TEKNIK KOMPILASI

Ernastuti & Sulistyo P

MATERI

- 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

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

Pendahuluan

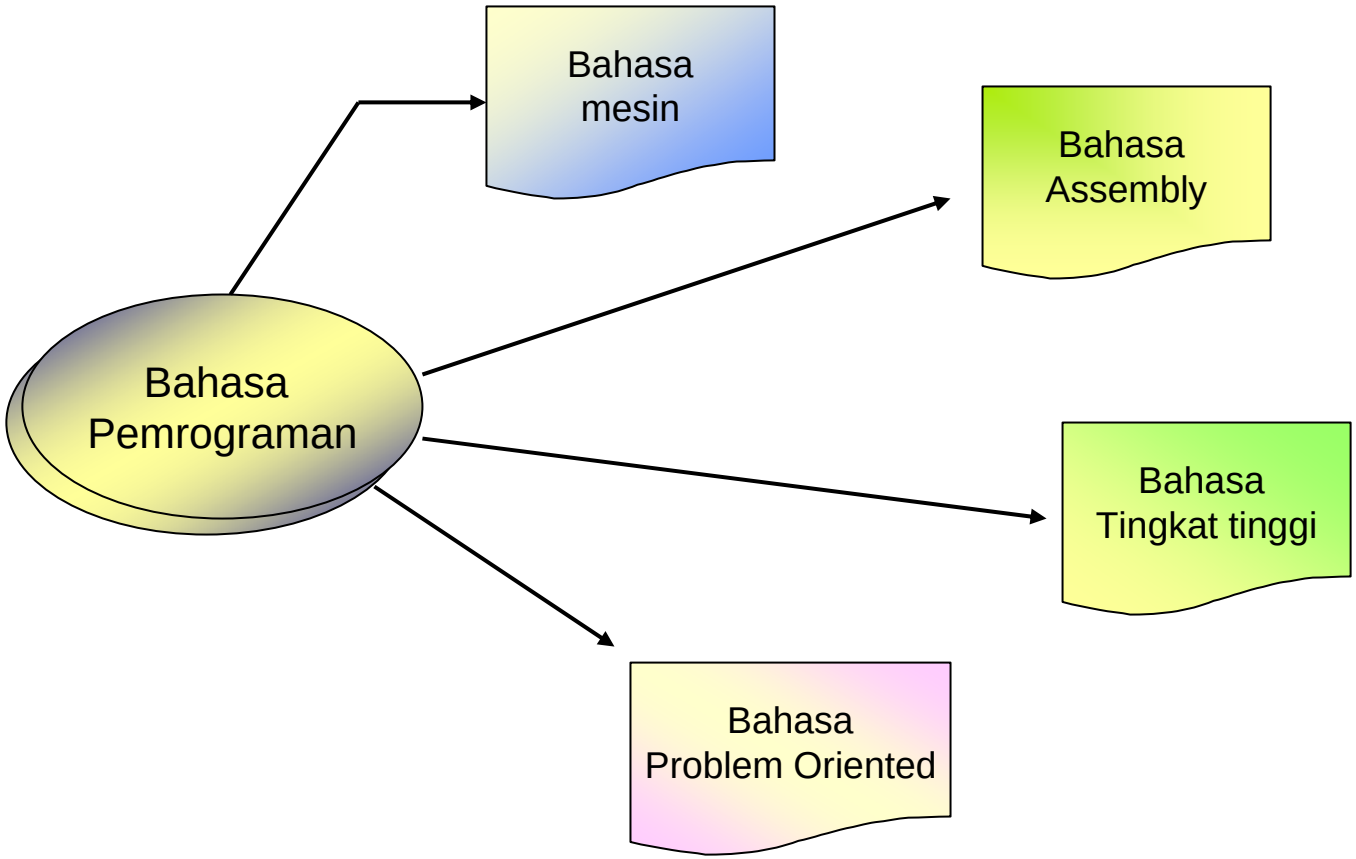
Tujuan Pembelajaran :

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



Bahasa mesin merupakan bentuk terendah dari bahasa komputer. Instruksi direpresentasikan dalam kode numerik.

Bahasa tingkat tinggi (user oriented) lebih banyak memberikan fungsi kontrol program, kalang, block, dan prosedur.

Program Language

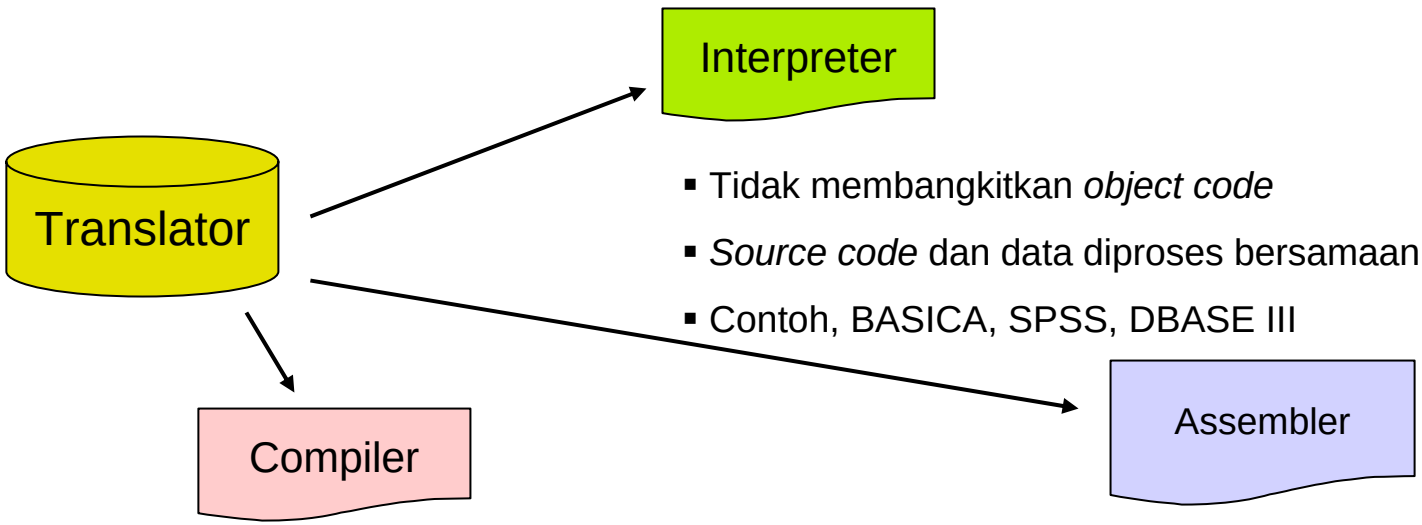
Bahasa Assembly merupakan bentuk simbolik dari bahasa mesin. Kode misalnya ADD, MUL, dsb

