n, maka semua elemen First(Y_i) adalah juga elemen First(X).

b. Follow(X)

b1. Jika X = S = simbol awal maka $\$ \in$ Follow(S)

b2. Jika $X \rightarrow \alpha Y \beta$, $\beta \neq \varepsilon$, maka {First(β) - {ε}} $\subset$ Follow(Y)

b3. Jika 1. $X \rightarrow \alpha Y$ atau 2. $X \rightarrow \alpha Y \beta$ dimana $\varepsilon \in$ First(β) maka Follow(X) $\subset$ Follow(Y)

Contoh :

Diketahui himpunan produksi :

1. $E \rightarrow TE'$, 2. $E' \rightarrow +TE' \mid \varepsilon$, 3. $T \rightarrow FT'$, 4. $T' \rightarrow *FT' \mid \varepsilon$, 5. $F \rightarrow (E) \mid id$

Fisrt : ◊ dari (5) dengan aturan (a2) : First(F) = {(, id}

◊ dari (3) dengan aturan (a3) : First(T) = First(F)
dari (1) dengan aturan (a3) : First(E) = Fisrt(T)
sehingga : First(E) = Fisrt(T) = First(F) = {(, id}

◊ dari(2) dengan aturan (a2) : First(E') = {+, ε}

◊ dari (4) dengan aturan (a2) : First(T') = {*, ε}

Follow : ◊ dari(1) dengan aturan (b1) : $\$ \in$ Follow(E) karena E = simbol awal,
dari (5) dengan aturan (b2) dan (a1) :) $\in$ Follow(E)
sehingga : Follow(E) = {$,)}

◊ dari(1) dengan aturan (b3.1) $X \rightarrow \alpha Y$: Follow(E) $\subset$ Follow(E')
sehingga Follow(E') = {$,)}

◊ dari(1) (dan (2)) dengan aturan (b2) : {First(E') - {ε}} = {+} $\subset$ Follow(T)
dari(1) dan aturan (b3.2) $X \rightarrow \alpha Y \beta$, $\alpha = \varepsilon$: Follow(E) = {$,)}$\subset$ Follow(T)
sehingga : Follow(T) = {$,), +}

◊ dari(3) dengan aturan(b3.1) : $X \rightarrow \alpha Y$: Follow(T) $\subset$ Follow(T')
sehingga : Follow(T') = {$,), +}

◊ dari(4) dengan aturan (b2) : {First(T') - {ε}} = {*} $\subset$ Follow(F)
dari(3) dengan aturan(b3.2) $X \rightarrow \alpha Y \beta$, $\alpha = \varepsilon$: Follow(T) $\subset$ Follow(F)
sehingga : Follow(F) = {$,), +,*}

Singkatnya : First(E) = Fisrt(T) = First(F) = {(, id}
First(E') = {+, ε}
First(T') = {*, ε}
Follow(E) = Follow(E') ={$,)}
Follow(T) = Follow(T') = {$,), +,*}
Follow(F) = {$,), +,*}

Dengan kedua fungsi First(X) dan Follow(X) di atas maka Parsing Table M disusun melalui algoritma berikut :

Algoritma Parsing_Table

1. for setiap produksi $A \rightarrow \alpha$ do begin
1a. for setiap $a \in$ First(α) do M(A, a) = $A \rightarrow \alpha$
1b. if $\varepsilon \in$ First(α) then
for setiap $b \in$ Follow(A) do M(A, b) = $A \rightarrow \alpha$
1c. if ($\varepsilon \in$ First(α)) and ($\in$ Follow(A) then M(A, $) = $A \rightarrow \alpha$
end
2. if M(A, a) = blank then M(A, a) = error

Jika algoritma di atas diterapkan kepada grammar :

1. $E \rightarrow TE'$, 2. $E' \rightarrow +TE' \mid \varepsilon$, 3. $T \rightarrow FT'$, 4. $T' \rightarrow *FT' \mid \varepsilon$, 5. $F \rightarrow (E) \mid id$

Maka :

◊ $E \rightarrow TE'$
Karena : First(TE') = Fisrt(T) = {(, id} maka menurut (1a) :
M(E, () = M(E, id) = $E \rightarrow TE'$

◊ $E' \rightarrow +TE'$
Karena : + $\in$ FIRST(+TE') maka menurut aturan (1a) : M(E', +) = $E' \rightarrow +TE'$

◊ $E' \rightarrow \varepsilon$
Karena : $\varepsilon \in$ FIRST(E') dan {), $) $\subset$ Follow(E') maka menurut aturan (1b) :
M(E',)) = M(E', $) = $E' \rightarrow \varepsilon$

◊ $T \rightarrow FT'$
Karena : First(FT') = First(F) = {(, id) maka menurut aturan (1a) :
M(T, () = M(T, id) = $T \rightarrow FT'$

◊ $T' \rightarrow *FT'$
Karena : * $\in$ First(*FT') maka menurut aturan (1a) : M(T', *) = $T' \rightarrow *FT'$

◊ $T' \rightarrow \varepsilon$
Karena : $\varepsilon \in$ First(T') dan {+,), $) $\subset$ Follow(T') maka menurut aturan (1b) :
M(T', +) = M(T',)) = M(T', $) = $T' \rightarrow \varepsilon$

◊ $F \rightarrow id$
Karena : id $\in$ First(F) maka menurut aturan (1a) : M(F, id) = $F \rightarrow id$

◊ $F \rightarrow (E)$
Karena : ($\in$ First(F) maka menurut aturan (1a) : M(F, () = $F \rightarrow (E)$

Akhirnya diperoleh tabel berikut :

Non Terminal	**Simbol Input**					
	id	+	*	(	)	$
E	$E \rightarrow TE'$	error	error	$E \rightarrow TE'$	error	error
E'	error	$E' \rightarrow +TE'$	error	error	$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$	error	error	$T \rightarrow FT'$	error	error
T'	error	$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$	error	$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$	error	error	$F \rightarrow (E)$	error	error

Berikut ini ditunjukkan tabulasi aksi yang dilakukan *predictive parser* berdasarkan *parser table M* di atas terhadap rangkaian token : id + id * id :

Stack	Input	Output	Keterangan
$E	id + id * id $		E = simbol awal, M(E, id) = E → TE' ganti E dengan TE', T di *top of stack*
$E'T	id + id * id $	E → TE'	M(T, id) = T→ FT' (ganti T dengan FT')
$E'T'F	id + id * id $	T→ FT'	M(F, id) = F → id (ganti F dengan id)
$E'T'id	id + id * id $	F → id	*top of stack = left most input* : *drop them !*
$E'T'	+ id * id $		M(T', +) = T'→ ε (ganti T' dengan ε)
$E'	+ id * id $	T'→ ε	M(E', +) = E'→ +TE' (ganti E' dg. TE')
$E'T+	+ id * id $	E'→ +TE'	*top of stack = left most input* : *drop them !*
$E'T	id * id $		M(T, id) = T → FT' (ganti T dengan FT')
$E'T'F	id * id $	T → FT'	M(F, id) = F → id (ganti F dengan id)
$E'T'id	id * id $	F → id	*top of stack = left most input* : *drop them !*
$E'T'	* id $		M(T', *) = T'→ *FT' (ganti T' dg. *FT')
$E'T'F*	* id $	T'→ ε	*top of stack = left most input* : *drop them !*
$E'T'F	id $		M(F, id) = F → id (ganti F dengan id)
$E'T'id	id $	F → id	*top of stack = left most input* : *drop them !*
$E'T'	$		M(T', $) = T'→ ε (ganti T' dengan ε)
$E'	$	T'→ ε	M(E', $) = E'→ ε (ganti E' dengan ε)
$	$	E'→ ε	*top of stack = left most input = $: finish !!* rangkaian token : id + id * id sesuai sintaks

KOMPILASI, Catatan ke-4 : Penganalisa Sintaks (*Parser*) (*lanjutan*)

