BAB 1

PENDAHULUAN KOMPILASI

Pendahuluan

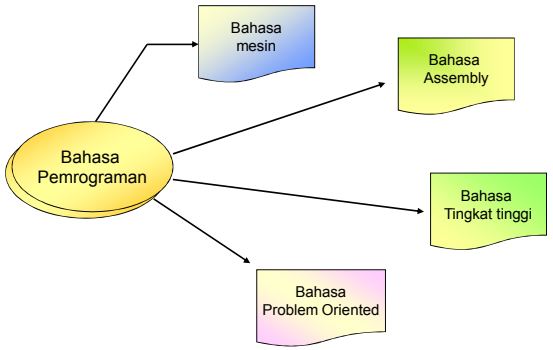
Tujuan Pembelajaran :

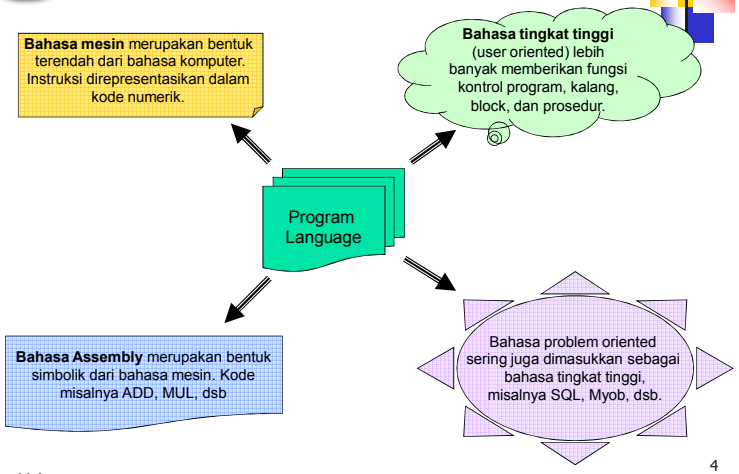
1. Mahasiswa memahami cara kerja serta proses yang terjadi pada sebuah Compiler
2. Mahasiswa memahami konsep pembuatan sebuah Compiler
3. Mahasiswa mengetahui bagaimana sebuah bahasa pemrograman dapat diterjemahkan oleh mesin.

Metari Pembelajaran

1. Bentuk-bentuk karakter dan kelas Grammar.
2. Ekspresi Regular dan Automata
3. Analisa Leksikal sebagai tahap awal kompilasi
4. Analisa Sintaks, bentuk-bentuk derivasi serta implementasi parsing.
5. Analisa Semantik dan tahapan Sintesa.
6. Penanganan kesalahan kompilasi dan fungsi tabel informasi.

1. Bahasa Pemrograman

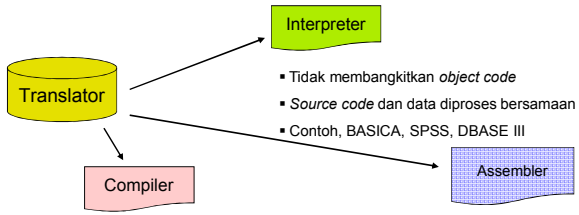




2. Translator

Translator melakukan pengubahan source code / source program kedalam target code / object code

Interpreter dan Compiler termasuk dalam kategori translator.

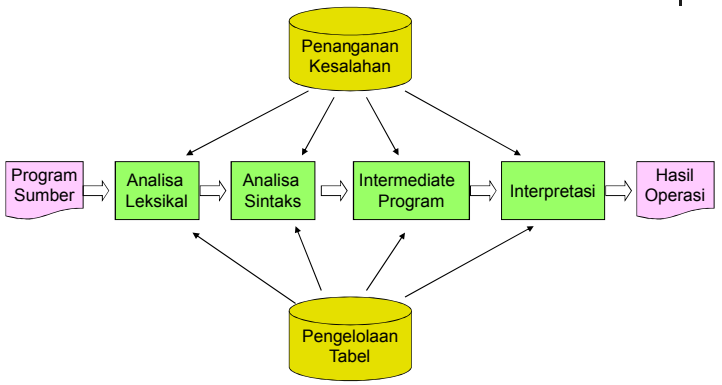


- Tidak membangkitkan *object code*
- *Source code* dan data diproses bersamaan
- Contoh, BASICA, SPSS, DBASE III

- *Source Code* adalah bahasa tingkat tinggi.
- *Object Code* adalah bahasa mesin atau assembly
- *Source code* dan data diproses tidak bersamaan
- Contoh, PASCAL, C

- *Source Code* adalah bahasa Assembly
- *Object Code* adalah bahasa mesin
- Contoh, Turbo Assembler

Interpreter



Assembler

.ASM **.OBJ** **.EXE / .COM**

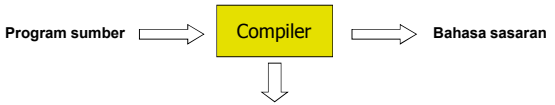
Proses Sebuah Kompilasi pada Bahasa Assembler

- Source Code adalah bahasa Assembler, Object Code adalah bahasa mesin
- Object Code dapat berupa file object (.OBJ), file .EXE, atau file .COM
- Contoh : Turbo Assembler (dari IBM) dan Macro Assembler (dari Microsoft)

Compiler

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

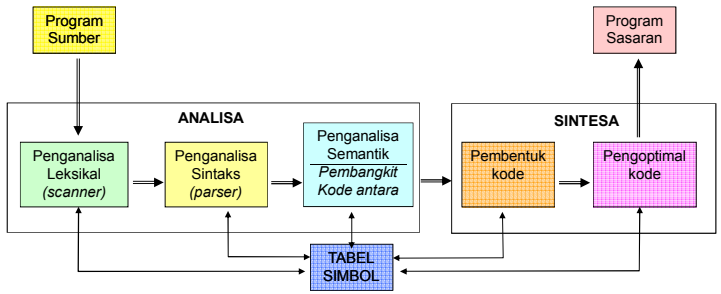
Proses kompilasi dapat digambarkan melalui sebuah blok diagram sebagai berikut :



Proses Kompilasi dikelompokkan kedalam 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

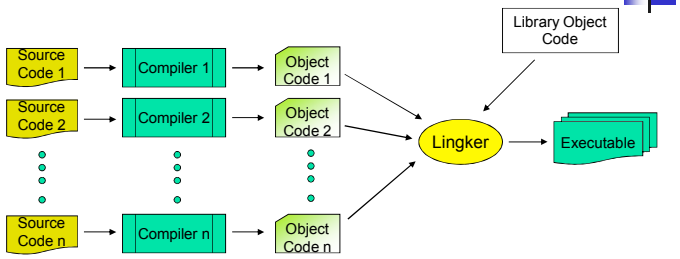
Blok Diagram



Bagan pokok proses kompilasi

Keterangan

1. ***Program Sumber*** ditulis dalam bahasa sumber, misal Pascal, Assembler, dsb
2. ***Program Sasaran*** dapat berupa bahasa pemrograman lain atau bahasa mesin pada suatu komputer
3. ***Scanner*** : Memecah program sumber menjadi besaran leksik/token
4. ***Parser*** : Memeriksa kebenaran dan urutan kemunculan token
5. ***Penganalisa semantik*** : Melakukan analisa semantik, biasanya dalam realisasi akan digabungkan Dengan *intermediate code generator* (bagian yang berfungsi membangkitkan kode antara)
6. ***Pembentuk Kode*** : Membangkitkan kode objek
7. ***Pengoptimal Kode*** : Memperkecil hasil dan mempercepat proses
8. ***Tabel*** : Menyimpan semua informasi yang berhubungan dengan proses kompilasi



- Pembentukan file Executable berdasar dari beberapa Source Code
- Source Code dapat terdiri dari satu atau lebih behasa pemrograman.

Pembuatan Compiler

Pembuatan kompilator dapat dilakukan dengan :

1. Bahasa Mesin
   Tingkat kesulitannya tinggi, bahkan hampir mustahil dilakukan
2. Bahasa Assembly
   Bahasa Assembly bisa dan biasa digunakan sebagai tahap awal pada proses pembuatan sebuah kompilator
3. Bahasa Tingkat Tinggi lain pada ,mesin yang sama
   Proses pembuatan kopilator akan lebih mudah
4. Bahasa tingkat tinggi yang sama pada mesin yang berbeda
   Misal, pembuatan kompilator C untuk DOS, berdasar C pada UNIX
5. Bootstrap
   Pembuatan kompilator secara bertingkat.

BAB 2

- KONSEP DAN NOTASI

Konsep dan Notasi Bahasa

Teori Bahasa

Bahasa adalah kumpulan kalimat. Kalimat adalah rangkaian kata. Kata adalah komponen terkecil kalimat yang tidak bisa dipisahkan lagi.

Contoh : Si Kucing kecil menendang bola besar ▸ Bhs Indonesia

The little cat kicks a big ball ▸ Bhs Inggris

for i := start to finish do A[i] := B[i]*sin(i*pi/16.0) ▸ Bhs Pascal

Dalam bahasa pemrograman, *kalimat* dikenal sebagai *ekspresi*, dan *kata* sebagai *token*. Kata terdiri atas beberapa karakter. Kelompok karakter yang membentuk sebuah token dinamakam *lexeme* untuk token tersebut. Setiap token yang dihasilkan, disimpan dalam tabel simbol.

Derivasi adalah sebuah proses dimana suatu himpunan produksi akan diturunkan / dipilah-pilah dengan melakukan sedertan produksi sehingga membentuk untai terminal.