Bahasa problem oriented sering juga dimasukkan sebagai bahasa tingkat tinggi, misalnya SQL, Myob, dsb.

2. Translator

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

Interpreter dan Compiler termasuk dalam kategori translator.

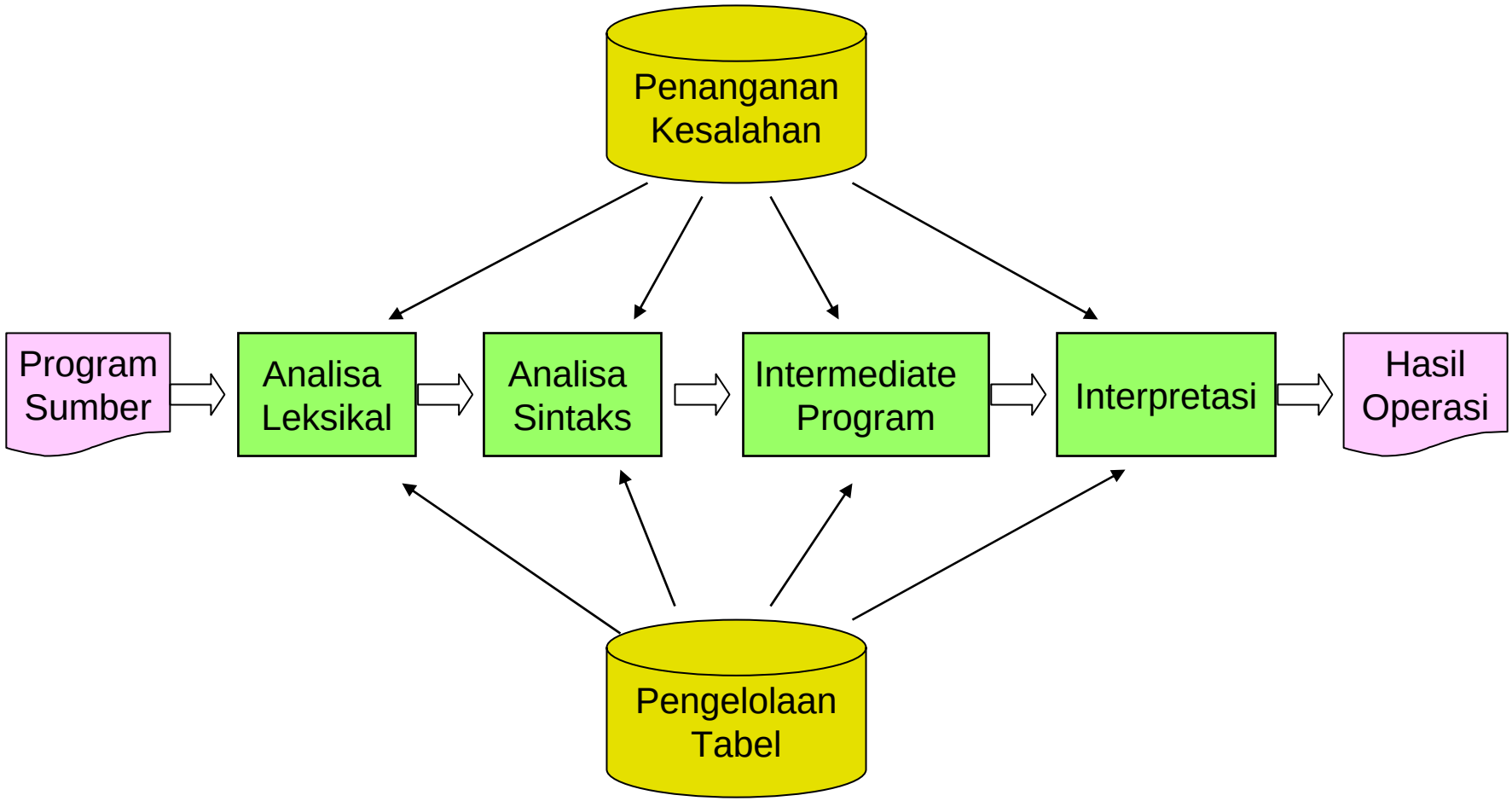


- Tidak membangkitkan *object code*
- *Source code* dan data diproses bersamaan
- Contoh, BASICA, SPSS, DBASE III