5. Grammar LL(1)

Grammar yang *bagus* adalah grammar yang mempunyai nilai tunggal untuk setiap sel M(A,a) pada *Parsing Table*. Tetapi sayangnya tidak semua grammar bersifat *bagus*. Untuk mengetahui syarat-sayarat grammar yang *bagus* perhatikan tiga contoh grammar berikut, ketiganya mengandung dua produksi dengan pola $A \rightarrow \alpha \mid \beta$:

1. Q1 = {$A \rightarrow aBc \mid aCb, B \rightarrow b, C \rightarrow c$}
   {$A \rightarrow \alpha \mid \beta \Leftrightarrow A \rightarrow aBc \mid aCb$}
   a. Dari : $A \rightarrow aBc$, maka First($\alpha = aBc$) = {a} dan M(A,a) = $A \rightarrow aBc$
   b. Dari : $A \rightarrow aCb$, maka First($\beta = aCb$) = {a} dan M(A,a) = $A \rightarrow aCb$
   Dari (a) dan (b) diperoleh : M(A,a) = { $A \rightarrow aBc$, $A \rightarrow aCb$}
   atau :

	a
A	$A \rightarrow aBc$ $A \rightarrow aCb$

2. Q2 = {$S \rightarrow A \mid B$, $A \rightarrow a \mid \varepsilon$, $B \rightarrow b \mid \varepsilon$ }
   {$A \rightarrow \alpha \mid \beta \Leftrightarrow S \rightarrow A \mid B$}
   a. Dari $A \rightarrow \varepsilon$, maka $\varepsilon \in$ First(A). Karena $S \rightarrow A$ harus diselidiki Follow(S).
   Karena S adalah simbol awal, maka $\$ \in$ Follow(S)
   Dari keduanya diperoleh : M(S, $) = $S \rightarrow A$
   b. Dari $B \rightarrow \varepsilon$, maka $\varepsilon \in$ First(B). Karena $S \rightarrow B$ harus diselidiki Follow(S).
   Karena S adalah simbol awal, maka $\$ \in$ Follow(S)
   Dari keduanya diperoleh : M(S, $) = $S \rightarrow B$
   Dari (a) dan (b) diperoleh : M(S, $) = { $S \rightarrow A$, $S \rightarrow B$}
   atau :

	$
S	$S \rightarrow A$ $S \rightarrow B$

3. Q3 = {$S \rightarrow iEtSS' \mid a$, $S' \rightarrow eS \mid \varepsilon$, $E \rightarrow b$}
   {$A \rightarrow \alpha \mid \beta \Leftrightarrow S' \rightarrow eS \mid \varepsilon$ }
   a. Dari : $S' \rightarrow eS$, maka $e \in$ First(S') sehingga M(S', e) = $S' \rightarrow eS$
   b. Dari $S' \rightarrow \varepsilon$, maka $\varepsilon \in$ First(ε), selanjutnya harus diteliti Follow(S').
   Dari : $S' \rightarrow eS$, maka Follow(S') $\subset$ Follow (S)
   dari : $S \rightarrow iEtSS'$ (lihat suku SS'), maka Follow(S) = Fisrt{S'} - {ε} = {e}
   Dari keduanya diperoleh : M(S', e) = $S' \rightarrow \varepsilon$
   Dari (a) dan (b) diperoleh : M(S', e) = { $S' \rightarrow eS$, $S' \rightarrow \varepsilon$)
   atau :

	$
S	$S \rightarrow eS$ $S \rightarrow \varepsilon$

Sekarang kita analisa mengapa pada ketiga contoh di atas M(A, a) tidak bernilai tunggal. Sifat-sifat produksi pada masing-masing contoh (yaitu produksi yang berpola $A \rightarrow \alpha \mid \beta$) adalah :

1. $A \rightarrow aBc \mid aCb$: mempunyai satu sifat : First(aBc) = Fisrt(aCb) = {a}
2. $S \rightarrow A \mid B$: mempunyai satu sifat : A dan B bisa menurunkan simbol ε
3. $S' \rightarrow eS \mid \varepsilon$: mempunyai dua sifat : sifat pertama adalah ε bisa menurunkan simbol ε , dan sifat kedua adalah $e \in$ First(eS) dan juga $e \in$ Follow(S')

Agar sebuah grammar selalu mempunyai nilai M(A,a) yang tunggal msaka tentu saja grammar tersebut tidak boleh memiliki sifat-sifat yang dimiliki oleh ketiga contoh gammar di atas. Secara formal grammar yang *bagus* tersebut harus bersifat :

Untuk setiap produksi berbentuk $A \rightarrow \alpha \mid \beta$ maka :

1. $First(\alpha) \cap First(\beta) = \{\}$
2. If $\alpha \Rightarrow \cdots \Rightarrow \varepsilon$ then $\beta \Rightarrow \cdots \Rightarrow not(\varepsilon)$
3. If $\alpha \Rightarrow \cdots \Rightarrow \varepsilon$ and $a \in First(\beta)$ then $a \notin Follow(A)$

Grammar yang mempunyai ketiga sifat di atas dinamakan *grammar LL(1),* singkatan dari *scanning the input from <u>Left</u> to right for producing <u>Leftmost</u> derivation using <u>1 (one)</u> input symbol of lookahead at each step to making parsing action decision*.

6. Struktur Data Untuk Implementasi Parsing Table

Tabel Parsing banyak memuat sel-sel kosong. Oleh karena itu perlu ditetapkan struktur data yang tepat untuk mengimplementasikannya agar penggunaan *space* (RAM) menjadi optimal. Struktur data yang tepat adalah *linked list*. Dengan menggunakan *linked list* maka setiap simbol nonterminal dihubungkan dengan hanya sel-sel yang terkait dan tidak kosong. Untuk tujuan ini maka dilakukan *penomoran* terhadap tiga komponen grammar, yaitu : simbol nonterminal A, simbol terminal (token) a, dan produksi yang merupakan nilai dari M(A, a). Selanjutnya yang dilibatkan dalam *linked list* ini hanyalah nomor-nomor tersebut.Berikut ini adalah pendefinisian tipe data untuk isi sel tabel parsing.

```
Parsing_Table_Entry_Node = ^Parsing_Table_Entry_Node
Parsing_Table_Entry_Node = record
                               Terminal : byte
                               Production : byte
                               Next : Parsing_Table_Entry_Node
                           end;
```

Berikut ini contoh penomoran ketiga komponen grammar yang diterapkan untuk grammar contoh yang lalu :

<u>Penomoran produksi :</u>

Q = {1. $E \rightarrow TE'$ 2. $E' \rightarrow +TE'$ 3. $E' \rightarrow \varepsilon$ 4. $T \rightarrow FT'$
5. $T' \rightarrow *FT'$ 6. $T' \rightarrow \varepsilon$ 7. $F \rightarrow (E)$ 8. $F \rightarrow id$ }

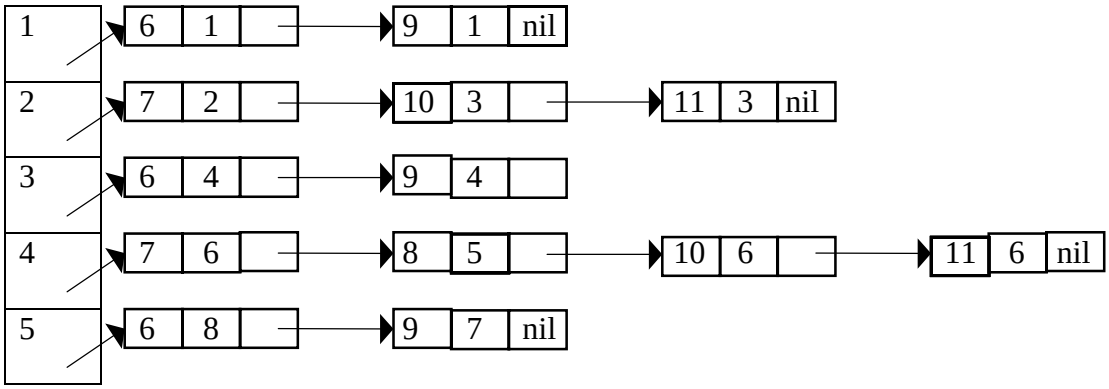
<u>Penomoran simbol nonterminal :</u>

1. E 2. E' 3. T 4. T' 5. F

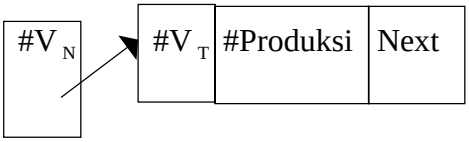
<u>Penomoran simbol terminal (berlanjut dari penomoran simbol nonterminal) :</u>

