3 PENGENALAN KOMPILASI

3–1 BAHASA PEMROGRAMAN

Kita semua tahu, bahwa untuk menyampaikan sesuatu hal atau pendapat kita kepada teman, atau katakanlah bahwa kita berkomunikasi antar sesama manusia, dibutuhkan suatu bahasa.

Bahasa tersebut sebaiknya dimengerti benar oleh pihak-pihak yang terlibat dalam interaksi tersebut. Tentunya komunikasi akan lebih lancar apabila kita menggunakan bahasa Indonesia sewaktu berkomunikasi dengan teman kita sesama bangsa Indonesia, dibandingkan menggunakan bahasa Perancis misalnya. Atau, kita menggunakan bahasa Internasional, bahasa Inggeris dalam Pertemuan Resmi tingkat Internasional, bukan bahasa lain.

Interaksi yang melibatkan manusia, akan memberikan hasil yang paling efektif, jika dilaksanakan melalui alat bahasa yang sesuai. Bahasa merupakan alat untuk menyatakan ekspresi serta ungkapan dari pikiran dan ide. Tanpa bahasa, komunikasi akan menjadi sangat sulit.

Untuk memanfaatkan komputer, kita perlu berkomunikasi dengan komputer ter-sebut. Segala instruksi kepada komputer harus kita susun lebih dahulu dan harus kita tulis dalam sebuah bahasa yang hendaknya dapat "dimengerti" oleh komputer. Bahasa seperti yang dimaksud ini kita kenal sebagai bahasa pemrograman.

Dalam pemrograman komputer, suatu bahasa pemrograman berfungsi melayani, sebagai suatu alat komunikasi antara kita dengan suatu masalah yang akan kita selesaikan.

Dengan bahasa pemrograman itu kita menulis program penyelesaian masalah tersebut, dan di sini komputer menolong kita menyelesaikan masalah itu.

Bahasa pemrograman harus mampu menjembatani alam pikiran manusia yang sering tidak terstruktur dengan ketepatan yang dituntut dalam pemanfaatan serta pelaksanaan kerja komputer.

Suatu program penyelesaian masalah, akan lebih mudah dan alami untuk kita peroleh jika bahasa pemrograman yang digunakan cukup sesuai dengan tipe masalah. Bahasa pemrograman harus mengandung susunan serta struktur yang mencerminkan terminologi elemen yang digunakan dalam penyajian masalah, dan juga hendaknya bebas dari komputer yang digunakan. Biasanya bahasa pemrograman tingkat tinggi memenuhi hal di atas. Komputer digital justru sebaliknya. Ia hanya menerima dan mengerti bahasa mereka sendiri, khususnya bahasa tingkat rendah. Bahasa tingkat rendah ini terdiri dari barisan panjang 0

dan 1. Barisan ini pada umumnya sangat sukar untuk dimengerti oleh manusia. Bahasa tingkat rendah tipe ini sangat berbeda dengan bahasa tingkat tinggi yang digunakan untuk memaparkan masalah ke dalam suatu model yang sesuai.

Hirarki atau tingkatan dari bahasa pemrograman berdasarkan pada peningkatan kebebasan mesin, adalah sebagai berikut :

1. Bahasa mesin (machine-level languages)
2. Bahasa rakitan (Assembly languages)
3. Bahasa tingkat tinggi atau bahasa user-oriented
   (Higer-level atau user-oriented languages)
4. Bahasa Berorientasi Masalah (Problem-oriented languages)

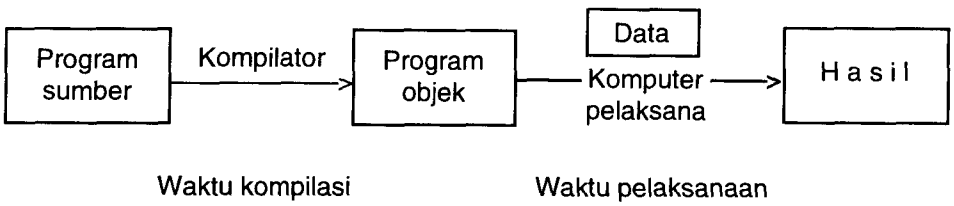
Program bahasa tingkat tinggi, tentu saja, harus secara otomatis diterjemahkan menjadi program bahasa-mesin yang ekivalen. Pembicaraan tentang beberapa penterjemah (Translator) diberikan pada bagian berikut.

