Bab 2

Landasan Teori dan Konsep Dasar Kompilasi

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 1.1:** Menjelaskan perbedaan antara interpreter dan compiler
- **Sub-CPMK 1.2:** Mengidentifikasi fase-fase kompilator dalam arsitektur kompilator nyata
- **Sub-CPMK 1.3:** Menganalisis trade-off antara one-pass vs multi-pass compiler

2.1 Konsep dan Definisi Kunci

2.1.1 Definisi Kompilator

Secara tradisional, kompilator dipandang sebagai "kotak hitam" yang mengubah kode sumber menjadi kode target executable. Menurut [1], proses ini sebenarnya terdiri dari serangkaian fase yang saling terkait [2]. Secara formal, ***compiler*** adalah program yang melakukan translasi dari bahasa sumber (source language) ke bahasa target (target language), dengan mempertahankan makna semantik dari program sumber.

Menurut Aho, Lam, Sethi, dan Ullman dalam buku klasik "Compilers: Principles, Techniques, dan Tools"[1]:

> “A compiler is a program that can read a program in one language (the source language) and translate it into an equivalent program in another language (the target language).”

2.1.2 Karakteristik Kompilator

Kompilator memiliki beberapa karakteristik penting yang membedakannya dari pemroses bahasa lainnya:

- **Translasi Lengkap**: Kompilator membaca dan menganalisis seluruh program sumber sebelum menghasilkan output.
- **Analisis Mendalam**: Melakukan pengecekan struktur (*syntax*) dan makna (*semantic*) secara menyeluruh.
- **Output Terpisah**: Menghasilkan file terpisah (seperti *object file* atau *executable*) yang dapat berjalan tanpa kode sumber asli.
- **Optimasi**: Menggunakan teknik matematis dan heuristik untuk meningkatkan efisiensi kode hasil translasi.

2.1.3 Interpreter vs Compiler

Perbedaan fundamental antara ***interpreter*** dan ***compiler***:

Aspek	**Compiler**	**Interpreter**
Eksekusi	Compile lalu run	Run langsung
Kecepatan	Lebih cepat	Lebih lambat
Debugging	Lebih sulit	Lebih mudah
Platform	Platform-dependent	Platform-independent
Memory usage	Lebih besar	Lebih kecil

Tabel 2.1: Perbandingan Compiler dan Interpreter

2.1.4 Arsitektur Kompilator

Kompilator modern memiliki arsitektur berlapis yang terdiri dari beberapa fase:

1. **Analysis Phase**
    - Lexical Analysis (Scanner)
    - Syntax Analysis (Parser)
    - Semantic Analysis
2. **Synthesis Phase**
    - Intermediate Code Generation
    - Code Optimization
    - Code Generation

2.2 Teori Utama Compiler

2.2.1 Formal Language Theory

Teori bahasa formal menjadi dasar bagi compiler design:

- **Regular Expressions**: Untuk lexical analysis
- **Context-Free Grammars**: Untuk syntax analysis
- **Finite Automata**: Model recognizer untuk tokens
- **Pushdown Automata**: Model recognizer untuk parsing

2.2.2 One-Pass vs Multi-Pass Compiler

One-Pass Compiler

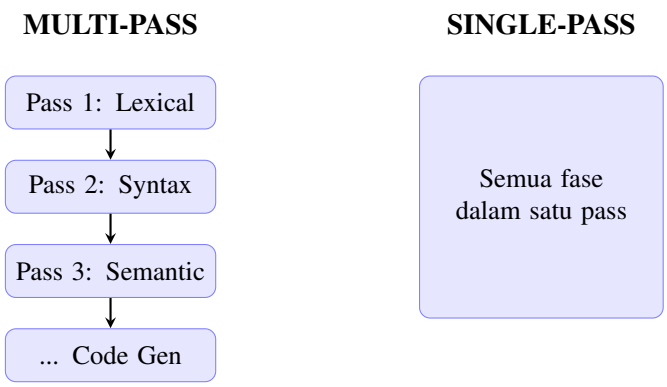
Kompilator *single-pass* mencoba menyelesaikan semua fase dalam satu kali pembacaan kode (*pass*).

- **Kelebihan**: Proses kompilasi sangat cepat dan hemat memori.
- **Kekurangan**: Lebih sulit dikembangkan, optimasi sangat terbatas, dan tidak fleksibel terhadap struktur bahasa yang kompleks.
- **Contoh**: Pascal, implementasi awal bahasa C.

Multi-Pass Compiler

Kompilator modern umumnya menggunakan pendekatan *multi-pass*, di mana setiap fase (atau kelompok fase) dijalankan dalam *pass* terpisah.