Grammar dan bahasa

Pengertian dasar

1. Setiap anggota alfabet, dinamakan sebagai simbol terminal atau token
2. Himpunan simbol terminal dinyatakan sebagai V_N, sedangkan himpunan simbol non terminal dinyatakan sebagai V_T.
3. Simbol-simbol berikut adalah simbol terminal :
   - Huruf kecil awal alfabet, misal x, y, z.
   - Simbol operator, misal +, -, dan x
   - Simbol tanda baca, misal (,), dan ;
   - String yang tercetak tebal, misal, **if, then,** dan **else**
4. Simbol-simbol berikut adalah simbol non terminal
   - Huruf besar awal alfabet, misal X, Y, Z.
   - Huruf S sebagai simbol awal
   - String yang tercetak miring, misal *expr* dan *stmt*
5. Huruf besar akhir alfabet melambangkan simbol terminal atau non terminal, misal X, Y, Z
6. Huruf kecil akhir alfabet melambangkan string yang tersusun atas simbol-simbol terminal, misalnya : x, y, z.

Grammar dan bahasa

6. Sebuah produksi dilambangkan sebagai $\alpha \rightarrow \beta$, artinya : dalam sebuah derivasi dapat dilakukan penggantian simbol α dengan simbol β.
7. Simbol α dalam produksi berbentuk $\alpha \rightarrow \beta$ disebut ruas kiri produksi sedangkan simbol β disebut ruas kanan produksi.
8. Pengertian terminal berasal dari kata *terminate* (berakhir), maksudnya derivasi berakhir jika sentensial yang dihasilkan adalah sebuah kalimat (yang tersusun atas simbol-simbol terminal itu).
9. Pengertian non terminal berasal dari kata *not terminate* (belum/tidak berakhir), maksudnya derivasi belum/tidak berakhir jika sentensial yang dihasilkan mengandung simbol non terminal.
10. String adalah deretan terbatas (*finite*) simbol-simbol. Sebagai contoh, jika *a*, *b*, dan *c* adalah tiga buah simbol maka *abcb* adalah sebuah string yang dibangun dari ketiga simbol tersebut.
11. Jika *w* adalah sebuah string maka panjang string dinyatakan sebagai $|w|$ dan didefinisikan sebagai cacahan (banyaknya) simbol yang menyusun string tersebut. Sebagai contoh, jika $w = abcb$ maka $|w| = 4$.
12. String hampa adalah sebuah string dengan nol buah simbol. String hampa dinyatakan dengan simbol ε (atau ^) sehingga $|\varepsilon| = 0$. String hampa dapat dipandang sebagai simbol hampa karena keduanya tersusun dari nol buah simbol.

Grammar dan Klasifikasi Chomsky

Grammar G didefinisikan sebagai pasangan 4 tuple : V_N, V_T, S, dan Q, dan dituliskan sebagai $G(V_N, V_T, S, Q)$, dimana :

V_T : himpunan simbol-simbol terminal (atau himpunan token -token, atau alfabet)
V_N : himpunan simbol-simbol non terminal
$S \in V_N$: simbol awal (atau simbol start)
Q : himpunan produksi

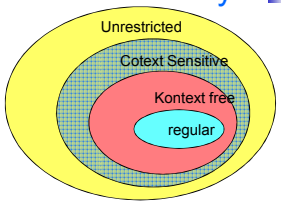
Aturan produksi dinyatakan sebagai $\alpha \rightarrow \beta$, artinya α menurunkan β

Berdasarkan komposisi bentuk ruas kiri dan ruas kanan produksinya ($\alpha \rightarrow \beta$), Noam Chomsky mengklasifikasikan 4 tipe grammar :

1. Grammar tipe ke-0 : Unrestricted Grammar (UG)
   Ciri : $\alpha, \beta \in (V_T | V_N)^*$, $|\alpha| > 0$
2. Grammar tipe ke-1 : Context Sensitive Grammar (CSG)
   Ciri : $\alpha, \beta \in (V_T | V_N)^*$, $0 < |\alpha| \leq |\beta|$
3. Grammar tipe ke-2 : Context Free Grammar (CFG)
   Ciri : $\alpha \in V_N$, $\beta \in (V_T | V_N)^*$
4. Grammar tipe ke-3 : Regular Grammar (RG)
   Ciri : $\alpha \in V_N$, $\beta \in \{V_T, V_T V_N\}$ atau $\alpha \in V_N$, $\beta \in \{V_T, V_N V_T\}$
   Ciri-ciri RG sering dituliskan sebagai :
   $\alpha \in V_N$, $\beta \in \{a, bC\}$ atau $\alpha \in V_N$, $\beta \in \{a, Bc\}$

Kelas Bahasa

Hirarki Chomsky



Mesin Pengenal bahasa

Type *Grammar*	Kelas Bahasa	Mesin Pengenal Bahasa
Unrestricted Grammar (UG)/type-0	*Unrestricted*	Mesin Turing (*Turing Machine*), TM
Context Sensitive Grammar (CSG)/type-1	*Context Sensitive*	*Linear Bounded Automaton*, LBA
Context Free Gammar (CFG)/type-2	*Context Free*	Automata Pushdown (*Pushdown Automata*), PDA
Regular Grammar (RG)/type-3	*Regular*	Automata Hingga (*Finite Automata*)

Notasi BNF

Aturan-aturan produksi dapat dinyatakan dalam bentuk BNF (*Backus Naur Form*)

Beberapa simbol yang dipakai dalam notasi BNF

::=	Identik dengan simbol → pada aturan produksi
|	Menyatakan "atau"
< >	Mengapit simbol variabel / non terminal
{ }	Pengulangan 0 sampai n kali

Contoh, terdapat aturan produksi sebagai berikut :

→ E→ T | T+E | T-E, T a

Notasi BNF :

E ::= <T> | <T> + <E> | <T> - <E>, T ::= a

Diagram Sintaks

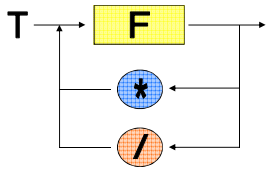
Diagram sintaks merupakan alat bantu dalam pembentukan parser / analisis sintaks. Notasi yang terdapat dalam diagram sintaks :

- Empat persegi panjang melambangkan simbol variabel / non terminal.
- Bulatan melambangkan simbol terminal

Misal, terdapat aturan produksi :

→ T F*T | F/T | F

Diagram sintaksnya adalah sebagai berikut :



BAB 3

- BAHASA REGULER

PENDAHULUAN

- Bahasa regular adalah penyusun *ekspresi reguler* (ER)
- Ekspresi reguler terdiri dari kombinasi simbol-simbol atomik menggunakan 3 operasi yaitu :
  - katenasi,
  - alternasi, dan
  - repetisi /closure
- Pada kasus scanner, simbol-simbol atomik adalah karakter-karakter di dalam program sumber.
- Dua buah ekspresi regular adalah ekuivalen jika keduanya menyatakan bahasa yang sama

Operasi Regular - katenasi

- Katenasi /konkatenasi atau sequencing disajikan dengan physical adjacency
  - e.g. ekspresi regular '<letter> <digit>' bentuk penyajian sederhana (diasumsikan sebagai definisi yang jelas dari letter dan digit) komposisi terurut dari letter diikuti dengan digit
  - » "<" dan ">" digunakan untuk mengidentifikasi simbol-simbol yang merepresentasikan simbol-simbol spesifik (menggunakan ekspresi regular)
  - » Kita bisa menggunakan "::=" (ekivalensi) untuk menggabungkan ekspresi regular yang didefinisikan dengan <letter> dan <digit>

Operasi Regular - alternasi

- Alternasi membolehkan pilihan dari beberapa pilihan dan biasanya disajikan dengan operator '|'
  - E.g. <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  - » contoh yang menggunakan juga operator ekivalensi
- Bentuk tulisan cepat tertentu juga biasanya digunakan dengan alternasi (khususnya ellips)
  - E.g. <letter> ::= a | b | … | z | A | B | … | Z
  - » Can use the ellipses ("…") when a sequence is well defined

Operasi Regular - repetisi

- Terakhir, repetisi membolehkan ekspresi dari kontruksi yang diulang beberapa kali
- Terdapat 2 operator yang digunakan yaitu superscript '+' dan superscript '*'
  - E.g. <word> ::= <letter>+
  - » this implies a word consists of one or more letters (* would imply zero or more letters and a word must have at least one letter so we use +)

Ekivalensi ER [1]

Contoh :

$L = \{aba \mid n \geq 1, m \geq 1\} \Leftrightarrow er = a\ b\ a$

$L = \{aba \mid n \geq 0, m \geq 0\} \Leftrightarrow er = a^*\ b\ a^*$

- Perhatikan bahwa kita tidak bisa membuat ekspresi regular dari bahasa $L = \{aba \mid n \geq 1\}$ atau $L = \{aba \mid n \geq 0\}$, karena keduanya tidak dihasilkan dari grammar regular.

Ekivalensi ER[2]

- $(a\ b)^*\ a = a\ (b\ a)^*$

  Bukti :

  $(a\ b)^*\ a = (\varepsilon \mid (ab) \mid (abab) \mid \ldots)\ a$

  $= (\varepsilon\ a \mid (aba) \mid (ababa) \mid \ldots)$

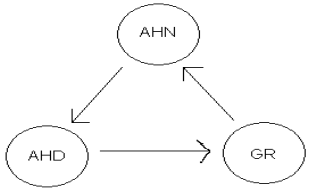
  $= (a \mid (aba) \mid (ababa) \mid \ldots)$

  $= a\ (\varepsilon \mid (ba) \mid (baba) \mid \ldots)$

  $= a\ (b\ a)^*$

Ekivalensi AHN, AHD, dan GR

- AHD bisa dibentuk dari AHN.
- GR bisa dibentuk dari AHD.
- AHN bisa dibentuk dari GR.



Pembentukan AHD dari AHN

Diberikan sebuah AHN F = (K, V, M, S, Z). Akan dibentuk sebuah AHD F' = (K', V', M', S', Z') dari AHN F tersebut.

