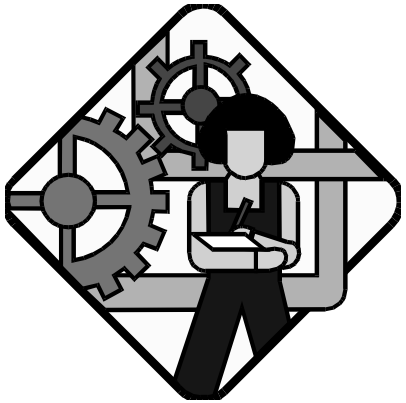
TEKNIK KOMPILASI



Hari Soetanto, S.Kom, M.Sc

Catatan kuliah (versi - 2005)

T U J U A N

- Mengetahui Penerapan konsep ilmu komputer pada perilaku komputer yaitu algoritma, arsitektur komputer, stuktur data maupun penerapan teori bahasa dan automata

- Compiler adalah merupakan konstruksi inti dari ilmu komputer

DAFTAR PUSTAKA

- Practice and principles of Compiler building with C, Henk Alblas, Albert Nymeyer, Prentice Hall, 1996
- Introduction to The theory of computation, Michael sipser, PWS publishing Company, 1997
- The Essence of Compilers, Robin Hunter,Prentice Hal Europe, 1999
- Modern Compiler Design, Dick Grune, Henri E. Bal, Et all, John Wiley & Son, 2000

Bahasan Materi Kuliah

- Pendahuluan: arti dari Kompilasi
- Translator: Compiler dan interpreter
- Bahasa Pemrograman
- Pembuatan Compiler
- Konsep bahasa dan Notasi
- Hirarki Comsky
- Aturan Produksi
- Diagram state
- Notasi BNF
- Diagram Syntax
- Kualitas Compiler

Bahasan Materi Kuliah

- Beberapa translator
- Struktur Compiler
- Lexical Analysis
- Analysis Syntax
- Analysis Semantics
- Error Handling
- Optimation
- Tabel informasi

ARTI KATA TEKNIK KOMPILASI

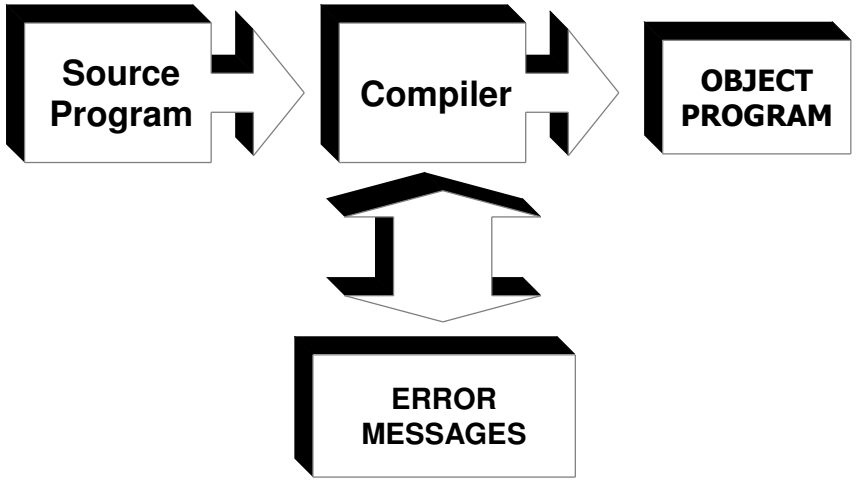
- **Teknik** :
  - Metode atau Cara
- **Kompilasi** :
  - Proses mengabungkan serta menterjermahkan sesuatu (source program) menjadi bentuk lain
- **Compile** :
  - To translate a program written in a high-level programming language into machine language.

Translator : Compiler & Interpreter

Translator :

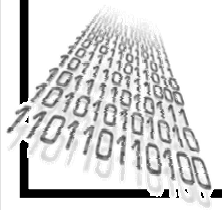
- Adalah suatu program dimana mengambil input sebuah program yang ditulis pada satu bahasa program (source language) ke bahasa lain (The object on target language)
- Jika source language adalah high level language, seperti cobol, pascal, fortran dan object language adalah low-level language atau mesin language. Maka translator seperti ini disebut COMPILER

Translator : Compiler & Interpreter



Kenapa perlu Translator ?

- Dengan bahasa mesin adalah bahasa bentuk bahasa terendah komputer, berhubungan langsung dengan bagian bagian komputer seperti bits, register & sangat primitive
- Jawaban atas pertanyaan ini akan membingungkan bagi programmer yang membuat program dengan bahasa mesin.
- Bahasa mesin adalah tidak lebih dari urutan 0 dan 1
- Instruksi dalam bahasa mesin bisa saja dibentuk menjadi *micro-code*, semacam prosedur dalam bahasa mesin
- Bagaimana dengan orang tidak mengerti bahasa mesin

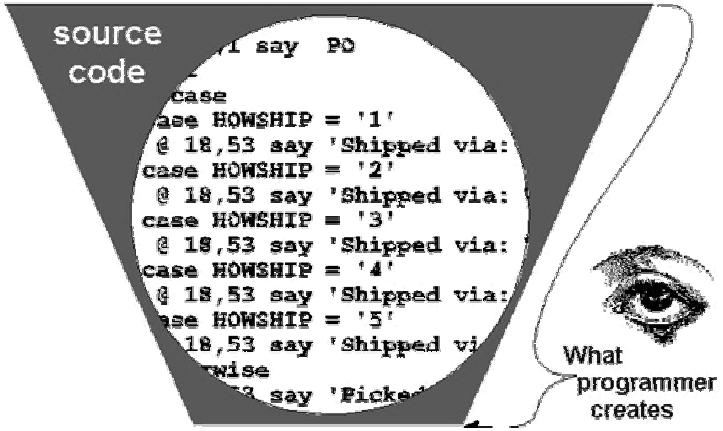


Bahasa Tingkat Tinggi (Pemrograman)

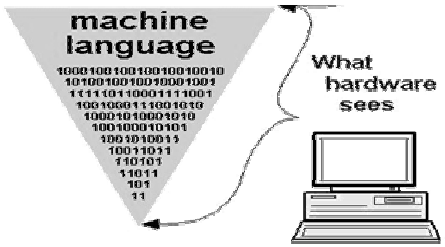
- Bahasa yang lebih dikenal oleh manusia, maksudnya adalah *statement* yang digunakan menggunakan bahasa yang dipakai oleh manusia (inggris),
- Memberikan fasilitas yang lebih banyak, seperti struktur kontrol program yang terstruktur, blok-blok serta prosedur dan fungsi-fungsi
- Progam mudah untuk di koreksi (debug)
- Tidak tergantung pada salah satu mesin
- Kontrol struktur seperti : kondisi (if .. Then.. Else), perulangan (For, while), Struktur blok (begin.. End { .. })