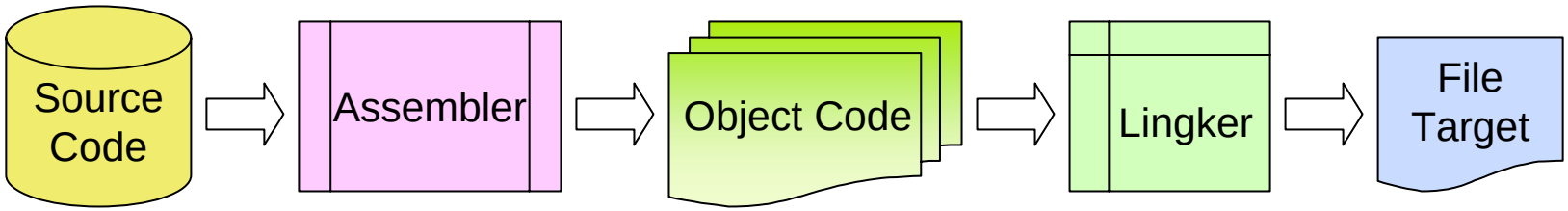
- *Source Code* adalah bahasa tingkat tinggi.
- *Object Code* adalah bahasa mesin atau assembly
- *Source code* dan data diproses tidak bersamaan

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

Interpreter



Assembler



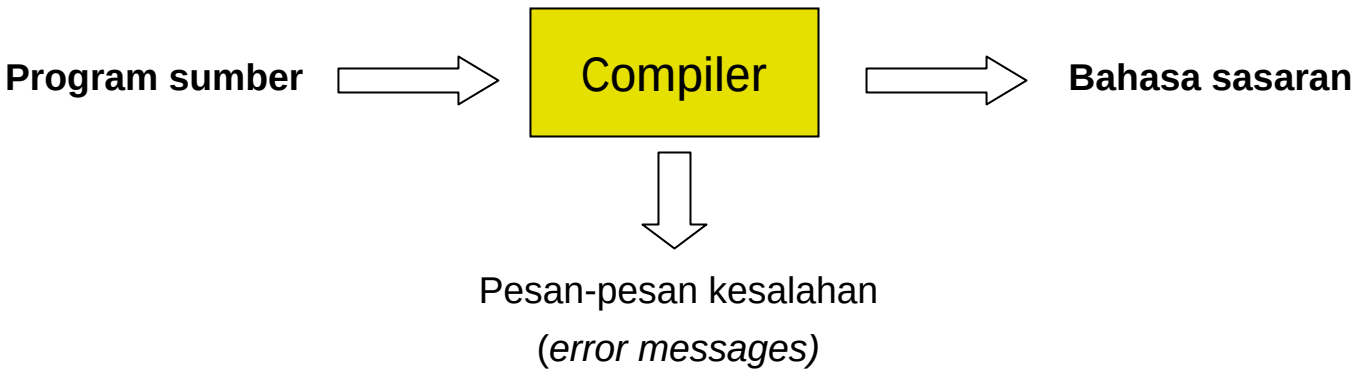
Proses Sebuah Kompilasi pada
Bahasa Assembler

- 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*)

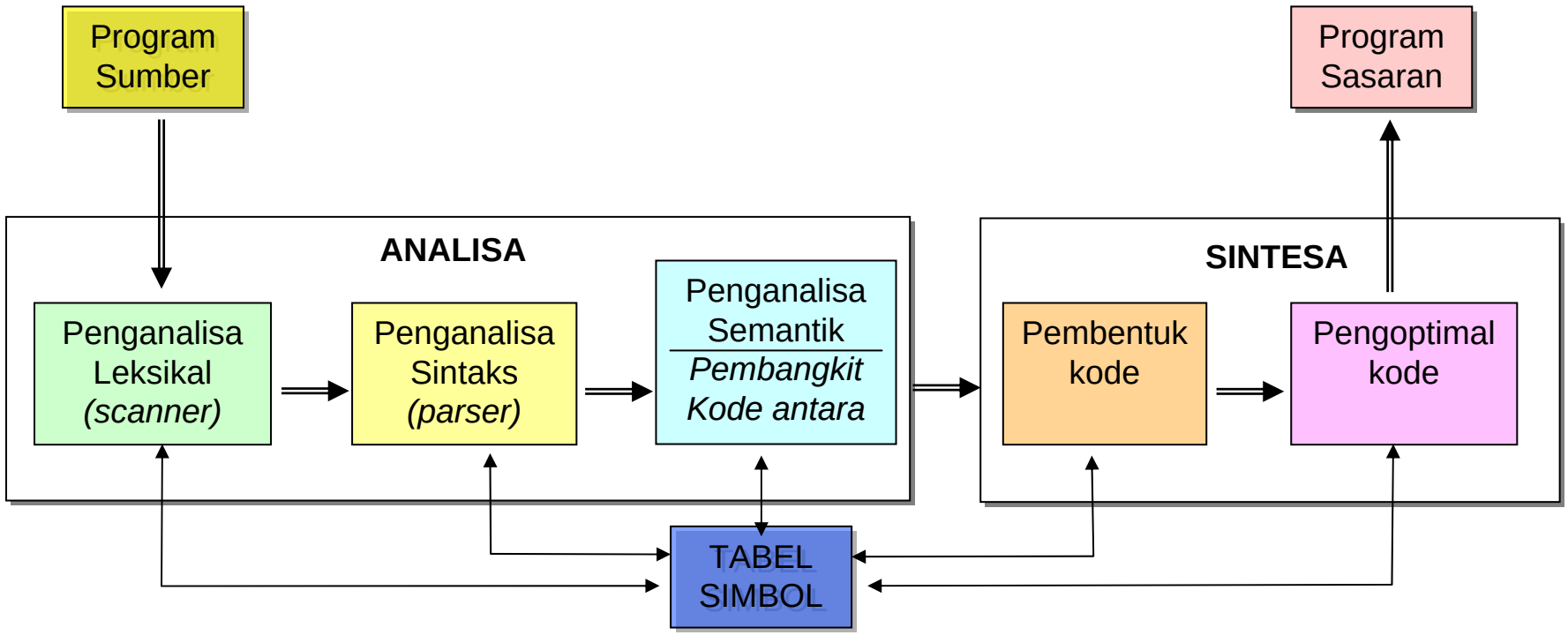
Proses kompilasi dapat digambarkan melalui sebuah blok diagram sebagai berikut :