6. id 7. + 8. * 9. (10.) 11. $ 12. ε

Dengan implementasi *linked list* serta penomoran produksi seperti di atas maka ilustrasi implementasi tabel parsing yang lalu adalah :



Keterangan gambar :



Tabel parsing disajikan dengan variabel berikut :

```
Parsing_Table : array[1.. 5] of Parsing_Table_Entri_Node
```

Misalnya, dalam suatu tahap parsing, *top of stack* adalah sebuah simbol nonterminal yang bernomor n, sedangkan token input yang sedang ditunjuk adalah token bernomor m, maka *routine* Parsing_Table_Cell berikut ini digunakan untuk membaca isi sel pada baris ke-n dan kolom ke-m pada tabel parsing :

```
function Parsing_Table_Cell (n, m : byte) : byte;
var  Cell : Parsing_Table_Entry_Node;
begin
   Cell := Parsing_Table(n);
   while (Cell <> nil) and (Cell^.Terminal <> m) do Cell := Cell^.Next;
   if Cell <> nil then
       Parsing_Table_Cell := Cell^.Production
   else Parsing_Table_Cell := error;
end
```

Jika *function* di atas dipanggil dalam, misalnya : x := Parsing_Table_Cell(4, 8) maka x akan bernilai 5. Karena 4 adalah nomor untuk V_N = T' , 8 adalah nomor untuk V_T = *, dan 5 adalah nomor untuk produksi = T' → *FT', maka x := Parsing_Table_Cell(4,10) ekuivalen dengan M(T', *) = T' → *FT'.

7. Pemulihan Kesalahan (*Error Recovery*)

Jika dilakukan pemanggilan x := Parsing_Table_Cell(3, 7) maka diperoleh x := *nil* yang ekuivalen dengan M(T, +) = *error*. Inilah yang dinamakan *syntax error*. Akibat terdeteksinya *syntax error* maka kompilator akan menghentikan kegiatan parsingnya. Adalah sangat mungkin, terutama bagi pemula, untuk menemukan kejadian ini (yaitu kejadian kompilator berhenti bekerja). Jika setiap *syntax error* menyebabkan kompilator berhenti maka tentu saja hal ini tidak efisien. Untuk itu harus dipikirkan bagaimana supaya hal ini tidak terjadi. Artinya : (1) kompilator terus bekerja walaupun ditemukan *syntax error*, atau dengan kata lain : (2) kompilator harus menemukan sebanyak mungkin *syntax error* yang mungkin terdapat dalam *source program*. *Stategi pemulihan kesalahan* adalah suatu upaya agar kedua hal di atas dapat terlaksana.

Terdapat tiga hal yang harus dipenuhi dalam merancang suatu strategi pemulihan kesalahan; ketiganya yaitu :

1. Melaporkan adanya kesalahan dengan jelas dan akurat. Contoh laporan adalah posisi baris dan posisi token pada baris tersebut, sedangkan *syntax error* yang terjadi adalah *unballanced parenthesis* (kurung buka dan kurung tutup yang tidak seimbang)
2. Dapat mendeteksi *syntax error* berikutnya segera setelah suatu *syntax error* ditemukan
3. Tidak boleh memperlambat proses parsing jika dilakukan terhadap *source program* yang benar.

Empat strategi pemulihan kesalahan yang sering digunakan adalah :

1. mode panik (*panic mode*)
2. tingkat kombinasi kata-kata (*phrase level*)
3. produksi-produksi kesalahan (*error production*)
4. pembetulan global (*global correction*)

7.1. Mode panik

Strategi ini adalah metoda yang paling sederhana. Jika sebuah *syntax error* ditemukan, yaitu jika M(A, a) = *error* untuk suatu token a, maka *parser* akan melewatkan (*skip*) token a dan token-token berikutnya sampai ditemukan sebuah *token penyelaras* (*syncronizing token*). Pengertian *melewatkan sebuah token* adalah *parser* membaca token tersebut tetapi tanpa *menganalisanya* yaitu tanpa membandingkan dengan simbol *top of stack*. Token-token penyelaras adalah token-token yang mempunyai *peran yang jelas* dalam *source program*, misalnya sebagai pembatas (*delimiter*). Dua contoh token penyelaras dalam Pascal yang berperan sebagai pembatas adalah : titik koma (mengakhiri sebuah *statement*), end (mengakhiri sebuah *blok statement*). Tugas perancang kompilator adalah menentukan himpunan token penyelaras ini. Mode panik ini tidak efektif jika sebuah baris *statement* mengandung beberapa *syntax error*.

7.2. Tingkat kombinasi kata-kata

Jika sebuah *syntax error* ditemukan maka *parser* akan melakukan *pembetulan lokal* kepada *prefix* dari sisa input dengan suatu string sehingga proses *parsing* bisa dilanjutkan. Contoh pembetulan yang lazim dilakukan adalah : mengganti koma dengan titik koma, menghapus titik koma yang berlebihan (seperti titik koma sebelum token *else* di dalam Pascal), atau mengisi titik koma yang hilang. Hati-hati dengan metode karena pembetulan lokal dapat menyebabkan kesalahan terhadap string berikutnya yang sebenarnya sudah sesuai sintaks.

7.3. Produksi-produksi kesalahan

Yang dimaksud dengan produksi-produksi kesalahan adalah sekumpulan produksi yang akan menghasilkan konstruksi yang salah (*wrong parse tree*, yaitu *parse tree* yang meng-implementasikan sebuah *statement* yang salah). Jika kita *telah mengetahui hal-hal yang dapat menyebabkan syntax error* maka kita dapat memperluas *grammar* yang mendasari *parser* dengan menambahkan produksi-produksi kesalahan. Jika produksi-produksi kesalahan ini dipakai oleh *parser* maka selanjutnya dapat dilakukan diagnosa atas *wrong parse tree* tersebut. Selanjutnya lakukan penyempurnaan atas grammar.

7.4. Pembetulan global

Misalkan x adalah string input yang salah sedangkan y adalah string input yang benar. Yang dimaksud dengan pembetulan global adalah sejumlah aksi penyisipan, penghapu-san, atau perubahan terhadap suatu token di dalam string x sedemikian rupa sehingga x berubah menjadi y. Yang perlu diperhatikan adalah bahwa aksi penyisipan, penghapusan, atau perubahan ini harus seminimal mungkin.

8. Pemulihan Kesalahan Pada *Predictive Parser* Dengan Mode Panik

Hal utama yang harus dilakukan dalam mode panik adalah *menentukan himpunan token penyelaras H*. Berikut ini adalah sebuah strategi pemulihan kesalahan pada *predictive parser* dengan menggunakan *mode panik*.