- **Kelebihan**: Modularitas tinggi, pemisahan perhatian tiap fase, dan memungkinkan optimasi global yang mendalam.
- **Kekurangan**: Membutuhkan lebih banyak memori dan waktu kompilasi dibandingkan *single-pass*.
- **Contoh**: GCC, LLVM/Clang, modern C++, Java compiler.

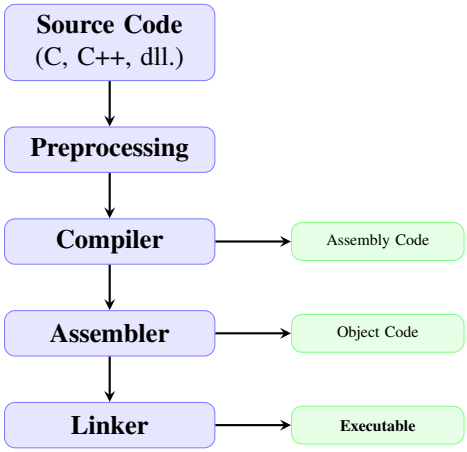


Gambar 2.1: Perbandingan arsitektur Multi-Pass dan Single-Pass Compiler

2.3 Alur Kerja dan Arsitektur Kompilator

2.3.1 Alur Kerja Kompilator: Dari Source ke Executable

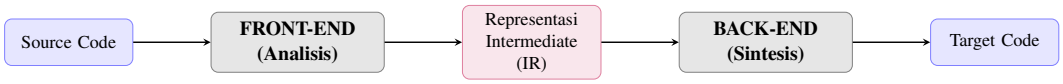
Sebelum membahas detail arsitektur, mari kita lihat gambaran umum alur kerja kompilator secara utuh. Gambar 2.2 menunjukkan transformasi kode dari teks mentah hingga menjadi file yang dapat dieksekusi oleh mesin, sesuai dengan model sistem kompilasi standar [3].



Gambar 2.2: Alur kerja sistem kompilasi secara keseluruhan

2.3.2 Dua Sisi Kompilator: Front-End dan Back-End

Kompilator modern umumnya dibagi menjadi dua bagian utama: **front-end** (analisis) dan **back-end** (sintesis). Pemisahan ini memungkinkan satu front-end (misal untuk bahasa C) digunakan untuk berbagai back-end (misal untuk mesin Intel x86 dan ARM).



Gambar 2.3: Struktur logis kompilator: Pemisahan Analisis dan Sintesis

2.4 Fase-Fase Kompilasi Secara Detail

Di balik pembagian besar front-end dan back-end, terdapat enam fase utama yang bekerja secara sekuensial untuk melakukan transformasi kode.

2.4.1 Analisis Leksikal (Scanner)

Fase ini memecah karakter-karakter dalam *source code* menjadi unit-unit atomik bermakna yang disebut **token**.

- **Input**: String karakter kode sumber.
- **Output**: Stream token (`keyword`, `id`, `literal`, dll.).

2.4.2 Analisis Sintaksis (Parser)

Mengambil token dari scanner dan memeriksa apakah urutannya membentuk struktur yang valid sesuai *grammar* bahasa.

- **Output**: *Abstract Syntax Tree* (AST).

2.4.3 Analisis Semantik

Memeriksa makna dari kode, seperti kecocokan tipe data (*type checking*) dan keberadaan deklarasi variabel (*scope resolution*).

2.4.4 Generasi Intermediate Code (IR)

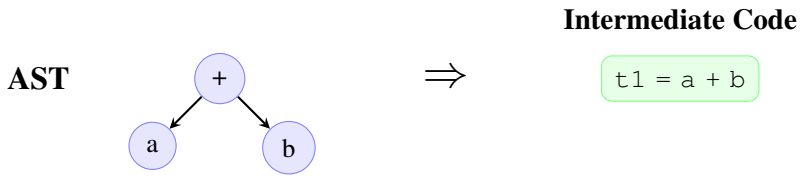
Translasi AST ke bentuk yang lebih dekat dengan instruksi mesin tetapi tetap independen terhadap jenis prosesor tertentu.

2.4.5 Optimasi Kode

Transformasi IR untuk menghasilkan kode yang lebih efisien (cepat dijalankan atau hemat memori) tanpa mengubah maksud program aslinya.

2.4.6 Generasi Kode Target

Tahap akhir yang mengubah IR menjadi instruksi spesifik untuk arsitektur mesin target (misalnya assembly x86 atau ARM).



Gambar 2.4: Visualisasi transformasi dari AST ke Intermediate Code (TAC)

2.5 Contoh Praktis: Alur Kompilasi Program

Mari kita simulasikan bagaimana satu baris kode diolah oleh berbagai fase kompilator.