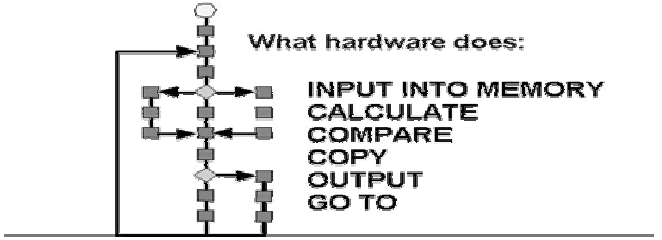
Oleh karena itu dari bahasa tingkat tinggi kedalam bahasa mesin maka dibutuhkan sesuatu untuk menterjemahkan agar mesin (komputer) mengerti apa yang inginkan oleh manusia

View dari programmer



Mesin View





Pembuatan compiler

Bahasa mesin

- Sangat sukar dan sangat sedikit kemungkinannya untuk membuat compiler dengan bahasa ini, karena manusia susah mempelajari bahasa mesin,
- Sangat tergantung pada mesin,
- Bahasa Mesin kemungkinan digunakan pada saat pembuatan Assembler

Pembuatan compiler

Assembly

- Hasil dari program mempunyai Ukuran yang relatif kecil
- Sulit dimengerti karena statement/perintahnya singkat-singkat, butuh usaha yang besar untuk membuat
- Fasilitas yang dimiliki terbatas

Pembuatan compiler

Bahasa Tingkat Tinggi (high level language)

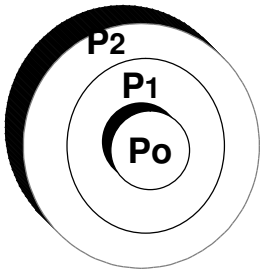
- Lebih mudah dipelajari
- Fasilitas yang dimiliki lebih baik (banyak)
- Memiliki ukuran yang relatif besar, misal membuat compiler pascal dengan menggunakan bahasa C
- Untuk mesin yang berbeda perlu dikembangkan tahapan-tahapan tambahan.
- Misal membuat compiler C pada Dos bedasarkan compiler C pada unix

Pembuatan compiler

BootStrap

- Untuk membangun sesuatu yang besar, dibangun/dibuat dulu bagian intinya (niklaus Wirth - saat membuat pascal compiler)

BootStrap



- PO dibuat dengan assembly, P1 dibuat dari P0, dan P2 dibuat dari P1, jadi compiler untuk bahasa P dapat dibuat tidak harus dengan menggunakan assembly secara keseluruhan

Contoh dari source program ke dalam kode mesin

Source code	Assembly Language	Machine language

IF COUNT =10
GOTO DONE
ELSE
GOTO AGAIN
ENDIF
	Compare A to B
	If equal go to C
	Go to D
		Compare 3477 2883
		If = go to 23883
		Go to 23343

Actual machine code

```
10010101001010001010100
10101010010101001001010
10100101010001010010010
```


Konsep dan Notasi bahasa

- Teknik Kompilasi merupakan kelanjutan dari konsep-konsep yang telah kita pelajari dalam teori bahasa dan automata
- Tata bahasa (grammar) adalah sekumpulan dari himpunan variabel-variabel, simbol-simbol terminal, simbol non-terminal, simbol awal yang dibatasi oleh aturan-aturan produksi
- Thn 56-59 Noam chomsky melakukan penggolongan tingkatan dalam bahasa, yaitu menjadi 4 class
- Penggolongan tingkatan itu disebut dengan hirarki Comsky
- 1959 Backus memperkenalkan notasi formal baru untuk syntax bahasa yang lebih spesifik
- Peter Nour (1960) merevisi metode dari syntax. Sekarang dikenal dengan BNF (backus Nour Form)

Contoh Tata Bahasa Sederhana

- <program> → **BEGIN** <Statement-list> **END**

- <Statement-list> → <statement> | <statement>**;** <statement-list>
- <statement> → <var> **:=** <expression>
- <Expression> → <term> | <term><op1> <expression>
- <term> → <factor> | <factor> <op2> <term>
- <factor> → <var> | <constant>

- <var> → **A**|**B**| ….| **Z**
- <op1> → + | - | =
- <op2> → **^** | * | /
- <constant> → <real_number> | <integer_part>

- <real_number> → <integer_part> . <fraction>
- <integer_part> → <digit> | <integer_part> < digit>
- <fraction> → <digit> | <digit> <fraction>
- <digit> → 0|1|….|9

Contoh

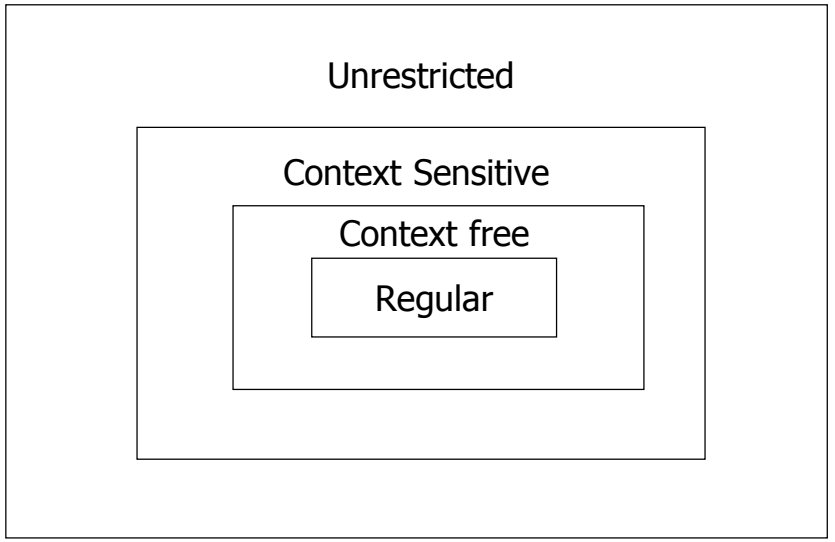
```
Begin
   A := 1;
   B := A + 2
END
```


Konsep dan Notasi bahasa

Penggolongan Chomsky

Bahasa	Mesin Automata	Aturan Produksi
Tipe 3 Atau Regular	Finite state automata (FSA) meliputi; deterministic Finite Automata (DFA) & Non Deterministic Finite Automata (NFA)	α adalah simbol variabel β maksimal memiliki sebuah simbol variabel yang bila ada terletak diposisi paling kanan
Tipe 2 Atau Contex Free	Push Down Automata	α adalah simbol variabel
Tipe 1 Atau Contex Sensitive	Linier Bounded Automata	$\lvert\alpha\rvert <= \lvert\beta\rvert$
Tipe 0 Atau Unrestricted/ Phase Structure/ natural language	Mesin Turing	Tidak ada Batasan