1. Salah satu jenis token yang dapat dipilih sebagai token anggota H adalah token pembatas (*delimiter*). Token pembatas dari string input dalam *predictive parser* adalah simbol $ yang merupakan elemen dari Follow(X), dimana X adalah simbol non-terminal. Kita dapat memasukkan semua elemen Follow(X) ini (termasuk simbol terminal yang bukan $) sebagai anggota H. Jika beberapa token dilewatkan (*skip*) sampai ditemukan salah satu elemen Follow(X) serta simbol X di-*pop* dari *top of stack* maka kemungkinan besar proses parsing akan berlanjut. Jika setelah pop ternyata isi *top of stack* adalah $ maka token input harus diabaikan walaupun token itu elemen H.
2. Ternyata penetapan Follow(X) saja sebagai token penyelaras tidak cukup. Perhatikan contoh berikut. Titik koma adalah token pembatas, jadi titik koma dapat dimasukkan ke dalam H. Di lain pihak token yang mengawali sebuah *statement* belum tentu merupakan elemen dari H. Sekarang misalnya sebuah titik koma di akhir sebuah *ekspresi matematis* tidak tertulis dalam *source program*. Ini akan menyebabkan *parser* akan melewatkan token-token berikutnya *termasuk token yang mengawali statement sesudah ekspresi matematis* tersebut. Untuk itu maka *token yang mengawali setiap statement harus dimasukkan ke dalam H*.
3. Kita juga dapat memasukkan token-token dalam First(X) sebagai elemen H.
4. Jika $M(X, a) = X \rightarrow \varepsilon$, maka *parsing* sebaiknya dihentikan. Hal ini akan mengurangi jumlah nonterminal yang harus diperhatikan dalam proses pemulihan kesalahan.
5. Jika suatu simbol terminal pada *top of stack* tidak sesuai dengan token yang sedang diamati maka tanggapan yang paling sederhana adalah melakukan *pop* terhadap simbol terminal tersebut dan *parsing* dapat dilanjutkan. Ini sama saja dengan menetap-kan bahwa *himpunan token penyelaras dari sebuah token adalah semua token lainnya*.

Berikut ini adalah contoh pemulihan kesalahan pada *predivtive parser* dengan mode panik. Dimisalkan bahwa himpunan token penyelaras adalah semua elemen Follow(X). Jika b ∈ Follow(X) dan M(X, a) = error maka M(X, a) = synch. Dengan cara ini maka simbol tabel yang lalu menjadi :

Non Terminal	**Simbol Input**					
	id	+	*	(	)	$
E	E → TE'			E → TE'	synch	synch
E'		E'→ +TE'			E'→ ε	E'→ ε
T	T→ FT'	synch		T→ FT'	synch	synch
T'		T' → ε	T'→ *FT'		T'→ ε	T'→ ε
F	F → id	synch	synch	F→ (E)	synch	synch

Tabel di atas akan digunakan untuk melakukan *parsing* string masukan x =)id*+id.

Stack	**Input**	**Keterangan**
$E	)id *+ id $	**M(E,)) = synch, *pop* E**. Karena *top of stack* = $ maka token) harus diabaikan, *parsing* dimulai lagi dari awal
$E	id *+ id $	M(E,id) = E → TE', ganti E dengan TE', *top of stack* = T
$E'T	id *+ id $	M(T,id) = T→ FT', ganti T dengan FT', *top of stack* = F
$E'T'F	id *+ id $	M(F,id) = F → id, ganti F dengan id
$E'T'id	id *+ id $	*drop both* id's
$E'T'	*+ id $	M(T',*) = T'→ *FT', ganti T' dengan *FT', *top of stack* = *
$E'T'F*	*+ id $	*drop both* *'s
$E'T'F	+ id $	**M(F,*) = synch, *pop* F**
$E'T'	+ id $	M(T',+) = T'→ ε , ganti T' dengan ε
$E'	+ id $	M(E',+) = E'→ +TE', ganti E' dg. +TE', *top of stack* = +
$E'T+	+ id $	*drop both* +'s
$E'T	id $	M(T,id) = T → FT', ganti E' dengan FT', *top of stack* = F
$E'T'F	id $	M(F,id) = F → id, ganti F dengan id
$E'T'id	id $	*drop both* id's
$E'T'	$	M(T',$) = T'→ ε , ganti T' dengan ε
$E'	$	M(E',$) = E'→ ε , ganti T' dengan ε
$	$	*finish*

KOMPILASI, Catatan ke-5 : Penganalisa Semantik

1. Tabel Simbol

Pada catatan yang lalu (hal. 2) telah disinggung tentang batasan-batasan yang ditetapkan untuk sebuah token ataupun ekspresi. Batasan-batasan tersebut misalnya : panjang maksimum sebuah token pengenal (*identifier*) atau sebuah ekspresi tunggal, jangkauan bilangan bulat, tipe operan dalam ekspresi aritmatika, dan apakah suatu pengenal sudah/belum dideklarasikan (atau bahkan dideklarasikan ganda), bagaimana data suatu pengenal dideklarasikan, dan dimana data tersebut disimpan.

Referensi atau jawaban atas batasan-batasan di atas umumnya harus diperoleh melalui sebuah *tabel simbol* (*symbol table*). Sebuah simbol tidak lain adalah sebuah pengenal. Setiap entri simbol tabel terdiri dari pasangan (nama_pengenal, informasi_pengenal). Sebuah nama_pengenal bisa merupakan : nama prosedur, nama peubah, nama konstanta, dan sebagainya. Informasi_pengenal yang perlu disertakan dalam tabel misalnya adalah :

a. untai karakter (token) yang menyajikan nama_pengenal
b. atribut-atribut nama_pengenal, misalnya :
   b.1. peran dari nama (misalnya sebagai : konstanta, label, peubah, atau prosedur)
   b.2. tipe data (misalnya : *integer*, *real*, *character*)
c. parameter nama_pengenal (misalnya dimensi array, batas atas/bawah array)
d. alamat dan besar RAM yang dialokasikan untuk nama_pengenal

Sebuah tabel simbol harus dibuat sedemikian rupa sehingga memiliki kemampuan-kemampuan berikut :

a. dapat menentukan apakah suatu nama_pengenal terdapat dalam tabel
b. dapat memasukkan nama_pengenal baru ke dalam tabel
c. dapat memasukkan informasi baru tentang nama_pengenal baru ke dalam tabel
d. dapat mengakses informasi mengenai suatu nama_pengenal
e. menghapus satu atau beberapa nama_pengenal dari tabel (membebaskan RAM)

2. Hash Table

Struktur data yang layak digunakan untuk mengimplementasikan tabel simbol adalah *linked list* terutama dalam tiga manifetasinya : *linear linked list*, *tree*, atau *hash*. Dari ketiga manifestasi tersebut *hash* (atau sering disebut *hash tabel*) adalah yang paling layak untuk dipilih. Sebuah *hash table* terdiri dari beberapa *bucket*. Setiap *bucket* terdiri dari beberapa *slot*. Setiap *bucket* diberi nomor, dimulai dari nomor 0 (nol). Berikut ini adalah contoh sebuah *hash table* yang terdiri dari *26 bucket* (b = 26) dan *2 slot/bucket* (s = 2) :

Dalam pembicaraan tentang *hash table* dikenal istilah *hash function f*. Fungsi f ini digunakan untuk memasukkan nama_pengenal ke dalam tabel simbol. Salah satu contoh ungkapan fungsi f adalah :

f(X) = kode dari karakter pertama dari X

dimana X adalah nama_pengenal. Misalkan terdapat 5 nama_pengenal : *aster*, *anyelir*, *anggrek*, *bakung*, dan *cempaka*. Misalkan pula aturan pengkodean adalah : a = 0, b = 1, dan d = 2. Tabel simbol yang dihasilkan adalah :

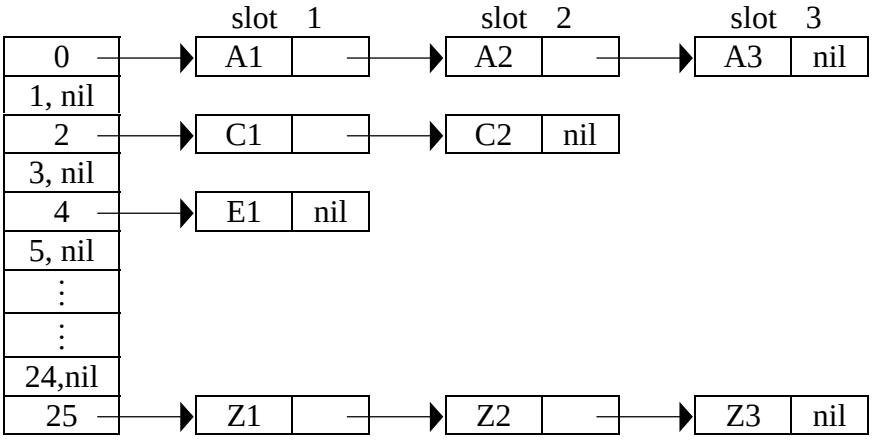
	slot-2	slot-1
0	aster	anyelir
1	bakung	
2	cempaka	
⋮		
24		
25		

Perhatikan bahwa nama_pengenal *aster*, *anyelir*, dan *anggrek* mempunyai nilai f(X) yang sama. Karena setiap *bucket* hanya terdiri dari 2 slot maka nama_pengenal *anggrek* tidak dapat dimasukkan ke dalam tabel simbol. Jika sebuah nama_pengenal dipetakan ke dalam *bucket* yang sudah penuh akan terjadi *overflow*. *Collision* terjadi jika dua nama_pengenal berbeda mempunyai nilai fungsi yang sama (artinya dipetakan ke *bucket* yang sama). Jika s = 1 maka *collision* dan *overflow* terjadi bersamaan.

