KOMPILASI

Translator (penerjemah) adalah sebuah program yang menerjemahkan sebuah program sumber (*source program*) menjadi program sasaran (*target program*)
Proses translasi suatu program dari bentuk *syntax* aslinya ke dalam bentuk *executable* merupakan pusat dari implementasi semua bahasa pemrograman yang ada. Proses translasi ini dapat sederhana, tetapi dapat pula sangat kompleks.

Kesederhanaan suatu proses translasi dipicu oleh kesederhanaan suatu *translator*, biasanya *interpreter* dan tidak mementingkan kecepatan eksekusi. Dalam banyak hal, kecepatan dan efisiensi eksekusi sangat dibutuhkan dan menjadi tujuan utama pembuatan bahasa pemrograman sehingga dapat mentranslasi suatu program ke dalam suatu struktur *executable* yang efisien, khususnya kode mesin yang *hardware-interpretable*.

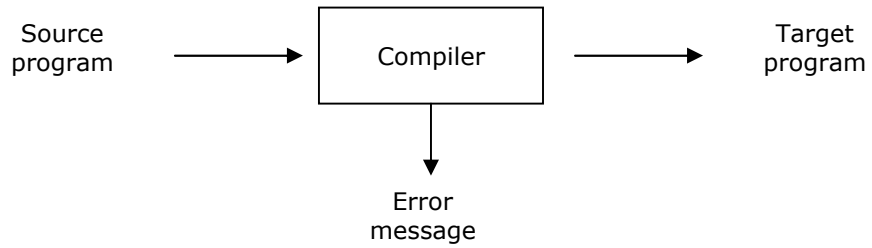
Jenis-jenis translator berdasarkan bahasa pemrograman yang bersesuaian dengan input dan outputnya adalah:

Jenis Translator	Bahasa Pemrograman	
	Input	Output
Assembler	Bahasa Rakitan	Bahasa Mesin
Compiler (Kompilator)	Bahasa Tingkat Tinggi	Bahasa tingkat rendah

KOMPILATOR (COMPILER)

Pengertian kompilator selalu mengandung dua komponen bahasa yaitu bahasa yang dibaca oleh kompilator, disebut bahasa sumber (source language), dan bahasa yang diterjemahkan oleh kompilator disebut sebagai bahasa sasaran (target language).

Jadi kompilator membaca suatu program yang ditulis ke dalam bahasa sumber dan menerjemahkan bahasa sumber tadi ke dalam suatu bahasa lain yang disebut dengan naman bahasa sasaran. Dalam melakukan proses penerjemahan tersebut, sudah barang tentu kompilator akan melaporkan adanya keanehan-keanehan atau kesalahan yang mungkin ditemukan. Proses penerjemahan yang dilakukan oleh kompilator ini disebut dengan proses kompilasi (compiling) yang dapat digambarkan seperti dibawah ini:



Bila dipandang sepintas lalu maka akan dapat timbul beraneka ragam kompilator yang dapat dibuat. Seperti telah diketahui umum, ada banyak sekali bahasa yang dapat menjadi sumber bahasa seperti bahasa FORTRAN, PASCAL, C dan juga bahasa-bahasa lain yang sifat dan pemakaiannya agak spesifik atau khusus, seperti bahasa untuk program DBASE, SPSS dan lain sebagainya. Hal yang sama juga terjadi pada bahasa sasaran yang akan dituju. Bahasa sasaran dapat berupa bahasa sumber lain seperti C, FORTRAN dan sebagainya, atau bahasa mesin (machine language) yang digunakan oleh suatu prosesor mikro atau suatu komputer besar maupun komputer super.

Karena itu suatu paket kompilator yang tertentu sudah pasti hanya dapat digunakan untuk menerjemahkan suatu bahasa sumber yang tertentu dan menerjemahkannya ke dalam bahasa sasaran yang tertentu juga.

Sejarah perkembangan suatu kompilator sudah dimulai sejak lama, yaitu pada saat mulai diketemukannya komputer pada awal tahun 1950-an. Sejak waktu tersebut teknik dan cara pembentukan suatu kompilator telah berkembang dengan sangat pesat dan pembentukan kompilator dapat dilakukan makin mudah. Demikian pula program bantu (tools) untuk membuat suatu kompilator sudah dapat diperoleh sehingga pembentukan suatu kompilator dapat dilakukan dengan cepat. Kompilator pertama yang dibuat adalah kompilator untuk bahasa FORTRAN.