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

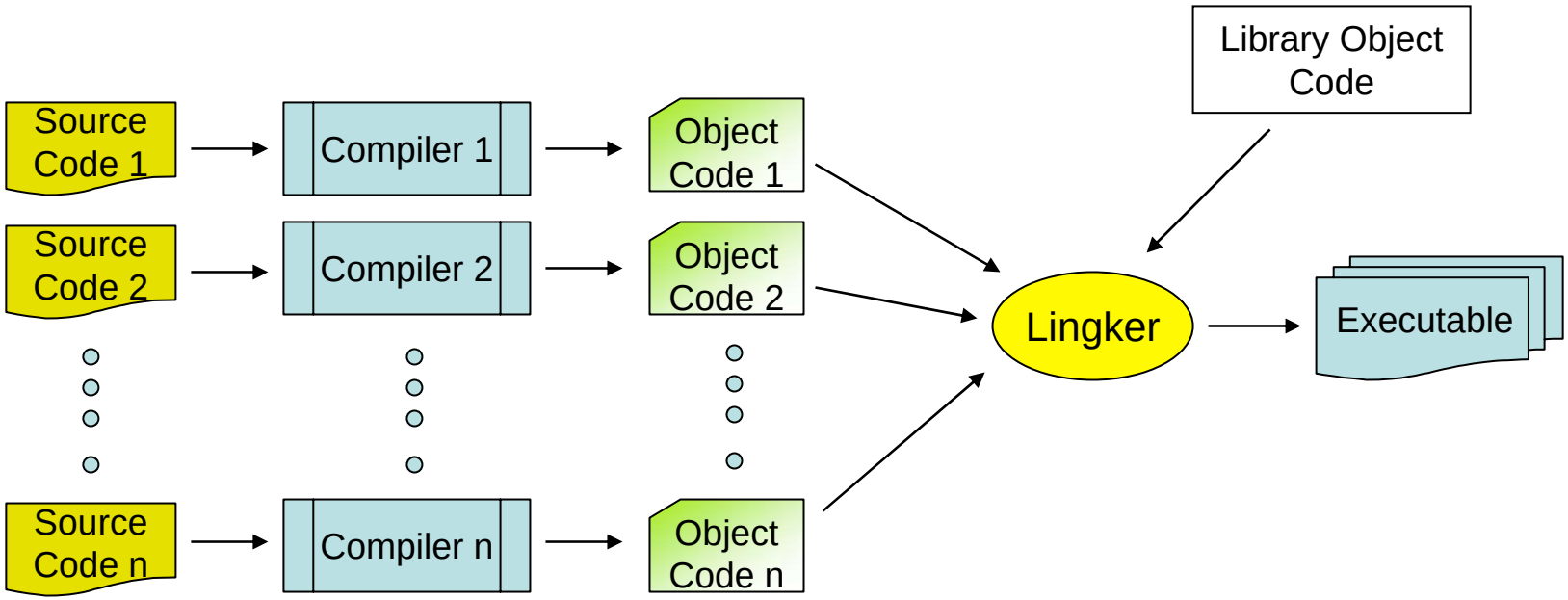
Blok Diagram



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

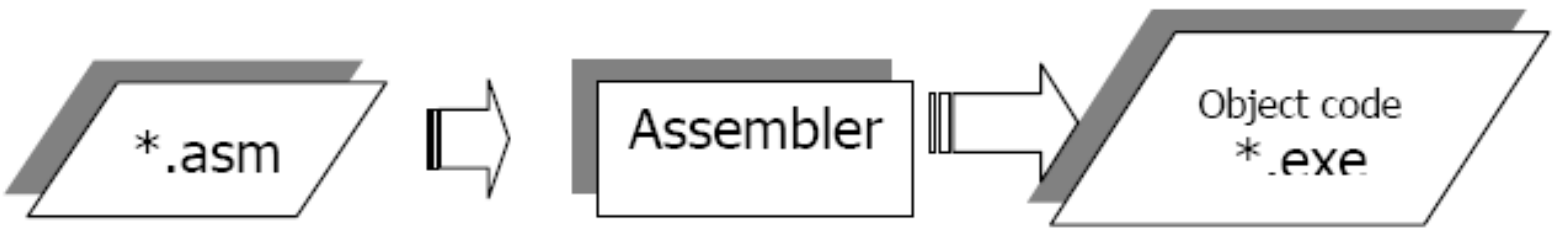
   Pembuatan kompilator secara bertingkat.

Jenis Translator: ASSEMBLER

Ada Beberapa jenis Translator untuk menterjemahkan agar dikenali oleh mesin, diantaranya

