BUKU AJAR

TEKNIK KOMPILASI

Berbasis Outcome-Based Education (OBE)

Praktis dengan C/C++

Oleh

[Nama Dosen Pengampu]

Digunakan di lingkungan sendiri, sebagai buku ajar mata kuliah ***Teknik Kompilasi*** *pada Program Studi S1 Teknik Informatika*

Program Studi S1 Teknik Informatika

Fakultas Teknik

Universitas

2026

INFORMASI BUKU

Judul	: Buku Ajar Teknik Kompilasi
Subjudul	: Berbasis Outcome-Based Education (OBE), Praktis dengan C/C++
Penulis	: [Nama Dosen Pengampu]
Program Studi	: S1 Teknik Informatika
Fakultas	: Fakultas Teknik
Universitas	: Universitas
Mata Kuliah	: Teknik Kompilasi
SKS	: 3 SKS
Pertemuan	: 16 Pertemuan
Tahun	: 2026
ISBN	: [ISBN jika ada]

*Buku ajar ini disusun sebagai bahan pembelajaran untuk mata kuliah **Teknik Kompilasi** pada Program Studi S1 Teknik Informatika. Buku ini dirancang mengikuti pendekatan **Outcome-Based Education (OBE)** dengan fokus pada pembelajaran berbasis praktik.*

Prakata

Buku ajar ini disusun sebagai bahan pembelajaran untuk mata kuliah **Teknik Kompilasi** pada Program Studi S1 Teknik Informatika. Buku ini dirancang mengikuti pendekatan *Outcome-Based Education (OBE)* dengan fokus pada pembelajaran berbasis praktik.

Buku ini bertujuan untuk memberikan pemahaman dan keterampilan praktis dalam merancang dan mengimplementasikan kompilator untuk bahasa pemrograman. Setiap bab dilengkapi dengan contoh praktis menggunakan bahasa pemrograman C atau C++, serta latihan yang mengarah pada pembangunan komponen-komponen kompilator secara bertahap.

Mata kuliah ini mencakup fase-fase kompilator dari Analisis Leksikal (Lexical Analysis), Analisis Sintaksis (Syntax Analysis), Analisis Semantik (Semantic Analysis), hingga Generasi Kode (Code Generation). Sesuai dengan pendekatan OBE, mahasiswa diharapkan tidak hanya memahami teori, tetapi juga mampu mengimplementasikan setiap fase kompilator secara praktis.

Penyusun
6 Februari 2026

Cara Menggunakan Buku Ini

Buku ajar ini dirancang dengan pendekatan OBE untuk memaksimalkan pencapaian pembelajaran Anda. Berikut panduan penggunaan buku ini:

Struktur Buku

Bab I: Pendahuluan dan Orientasi
Memperkenalkan tujuan buku, keterkaitan dengan RPS, dan konteks kurikulum OBE.

Bab II: Landasan Teori
Menyajikan fondasi teoretis Teknik Kompilasi yang menjadi basis pembelajaran seluruh bab berikutnya.

Bab III-XIX: Unit Materi Inti
Setiap bab mencakup satu topik utama Teknik Kompilasi dengan struktur lengkap: Sub-CPMK, materi, aktivitas, latihan, asesmen, dan checklist.

Bab Lainnya: Evaluasi dan Integrasi
Berisi asesmen komprehensif dan panduan refleksi untuk mengukur pencapaian kompetensi secara menyeluruh.

Komponen dalam Setiap Bab

1. **Sub-CPMK**: Baca dengan seksama untuk memahami kompetensi yang harus dicapai
2. **Materi Pokok**: Pelajari dengan cermat, jalankan semua contoh kode
3. **Aktivitas Pembelajaran**: Lakukan secara mandiri atau berkelompok
4. **Latihan**: Kerjakan untuk menguji pemahaman Anda
5. **Asesmen**: Gunakan untuk mengukur pencapaian Sub-CPMK
6. **Checklist**: Centang setelah yakin menguasai setiap indikator

Identitas Mata Kuliah

Nama Program Studi	: S1 Teknik Informatika
Nama Mata Kuliah	: Teknik Kompilasi
Kode Mata Kuliah	: [Kode MK]
Semester	: [Semester]
SKS / Bobot Kredit	: 3 SKS
Dosen Pengampu	: [Nama Dosen Pengampu]
Tanggal Penyusunan	: 6 Februari 2026

Capaian Pembelajaran Lulusan (CPL)

CPL yang dibebankan pada mata kuliah ini mencakup kompetensi lulusan dalam aspek pengetahuan, keterampilan, dan sikap sesuai dengan kurikulum OBE.

Capaian Pembelajaran Mata Kuliah (CPMK)

Kemampuan atau kompetensi spesifik yang diharapkan mahasiswa kuasai setelah menyelesaikan mata kuliah:

1. **CPMK-1:** Mampu menjelaskan fase-fase kompilasi dan arsitektur kompilator.
2. **CPMK-2:** Mampu menerapkan teori bahasa formal dalam analisis leksikal dan sintaksis.
3. **CPMK-3:** Mampu mengimplementasikan komponen kompilator sederhana menggunakan C/C++.
4. **CPMK-4:** Mampu melakukan optimasi kode dan menangani error dalam proses kompilasi.

Daftar Isi

Daftar Gambar

Daftar Tabel

Bab 1

Pendahuluan dan Orientasi OBE

Sub-CPMK yang Dicakup dalam Bab Ini:

- Memahami keterkaitan antara kurikulum *Outcome-Based Education* (OBE) dengan desain sistem kompilator.
- Mengidentifikasi alur pembelajaran Teknik Kompilasi melalui peta konsep buku ajar.
- Merencanakan strategi belajar mandiri untuk menguasai setiap fase kompilasi secara sistematis.

1.1 Tujuan Buku Ajar

Buku ajar ini dirancang sebagai panduan komprehensif untuk menguasai ***Teknik Kompilasi*** secara sistematis dan terukur, selaras dengan standar *Outcome-Based Education* (OBE) [1]. Fokus utama buku ini adalah pada pembangunan fondasi teoretis dan keterampilan praktis dalam membangun arsitektur kompilator modern. Menurut [2], kompilasi adalah proses transformasi dari bahasa sumber ke bahasa sasaran.

1. Memberikan pemahaman mendalam tentang setiap fase kompilasi, mulai dari analisis leksikal hingga generasi kode target.
2. Mengembangkan kemampuan merancang dan mengimplementasikan komponen-komponen utama kompilator seperti *lexer*, *parser*, dan *semantic analyzer*.
3. Membangun keterampilan dalam optimasi kode dan manajemen memori pada *runtime*.
4. Memfasilitasi pencapaian *Capaian Pembelajaran Lulusan* (CPL) dan *Capaian Pembelajaran Mata Kuliah* (CPMK) yang telah ditetapkan dalam kurikulum.

Setelah mempelajari buku ini secara menyeluruh, mahasiswa diharapkan mampu:

- Menjelaskan arsitektur kompilator dan fungsi setiap fasenya.
- Membangun pemroses bahasa (*language processor*) menggunakan teknik manual maupun generator (*Flex/Bison*).
- Mengelola struktur data kompleks seperti *Symbol Table* dan *Abstract Syntax Tree* (AST).
- Menghasilkan kode target yang efisien untuk arsitektur mesin tertentu.
- Melakukan evaluasi performa dan optimasi pada tingkat *intermediate code* dan *target code*.

1.2 Keterkaitan Buku Ajar dengan RPS Berbasis OBE

Buku ajar ini dirancang selaras dengan Rencana Pembelajaran Semester (RPS) mata kuliah Teknik Kompilasi yang berbasis *Outcome-Based Education* (OBE). Keterkaitan ini memastikan bahwa setiap materi yang disajikan memiliki kontribusi langsung terhadap kompetensi lulusan, sebagaimana diterapkan dalam standar kurikulum internasional [3].

1.3 Arsitektur Kompilator Modern

Arsitektur kompilator modern umumnya terbagi menjadi front-end dan back-end [4].

1.3.1 Alignment dengan CPL dan CPMK

Struktur buku ini disusun untuk mendukung pencapaian indikator-indikator berikut:

- **CPL-1 (Pengetahuan)**: Menguasai konsep teoretis analisis leksikal, sintaksis, semantik, dan generasi kode secara mendalam.
- **CPL-3 (Keterampilan Khusus)**: Mampu merancang, mengimplementasikan, dan mengevaluasi sistem kompilator lengkap.
- **CPMK-1 s.d CPMK-6**: Meliputi seluruh spektrum pengembangan kompilator dari arsitektur awal hingga evaluasi performa akhir.

Setiap bab dalam buku ini memuat daftar **Sub-CPMK** di bagian awal untuk memberikan fokus yang jelas bagi mahasiswa mengenai kompetensi spesifik yang akan dikuasai.

1.3.2 Integrasi Metode Pembelajaran Aktif

Sesuai dengan RPS berbasis OBE, buku ini mendukung berbagai metode pembelajaran:

- **Problem-Based Learning**: Melalui studi kasus penanganan *semantic errors* dan optimasi lokal.
- **Project-Based Learning**: Pengembangan kompilator secara bertahap dalam setiap bab.
- **Praktikum Terbimbing**: Implementasi komponen menggunakan *tooling* industri seperti LLVM atau Clang.

1.3.3 Sistem Evaluasi Berbasis Kompetensi

Komponen asesmen yang disediakan di setiap akhir bab (latihan, asesmen, dan *checklist*) dirancang untuk mengukur pencapaian *Sub-CPMK* secara objektif, yang nantinya akan menjadi bobot penilaian utama dalam UTS dan UAS (total 35% sesuai RPS).

1.4 Petunjuk Penggunaan Buku Ajar

1.4.1 Untuk Mahasiswa

Mengingat kompleksitas pengembangan kompilator, mahasiswa disarankan untuk menggunakan buku ini dengan langkah-langkah berikut:

Tahap Persiapan:

1. Pahami target *Sub-CPMK* di awal bab agar fokus belajar tetap terjaga.
2. Tinjau kembali materi prasyarat (Struktur Data dan Algoritma) jika diperlukan.

Tahap Implementasi (Eksperimental):

1. Pelajari kode contoh yang disediakan dan jalankan menggunakan *compiler* atau *interpreter* yang sesuai.
2. Lakukan modifikasi pada parameter *lexer* atau aturan *grammar* untuk melihat dampaknya terhadap proses *parsing*.
3. Gunakan perangkat lunak pendukung seperti *Flex*, *Bison*, atau *Graphviz* untuk memvisualisasikan AST.

Tahap Evaluasi:

1. Kerjakan latihan refleksi untuk memperdalam pemahaman teoretis.

2. Lakukan penilaian mandiri menggunakan *checklist* kompetensi di akhir bab.

3. Gabungkan komponen yang telah dibuat di setiap bab menjadi satu proyek kompilator utuh.

1.4.2 Untuk Dosen

Dosen dapat memanfaatkan buku ini sebagai instrumen pembelajaran utama:

- **Modul Praktikum**: Gunakan aktivitas pembelajaran di setiap bab sebagai panduan tugas mingguan.
- **Bank Soal**: Manfaatkan bagian asesmen sebagai referensi dalam menyusun soal UTS (CPMK-1, 2) dan UAS (CPMK 1 s.d 6).
- **Alat Ukur Capaian**: Gunakan rubrik penilaian dan indikator dalam buku untuk mengukur ketercapaian outcomes mahasiswa.

1.5 Konteks Kurikulum OBE

1.5.1 Filosofi OBE dalam Teknik Kompilasi

Outcome-Based Education (OBE) adalah pendekatan yang menekankan pada apa yang bisa dilakukan oleh mahasiswa di akhir masa studi, bukan sekadar apa yang diajarkan. Dalam Teknik Kompilasi, hal ini berarti mahasiswa tidak hanya menghafal algoritma *parsing*, tetapi mampu membangun sebuah program yang secara nyata dapat menerjemahkan sebuah bahasa ke bahasa lain.

Empat Prinsip Utama OBE dalam Buku Ini:

1. **Clarity of Focus**: Fokus pada hasil akhir berupa kompilator yang berfungsi.
2. **Designing Down**: Materi disusun mundur dari kebutuhan akhir sebuah sistem *backend* kompilator.
3. **High Expectations**: Mahasiswa didorong untuk mengimplementasikan optimasi kode yang efisien.
4. **Expanded Opportunity**: Menyediakan berbagai aktivitas belajar mulai dari teori hingga proyek tim.

1.5.2 Implementasi Tahapan OBE

Buku ini membagi proses pencapaian kompetensi dalam empat pilar utama:

Komponen OBE	Implementasi dalam Teknik Kompilasi
Defined Outcomes	Sub-CPMK eksplisit untuk setiap fase (Leksikal, Sintaksis, dst).
Designing Down	Kurikulum dimulai dari pengenalan bahasa ke deteksi kesalahan hingga emisi kode.
Student Activity	Implementasi manual mesin *state* dan penggunaan alat otomatisasi generator.
Continuous Assessment	*Weekly reflection* dan audit kualitas kode secara berkala.

Tabel 1.1: Penerapan OBE dalam Pengembangan Kompilator

1.5.3 Hierarki Capaian Pembelajaran

Konsep Penting

Pemahaman mahasiswa terhadap Teknik Kompilasi divalidasi melalui hierarki capaian:

- **CPL**: Mahasiswa menguasai teori kompilasi secara utuh sebagai sarjana teknik.
- **CPMK**: Mahasiswa mampu mengintegrasikan fase-fase kompilasi menjadi sistem yang koheren.
- **Sub-CPMK**: Mahasiswa mahir dalam satu spesialisasi fase (misalnya: *Register Allocation*).

1.6 Peta Konsep Teknik Kompilasi

Buku ini disusun dalam 19 bab yang mencakup seluruh spektrum pengembangan kompilator, mulai dari landasan teori hingga referensi pengembangan lanjut:

1. **Bab I**: Pengenalan dan Konteks OBE
2. **Bab II**: Arsitektur Kompilator - Gambaran umum sistem
3. **Bab III-IV**: *Front-end* - Analisis leksikal dan representasi regular
4. **Bab V-VI**: *Syntax Analysis* - *Parsing* dan *grammar* formal
5. **Bab VII-X**: *Middle-end* - *Intermediate code*, tabel simbol, analisis semantik, dan penanganan kesalahan
6. **Bab XI-XIV**: *Back-end* - Tata letak memori, *code generation*, alokasi register, dan manajemen *stack*
7. **Bab XV-XVI**: *Analysis & Evaluation* - *Compiler tools* dan evaluasi performa

8. **Bab XVII-XIX**: *Assessment & Resources* - Evaluasi kompetensi, lampiran, dan daftar referensi

Alur Pembelajaran:

- **Fase Analisis (Bab II-VI)**: Memahami bagaimana bahasa manusia diterjemahkan menjadi token dan pohon hirarki.
- **Fase Transformasi (Bab VII-X)**: Memastikan kebenaran makna dan mengubahnya menjadi representasi antara.
- **Fase Sintesis (Bab XI-XIV)**: Membangun instruksi mesin yang optimal sesuai arsitektur target.
- **Fase Profesional (Bab XV-XIX)**: Menggunakan alat bantu modern dan melakukan standarisasi kualitas.

1.7 Proyek Buku: Compiler Subset C

Sepanjang Bab 2 hingga Bab 16, kita secara bertahap membangun **satu compiler untuk subset bahasa C**. Setiap bab menambah satu lapis ke proyek yang sama: spesifikasi token (Bab 3), lexer hand-written (Bab 3), lexer Flex (Bab 4), grammar (Bab 5), parser hand-written (Bab 6), teori bottom-up (Bab 7), parser Bison (Bab 8), AST (Bab 9), symbol table (Bab 10), type checking (Bab 11), IR (Bab 12), runtime (Bab 13), code generation (Bab 14), optimasi (Bab 15), dan integrasi (Bab 16). Spesifikasi berikut menjadi acuan tunggal agar semua contoh dan kode mengacu ke bahasa yang sama.