Walaupun secara tradisional suatu kompilator dikaitkan denga suatu proses penerjemahan suatu bahasa sumber ke dalam suatu bahasa asembler atau bahasa mesin dari suatu komputer, tetapi ada beberapa proses lain yang secara langsung tidak terkait oleh kerja suatu kompilator, tetapi pengembangannya dapat dilakukan seperti pengembangan suatu kompilator. Berikut ini adalah beberapa proses yang pengembangannya dapat dilakukan sejalan dengan pengembangan suatu kompilator, tetapi penggunaannya bukan untuk penerjemahan dari suatu bahasa ke dalam bahasa lain:

a. Pemformatan teks

Suatu pemforamt teks mempunyai masukan berupa aliran karakter, dimana sebagian besar dari karakter tersebut akan dicetak, tetapi bagian lainnya digunakan untuk perintah perintah cetakan seperti ganti bari, keterangan gambar, struktur matematika seperti index dan lain sebagainya.

b. Kompilator silikon

Kompilator ini pada dasarnya hanya mengolah sinyal logika (0 atau 1) atau kelompok sinyal dari suatu sirkuit, sedangkan keluarannya adalah suatu rancangan sirkuit yang didefinisikan oleh suatu bahasa tertentu.

c. Interpreter query

Dalam aplikasi basis data suatu bahasa query, yang merupakan suatu predikat yang mengandung operator boolean maupun relasi, diterjemahkan ke dalam suatu proses pencarian suatu record dalam basis data yang memenuhi predikat yang telah diberikan.

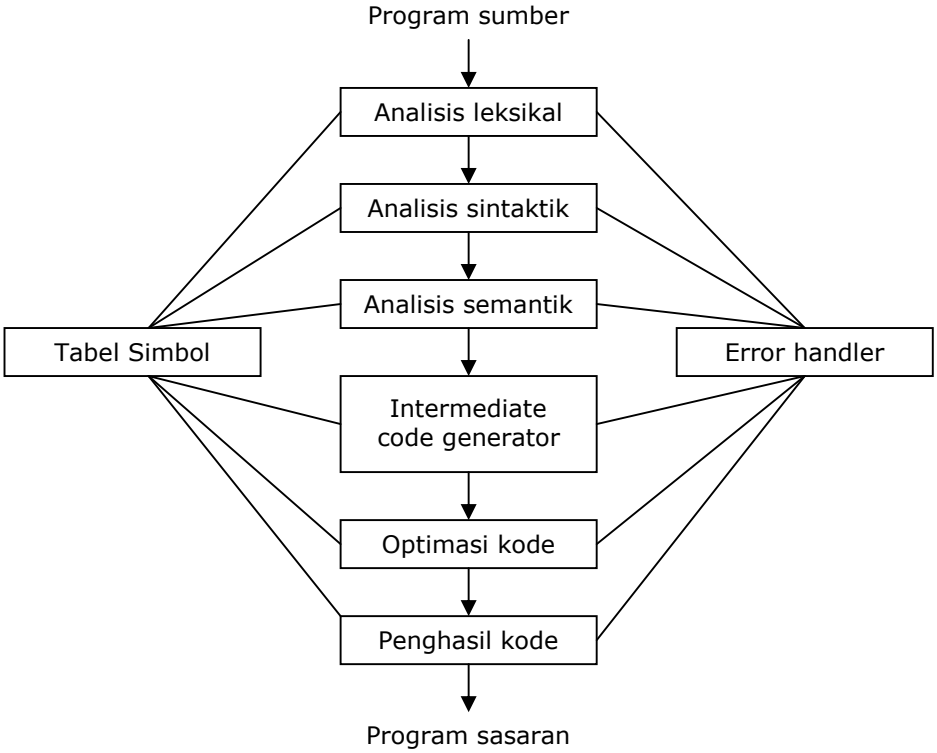
FASE-FASE KOMPILASI

Translator dikelompokkan secara kasar menurut jumlah pelewatan yang sudah dijalani seluruh program sumber. *Compiler* yang standar biasanya menggunakan dua pelewatan diseluruh program sumber. Analisis pelewatan yang pertama adalah menyusun kembali suatu program ke dalam komponen-komponen pokok dan mendapatkan suatu informasi dari program, misalnya penggunaan nama program. Pelewatan yang kedua biasanya menghasilkan suatu program objek dari informasi yang sudah dikumpulkan tersebut.