3–2 TRANSLATOR

Input dari sebuah Penterjemah atau Translator adalah sebuah program yang disebut *program sumber* atau source program. Program sumber ini selanjutnya diterjemahkan ke dalam sebuah *program obyek* atau *program target*. Program sumber ditulis dalam suatu bahasa sumber (source language), dan program obyek dalam suatu bahasa obyek (object language).

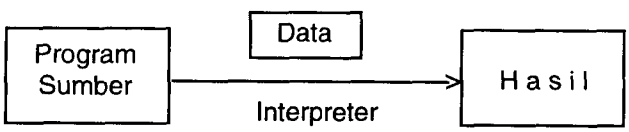
Jika bahasa sumber yang diterjemahkan tersebut adalah bahasa rakitan, dan program obyek yang bersangkutan merupakan program dalam bahasa mesin, penterjemah disebut *assembler*. Bahasa rakitan menyerupai bahasa mesin. Dalam suatu bahasa mesin elementer, banyak di antara instruksinya merupakan penyajian simbolik dari instruksi bahasa mesin yang bersesuaian. Setiap instruksi rakitan terdiri dari sebuah himpunan terurut (ordered set) dari field. Sebagai contoh, field pertama harus mewakili suatu label yang diikuti oleh suatu field operasi. Bahasa rakitan yang lebih canggih akan mendukung istruksi makro. Suatu instruksi makro dapat diperluas menjadi sederetan instruksi sederhana bahasa mesin. Bahasa tingkat tinggi mampu membentuk statemen case, statemen bersarang (nested), dan blok, yang biasanya tidak terdapat dalam bahasa rakitan.

Suatu Translator yang mentransformasikan suatu bahasa tingkat tinggi, seperti FORTRAN, PASCAL, atau COBOL, ke dalam bahasa mesin komputer atau ke dalam bahasa rakitan, disebut *Kompilator* atau *Compiler*. Waktu yang digunakan untuk mengkonversikan suatu program sumber ke program obyek, disebut *waktu kompilasi* atau compile time. Program obyek dijalankan dalam waktu yang disebut *waktu pelaksanaan* atau run time. Gambar 3-1 merupakan ilustrasi dari proses kompilasi. Sebagai catatan, program sumber dan data diproses pada waktu yang berbeda, masing-masing sebagai waktu kompilasi dan waktu pelaksanaan.



Gambar 3-1 Proses Kompilasi

Salah satu bentuk lain dari Translator, adalah *interpreter*. Pada Interpreter, proses terhadap suatu bentuk internal dari program sumber dan proses terhadap data berlangsung pada waktu yang sama. Di sini interpretasi terhadap bentuk internal dari program sumber dilaksanakan pada saat run time. Di sini tidak ada program obyek yang terbentuk. Gambar 3-2 merupakan ilustrasi proses interpretasi. Beberapa interpreter bekerja menganalisis setiap statemen program sumber pada setiap kali dijalankan. Suatu cara atau pendekatan lain yang lebih efisien adalah dengan menerapkan teknik kompilasi untuk menterjemahkan program sumber ke suatu bentuk program sumber intermediate. Bentuk program sumber intermediate inilah yang kemudian diterjemahkan oleh program interpreter.



Gamber 3-2 Proses Interpreter

Interpreter menjadi populer belakangan ini, khususnya dalam lingkungan mikro-komputer. Di sini biaya tambahan dari proses interpretesi boleh dibilang tidak ada. Apalagi bila program yang dibuat tidak terlalu panjang, dan digunakan tidak terlalu sering. Sebagai contoh, adalah merupakan alasan utama mengapa bahasa seperti BASIC, APL, LISP dan SMALLTALK-80 menjadi sangat populer, yakni karena mereka dapat diimplementasikan berturut-turut dalam suatu lingkungan.