1. Assembler

Source code adalah bahasa *assembly*, *Object code* adalah bahasa mesin

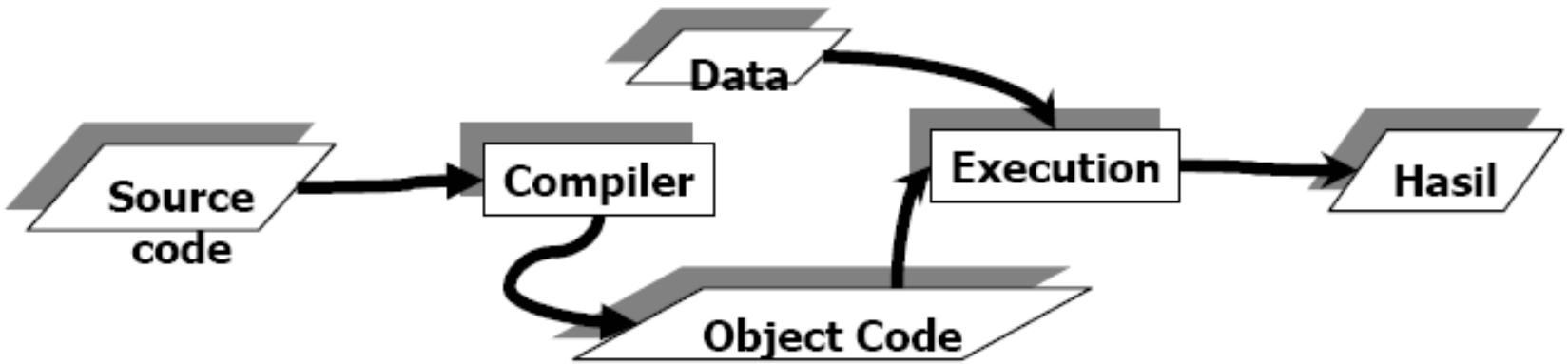


Gambar 3: Penterjemah assembler

Jenis Translator: COMPILER

2. Compiler

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

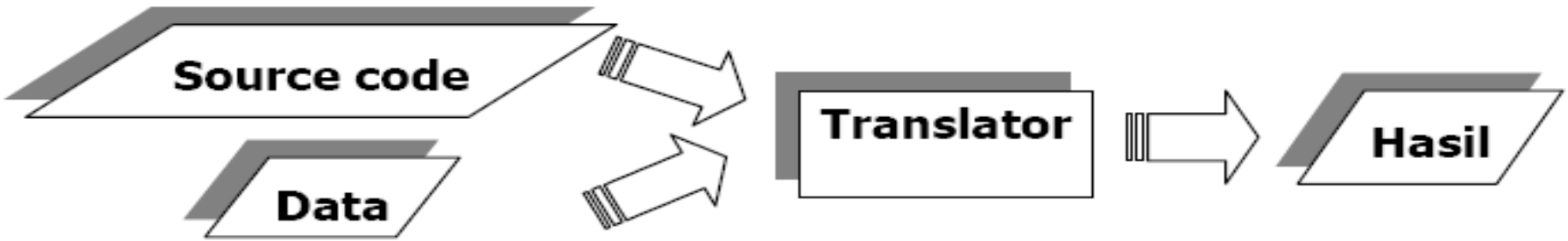


Gambar 3: Penterjemah Compiler

Jenis Translator: Interpreter

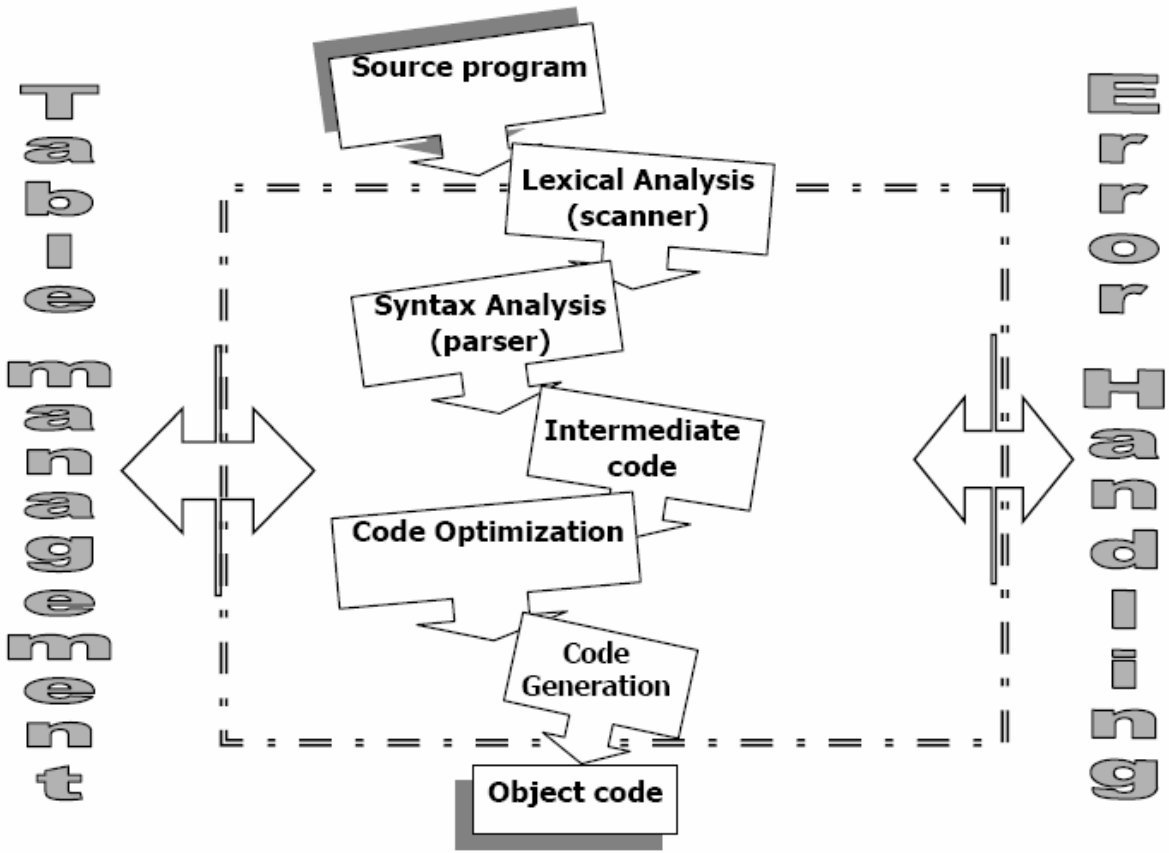
3. Interpreter

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



Gambar 4: Penterjemah interpreter

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

berhubungan dengan bahasa

Sering disebut dengan scanner, bertugas sebelum proses Syntax Analyzer, dan Intermediate Co de dilakukan, dimana tugas lexical analisis ini mendekomposisi program sumber menjadi bagian-bagian kecil