Hirarki Comsky



Aturan Produksi

- Aturan produksi dinyatakan dalam bentuk $\alpha \rightarrow \beta$, α menghasilkan / menurunkan β
- α simbol-simbol untuk ruas kiri dan β simbol-simbol untuk ruas kanan
- Simbol-simbol bisa berupa ternimal dan Non-terminal, dimana Non-terminal masih bisa diturunkan menjadi simbol yang lainnya
- Umumnya simbol terminal disimbolkan dengan huruf kecil (a,b,c dst), sedangkan untuk simbol non-terminal disimbolkan dengan huruf besar (A, B, C, dst)
- Contoh aturan produksi :

  $T \rightarrow a$, T menghasilkan a

  $E \rightarrow T \mid T + E$, E mengasilkan T, atau E menghasilkan T + E

Aturan Produksi

- **Tipe 0** / Unrestricted: Tidak Ada batasan pada aturan produksi

  Abc → De
- **Tipe 1** / Context sensitive: Panjang string ruas kiri harus lebih kecil atau sama dengan ruas kanan

  Ab → DeF

  CD → eF
- **Tipe 2** / Context free grammar: Ruas kiri haruslah tepat satu simbol variable

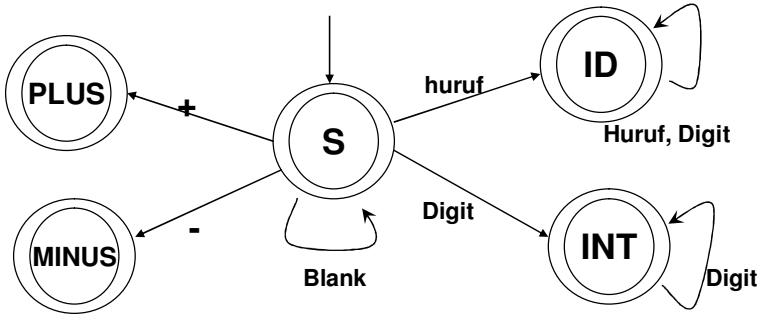
  B → CDeFg

  D → BcDe
- **Tipe 3** / Regular: Ruas kanan hanya memiliki maksimal 1 simbol non terminal dan diletakkan paling kanan sendiri

  A → e
  A → efg
  A → efgH
  C → D

Diagram State

- Digunakan untuk mendapatkan token, mempermudah melakukan analisis lexical
- **Token** adalah simbol **terminal** dari teori bahasa dan automata
- **Contoh** token ID untuk karakter huruf a-z, 0-9, token INT untuk digit, token PLUS untuk Penjumlahan dan token MINUS untuk Pengurangan



Notasi BNF (Backus-Nour Form)

- Aturan Produksi bisa dinyatakan dengan notasi BNF
- BNF menggunakan abstraksi untuk struktur syntax

  ::= sama identik dengan simbol →

  | sama dengan atau

  < > pengapit simbol non terminal

  { } Pengulangan dari 0 sampai n kali

Misalkan aturan produksi sbb:

E → T | T+E | T-E

T → a

Notasi BNFnya adalah

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

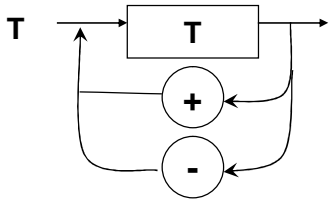
T ::= a

Diagram Syntax

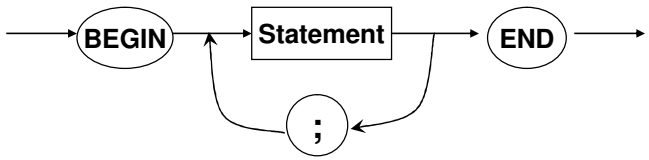
- Alat bantu (tools) dalam pembuatan parser/ analisis sintaksis
- Menggunakan simbol persegi panjang untuk non terminal
- Lingkaran untuk simbol terminal

Misalnya

$E \rightarrow T \mid T+E \mid T-E$



BNF: <Block> ::= BEGIN <statement> { SEMICOL <statement>} END



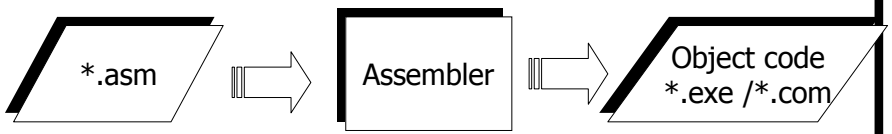
Kualitas dari Compiler

- Waktu yang dibutuhkan untuk kompilasi; tergantung dari
  - ✓ Algoritma compiler
  - ✓ Pembuat (compilator) Compiler itu sendiri
- Kualitas dari obyek program yang dihasilkan
  - ✓ Ukuran yang dihasilkan
- Fasilitas-fasilitas Integrasi yang lainnya
  - ✓ IDE (integrated Development Environment)

Ada Beberapa Translator

1. Assembler

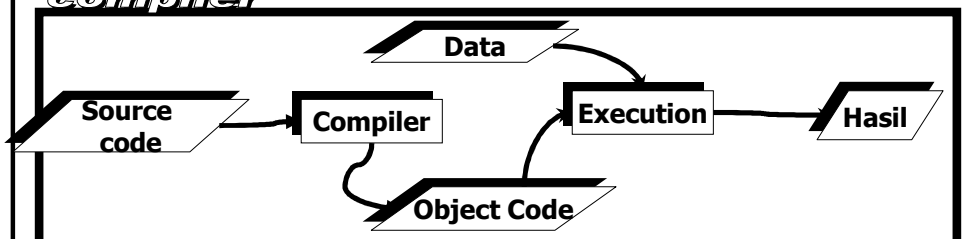
Source code adalah bahasa assembly, Object code adalah bahasa mesin



2. Compiler

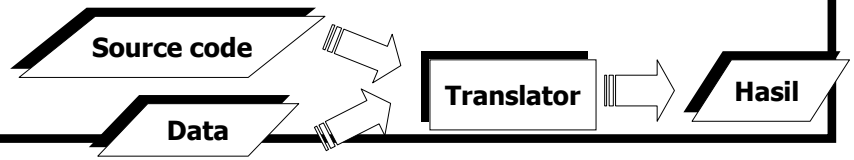
Source code adalah bahasa tingkat tinggi, object code adalah bahasa mesin atau bahasa assembly. Source code dan data diproses berbeda

compiler

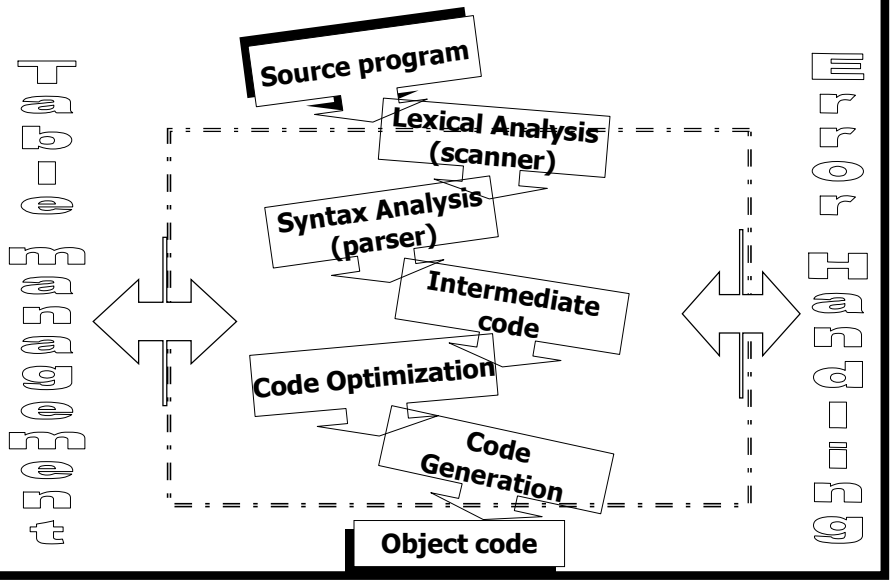


3. Interpreter

Interpreter tidak menghasilkan bentuk object code, tetapi hasil translasinya hanya dalam bentuk internal, dimana program induk harus selalu ada-berbeda dengan compiler