Dalam buku ini, perhatian kita titik beratkan pada teknik Kompilasi. Walaupun suatu Kompilator lengkap tidak akan ditemukan dalam naskah ini, pembaca dapat menengok ke buku *An Implementation Guide to Compiler Writing*, yang membahas suatu Kompilator untuk bahasa pemrograman tingkat tinggi GAUSS secara lebih dalam. Suatu Kompilator pada hakikatnya dapat dibahas sebagai beberapa program modular.

3–3 MODEL KOMPILATOR

Pekerjaan untuk membuat sebuah Kompilator untuk suatu bahasa sumber adalah teramat rumit. Kerumitan serta sifat dari proses kompilasi tergantung pada tingkat keluasan dari bahasa sumber. Kerumitan Kompilator dapat dikurangi jika perancang bahasa pemrograman mempertimbangkan bermacam-macam faktor perancangan. Namun, bagaimanapun halnya, suatu model dasar dari Kompilator dapat kita rumuskan. Suatu model diberikan seperti pada Gambar 3-3. Walaupun model ini boleh bervariasi untuk berbagai bahasa tingkat tinggi yang berbeda, model ini telah dapat mewakili proses kompilasi secara umum.

Suatu Kompilator harus melaksanakan 2 tugas utama, yakni tugas analisis terhadap program sumber, dan tugas sintesis memadukan menjadi program obyek yang ekivalen. Tugas analisis berhubungan dengan pemisahan bagian dari program sumber menjadi bagian-bagian dasar, dan menggunakan bagian-bagian ini dalam tugas sintesis membangun modul program obyek yang ekivalen. Pelaksanaan tugas ini lebih mudah jika dibantu dengan pembuatan beberapa tabel.

Suatu program sumber adalah suatu untai atau string dari simbol yang umumnya berupa huruf, angka atau simbol khusus seperti +.–, (,), dan sebagainya. Suatu program sumber berisi bentuk dasar bahasa seperti

* nama variabel
* label
* konstanta
* katakunci ataupun operator.

3–4 ANALISIS LEKSIKAL

Program sumber (lihat Gambar 3-3) merupakan input dari Penganalisis Leksikal atau Scanner. Analisis Leksikal mempunyai tujuan untuk memisahkan naskah program sumber yang masuk, menjadi bagian leksikografis terkecil atau *Token* seperti misalnya konstanta, nama variabel, kata kunci (seperti DO, IF dan THEN dalam PL/I), dan operator. Pada hakikatnya, inti tugas Penganalisis Leksikal adalah melaksanakan analisis Sintaks tingkat rendah.

Untuk alasan efisiensi, kepada setiap kelas dari Token diberikan suatu bilangan yang mewakili harga internalnya, secara unik. Sebagai contoh, suatu nama variabel mungkin disajikan atau diwakili oleh bilangan 1, suatu konstanta disajikan dengan bilangan 2, suatu label dengan bilangan 3, operator penjumlahan dengan bilangan 4, dan sebagainya.

Sebagai contoh pandang Statemen PL/I :

TEST : IF A > B THEN X=Y;

Statemen ini dapat diubah oleh Penganalisis Leksikal ke dalam deretan Token, dan masing-masingnya disajikan / diwakili oleh bilangan;