Proses kompilasi dari suatu kompilator pada dasarnya dapat dibagi ke dalam dua bagian utama yaitu bagian analisis dan bagian sintesis. Pada tahap analisis, program yang ditulis dalam bahasa sumber dibagi dan dipecah ke dalam beberapa bagian yang kemudian akan direpresentasikan ke dalam suatu bentuk antara (intermediate presentation) dari program sumber. Kemudian pada tahap sintetis program sasaran dibentuk berdasarkan representasi antar yang dihasilkan pada tahap analisis. Kedua tahap ini pada umumnya tidak dapat secara jelas dipisahkan, karena adanya keterkaitan yang sangat erat antar kedua tahap tersebut.

Secara umum proses dalam tahap analisis terdiri dari tiga bagian utama yaitu proses analisis leksikal, proses analisis sintaktik dan proses analisis

semantik, sedangkan untuk tahap sintesis terdiri dari dua bagian utama yaitu proses yang menghasilkan kode (*code generator*) dan proses optimasi kode (*code optimizer*), sebelum program sasaran dapat dihasilkan. Dalam melakukan hal ini setiap bagian utama akan berhubungan dan berkomunikasi dengan suatu berkas tabel yang disebut tabel simbol (*symbol table*) yaitu suatu tabel yang berisi semua simbol yang digunakan dalam program sumber. Hubungan antara tahap analisis dan sitesis digambarkan berikut ini:



Jika kecepatan kompilasi merupakan hal yang penting maka strategi satu pelewatan (*one-pass*) dapat digunakan. Pada strategi ini, selagi program dianalisa, segera konversikan juga ke dalam kode objek. Contohnya adalah bahasa Pascal. Jika kecepatan kompilasi tidak terlalu penting maka dapat menggunakan tiga pelewatan (*three-pass*) atau lebih. Pelewatan yang pertama menganalisa program sumber, pelewatan yang kedua menulis kembali program sumber ke dalam bentuk yang lebih efisien dengan menggunakan suatu algoritma optimisasi yang sudah dikenal, dan pelewatan yang ketiga akan menghasilakn suatu kode objek.

Dengan semakin berkembangnya teknologi, baik itu teknologi *hardware* maupun *software*, teknologi *compiler* pun juga berkembang, sehingga hubungan antar jumlah pelewatan dan kecepatan *compiler* menjadi tidak jelas lagi. Hal yang penting untuk dikembangkan adalah kompleksitas bahasa, bukan jumlah pelewatan yang dibutuhkan untuk menganalisa suatu program sumber.

ANALISA INPUT PROGRAM SUMBER

Program sumber dilihat oleh *translator* sebagai suatu kumpulan urutan simbol yang todak berbeda dengan panjang dari ribuan, bahkan sampai ratusan ribu karakter. Suatu bentuk subprogram, ataupun statemen yang diatur rapi oleh programmer, tidak akan terlihat oleh *translator*.

a. Analisa *Lexical*

Merupakan tahap dasar dari suatu kompilasi yang membaca program sumber, karakter demi karakter. Sederetan (satu atau lebih) karakter dikelompokkan menajdi satu kesatuan mengacu kepada pola kesatuan kelompok karakter (token) yang ditentukan dalam bahasa sumber. Analisa lexical mengerjakan pengelompokan urutan karakter ke dalam komponen pokok: *identifier*, *delimiter*, simbol operator, angka, *keyword*, *noise word*, *blank*, komentar, dan seterusnya. Pengelompokan ini akan menghasilkan lexical token (*lexeme*)yang akan digunakan pada tingkatan selanjutnya. Setiap token yang dihasilkan disimpan di dalam tabel simbol. Sederetan karakter yang tidak mengikuti pola token akan dilaporkan sebagai token tak dikenal (*unidentified token*).

Contoh : Misalnya pola token untuk identifier I adalah

I → huruf(huruf|angka)

Lexeme *ab2c* dikenali sebagai token sementara, sedangkan lexeme *2abc* atau *abC* tidak dikenal.

b. Analisa *Syntactic*

Tahap kedua dari kompilasi adalah analisa *syntactic* atau sering disebut *parsing*. Disinilah struktur program yang lebih besar diidentifikasi (statemen, deklarasi, ekspresi, dan lainnya) menggunakan *lexical token* yang dihasilkan oleh *lexical analyzer*. Tahap ni memeriksa kesesuaian pola deretan token dengan aturan sintaks yang ditentukan dalam bahasa sumber. Deretan token yang tidak sesuai aturan sintaks akan dilaporkan sebagai kesalahan sintaks (*syntax error*). Secara logika deretan token yang bersesuaian dengan sintaks tertentu akan dinyatakan ssebagai pohon parsing (*parse tree*). Analisa *syntactic* selalu bekerja bergantian dengan analisa *semantic*. Pertama, *syntactic analyzer* mengidentifikasikan urutan *lexical token* seperti ekspresi, statemen, subprogram, dan lainnya. *Semantic analyzer* kemudian dipanggil untuk memproses unit ini.