Struktur COMPILER



Keterangan

- Lexical Analyzer = scanner, Syntax Analyzer, dan Intermediate Code merupakan fungsi Analisis dalam compiler, yang bertugas mendekomposisi program sumber menjadi bagian-bagian kecil
- Code generation dan Code optimization adalah merupakan fungsi synthesis yang berfungsi melakukan pembangkitan/ pembuatan dan optimasi program (object program)
- Scanner adalah mengelompok-an program asal/sumber menjadi token
- Parser (mengurai) bertugas memeriksa kebenaran dan urutan dari token-token yang terbentuk oleh scanner

Lexical Analysis (scanner) - berhubungan dengan bahasa

- Mengidentifikasikan semua besaran yang membuat suatu bahasa
- Mentransformasikan ke token-token
- Menentukan jenis dari token-token
- Menangani kesalahan
- Menangani tabel simbol
- Scanner, didesign untuk mengenali - keyword, operator, identifier
- Token : separates characters of the source language into group that logically belong together
- Misalnya : konstanta, nama variabel ataupun operator dan delimiter (atau sering disebut menjadi besaran lexical)

Lexical Analysis (*Besaran leksikal*)

- Identifier dapat berupa keyword atau nama kunci, seperti IF..ELSE, BEGIN..END (pada Pascal), INTEGER (pascal), INT, FLOAT (Bhs C)
- Konstanta : Besaran yang berupa bilangan bulat (integer), bilangan pecahan (float/Real), boolean (true/false), karakter, string dan sebagainya
- Operator; Operator arithmatika (+ - * /), operator logika (< = >)
- Delimiter; Berguna sebagai pemisah/pembatas, seperti kurung-buka, kurung -tutup, titik, koma, titik-dua, titik-koma, white-space
- White Space: pemisah yang diabaikan oleh program, seperti enter, spasi, ganti baris, akhir file

Lexical Analysis - Contoh

- **Contoh 1:**

ada urutan karakter yang disebut dengan statement

fahrenheit := 32 + celcius * 1.8,

Maka akan diterjemahkan kedalam token-token seperti dibawah ini

identifier	→	fahrenheit
operator	→	:=
integer	→	32
operator penjumlahan	→	+
Identifier	→	celcius
operator perkalian	→	*
real / float	→	1.8

Lexical Analysis - Contoh 2

- Setiap bentuk dari token di representasi sebagai angka dalam bentuk internal, dan angkanya adalah unik
- ***Misalnya*** nilai 1 untuk variabel, 2 untuk konstanta, 3 untuk label dan 4 untuk operator, dst
- Contoh instruksi :

Kondisi : IF A > B THEN C = D;

- Maka scanner akan mentransformasikan kedalam token-token, sbb:

Kondisi	3	**B**	1
:	26	**THEN**	21
IF	20	**C**	1
A	1	**D**	1
>	15	**;**	27

Token-token ini sebagai inputan untuk *syntax Analyser* , token-token ini bisa berbentuk pasangan item. Dimana Item pertama menunjukkan alamat atau lokasi dari token pada tabel simbol. Item kedua adalah representasi internal dari token. Semua token direpresentasikan dengan informasi yang panjangnya tetap (konstan), suatu alamat (address atau pointer) dan sebuah integer (bilangan bulat)

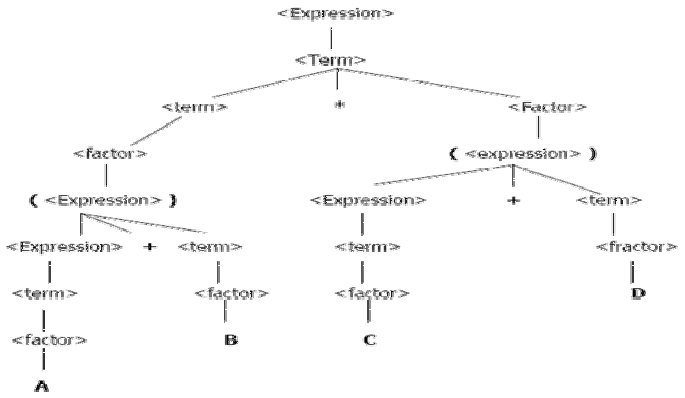
Syntax Analyzer

- Pengelompokan token-token kedalam class syntax (bentuk syntax), seperti procedure, Statement dan expression
- Grammar : sekumpulan aturan-aturan, untuk mendefinisikan bahasa sumber
- Grammar dipakai oleh syntax analyser untuk menentukan struktur dari program sumber
- Proses pen-deteksian-nya (pengenalan token) disebut dengan parsing
- Maka Syntax analyser sering disebut dengan parser
- Pohon sintaks yang dihasilkan digunakan untuk semantics analyser yang bertugas untuk menentukan 'maksud' dari program sumber, misalnya operator penjumlahan maka semantics analyser akan mengambil aksi apa yang harus dilakukan

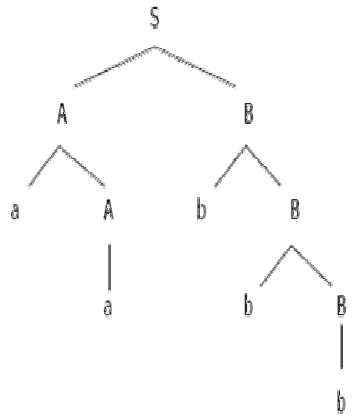
Contoh

- Terdapat statement : **(A + B) * (C + D)**
- Akan menghasilkan bentuk sintaksis:

  <factor>, <term> & <expression>



Syntax tree



- Pohon sintaks/ Pohon penurunan (syntax tree/ parse tree) beguna untuk menggambarkan bagaimana memperoleh suatu *string* dengan cara menurunkan simbol-simbol variable menjadi simbol-simbol terminal.
- Misalnya: $S \rightarrow AB$
  $A \rightarrow aA \mid a$
  $B \rightarrow bB \mid B$