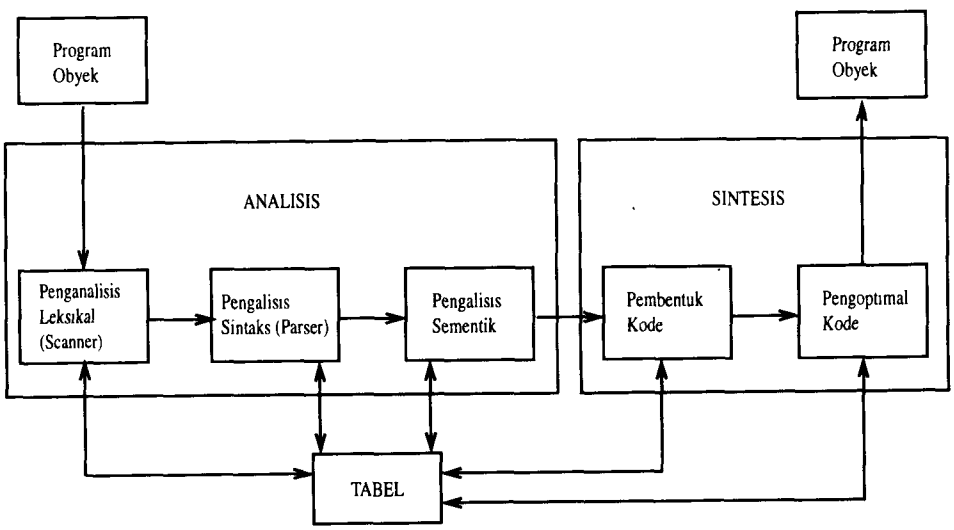
TOKEN	DIWAKILI OLEH	KETERANGAN JENIS TOKEN
TEST	3	Label
:	26	Tanda baca
F	20	Kata kunci
A	1	Nama variabel
>	13	Operator relasi
B	1	Nama wariabel
THEN	4	Kata kunci
X	1	Nama variabe
=	10	Operator assignment
Y	1	Nama variabe
;	27	Tanda baca

Sebagai catatan, dalam pelacakan statemen sumber dan pembentukan bilangan penyajian dari setiap Token, mungkin dapat dihasilkan blank atau spasi dalam statemen. Penganalisis Leksikal harus dapat mengenali adanya spasi dan komentar (REMARK). Kedua kelas item jenis ini, meskipun adalah merupakan bagian dari program, tidak akan dilaksanakan, dan akan diabaikan. Sebagai contoh adalah dalam bahasa PL/I.

Bahasa pemrograman tertentu mengizinkan adanya statemen baris jamak. Penganalisis lesikal harus dapat melakukan proses input seperti statemen baris jamak ini. Beberapa Scanner menempatkan konstanta lebel dan nama variabel pada tabel-tabel yang sesuai. Isi suatu tabel dapat berupa tipe variabel (INTEGER, atau BOOLEAN), alamat program obyek, harga dan di mana variabel tersebut dideklarasikan.

Penganalisis Leksikal mengirimkan Token kepada Penganalisis Sintaks. Token ini dapat berbentuk beberapa potong item. Item pertama memberikan alamat/lokasi dari Token dalam beberapa tabel simbol. Item kedua adalah bilangan yang mewakili Token.

Hal ini merupakan suatu pendekatan yang menguntungkan, yakni semua Token diwakili oleh panjang informasi yang tetap, yang terdiri dari suatu alamat (atau pointer) dan suatu bilangan bulat.



Gambar 3-3 Komponen sebuah Kompilator

3-5 ANALISIS SINTAKS

Penganalisis Sintaks lebih rumit dibandingkan dengan Penganalisis Leksikal. Fungsinya adalah menerima program sumber (dalam bentuk barisan Token) dari Penganalisis Leksikal, dan selanjutnya Penganalisis Sintaks ini akan menetukan struktur secara keseluruhan dari program sumber. Proses terakhir ini analog dengan menentukan struktur dari suatu kalimat dalam bahasa Inggris. Dalam contoh kita pada bab terdahulu, kita mengidentifikasikan struktur kalimat atas kelas tertentu, seperti subyek, predikat, kata kerja, kata benda, dan kata sifat.