Lexical Analysis

Tugas tugasnya secara detail adalah

- Mengidentifikasikan semua besaran yang membangun 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

Contoh : 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;** Bergun a 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 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

- 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;

- Contoh instruksi :

  Kondisi : IF A > B THEN C = D;

- Maka scanner akan mentransformasikan kedalam token-token, sbb:
  - **Kondisi** 3
  - **:** 26
  - **IF** 20
  - **A** 1
  - > 15
  - **B** 1
  - **THEN** 21
  - **C** 1
  - **D** 1
  - **;** 27

Lexical Analysis

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)

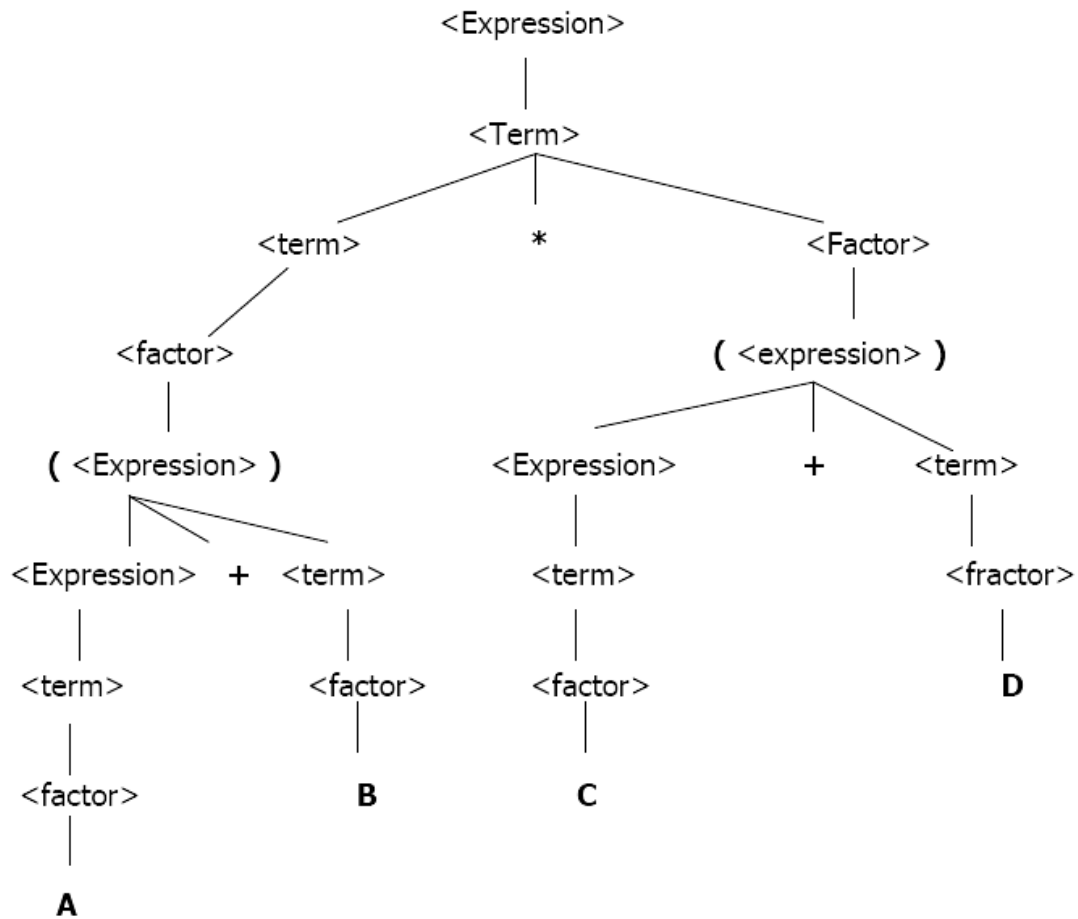
Syntax Analysis

Bagian kedua dari compiler bertugas memeriksa kebenaran dan urutan dari token-token yang terbentuk oleh lexical analysis. Tugas dari syntax analyser adalah:

Tugas dari Syntax Analizer adalah:

- 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
- Terdapat statement : **(A + B) * (C + D)**
- Akan menghasilkan bentuk sintaksis: **<factor>, <term> & <expression>**

Statement: (A+B)*(C+D)