1.7.1 Spesifikasi Token Proyek Subset C

Token yang dikenali oleh compiler proyek (untuk Bab 3–4):

- **Identifier**: huruf atau underscore diikuti huruf, angka, atau underscore. Pola: `[a-zA-Z_][a-zA-Z0-9_]*`
- **Kata kunci**: `int`, `float`, `print`. (Nanti dapat diperluas: `if`, `else`, `while`.)
- **Literal**: integer `[0-9]+`, float `[0-9]+.[0-9]+`, string `"..."` dalam tanda kutip ganda.
- **Operator**: `+`, `-`, `*`, `/`, `=`, `==`, `!=`, `<`, `>`, `<=`, `>=`.
- **Punctuator**: `;`, `,`, `()`, kurung kurawal `{}`.
- **Komentar**: satu baris `//` dan banyak baris `/* */`; serta whitespace (spasi, tab, newline) diabaikan.

1.7.2 Spesifikasi Grammar Proyek Subset C

Grammar dalam BNF untuk Bab 5–8 (dan parser proyek):

- **Program**: barisan statement.
- **Statement**: deklarasi `;` | assignment `;` | print-statement `;`
- **Deklarasi**: `int` identifier | `float` identifier
- **Assignment**: identifier = ekspresi
- **Print-statement**: `print (` string-literal `)` | `print (` ekspresi `)`
- **Ekspresi**: term | ekspresi + term | ekspresi − term
- **Term**: factor | term $\star$ factor | term / factor
- **Factor**: literal | identifier | `(` ekspresi `)`

Precedence: $\star$ dan / lebih tinggi dari + dan −; associativity kiri untuk semuanya.

1.7.3 Peta Bab ke Lapis Proyek

Bab	Lapis proyek
3	Spesifikasi token + teori RE/FA
3	Lexer hand-written (mengikuti spec token)
4	Lexer proyek (Flex, file `simplec.l`)
5	Grammar proyek (BNF/EBNF di atas)
6	Parser hand-written (mengikuti grammar proyek)
7	Teori LR; grammar proyek termasuk kelas LR
8	Parser proyek (Bison, file `simplec.y`)
9	AST proyek (`ast.h`/`ast.c`)
10	Symbol table proyek (`symtab.h`/`symtab.c`)
11	Type checking proyek
12	IR proyek (TAC/quadruples dari AST)
13	Runtime; asumsi proyek untuk stack/activation record
14	Code generation proyek (IR $\rightarrow$ assembly)
15	Optimasi proyek (basic block, constant folding, dll.)
16	Integrasi dan presentasi compiler subset C lengkap

Semua bab dari Bab 3 sampai Bab 16 merujuk ke spesifikasi ini. Kode dan contoh dalam bab tersebut mengacu ke token set dan grammar di atas, serta ke file proyek (`simplec.l`, `simplec.y`, dan seterusnya) yang tumbuh di folder `proyek-compiler-subset-c/`.

Aktivitas Pembelajaran

1. **Analisis RPS**: Pelajari RPS Teknik Kompilasi dan identifikasi bagaimana CPL Pengetahuan (CPL-1) diukur melalui proyek pengembangan kompilator.

2. **Pemetaan Fase**: Buat diagram alir yang memetakan bab-bab dalam buku ini ke dalam tiga pilar utama: *Front-end*, *Middle-end*, dan *Back-end*.

3. **Tooling Audit**: Identifikasi *software* pendukung (GCC, Flex, Bison, Clang) yang akan digunakan di setiap bab berdasarkan peta konsep.

4. **Diskusi Kurikulum**: Diskusikan mengapa penguasaan Teknik Kompilasi sangat krusial dalam mencapai standar kompetensi lulusan Teknik Informatika di industri modern.

Latihan dan Refleksi

1. Jelaskan secara spesifik bagaimana pendekatan OBE membantu mahasiswa dalam menghadapi kompleksitas algoritma *parsing*!

2. Mengapa setiap aktivitas praktikum harus memiliki rubrik penilaian yang eksplisit dalam konteks OBE?

3. Bagaimana cara Anda memantau kemajuan pembangunan proyek kompilator Anda menggunakan *checklist* kompetensi?

4. Hubungkan antara *Target Architecture* (Bab XII) dengan tujuan akhir pencapaian kompetensi dalam RPS!

5. **Refleksi**: Sejauh mana Anda memahami bahwa membangun sebuah kompilator adalah bukti nyata pencapaian kompetensi teknik yang utuh?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Orientasi OBE Kompilator

A. Pilihan Ganda

1. Manakah yang merupakan contoh *Outcome* nyata dalam mata kuliah ini?

(a) Membaca buku Naga (Dragon Book)

(b) Lulus ujian teori

(c) Menghasilkan kode assembly yang dapat dijalankan

(d) Mengetahui sejarah FORTRAN

2. *Designing Down* dalam buku ini berarti:

(a) Mendesain dari tingkat mesin ke bahasa tingkat tinggi

(b) Menyusun materi berdasarkan urutan fase kompilasi untuk mencapai produk akhir

(c) Mengurangi beban materi yang sulit

(d) Hanya fokus pada latihan praktak

B. Essay

1. Jelaskan keterkaitan antara peta konsep (Bab I s.d XIX) dengan pencapaian kompetensi profesional seorang *System Programmer*!

2. Desainlah satu indikator pencapaian kompetensi untuk Sub-CPMK "Membangun Lexer Hand-written"!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya memahami filosofi OBE dalam konteks pengembangan sistem kompilator.

☐ Saya dapat memetakan hierarki CPL, CPMK, dan Sub-CPMK Teknik Kompilasi ke dalam konten buku.

☐ Saya memahami alur kerja *Front-end*, *Middle-end*, dan *Back-end* melalui peta konsep.

☐ Saya telah menyiapkan *software stack* yang diperlukan untuk proyek semester ini.

☐ Saya berkomitmen untuk melakukan *self-assessment* secara berkala di setiap akhir bab.

Rangkuman

Bab ini memberikan fondasi bagi mahasiswa untuk memahami bagaimana buku ajar ini

disusun menggunakan standar OBE guna menjamin penguasaan Teknik Kompilasi yang utuh dan profesional.

Poin Kunci:

- Fokus utama adalah pencapaian *outcome* berupa sistem kompilator yang berfungsi.
- Kurikulum dirancang mundur (*designing down*) untuk memandu mahasiswa dari analisis ke sintesis kode.
- Peta konsep menunjukkan integrasi 19 bab sebagai satu kesatuan ekosistem pembelajaran.
- Peran aktif mahasiswa dalam evaluasi diri adalah kunci keberhasilan dalam sistem OBE.

Kata Kunci: ***OBE***, ***Teknik Kompilasi***, ***Peta Konsep***, ***CPL***, ***Outcome***, ***Compiler Design***

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 [2], proses ini sebenarnya terdiri dari serangkaian fase yang saling terkait [5]. 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"[2]:

> “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

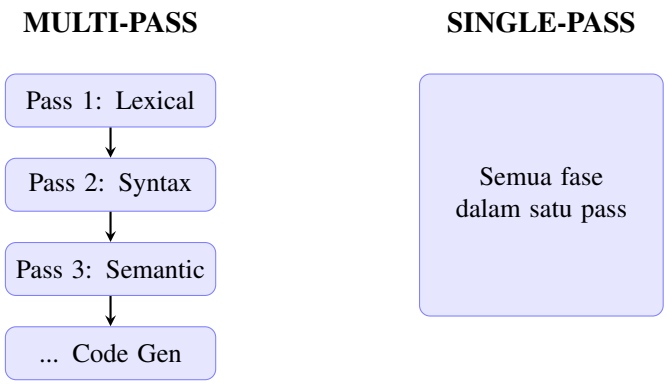
Kompilator *single-pass* mencoba menyelesaikan semua fase dalam satu kali pembacaan kode (*pass*).

- **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.

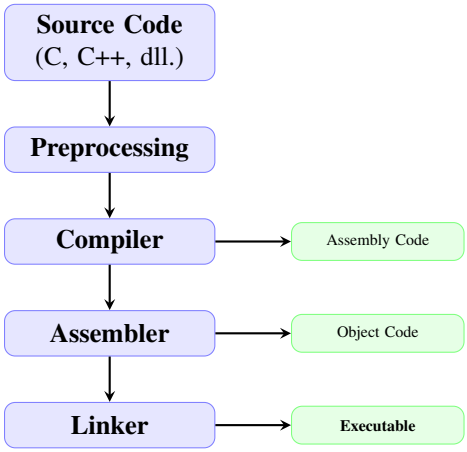


Gambar 2.1: Perbandingan arsitektur Multi-Pass dan Single-Pass Compiler

2.3 Alur Kerja dan Arsitektur Kompilator

2.3.1 Alur Kerja Kompilator: Dari Source ke Executable

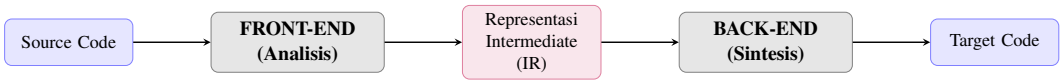
Sebelum membahas detail arsitektur, mari kita lihat gambaran umum alur kerja kompilator secara utuh. Gambar 2.2 menunjukkan transformasi kode dari teks mentah hingga menjadi file yang dapat dieksekusi oleh mesin, sesuai dengan model sistem kompilasi standar [6].



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)

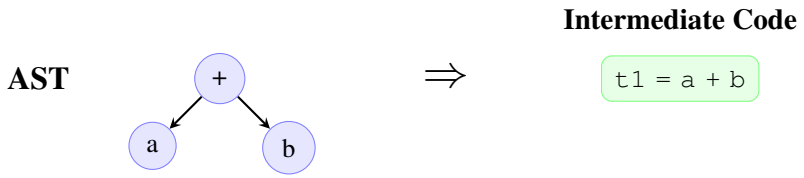
Translasi AST ke bentuk yang lebih dekat dengan instruksi mesin tetapi tetap independen terhadap jenis prosesor tertentu.

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***

Bab 3

Lexical Analysis dan Regular Expression

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 2.1:** Membuat regular expression untuk token spesifik bahasa
- **Sub-CPMK 2.2:** Mengimplementasikan NFA dan DFA untuk token recognition

3.1 Pengenalan Lexical Analysis

3.1.1 Peran Lexical Analyzer

Lexical Analysis (atau ***Lexer/Scanner***) adalah tahap pertama dalam kompilasi yang bertugas membaca aliran karakter dari program sumber dan mengelompokkannya ke dalam unit-unit bermakna yang disebut *token* [7, 8].

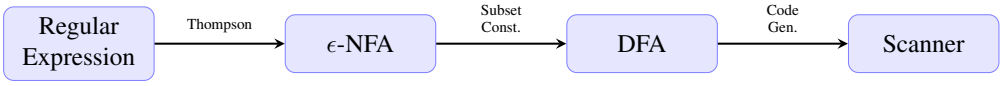
- Menghapus whitespace dan komentar.
- Melaporkan lexical error.

3.1.2 Alur Teoretis Lexical Analysis

Sebagai landasan untuk memahami lexical analysis, kita perlu mempelajari teori formal yang mendasarinya. Alur ini menunjukkan bahwa lexical analysis dalam kompilator modern menggunakan teori *formal language*, khususnya *regular languages*.

3.1.3 Token dan Lexeme

- **Token**: Kategori unit leksikal (misal: `IDENTIFIER`, `KEYWORD`).



Gambar 3.1: Alur konversi dari regular expression ke implementasi scanner

- **Lexeme**: String aktual yang mewakili token (misal: `x`, `if`).
- **Attribute Value**: Informasi tambahan (misal: nilai literal, entri tabel simbol).

3.2 Regular Expression

3.2.1 Definisi dan Operasi Dasar

Regular expression (regex) adalah notasi formal untuk mendeskripsikan pola string dalam suatu **regular language**. Operasi dasar meliputi:

Operasi	Simbol	Contoh
Karakter	a	string "a"
Alternasi	$\|$	$a\|b$ (a atau b)
Konkatenasi	ab	string "ab"
Kleene Star	$*$	a^* (0 atau lebih)
Positive Plus	$+$	a^+ (1 atau lebih)
Optional	$?$	$a?$ (0 atau 1)

Tabel 3.1: Operator dasar Regular Expression

3.2.2 Implementasi Leksikal dengan Regex

Dalam pembangunan kompilator, setiap elemen leksikal (token) didefinisikan dengan regex guna menjamin kepastian pola:

- **Identifier**: `[a-zA-Z_][a-zA-Z0-9_]*`
- **Integer**: `[0-9]+`
- **Float**: `[0-9]+\.[0-9]+([eE][+-]?[0-9]+)?`
- **String Literal**: `"([^"]|\\.)*"`

3.2.3 Sifat Regular Language

Bahasa reguler dapat dikenali secara efisien menggunakan finite automata dan tertutup terhadap operasi gabungan, pengulangan, serta penyambungan, namun tidak dapat mendeskripsikan struktur *nested* mendalam.

3.3 Finite Automata

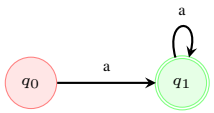
Finite automata adalah model matematika yang digunakan untuk mengenali string dalam suatu bahasa.

3.3.1 NFA (Nondeterministic Finite Automata)

NFA dapat memiliki beberapa kemungkinan transisi untuk simbol input yang sama dan mengizinkan ϵ-transitions (pindah state tanpa membaca input). Hal ini memudahkan konstruksi dari regex tetapi memerlukan simulasi yang lebih kompleks.

3.3.2 DFA (Deterministic Finite Automata)

DFA bersifat deterministik (tepat satu transisi untuk setiap simbol) dan tidak mengizinkan ϵ-transitions. DFA sangat efisien untuk dijalankan di mesin karena hanya membutuhkan satu kali pembacaan karakter untuk setiap transisi state.



DFA untuk a^+

Gambar 3.2: Contoh DFA sederhana

3.3.3 Konversi NFA ke DFA: Subset Construction

Algoritma subset construction digunakan untuk menghasilkan DFA yang ekuivalen dari sebuah NFA.

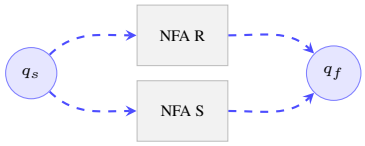
1. **Start state DFA** adalah ϵ-closure dari start state NFA.
2. Untuk setiap state DFA, hitung transisi ke set NFA states berikutnya berdasarkan simbol input.
3. Ambil ϵ-closure dari setiap set tersebut untuk menjadi state DFA baru.
4. State DFA menjadi **accept state** jika mengandung setidaknya satu accept state NFA.

3.4 Metode Konstruksi NFA: Algoritma Thompson

Algoritma Thompson menyediakan template standar untuk mengubah setiap operator Regular Expression menjadi bagian dari NFA secara rekursif.

3.4.1 Template Dasar Thompson

- **Literal (a)**: Transisi tunggal dari state awal ke akhir dengan simbol 'a'.
- **Union (R|S)**: Menggunakan ϵ-transitions untuk bercabang ke NFA R dan NFA S secara paralel.
- **Concatenation (RS)**: Menghubungkan NFA R langsung ke NFA S melalui transisi ϵ.
- **Kleene Star (R^*)**: Menambahkan loop balik ϵ dan jalur pintas ϵ untuk mengakomodasi pengulangan nol kali.



Gambar 3.3: Template Thompson untuk Union ($R|S$)

3.5 Implementasi Lexer Hand-written

Pendekatan *hand-written* memberikan kontrol penuh dan sangat berguna untuk memahami detail proses tokenisasi.

3.5.1 Struktur Data Token

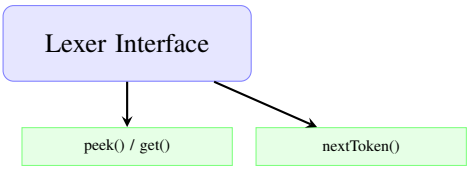
Token minimal harus menyimpan kategori (*type*), string asli (*lexeme*), serta informasi posisi (baris dan kolom) untuk keperluan pelaporan kesalahan.

```
enum class TokenType {
    IDENTIFIER, KEYWORD, INTEGER_LITERAL, FLOAT_LITERAL,
    OP_PLUS, OP_ASSIGN, SEMICOLON, END_OF_FILE, INVALID
};

struct Token {
    TokenType type;
    std::string lexeme;
    int line;
    int column;
};
```

3.5.2 Arsitektur Kelas Lexer

Kelas Lexer mengelola input string dan memprosesnya karakter demi karakter menggunakan pointer posisi.