Dalam melakukan analisis Sintaks, kita memandang kelompok Token sebagai kelas Sintaks yang lebih besar, seperti *ekspresi*, *statemen*, dan *prosedur*. Penganalisis Sintaks, yang disebut juga *Parser*, akan menghasilkan pohon Sintaks atau sejenisnya. Di sini simpul daun merupakan Token, dan setiap simpul yang bukan daun mewakili suatu tipe kelas Sintaks (yang kita kenal sebagai simbol nonterminal atau variabel). Sebagai contoh adalah analisis dari statemen sumber :

(A+B)*(C+D)

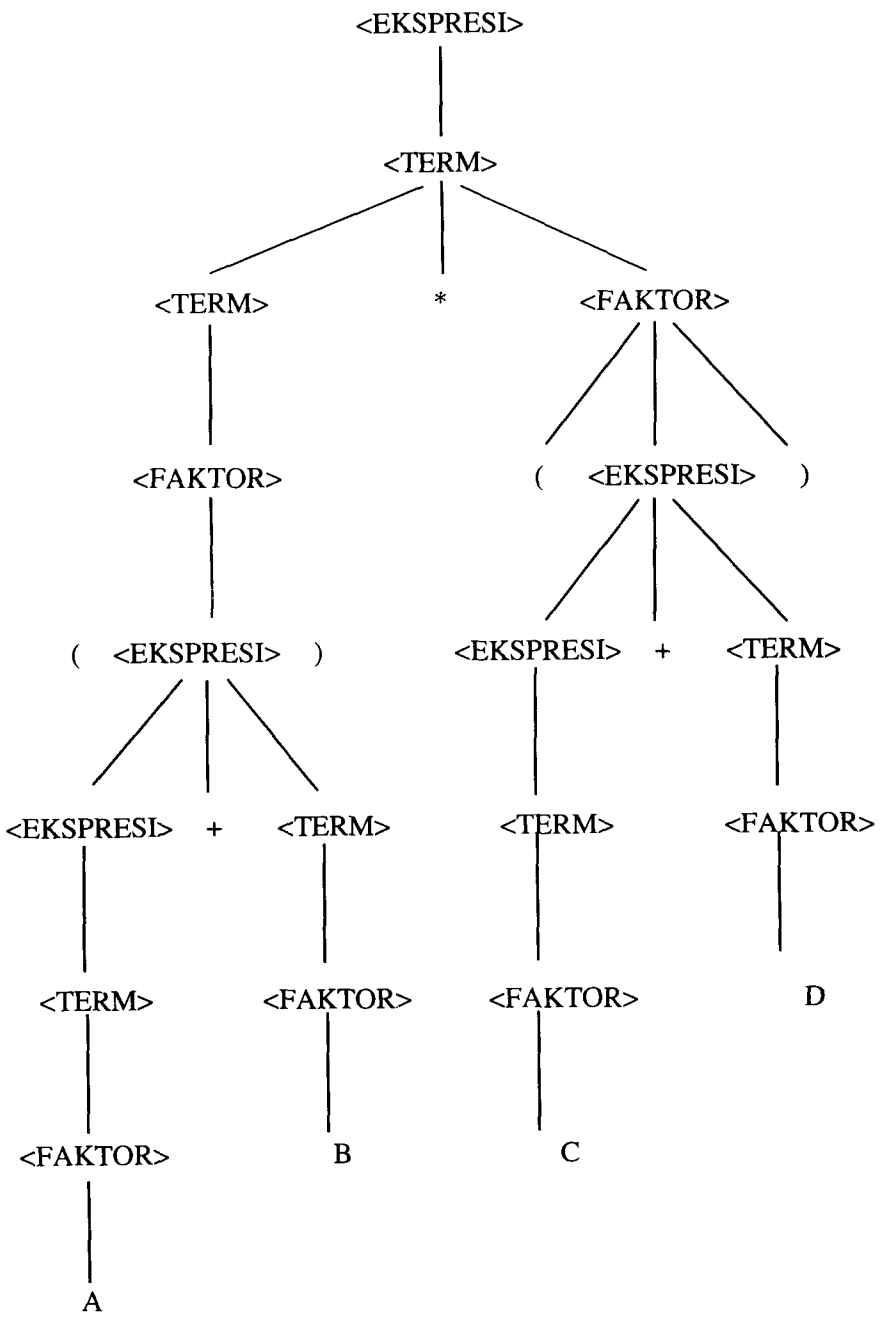
dapat menghasilkan kelas <factor>, <term> dan <expression> seperti yang diperlihatkan dalam pohon Sintaks pada Gambar 3-4.