Algoritma pembentukannya adalah sbb. :

- Tetapkan : S' = S dan V' = V
- Copykan tabel AHN F sebagai tabel AHD F'. Mula-mula K' = K dan M' = M
- Setiap stata q yang merupakan *nilai* (atau *peta*) dari fungsi M dan q $\notin$ K, ditetapkan sebagai elemen baru dari K'. Tempatkan q tersebut pada kolom Stata M', lakukan pemetaan berdasarkan fungsi M
- Ulangi langkah diatas sampai tidak diperoleh stata baru
- Elemen Z' adalah semua stata yang mengandung stata elemen Z.

Pembentukan GR dari AHD

Diketahui sebuah AHD F = (K, V, M, S,Z). Akan dibentuk GR G = (V',V, S', Q).

Algoritma pembentukan GR dari AHD adalah sebagai berikut :

- Tetapkan V' = V, S' = S, V = S
- Jika A, A$\in$ K dan a $\in$ V, maka :

  M(A, a) = A ekuivalen dengan produksi :

$$\begin{cases} A_p \rightarrow aA_q, & \text{jika } A_q \notin Z \\ A_p \rightarrow a, & \text{jika } A_q \in Z \end{cases}$$

Pembentukan AHN dari GR

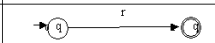
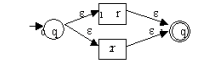
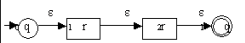
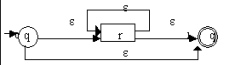
Diketahui GR G = (V,V, S, Q). Akan dibentuk AHN F = (K,V', M, S', Z).

Algoritma pembentukan AHN dari GR :

- Tetapkan V' = V, S' = S, K = V
- Produksi A $\rightarrow$ a A ekuivalen dengan M(A, a) = A

  Produksi A $\rightarrow$ a ekuivalen dengan M(A, a) = X, dimana X $\notin$ V
- K = = K $\cup$ {X}
- Z = {X}

Ekivalensi AHN-ε Dengan ER (Ekspresi Regular)

Jenis ER	Simbol ER	AHN
Simbol hampa	ε	
ER hampa	ϕ atau {}	
ER umum	r	
Alternation	r1 | r2	
Concatenation	r1 r2	
Kleene Closure	r*	

ANALISIS LEKSIKAL

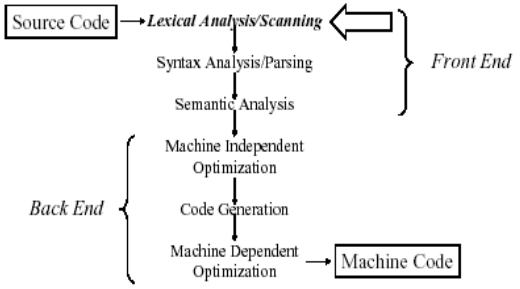
BAB 4

- ANALISA LEKSIKAL

Pembahasan

- Letak analisis leksikal pada struktur kompiler
- Pengenalan analisis leksikal
- Scanning berdasarkan MSH
- Tugas analisis leksikal

Struktur Kompiler



Analisis Leksikal – Apa itu?[1]

- Masukan bagi sebuah compiler/interpreter adalah program sumber yang strukturnya berupa deretan dari karakter-karakter
  - or rather *unstructured*
- Pemrosesan individual karakter yang ketidakefisiennya sangat tinggi
  - » Imagine recognizing 'while' as 'w' 'h' 'i' 'l' 'e'
- Oleh karenanya, hal pertama yang kita perhatikan adalah bentuk kode sumbernya

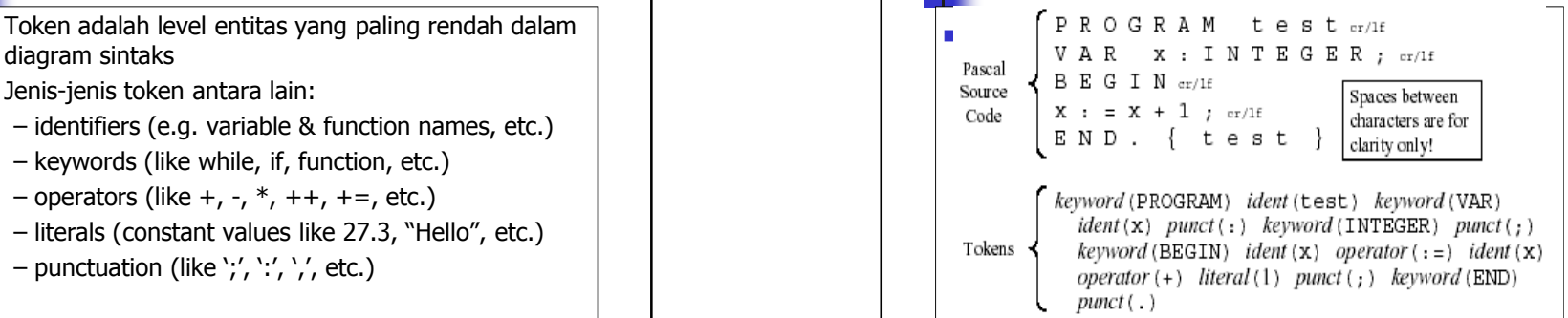
Analisis Leksikal – Apa itu?[2]

- A *Lexical Analyzer* (*scanner)* mengubah deretan karakter-karakter menjadi deretan token-token
  - i.e. a scanner "tokenizes" the input
- Sebuah *token* (*lexeme* or *syntactic unit*) adalah komponen dasar leksikal dari program

Analisis Leksikal–Token[3]

- Token adalah level entitas yang paling rendah dalam diagram sintaks
- Jenis-jenis token antara lain:
  - identifiers (e.g. variable & function names, etc.)
  - keywords (like while, if, function, etc.)
  - operators (like +, -, *, ++, +=, etc.)
  - literals (constant values like 27.3, "Hello", etc.)
  - punctuation (like ';', ':', ',', etc.)

Analisis Leksikal–Tokens[4]



Fungsi Scanner

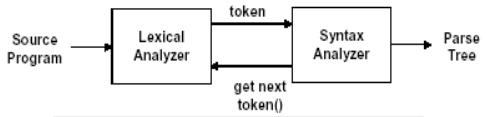
- Melakukan pembacaan kode sumber dengan merunut karakter demi karakter
- Mengenali besaran leksik
- Mentransformasi menjadi sebuah token dan menentukan jenis tokennya
- Mengirim token
- Membuang blank dan komentar dalam program
- Menangani kesalahan
- Beberapa scanners memasukkan simbol ke dalam tabel simbol (dibahas kemudian)

Scanning berdasarkan MSH

- Hampir sebagian besar teknik yang digunakan untuk membangun scanners menggunakan *mesin stata hingga* (MSHs)
- MSHs dapat dengan mudah digunakan untuk mengenali kontruksi bahasa (i.e. tokens) yang digambarkan dengan bahasa regular

Membangun Scanner

- Bagaimana scanner berinteraksi dengan parser?
  - parser akan menjadi bagian selanjutnya dari kompilasi
- Perhatikan gambar berikut:



It's a subroutine calling relationship!

Aksi Scanner [1]

- Karena scanner mengubah dari stata ke stata, maka harus dilakukan sesuatu dengan karakter-karakter tersebut untuk mengenali sesuai dengan pembentukan token yang akan dikembalikan pada tahap parser
- Dalam beberapa kasus, harus menambahkan character seperti terlihat pada pembentukan token dan memanfaatkannya (menjadikan karakter masukan berikutnya menjadi kelihatan)
  - E.g. when scanning characters in an identifier

Aksi Scanner [2]

- Dalam kasus lainnya harus menjaga character dan mengembalikan dalam token lengkap
  - E.g. MaxVal := -999; ← a Pascal assignment
    - » After scanning the ':' we know that we have found the end of the identifier 'MaxVal' so we want to return that to the parser but we must not lose the ':' (so we must *preserve* it)
- Aksi kemungkinan lainnya adalah menghilangkan karakter agar lebih sederhana
  - E.g. karakter pada komentar

BAB 5 - 6

- CF DAN PARSING

CONTEXT-FREE GRAMMAR (CFG) DAN PARSING

- Bentuk umum produksi CFG adalah :
  $\alpha \rightarrow \beta, \quad \alpha \in V_N, \quad \beta \in (V_N | V_T)^*$
- Analisis sintaks :
  Penelusuran sebuah kalimat (sentensial) sampai pada simbol awal grammar. Analisis sintaks dapat dilakukan melalui derivasi atau *parsing*. Penelusuran melalui *parsing* menghasilkan *pohon sintaks*.

CONTEXT-FREE GRAMMAR (CFG) DAN PARSING

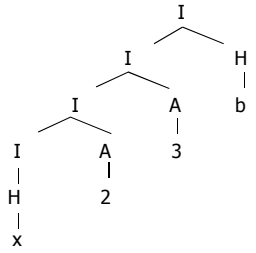
- Contoh :

Diketahui grammar G = {I → H | I H | IA, H → a | b | c | ... | z, A → 0 | 1 | 2 | ... | 9} dengan I adalah simbol awal.
Berikut ini kedua cara analisa sintaks untuk kalimat x23b.

cara 1 (derivasi)

I ⇒ IH
⇒ IAH
⇒ IAAH
⇒ HAAH
⇒ xAAH
⇒ x2AH
⇒ x23H
⇒ x23b

cara 2 (parsing)

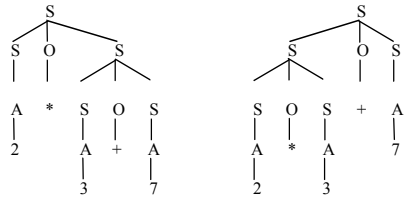


CONTEXT-FREE GRAMMAR (CFG) DAN PARSING

- Contoh :

  Diketahui grammar G = {S → SOS | A , O → * | +, A → 0 | 1 | 2 | ... | 9}

  Kalimat : 2*3+7 mempunyai dua pohon sintaks berikut :



Sebuah kalimat yang mempunyai lebih dari satu pohon sintaks disebut *kalimat ambigu* (*ambiguous*). Grammar yang menghasilkan paling sedikit sebuah kalimat ambigu disebut *grammar ambigu*.