Penurunan untuk menhasilkan string aabbb

Parsing atau Proses Penurunan

Parsing dapat dilakukan dengan cara :

- Penurunan terkiri (*leftmost derivation*) : simbol variable yang paling kiri diturunkan (tuntas) dahulu
- Penurunan terkanan (*rightmost derivation*): variable yang paling kanan diturunkan (tuntas) dahulu
- Misalkan terdapat ingin dihasilkan string *aabbaa* dari

  context free language: $S \rightarrow a\,AS \mid a$,

  $A \rightarrow SbA \mid ba$

Parsing atau Proses Penurunan

Penurunan kiri :

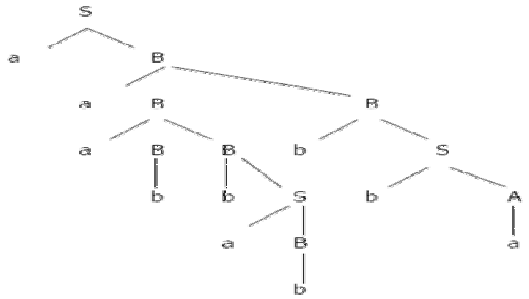
S => aAS
=> aSbAS
=> aabAS
=> aaabbaS
=> aabbaa

Penurunan kanan :

S => aAS
=> aAa
=> aSbAa
=> aSbbaa
=> aabbaa

Parsing

Misalnya: S -> aB | bA
A -> a | aS |bAA
B -> b | bS | aBB



Penurunan untuk string aaabbabba

Dalam hal ini perlu untuk melakukan percobaan pemilihan aturan produksi yang bisa mendapatkan solusi

Metode Parsing

Perlu memperhatikan 3 hal:

- Waktu Eksekusi
- Penanganan Kesalahan
- Penanganan Kode

Parsing digolongkan menjadi:

- ***Top-Down***

  Penelusuran dari *root ke leaf* atau dari simbol awal ke simbol terminal

  metode ini meliputi:

  - *Backtrack/backup : Brute Force*
  - *No backtrack : Recursive Descent Parser*

- ***Bottom-Up***

  Metode ini melakukan penelusuran dari *leaf ke root*

Parsing: Brute force

- Memilih aturan produksi mulai dari kiri
- Meng-expand simbol non terminal sampai pada simbol terminal
- Bila terjadi kesalahan (string tidak sesuai) maka dilakukan *backtrack*
- Algoritma ini membuat pohon parsing secara top-down, yaitu dengan cara mencoba segala kemungkinan untuk setiap simbol non-terminal
- Contoh suatu language dengan aturan produksi sebagai berikut

  $S \rightarrow aAd \mid aB$

  $A \rightarrow b \mid c$

  $B \rightarrow ccd \mid ddc$

- Misal ingin dilakukan parsing untuk string 'accd'

Parsing: Brute force

(i) S

(ii) S

a A d

(iii) S

a A d

b

Terjadi kegagalan (iii), dilakukan back track

(iv) S

a A d

c

(v) S

a B

(vi) S

a B

c c d

Terjadi kegagalan lagi (iv), dilakukan back-track

Parsing: Brute force

Kelemahan dari metode-metode *brute-force*

- Mencoba untuk semua aturan produksi yang ada sehingga menjadi lambat (waktu eksekusi)
- Mengalami kesukaran untuk melakukan pembetulan kesalahan
- Memakan banyak memakan memori, dikarenakan membuat *backup* lokasi *backtrack*
- Grammar yang memiliki R*ekursif Kiri* tidak bisa diperiksa, sehingga harus diubah dulu sehingga tidak rekursif kiri, Karena rekursif kiri akan mengalami ***Loop*** yang terus-menerus

Brute force : Contoh

Terdapat grammar/tata bahasa G = (V,T,P,S), dimana

V= (“E”,”T”,”F”) Simbol NonTerminal (variable)

T= (“i”,”*”,”/” ,”+”,”-”) Simbol Terminal

S=”E” Simbol Awal / Start simbol

String yang diinginkan adalah ***i * i***

aturan produksi (P) yang dicobakan adalah

1. $E \rightarrow T \mid T + E \mid T - E$

   $T \rightarrow F \mid F * T \mid F / T$

   $F \rightarrow i$

accept (diterima)

Brute force : Contoh

2. $E \rightarrow T \mid E+T \mid E-T$

$T \rightarrow F \mid T* F \mid T / F$

$F \rightarrow i$

accept (diterima)

- Meskipun ada rekursif kiri, tetapi tidak diletakkan sebagai aturan yang paling kiri

3. $E \rightarrow E+T \mid E-T \mid T$

$T \rightarrow T* F \mid T / F \mid F$

$F \rightarrow i$

Rekursif kiri, program akan mengalami loop

Brute force : Aturan produksi

Aturan Produksi yang rekursif memiliki ruas kanan (hasil produksi) yang memuat simbol variabel pada ruas kiri

Sebuah produksi dalam bentuk

$A \rightarrow \beta A$ merupakan produksi rekursif kanan

β = berupa kumpulan simbol variabel dan terminal

contoh: $S \rightarrow d S$

$B \rightarrow ad B$

bentuk produksi yang rekursif kiri

$A \rightarrow A \beta$ merupakan produksi rekursif Kiri

contoh:

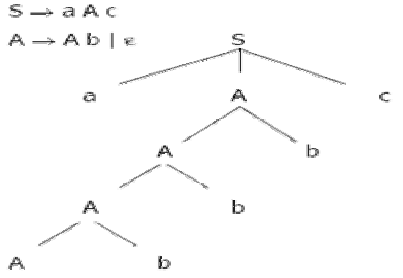
$S \rightarrow S d$

$B \rightarrow B ad$

Aturan produksi : Brute force

Produksi yang rekursif kanan akan menyebabkan penurunan tumbuh kekanan, Sedangkan produksi yang rekursif kiri akan menyebabkan penurunan tumbuh ke kiri.

Contoh: Context free Grammar dengan aturan produksi sebagai berikut:



Aturan produksi : Brute force

Dalam Banyak penerapan tata-bahasa, ***rekursif kiri*** tidak diinginkan, Untuk menghindari penurunan kiri yang looping, perlu dihilangkan sifat rekursif, dengan langkah-langkah sebagai berikut:

- Pisahkan Aturan produksi yang rekursif kiri dan yang tidak; misalnya

  Aturan produksi yang **rekursif kiri**

  $A \rightarrow A \alpha 1 \mid A \alpha 2 \mid \ldots \mid A \alpha n$

  Aturan produksi yang ***tidak*** **rekursif kiri**

  $A \rightarrow \beta 1 \mid \beta 2 \mid \ldots \mid \beta n$

Aturan produksi : Brute force

- lakukan per-ganti-an aturan produksi yang rekursif kiri, sebagai berikut:

  1. $A \rightarrow \beta 1\ Z \mid \beta 2\ Z \mid \ldots \mid \beta n\ Z$
  2. $Z \rightarrow \alpha 1 \mid \alpha 2 \mid \ldots \mid \alpha n$
  3. $Z \rightarrow \alpha 1\ Z \mid \alpha 2\ Z \mid \ldots \mid \alpha n\ Z$

Aturan produksi : Brute force

- Pergantian dilakukan untuk setiap aturan produksi dengan simbol ruas kiri yang sama, bisa muncul variabel Z1, Z2 dst, sesuai dengan variabel yang menghasilkan rekurisif kiri

Contoh: Tata Bahasa Context free

S → Sab | aSc | dd | ff | Sbd

- Pisahkan aturan produksi yang rekursif kiri

S → Sab | Sbd

Ruas Kiri untuk S: α1=ab , α2=bd

- Aturan Produksi yang tidak rekursif kiri

S → aSc | dd | ff

dari situ didapat untuk Ruas Kiri untuk S: β1 = aSc, β2 = dd, β3= ff

Aturan produksi : Brute force

- Langkah berikutnya adalah penggantian yang rekursif kiri

S → Sab | Sbd, dapat digantikan dengan

1. S → aSc***Z1*** | dd***Z1*** | ff***Z1***
2. Z1 → ab | bd
3. Z1 → ab***Z1*** | bd***Z1***

- Hasil akhir yang didapat setelah menghilangkan rekursif kiri adalah sebagai Berikut:

S → aSc | dd | ff

S → aSc***Z1*** | dd***Z1*** | ff***Z1***

Z1 → ab | bd

Z1 → ab***Z1*** | bd***Z1***

Aturan produksi : Brute force

- Kalau pun tidak mungkin menghilangkan rekursif kiri dalam penyusunan aturan produksi maka produksi rekursif kiri diletakkan pada bagian belakang atau terkanan, hal ini untuk menghindari looping pada awal ***proses parsing***
- Metode ini jarang digunakan, karena semua kemungkinan harus ditelusuri, sehingga butuh waktu yang cukup lama serta memerlukan memori yang besar untuk penyimpanan stack (backup lokasi backtrack)
- Metode ini digunakan untuk aturan produksi yang memiliki alternatif yang sedikit

Parsing: Recursive Descent Parser

Parsing dengan ***Recursive Descent Parser***

- Salah satu cara untuk meng-aplikasikan bahasa context free
- Simbol terminal maupun simbol variabelnya sudah bukan sebuah karakter
- Besaran leksikal sebagai simbol terminalnya, besaran syntax sebagai simbol variablenya /non terminalnya
- Dengan cara penurunan secara recursif untuk semua variabel dari awal sampai ketemu terminal
- Tidak pernah mengambil token secara mumdur (back tracking)
- Beda dengan turing yang selalu maju dan mundur dalam melakukan *parsing*

Parsing: Bottom-Up Parsing

Kelemahan dari metode-metode *brute-force*

- *Mencoba untuk semua aturan produksi yang ada sehingga menjadi lambat (waktu eksekusi)*
- *Mengalami kesukaran untuk melakukan pembetulan kesalahan*
- *Memakan banyak memakan memori, dikarenakan membuat backup lokasi backtrack*
- *Grammar yang memiliki Rekursif Kiri tidak bisa diperiksa, sehingga harus diubah dulu sehingga tidak rekursif kiri, Karena rekursif kiri akan mengalami **Loop** yang terus-menerus*

Semantics Analyser

- Proses ini merupakan proses kelanjutan dari proses kompilasi sebelumnya, yaitu analisa leksikal (scanning) dan analisa sintaks (parsing)
- Bagian terakhir dari tahapan analisis adalah analisis semantik
- Memanfaatkan pohon sintaks yang dihasilkan dari *parsing*
- Proses analisa sintak dan analisa semantik merupakan dua proses yang sangat erat kaitannya, dan sulit untuk dipisahkan

Semantics Analyser

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

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

Semantics Analyser

Untuk mengetahui makna, maka routin ini akan memeriksa:

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

Semantics Analyser

Pengecekan yang dilakukan dapat berupa:

- Memeriksa penggunaan nama-nama (keberlakuannya)
  - **Duplikasi**
    Apakah sebuah nama terjadi pendefinisian lebih dari dua kali. Pengecekan dilakukan pada bagian pengelolaan block
  - **Terdefinisi**
    Apakah nama yang dipakai pada program sudah terdefinisi atau belum. Pengecekan dilakukan pada semua tempat kecuali block

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

Semantics Analyser

Contohnya;

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

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

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

Intermediate Code

- Memperkecil usaha dalam membuat compilator dari sejumlah bahasa ke sejumlah mesin
- Lebih *Machine Independent*, hasil dari intermediate code dapat digunakan lagi pada mesin lainnya
- Proses Optimasi lebih mudah. Lebih mudah dilakukan pada intermediate code dari pada *program sumber* (source program) atau pada kode *assembly* dan kode mesin
- Intermediate code ini lebih mudah dipahami dari pada *kode assembly* atau kode mesin
- Kerugiannya adalah melakukan 2 kali transisi, maka dibutuhkan waktu yang relatif lama

Intermediate Code

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

Notasi **POSTFIX**

<Operand> <Operand> < Operator>

Misalnya :

(a +b) * (c+d)

maka Notasi postfixnya

ab+ cd+ *

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

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

POSTFIX

Diubah ke postfix menjadi ;

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

BZ : Branch if zero (salah)
BR: melompat tanpa harus ada kondisi yang ditest

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

POSTFIX

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

Dalam bentuk Postfix

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

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

POSTFIX

Contoh:

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

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

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

Dalam bentuk Postfix

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

TRIPLES NOTATION

Notasi pada triple dengan format

<operator> <operand> <operand>

Contoh:

A := D * C + B / E

Jika dibuat intermidiate code triple:

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

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

TRIPLES NOTATION

Contoh lain:

IF X > Y **THEN**

X := a - b

ELSE

X := a + b

Intermidiate code triple:

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

TRIPLES NOTATION

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

A := B + C * D / E

F := C * D

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

Quardruples Notation

Format dari quardruples adalah

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

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

Contoh:

A := D * C + B / E

Jika dibuat intermidiate codenya :

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

Quardruples Notation

- Hasil dari tahapan anlisis diterima oleh code generator (pembangkit kode)
- Intermediate code ditansfromasikan kedalam bahasa assembly atau mesin
- Misalnya

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

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

Dapat ditranslasikan kedalam bahasa assembly dengan accumulator tunggal:

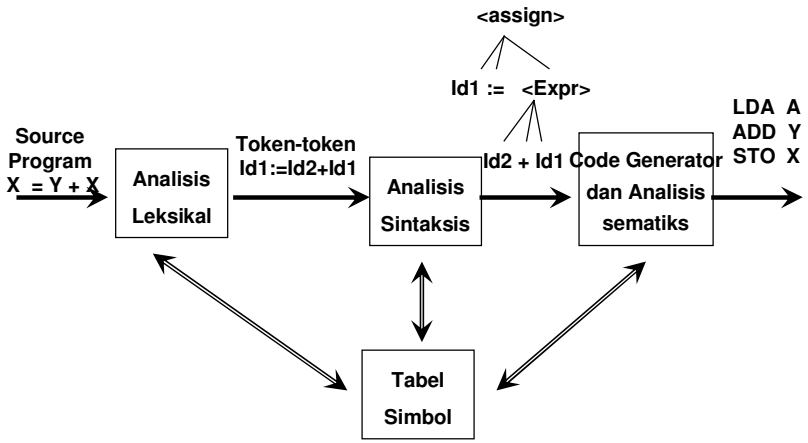
Code Generator

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

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

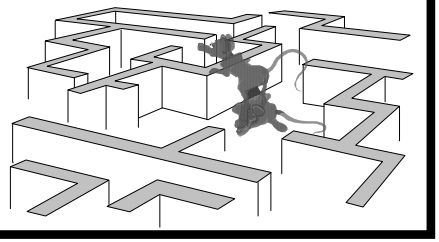
```
LDA   A
ADD   B
STO   T1
LDA   C
ADD   D
MUL   T1
STO   T2
```


Perjalanan sebuah intruksi



Error Handling

- Kesalahan Program
- Penanganan Kesalahan
- Reaksi Compiler Pada kesalahan
- Error Recovery
- Error repair



Kesalahan Program

Kesalahan Program dapat berupa

- ✓ Kesalahan **leksikal**
- ✓ Kesalahan **Sintaks**
- ✓ Kesalahan Semantics

Error Handling - Kesalahan Program

Kesalahan Program dapat berupa

- ✓ Kesalahan **leksikal**
  - Kesalahan dalam mengetik/mengeja
  - Misal **THEN** dituliskan dengan **TEN** atau **THN**
- ✓ Kesalahan **Sintaks**
  - misalnya dalam operasi aritmatika dengan tanda kurung yang jumlahnya kurang, contoh
  - A:= X + (B * (C+D)
- ✓ Kesalahan Semantics

Error Handling - Kesalahan Program

- ✓ Kesalahan **Semantics**
  - *Tipe data yang salah*

    Contoh : int c;

    c = 1.5 * 0.78
  - *Variable belum didefinisikan*

    Misal : B := B + 1

    tetapi b belum didefinisikan

Error Handling - Penanganan Kesalahan

Langkah-langkah:

- ✓ Mendeteksi kesalahan
- ✓ Melaporkan kesalahan
- ✓ Tindak lanjut perbaikan

Error Handling - Penanganan Kesalahan

- ✓ Misal: compiler menemukan kesalahan, yang bisa meliputi
  - *Kode kesalahan*
  - *Pesan Kesalahan dalam bahasa alami*
  - *Nama dan atribut identifier*
  - contoh : error 162 Jumlah: Unknow identifier
  - Dapat diartikan: Kode kesalahan =162, pesan kesalahan = *unknown identifier*, nama *identifier* = jumlah

Error Handling - Reaksi terhadap Kesalahan

Ada Beberapa reaksi yang dilakukan oleh compiler

- ✓ ***Reaksi-reaksi yang tidak dapat diterima***
- ✓ ***Reaksi yang benar, tapi kurang dapat diterima dan kurang bermanfaat***

Error Handling - Reaksi terhadap Kesalahan

Ada Beberapa reaksi yang dilakukan oleh compiler

- ✓ ***Reaksi-reaksi yang tidak dapat diterima***
  - *Compilator crash*: Berhenti atau hang
  - *Looping :* compilator tidak bisa berhenti (infinite/onbounded loop)
  - Menghasilkan Obyek program yang salah : berbahaya, bisa diketahui/muncul setelah program dieksekusi

Error Handling - Reaksi terhadap Kesalahan

Ada Beberapa reaksi yang dilakukan oleh compiler

- ✓ ***Reaksi yang benar, tapi kurang dapat diterima dan kurang bermanfaat***
  - Compilator menemukan kesalahan pertama, melaporkannya, lalu berhenti (halt)
  - Pemrogram membuang waktu untuk melakukan pengulangan compilasi untuk setiap kali terdapat sebuah error

Error Handling - Reaksi terhadap Kesalahan

- ✓ **Reaksi-reaksi yang dapat diterima**
  - Reaksi yang sudah dapat dilakukan ; *Compilator* melaporkan Error
    - Recovery : Pemulihan
    - Repair : Perbaikan
  - Reaksi yang belum dapat dilakukan
    - Compiler mengkoreksi kesalahan
    - Menghasilkan obyek program sesuai yang diinginkan pemrogram
    - Compiler memiliki kemampuan untuk 'mengetahui' maksud dari pemrogram
    - Belum diimplementasikan pada program (sekarang ini)

Error Handling - Error Recovery

Bertujuan mengembalikan *parser* ke kondisi stabil agar supaya dapat melanjutkan proses *parsing* ke posisi selanjutnya.

✓ ***Mekanisme Ad Hoc***

- Recovery yang dilakukan tergantung dari si pembuat compiler
- Tidak terikat pada suatu aturan tertentu
- Disebut juga dengan istilah *purpose error recovery*

✓ ***Syntax directed Recovery***

misal begin

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


Error Handling - Error Recovery

Pada contoh diatas, compiler akan mengenali sebagai (dalam Notasi BNF)

*begin <statement> **? , statement> ; <statement> end;***

*? Akan diperlakukan sebagai **';'***

- **Second Error Recovery** : untuk melokalisir kesalahan
  - ***Panic Mode***
    - Maju terus sampai ketemu delimiter
    - Contoh : IF A = 1 Kondisi := true;
    - Pada kondisi diatas **THEN** tidak ada, compiler melanjutkan sampai ketemu delimiter (;)
  - ***Unit Deletion***
    - Menghapus keseluruhan suatu unit sintaksik (misalnya : <block>, <exp>, <statement> dan sebagainya
    - Mempermudah untuk melakukan error repairing

Error Handling - Error Recovery

- **Context Sensitive Recovery**
  - Berkaitan dengan semantics
  - contoh : B := 'Budi Luhur'
  - Pada awal program variabel B belum dideklarasikan, maka berdasarkan permunculannya maka diasumsikan variabel B bertipe *string*

Error Handling - Error repair

Memperbaiki kesalahan dan membuat source program valid (memodifikasi)

✓ Mekanisme Ad Hoc

- Tergantung pada sipembuat compiler

✓ Syntax directed Repair

- Menyisipkan / membuang simbol terminal yang dianggap hilang atau yang menyebabkan error
- contoh **WHILE** A < 1

  I := I = 1;
- compiler akan menyisipkan **DO**

Error Handling - Error repair

- Contoh lain

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

- Kelebihan simbol **end**, yang menyebabkan kesalahan, maka compiler akan membuangnya

Error Handling - Error repair

- ✓ **Context Sensitive Repair**
  - Tipe identifier: membuat *identifier dummy*

```
var A : String
begin
 A := 0;
end
```

maka compilator akan memperbaiki kesalahan dengan membuat *identifier baru* , misalnya B bertipe integer