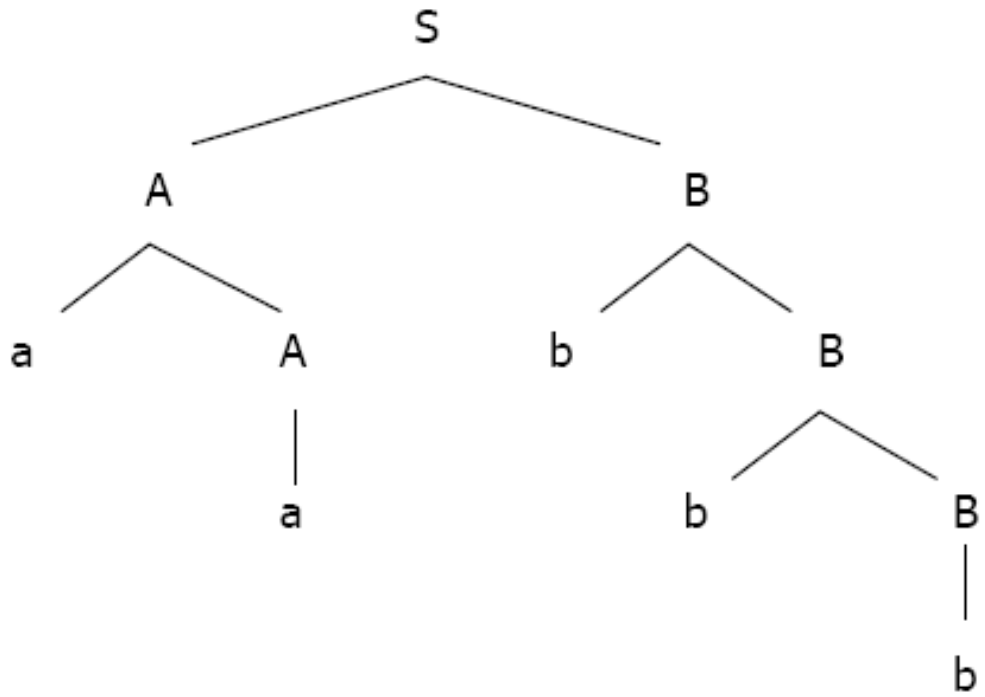
Syntax Tree

- 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 menghasilkan string **aabbb**



- Penurunan untuk menghasilkan string aabbb

Parsing atau Proses Penurunan

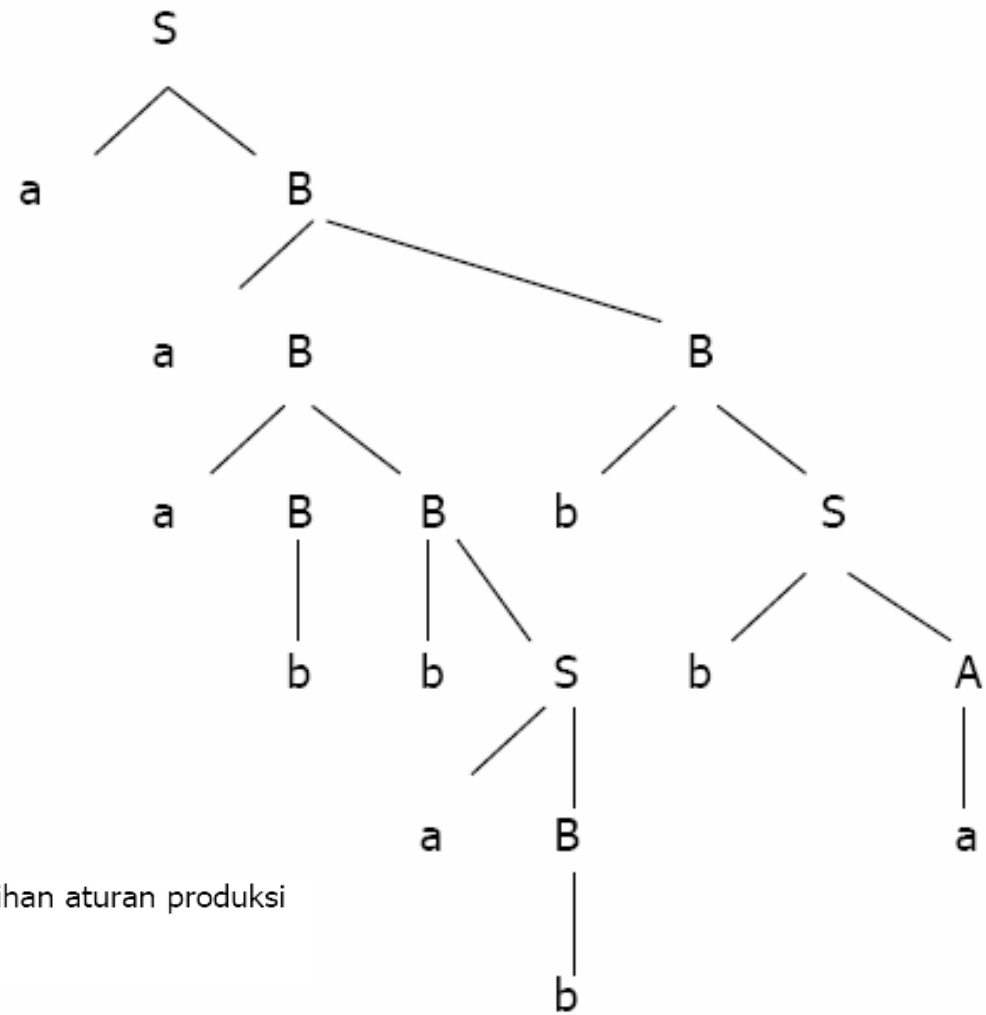
Parsing dapat dilakukan dengan cara :

- Penurunan terkiri (*leftmost derivation*) : simbol variable yang paling kiri diturunkan (tuntas) dahulu
- Penurunan terkanan (*rightmost derivation*): variable yang paling kanan diturunkan (tuntas) dahulu

Parsing atau Proses Penurunan

- Misalkan:ingin dihasilkan string *aabbaa* dari
- context free language: $S \rightarrow a\ AS \mid a$,
  $A \rightarrow SbA \mid ba$
- Penurunan kiri : S => a**A**S
  => a**SbA**S
  => aabAS
  => aaabbaS
  => aabbaa
- Penurunan kanan : S => aAS
  => aAa
  => aSbAa
  => aSbbaa
  => aabbaa