Metoda Parsing

Ada 2 metoda parsing : top-down dan bottom-up.

- Parsing top-down :

  Parsing dimulai dari simbol awal S sampai kalimat x
- Parsing bottom-up :

  Parsing dimulai dari kalimat x sampai simbol awal S

Parsing Top-down

- Ada 2 kelas metoda parsing *top-down :*

  1. kelas metoda *dengan backup,*

     Contoh: metoda *Brute-Force*
  2. kelas metoda *tanpa backup*

     Contoh: metoda *recursive descent*.
- **Metoda *Brute-Force***

  Kelas metoda *dengan backup*, termasuk metoda *Brute-Force*, adalah kelas metoda parsing yang menggunakan produksi alternatif, jika ada, ketika hasil penggunaan sebuah produksi tidak sesuai dengan simbol input. Penggunaan produksi sesuai dengan nomor urut produksi.

Parsing Top-down

- **Metoda *Recursive-Descent***
- Kelas metoda *tanpa backup*, termasuk metoda *recursive descent*, adalah kelas metoda parsing yang tidak menggunakan produksi alternatif ketika hasil akibat penggunaan sebuah produksi tidak sesuai dengan simbol input. Jika produksi A mempunyai dua buah ruas kanan atau lebih maka produksi yang dipilih untuk digunakan adalah *produksi dengan simbol pertama ruas kanannya sama dengan input yang sedang dibaca*. Jika tidak ada produksi yang demikian maka dikatakan bahwa parsing tidak dapat dilakukan.
- Ketentuan produksi yang digunakan metoda *recursive descent* adalah : *Jika terdapat dua atau lebih produksi dengan ruas kiri yang sama maka karakter pertama dari semua ruas kanan produksi tersebut tidak boleh sama*. Ketentuan ini tidak melarang adanya produksi yang bersifat rekursi kiri.

Parsing Bottom-Up

- Salah satunya adalah ***grammar preseden sederhana*** **(GPS).**
- Pengertian Dasar
  - Jika α dan x keduanya diderivasi dari simbol awal grammar tertentu, maka α disebut *sentensial* jika $\alpha \in (V \mid V)^*$, dan x disebut *kalimat* jika $x \in (V)^*$
  - Misalkan $\alpha = Q_1 \beta Q_2$ adalah sentensial dan $A \in V_N$:

- β adalah *frase* dari sentensial α jika : $S \Rightarrow \ldots \Rightarrow Q_1 \alpha Q_2$ dan $\alpha \Rightarrow \ldots \Rightarrow \beta$
- β adalah *simple frase* dari sentensial α jika : $S \Rightarrow \ldots \Rightarrow Q_1 \alpha Q_2$ dan $\alpha \Rightarrow \beta$
- Simple frase terkiri dinamakan *handel*
- frase, simple frase, dan handel adalah string dengan panjang ≥ 0

Parsing Bottom-Up

Contoh 6 :

$I \Rightarrow I\ H$
$\Rightarrow H\ H$
$\Rightarrow H\ b$

Hb adalah sentensial dan b adalah simple frase
(dibandingkan dengan $Q_1 \beta Q_2$ maka Q= H, β = b, dan Q = ε)
Perhatikan : simple frase (b) adalah yang terakhir diturunkan

$I \Rightarrow I\ H$
$\Rightarrow I\ b$
$\Rightarrow H\ b$

Hb adalah sentensial dan H adalah simple frase
(dibandingkan dengan $Q_1 \beta Q_2$ maka Q= ε, β = H, dan Q = b)
Perhatikan : simple frase (H) adalah yang terakhir diturunkan

- Sentensial Hb mempunyai dua simple frase (b dan H), sedangkan handelnya adalah H.

BAB 7

- ANALISA SEMANTIK

ANALISIS SEMANTIK, KODE ANTARA, DAN PEMBANGKITAN KODE

ANALISIS SEMANTIK

Analisis semantik ini memanfaatkan pohon sintaks yang dihasilkan pada proses parsing (analisa sintaks).

Fungsi dari analisa semantik adalah untuk menentukan makna dari serangkaian instruksi yang terdapat dalam program sumber.

Untuk mengetahui makna, maka rutin analisa semantik akan memeriksa :
•Apakah variabel yang ada telah didefinisikan sebelumnya
•Apakah variabel – variabel tersebut tipenya sama
•Apakah operan yang akan dioperasikan tersebut ada nilainya dan seterusnya.

Untuk dapat menjalankan fungsi tersebut dengan baik, semantic analyzer seringkali menggunakan tabel simbol. Pemeriksaan bisa dilakukan pada tabel *identifier*, tabel *display* dan tabel blok, misal pada *field link*.

ANALISIS SEMANTIK

Pengecekan yang dilakukan oleh analisis semantik adalah :

- Memeriksa keberlakuan nama – nama meliputi pemeriksaan :
  - Duplikasi
    Pengecekan apakah sebuah nama terjadi pendefinisian lebih dari dua kali. Pengecekan dilakukan pada bagian pengelola blok.
  - Terdefinisi
    Pengecekan apakah sebuah nama yang dipakai pada tubuh program sudah terdefinisi atau belum. Pengecekan dilakukan pada semua tempat kecuali blok
- Memeriksa tipe
  Melakukan pemeriksaan terhadap kesesuaian tipe dalam statemen – statemen yang ada.
  Misal; bila ada operasi antara dua operan maka tipe operan pertama harus bisa dioperasikan dengan operan kedua.

KODE ANTARA

Kegunaan dari Kode Antara / *intermediate code* :

- Untuk memperkecil usaha dalam membangun kompilator dari sejumlah bahasa ke sejumlah mesin
- Proses optimasi lebih mudah. (dibandingkan pada program sumber atau kode *assembly* dan kode mesin)
- Bisa melihat program internal yang gampang dimengerti

2 macam Kode Antara yang biasa digunakan adalah Notasi Postfix dan N-*Tuple*

KODE ANTARA

Notasi Postfix

Pada Notasi Postfix operator diletakkan paling akhir.

Sintaks Notasi Postfix:
< operan><operan><operator>

misalkan ekspresi :
(a+b)*(c+d)
dapat dinyatakan dalam bentuk Notasi Postfix :
ab+cd+*

KODE ANTARA

Kontrol program yang ada dapat diubah kedalam bentuk notasi postfix, misalnya:
IF <exp> THEN <stmt1> ELSE <stmt2>

Diubah kedalam Notasi Postfix :
<exp> <label1> BZ <stmt1> <label2> BR <stmt2>

label1 label2

Keterangan :
- BZ : branch if zero (zero = salah) {bercabang jika kondisi yang dites salah}
- BR : branch {bercabang tanpa ada kondisi yang dites}

Arti dari notasi Postfix diatas adalah :
"Jika kondisi ekspresi salah, maka instruksi akan meloncat ke Label1 dan menjalankan statement2. Bila kondisi ekspresi benar, maka statement1 akan dijalankan lalu meloncat ke Label2. Label1 dan Label2 sendiri menunjukkan posisi tujuan loncatan, untuk Label1 posisinya tepat sebelum statement2 dan Label2 setelah statement2."

KODE ANTARA

Contoh lain :

WHILE <exp> DO <stat>

Diubah ke postfix :

<exp><label1>BZ<stat><label2>BR

label1 label2

Notasi N-*Tuple*

Pada notasi N-*Tuple* setiap baris bisa terdiri dari beberapa *tupel*.
Format umum dari notasi N-*Tuple* adalah :
operator ……..N-1 operan

Notasi N-*Tuple* yang biasa digunakan adalah notasi 3 *tupel* dan 4 *tupel*.

KODE ANTARA

Triples Notation

Memiliki format

<operator><operand><operand>

contoh, instruksi :
A:=D*C+B/E

Bila dibuat Kode Antara *tripel*:
1. *,D,C
2. /,B,E
3. +,(1),(2)
4. :=,A,(3)

KODE ANTARA

Kekurangan dari notasi *tripel* adalah sulit pada saat melakukan optimasi, maka dikembangkan *Indirect Triples* yang memiliki dua *list* (senarai), yaitu *list* instruksi dan *list* eksekusi. *List* instruksi berisi notasi *tripel*, sedangkan *list* eksekusi mengatur urutan eksekusinya. Misalnya terdapat urutan instruksi :

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

KODE ANTARA

Quadruples Notation

Format instruksi *Quadruples*

<operator><operan><operan><hasil>

- hasil adalah temporary yang bisa ditempatkan pada *memory* atau *register*

contoh instruksi:
A:=D*C+B/E

Bila dibuat dalam Kode Antara :
1. *,D,C,T1
2. /,B,E,T2
3. +,T1,T2,A

ANALISIS SEMANTIK

Analisis Semantik adalah proses setelah melewati proses scanning dan parsing. Pada tahap ini dilakukan pengecekan pada struktur akhir yang telah diperoleh dan diperiksa kesesuaiannya dengan komponen program yang ada. Secara global, fungsi dari *semantic analyzer* adalah untuk menentukan makna dari serangkaian instruksi yang terdapat dalam program sumber.

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

maka penganalisis semantik harus mampu menentukan aksi apa yang akan dilakukan oleh operator-operator tersebut. Dalam sebuah proses kompilasi, andaikata parser menjumpai ekspresi seperti diatas, parser hanya akan mengenali simbol-simbol ':=' , '+' , dan '*'. Parser tidak tahu makna apa yang tersimpan dibalik simbol simbol tersebut. Untuk mengenalinya, kompiler akan memanggil rutin semantik yang akan memeriksa :

- Apakah variabel-variabel yang ada telah didefinisikan sebelumnya?
- Apakah variabel-variabel tersebut tipenya sama?
- Apakah operand yang akan dioperasikan tersebut ada nilainya?, dan seterusnya.