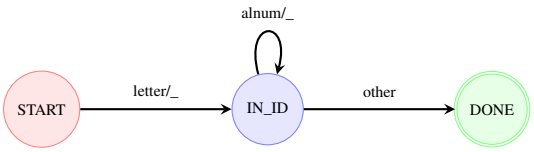
Gambar 3.4: Komponen utama kelas Lexer

3.6 Logika State Machine Token

Dalam implementasi manual, kita menggunakan logika *Finite State Machine* (FSM) untuk mengenali berbagai jenis token.

3.6.1 State Machine untuk Identifier dan Keywords

Transisi dimulai dari karakter alfabet atau _, kemudian diikuti oleh karakter alfanumerik. Setelah lexeme terkumpul, dilakukan pengecekan apakah ia termasuk kata kunci (*keyword*). Selain Flex, terdapat berbagai generator lexer modern lainnya seperti re2c yang berorientasi pada kecepatan [9] dan RE/flex yang mendukung C++ dengan lebih baik [10].



Gambar 3.5: State machine untuk identifikasi identifier

3.6.2 State Machine untuk Angka (Literals)

- **Integer**: Kumpulan digit [0-9].
- **Float**: Digit diikuti oleh titik (.), kemudian digit lagi.

```
Token Lexer::scanNumber() {
    bool isFloat = false;
    std::string lexeme;
    while (isdigit(peek())) lexeme += get();
    if (peek() == '.') {
        isFloat = true;
        lexeme += get();
        while (isdigit(peek())) lexeme += get();
    }
    return Token(isFloat ? FLOAT : INT, lexeme);
}
```

3.7 Handling Komentar dan Kesalahan

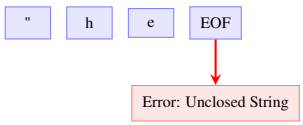
3.7.1 Whitespace dan Komentar

Kompilator biasanya mengabaikan spasi, tab, dan baris baru. Komentar (`//` atau `/*...*/`) juga harus diabaikan tanpa menghasilkan token ke parser.

```
void Lexer::skipLineComment() {
    get(); get(); // skip //
    while (peek() != '\n' && peek() != '\0') get();
}
```

3.7.2 Penanganan Kesalahan Leksikal

Jika lexer menemui karakter yang tidak dikenal atau string yang tidak tertutup hingga akhir file (*unclosed string*), maka harus dihasilkan informasi kesalahan yang informatif.



Gambar 3.6: Ilustrasi unclosed string error

3.8 Lexer Generator dan Best Practices

3.8.1 Penggunaan Flex

Untuk proyek skala besar, penggunaan generator seperti *Flex* lebih disarankan. Kita cukup menulis pola regex, dan Flex akan men-generate code C/C++ yang sangat efisien secara otomatis.

3.8.2 Best Practices

1. **Lookahead**: Gunakan fungsi `peek()` untuk mengintip karakter berikutnya tanpa menghabiskannya.

2. **Position Tracking**: Simpan baris dan kolom untuk mempermudah *debugging* bagi pemrogram.

3. **Longest Match**: Selalu ambil token terpanjang yang cocok (misal: == lebih prioritas daripada =).

3.9 Lexer Generator: Flex dan re2c

Lexer generator adalah alat yang secara otomatis membangun *finite automata* dari spesifikasi ekspresi reguler.

3.9.1 Flex (Fast Lexical Analyzer)

Flex adalah generator standar yang paling luas digunakan. Spesifikasinya ditulis dalam file `.l` dengan tiga bagian utama: *Definitions*, *Rules*, dan *User Code*.

Gambar 3.7: Alur kerja Flex

3.9.2 re2c

re2c adalah generator modern yang menghasilkan kode C/C++ berperforma sangat tinggi. Berbeda dengan Flex, re2c menyisipkan spesifikasinya langsung di dalam komentar khusus pada file sumber C/C++.

3.9.3 Perbandingan Pendekatan

Fitur	Hand-written	Flex	re2c
Produktivitas	Rendah	Tinggi	Tinggi
Performa	Tinggi	Sedang	Sangat Tinggi
Maintenance	Sulit	Mudah	Mudah

Tabel 3.2: Perbandingan metode konstruksi lexer

Aktivitas Pembelajaran

1. **Regex Practice**: Buat regular expression untuk email address, URL, dan phone number.
2. **NFA to DFA**: Konversi NFA untuk identifier ke DFA dan gambarkan state diagramnya.
3. **Lexer Implementation**: Implementasikan lexer sederhana untuk bahasa dengan 5 token types.
4. **Tool Exploration**: Coba Flex atau ANTLR untuk generate lexer dari regex definitions.

5. **Performance Analysis**: Bandingkan performance hand-coded vs generated lexer.

Latihan dan Refleksi

1. Buat regular expression untuk mengenali semua valid identifiers dalam bahasa C!
2. Gambarkan NFA dan DFA untuk regular expression `(a|b)*abb`!
3. Implementasikan DFA recognizer untuk binary numbers yang habis dibagi 3!
4. Analisis kelebihan dan kekurangan menggunakan generated lexer!
5. Desain transition table untuk lexer dengan 10 token types!
6. **Refleksi**: Bagian mana dari lexical analysis yang paling sulit dan mengapa?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 2.1-2.2

A. Pilihan Ganda

1. Regular expression `a(b|c)*d` mengenali:
   (a) ad, abd, acd
   (b) ad, abd, acd, abcd, acbd
   (c) ad, abd, acd, abcd, accd, abccd
   (d) Hanya string yang dimulai dengan a dan diakhiri d
2. Perbedaan utama NFA dan DFA adalah:
   (a) NFA lebih cepat dari DFA
   (b) NFA memiliki epsilon transitions
   (c) DFA memiliki lebih banyak states
   (d) NFA tidak bisa mengenali regular languages
3. Tool yang paling umum untuk generate lexer adalah:
   (a) GCC
   (b) Flex

(c) Make

(d) GDB

B. Essay

1. Jelaskan langkah-langkah mengkonversi NFA ke DFA dengan contoh konkret!
2. Desain dan implementasikan lexer untuk bahasa sederhana dengan minimal 8 token types!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat membuat regular expression untuk token spesifik bahasa

☐ Saya dapat mengimplementasikan NFA untuk token recognition

☐ Saya dapat mengkonversi NFA ke DFA

☐ Saya dapat mengimplementasikan DFA lexer

☐ Saya memahami perbedaan hand-coded vs generated lexer

☐ Saya dapat menggunakan tools modern untuk lexical analysis

Rangkuman

Bab ini membahas lexical analysis, regular expression, dan finite automata sebagai fondasi untuk implementasi lexer. Mahasiswa belajar membuat regex, mengimplementasikan NFA/DFA, dan memahami trade-off berbagai pendekatan lexer implementation.

Poin Kunci:

- Lexical analysis mengubah stream of characters menjadi stream of tokens
- Regular expression adalah notasi powerful untuk mendeskripsikan token patterns
- NFA mudah dibuat tapi DFA lebih efisien untuk execution
- Generated lexer (Flex) lebih maintainable, hand-coded lebih optimal
- Table-driven lexer adalah pendekatan yang balance antara performance dan maintainability

Kata Kunci: ***Lexical Analysis***, ***Regular Expression***, ***NFA***, ***DFA***, ***Token***, ***Lexeme***, ***Flex***, ***Table-Driven Lexer***

Bab 4

Parsing dan Syntax Analysis

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 2.3:** Membangun recursive descent parser untuk grammar LL(1)
- **Sub-CPMK 2.4:** Menggunakan parser generator (Flex/Bison) untuk bahasa sederhana

4.1 Dasar Syntax Analysis dan CFG

Analisis sintaksis atau *parsing* bertujuan untuk memastikan bahwa urutan token yang dihasilkan oleh lexer sesuai dengan aturan tata bahasa (grammar) yang telah ditentukan. Dalam praktiknya, proses ini sering kali diotomatisasi menggunakan generator parser seperti GNU Bison [11, 12].

4.1.1 Context-Free Grammar (CFG)

CFG merupakan notasi formal untuk mendefinisikan bahasa. Sebuah CFG G didefinisikan sebagai 4-tuple (V, Σ, R, S) di mana:

- V: Himpunan variabel atau *non-terminals*.
- Σ: Himpunan *terminals* (token).
- R: Aturan produksi (*rules*).
- S: Simbol awal (*start symbol*).

4.1.2 Notasi BNF dan EBNF

BNF (Backus-Naur Form) menggunakan produksi eksplisit, sedangkan **EBNF (Extended BNF)** menambahkan operator seperti $\star$ (repetisi 0+), + (repetisi 1+), dan [ ] (opsional) untuk membuat grammar lebih ringkas.

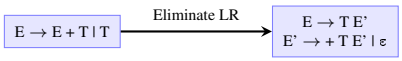
4.2 Top-Down Parsing dan LL(1)

4.2.1 Recursive Descent Parser

Merupakan parser top-down yang menggunakan sekumpulan prosedur rekursif untuk mengenali struktur bahasa pemrograman.

4.2.2 Eliminasi Masalah Grammar

Recursive descent tidak dapat menangani **Left Recursion** dan seringkali membutuhkan **Left Factoring** agar keputusan lookahead menjadi deterministik.



Gambar 4.1: Transformasi eliminasi Left Recursion

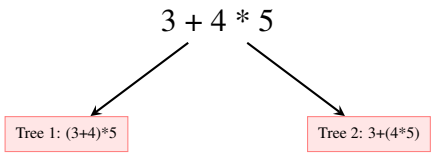
4.3 Struktur Derivasi dan Ambiguitas

4.3.1 Leftmost vs Rightmost Derivation

Leftmost derivation (LD) selalu mengganti non-terminal paling kiri, sering digunakan pada top-down parser. Sebaliknya, *Rightmost derivation* (RD) mengganti non-terminal paling kanan, umum pada bottom-up parser.

4.3.2 Ambiguitas Grammar

Grammar disebut ambigu jika satu input dapat menghasilkan lebih dari satu *parse tree*. Hal ini berbahaya karena dapat menyebabkan salah interpretasi logika program (misal: urutan operasi aritmatika).



Gambar 4.2: Ilustrasi ambiguitas grammar

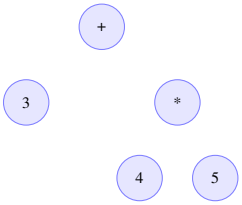
4.4 Parse Tree dan Abstract Syntax Tree (AST)

4.4.1 Parse Tree (Concrete Syntax Tree)

Representasi visual lengkap dari derivasi string, termasuk semua terminal dan non-terminal pendukung.

4.4.2 Abstract Syntax Tree (AST)

Versi ringkas dari parse tree yang hanya menyimpan informasi esensial untuk tahap kompilasi berikutnya (semantik dan *code generation*).



Gambar 4.3: Contoh struktur AST

4.5 Integrasi Lexer dan Parser (Bison)

Parser generator modern seperti **Bison** bekerja sama dengan **Flex** melalui mekanisme integrasi token.

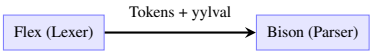
4.5.1 Mekanisme yylval

Lexer mengirimkan nilai semantik token (misal: nilai konstanta atau nama variabel) ke parser melalui variabel global `yylval`.

4.5.2 Struktur File Bison (.y)

```
%token NUMBER IDENTIFIER
%left '+' '-'
%left '*' '/'
```

```
%%
expr: expr '+' expr | NUMBER ;
%%
```



Gambar 4.4: Integrasi data antara Flex dan Bison

4.6 Implementasi: Hand-written Recursive Descent

Penerapan top-down parsing yang paling umum secara manual adalah melalui *Recursive Descent Parser*. Setiap *non-terminal* dalam grammar direpresentasikan oleh satu fungsi C++.

4.6.1 Arsitektur Parser

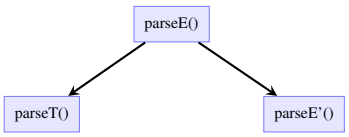
Fungsi-fungsi parser akan saling memanggil secara rekursif mengikuti struktur grammar. Terminal dicocokkan menggunakan fungsi `match(tokenType)`.

4.6.2 Contoh Implementasi

Berikut adalah kerangka parser sederhana untuk ekspresi:

```
void Parser::parseE() {
    parseT();
    parseEPrime();
}

void Parser::parseEPrime() {
    if (lookahead == '+') {
        match('+');
        parseT();
        parseEPrime();
    }
    // Epsilon case: do nothing
}
```



Gambar 4.5: Hierarki pemanggilan fungsi Recursive Descent

4.7 Precedence dan Error Recovery

4.7.1 Menangani Precedence

Dalam recursive descent, precedence diatur melalui level pemanggilan fungsi. Operator dengan precedence lebih tinggi (misal: perkalian) dipanggil lebih dalam dibanding operator dengan precedence rendah (misal: penjumlahan).

4.7.2 Error Recovery: Panic Mode

Agar parser tidak berhenti pada kesalahan pertama, kita menerapkan *Panic Mode Recovery*. Jika terjadi kesalahan, parser akan membuang token (*skipping*) sampai menemukan token sinkronisasi seperti `;` atau `}`.

```
void Parser::synchronize() {
    while (!isAtEnd()) {
        if (peek().type == SEMICOLON) return;
        switch (peek().type) {
            case CLASS: case FUN: case VAR: case IF: return;
        }
        advance();
    }
}
```

4.8 Bottom-Up Parsing: Shift-Reduce dan LR

4.8.1 Konsep Shift-Reduce

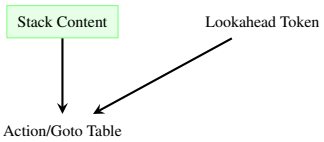
Bottom-up parsing bekerja dengan dua operasi utama:

- **Shift**: Memindahkan token dari input ke stack.
- **Reduce**: Mengganti simbol di puncak stack yang cocok dengan *handle* (RHS sebuah produksi) menjadi non-terminal (LHS).

4.8.2 LR Parsing

LR parsing (L: Left-to-right, R: Rightmost derivation in reverse) adalah metode yang paling powerful. Varian LR meliputi:

- **SLR(1)**: Simple LR, menggunakan *Follow set* dasar.
- **LALR(1)**: Look-Ahead LR, standar industri (digunakan Bison/Yacc).
- **Canonical LR(1)**: Paling powerful namun membutuhkan tabel sangat besar.



Gambar 4.6: Model konseptual LR Parser

4.9 Parser Generator: Bison Deep Dive

Integrasi antara Flex dan Bison merupakan standar industri dalam membangun front-end kompilator yang tangguh dan efisien [13]. Bison menghasilkan parser LALR(1) yang sangat efisien dari spesifikasi formal.

4.9.1 Resolusi Konflik

Bison dapat mendeteksi dua tipe konflik:

1. **Shift/Reduce Conflict**: Parser bingung apakah akan memasukkan token baru atau melakukan reduksi (misal: masalah *dangling else*).

2. **Reduce/Reduce Conflict**: Beberapa aturan produksi cocok dengan handle yang sama (biasanya menandakan grammar ambigu yang parah).

4.9.2 Manajemen Precedence di Bison

Bison memungkinkan pengaturan precedence tanpa mengubah grammar melalui direktif `%left`, `%right`, dan `%nonassoc`.

```
%left '+' '-'
%left '*' '/'
%right '^'
%%
expr: expr '+' expr | expr '*' expr | NUMBER ;
```

Dengan deklarasi di atas, Bison secara otomatis memproses `3 + 4 * 5` sebagai `3 + (4 * 5)` tanpa perlu grammar multi-level yang kompleks.

Aktivitas Pembelajaran

1. **Grammar Design**: Buat CFG untuk bahasa ekspresi aritmatika sederhana.

2. **LL(1) Conversion**: Konversi grammar dengan left recursion ke LL(1).

3. **Recursive Descent**: Implementasikan recursive descent parser untuk kalkulator.