Fungsi hash yang lebih baik dan sering dipakai adalah *uniform hash function*. Dikatakan *uniform* karena dengan fungsi ini semua nama_pengenal mempunyai peluang yang sama untuk dipetakan ke setiap *bucket*. Salah satu bentuk *unform hash function* ini adalah :

f(X) = kode(X) mod M

dengan M adalah bilangan prima yang lebih besar dari 20. Dengan menggunakan fungsi f ini maka ukuran *bucket* adalah b = M dengan s tertentu. Bagaimanapun dengan s tertentu dapat juga terjadi *overflow*. Untuk mengatasi hal ini *chaining hash table* adalah solusi yang tepat. Dengan *chaining* ini maka ukuran *slot* setiap *bucket* dapat bervariasi. Berikut ini adalah bentuk *chaining hash table* :



3. Table Simbol dengan Hash Table

Akan dirancang sebuah tabel simbol untuk menyimpan semua nama_pengenal. Misalkan informasi setiap nama_pengenal yang akan dicantumkan adalah peran (label, peubah, konstanta, procedure, dan function), tipe peubah dan konstanta mempunyai tipe (integer, real, dan character), dan alokasi setiap nama_pengenal di dalam RAM. Jika ukuran tabel tersebut adalah b = Max_Buck, maka jangkauan alamat *bucket* pada tabel adalah :

Buck_Range = 0..Max_Buc-1

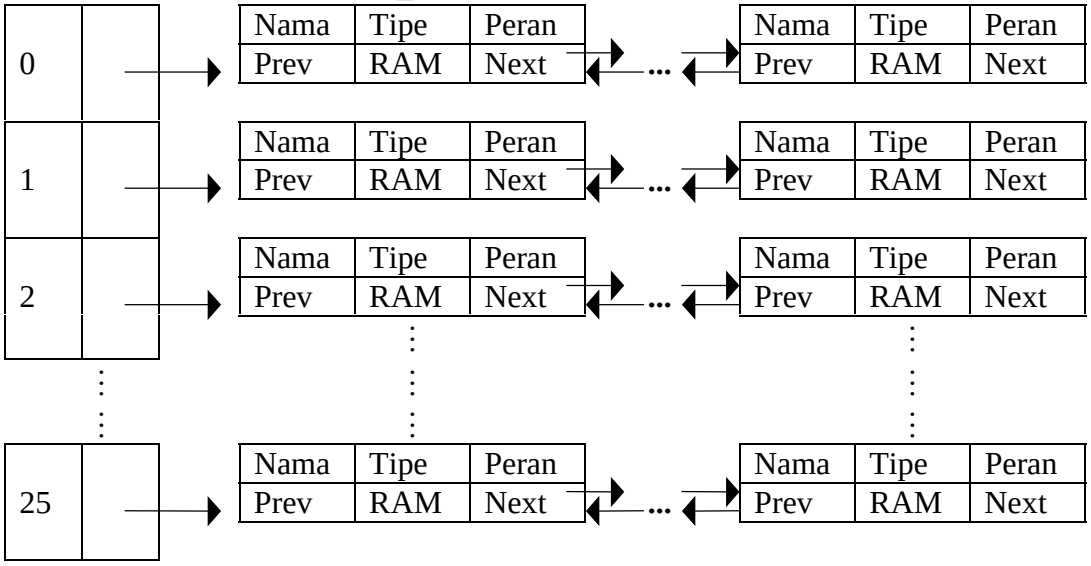
Tabel lambang dideklarasikan sebagai :

```
Symbol_Tabel : array [Buck_Range] of Id_Node;
```

dimana :

```
Id_Node = ^Id_Rec;
Id_Rec = record
            Nama : string[8];
            Peran : [label, peubah, konstanta, procedure, function];
            Type : [integer, real, character];
            RAM : longint;
            Prev, Next : Id_Node
         end
```

Ilustrasi tabel simbol untuk Max_Buc = 26 adalah :



Untuk menentukan alamat *bucket* nama_pengenal X maka fungsi f-nya adalah :

f(X) = ASCII(X) mod Max_Buc

ASCII(X) menghitung jumlah kode semua karakter di dalam nama X. Perhitungan ini dilakukan oleh *function* berikut :

```
function Hashing(Ident : string) : Buck_Range;
var Ascii_X : word;  I : byte;
begin
    Ascii_X := 0;
    for I := 1 to Length(Ident) do Ascii_X := Ascii_X + ord(Ident[ I ]);
    Hashing := Ascii_X mod Max_Buck;
end;
```

Selanjutnya, *routine-routine* berikut melakukan melakukan beberapa proses.

1. Melacak apakah sebuah nama_pengenal terdapat di dalam tabel simbol atau tidak

```
function Id_Search(Ident : string;  var Found_Id : Id_Node) : boolean;
begin
    Found_Id := Symbol_Table[Hashing(Ident)];
    while (Found_Id <> nil) and (Found_Id^.Name <> Ident) do
        Found_Id := Found_Id^.Next;
    Id_Search := (Found_Id <> nil);
end;
```

2. Memasukkan sebuah nama_pengenal ke dalam tabel simbol

Dalam proses ini terdapat 2 asumsi :

a. pemasukan nama belum disertai nilai-nilai informasinya

b. simpul nama_pengenal baru selalu diletakkan sebagai simpul pertama pada *linked list bucket* yang bersangkutan

```
procedure New_Identifier(Ident : string; var New_Id : Id_Node);
var Loc : byte;
begin
    new(New_Id);
    with New_Id^ do begin
        Name := Ident;
        Tipe := nil;
        Peran := nil;
        RAM := nil;
        Prev := nil;
        Next := nil;
    end;
    Loc := Hashing(Ident);
    if Symbol_Table[Loc] = nil then Symbol_Table[Loc] := New_Id
    else begin
        New_Id^.Next := Symbol_Table[Loc];
        Symbol_Table[Loc]^.Prev := New_Id;
        Symbol_Table[Loc] := New_Id;
    end;
end;
```

KOMPILASI, Catatan ke-6 : Penganalisa Semantik (*lanjutan*)

4. Pengelolaan RAM

Pada saat sebuah program sedang berjalan, RAM dialokasikan ke sejumlah variable untuk menyimpan data. Jika tidak digunakan lagi, RAM yang dialokasikan tersebut harus dibebaskan kembali agar dapat digunakan untuk keperluan lain. RAM yang sedang dialokasikan dikelola dengan menggunakan struktur data *linked list*. Setiap simpul *linked list* menyimpan informasi *alamat* dan *ukuran* RAM yang dialokasikan.