Fungsi ini terkait dengan tabel simbol. Pengecekan yang dilakukan oleh analisis semantik adalah sebagai berikut :

a) Memeriksa keberlakuan nama-nama meliputi pemeriksaan berikut.
   - Duplikasi : pada tahap ini dilakukan pengecekan apakah sebuah nama terjadi pendefinisian lebih dari dua kali. Pengecekan dilakukan pada bagian pengelola blok.
   - Terdefinisi : Melakukan pengecekan apakah sebuah nama yang dipakai pada tubuh program sudah terdefinisi atau belum. Pengecekan dilakukan pada semua tempat kecuali blok.

b) Memeriksa tipe. Melakukan pemeriksaan terhadap kesesuaian tipe dalam statement-statement yang ada. Misalkan bila terdapat suatu operasi, diperiksa tipe operand. Contohnya bila ekspresi yang mengikuti instruksi IF berarti tipenya boolean, akan diperiksa tipe identifier dan tipe ekspresi. Bila ada operasi antara dua operand, maka tipe operand pertama harus bisa dioperasikan dengan operand kedua.

Analisa semantik sering juga digabungkan pada pembangkitan kode antara yang menghasilkan Output intermediate code, yang nantinya akan digunakan pada proses kompilasi berikutnya.

KODE ANTARA

Kode antara/*Intermediate code* merupakan hasil dari tahapan analisis, yang dibuat oleh kompilator pada saat mentranslasikan program dari bahasa tingkat tinggi. Kegunaan dari kode antara sebagai berikut:

- untuk memperkecil usaha dalam membangun kompilator dari sejumlah bahasa ke sejumlah mesin. Dengan adanya kode antara yang lebih *machine independent* maka kode antara yang dihasilkan dapat digunakan lagi pada mesin lainnya.
- Proses optimasi masih lebih mudah. Beberapa strategi optimisasi lebih mudah dilakukan pada kode antara daripada pada program sumber atau pada kode assembly dan kode mesin.
- Bisa melihat program internal yang mudah dimengerti. Kode antara ini akan lebih mudah dipahami dari pada kode assembly atau kode mesin.

Terdapat dua macam kode antara, yaitu Notasi Postfix dan N-Tuple

NOTASI POSTFIX

Sehari-hari kita biasa menggunakan operasi dalam notasi infix (letak operator di tengah). Pada notasi Postfix operator diletakkan paling akhir maka disebut juga dengan notasi Sufix atau Reverse Polish.
Sintaks notasi Postfix :

```
<operan><operan><operator>
```

Misalkan ekspresi :

```
(a + b)*(c + d)
```

kalau kita nyatakan dalam postfix :

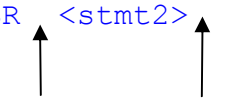
```
ab + cd + *
```

Kita dapat mengubah instruksi kontrol program yang ada ke dalam notasi Postfix. Misal :

```
IF<exp>THEN<stmt1>ELSE<stmt2>
```

diubah ke dalam Postfix

<exp><label1>BZ<stmt1><label2>BR <stmt2>



Keterangan :

BZ = branch if zero (zero = salah) {bercabang/meloncat jika kondisi yang dites salah}

BR = branch {bercabang/meloncat tanpa ada kondisi yang dites}

Arti dari notasi Postfix di atas adalah sebagai berikut.

"Jika kondisi ekspresi salah, maka instruksi akan meloncat ke Label1dan menjalankan statement2. Bila kondisi ekspresi benar, maka statement1 akan dijalankan lalu meloncat ke Label2. Label1 dan Label1 dan Label2 sendiri menunjukan posisi tujuan loncatan, untuk Label1 posisinya tepat sebelum statement2, dan Label2 setelah statement2"

Dalam implementasi ke kode antara, label bisa berupa nomor baris instruksi. Untuk lebih jelasnya bisa dilihat contoh berikut.

```
IF  a > b  THEN
     c := d
ELSE
     c := e
```

Bila diubah ke salam Postfix

```
11.  a
12.  b
13.  >
14.  22          {menunjuk label1}
15.  BZ
16.  c
17.  d
18.  :=
19.
20.  25          {menunjuk label2}
```

```
21.  BR
22.  c
23.  e
24.  :=
25.
```

Notasi Postfix di atas bisa dipahami sebagai berikut.

- Bila ekspresi (a > b) salah, maka loncat ke instruksi no.22
- Bila ekspresi (a > b) benar, tidak terjadi loncatan, instruksi berlanjut ke 16 sampai 18, lalu loncat ke 25.

Contoh lain :

```
WHILE<exp>DO<stat>
```

diubah ke postfix

```
<exp><label1> BZ<stat><label2> BR
```

label1 label2

Contoh, instruksi

```
a := 1
WHILE   a<5   DO
a := a + 1
```

diubah ke notasi postfix menjadi sebagai berikut :

```
10.  a
11.  1
12.  :=
13.  a
14.  5
15.  <
16.  26          {menunjuk label1}
```

```
17.  BZ
18.  a
19.  a
20.  1
21.  +
22.  :=
23.
24.  13          {menunjuk label2}
25.  BR
```

NOTASI N-TUPLE

Bila pada Postfix setiap baris instruksi hanya terdiri dari satu tupel, pada notasi N-tuple setiap baris terdiri dari beberapa tupel. Format umum dari Notasi N-Tuple ada sebagai berikut:

```
operator.....(N-1) operand
```

selanjutnya akan dibahas notasi 3 tupel dan 4 tupel.

TRIPLE NOTATION

Notasi tripel memiliki format sebagai berikut :

```
<operator><operan><operan>
```

contoh, instuksi :

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

Kode antara tripel :

```
1.   *, D,C
2.   /, B, E
```

```
3.   +, (1), (2)
4.   :=, A, (3)
```

operasi perkalian/pembagian lebih prioritas dibandingkan penjumlahan/pengurangan

contoh lain:

```
IF  x > y  THEN
      x:= a - b
ELSE
      x:= a + b
```

kode antara tripelnya :

```
1.   >,x,y
2.   BZ,(1),(6) {bila kondisi (1) salah satu loncat ke no (6)}
3.   -,a,b
4.   :=,x,(3)
5.   BR, ,(8)
6.   +,a,b
7.   :=,x,(6)
```

Kekurangan dari notasi tripel adalah sulit pada saat melakukan optimasi, maka dikembangkan *Indirect triples* yang memiliki dua list (senarai), yaitu list instruksi yang berisi notasi tripel dan list eksekusi yang berisi urutan eksekusinya.

Contoh :

```
A:= B+C*D/E
F:= C*D
```

List Instruksinya:

1. *, C, D
2. /, (1), E

3. +, B, (2)
4. :=, A, (3)
5. :=, F, (1)

List Eksekusinya :

1. 1
2. 2
3. 3
4. 4
5. 1
6. 5

QUADRUPLES NOTATION

Format notasi kuadrupel :

```
<operator><operan><operan><hasil>
```

hasil adalah temporary variable yang bisa ditempatkan pada memory atau register. Masalah yang ada bagaimana mengelola *temporary variable* (hasil) seminimal mungkin.

Contoh instruksi :

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

bila dibuat dalam kode antara :

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

PEMBANGKITAN KODE

Hasil dari tahapan analisis akan diterima oleh bagian pembangkitan kode (*code generator*). Disini kode antara dari program biasanya ditranslasikan ke bahasa assembly atau bahasa mesin.

Contoh :

```
(A+B)*(C+D)
```

Notasi Kuadrupel :

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

Dapat ditranslasikan ke dalam bahasa Assembly dengan akumulator tunggal :

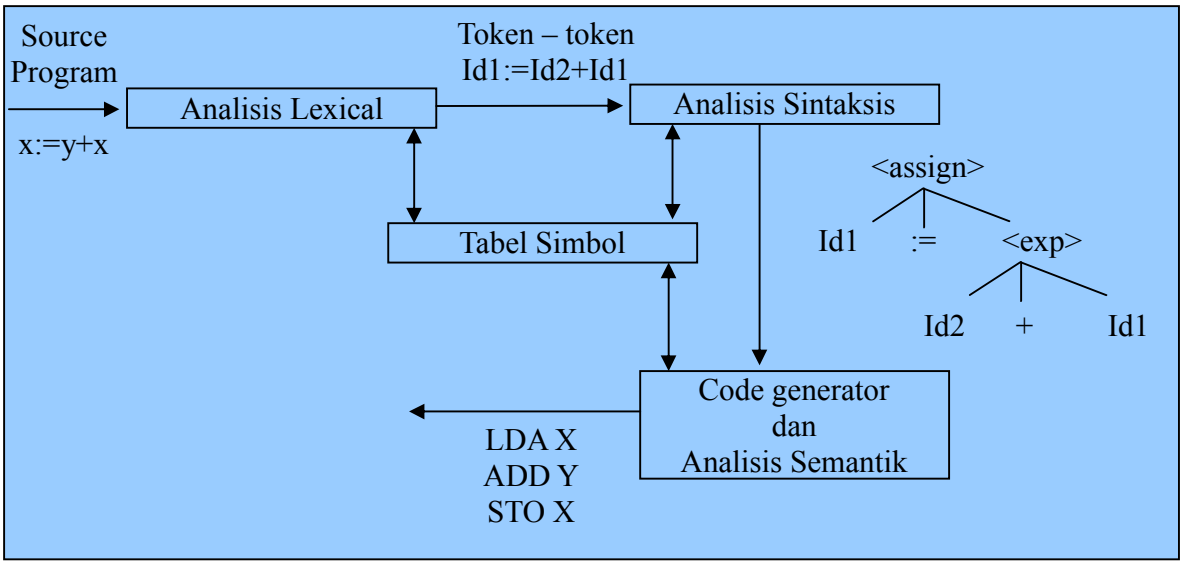
```
LDA  A    {Muat isi A ke akumulator}
ADD  B    {Tambahkan isi akumulator dengan B}
STO  T1   {Simpan isi akumulator ke T1}
LDA  C
ADD  D
STO  T2
LDA  T1
MUL  T2
STO  T3
```

Keluaran dari code generator akan diterima oleh *code optimizer*. Misalkan untuk kode assembly diatas bisa dioptimasi menjadi :

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

Notes :

- Perintah `LDA` : Memuat isi dari register/memory ke akumulator (*load to accumulator*)
- Perintah `STO` : Menyimpan isi akumulator ke register/memory (*store from accumulator*)



CARA PENANGANAN KESALAHAN

Kesalahan Program bisa merupakan :

1. Kesalahan Leksikal : THEN ditulis TEN
2. Kesalahan Sintaks : `A:=X+(B*(C+D)` {jumlah kurungnya kurang}
3. Kesalahan Semantik :
   - Tipe data yang salah.