4. **Bison Practice**: Buat parser untuk bahasa dengan assignment dan control flow.

5. **Error Recovery**: Implementasikan panic mode recovery dalam parser Anda.

Latihan dan Refleksi

1. Identifikasi apakah grammar berikut LL(1): $S \rightarrow aSb|\epsilon$!

2. Hapus left recursion dari grammar: $E \rightarrow E + T|T$!

3. Buat recursive descent parser untuk grammar: $S \rightarrow (S)S|\epsilon$!

4. Jelaskan perbedaan antara SLR(1) dan LALR(1) parser!

5. Implementasikan error recovery with panic mode untuk parser Anda!

6. **Refleksi**: Konsep parsing mana yang paling sulit dan bagaimana cara memahaminya?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 2.3-2.4

A. Pilihan Ganda

1. Grammar dengan left recursion TIDAK cocok untuk:
   (a) LL(1) parser
   (b) LR(1) parser
   (c) Recursive descent parser
   (d) Top-down parser

2. Tool yang digunakan untuk generate LR parser adalah:
   (a) Flex
   (b) Bison
   (c) Make
   (d) GCC

3. Lookahead dalam LL(1) berarti:

(a) 1 token lookahead

(b) 1 character lookahead

(c) 1 production lookahead

(d) 1 derivation lookahead

B. Essay

1. Jelaskan langkah-langkah mengkonversi grammar dengan left recursion ke LL(1)!
2. Desain dan implementasikan parser untuk bahasa dengan variabel assignment dan arithmetic expressions!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat membangun recursive descent parser untuk grammar LL(1)

☐ Saya dapat menggunakan parser generator (Flex/Bison) untuk bahasa sederhana

☐ Saya dapat mengidentifikasi apakah suatu grammar LL(1)

☐ Saya dapat menghapus left recursion dari grammar

☐ Saya dapat mengimplementasikan error recovery dalam parser

☐ Saya memahami perbedaan top-down dan bottom-up parsing

Rangkuman

Bab ini membahas parsing dan syntax analysis, termasuk CFG, LL(1) parsing, recursive descent, LR parsing, dan parser generator tools. Mahasiswa belajar membangun parser manual dan menggunakan tools modern.

Poin Kunci:

- Parser memverifikasi sintaks dan membangun parse tree dari token stream
- LL(1) grammar cocok untuk recursive descent parser
- LR parser lebih powerful tapi lebih kompleks
- Parser generator (Bison) otomatisasi pembuatan parser dari grammar

- Error recovery penting untuk parser yang robust

Kata Kunci: ***Parsing***, ***CFG***, ***LL(1)***, ***Recursive Descent***, ***LR Parser***, ***Bison***, ***Syntax Analysis***, ***Error Recovery***

Bab 5

Symbol Table dan Scope Management

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 3.1:** Mengimplementasikan symbol table dengan nested scopes
- **Sub-CPMK 3.2:** Melakukan type checking untuk ekspresi kompleks

5.1 Dasar Symbol Table dan Perannya

Symbol Table adalah struktur data fundamental yang menyimpan informasi tentang identifiers (variabel, fungsi, tipe) yang muncul dalam kode sumber.

5.1.1 Kebutuhan Symbol Table

Dalam fase analisis semantik, kompilator perlu menjawab pertanyaan:

- Apakah variabel x sudah dideklarasikan?
- Apa tipe data dari y?
- Apakah z merupakan variabel lokal atau global?

5.1.2 Komponen Entry

Setiap entri dalam symbol table minimal menyimpan:

- **Name**: Nama identifier.
- **Type**: Tipe data (misal: `int`, `float`).
- **Scope**: Level nesting tempat identifier berada.
- **Location**: Alamat memori atau offset (untuk fase *code generation*).

5.2 Struktur Data: Hash Table dan Scope Stack

Agar operasi pencarian (*lookup*) berjalan efisien, symbol table biasanya diimplementasikan menggunakan **Hash Table**.

5.2.1 Implementasi Hash Table

Dalam C++, kita dapat menggunakan `std::unordered_map` untuk memetakan nama identifier ke objek `Symbol`.

```
struct Symbol {
    string name;
    string type;
    int line;
};

class Scope {
    unordered_map<string, Symbol*> table;
    Scope* parent;
public:
    Symbol* lookup(string name) {
        if (table.count(name)) return table[name];
        if (parent) return parent->lookup(name);
        return nullptr;
    }
};
```

5.2.2 Manajemen Scope Stack

Setiap kali kompilator menemukan blok baru (`{`), scope baru dibuat dan ditumpuk di atas scope sebelumnya. Saat keluar dari blok (`}`), scope teratas akan dihapus atau dinonaktifkan.

5.3 Shadowing dan Resolusi Nama

5.3.1 Shadowing

Shadowing terjadi ketika sebuah identifier di scope dalam memiliki nama yang sama dengan identifier di scope luar. Dalam kasus ini, referensi ke nama tersebut akan merujuk pada deklarasi yang paling dekat (paling dalam).

5.3.2 Resolusi Nama (Name Resolution)

Proses ini mencocokkan setiap penggunaan nama variabel dengan deklarasi yang tepat. Algoritma lookup akan mencari secara bertahap:

1. Cari di scope saat ini.

2. Jika tidak ada, naik ke parent scope.
3. Ulangi sampai ke scope global.
4. Jika tetap tidak ditemukan, laporkan *Undeclared Identifier Error*.

Gambar 5.1: Alur resolusi nama pada nested scopes

5.4 Penanganan Scope Entry dan Exit

5.4.1 Function Scope

Saat mendeklarasikan fungsi, parser harus membuat scope baru untuk menampung parameter fungsi dan variabel lokal di dalamnya.

5.4.2 Block Scope

Setiap blok kode yang dibatasi kurung kurawal menciptakan scope sementara. Kompilator memanggil `beginScope()` saat menemukan `{` dan `endScope()` saat menemukan `}`.

5.4.3 Contoh Kasus

```
int x = 1; // Global
void f() {
    int x = 2; // Shadows global x
    {
        int x = 3; // Shadows f's x
    }
}
```

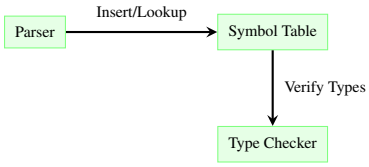
Symbol table akan mengelola tiga versi variabel `x` tersebut pada level nesting yang berbeda.

5.5 Implementasi Symbol Table yang Lengkap

Implementasi yang robust memerlukan manajemen memori yang baik untuk setiap entri simbol.

```
class SymbolTable {
    Scope* current;
public:
    void beginScope() { current = new Scope(current); }
    void endScope() {
        Scope* old = current;
```

```
        current = current->parent;
        delete old;
    }
    bool insert(string name, Symbol* s) {
        return current->add(name, s);
    }
    Symbol* lookup(string name) {
        return current->find(name);
    }
};
```



Gambar 5.2: Interaksi Symbol Table dalam integrasi sistem

Aktivitas Pembelajaran

1. **Hash Table Implementation**: Implementasikan symbol table dengan hash table dan chaining.

2. **Scope Testing**: Buat program dengan nested scopes dan test symbol table operations.

3. **Type Checker**: Implementasikan type checker untuk ekspresi aritmatika dan assignment.

4. **Debug Visualization**: Buat tool untuk visualisasi symbol table dan scope hierarchy.

5. **Performance Analysis**: Bandingkan berbagai hash functions untuk symbol table.

Latihan dan Refleksi

1. Implementasikan symbol table with support for nested scopes menggunakan hash table!

2. Buat type checker untuk bahasa dengan int, float, dan boolean types!

3. Analisis complexity dari symbol table operations (insert, lookup, delete)!

4. Implementasikan scope stack untuk bahasa dengan function definitions!

5. Desain error reporting system untuk type errors!

6. **Refleksi**: Bagian mana dari symbol table implementation yang paling menantang?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 3.1-3.2

A. Pilihan Ganda

1. Data structure yang PALING cocok untuk symbol table adalah:
   (a) Linked list
   (b) Array
   (c) Hash table
   (d) Binary tree

2. Scope management menggunakan:
   (a) Queue
   (b) Stack
   (c) Priority queue
   (d) Heap

3. Type checking memastikan:
   (a) Syntax correctness
   (b) Type compatibility
   (c) Runtime efficiency
   (d) Code optimization

B. Essay

1. Jelaskan implementasi symbol table dengan nested scopes dan berikan contoh kode!

2. Desain type system untuk bahasa dengan arrays, functions, dan pointers!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat mengimplementasikan symbol table dengan nested scopes

☐ Saya dapat melakukan type checking untuk ekspresi kompleks

☐ Saya memahami berbagai implementasi symbol table

☐ Saya dapat mengelola scope dengan stack structure

☐ Saya dapat mendesain type system yang sederhana

☐ Saya dapat mengimplementasikan type inference algorithm

Rangkuman

Bab ini membahas symbol table dan scope management, termasuk implementasi hash table, nested scopes, type system, dan type checking. Mahasiswa belajar membangun data structure fundamental untuk semantic analysis.

Poin Kunci:

- Symbol table menyimpan informasi identifiers dengan akses cepat
- Nested scopes dikelola menggunakan stack structure
- Type checking memastikan type compatibility dalam expressions
- Hash table adalah implementasi efisien untuk symbol table
- Type system adalah fondasi untuk semantic analysis

Kata Kunci: ***Symbol Table***, ***Scope Management***, ***Type System***, ***Type Checking***, ***Hash Table***, ***Nested Scopes***, ***Type Inference***

Bab 6

Semantic Analysis dan Error Handling

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 3.3:** Menangani semantic error dengan pesan yang informatif

6.1 Abstract Syntax Tree (AST) Deep Dive

Abstract Syntax Tree (AST) adalah representasi internal program yang telah disederhanakan. Analisis semantik bertugas memastikan bahwa program memiliki makna yang valid sesuai dengan aturan bahasa pemrogramannya, di luar sekadar kebenaran struktur sintaksisnya [14].

6.1.1 Struktur Node AST

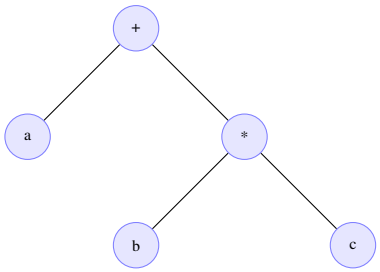
Setiap node dalam AST mewakili konstruk bahasa (misal: `IfStmt`, `BinaryExpr`).

1. **Expression Nodes**: Literal, Identifier, Unary/Binary operations.
2. **Statement Nodes**: Assignment, Loop, Conditional branches.
3. **Declaration Nodes**: Variabel, Fungsi, Tipe.

6.2 Sistem Tipe dan Type Checking

6.2.1 Tipe Data dalam Kompilator

Sistem tipe mendefinisikan aturan bagaimana tipe data diasosiasikan dengan ekspresi. Tipe dasar meliputi `int`, `float`, `bool`, dan `void`.



Gambar 6.1: Representasi AST untuk ekspresi `a + b * c`

6.2.2 Algoritma Type Checking

Type checking dilakukan dengan menelusuri (*traversing*) AST secara *post-order*.

1. Untuk leaf node: ambil tipe dari symbol table atau literal.

2. Untuk node internal: verifikasi kompatibilitas tipe operan (misal: `int + float` mungkin memerlukan promosi tipe).

```
Type checkBinaryExpr(BinaryExpr* node) {
    Type left = check(node->left);
    Type right = check(node->right);
    if (!isCompatible(left, right)) {
        error("Type mismatch: " + left.str() + " and " + right.str());
    }
    return resultType(left, right);
}
```

6.3 Analisis Kontekstual dan Validasi

Selain tipe, kompilator juga melakukan pengecekan aturan kontekstual:

- **Control Flow**: Memastikan statement `break` atau `continue` hanya muncul di dalam loop.

- **Function Signature**: Verifikasi jumlah dan tipe parameter pada pemanggilan fungsi.

- **Return Type**: Memastikan statement `return` memberikan nilai sesuai tanda tangan fungsi.

6.3.1 Error Reporting yang Informatif

Setiap kesalahan harus dilaporkan dengan posisi (baris dan kolom) serta konteks yang jelas untuk membantu pemrogram memperbaiki kode.

6.4 Praktikum: Implementasi Type Checker

Mahasiswa akan mengimplementasikan *semantic analyzer* sederhana menggunakan pola *Visitor Pattern* untuk menelusuri AST.

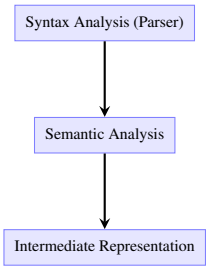
6.4.1 Visitor Pattern

Pola ini memisahkan algoritma analisis dari struktur data AST, sehingga memudahkan penambahan aturan semantik baru tanpa mengubah kelas node AST.

```
class TypeChecker : public ASTVisitor {
    void visit(BinaryExpr* node) override {
        // Cek tipe operan biner
    }
    void visit(Assignment* node) override {
        // Cek kecocokan tipe variabel dan nilai
    }
};
```

6.5 Kesimpulan Analisis Semantik

Analisis semantik menjembatani antara tugas parser (struktur) dan code generator (instruksi). Keberhasilan fase ini menjamin bahwa program yang dikompilasi memiliki makna yang konsisten dan mematuhi spesifikasi bahasa.



Gambar 6.2: Aliran informasi melalui fase semantik

Aktivitas Pembelajaran

1. **Semantic Rules**: Definisikan semantic rules untuk bahasa sederhana.
2. **AST Builder**: Implementasikan AST construction dari parse tree.
3. **Type Checker**: Bangun type checker dengan error reporting informatif.
4. **Error Recovery**: Implementasikan error recovery untuk semantic errors.

5. **Symbol Table Integration**: Integrasikan semantic analyzer dengan symbol table.

Latihan dan Refleksi

1. Identifikasi semantic errors dalam potongan kode yang diberikan!
2. Buat semantic rules untuk function calls dan parameter passing!
3. Implementasikan type checker untuk expressions dengan multiple types!
4. Desain error messages yang informatif untuk berbagai semantic errors!
5. Analisis trade-off antara strict vs lenient semantic checking!
6. **Refleksi**: Bagaimana semantic analysis mempengaruhi kualitas compiler?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 3.3

A. Pilihan Ganda

1. Semantic analyzer bertugas untuk:
   (a) Memverifikasi syntax correctness
   (b) Memverifikasi semantic correctness
   (c) Mengoptimasi kode
   (d) Mengenerate kode target
2. Error reporting yang baik harus mencakup:
   (a) Error message saja
   (b) Line number saja
   (c) Line number, column, dan context
   (d) Hanya error code
3. Type coercion adalah:
   (a) Error handling mechanism
   (b) Automatic type conversion

(c) Type checking algorithm

(d) Optimization technique

B. Essay

1. Jelaskan strategi error recovery dalam semantic analysis dan berikan contoh!
2. Desain dan implementasikan semantic analyzer untuk bahasa dengan variabel assignment dan arithmetic expressions!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat menangani semantic error dengan pesan yang informatif

☐ Saya dapat mengimplementasikan syntax-directed translation

☐ Saya dapat membangun AST dari parse tree

☐ Saya dapat mengintegrasikan semantic analyzer dengan symbol table

☐ Saya dapat mendesain error recovery strategies

☐ Saya dapat mengimplementasikan type checking dengan coercion

Rangkuman

Bab ini membahas semantic analysis dan error handling, termasuk syntax-directed translation, type system implementation, error detection, dan recovery strategies. Mahasiswa belajar membangun semantic analyzer yang robust.

Poin Kunci:

- Semantic analysis memverifikasi meaning dan correctness program
- Syntax-directed translation menggabungkan parsing dengan semantic analysis
- Type checking memastikan type compatibility dan consistency
- Error reporting yang baik memberikan informasi yang jelas dan berguna
- Error recovery memungkinkan compiler melanjutkan proses compilation

Kata Kunci: ***Semantic Analysis***, ***Syntax-Directed Translation***, ***Type Checking***, ***Error Handling***, ***AST***, ***Type Coercion***, ***Error Recovery***

Bab 7

Three-Address Code Generation

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 4.1:** Merancang three-address code representation

7.1 Pengenalan Three-Address Code (TAC)

Three-Address Code (TAC) adalah bentuk Intermediate Representation (IR) yang setiap instruksinya memiliki maksimal tiga operandi (biasanya satu hasil dan dua operan).