Contoh : Misalnya sintaks untuk ekspresi *if-then* E adalah

E → if L then

L → IOA

I → huruf(huruf|angka)

O → <|=|>|<= | >=

A → 0 | 1 | … | 9

Ekspresi *if a2 < 9* then adalah ekspresi sesuai sintaks, sementara ekspresi *if then a2B < 9* atau *if a2 < 9 do* tidak sesuai. Perhatikan bahwa contoh ekspresi terakhir juga mengandung token yang tidak dikenal.\

c. Analisa *Semantic*

Merupakan pusat dari tahapan kompilasi. Disini, struktur *syntactic* yang dikenali oleh *syntactic analyzer* diproses, dan struktur objek *executable* sudah mulai dibentuk. Analisa *semantic* kemudian menjadi jembatan antar *analysis* dan *synthesis* dari kompilasi.

Analyzer semantic menghasilkan suatu kode objek yang *executable* dalam kompilasi yang sederhana, tetapi biasanya bentuk dari kode objek yang *executable* ini merupakan bentuk internal dari final program *executable*, yang kemudian dimanipulasi oleh tahap optimisasi dari *translator* sebelum akhirnya kode *executable* benar-benar dihasilkan.

Tahap ini memeriksa token dan ekspresi dengan acuan batasan-batasan yang ditetapkan, misalnya:

(a) Panjang maksimum token identifier adalah 8 karakter,
(b) Panjang maksimum ekspresi tunggal adalah 80 karakter,
(c) Nilai bilangan bulat adalah –32768 s/d 32767,
(d) Operasi aritmatika harus melibatkan operan-operan yang bertipe sama.

Pada tahap ini, muncul pula fungsi-fungsi tambahan yang penting lainnya, yaitu:

- Symbol-table Maintenance
  Symbol-table merupakan salah satu pusat struktur data di setiap *translator*. *Symbol-table* biasanya berisi suatu masukan untuk setiap identifier yang berbeda yang ditemukan diprogram sumber. *Symbol-table* berisi sesuatu yang lebih dari identifier itu sendiri karena berisi data tambahan tentang atribut *identifier* tersebut: jenisnya (misal varibel sederhana, nama array, nama subprogram, parameter formal, dan lainnya), jenis nilai (integer, riil, dan lainnya), lingkungan referensi, dan informasi lain yang didapat dari program input sampai dengan deklarasi dan penggunaan.
- Penyisipan Implisit Information
  Seringkali informasi bersifat implisit yang harus dibuat eksplisit di program objek yang berlevel lebih rendah.kebanyakan informasi yang implisit ini berada dalam aturan default. Interpretasi digunakan ketika *programmer* tidak memberikan spesifikasi yang eksplisit.
- Pendeteksi Error
  Syntactic dan *semantic analyzer* harus dipersiapkan untuk dapat menangani dengan baik program yang tidak benar seperti halnya menangani program yang benar. *Error* yang umum terjadi adalah salah ketik (misalnya ada delimiter ditengah statemen, deklarsi yang ada ditengah statemen, dan lainnya), penggunaan variabel riil di variabel integer, penggunaan array dua dimensi yang diisi dengan array tiga dimensi, dan lainnya.
  Semantic analyzer tidak hanya mengenali *error* "umum" yang muncul dan memunculkan pesan *error* yang tepat, tetapi juga menentukan cara yang tepat untuk melanjutkan dengan analisa *syntactic* pada sisa program.
- Macro Processing
  Macro merupakan bagian dari teks program yang didefinisikan secara terpisah dan disisipkan ke dalam program pada waktu translasi ketika ada *macro call* di program sumber. Dengan demikian, *macro* dapat juga disebut subprogram. Namun berbeda dengan subprogram yang lain, *macro* ditransalsikan secara terpisah dan dipanggil pada waktu *run-time*. Ketika *macro* diperbolehkan maka *semantic analyzer* harus mengidentifikasi pemanggilan *macro* dalam program sumber dan mempersiapkan *body macro* untuk pemanggilan.
- Compile-time Operations
  Merupakan operasi yang harus ada selama translasi berlangsung untuk mengontrol tranlasi program sumber. Bahasa C mempunyai operasi-operasi tersebut. Operasi "#define" mengizinkan konstanta atau ekspresi untuk dievaluasi sebelum program di-*compile*. Operasi "#ifdef" mengizinkan urutan kode alternatif untuk di-*compile* tergantung dari adanya atau tidak adanya suatu variabel yang pasti. Operasi-operasi bahasa C tersebut sebenernya juga merupakan sebuah macro.