     Contoh : Var Siswa : Integer

     Siswa := 'Yanuar' {tipe string}
   - Variabel belum didefinisikan.

     Contoh : B := B + 1 {B belum didefinisikan}

Langkah-langkah Penanganan Kesalahan adalah sebagai berikut :

- Mendeteksi Kesalahan
- Melaporkan Kesalahan
- Tindak lanjut pemulihan/perbaikan

sebuah kompilator yang menemukan kesalahan akan melakukan pelaporan kesalahan, yang biasanya meliputi :

- Kode kesalahan
- Pesan kesalahan dalam bahasa natural
- Nama dan atribut identifier
- Tipe-tipe yang terkait bila *type checking*

Contoh : Error Massage: Error 162 Jumlah := unknown identifier
artinya :

- ➔ kode kesalahan = 162
- ➔ pesan kesalahan = unknown identifier
- ➔ nama identifier = Jumlah

Adanya pesan kesalahan tersebut akan memudahkan pemrogram dalam mencari dan mengoreksi sumber dari kesalahan.

REAKSI KOMPILATOR PADA KESALAHAN

Terdapat beberapa tingkatan reaksi yang dilakukan oleh kompilator saat menemukan kesalahan, yaitu :

1. Reaksi-reaksi yang tidak dapat diterima (tidak melaporkan error);
   - Kompilator crash: berhenti atau hang.
   - Looping: kompilator masih berjalan tapi tidak pernah berakhir karena looping tak berhingga (*indefinite/onbounded loop*)
   - Menghasilkan program objek yang salah: kompilator melanjutkan proses sampai selesai tapi program objek yang dihasilkan salah. Ini berbahaya bila tidak diketahui pemrogram, karena baru akan muncul saat program dieksekusi.
2. Reaksi yang benar tapi kurang dapat diterima dan kurang bermanfaat. Kompilator menemukan kesalahan pertama, melaporkannya, lalu berhenti (*halt*). Ini bisa muncul bila pembuat kompilator menganggap jarang terjadi kemunculan error dalam program sehingga kemampuan kompilator untuk mendeteksi dan melaporkan kesalahan hanya satu untuk setiap kali kompilasi. Pemrogram akan membuang waktu untuk melakukan pengulangan kompilasi setiap kali terdapat sebuah error.

3. Reaksi-reaksi yang dapat diterima:
   - Reaksi yang sudah dapat dilakukan, yaitu kompilator melaporkan error, dan selanjutnya melakukan :
     - Recovery/pemulihan, lalu melanjutkan mencari error lain bila masih ada.
     - Repair/perbaikan kesalahan, lalu melanjutkan proses translasi dan menghasilkan program objek yang valid.

     Kebanyakan kompilator dewasa ini sudah memiliki kemampuan recovery dan repair.
   - Reaksi yang belum dapat dilakukan, yaitu kompilator mengkoreksi kesalahan, lalu menghasilkan program objek sesuai dengan yang diinginkan pemrogram. Disini komputernya sudah memiliki kecerdasan untuk mengetahui maksud pemrogram. Tingkatan respon ini belum dapat diimplementasikan pada kompilator yang ada dewasa ini.

ERROR RECOVERY

Pemulihan kesalahan bertujuan mengembalikan kondisi parser kekondisi stabil (supaya bisa melanjutkan proses parsing keposisi selanjutnya). Strategi untuk melakukan *error recovery* sebagai berikut:

1. Mekanisme *Ad Hoc*. Recovery yang dilakukan tergantung dari pembuat kompilator sendiri/spesifik dan tidak terikat pada suatu aturan tertentu. Cara ini bisa disebut juga sebagai *special purpose error recovery*.
2. *Syntax directed recovery*. Melakukan recovery berdasarkan syntax. Contoh :

```
Begin
     A:=A+1
     B:=B+1;
     C:=C+1
end;
```

   kompilator akan mengenali sebagai (dalam notasi BNF):

```
begin<statement>?<statement>;<statement>end;
```

   `'?'` akan dikenali sebagai `';'`
3. Secondary Error Recovery berguna untuk melokalisir error, dengan cara sebagai berikut:
   - Panic Mode. Maju terus dan mengabaikan teks sampai bertemu delimiter (';'). contoh,

```
IF A:=1
Kondisi := true;
```

Pada teks diatas tidak terdapat instuksi THEN, kompilator akan maju terus/skip sampai bertemu titik koma.

- Unit Deletion. Menghapus keseluruhan suatu unit sintaktik (misal: <blok>, <exp>, <statement>). Efeknya mirip dengan panic mode tetapi unit deletion memelihara kebenaran sintaksis dari source program dan mempermudah untuk melakukan error repairing lebih lanjut.

4. *Context Sensitive Recovery*. Berkaitan dengan semantik, misal bila terdapat variabel yang belum dideklarasikan (*Undefined Variable*) maka diasumsikan tipenya berdasarkan kemunculannya.

   Contoh :

   B:= 'nama'

   sementara diawal program variabel B belum dideklarasikan, maka berdasarkan kemunculannya diasumsikan variabel B bertipe string.

ERROR REPAIRING

Perbaikan kesalahan bertujuan memodifikasi source program dari kesalahan dan membuatnya valid sehingga memungkinkan kompilator untuk melakukan translasi program yang mana akan dialirkan ketahapan selanjutnya pada proses kompilasi. Mekanismenya sebagai berikut :

1. Mekanisme Ad Hoc. Tergantung dari pembuat kompilator sendiri/spesifik.
2. Syntax Directed Repar. Menyisipkan simbol terminal yang dianggap hilang atau membuang terminal penyebab kesalahan. Contoh :algoritma berikut kurang instruksi DO

```
WHILE A < 1
   I:=I+1;
```

   Kompilator akan menyisipkan DO

   contoh lain :

```
Procedure Increment;
begin
   x:=x+1;
end;
end;
```

   terdapat kelebihan simbol end, yang menyebabkan kesalahan maka kompilator akan membuangnya.

3. *Context Sensitive Repair*. Perbaikan dilakukan pada kesalahan berikut.

- Tipe Identifier. Diatasi dengan membangkitkan identifier dummy, contoh:

```
Var A:string;
begin
   A:=0;
end;
```

  kompilator akan memperbaiki kesalahan dengan membangkitkan identifier baru, misal B yang bertipe integer.

- Tipe Konstanta diatasi dengan membangkitkan konstanta baru dengan tipe yang tepat.

4. *Spelling Repair*. Memperbaiki kesalahan pengetikan pada identifier, misal:

```
WHILLE A=1 DO
```

identifier yang salah tersebut akan diperbaiki menjadi `WHILE`.

PEMBANGKITAN KODE

Kode Antara dari program biasanya ditranslasikan ke bahasa assembly atau bahasa mesin.

(A+B)*(C+D) → kode antaranya dalam notasi *Quadruples*

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

dapat ditranslasikan ke dalam bahasa assembly dengan akumulator tunggal :

```
LDA A     {muat isi A ke akumulator}
ADD B     {tambahkan isi akumulator dengan B}
STO T1    {simpan isi akumulator ke T1}
LDA C
ADD D
STO T2
LDA T1
MUL T2
STO T3
```


BAB 8

- PENAGANAN KESALAHAN

CARA PENANGAN KESALAHAN

Sebuah kompilator akan sering menemui program yang mengandung kesalahan, maka kompilator harus memiliki strategi apa yang harus dilakukan untuk menangani kesalahan - kesalahan tersebut

KESALAHAN PROGRAM

- Kesalahan Leksikal
  - Misalnya kesalahan mengeja *keyword*,
    contoh: then ditulis ten
- Kesalahan Sintaks
  - Misalnya pada operasi aritmatika kekurangan jumlah *paranthesis* (kurung).
    contoh : A:=X+(B*(C+D)
- Kesalahan Semantik
  - Tipe data yang salah, misal tipe data *integer* digunakan untuk variabel *string*.
  - Variabel belum didefinisikan tetapi digunakan dalam operasi.

PENANGANAN KESALAHAN

- Prosedur penangan kesalahan terdiri dari :
  - Mendeteksi kesalahan
  - Melaporkan kesalahan
  - Tindak lanjut perbaikan
- Pelaporan kesalahan yang dilakukan oleh sebuah kompilator meliputi :
  - Kode kesalahan
  - Pesan kesalahan dalam bahasa natural
  - Nama dan atribut *identifier*
  - Tipe – tipe yang terkait bila *type checking*

    Contoh : *Error* 162 *jumlah: unknown identifier*
    - Kode kesalahan = 162
    - Pesan kesalahan = *unknown identifier*
    - Nama identifier = *jumlah*

REAKSI KOMPILATOR PADA KESALAHAN

- Pada saat kompilator menemukan kesalahan terdapat beberapa tingkatan diantaranya adalah :
  - Reaksi yang tidak dapat diterima (tidak melaporkan *error*)
    - Kompilator *crash* : berhenti atau *hang*
    - *Looping*
    - Kompilator melanjutkan proses sampai selesai tapi program program objek yang dihasilkan salah.
  - Reaksi yang benar tapi kurang dapat diterima dan kurang bermanfaat.