7.1.1 Format Instruksi TAC

Format umum instruksi TAC adalah: `x = y op z`.

- `x`: Tujuan (biasanya variabel *temporary*).
- `y`, `z`: Operan (variabel, konstanta, atau temporary).
- `op`: Operator (aritmatika, logika, atau relasional).

7.1.2 Temporary Variables

Kompilator secara otomatis membangkitkan variabel sementara (*temporaries*) untuk menyimpan hasil antara dari ekspresi kompleks. Misal, `a + b * c` ditransformasikan menjadi:

```
t1 = b * c
t2 = a + t1
```

7.2 Representasi: Quadruples dan Triples

Ada beberapa cara menyimpan instruksi TAC di dalam memori:

7.2.1 Quadruples

Setiap instruksi disimpan dalam struktur dengan empat *fields*: `(op, arg1, arg2, result)`. Keunggulannya adalah kemudahan dalam optimasi karena setiap instruksi memiliki lokasi eksplisit.

ID	op	arg1	arg2	result
(0)	*	b	c	t1
(1)	+	a	t1	t2

Gambar 7.1: Representasi Quadruples untuk `a + b * c`

7.2.2 Triples

Representasi yang lebih efisien memori dengan mengacu pada ID instruksi sebelumnya sebagai operan: `(op, arg1, arg2)`. Hasil antara direferensikan melalui index array instruksi tersebut.

7.3 Translasi Ekspresi dan Penugasan

7.3.1 Skema Translasi AST ke TAC

Proses ini biasanya dilakukan melalui penelusuran AST (*recursive traversal*). Fungsi translasi untuk ekspresi biner mengembalikan variabel *temporary* yang menampung hasilnya.

```
string translateExpr(ASTNode* node) {
    if (node->isLiteral()) return node->value;
    if (node->isBinary()) {
        string t1 = translateExpr(node->left);
        string t2 = translateExpr(node->right);
        string temp = generateTemp();
        emit(node->op, t1, t2, temp);
        return temp;
    }
}
```

7.3.2 Manajemen Label

Untuk struktur kontrol, kita memerlukan label unik agar parser dapat melakukan *jump* (lompatan) antar blok kode.

7.4 Struktur Kontrol: Label dan Jump

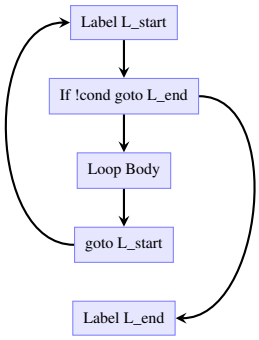
7.4.1 Translasi If-Then-Else

Struktur kondisional diterjemahkan menggunakan instruksi lompatan bersyarat (*conditional jump*) dan tanpa syarat (*unconditional jump*).

```
    if (condition) goto L1
    // Code for False branch
    goto L2
L1: // Code for True branch
L2: // Next statement
```

7.4.2 Translasi While-Loop

Loop memerlukan label di awal untuk pengulangan dan label di akhir untuk terminasi berdasarkan evaluasi kondisi.



Gambar 7.2: Struktur aliran TAC untuk statement While

7.5 Praktikum: TAC Generator dari AST

Mahasiswa akan mengimplementasikan fungsi `genTAC()` pada setiap kelas node AST untuk menghasilkan instruksi IR secara otomatis.

7.5.1 Pengelolaan Temp Table

Penting untuk memastikan nama variabel *temporary* (`t1, t2, ...`) unik di seluruh program dan didaftarkan ke *temporary table* untuk manajemen alokasi memori nantinya.

```
void genTAC(ASTNode* node) {
    if (node->type == ASSIGN) {
        string src = translateExpr(node->right);
        emit("STORE", src, "-", node->left->name);
    }
}
```

Aktivitas Pembelajaran

1. **Expression Translation**: Konversi berbagai ekspresi kompleks ke three-address code.
2. **Control Flow**: Implementasikan translator untuk if-then-else dan while loops.
3. **Function Translation**: Bangun translator untuk function calls dan parameter passing.
4. **Array Handling**: Implementasikan array access dan assignment dalam three-address code.
5. **TAC Generator**: Buat generator lengkap dari AST ke three-address code.

Latihan dan Refleksi

1. Konversi ekspresi `a * (b + c) - d / e` ke three-address code!
2. Terjemahkan statement `for(i=0; i<10; i++) x = x + i;` ke three-address code!
3. Implementasikan temporary variable management dengan optimalisasi!
4. Analisis keuntungan three-address code untuk optimasi!
5. Bandingkan three-address code dengan SSA form!
6. **Refleksi**: Bagaimana three-address code menyederhanakan code generation?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 4.1

A. Pilihan Ganda

1. Three-address code memiliki maksimal:
   (a) 2 operands
   (b) 3 operands
   (c) 4 operands

(d) Tidak terbatas

2. Temporary variables digunakan untuk:

   (a) Menyimpan hasil sementara

   (b) Optimasi kode

   (c) Error handling

   (d) Debugging

3. Keuntungan utama three-address code adalah:

   (a) Eksekusi lebih cepat

   (b) Ukuran kode lebih kecil

   (c) Memudahkan optimasi

   (d) Debugging lebih mudah

B. Essay

1. Jelaskan proses konversi dari AST ke three-address code dengan contoh kompleks!

2. Desain dan implementasikan three-address code generator untuk bahasa dengan arrays dan function calls!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat merancang three-address code representation

☐ Saya dapat mengkonversi ekspresi aritmatika ke three-address code

☐ Saya dapat menerjemahkan control flow structures

☐ Saya dapat mengimplementasikan temporary variable management

☐ Saya dapat menangani arrays dan pointers

☐ Saya dapat mengimplementasikan function calls

Rangkuman

Bab ini membahas three-address code generation, termasuk format instruksi, translation dari AST, implementasi data structures, dan penanganan konstruksi kompleks. Mahasiswa belajar membangun intermediate code generator.

Poin Kunci:

- Three-address code adalah IR dengan maksimal 3 operands per instruksi
- Translation dari AST ke TAC memerlukan temporary variables
- Control flow structures memerlukan label dan jump instructions
- TAC memudahkan optimasi dan code generation
- Arrays dan pointers memerlukan address arithmetic

Kata Kunci: ***Three-Address Code***, ***Intermediate Representation***, ***TAC***, ***Temporary Variables***, ***AST Translation***, ***Control Flow***

Bab 8

Basic Block Identification dan Local Optimization

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 4.2:** Mengimplementasikan basic block identification

8.1 Dasar Basic Block dan Karakteristiknya

Basic Block adalah sekumpulan instruksi kode antara (*intermediate code*) yang dieksekusi secara berurutan tanpa adanya instruksi lompatan (*jump/branch*) di tengah-tengahnya.

8.1.1 Definisi Formal

Sebuah blok instruksi disebut *Basic Block* jika:

- Hanya memiliki satu titik masuk (*entry point*), yaitu instruksi pertama.
- Hanya memiliki satu titik keluar (*exit point*), yaitu instruksi terakhir.
- Semua instruksi di dalamnya dieksekusi secara sekuensial.

8.1.2 Contoh Basic Block

Perhatikan potongan kode berikut:

```
t1 = a + b
t2 = t1 * c
x = t2
```

Setelah eksekusi dimulai dari baris pertama, semua baris berikutnya dijamin akan dieksekusi secara berurutan. Ini adalah satu *Basic Block*.

8.2 Algoritma Identifikasi Leader

Untuk membagi kode antara menjadi beberapa *basic block*, kita harus mengidentifikasi instruksi-instruksi yang menjadi "pemimpin" (*leader*).

8.2.1 Aturan Identifikasi Leader

Instruksi adalah *leader* jika memenuhi salah satu syarat berikut:

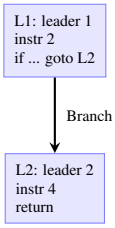
1. Instruksi pertama dalam program/fungsi.

2. Instruksi yang merupakan target dari sebuah lompatan (*conditional jump* atau *unconditional jump*).

3. Instruksi yang muncul tepat setelah instruksi lompatan.

8.2.2 Pembentukan Blok

Setelah semua *leader* ditemukan, setiap *basic block* terdiri dari sebuah *leader* dan semua instruksi berikutnya hingga (tapi tidak termasuk) instruksi *leader* berikutnya.



Gambar 8.1: Identifikasi leader dan pembentukan blok

8.3 Local Optimization: Constant Folding

Constant Folding adalah teknik optimasi yang mengevaluasi ekspresi dengan operan konstanta pada waktu kompilasi (*compile-time*) alih-alih pada waktu eksekusi (*run-time*).

8.3.1 Mekanisme Kerja

Jika kedua operan dalam operasi biner adalah literal (misal: `3 + 5`), kompilator langsung menggantinya dengan hasilnya (`8`).

8.3.2 Keuntungan

Teknik ini mengurangi jumlah instruksi yang dieksekusi oleh prosesor, mempercepat program, dan seringkali membuka peluang optimasi lebih lanjut (seperti eliminasi variabel yang tidak terpakai).

```
// Sebelum optimasi
t1 = 10 * 2
t2 = a + t1

// Sesudah Constant Folding
t1 = 20
t2 = a + 20
```

8.4 Local Optimization: Constant Propagation

Constant Propagation bekerja sama dengan *constant folding* dengan menyebarkan nilai konstanta ke penggunaan variabel berikutnya di dalam blok yang sama.

8.4.1 Prinsip Kerja

Jika sebuah variabel diberikan nilai konstanta (misal: `x = 10`), setiap penggunaan `x` di instruksi berikutnya (sebelum `x` diubah kembali) diganti langsung dengan nilai `10`.

8.4.2 Contoh Kasus

```
x = 5
y = x + 2
z = y * 10
```

Akan dioptimasi secara bertahap menjadi:

1. `y = 5 + 2` (Propagation)
2. `y = 7` (Folding)
3. `z = 7 * 10` (Propagation)
4. `z = 70` (Folding)

8.5 Control Flow Graph (CFG) Construction

Control Flow Graph (CFG) adalah representasi berarah di mana setiap *node* adalah sebuah *basic block* dan setiap *edge* menunjukkan kemungkinan aliran kendali antar blok tersebut.

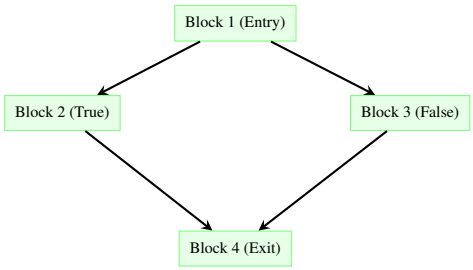
8.5.1 Membangun Tepi (Edges)

Sebuah tepi ditambahkan dari blok B_1 ke blok B_2 jika:

- Terdapat instruksi lompatan (jump) dari akhir B_1 ke awal B_2.
- B_2 terletak tepat setelah B_1 dalam urutan program, dan B_1 tidak diakhiri dengan lompatan mutlak (*unconditional jump*).

8.5.2 Pentingnya CFG

CFG adalah struktur data utama untuk melakukan *Global Optimization* dan analisis aliran data (*Data Flow Analysis*) yang mencakup seluruh fungsi/program.



Gambar 8.2: Contoh Control Flow Graph sederhana

Aktivitas Pembelajaran

1. **Block Identification**: Implementasikan algoritma basic block identification.
2. **CFG Construction**: Bangun control flow graph dari three-address code.
3. **Local Optimizations**: Implementasikan constant folding dan algebraic simplification.
4. **Data Flow**: Implementasikan reaching definitions analysis.
5. **Optimization Pipeline**: Buat pipeline untuk multiple local optimizations.

Latihan dan Refleksi

1. Identifikasi basic blocks dalam potongan three-address code yang diberikan!

2. Gambarkan CFG untuk program dengan nested if-else dan while loops!

3. Implementasikan constant folding untuk ekspresi aritmatika kompleks!

4. Analisis reaching definitions untuk program sederhana!

5. Identifikasi dead code yang dapat dieliminasi!

6. **Refleksi**: Bagaimana basic blocks menyederhanakan optimasi compiler?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 4.2

A. Pilihan Ganda

1. Basic block memiliki:
   (a) Multiple entry points
   (b) Single entry point
   (c) Multiple exit points
   (d) No exit points

2. Leader adalah instruksi yang:
   (a) Pertama dalam program
   (b) Target dari jump
   (c) Setelah conditional jump
   (d) Semua jawaban benar

3. Constant folding adalah:
   (a) Menghapus konstanta
   (b) Menggabungkan konstanta
   (c) Mengidentifikasi konstanta
   (d) Mengoptimasi loops

B. Essay

1. Jelaskan algoritma basic block identification dengan contoh konkret!

2. Implementasikan local optimization pipeline dengan constant folding, copy propagation, dan dead code elimination!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat mengimplementasikan basic block identification

☐ Saya dapat membangun control flow graph

☐ Saya dapat melakukan constant folding

☐ Saya dapat menerapkan algebraic simplification

☐ Saya dapat mengimplementasikan copy propagation

☐ Saya dapat melakukan dead code elimination

Rangkuman

Bab ini membahas basic block identification dan local optimization, termasuk algoritma identifikasi, CFG construction, dan berbagai teknik optimasi lokal. Mahasiswa belajar membangun fondasi untuk global optimizations.

Poin Kunci:

- Basic blocks adalah unit analisis untuk optimasi compiler
- Leaders menandai awal dari setiap basic block
- CFG merepresentasikan alur kontrol antar basic blocks
- Local optimizations bekerja dalam satu basic block
- Data flow analysis menyediakan informasi untuk optimasi

Kata Kunci: ***Basic Block***, ***Control Flow Graph***, ***Local Optimization***, ***Constant Folding***, ***Copy Propagation***, ***Dead Code Elimination***, ***Data Flow Analysis***

Bab 9

Local Optimization dan Data Flow Analysis

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 4.3:** Menganalisis dan mengoptimasi kode tingkat lokal

9.1 Pendahuluan Data-Flow Analysis

Data-Flow Analysis adalah teknik untuk mengumpulkan informasi tentang bagaimana nilai-nilai (*definitions*, *uses*) mengalir melalui Control Flow Graph (CFG) sebuah program.

9.1.1 Komponen Analisis

- **Transfer Function**: Aturan yang menjelaskan bagaimana instruksi dalam sebuah blok mengubah informasi aliran data.
- **Meet Operation**: Aturan untuk menggabungkan informasi dari beberapa jalur yang bertemu di satu *node*.

9.1.2 Arah Analisis

Analisis dapat dilakukan secara **Forward** (dari entry ke exit, misal: *Reaching Definitions*) atau **Backward** (dari exit ke entry, misal: *Live Variable Analysis*).

9.2 Reaching Definitions Analysis

Analisis ***Reaching Definitions*** menentukan definisi variabel mana yang mungkin "mencapai" titik tertentu dalam kode tanpa diubah oleh definisi lain di tengah jalan.

9.2.1 Aplikasi

Informasi ini sangat krusial untuk:

- *Constant Propagation* skala global.
- Mendeteksi penggunaan variabel yang mungkin belum diinisialisasi.

9.2.2 Persamaan Aliran Data

Untuk setiap blok B:

$$OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$$

Di mana GEN adalah definisi baru di dalam blok dan $KILL$ adalah definisi lain dari variabel yang sama di blok lain.

9.3 Live Variable Analysis dan DCE

Analisis ***Live Variable*** menentukan apakah sebuah variabel masih "hidup" (*live*), artinya nilainya mungkin akan dibaca lagi di masa depan sebelum program berakhir atau variabel tersebut di-assign ulang.

9.3.1 Analisis Mundur (Backward Analysis)

Analisis ini dimulai dari titik keluar program dan merambat mundur ke atas. Variabel di-mark sebagai *live* jika ia dibaca dalam instruksi saat ini.