Sebuah Grammar dapat digunakan oleh Penganalisis Sintaks untuk menentukan struktur dari program sumber. Proses pengenalan ini disebut *parsing*, dan karena itu Penganalisis Sintaks disebut juga *parser*.

Pohon Sintaks yang dihasilkan oleh parser kemudian digunakan oleh Penganalisis Semantik. Fungsi Penganalisis Semantik adalah memberi *arti*, atau *Semantik* dari program sumber. Meskipun secara konsep kita memisahkan Sintaks program sumber terhadap Semantik, Penganalisis Sintaks dan Semantik pada hakikatnya bekerja bersama secara sangat dekat. Walaupun demikian dalam sebuah Kompilator, proses analisis Semantik merupakan suatu proses yang berdiri sendiri, dan unik.

Untuk ekspresi (A+B)*(C+D), dari contoh, Penganalisis Semantik harus menentukan aksi apa yang dilakukan oleh operator aritmetika penambahan dan perkalian tersebut. Ketika parses mengenali sebuah operator '+' atau '* ', parser memanggil rutin SEMANTIC–ROUTINE, yang akan menggolongkan aksi yang akan dikerjakan.

Rutin ini mungkin memeriksa apakah kedua operand yang ditambahkan sudah dideklarasikan, serta apakah mempunyai tipe yang sama (jika tidak, bisa



Gambar 3-4 suatu pohon Sintaks untruk ekspresi (A+B)*(C+D)

saja rutin tersebut membuat menjadi sama), dan apakah kedua operand mempunyai nilai. Penganalisis Semantik sering berinteraksi dengan berbagai tabel dari Kompilator dalam menjalankan tugasnya.

Aksi Penganalisis Semantik mungkin mengandung bentuk intermediate dari kode sumber. Untuk ekspresi (A+B)*(C+D), bentuk intermediate dari kode sumber mungkin berbentuk kuadrupel, seperti:

(+,A,B,T1)

(+,C,D,T2)

(*,T1,T2,T3)

Di sini dapat diartikan :

(+,A,B,T1) Menambah A dengan B, dan menempatkan hasilnya sementara sebagai T1

(+,C,D,T2) Menambah C dengan D, dan menempatkan hasilnya sementara sebagaiT2

(*,T1,T2,T3) Mengalikan T1 dengan T2, dan menempatkan hasilnya di T3

Bentuk yang tepat dari bahasa sumber intermediate tergantung pada besar kecilnya bahasa sumber, dan tergantung bagaimana ia diproses dalam tugas sintesis.

Sebuah ekspresi infix dapat dikonversikan ke bentuk intermediate yang disebut *Notasi Polish*. Sebuah ekspresi infix (A+B)*(C+D) dapat dikonversikan sehingga ekivalen dengan suffix-polish (postfix-polish) AB+CD+*. Ekspresi yang terakhir ini mengandung operator aritmetika dalam urutan sebagaimana ia dilaksanakan. Jenis lain dari bentuk intermediate bahasa sumber dapat berupa sebuah pohon Sintaks yang menyatakan hasil uraian atau parse dari program sumber.

Output dari Penganalisis Semantik dikirim kepada Pembentuk Kode (code generator). Pada saat itu bentuk intermediate program sumber, biasanya diterjemahkan ke dalam bahasa rakitan (assembly language), ataupun ke dalam bahasa mesin (machine language). Sebagai contoh, penterjemahan dari ketiga kuadrupel di atas, dapat menghasilkan barisan alamat-tunggal (single-address), instruksi akumulator-tunggal bahasa rakitan (single-accumulator assembly language) berikut ini:

```
LDA A   Panggil (LOAD) isi A, masukkan ke dalam akumulator
ADD B   Tambahkan (ADD) isi B ke dalam akumulator
STO T1  Simpan(STORE) isiakumulator ke dalam penyimpan sementara T1
LDA C   Panggil (LOAD) isi C, masukkan ke dalam akumulator
ADD D   Tambahkan (ADD) isi B ke dalam akumulator
STO T2  Simpan (STORE) isiakumulatorke dalam penyimpan sementara T2
LDA T1  Panggil (LOAD) isi T1, masukkan ke dalam akumulator
MUL T2  Kalikan (MULTIPLY) isi T2 dengan isi akumulator
STO T3  Simpan (STORE) isi akumulator ke dalampenyimpan sementara T3
```