    Kemampuan kompilator untuk mendeteksi dan melaporkan kesalahan hanya satu kali untuk setiap kali kompilasi.

REAKSI KOMPILATOR PADA KESALAHAN

- Reaksi yang dapat diterima
  - Reaksi yang sudah dapat dilakukan (dewasa ini), yaitu melaporkan kesalahan, dan selanjutnya melakukan:
    - *Recovery* / pemulihan, lalu melanjutkan menemukan kesalahan yang lain bila masih ada.
    - *Repair* / Perbaikan kesalahan, lalu melanjutkan proses translasi dan menghasilkan program objek yang valid
  - Reaksi yang belum dapat dilakukan (dewasa ini), yaitu kompilator mengkoreksi kesalahan, lalu menghasilkan program objek sesuai dengan yang diinginkan pemrogram.

Pemulihan Kesalahan

Tujuannya mengembalikan *parser* ke kondisi stabil (supaya bisa melanjutkan proses *parsing* ke posisi selanjutnya).

Strategi yang dilakukan sebagai berikut :

- *Mekanisme Ad Hoc*
- *Syntax Directed Recovery*
- *Secondary Error Recovery*
- *Context Sensitive Recovery*

Pemulihan Kesalahan

- *Mekanisme Ad Hoc*
  - Recovery yang dilakukan tergantung dari pembuat kompilator sendiri / Spesifik, dan tidak terikat pada suatu aturan tertentu. Cara ini biasa disebut juga *special purpose error recovery*
- *Syntax Directed Recovery*
  - Melakukan *recovery* berdasarkan *syntax*
  - Contoh : ada program

```
begin
    A:=A+1
 B:=B+1;
    C:=C+1
end;
```

kompilator akan mengenali sebagai (dalam notasi BNF)

begin < statement>?<statement>;<statement>end;

? akan diperlakukan sebagai ";"

Pemulihan Kesalahan

- *Secondary Error Recovery*
  - Berguna untuk melokalisir kesalahan, caranya :
  - *Panic mode*

    Maju terus dan mengabaikan teks sampai bertemu delimeter (misal ';')

    contoh :

```
if A := 1
    Kondisi := true;
```

    Teks diatas terjadi kesalahan karena tidak ada instruksi THEN, kompilator akan maju terus sampai bertemu ';'
  - *Unit deletion*

    Menghapus keseluruhan suatu unit sintaktik (misal: <block>,<exp>,<statement> dan sebagainya), efeknya sama dengan *panic mode* tetapi *unit deletion* memelihara kebenaran sintaksis dari source program.

Pemulihan Kesalahan

- *Context Sensitive Recovery*
  - Berkaitan dengan semantik, misal bila terdapat variabel yang belum dideklarasikan (*undifined variabel*) maka diasumsikan tipenya berdasarkan kemunculannya.

ERROR REPAIR

- Bertujuan untuk memodifikasi *source program* dari kesalahan dan membuatnya valid.
- Mekanisme *error repair* meliputi :
  - *Mekanisme Ad Hoc*

    Tergantung dari pembuat kompilator sendiri
  - *Syntax Directed Repair*

    Menyisipkan simbol terminal yang dianggap hilang atau membuang terminal penyebab kesalahan

    Contoh :

```
While a<1
        I:=I+1;
```

    Kompilator akan menyisipkan DO karena kurang simbol DO

ERROR REPAIR

- *Context Sensitive Repair*
  Perbaikan dilakukan pada kesalahan :
  - Tipe *identifier*. Diatasi dengan membangkitkan *identifier dummy*,
    misalkan :

```
Var A : string;
 begin
A:=0;
 end;
```

  - Tipe konstanta
    Diatasi dengan membangkitkan konstanta baru dengan tipe yang tepat.
- *Spelling repair*
  Memperbaiki kesalahan pengetikan pada identifier,
  misal :

```
WHILLE A = 1 DO
```

Identifier yang salah tersebut akan diperbaiki menjadi WHILE

BAB 9

- TEKNIK OPTIMASI

TEKNIK OPTIMASI

- Menghasilkan kode program dengan ukuran yang lebih kecil, sehingga lebit cepat eksekusinya.

Berdasarkan ketergantungan pada mesin :

- Machine Dependent Optimizer
- Machine Independent Optimizer

Machine Independent Optimizer

- Optimasi Lokal
  Dilakukan hanya pada suatu blok dari *source code*.
- Optimasi Global
  Dilakukan dengan analisis flow, yaitu suatu graph berarah yang menunjukkan jalur yang mungkin selama eksekusi program.

Optimasi Lokal

1. Folding

   Nilai konstanta atau ekspresi pada saat compile time diganti dengan nilai komputasinya.

   Contoh instruksi :

```
A:=2+3+B
```

   diganti menjadi

```
A:=5+B
```

Optimasi Lokal

2. Redundant – Subexpression Elimination

   Menggunakan hasil komputasi terdahulu daripada melakukan komputasi ulang.

   Contoh urutan instruksi :

```
A:=B+C
X:=Y+B+C
```

   B+C redundan, bisa memanfaatkan hasil komputasi sebelumnya, selama tidak ada perubahan nilai pada variabel.

Optimasi Lokal

3. Optimasi dalam sebuah iterasi
   - Loop Unrolling : menggantikan suatu loop dengan menulis statement dalam loop beberapa kali.

     Contoh instruksi :

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

Optimasi Lokal

dioptimasi menjadi

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

Pada instruksi pertama yang menggunakan iterasi perlu dilakukan inisialisasi setiap eksekusi loop, pengetesan, adjustment, dan operasi pada tubuh perulangan. Yang kesemuanya itu menghasilkan banyak instruksi. Karena itu dengan optimasi hanya memerlukan dua instruksi assignment.

Optimasi Lokal

- Frequency Reduction : memindahkan statement ke tempat yang lebih jarang dieksekusi.

  Contoh instruksi :

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

Optimasi Lokal

variabel X dapat dikeluarkan dari iterasi, menjadi :

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

Optimasi Lokal

4. Strength Reduction

   Mengganti suatu operasi dengan jenis operasi lain yang lebih cepat dieksekusi.

   Contoh :

   pada beberapa komputer operasi perkalian memerlukan waktu lebih banyak dari pada operasi penjumlahan.

BAB 10

- TABLE INFORMASI

TEKNIK OPTIMASI

Dependensi optimasi. Tahapan optimasi kode bertujuan untuk menghasilkan kode program yang berukuran lebih kecil dan lebih cepat eksekusinya. Berdasarkan ketergantungannya pada mesin, optimasi dibagi menjadi :

1. Machine Dependent Optimizer. Kode dioptimasi sehingga lebih efisien pad mesin tertentu. Optimasi ini memerlukan informasi mengenai feature yang ada pada mesin tujuan dan mengambil keuntungan darinya untuk menghasilkan kode yang lebih pendek atau dieksekusi lebih cepat.
2. Machine Independent Optimizer. Strategi optimasi yang bisa diaplikasikan tanpa tergantung pada mesin tujuan tempat kode yang dihasilkan akan dieksekusi nantinya. Mesin ini meliputi optimasi lokal dan optimasi global.

Optimasi Lokal adalah optimasi yang dilakukan hanya pada suatu blok dari source code, cara-caranya sebagai berikut:

1. Folding. Mengganti konstansta atau ekspresi yang bisa dievaluasi pada saat compile time dengan nilai komputasinya. Misalkan Instruksi ;
   A:=2+3+B
   bisa diganti menjadi
   A:=5+B
2. Redundant-Subexpression Elimination. Sebuah ekspresi yang sudah pernah dikomputasi, digunakan lagi hasilnya, ketimbang melakukan komputasi ulang. Misalkan terdapat urutan instruksi :
   A:=B+C
   X:=Y+B+C
   kemunculan kedua dari B+C yang redundan bisa diatasi dengan memanfaatkan hasil komputasinya yang sudah ada pada instruksi sebelumnya. Perhatikan, hal ini bisa dilakukan dengan catatan belum ada perubahan pada variabel yang berkaitan.
3. Optimisasi dalam sebuah iterasi.
   - Loop Unrolling: Menggantikan suatu loop dengan menulis statement dalam loop beberapa kali. Hal ini didasari pemikiran, sebuah iterasi pada implementasi level rendah akan memerlukan operasi sebagai berikut.
     - Inisialisasi/pemberian nilai awal pada variabel loop. Dilakukan sekali pada saat permulaan eksekusi loop.
     - Pengujian, 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).

     Dalam setiap perulangan akan terjadi pengujian dan adjusment yang menambah waktu eksekusi. Contoh pada instruksi :
     FOR I:=1 to 2 DO
     A[I]:=0;
     terdapat instruksi untuk inisialisasi I menjadi 1. serta operasi penambahan nilai/increment 1 dan pengecekan variabel I pada setiap perulangan. Sehingga untuk perulangan saja memerlukan lima instruksi, ditambah dengan instruksi assignment pada tubuh perulangan menjadi tujuh instruksi. Dapat dioptimasikan menjadi :
     A[1]:=0;
     A[2]:=0;
     yang hanya memerlukan dua instruksi assignment saja. Untuk menentukan optimasi ini

perlu dilihat perbandingan kasusnya dengan tanpa melakukan optimasi.