9.3.2 Dead Code Elimination (DCE)

Jika hasil dari sebuah operasi (*assignment*) disimpan ke variabel yang sudah tidak lagi *live* (mati), maka instruksi tersebut dapat dihapus dengan aman karena tidak mempengaruhi luaran program.

```
x = 10;      // x live?
y = 20;      // y live?
print(x);    // x dibaca, y mati
// y = 20 dapat dihapus!
```

9.4 Available Expressions dan CSE

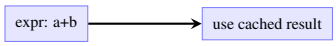
Ekspresi $x + y$ disebut ***Available*** di titik p jika sudah pernah dihitung sebelumnya dan nilai x maupun y belum berubah sejak penghitungan tersebut.

9.4.1 Common Subexpression Elimination (CSE)

Jika sebuah ekspresi sudah *available*, kompilator akan mengganti perhitungan ulang ekspresi tersebut dengan nilai yang sudah ada di variabel perantara.

```
// Sebelum CSE
t1 = a + b
t2 = c * d
t3 = a + b  // Perhitungan berulang

// Sesudah CSE
t1 = a + b
t2 = c * d
t3 = t1
```



Gambar 9.1: Konsep eliminasi sub-ekspresi umum

9.5 Algebraic Simplification dan Strength Reduction

9.5.1 Penyederhanaan Aljabar

Menggunakan identitas matematika untuk menyederhanakan kode:

- $x + 0 \Rightarrow x$
- $x \times 1 \Rightarrow x$
- $x \times 0 \Rightarrow 0$

9.5.2 Strength Reduction

Mengganti operasi yang "mahal" (membutuhkan banyak siklus CPU) dengan operasi yang lebih "murah":

- $x \times 2 \Rightarrow x + x$ atau $x \ll 1$
- $x^2 \Rightarrow x \times x$
- $x/4 \Rightarrow x \gg 2$

Teknik ini sangat efektif terutama di dalam loop yang dieksekusi jutaan kali.

9.6 Pipeline dan Interaksi Optimasi

Optimasi kompilator tidak berjalan sekali saja, melainkan dalam beberapa lintasan (*passes*). Satu optimasi seringkali menjadi pemicu untuk optimasi lainnya.

9.6.1 Optimization Loop

Karena satu perubahan sering memicu peluang baru, optimizer biasanya bekerja dalam sebuah loop iteratif:

```
while (terjadi perubahan):
    lakukan constant folding
    lakukan copy propagation
    lakukan dead code elimination
```

9.6.2 Trade-off

Optimasi yang lebih agresif membutuhkan waktu kompilasi yang lebih lama. Kompilator modern menyediakan level optimasi (seperti `-O1`, `-O2`, `-O3` pada GCC) untuk memberikan fleksibilitas bagi pemrogram.

Aktivitas Pembelajaran

1. **Algebraic Optimization**: Implementasikan berbagai algebraic optimizations.
2. **Constant Propagation**: Bangun constant propagation analyzer.
3. **Copy Propagation**: Implementasikan copy propagation algorithm.
4. **Dead Code Elimination**: Identifikasi dan hapus dead code.
5. **CSE**: Implementasikan common subexpression elimination.

Latihan dan Refleksi

1. Identifikasi semua algebraic optimizations yang mungkin dalam potongan kode!
2. Implementasikan constant propagation untuk program dengan multiple assignments!
3. Analisis copy propagation chains dan handling conflicts!

4. Identifikasi dead code dalam program dengan complex control flow!

5. Implementasikan CSE untuk expressions dengan multiple operands!

6. **Refleksi**: Bagaimana local optimizations mempengaruhi performance compiler?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 4.3

A. Pilihan Ganda

1. Strength reduction mengganti:
   (a) Operasi mahal dengan operasi murah
   (b) Konstanta dengan variabel
   (c) Variabel dengan konstanta
   (d) Dead code dengan live code

2. Constant propagation:
   (a) Menyebarkan nilai konstanta
   (b) Menghapus konstanta
   (c) Mengidentifikasi konstanta
   (d) Mengoptimasi loops

3. Dead code elimination menghapus:
   (a) Instruksi yang tidak digunakan
   (b) Instruksi yang lambat
   (c) Instruksi yang error
   (d) Semua instruksi

B. Essay

1. Jelaskan implementasi complete local optimization pipeline dengan semua teknik yang dibahas!

2. Analisis trade-off antara berbagai local optimizations dalam terms of performance vs complexity!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat menganalisis dan mengoptimasi kode tingkat lokal

☐ Saya dapat mengimplementasikan algebraic optimizations

☐ Saya dapat melakukan constant propagation

☐ Saya dapat menerapkan copy propagation

☐ Saya dapat mengidentifikasi dan menghapus dead code

☐ Saya dapat melakukan common subexpression elimination

Rangkuman

Bab ini membahas local optimization dan data flow analysis, termasuk algebraic optimizations, constant propagation, copy propagation, dead code elimination, dan common subexpression elimination. Mahasiswa belajar membangun optimization pipeline yang efektif.

Poin Kunci:

- Local optimizations bekerja dalam satu basic block
- Algebraic optimizations menyederhanakan ekspresi matematis
- Constant propagation menyebarkan nilai konstanta
- Copy propagation menghilangkan variabel perantara
- Dead code elimination menghapus instruksi yang tidak berguna
- CSE menghilangkan perhitungan berulang

Kata Kunci: ***Local Optimization***, ***Algebraic Optimization***, ***Constant Propagation***, ***Copy Propagation***, ***Dead Code Elimination***, ***Common Subexpression Elimination***, ***Data Flow Analysis***

Bab 10

Runtime Environment dan Memory Management

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 5.1:** Mendesain runtime environment untuk bahasa pemrograman

10.1 Layout Memori dan Segmentasi

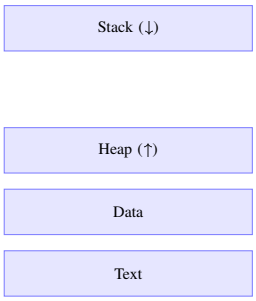
Runtime Environment mengelola bagaimana program menggunakan memori saat dieksekusi. Manajemen memori saat runtime merupakan salah satu aspek yang paling krusial dalam performa dan keamanan sebuah program hasil kompilasi [15].

10.1.1 Segmen Memori Utama

- **Text/Code Segment**: Menyimpan instruksi biner hasil kompilasi. Biasanya bersifat *read-only*.
- **Data Segment**: Menyimpan variabel global dan statis yang sudah diinisialisasi.
- **Stack**: Segmen yang tumbuh ke bawah untuk mengelola pemanggilan fungsi dan variabel lokal.
- **Heap**: Segmen yang tumbuh ke atas untuk alokasi memori dinamis (*runtime allocation*).

10.2 Activation Records dan Stack Frame

Setiap kali sebuah fungsi dipanggil, struktur data yang disebut ***Activation Record*** (atau *Stack Frame*) dibuat di puncak stack.



Gambar 10.1: Model konseptual organisasi memori runtime

10.2.1 Isi Activation Record

Sebuah stack frame biasanya menampung informasi kritis:

1. **Return Address**: Alamat instruksi berikutnya di penyeru (*caller*) setelah fungsi selesai.
2. **Control Link**: Referensi ke stack frame milik penyeru.
3. **Saved Registers**: Nilai register prosesor yang harus dipulihkan nantinya.
4. **Local Variables**: Ruang untuk variabel yang dideklarasikan di dalam fungsi.

10.2.2 Manajemen Frame Pointer

Frame Pointer (FP) digunakan sebagai referensi tetap untuk mengakses parameter (offset positif) dan variabel lokal (offset negatif) di dalam frame saat ini.

10.3 Mekanisme Panggilan: Prologue dan Epilogue

10.3.1 Prologue

Saat fungsi dipanggil, sekumpulan instruksi *prologue* dijalankan untuk menyiapkan lingkungan baru:

```
push rbp          ; Simpan frame pointer lama
mov rbp, rsp      ; Set frame pointer baru
sub rsp, 16       ; Alokasi ruang untuk lokal
```

10.3.2 Epilogue

Sebelum kembali ke penyeru, instruksi *epilogue* dijalankan untuk membersihkan stack:

```
mov rsp, rbp      ; Dealokasi ruang lokal
pop rbp           ; Pulihkan frame pointer lama
ret               ; Kembali ke return address
```

10.4 Manajemen Dinamis dan Heap

Berbeda dengan stack yang dikelola secara otomatis (LIFO), ***Heap*** memungkinkan alokasi memori yang waktu hidupnya (*lifetime*) tidak terikat pada durasi pemanggilan fungsi.

10.4.1 Alokasi dan Dealokasi

Dalam bahasa C++, ini dikelola melalui operator `new` dan `delete`. Kompilator bertanggung jawab menyisipkan panggilan ke sistem operasi atau pengelola memori (*allocator*).

10.4.2 Masalah Manajemen Heap

- **Fragmentasi**: Tersebarnya blok-blok memori kosong yang kecil sehingga tidak bisa menampung alokasi besar.
- **Memory Leak**: Kegagalan membebaskan memori yang sudah tidak digunakan, menyebabkan konsumsi RAM terus meningkat.

10.5 Praktikum: Simulator Runtime Stack

Mahasiswa akan mempelajari cara kerja stack frame melalui implementasi simulator sederhana. Simulator ini melacak pembuatan dan penghapusan *Activation Record* setiap kali ada simulasi pemanggilan fungsi.

```
struct Frame {
    string funcName;
    Frame* caller;
    map<string, int> locals;
};

class StackSimulator {
    Frame* top = nullptr;
public:
    void call(string name) {
        top = new Frame{name, top};
    }
    void returnFunc() {
        Frame* old = top;
        top = top->caller;
        delete old;
    }
};
```

Simulator ini membantu memvisualisasikan bagaimana variabel lokal diisolasi antar fungsi dan bagaimana rekursi dapat menyebabkan *Stack Overflow* jika tidak terkendali.

10.6 Garbage Collection

10.6.1 Reference Counting

Simple garbage collection:

```
typedef struct Object {
    int ref_count;
    void *data;
    struct Object **references;
    int ref_count_size;
} GCObject;

void gc_retain(GCObject *obj) {
    if (obj) obj->ref_count++;
}

void gc_release(GCObject *obj) {
    if (obj && --obj->ref_count == 0) {
        // Release all references
        for (int i = 0; i < obj->ref_count_size; i++) {
            gc_release(obj->references[i]);
        }
        free(obj);
    }
}
```

10.6.2 Mark and Sweep

```
void mark_and_sweep(HeapManager *heap) {
    // Mark phase
    mark_phase(heap);

    // Sweep phase
    sweep_phase(heap);
}

void mark_phase(HeapManager *heap) {
    // Mark all reachable objects from roots
    Object **roots = get_gc_roots();
    for (int i = 0; i < num_roots; i++) {
        mark_object(roots[i]);
    }
}

void sweep_phase(HeapManager *heap) {
    MemoryBlock *block = heap->allocated_list;
    while (block) {
        if (!is_marked(block->object)) {
            // Free unreachable object
            free_object(block->object);
            remove_from_allocated(block);
            add_to_free(block);
```

```
25          } else {
26              // Clear mark for next GC
27              clear_mark(block->object);
28          }
29          block = block->next;
30      }
31  }
```

Aktivitas Pembelajaran

1. **Memory Layout**: Implementasikan simulator memory layout program.
2. **Activation Records**: Bangun activation record management system.
3. **Parameter Passing**: Implementasikan berbagai parameter passing methods.
4. **Heap Management**: Implementasikan heap allocator dengan first-fit algorithm.
5. **Garbage Collection**: Bangun simple mark-and-sweep garbage collector.

Latihan dan Refleksi

1. Gambarkan memory layout untuk program dengan recursive functions!
2. Implementasikan activation record untuk function dengan nested scopes!
3. Analisis perbedaan pass by value, reference, dan value-result!
4. Implementasikan heap manager dengan fragmentation handling!
5. Desain garbage collection algorithm untuk object-oriented language!
6. **Refleksi**: Bagaimana runtime environment mempengaruhi performance program?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 5.1

A. Pilihan Ganda

1. Activation record disimpan di:

   (a) Heap
   (b) Stack
   (c) Static area
   (d) Code segment

2. Pass by reference mengirimkan:
   (a) Nilai variabel
   (b) Alamat variabel
   (c) Copy variabel
   (d) Pointer ke pointer

3. Garbage collection menghapus:
   (a) Semua objek
   (b) Objek yang tidak reachable
   (c) Objek yang besar
   (d) Objek yang lama

B. Essay

1. Jelaskan prosedur call mechanism lengkap dengan activation record management!
2. Desain runtime environment untuk bahasa dengan support untuk dynamic arrays dan garbage collection!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat mendesain runtime environment untuk bahasa pemrograman

☐ Saya dapat mengimplementasikan memory layout yang efisien

☐ Saya dapat membangun activation record management

☐ Saya dapat mengimplementasikan berbagai parameter passing methods

☐ Saya dapat mendesain heap management system

☐ Saya dapat mengimplementasikan garbage collection

Rangkuman

Bab ini membahas runtime environment dan memory management, termasuk memory layout, activation records, parameter passing, heap management, dan garbage collection. Mahasiswa belajar mendesain infrastruktur runtime yang efisien.

Poin Kunci:

- Runtime environment menyediakan infrastruktur untuk eksekusi program
- Memory layout terdiri dari stack, heap, static area, dan code segment
- Activation records mengelola function calls dan local variables
- Parameter passing methods memiliki trade-off berbeda
- Heap management mengelola dynamic memory allocation
- Garbage collection otomatis menghapus objek yang tidak digunakan

Kata Kunci: ***Runtime Environment***, ***Memory Layout***, ***Activation Record***, ***Parameter Passing***, ***Heap Management***, ***Garbage Collection***, ***Stack Frame***

Bab 11

Memory Layout dan Addressing Modes

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 5.2:** Mengimplementasikan addressing modes untuk variabel dan arrays

11.1 Metode Pengalamatan (Addressing Modes)

Addressing Modes menentukan bagaimana operan diambil dari memori atau register pada tingkat bahasa mesin. Intermediate Code Generation bertindak sebagai jembatan antara front-end yang dependen pada bahasa sumber dan back-end yang dependen pada mesin target [16].

11.1.1 Jenis Pengalamatan Umum

- **Register Addressing**: Operan berada langsung di register (`ADD R1, R2`).
- **Immediate Addressing**: Operan adalah konstanta yang tertanam dalam instruksi (`ADDI R1, R1, 10`).
- **Displacement/Indexed**: Mengakses memori dengan alamat *base register* ditambah *offset* (`LW R1, 8(R2)`). Sangat berguna untuk *stack frame* dan akses *struct*.

11.1.2 Kalkulasi Alamat Array

Akses array `A[i]` diterjemahkan menjadi alamat:

$$\text{Alamat} = \text{Base}(A) + (i \times \text{ukuran\_elemen})$$

Kompilator harus menghasilkan instruksi perkalian dan penambahan untuk menghitung offset ini di setiap akses array.