  - Spelling Repair: memperbaiki kesalahan pengetikan pada identifier, misalnya:

WHILLE A = 1 DO

identifier yang salah tersebut diperbaiki menjadi **WHILE**

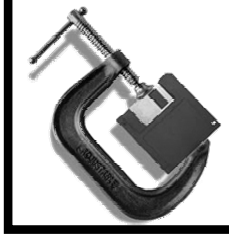
Teknik Optimasi

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

- ***Dependensi Optimasi***

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

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



Teknik Optimasi : Optimasi Lokal

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

- ✓ ***Folding***

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

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

5 dapat mengantikan ekspresi 2 + 3

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

hasilnya digunakan lagi dari pada dilakukan computasi ulang, contoh:

A:= **B + C**

X := Y + **B + C**

Teknik Optimasi : Optimasi Lokal

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

Teknik Optimasi : Optimasi Lokal

- Contoh :

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

  dapat dioptimasikan menjadi
  A[1] := 0;
  A[2] := 0;

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

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


Teknik Optimasi : Optimasi Lokal

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

    A:= A + 1
  - dapat digantikan dengan

    INC(A)

Teknik Optimasi : Optimasi GLobal

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

ada dua kegunaan yaitu bagi programmer dan compiler itu sendiri

- ✓ Bagi Programmer
  - ***Unreachable/dead code***: Kode yang tidak pernah dieksekusi
  - misalnya :

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

Instruksi
A := A + 1 tidak pernah dikerjakan

Teknik Optimasi : Optimasi GLobal

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

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

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

Teknik Optimasi : Optimasi GLobal

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

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

- B tidak pernah digunakan

Teknik Optimasi : Optimasi GLobal

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

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

  - variabel b digunakan tetapi tidak memiliki harga awal

✓ ***Bagi Compiler***
- Meningkatkan efisiensi eksekusi program
- Menghilangkan useless code/kode yang tidak terpakai

Tabel Informasi

Dua fungsi penting Tabel Informasi

- Untuk membantu pemeriksaan kebenaran semantik dari program sumber
- Untuk membantu dan mempermudah dalam pembuatan intermediate code dan proses pembuatan kode-kode (pembangkitan kode)

Tabel Informasi

Secara umum, sebuah tabel simbol bisa memiliki elemen-elemen tabel sebagai berikut, meskipun tidak semuanya dipergunakan oleh semua compiler

- No.urut identifier: menentukan nomor urut pada tabel simbol
- Nama identifier

Tabel Informasi - Kegunaan

- Tipe identifier
- Object time address
- Dimensi dari identifier yang bersangkutan
- Nomor baris variabel yang dideklarasikan
- Nomor baris variabel yang direferensikan
- Field link

Tabel Informasi - Implementasi

Ada beberapa jenis Tabel Informasi

- ***Tabel identifier***; berfungsi menampung semua identifier yang terdapat dalam program
- ***Tabel Array:*** berfungsi menampung informasi tambahan untuk sebuah array
- ***Tabel blok***: mencatat varibel-variabel yang ada pada blok yang sama
- ***Tabel Real***: Menyimpan elemen tabel bernilai real
- ***Tabel string***: menyimpan informasi string
- ***Tabel display***: mencatat blok yang aktif

Tabel Informasi - Identifier

Tabel Identifier memiliki;

- No Urut identifier dalam tabel
- Nama Identifier
- Jenis dari identifier; seperti Prosedur, fungsi, tipe variabel dan konstanta
- Tipe dari identifier yang bersangkutan; seperti Integer (bilangan bulat), Char, boolean , array, record, file
- level dari identifier (depth of block); hal ini menyangkut letak identifier dalam program, konsepnya sama dengan pembentukan *tree*, misalnya main program level 0

Tabel Informasi - Identifier

Untuk **identifier**, pencatatan dapat berupa seperti;

- Alamat *relatif/address* dari identifier untuk implementasi
- Informasi referensi dari identifier terntentu ke alamat tabel identifier yang lainnya
- *link;* menghubung antar identifier
- Normal: digunakan pada pemanggilan parameter, untuk membedakan parameter by value dan by reference
- Contoh (dalam pascal)

Tabel Informasi - Identifier

```
Program A;
Var B : Integer;
    Procedure  X (Z: char)
    var C : Integer
    begin
      ....dst
```

Tabel identifier akan mencatat semua identifier;

0	A
1	B
2	X
3	Z
4	C

Tabel Informasi - contoh

```
TabId: Array [0..tabmax] of  record
         nama        : String;
         link        : integer;
         Obj         : object;

      Tipe : Types;
         ref         : Integer;
        normal       : Boolean;
        Level        : 0.. Maxlevel;
        address      : Integer;
      End
```

Dimana

objek =(konstant, variabel, prosedur, fungsi)

Types = (notipe, int, reals, booleans, chars, arrays, record

Tabel Informasi - Array

Tabel Array

dipergunakan untuk menyimpan informasi suatu identifier yang bertipe array, tabel ini memilik field:

- No. Urut suatu array dalam tabel
- Tipe dari indeks array yang bersangkutan
- Tipe element array
- Referensi dari elemen array
- Index batas atas dan bawah array
- Jumlah elemen array
- Ukuran total array (total = atas - bawah + 1) x elemen size
- Elemen size

Tabel Informasi - Block

Tabel Blok

Dipergunakan untuk menyimpan informasi blok-blok yang ada pada tabel utama. Berisikan field

- no urut blok
- batas awal blok
- batas akhir blok
- ukuran parameter/parameter size
- ukuran variabel/ variabel size
- last variabel
- last parameter

Contoh

```
Program A
Var B : Integer;
  Procedure X (Z:char);
   Var C : Integer;
    begin
    ....
```

Untuk	Blok **A**	Blok **B**
last variable	= 2	4
Variable size	= 2 (dianggap integer 2 byte) 2	
Last parameter	= 0 (tanpa parameter)	3
parameter size	= 0	1 (char butuh 1 byte)

Tabel Informasi - Implementasi

Tabel Real

Dipergunakan untuk menyimpan nilai dari suatu identifier yang bertipe real (pecahan). Elemen-elemen dari tabel ini adalah sebagai berikut;

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

Pemikirannya disini setiap tipe yang memiliki oleh suatu bahasa akan memiliki tabelnya sendiri

Tabel Informasi - Implementasi

Tabel String

Dipergunakan untuk menyimpan informasi string yang terdapat pada program sumber. Elemen-elemen yang terdapat dalam tabel ini adalah:

- no Urut elemen
- Karakter-karakter yang merupakan konstanta

Tabel Informasi - Implementasi

Tabel Display

menyimpan informasi-informasi mengenai blok-blok yang lagi aktif. Elemen-elemen yang terdapat dalam tabel ini adalah:

- No Urut tabel
- Blok yang aktif

Pengisian tabel display dilakukan dengan konsep stack