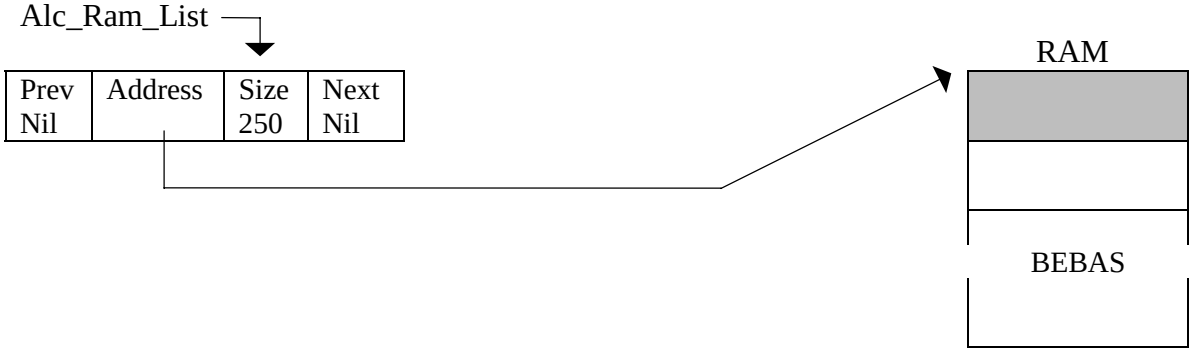
```
RAM_Node = ^RAM_Rec
RAM_Rec = record
                Address : pointer; {Alamat awal}
                Size : word;
                Prev, Next : Ram_Node;
          End;
```

RAM yang sedang dikelola ditetapkan (*assign*) pada variable Alc_RAM_List. Berikut ini adalah *function* Allocating_RAM yang mengalokasikan RAM sebesar RAM_Size. *Function* ini adalah boolean. Satuan ukuran RAM dinyatakan dengan Unit_Size. Jika RAM yang tersedia cukup untuk dialokasikan sebesar RAM_Size maka *function* akan mengembalikan (*return*) nilai *true* dan sebaliknya *false*. Variabel Alc_RAM_List selalu menunjuk pada simpul pengalokasian RAM yang terakhir dibuat.

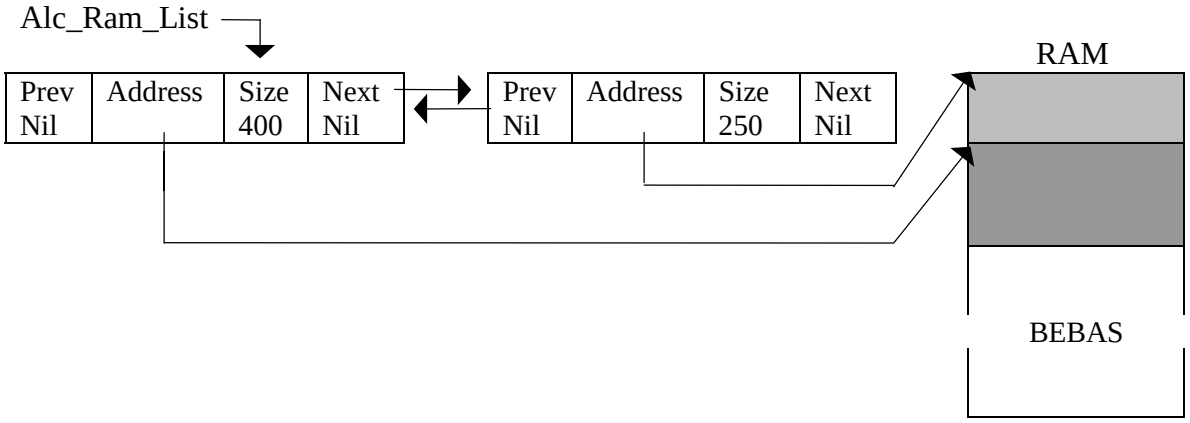
```
Function Allocating_RAM(RAM_Size : word) : boolean;
var Temp : RAM_Node;
begin
   if maxavail >= RAM_Size*Unit_Size then begin
        new(Temp);
        with Temp^ do begin
             Size := Ram_Size*Unit_Size;
             getmem(Address, Size);
             Next := Nil;
             Prev := Nil;
        end;
        if Alc_Ram_List = nil then Alc_RAM_List := Temp
        else begin
             Temp^.Next := Alc_RAM_List;
             Alc_RAM_List^.Prev := Temp;
             Alc_Ram_List := Temp;
        end;
        Allocating_RAM := true;
   end
   else Allocating_RAM := false;
end;
```

Ilustrasi pengalokasian RAM ditunjukkan dengan gambar di bawah ini. Dalam ilustrasi tersebut dimisalkan telah dialokasikan RAM sebesar 250 dan akan dialokasikan RAM sebesar 400 lagi. Perhatikan perubahan posisi pointer Alc_RAm_List yang terjadi.

Sebelum Allocating_RAM(400) :



Sesudah Allocating_RAM(400) :

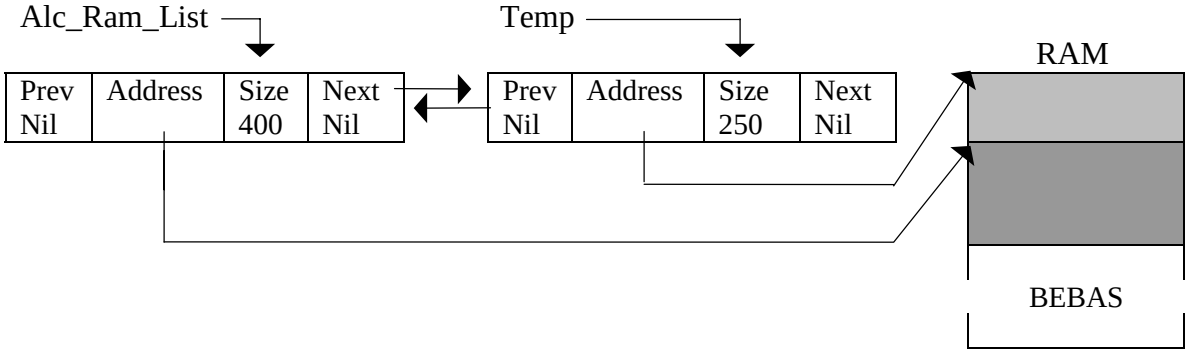


Pembebasan kembali RAM dilakukan dengan *procedure* Deallocating_RAM. RAM yang akan dibebaskan ditetapkan (*assign*) pada variable Allocated. Simpul Allocated ini selanjutnya akan dihapus dari *linked list* RAM.

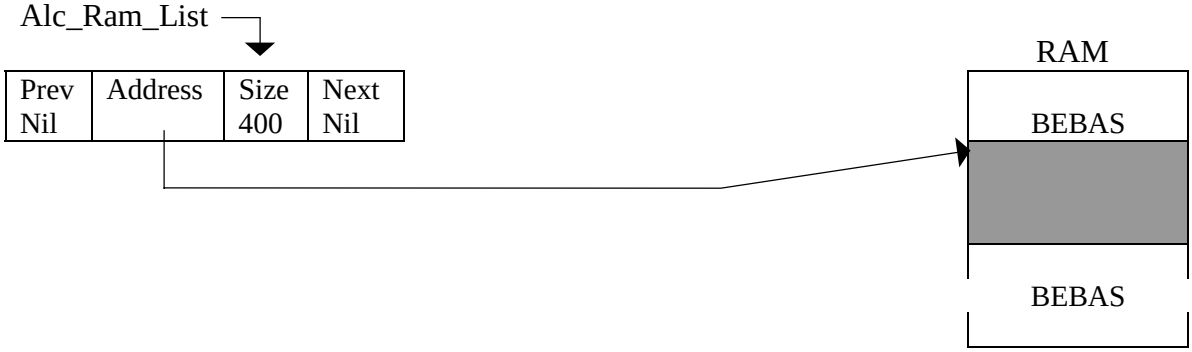
```
procedure Deallocating_RAM(Allocated : RAM_Node);
var Temp : RAM_Node;
begin
    Temp := Allocated;
    with Temp^ do begin
          freemem(Address, Size);
          Address := Nil;
          if Prev = Nil then Alc_RAM_List := Next
          else Prev^.Next := Next;
          if Next <> Nil then then Next^.Prev := Prev;
          Next := Nil;
          Prev := Nil;
    end;
    dispose(Temp);
end;
```

Ilustrasi pembebasan RAM ditunjukkan dengan gambar di bawah ini. Dalam ilustrasi ini dimisalkan bahwa simpul yang akan dibebaskan adalah simpul berukuran 250.