SYNTHESIS PROGRAM OBJEK YANG EXECUTABLE

Tahapan akhir suatu tranlasi terfokus pada pembangunan program yang *executable* dari *output* yang dihasilkan oleh *semantic analyzer*. Tahap ini melibatkan penghasil kode (*code generation*) jika dibutuhkan dan melibatkan juga optimisasi pada program yang sudah dihasilkan. Jika subprogram ditranslasikan secara terpisah atau jika pustaka subprogram digunakan maka tahapan *final linking* dan *loading* dibutuhkan untuk menghasilkan suatu program lengkap yang siap dieksekusi.

a. Pembentukan/Pembangkitan Kode (Code Generator).
Dalam tahap ini bentuk antara dari bahasa sumber yang berupa suatu pohon sintaks diterjemahkan ke dalam suatu bahasa assembli atau bahasa mesin. Tahap ini membangkitkan kode antara (*intermediate code*) berdasarkan pohon parsing. Pohon parsing selanjutnya diterjemahkan oleh suatu penerjemah, misalnya penerjemah berdasarkan sintaks (*syntax-directed translator*). Hasil penerjemahan ini biasanya merupaka perintah tiga alamat (*three-address code*) yang merupakan representasi program untuk suatu mesin abstrak. Bahasa mesin yang dihasilkan adalah bahasa assembli yang merupakan suatu perintah 1 alamat, 1 akumulator. Perintah tiga alamat bisa berbentuk quadruples (*op, arg1, arg2, result*), triples (*op,arg1,arg2*). Ekspresi dengan satu argumen dinyatakan dengan menetapkan arg2 dengan – (*strip,dash*).

b. Optimalisasi Kode
Hasil pembentukan kode yang diperoleh kemudian dibuat lebih kompak lagi dengan melakukan beberapa teknik optimasi supaya dapat diperoleh program yang lebih efisien. Dalam hal ini dilakukan beberapa hal seperti pendeteksian suatu ekspresi yang sering terjadi, sehingga pengulangan tidak perlu terjadi dan lain sebagainya. Pada tahap ini melakukan optimasi (peghematan space dan waktu komputasi) jika mungkin terhadap kode antara. *Semantic analyzer* biasanya menghasilkan suatu *output* program *executable* yang sudah ditranslasi yang berbentuk *intermediate code*, yang kadangkala merupakan kode yang buruk atau tidak efisien. Sebagai contoh, statemen:

```
A = B + C + D
```

Akan menghasilkan suatu *intermediate code*:

```
(a) temp1 = B + C
(b) temp2 = temp1 + D
(c) A = temp2
```

Dimana sebenarnya merupakan suatu kode yang tidak efisien:

1. *Load* register dengan **`B`** (dari (a))
2. *Add* **`C`** ke register
3. *Store* register di **`temp1`**
4. *Load* register dengan **`temp1`** (dari(b))
5. *Add* **`D`** ke register
6. *Store* register di **`temp2`**
7. *Load* register dengan **`temp2`** (dari (c))
8. *Store* register di **`A`**

Instruksi 3 dan 4 sama seperti instruksi 6 dan 7 yang berarti adalah *redundant*, ketika semua data dapat disimpan di register sebelum penyimpanan hasil di **A**.
Optimalisasi diperlukan pada kasus ini untuk menghilangkan inefisiensi kode. Optimalisasi akan mengubah kode yang tidak efisien menjadi kode yang efisien.

c. Penghasil Kode (Code Generation)
Setelah program yang ditranslasi dan representasi internalnya dioptimalisasi maka harus dibentuk sebagai statemen bahasa assembly, kode mesin, atau program objek yang lainnya yang menjadi *output* dari translasi. Kode *output* ini mungkin dapat langsung dieksekusi, atau membutuhkan langkah translasi berikutnya, yaitu *Linking* dan *Loading*.

d. Linking dan Loading
Tahapan akhir yang bersifat opsional adalah menggabungkan potongan-potongan kode yang dihasilkan dari translasi terpisah suatu subprogram ke dalam suatu program *final executable* yang utuh. Hal ini dapat terjadi karena potongan-potongan kode tersebut mempunyai *loader tables* yang dihasilkan oleh *translator*. *Loader tables* inilah yang digunakan oleh *linking loader* untuk menggabungkan potongan-potongan kode tersebut di memori sehingga menghasilkan program *final executable* yang siap untuk dieksekusi.