Kode Sumber:

```
int sum = x + 10;
```

1. **Scanner**: Menghasilkan token (`INT`, `sum`, `ASSIGN`, `x`, `PLUS`, `10`, `SEMI`).
2. **Parser**: Membangun struktur pohon (AST) yang menunjukkan `sum` di-assign dengan hasil operasi binary plus antara `x` dan literal `10`.
3. **Type Checker**: Memastikan `x` adalah tipe numerik (misal `int`) yang valid untuk ditambah dengan `10`.
4. **IR Generator**: Menghasilkan *Three-Address Code*:

```
t1 = x + 10
sum = t1
```

5. **Code Generator**: Menghasilkan instruksi mesin (contoh dalam assembly x86):

```
mov eax, [x]
add eax, 10
mov [sum], eax
```

Aktivitas Pembelajaran

1. **Analisis Compiler**: Identifikasi compiler yang Anda gunakan sehari-hari dan klasifikasikan sebagai one-pass atau multi-pass.

2. **Studi Kasus**: Bandingkan GCC dan Clang dari segi arsitektur dan fase kompilasi.

3. **Eksperimen**: Implementasikan interpreter sederhana untuk kalkulator aritmatika.

4. **Research**: Pelajari compiler untuk bahasa modern (Rust, Go) dan identifikasi fitur inovatifnya.

5. **Debat**: Diskusikan keuntungan dan kerugian JIT compilation vs AOT compilation.

Latihan dan Refleksi

1. Jelaskan perbedaan mendasar antara compiler dan interpreter dengan contoh nyata!

2. Gambarkan flowchart lengkap dari source code hingga executable file!

3. Analisis trade-off antara one-pass dan multi-pass compiler untuk embedded system!

4. Mengapa semantic analysis diperlukan setelah syntax analysis?

5. Buat contoh regular expression untuk mengenali identifier dalam bahasa C!

6. **Refleksi**: Konsep mana yang paling menantang dalam bab ini dan bagaimana cara mengatasinya?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 1.1-1.3

A. Pilihan Ganda

1. Manakah yang BUKAN termasuk fase analysis phase?

   (a) Lexical Analysis

   (b) Syntax Analysis

   (c) Code Generation

   (d) Semantic Analysis

2. Keuntungan utama multi-pass compiler adalah:

(a) Kecepatan kompilasi lebih tinggi

(b) Memory usage lebih rendah

(c) Optimasi yang lebih baik

(d) Debugging lebih mudah

3. Regular expression digunakan dalam:

(a) Semantic Analysis

(b) Code Generation

(c) Lexical Analysis

(d) Syntax Analysis

B. Essay

1. Jelaskan perbedaan antara one-pass dan multi-pass compiler beserta contoh implementasinya!
2. Analisis arsitektur compiler favorit Anda dan jelaskan mengapa arsitektur tersebut efektif!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat menjelaskan perbedaan antara interpreter dan compiler

☐ Saya dapat mengidentifikasi fase-fase kompilator dalam arsitektur nyata

☐ Saya dapat menganalisis trade-off one-pass vs multi-pass compiler

☐ Saya memahami peran formal language theory dalam compiler design

☐ Saya dapat menggambar arsitektur kompilator lengkap

☐ Saya dapat menjelaskan fungsi setiap fase kompilasi

Rangkuman

Bab ini membahas landasan teori dan konsep dasar kompilator, termasuk perbedaan interpreter vs compiler, arsitektur kompilator, teori bahasa formal, dan perbandingan one-pass vs multi-pass compiler.

Poin Kunci:

- Compiler menerjemahkan source code ke target code melalui beberapa fase
- Interpreter mengeksekusi code langsung tanpa kompilasi terpisah
- Arsitektur kompilator terdiri dari analysis dan synthesis phase
- One-pass compiler cepat tapi terbatas, multi-pass fleksibel tapi kompleks
- Formal language theory adalah fondasi matematis untuk compiler design

Kata Kunci: ***Compiler***, ***Interpreter***, ***Lexical Analysis***, ***Syntax Analysis***, ***Semantic Analysis***, ***One-Pass***, ***Multi-Pass***, ***Regular Expression***, ***Context-Free Grammar***

Daftar Pustaka

[1] Alfred V. Aho **andothers**. *Compilers: Principles, Techniques, and Tools*. 2nd. Dragon Book. Pearson Education, 2006.

[2] Diznr. *Six Phases of Compiler*. Online educational resource. 2024. URL: https://diznr.com/six-phases-of-compiler-lexical-syntax-semantic-intermediate-code-generation-optimization-code/.

[3] University of Washington. *CSE P 501: Compiler Construction*. Course syllabus. 2024. URL: https://courses.cs.washington.edu/courses/csep501/21au/syllabus.html.