- 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

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

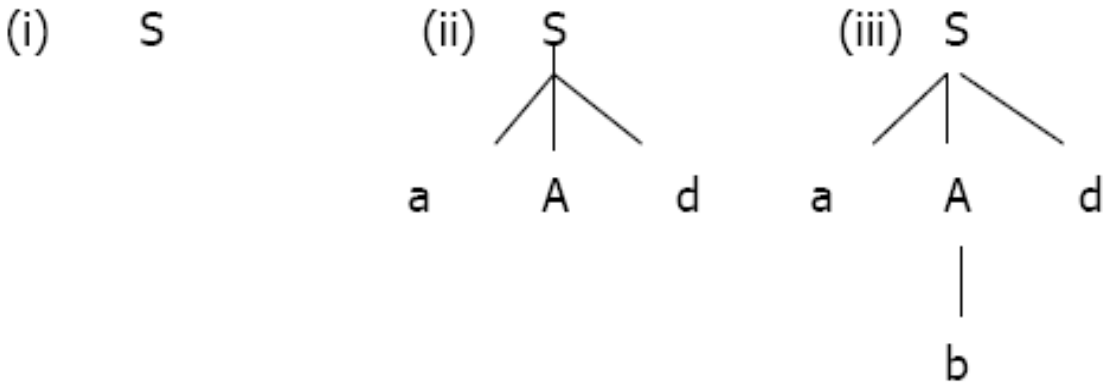
$S \rightarrow aAd \mid aB$

$A \rightarrow b \mid c$

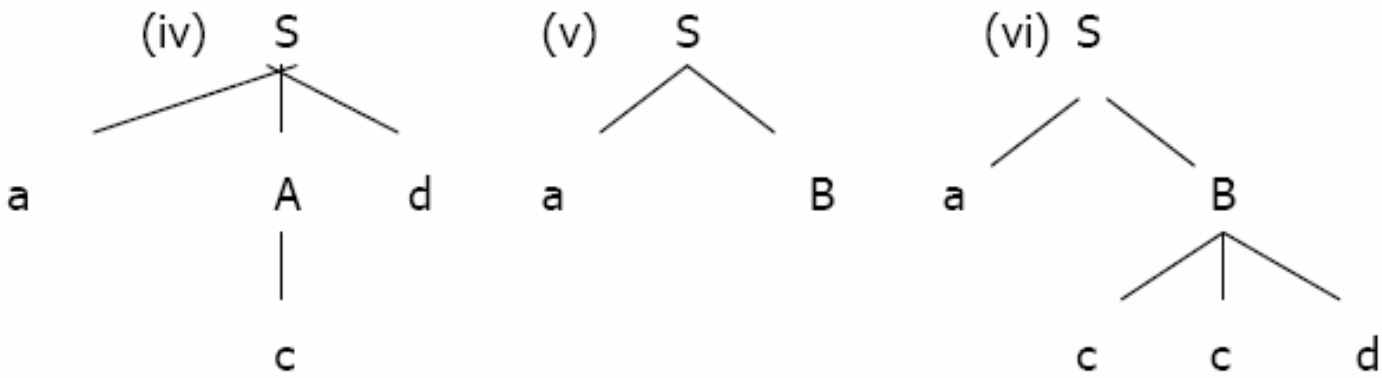
$B \rightarrow ccd \mid ddc$

Parsing: BRUTE FORCE

- Misal ingin dilakukan parsing untuk string 'addc'



Terjadi kegagalan (iii), dilakukan back track



Terjadi kegagalan lagi (iv), dilakukan back-track

Kelemahan metode BRUTE FORCE

- Mencoba untuk semua aturan produksi yang ada sehingga menjadi lambat (waktu eksekusi)
- Mengalami kesukaran untuk melakukan pembetulan kesalahan
- Banyak memakan memori, karena 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

Contoh pada brute-force

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)

Contoh pada brute-force

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***

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

Contoh pada brute-force

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***

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

Aturan produksi rekursif

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:

$S \rightarrow S\ d$

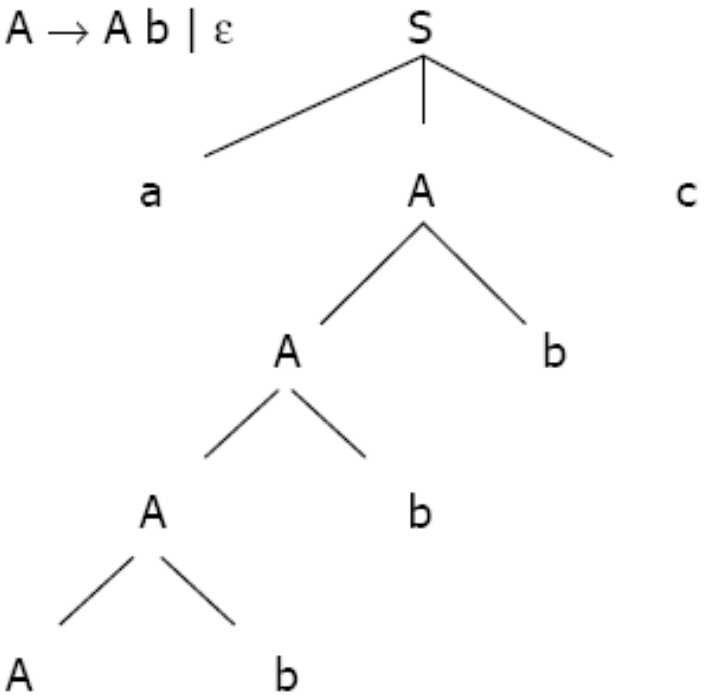
$B \rightarrow B\ ad$

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:

$S \rightarrow a\ A\ c$

$A \rightarrow A\ b \mid \varepsilon$



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**

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

- lakukan per-gantian 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. $Z \rightarrow \alpha 1\ Z \mid \alpha 2\ Z \mid \ldots \mid \alpha n\ Z$

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

- 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***

- 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 mengaplikasikan 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*

- Lanjut ke TEKNIK KOMPILASI III. ppt