11.2 Addressing Modes

11.2.1 Direct Addressing

- **Absolute**: Alamat memori langsung
- **Register**: Variabel dalam register
- **Immediate**: Konstanta dalam instruksi

```
// Direct addressing examples
int x = 10;           // x di memory dengan alamat spesifik
register int y = 20;  // y di register
#define Z 30          // Z sebagai immediate value
```

11.2.2 Indirect Addressing

- **Register indirect**: Alamat di register
- **Memory indirect**: Alamat di memory
- **Indexed**: Base + index

```
// Indirect addressing examples
int *ptr = &x;        // ptr menyimpan alamat x
int value = *ptr;     // Indirect melalui ptr
int arr[10];
int elem = arr[i];    // Indexed addressing
```

11.3 Array Addressing

11.3.1 One-Dimensional Arrays

Layout memory untuk 1D array:

```
int arr[5] = {10, 20, 30, 40, 50};

// Memory layout (each int = 4 bytes)
// arr[0] at base_address + 0*4
// arr[1] at base_address + 1*4
// arr[2] at base_address + 2*4
// arr[3] at base_address + 3*4
// arr[4] at base_address + 4*4

// Address calculation
int address = base_address + index * sizeof(int);
```

11.3.2 Multi-Dimensional Arrays

Row-major order untuk 2D arrays:

```c
int matrix[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};

// Address calculation: base + (row * cols + col) * sizeof(int)
int element_address = base_address +
                      (row * 4 + col) * sizeof(int);

// matrix[1][2] = base + (1*4 + 2)*4 = base + 6*4
```

11.3.3 Array Implementation

```c
typedef struct {
    void *base_address;
    int element_size;
    int num_dimensions;
    int *dimensions;
    int *bounds;  // lower bounds
} ArrayDescriptor;

int calculate_address(ArrayDescriptor *arr, int *indices) {
    int offset = 0;
    int stride = 1;

    // Calculate offset from last dimension to first
    for (int i = arr->num_dimensions - 1; i >= 0; i--) {
        offset += (indices[i] - arr->bounds[i]) * stride;
        stride *= arr->dimensions[i];
    }

    return (int)arr->base_address + offset * arr->element_size;
}
```

11.4 Structure and Union Layout

11.4.1 Structure Memory Layout

```c
struct Student {
    char name[50];      // 50 bytes
    int age;            // 4 bytes (aligned to 4)
    double gpa;         // 8 bytes (aligned to 8)
    char grade;         // 1 byte
    // Padding: 3 bytes for alignment
};

```

```
// Total size: 50 + 4 + 4(padding) + 8 + 1 + 3(padding) = 70 bytes
```

11.4.2 Union Memory Layout

```
union Data {
    int i;              // 4 bytes
    float f;            // 4 bytes
    char c[4];          // 4 bytes
    double d;           // 8 bytes (largest member)
};

// Union size = size of largest member = 8 bytes
```

11.5 Pointer Arithmetic

11.5.1 Pointer Operations

```
int arr[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
int *ptr = arr;

// Pointer arithmetic
ptr++;        // ptr += sizeof(int) (moves to next element)
ptr += 3;     // ptr += 3 * sizeof(int)
ptr--;        // ptr -= sizeof(int)

// Array access via pointers
int value = *(ptr + 2);  // Same as ptr[2]
int diff = ptr - arr;    // Number of elements between pointers
```

11.5.2 Dynamic Arrays

```
typedef struct {
    int *data;
    int size;
    int capacity;
} DynamicArray;

DynamicArray* create_array(int initial_capacity) {
    DynamicArray *arr = malloc(sizeof(DynamicArray));
    arr->data = malloc(initial_capacity * sizeof(int));
    arr->size = 0;
    arr->capacity = initial_capacity;
    return arr;
}

void resize_array(DynamicArray *arr, int new_capacity) {
    arr->data = realloc(arr->data, new_capacity * sizeof(int));
    arr->capacity = new_capacity;
}
```

11.6 Address Translation

11.6.1 Virtual to Physical Address

```c
typedef struct {
    unsigned int page_number : 10;
    unsigned int offset : 12;
} VirtualAddress;

typedef struct {
    unsigned int frame_number : 10;
    unsigned int valid : 1;
    unsigned int protection : 2;
} PageTableEntry;

unsigned int virtual_to_physical(VirtualAddress va,
                                 PageTableEntry *page_table) {
    PageTableEntry entry = page_table[va.page_number];
    if (entry.valid) {
        return (entry.frame_number << 12) | va.offset;
    }
    return 0;  // Page fault
}
```

11.6.2 Segmentation

```c
typedef struct {
    unsigned int base;
    unsigned int limit;
    int protection;
} SegmentDescriptor;

unsigned int segmented_address(unsigned int logical_addr,
                               SegmentDescriptor *segment) {
    if (logical_addr <= segment->limit) {
        return segment->base + logical_addr;
    }
    return 0;  // Segmentation fault
}
```

Aktivitas Pembelajaran

1. **Memory Layout**: Implementasikan simulator memory layout untuk berbagai tipe data.

2. **Array Addressing**: Bangun array descriptor untuk multi-dimensional arrays.

3. **Pointer Arithmetic**: Implementasikan pointer arithmetic operations.

4. **Structure Layout**: Analisis memory layout untuk structures dengan padding.

5. **Address Translation**: Implementasikan simple virtual memory translation.

Latihan dan Refleksi

1. Hitung memory layout untuk struktur dengan nested structures!

2. Implementasikan address calculation untuk 3D array dengan arbitrary bounds!

3. Analisis overhead dari different addressing modes!

4. Desain memory layout untuk object-oriented language dengan inheritance!

5. Implementasikan garbage collection-aware memory management!

6. **Refleksi**: Bagaimana memory layout mempengaruhi performance program?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 5.2

A. Pilihan Ganda

1. Row-major order untuk 2D array menghitung:
   (a) row * cols + col
   (b) col * rows + row
   (c) row + col * cols
   (d) col + row * rows

2. Structure padding digunakan untuk:
   (a) Mengurangi memory usage
   (b) Alignment optimization
   (c) Error detection
   (d) Security

3. Pointer arithmetic pada int pointer menambah:

(a) 1 byte

(b) 2 bytes

(c) 4 bytes

(d) 8 bytes

B. Essay

1. Jelaskan implementasi complete addressing modes untuk bahasa dengan arrays, structures, dan pointers!
2. Desain memory layout system yang efisien untuk dynamic data structures!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat mengimplementasikan addressing modes untuk variabel dan arrays

☐ Saya dapat menghitung memory layout untuk structures dan unions

☐ Saya dapat melakukan pointer arithmetic operations

☐ Saya dapat mengimplementasikan array descriptor systems

☐ Saya memahami virtual memory address translation

☐ Saya dapat mendesain efficient memory layouts

Rangkuman

Bab ini membahas memory layout dan addressing modes, termasuk storage classes, addressing modes, array implementation, structure layout, pointer arithmetic, dan address translation. Mahasiswa belajar mengimplementasikan efficient memory management.

Poin Kunci:

- Memory layout menentukan bagaimana data disimpan dan diakses
- Addressing modes menyediakan berbagai cara mengakses data
- Array addressing memerlukan perhitungan offset yang tepat
- Structure layout mempertimbangkan alignment dan padding

- Pointer arithmetic memungkinkan efficient data access
- Virtual memory translation memisahkan logical dan physical addresses

Kata Kunci: ***Memory Layout***, ***Addressing Modes***, ***Array Addressing***, ***Structure Layout***, ***Pointer Arithmetic***, ***Virtual Memory***, ***Memory Alignment***

Bab 12

Code Generation dan Target Machine

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 5.3:** Mengimplementasikan code generator untuk arsitektur target

12.1 Pengenalan Target Machine (RISC vs CISC)

12.1.1 Arsitektur RISC (Reduced Instruction Set Computer)

Contoh: RISC-V, ARM.

- Instruksi berukuran tetap (biasanya 32-bit).
- Hanya instruksi `Load` dan `Store` yang bisa mengakses memori.
- Memiliki banyak register umum (*general purpose*).

12.1.2 Arsitektur CISC (Complex Instruction Set Computer)

Contoh: x86 (Intel/AMD).

- Instruksi berukuran variabel (1-15 byte).
- Instruksi aritmatika bisa langsung mengakses operan di memori.
- Jumlah register fisik lebih terbatas dibanding RISC.

12.1.3 Dampak pada Code Generation

Code generator harus menyadari arsitektur target. Pada RISC, operasi aritmatika yang kompleks harus diurai menjadi beberapa instruksi dasar `load-load-op-store`.

12.2 Pemilihan Instruksi (Instruction Selection)

Instruction Selection adalah proses memetakan instruksi tingkat menengah (*TAC*) ke instruksi spesifik mesin target yang memberikan performa terbaik.

12.2.1 Pattern Matching

Kompilator modern sering menggunakan pencocokan pola pohon (*Tree Pattern Matching*) untuk memilih instruksi kompleks. Misal: `a = b * c + d` dapat diganti dengan satu instruksi *Multiply-Add* jika CPU mendukungnya.

12.2.2 Biaya Instruksi (Cost Estimation)

Setiap opsi instruksi memiliki bobot (siklus CPU). *Instruction selector* mencoba meminimalkan total bobot instruksi yang dihasilkan untuk sebuah blok kode.

12.3 Register Allocation

12.3.1 Register Allocation Problem

Masalah alokasi register:

- Variabel terbatas vs unlimited temporaries
- Register interference
- Spilling ke memory
- Calling convention constraints

12.3.2 Linear Scan Allocation

```
typedef struct {
    char *var_name;
    int start_point;     // First use
    int end_point;       // Last use
    int register_num;    // Assigned register (-1 if spilled)
} LiveRange;

void linear_scan_allocation(LiveRange *ranges, int count,
                            int num_registers) {
    // Sort ranges by start point
    sort_ranges_by_start(ranges, count);

    bool *registers_used = calloc(num_registers, sizeof(bool));

```

```
    for (int i = 0; i < count; i++) {
        // Free registers whose ranges have ended
        free_expired_registers(ranges[i].start_point,
                               registers_used, num_registers);

        // Find free register
        int reg = find_free_register(registers_used, num_registers);
        if (reg != -1) {
            ranges[i].register_num = reg;
            registers_used[reg] = true;
        } else {
            // Spill to memory
            ranges[i].register_num = -1;
        }
    }
}
```

12.4 Target Architecture

12.4.1 x86 Architecture

Karakteristik x86:

- CISC (Complex Instruction Set Computer)
- Variable-length instructions
- Rich addressing modes
- Backward compatibility

```
// x86 instruction examples
MOV EAX, EBX          ; Register to register
MOV EAX, [EBX+4]      ; Base + offset addressing
MOV EAX, [EBX+ECX*4] ; Base + index*scale
LEA EAX, [EBX+ECX*2] ; Load effective address
PUSH EAX              ; Stack operation
POP EBX               ; Stack operation
```

12.4.2 RISC Architecture

Karakteristik RISC (MIPS/ARM):

- Fixed-length instructions
- Load/store architecture
- Simple addressing modes

- Large register file

```
// MIPS instruction examples
ADD $t0, $t1, $t2    ; $t0 = $t1 + $t2
LW $t0, 4($t1)       ; Load word from memory
SW $t0, 8($t1)       ; Store word to memory
ADDI $t0, $t1, 10    ; $t0 = $t1 + 10 (immediate)
```

12.5 Code Generation Algorithm

12.5.1 Basic Block Code Generation

```
void generate_block_code(BasicBlock *block,
                         RegisterAllocator *alloc) {
    for (int i = 0; i < block->instruction_count; i++) {
        TACInstruction *inst = &block->instructions[i];

        // Allocate registers for operands
        int reg1 = allocate_register(inst->arg1, alloc);
        int reg2 = allocate_register(inst->arg2, alloc);
        int reg3 = allocate_register(inst->result, alloc);

        // Generate target instruction
        switch (inst->op) {
            case OP_ADD:
                emit_add(reg3, reg1, reg2);
                break;
            case OP_MUL:
                emit_mul(reg3, reg1, reg2);
                break;
            case OP_ASSIGN:
                emit_mov(reg3, reg1);
                break;
            // ... other operations
        }

        // Release temporary registers
        release_register(reg1, alloc);
        release_register(reg2, alloc);
    }
}
```

12.5.2 Function Call Generation

```
void generate_function_call(FunctionCall *call) {
    // Save caller-saved registers
    save_caller_saved_registers();

    // Push parameters (right-to-left for cdecl)
    for (int i = call->param_count - 1; i >= 0; i--) {
        push_parameter(call->parameters[i]);
```

```
    }

    // Call function
    emit_call(call->function_name);

    // Clean up stack (callee or caller depending on convention)
    cleanup_stack(call->param_count * sizeof(int));

    // Restore caller-saved registers
    restore_caller_saved_registers();

    // Move return value to target register
    if (call->has_return_value) {
        emit_mov(call->return_reg, EAX);
    }
}
```

12.6 Optimization in Code Generation

12.6.1 Peephole Optimization

```
void peephole_optimization(Instruction *instructions, int count) {
    for (int i = 0; i < count - 1; i++) {
        // MOV reg, reg -> NOP
        if (is_mov_reg_to_reg(&instructions[i])) {
            instructions[i].opcode = NOP;
        }

        // PUSH reg; POP reg -> NOP
        if (is_push_pop_same_reg(&instructions[i], &instructions[i+1])) {
            instructions[i].opcode = NOP;
            instructions[i+1].opcode = NOP;
        }

        // MOV reg, imm; ADD reg, imm -> LEA reg, [imm]
        if (can_convert_to_lea(&instructions[i], &instructions[i+1])) {
            convert_to_lea(&instructions[i], &instructions[i+1]);
        }
    }
}
```

12.6.2 Instruction Scheduling

```
void schedule_instructions(Instruction *instructions, int count) {
    // Simple list scheduling for basic blocks
    Instruction *scheduled[count];
    int scheduled_count = 0;

    while (scheduled_count < count) {
        // Find ready instructions (no dependencies)
        for (int i = 0; i < count; i++) {
```

```
            if (!is_scheduled(&instructions[i]) &&
                is_ready(&instructions[i], scheduled, scheduled_count)) {
                scheduled[scheduled_count++] = instructions[i];
                break;
            }
        }
    }

    // Copy scheduled instructions back
    memcpy(instructions, scheduled, count * sizeof(Instruction));
}
```

Aktivitas Pembelajaran

1. **Instruction Selection**: Implementasikan instruction selector untuk subset x86.
2. **Register Allocation**: Bangun linear scan register allocator.
3. **Code Generation**: Implementasikan code generator untuk simple expressions.
4. **Peephole Optimization**: Buat peephole optimizer untuk assembly code.
5. **Function Calls**: Generate code untuk function calls dengan calling conventions.

Latihan dan Refleksi

1. Generate assembly code untuk expression tree kompleks!
2. Implementasikan register allocator dengan spilling strategy!
3. Analisis instruction selection for different target architectures!
4. Optimasi generated code dengan peephole optimizations!
5. Generate code untuk recursive functions dengan proper stack management!
6. **Refleksi**: Bagaimana target architecture mempengaruhi code generation strategy?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 5.3

A. Pilihan Ganda

1. Register allocation problem terjadi karena:
   (a) Terlalu banyak variabel
   (b) Terbatasnya jumlah register
   (c) Memory terlalu kecil
   (d) Instruksi terlalu kompleks
2. CISC architecture memiliki:
   (a) Fixed-length instructions
   (b) Variable-length instructions
   (c) Load/store only
   (d) Large register file
3. Peephole optimization bekerja pada:
   (a) Single instruction
   (b) Small window of instructions
   (c) Entire program
   (d) Basic blocks

B. Essay

1. Jelaskan complete code generation pipeline dari three-address code ke assembly!
2. Implementasikan code generator untuk bahasa sederhana dengan arithmetic expressions dan function calls!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat mengimplementasikan code generator untuk arsitektur target

☐ Saya dapat melakukan instruction selection yang efisien

☐ Saya dapat mengimplementasikan register allocation algorithms

☐ Saya dapat mengenerate code untuk function calls

☐ Saya dapat melakukan peephole optimizations

☐ Saya memahami perbedaan CISC dan RISC architectures

Rangkuman

Bab ini membahas code generation dan target machine, termasuk instruction selection, register allocation, target architectures, dan optimization techniques. Mahasiswa belajar membangun code generator yang efisien.

Poin Kunci:

- Code generation mengkonversi intermediate code ke target code
- Instruction selection memilih optimal target instructions
- Register allocation mengelola limited register resources
- Target architecture mempengaruhi generation strategy
- Peephole optimization mengoptimasi local instruction patterns
- Function calls memerlukan proper calling convention handling

Kata Kunci: ***Code Generation***, ***Instruction Selection***, ***Register Allocation***, ***Target Architecture***, ***x86***, ***RISC***, ***Peephole Optimization***, ***Calling Convention***

Bab 13

Register Allocation dan Optimization

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 5.3:** Mengoptimasi penggunaan register dan meminimalkan memory access

13.1 Alokasi Register: Strategi dan Kompleksitas

Karena jumlah register fisik dalam CPU sangat terbatas, kompilator harus memutuskan variabel mana yang layak menempati register dan mana yang harus dipindahkan (*spilled*) ke RAM.