Output dari Pembentuk Kode dikirim ke Pngoptimal Kode (code optimizer). Tidak semua Kompilator memiliki Code Optimizer ini. Proses ini dilaksanakan di banyak Kompilator yang canggih. Hal ini dimaksudkan agar dapat dihasilkan program objek yang lebih efisien. Optimisasi tersebut dapat juga terjadi pada level lokal, termasuk pada waktu evaluasi ekspresi, evaluasi konstanta, evaluasi penggunaan sifat operator tertentu, seperti sifat asosiatif, komutatif, ataupun distributif, dan pemeriksaan atau deteksi atas subekspresi yang sama. Karena adanya sifat komutatif dari operator perkalian, kode bahasa rakitan yang lalu dapat direduksi atau disederhanakan menjadi sebagai berikut :

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

Dapat dicatat bahwa evaluasi kode pada contoh di atas merupakan evaluasi terhadap ekspresi (C+D)*(A+B).

Optimisasi yang lebih global tentunya dapat pula dilaksanakan. Di sini termasuk misalnya evaluasi tunggal terhadap kemunculan berkali-kali dari subekspresi yang identik, dan penghapusan (removing) statement yang tertahan di dalam sebuah loop, dan menempatkan statement tersebut di luar loop. Bentuk optimisasi tersebut adalah bentuk yang bebas mesin (machine-independent). Sedangkan pengalokasian register secara optimal adalah salah satu contoh dari

jenis optimisasi bergantung mesin. Suatu pengoptimal kode yang baik dapat menghasilkan kode yang baik, bahkan dapat lebih baik daripada hasil pengerjaan seorang pemrogram bahasa rakitan tang berpengalaman.

Model Kompilator yang diberikan pada Gambar 3-3, merupakan model yang umum. Di sini kita adakan semacam pembagian urutan kerja, dalam fase (ada 5 fase), seperti yang secara singkat telah kita bicarakan di atas. Dalam beberapa Kompilator tertentu, fase tersebut ada yang dikombinasikan. Dalam pembahasan kita kali ini, pembahasan secara rinci tentang interaksi antar fase, tidak kita lakukan.

Kita akan memelihat beberapa kemungkinan interaksi antara Penganalisis Leksikal (Scanner) dengan Penganalisis Sintaks (parser). Kemungkinan pertama adalah bahwa Scanner begitu menghasilkan sebuah Token, langsung mengirimkannya kepada parser untuk diproses. Kemudian, parser memanggil ("calls") Scanner untuk meminta Token berikutnya. Kemungkinan yang lain adalah bahwa Scanner menghasilkan semua Token dari program sumber lebih dahulu, sebelum ia mengirimkannya kepada parser. Dalam hal ini, Scanner telah memeriksa seluruh program sumber yang masuk. Keadaan ini disebut *jalan terpisah* (separate pass).

Beberapa Kompilator berusaha agar terdapat sesedikit mungkin jalan atau pass yang dibuatnya, yakni satu jalan atau single pass. Namun ada Kompilator yang membuat lebih dari 30 pass, misalnya Kompilator bahasa PL/I, yang digunakan pada mesin keluaran pertama IBM. Faktor yang mempengaruhi banyaknya pass, pada suatu Kompilator khusus, adalah sebagai berikut :

1. memori yang tersedia
2. kecepatan dan ukuran Kompilator
3. kecepatan dan ukuran program obyek
4. kelengkapan debugging yang dibutuhkan
5. deteksi kesalahan (error-detection) dan teknik recovery yang diinginkan
6. jumlah orang dan waktu yang tersedia untuk menyelesaikan proyek penyusunan Kompilator tersebut

Sebagian Kompilator, terutama yang berorientasi untuk kegunaan pendidikan atau untuk belajar mahasiswa, seperti WATFIV dan PL/I adalah contoh utama dari Kompilator satu pass. Dalam Kompilator ini, sangat sedikit (jikapun ada) kode optimasi yang dibentuk. Alasannya, karena program sumber biasanya

dikompilasi beberapa kali dahulu, baru kemudian dieksekusi. Program tersebut seringkali hanya dieksekusi satu kali, dan langsung dibuang, tak dipakai lagi.