Sebelum Deallocating_RAM(Temp) :



Sesudah Deallocating_RAM(Temp) :



Inisialisasi pengelolaan RAM dilakukan dengan terlebih dahulu menentukan harga Unit_Size dan menyatakan bahwa *linked_list* Alc_RAM_List masih kosong. *Procedure* RAM_Management_Init berikut ini melakukan inisialisasi pengelolaan RAM. Dalam *procedure* tersebut dimisalkan bahwa data yang dialokasikan adalah data bertipe real.

```
procedure RAM_Management_Init;
begin
    Unit_Size := sizeof(real);
    Alc_Ram_List := Nil;
end;
```

Jika *linked list* pengelolaan RAM sudah tidak digunakan lagi maka *linked list* ini harus dihapus sekaligus RAM-nya harus dibebaskan. *Procedure* berikut melakukan penghapusan *linked list* pengelolaan RAM secara keseluruhan.

```
procedure Destroying_RAM_Management;
var Temp : RAM_Node;
begin
    while Alc_Ram_List <> Nil do begin
          Temp := Alc_Ram_list;
          Deallocating_RAM(Temp);
    end;
end;
```

KOMPILASI, Catatan ke-7 : Membentuk *Intermediate Code*

1. Pendahuluan

Kode antara (*intermediate code*) adalah sebuah representasi yang disiapkan untuk mesin abstrak tertentu. Dua sifat yang harus dipenuhi oleh kode antara adalah :
- dapat dihasilkan dengan mudah
- mudah ditranslasikan menjadi program sasaran (*target program*)

Representasi kode antara biasanya berbentuk perintah tiga alamat (*three-address code*), baik berbentuk *quadruples* ataupun *triples* (lihat kembali catatan hal 2-3).

2. *Syntax-Directed Translation*

Kode-antara (*intermediate code*) dibentuk dari sebuah kalimat x dalam bahasa *context free*. Kalimat x ini adalah keluaran dari *parser*. Kalimat ini tentu saja dapat dinyatakan dalam representasi pohon parsing (*parse tree*). *Syntax-directed transalation* adalah suatu *urutan proses* yang mentranslasikan *parse tree* menjadi kode-antara. Tahap pertama dari pembentukan kode antara adalah *evaluasi atribut setiap token dalam kalimat x*. Yang dapat menjadi atribut setiap token adalah semua informasi yang disimpan di dalam *tabel symbol*. Evaluasi dimulai dari *parse tree*.

Pandang sebuah *node* n yang ditandai sebuah token X pada *parse tree*. Kita tuliskan X.a untuk menyatakan atribut a untuk token X pada *node* n tersebut. Nilai X.a pada *node* n tersebut dievaluasi dengan menggunakan *aturan semantik* (*semantic rule*) untuk atribut a. Aturan semantik ini ditetapkan untuk setiap produksi dimana X adalah ruas kiri produksi. Sebuah *parse tree* yang menyertakan nilai-nilai atribut pada setiap *node*nya dinamakan *annonated parse tree*. Kumpulan aturan yang menetapkan aturan-aturan semantik untuk setiap produksinya dinamakan *syntax-directed definition*.

Untuk jelasnya berikut ini adalah sebuah *syntax-directed translation* yang mentranslasikan ekspresi *infix* menjadi ekspresi *postfix*. Ekspresi infix ini dapat dipandang sebagai sebuah kalimat yang dihasilkan oleh *parser*.

Contoh 1:

Diketahui : 1. kalimat x : 9 – 5 + 2

2. grammar Q = {E → E + T | E – T | T, T → 0 | 1 | 3 | ... | 9}

3. *syntax-directed definition* :

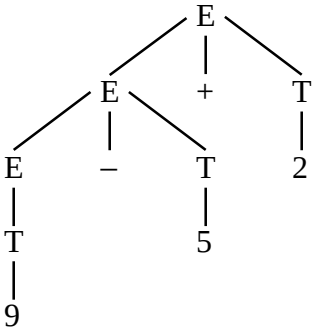
Produksi	Aturan Semantik
$E \rightarrow E_1 + T$	E := E_1 .t ‖ T.t ‖ '+ '
$E \rightarrow E_1 - T$	E := E_1 .t‖ T.t ‖ '–'
E → T	E := T.t
E → 0	E := '0'
E → 1	E := '1'
...	...
E → 9	E := '9'

catatan : 1. Lambang '‖' menyatakan *concatenation*.

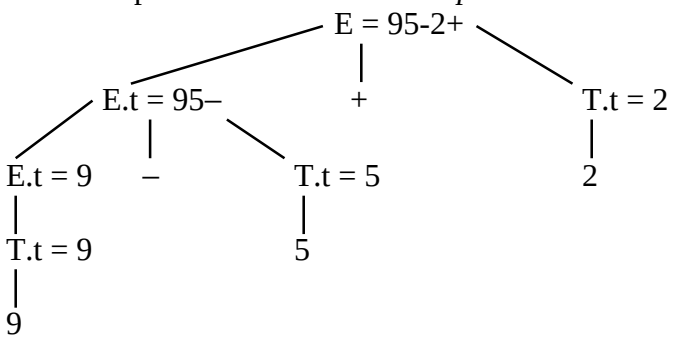
2. Mengingat catatan 1, aturan semantik kedua produksi pertama adalah *concate* dua operan diikuti sebuah operator.

Langkah-langkah translasi

1. pembentukan *parse tree* :



2. pembentukan *annonated parse tree* :

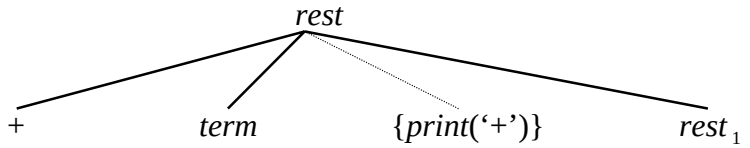


3. *Translation Scheme*

Translation scheme adalah grammar *context free* dimana sebuah frase yang disebut *semantic actions* disertakan di sisi kanan setiap produksinya. *Semantic actions* ini adalah sebuah nilai sebagai hasil evaluasi atribut pada sebuah *node*. Dalam *translation schemes*, frase yang menyatakan *action* diapit dengan kurung kurawal. Contoh sebuah *translation scheme* dengan *semantic action*-nya adalah :

$$rest \rightarrow + \; term \; \{print(\text{`+'})\} \; rest_1$$

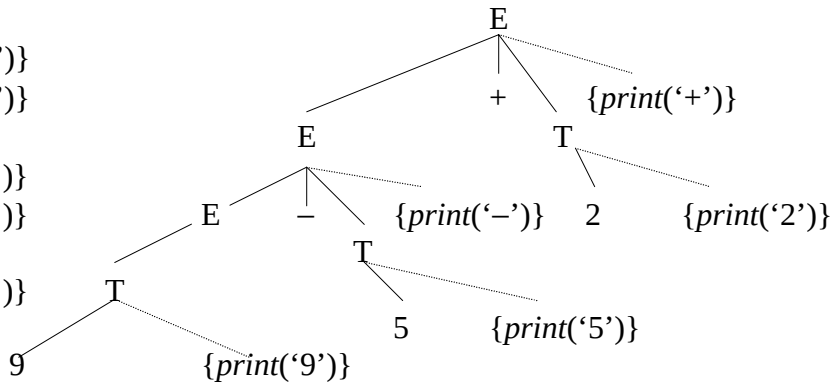
Pada contoh di atas, aksi {*print*(‘+’)} akan dilakukan setalah evaluasi terhadap *term* selesai dan evaluasi terhadap $rest_1$ belum dilakukan. Dalam representasi *parse tree*, *semantic action* pada *translation scheme* digambarkan dengan garis putus-putus sebagai *leaf* tambahan. *Parse tree* untuk contoh di atas digambarkan sebagai berikut :



Dari penjelasan di atas terlihat bahwa *translation scheme* mirip dengan *syntax-directed translation* kecuali dalam prosedur evaluasi nilai atribut pada setiap *node*nya. Pada *syntax-directed translation* hasil akhir evaluasi atribut ditetapkan pada *root* dari *parse tree* melalui kunjungan *dept-first traversal* sedangkan pada *translation scheme* hasil tersebut ditetapkan pada setiap node (termasuk *leaf*) juga secara *dept-first traversal*.