- Frequency Reduction: Pemindahan statement ke tempat yang lebih jarang dieksekusi. Contoh:

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

kita ,melihat bawa tidak terjadi perubahan /manipulasi pada variabel X didalam iterasi, karena itu kita bisa mengeluarkan instruksi tersebut keluar iterasi, menjadi:

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

4. Strength Reduction. Penggantian suatu operasi dengan jenis operasi lain yang lebih cepat dieksekusi. Misalkan pada beberapa komputer operasi perkalian memerlukan waktu lebih banyak untuk dieksekusi dari pada operasi penjumlahan, maka penghematan waktu bisa dilakukan dengan mengganti operasi perkalian tertentu dengan penjumlahan. Contoh lain, instruksi :
   A:=A+1;
   dapat digantikan dengan: INC(A);

OPTIMISASI GLOBAL

Optimisasi global biasanya dilakukan dengan analisis flow, yaitu suatu graf berarah yang menunjukkan jalur yang mungkin selama dieksekusi program. Kegunaannya adalah sebagai berikut:

a) Bagi pemrogram menginformasikan :
   - Unreachable/dead code: kode yang tidak akan pernah dieksekusi. Misalnya terdapat urutan instruksi:
     X:=5;
     IF X:=0 THEN
     A:=A+1
     Instruksi A:=A+1 tidak akan pernah dieksekusi
   - Unused parameter pada prosedur: parameter yang tidak akan pernah digunakan didalam prosedur. Contohnya :

```
Procedure Jumlah (a,b,c:integer);
var x: integer
begin
x:=a+b
end;
```

     kita lihat parameter c tidak pernah digunakan didalam prosedur, sehingga seharusnya tidak perlu diikutrestakan.
   - Unused Variable: Variabel yang tidak pernah dipakai dalam program. Contohnya :

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

variabel b tidak pernah digunakan dalam program sehingga bisa dihilangkan.

- Variabel yang dipakai tanpa nilai awal. Contohnya:

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

kita lihat variabel b digunakan tanpa memiliki nilai awal/belum di-assign.

b) Bagi kompilator:
   - Meningkatkan efisiensi eksekusi program.
   - Menghilangkan useless code/kode yang tidak terpakai.

TABEL INFORMASI

Tabel Informasi atau tabel simbol dibuat guna mempermudah pembuatan dan implementasi dari semantic analyzer. Tabel simbol ini mempunyai dua fungsi penting dalam proses translasi, yaitu:

TABEL INFORMASI / SIMBOL

Fungsi Tabel Informasi atau Tabel Simbol :

- ✓ Membantu pemeriksaan kebenaran semantik dari program sumber.
- ✓ Membantu dan mempermudah pembuatan *intermediate code* dan proses pembangkitan kode.

TABEL SIMBOL (lanjt)

Untuk mencapai fungsi tersebut dilakukan dengan menambah dan mengambil atribut variabel yang dipergunakan pada program dari tabel. Atribut, misalnya nama, tipe, ukuran variabel.

Tabel Simbol berisi daftar dan informasi *identifier* pokok yang terdapat dalam program sumber, disebut Tabel Pokok / Utama.

Tabel Pokok belum mengcover semua informasi, untuk itu disediakan tabel lagi sebagai pelengkap Tabel Pokok.

Untuk mengacu pada tabel simbol yang bersesuaian dengan suatu *indentifier* tertentu, maka pada Tabel Pokok harus disediakan field yang bisa menjembatani *identifier* dari Tabel Pokok ke tabel-tabel lain yang bersesuaian.

Untuk itu, pemilihan elemen tabel pada Tabel Pokok maupun tabel lainnya, merupakan sesuatu yang sangat penting.

Elemen TABEL SIMBOL (lanjt)

Elemen pada Tabel Simbol bermacam-macam, tergantung pada jenis bahasanya.
Misalnya :

1. No urut *identifier* : Menentukan nomor urut *identifier* dalam tabel simbol.
2. Nama *identifier* : Berisi nama-nama *identifier* (nama variabel, nama tipe, nama konstanta, nama procedure, nama fungsi, dll) yang terdapat pada program sumber. Nama-nama ini akan dijadikan referensi pada waktu analisa semantik, pembuatan intermediate code, serta pembangkitan kode.
3. Tipe *identifier* : Berisi keterangan/informasi tipe dari record dan string, maupun procedure dan function.
4. Object time address : address yang mengacu ke alamat tertentu.
5. Dimensi dari *identifier* yang bersangkutan.
6. Nomor baris variabel dideklarasikan.
7. Nomor baris variabel direferensikan.
8. Field link.

Implementasi Tabel Simbol

Beberapa jenis :

1. Tabel *Identifier* : Berfungsi menampung semua *identifier* yang terdapat dalam program.
2. Tabel *Array* : Berfungsi menampung informasi tambahan untuk sebuah *array*.
3. Tabel Blok : Mencatat variabel-variabel yang ada pada blok yang sama.
4. Tabel Real : menyimpan elemen tabel bernilai real.
5. Tabel String : Menyimpan informasi string.
6. Tabel *Display* : Mencatat blok yang aktif.

Tabel *Identifier*

Memiliki field :

- No urut *identifier* dalam tabel
- Nama *identifier*
- Jenis/obyektif dari *identifier :* Prosedur, fungsi, tipe, variabel, konstanta
- Tipe dari *identifier* yang bersangkutan : integer, char, boolean, array, record, file, no-type
- Level : Kedalaman *identifier* tertentu, hal ini menyangkut letak *identifier* dalam program. Konsepnya sama dengan pembentukan tree, misal *main program* = level 0. Fiel ini digunakan pada *run time* untuk mengetahui current activation record dan variabel yang bisa diakses.

Untuk *identifier* yang butuh penyimpanan dicatat pula :

- Alamat relatif/address dari *identifier* untuk implementasi
- Informasi referensi (acuan) tertentu ke alamat tabel lain yang digunakan untuk mencatat informasi-informasi yang diperlukan yang menerangkannya.
- *Link* : Menghubungkan *identifier* ke *identifier* lainnya, atau yang dideklarasikan pada level yang sama.
- Normal : Diperlukan pada pemanggilan parameter, untuk membedakan parameter *by value* dan *reference* (berupa suatu variabel boolean)

Contoh (*identifier)*

Terdapat listing program sebagai berikut :

```
Program A;
var B : integer;
  Procedure X(Z:char);
  var C : integer
  Begin
  …….
```

Tabel *Identifier* akan mencatat semua *identifier* :

0 A
1 B
2 X
3 Z
4 C

Contoh implementasi tabel *identifier* :

```
TabId: array [0..tabmax] of
  record
    name  : string;
    link  : integer;
    obj   : objek;
tipe  :  types;
    ref   : integer;
    normal      : boolean;
    level  :  0..maxlevel;
    address : integer;
end;
```

Di mana : objek = (konstant, variabel, prosedur, fungsi)
types = (notipe, int, reals, booleans, chars, arrays, record)

Tabel *Array*

Memiliki *field* :

- No urut suatu *array* dalam tabel
- Tipe dari indeks *array* yang bersangkutan
- Tipe elemen *array*
- Referensi dari elemen *array*
- Indeks batas bawah *array*
- Indeks batas atas *array*
- Jumlah elemen *array*
- Ukuran total *array* (*total size* = (atas-bawah+1) x *elemen size*)
- *Elemen size* (ukuran tiap elemen)

Tabel *Array* diacu dengan field referensi pada Tabel *Identifier*.

Contoh implementasi Tabel Array :

```
TabArray : array  [1...tabmax] of
  record
    indextype, elementype :  types;
    elemenref, low, high, elemensize, tabsize : integer
  end;
```


Tabel Blok

Memiliki *field* :

- No urut blok
- Batas awal blok
- Batas akhir blok
- Ukuran parameter / parameter size
- Ukuran variabel / variabel size
- Last variable
- Last parameter

Contoh implementasi tabel blok :

```
TabBlok: array [1..tabmax]  of
  record
    lastvar, lastpar, parsize, varsize: integer;
  end;
```

Dari contoh listing program berikut :

```
Program a;
var B: integer;
  Procedure X(Z:char);
  var C : integer
  Begin
  …….
```

implementasi tabel blok (lanjt)

Akan diperoleh, untuk blok Program A :

last variable = 2
variable size = 2 (dianggap integer butuh dua byte)
last parameter = 0 (tanpa parameter)
parameter size = 0

Untuk blok Procedure X :

last variable = 4
variable size = 2
last parameter = 3
parameter size = 1 (dianggap char butuh satu byte)

Tabel Real

Elemen tabel real :

- No urut elemen
- Nilai real suatu variabel real yang mengacu ke indeks tabel ini

Contoh implementasi tabel real :

```
TabReal : array [1..tabmax]  of real
```

(pemikiran : setiap tipe yang dimiliki oleh suatu bahasa akan memeiliki tabelnya sendiri)

Tabel String

Elemennya :

- No urut elemen
- Karakter-karakter yang merupakan konstanta

Contoh implementasi tabel string :

```
TabString: array[1..tabmax] of string
```


Tabel Display

Elemennya :

- No urut tabel
- Blok yang aktif

Pengisian tabel display dilakukan dengan konsep stack.
Urutan pengaksesan : Tabel Display – Tabel Blok – Tabel Simbol.

Contoh implementasi Tabel Display :

```
TabDisplay: array [1..tabmax]of
integer
```

Interaksi Antar Tabel

Pertama kali tabel display akan menunjuk blok mana yang sedang aktif. Dari blok yang aktif ini, akan diketahui identifier-identifier yang termasuk dalam blok tersebut. Untuk pertama kalinya, yang akan diacu adalah identifier yang paling akhir, kemudian identifier sebelumnya, dan seterusnya. Informasi suatu identifier ini mungkin belum lengkap. Untuk itu dari tabel identifier ini mungkin akan dicari kelengkapan informasi dari suatu identifier ke tabel yang sesuai (tabel real, tabel string, atau tabel array).