Juga, Penganalisis Semantik dan fase pembentukan kode (code-generation) dikombinasikan dalam satu fase. Kelebihan utama di sini adalah pada waktu debugging, deteksi error, error-recovery, dan kemampuan mendiagnostik (diagnostics capabilities). Namun bagaimanapun juga, bahasa sumber yang ada, tidak dapat dikompilasi dalam satu pass. Kompilator untuk PL/I dan FORTRAN biasanya mengandung beberapa pass. Dalam hal tertentu, fase optimasi mungkin membutuhkan beberapa fasa dari program sumber (atau bentuk intermediate) yang dibuat. Lagi pula, fase optimasi sering menyebar ke seluruh pass yang lain dari proses kompilasi Kombinasi lain mungkin masih terjadi. Kadang-kadang Penganalisis Sintaks, Penganalisis Semantik, dan fase pembentukan kode dapat dikombinasikan ke dalam satu fase.

Sebuah aspek dari pembuatan Kompilator, yang tidak terlihat pada Gambar 3-3 adalah aspek hubungan dengan pendeteksian error dan pemulihan atau recovery error. Recovery error sangat dekat hubungannya dengan fase analisis Sintaks. Recovery error akan memperpanjang waktu kompilasi dari program selama mungkin, sebelum Kom-pilator menyelesaikan tugasnya menterjemahkan program sumber. Suatu strategi dapat dijalankan, yakni dengan mengeksekusi semua kemungkinan dari program obyek. Cara ini dapat mengurangi waktu yang diperlukan untuk kompilasi program sumber, sebelum ditemukannnya keadaan bebas error. Beberapa recovery error dapat melakukan semacam penyisipan dan atau penghilangan serta pengabaian Token dalam rangka membetulkan hal-hal dalam program sumber yang secara Sintaks salah.

Aspek lain dari kompilasi yang diabaikan dalam pembicaraan kali imi adalah hal yang berkaitan dengan berbagai tabel simbol, yang harus dibuat dan dikelola selama eksekusi program obyek yang diminta. Juga, penyusunan bahasa tertentu dari bahasa sumber menyebabkan beberapa struktur data harus disimpan dan dikelola dalam memori komputer. Desain dan pelaksanaan struktur ini disebut pengorganisasian waktu kerja (run-time organization) atau pengelolaan memori (storage management). Untuk struktur blok dari bahasa, bahasa yang berorientasi untuk pengelolaan string seperti SNOBOL dan bahasa yang memungkinkan pendeklarasian dan manipulasi dari struktur data yang didefinisikan pemrogram (programmer-defined data structures), maka waktu kerja yang berkenaan dengan itu sangat signifikan dan kompleks.

Akhirnya, penjelasan secara relatif atas kesukaran dari implementasi setiap fase terlihat secara urut dan jelas pada Gambar 3-3. Fase Analisis Leksikal barangkali langsung dapat diketahui kesulitannya. Kesulitan yang berikut muncul

ketika fase Analisis Sintaks atau fase parsing. Dari banyak penelitian (pengamatan), berdasarkan atas teori bahasa formal, hal ini dapat diselesaikan dalam dua fase. Dan hasilnya nanti, fase scanning dan fase parsing, akan sebagian besar telah terotomasi.

Teori tentang scanning dibicarakan pada Bab 4. Pada Bab 6 akan diterangkan tentang teori parsing. Kesulitan yang nyata pada pembuatan Kompilator adalah pada fase Analisis Semantik, pembentukan kode, dan optimisasi kode. Problema yang lain adalah mengenai kemapanan/kemantapan (portability) dari program.