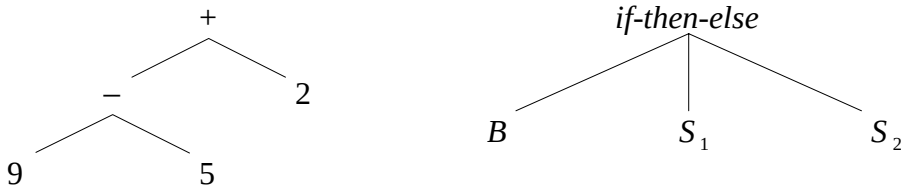
Gramar dan kalimat pada *contoh 1* di atas mempunyai *translation schemes* dan *parse tree* sebagai berikut :

$E \rightarrow E + T$ {*print*(‘+’)}
$E \rightarrow E - T$ {*print*(‘–’)}
$E \rightarrow T$
$T \rightarrow 0$ {*print*(‘0’)}
$E \rightarrow 1$ {*print*(‘1’)}
...
$E \rightarrow 9$ {*print*(‘9’)}



4. *Syntax Tree*

Parse tree seringkali disebut sebagai *concrete syntax tree*. *Parse tree* tidak efisien karena mengandung informasi yang berlebihan; sebagai gantinya, kalimat dinyatakan sebagai *abstract syntax tree* atau sering disingkat sebagai *syntax tree* saja. Dalam *syntax tree* setiap operator maupun *keyword* tidak muncul sebagai *leaf*. Berikut ini adalah contoh *syntax tree* untuk kalimat x : 9 – 5 + 2 dan y : *if B then* S_1 *else* S_2 :



5. Membentuk *Syntax Tree* Sebuah Ekspresi

Pada pasal ini akan dibentuk sebuah *syntax tree* dimana setiap *node*-nya dinyatakan sebagai data *record*. Setiap *record* paling tidak mengandung sebuah *label* yang dapat berupa identifier, konstanta, atau operator. Proses pembentukan ini memerlukan tiga buah *function* berikut :

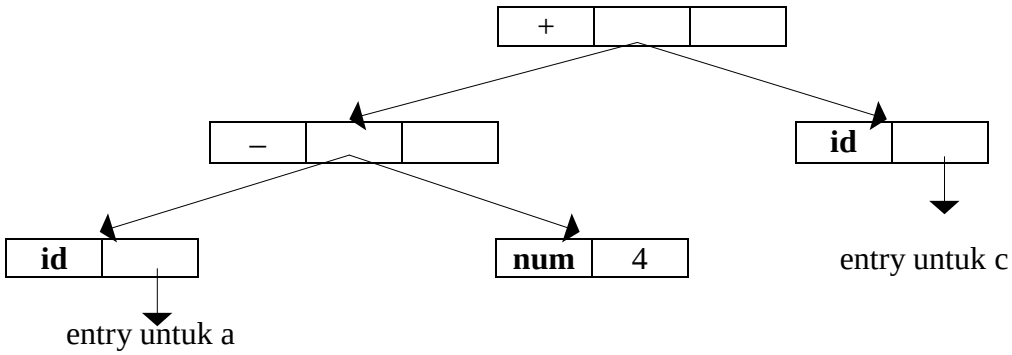
1. *mknode*(*op, left, right*) : membentuk node operator dengan label *op* dan dua *field* lainnya berupa pointer ke anak kiri dan anak kanan,
2. *mkleaf*(**id**, *entry*) : membentuk node identifier dengan label **id** dan sebuah *field* berupa pointer ke *symbol table*. Pelacakan identifier pada *symbol table*, lihat catatan ke-5 hal. 23-24.
3. *mkleaf*(**num**, *val*) : membentuk node konstanta dengan label **id** dan sebuah *field* yang mengandung nilai bilangan.

Contoh 2 :

Diberikan ekspresi : a – 4 + c, dengan a dan c adalah identifier. *Syntax tree* akan dibentuk secara *bottom-up*. Berikut ini adalah algoritma pembentukan *syntax tree* untuk ekspresi yang diinginkan beserta bentuk *syntax tree* yang dihasilkannya :

(1) *p1* := *mkleaf*(**id**, *entrya*);
(2) *p2* := *mkleaf*(**num**, 4);
(3) *p3* := *mknode*(‘–‘, *p1*, *p2*);
(4) *p4* := *mkleaf*(**id**, *entryc*);
(5) *p5* := *mknode*(‘+‘, *p3*, *p4*);

dimana :
- *p1, p2, p3, p4, p5* : pointer ke *node*
- *entrya, entryc* : pointer ke *symbol table* masing-masing untuk identifier a dan c



Grammar untuk ekspresi *contoh 2* adalah : Q = {E → E + T | E–T | T, T→ (E) | **id** | **num**}, sedangkan *syntax directed definition*-nya adalah :

Produksi	Aturan Semanti
$E \rightarrow E_1 + T$	*E.nptr* := *mknode*(‘+’, E_1 *.nptr, T.nptr*)
$E \rightarrow E_1 - T$	*E.nptr* := *mknode*(‘–’, E_1 *.nptr, T.nptr*)
E → T	*E.nptr* := *T.nptr*
E → (E)	*T.nptr* := *E.nptr*
E → **id**	*T.nptr* := *mkleaf*(**id**, **id**.*entry*)
E → **num**	*T.nptr* := *mkleaf*(**num**, **num**.*val*)

pada tabel di atas *nptr* adalah *synthesized attribute*, masing-masing untuk E dan T. Sebuah atribut di sebuah *node* adalah *synthesized attribute* jika nilai atribut pada *node* tersebut ditentukan dari nilai-nilai atribut anak-anaknya.

6. *Three Address Code*

Bentuk umum *three address code* bagi sebuah pernyataan adalah : x := y *op* z, dimana x, y, z adalah nama atau konstanta, sedangkan *op* adalah operator aritmatika atau operator logika. Ekspresi panjang seperti v := x + y * z dapat dinyatakan dengan *three address code* menjadi : t1 := y * z; t2 := x + t1; v := t2. Terdapat dua format untuk *three address code*, yaitu format *quadruple* (*op, arg1, arg2, result*) dan format *triple* (*op, arg1, arg2*). Isi dari *arg1*, *arg2*, dan *result* adalah pointer ke *symbol table*. Jika isi tersebut adalah *temporary identifier*, maka nama tersebut harus dimasukkan ke dalam *symbol table* begitu nama tersebut dideklarasikan atau diciptakan.

Contoh 3 :

Diberikan pernyataan : a := b * – c + b * – c.

Rangkaian *three address code* dari pernyataan ini adalah :

t1 := – c; t2 := b * t1; t3 := – c; t4 := b * t3; t5 := t2 := t4; a := t5

Pernyataan *quadruple* dan *triple* dari pernyataan di atas adalah :

Tabel format quadruple

	op	*arg1*	*arg2*	*result*
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Tabel format triple :

	op	*arg1*	*arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Format quadruple untuk opearor *unary* seperti ‘– c’ menyebabkan *arg2* tidak digunakan. Format *triple* digunakan untuk menghindari penggunaan *temporary identifier* seperti yang terjadi pada format quadruple. Format triple ini benar-benar mendayagunakan posisi baris pernyataan. Sebagai contoh, pada baris kedua dari tabel format triple terlihat bahwa isi *arg2* adalah (0), yang tak lain dari ‘uminus c’.