13.1.1 Graph Coloring

Masalah alokasi register dapat dimodelkan sebagai pewarnaan graf (*Graph Coloring*).

- Node: Variabel yang aktif (*live variables*).
- Edge: Saling berpotongan (*interfere*), artinya dua variabel hidup di waktu yang sama.
- Warna: Jumlah register fisik yang tersedia.

13.1.2 Linear Scan

Untuk kompilasi yang cepat (*JIT compilers*), algoritma *Linear Scan* digunakan karena lebih sederhana dibanding *graph coloring* namun tetap memberikan hasil yang memadai.

13.2 Interference Graph

13.2.1 Graph Coloring

Register allocation sebagai graph coloring:

- **Nodes**: Variabel/temporaries
- **Edges**: Interference (variables live simultaneously)
- **Colors**: Register assignments
- **Spilling**: Variables yang tidak dapat diwarnai

13.2.2 Interference Graph Construction

```
typedef struct {
    int var_id;
    char *var_name;
    int start_point;
    int end_point;
} LiveRange;

typedef struct {
    int num_vars;
    bool **adjacency;  // Interference matrix
    LiveRange *ranges;
} InterferenceGraph;

InterferenceGraph* build_interference_graph(BasicBlock *block) {
    // Calculate live ranges
    LiveRange *ranges = calculate_live_ranges(block);

    // Build interference graph
    InterferenceGraph *graph = create_graph(num_vars);

    for (int i = 0; i < num_vars; i++) {
        for (int j = i + 1; j < num_vars; j++) {
            if (ranges_interfere(ranges[i], ranges[j])) {
                add_interference_edge(graph, i, j);
            }
        }
    }

    return graph;
}
```

13.3 Graph Coloring Algorithm

13.3.1 Simplified Graph Coloring

```c
typedef struct {
    int var_id;
    int color;         // -1 = spilled, 0+ = register number
    int degree;        // Number of neighbors
    bool removed;
} GraphNode;

int graph_coloring(InterferenceGraph *graph, int num_registers) {
    GraphNode *nodes = initialize_nodes(graph);
    int spilled_count = 0;

    // Simplify phase
    while (has_uncolored_nodes(nodes)) {
        // Find node with degree < num_registers
        int node = find_low_degree_node(nodes, num_registers);

        if (node != -1) {
            // Remove from graph
            remove_node(nodes, node);
        } else {
            // Spill a node
            int spill_node = select_spill_node(nodes);
            nodes[spill_node].color = -1;  // Mark as spilled
            remove_node(nodes, spill_node);
            spilled_count++;
        }
    }

    // Select phase - assign colors
    assign_colors(nodes, num_registers);

    return spilled_count;
}
```

13.4 Linear Scan Allocation

13.4.1 Linear Scan Algorithm

Alokasi register yang lebih efisien:

```c
typedef struct {
    int var_id;
    int start;
    int end;
    int reg;           // -1 if not allocated
    bool active;
} Interval;

void linear_scan(Interval *intervals, int count, int num_registers) {
    // Sort intervals by start point
    sort_intervals_by_start(intervals, count);

```

```
    Interval *active[num_registers];
    int active_count = 0;

    for (int i = 0; i < count; i++) {
        // Expire old intervals
        expire_old_intervals(intervals[i].start, active, &active_count);

        if (active_count < num_registers) {
            // Allocate register
            intervals[i].reg = find_free_register(active, active_count);
            add_to_active(&intervals[i], active, &active_count);
        } else {
            // Spill
            int spill_index = select_spill_candidate(active, active_count
↪ );
            spill_interval(active[spill_index]);
            intervals[i].reg = active[spill_index]->reg;
            active[spill_index] = &intervals[i];
        }
    }
}
```

13.5 Spilling Strategies

13.5.1 Spill Cost Analysis

Menentukan variabel yang akan di-spill:

```
typedef struct {
    int var_id;
    float spill_cost;  // Lower = better to spill
    int memory_accesses;
    int loop_depth;
    int use_count;
} SpillCandidate;

float calculate_spill_cost(SpillCandidate *candidate) {
    // Consider multiple factors
    float cost = 0.0;

    // More memory accesses = higher cost (don't want to spill)
    cost += candidate->memory_accesses * 10.0;

    // Deeper in loops = higher cost
    cost += candidate->loop_depth * 5.0;

    // More uses = higher cost
    cost += candidate->use_count * 2.0;

    // Normalize by live range length
    int range_length = candidate->end - candidate->start;
    return cost / range_length;
}
```

```

\subsection{Spill Code Generation}

\begin{lstlisting}[language=C]
void generate_spill_code(Instruction *instructions, int *count,
                         SpillCandidate *spilled_vars, int num_spilled) {
    for (int i = 0; i < num_spilled; i++) {
        int var_id = spilled_vars[i].var_id;

        // Insert load before each use
        for (int j = 0; j < *count; j++) {
            if (uses_variable(&instructions[j], var_id)) {
                insert_load_before(&instructions[j], var_id, count);
                j++;  // Skip inserted instruction
            }
        }

        // Insert store after each definition
        for (int j = 0; j < *count; j++) {
            if (defines_variable(&instructions[j], var_id)) {
                insert_store_after(&instructions[j], var_id, count);
                j++;  // Skip inserted instruction
            }
        }
    }
}
```

13.6 Coalescing

13.6.1 Copy Coalescing

Menggabungkan variabel yang di-copy:

```
// Before coalescing:
t1 = x
y = t1
z = t1

// After coalescing (t1 eliminated):
y = x
z = x

bool can_coalesce(InterferenceGraph *graph, int var1, int var2) {
    // Check if variables interfere
    if (have_interference(graph, var1, var2)) {
        return false;
    }

    // Check if coalescing would create too high degree
    int combined_degree = calculate_combined_degree(graph, var1, var2);
    if (combined_degree >= num_registers) {
        return false;
    }
```

```
21
22      return true;
23  }
24
25  void coalesce_variables(InterferenceGraph *graph, int var1, int var2) {
26      // Merge var2 into var1
27      merge_nodes(graph, var1, var2);
28      update_interference_edges(graph, var1);
29  }
```

13.7 Advanced Optimizations

13.7.1 Register Renaming

Menghilangkan false dependencies:

```
1  // Before renaming:
2  t1 = a + b
3  t2 = t1 * c
4  t1 = d + e    // False dependency on previous t1
5  t3 = t1 * f
6
7  // After renaming:
8  t1 = a + b
9  t2 = t1 * c
10 t3 = d + e    // No false dependency
11 t4 = t3 * f
12
13 void rename_registers(BasicBlock *block) {
14     int next_temp = 0;
15     int register_map[MAX_VARIABLES];
16
17     for (int i = 0; i < block->instruction_count; i++) {
18         Instruction *inst = &block->instructions[i];
19
20         // Rename destination
21         if (inst->result) {
22             int new_reg = next_temp++;
23             register_map[inst->result] = new_reg;
24             inst->result = new_reg;
25         }
26
27         // Update source operands
28         if (inst->arg1 && register_map[inst->arg1]) {
29             inst->arg1 = register_map[inst->arg1];
30         }
31         if (inst->arg2 && register_map[inst->arg2]) {
32             inst->arg2 = register_map[inst->arg2];
33         }
34     }
35 }
36
37 \subsection{Loop Optimization}
```

```

\begin{lstlisting}[language=C]
void optimize_loop_registers(Loop *loop) {
    // Allocate loop invariants to registers
    identify_loop_invariants(loop);

    // Keep frequently used loop variables in registers
    analyze_loop_variable_usage(loop);

    // Minimize register pressure in loop body
    reduce_loop_register_pressure(loop);

    // Prefer registers for induction variables
    allocate_induction_variables(loop);
}
```

Aktivitas Pembelajaran

1. **Interference Graph**: Implementasikan interference graph construction.
2. **Graph Coloring**: Bangun graph coloring register allocator.
3. **Linear Scan**: Implementasikan linear scan allocation algorithm.
4. **Spilling**: Desain spill cost analysis dan spill code generation.
5. **Coalescing**: Implementasikan copy coalescing optimization.

Latihan dan Refleksi

1. Bangun interference graph untuk potongan kode dengan multiple variables!
2. Implementasikan graph coloring dengan backtracking untuk optimal solution!
3. Analisis spill cost untuk berbagai variabel dalam nested loops!
4. Implementasikan linear scan dengan heuristic improvements!
5. Optimasi register allocation untuk loop-intensive code!
6. **Refleksi**: Bagaimana register allocation mempengaruhi performance generated code?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 5.3

A. Pilihan Ganda

1. Interference graph edge menunjukkan:
   (a) Variables yang sama
   (b) Variables yang hidup bersamaan
   (c) Variables yang di-copy
   (d) Variables yang di-spill
2. Linear scan allocation memiliki complexity:
   (a) O(n)
   (b) O(n log n)
   (c) $O(n^2)$
   (d) $O(n^3)$
3. Spilling dilakukan ketika:
   (a) Register penuh
   (b) Memory penuh
   (c) Graph tidak bisa diwarnai
   (d) Variabel tidak digunakan

B. Essay

1. Jelaskan complete register allocation pipeline dengan interference graph and graph coloring!
2. Implementasikan register allocator dengan linear scan algorithm and spill handling!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat mengoptimasi penggunaan register dan meminimalkan memory access

☐ Saya dapat membangun interference graph untuk register allocation

☐ Saya dapat mengimplementasikan graph coloring algorithm

☐ Saya dapat melakukan linear scan register allocation

☐ Saya dapat mengimplementasikan spill strategies

☐ Saya dapat melakukan register coalescing dan renaming

Rangkuman

Bab ini membahas register allocation dan optimization, termasuk interference graph, graph coloring, linear scan allocation, spilling strategies, dan advanced optimizations. Mahasiswa belajar mengoptimasi penggunaan register untuk performance maksimal.

Poin Kunci:

- Register allocation memetakan variabel unlimited ke register terbatas
- Interference graph modeling conflicts antar variabel
- Graph coloring adalah NP-complete problem
- Linear scan memberikan heuristic yang efisien
- Spilling menangani kasus ketika register tidak cukup
- Coalescing dan renaming mengoptimasi register usage

Kata Kunci: ***Register Allocation***, ***Interference Graph***, ***Graph Coloring***, ***Linear Scan***, ***Spilling***, ***Coalescing***, ***Register Renaming***

Bab 14

Activation Records dan Stack Management

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 5.3:** Mengimplementasikan activation records untuk procedure calls

14.1 Pengenalan Activation Records

14.1.1 Definisi Activation Record

Activation Record (stack frame) adalah struktur data yang menyimpan:

- Local variables
- Parameters
- Return address
- Saved registers
- Dynamic link (pointer ke caller frame)
- Static link (pointer ke enclosing scope)

14.1.2 Stack Frame Layout

```
// Typical stack frame layout (grows downward)
+--------------------+  // Higher addresses
| Parameters         |  (from caller)
+--------------------+
| Return Address     |  (pushed by CALL)
+--------------------+
```

```
| Dynamic Link       |  (old frame pointer)
+--------------------+
| Saved Registers    |  (caller-saved)
+--------------------+
| Local Variables    |  (allocated by callee)
+--------------------+
| Temporaries        |  (temporary storage)
+--------------------+  // Lower addresses
```

14.2 Procedure Call Mechanism

14.2.1 Calling Convention

Prosedur call untuk function `func(a, b, c)`:

```
// Caller side:
push c              // Push parameters (right-to-left)
push b
push a
call func           // Push return address and jump

// Callee side (func entry):
push ebp            // Save old frame pointer
mov ebp, esp         // Set new frame pointer
sub esp, local_size // Allocate space for locals

// Function body here...

// Callee side (func exit):
mov esp, ebp         // Deallocate locals
pop ebp             // Restore old frame pointer
ret                 // Pop return address and jump
```

14.2.2 Parameter Passing

Method	Pros	Cons
Stack	Simple, unlimited parameters	Slow, memory access
Register	Fast, no memory access	Limited registers
Mixed	Balance of speed	Complex

Tabel 14.1: Parameter Passing Methods

14.3 Stack Management

14.3.1 Stack Operations

```c
typedef struct {
    void *base;         // Stack base
    void *top;          // Stack top
    size_t size;        // Current size
    size_t capacity;    // Maximum size
} Stack;

void push(Stack *stack, void *data, size_t data_size) {
    if (stack->size + data_size > stack->capacity) {
        stack_overflow_error();
    }

    memcpy(stack->top, data, data_size);
    stack->top = (char*)stack->top + data_size;
    stack->size += data_size;
}

void* pop(Stack *stack, size_t data_size) {
    if (stack->size < data_size) {
        stack_underflow_error();
    }

    stack->top = (char*)stack->top - data_size;
    stack->size -= data_size;
    return stack->top;
}
```

14.3.2 Frame Pointer vs Stack Pointer

```c
// Using frame pointer (EBP/RBP)
int access_local(int offset) {
    // Access local at [EBP - offset]
    return *(int*)((char*)EBP - offset);
}

// Using stack pointer only (ESP/RSP)
int access_local_no_fp(int offset_from_sp) {
    // Access local at [ESP + offset]
    return *(int*)((char*)ESP + offset_from_sp);
}
```

14.4 Nested Functions

14.4.1 Static Links

Untuk nested functions (Pascal-style):

```c
typedef struct {
    void *return_addr;
    void *dynamic_link;  // Pointer to caller frame
    void *static_link;   // Pointer to enclosing function frame
```

```
    LocalVars locals;
    Temporaries temps;
} ActivationRecord;

void nested_function() {
    // Access variable from enclosing function
    ActivationRecord *current = get_current_frame();
    ActivationRecord *enclosing = current->static_link;

    int outer_var = enclosing->locals.some_variable;
}
```

14.4.2 Display Method

Alternative untuk nested functions:

```
typedef struct {
    ActivationRecord *frames[MAX_NESTING];
    int current_level;
} Display;

void access_nested_var(int level, int offset) {
    ActivationRecord *frame = display.frames[level];
    return *(int*)((char*)frame + offset);
}
```

14.5 Exception Handling

14.5.1 Stack Unwinding

```
typedef struct {
    void *handler;
    void *frame_pointer;
    int handler_type;
} ExceptionHandler;

void throw_exception(int exception_type) {
    // Unwind stack looking for handler
    ActivationRecord *current = get_current_frame();

    while (current != NULL) {
        ExceptionHandler *handler = find_handler(current, exception_type)
↪ ;
        if (handler) {
            // Jump to handler
            longjmp(handler->jump_buffer, exception_type);
        }
        current = current->dynamic_link;
    }

    // No handler found - terminate
```

```
    terminate_program();
}
```

14.6 Optimization Techniques

14.6.1 Leaf Function Optimization

```
// Before optimization
int leaf_function(int x, int y) {
    return x + y;
}

// After optimization (no frame setup)
int leaf_function_optimized(int x, int y) {
    // Use registers only, no stack operations
    return x + y;
}
```

14.6.2 Tail Call Optimization

```
// Before optimization
int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);  // Need to preserve frame
}

// After tail call optimization
int factorial_tail(int n, int acc) {
    if (n <= 1) return acc;
    return factorial_tail(n - 1, n * acc);  // Can reuse frame
}
```

14.6.3 Register Saving Optimization

```
// Save only registers that are actually used
void save_used_registers(RegisterMask used_regs) {
    for (int i = 0; i < NUM_REGISTERS; i++) {
        if (used_regs & (1 << i)) {
            push_register(i);
        }
    }
}

void restore_used_registers(RegisterMask used_regs) {
    for (int i = NUM_REGISTERS - 1; i >= 0; i--) {
        if (used_regs & (1 << i)) {
            pop_register(i);
        }
    }
```

```
}
```

14.7 Variable Length Arrays

14.7.1 VLA on Stack

```
void function_with_vla(int n) {
    int vla[n];  // Variable length array

    // Stack layout after VLA allocation:
    // +--------------------+
    // | VLA (n * sizeof(int)) |
    // +--------------------+
    // | Other locals       |
    // +--------------------+
    // | Frame pointer      |
    // +--------------------+

    // Access VLA
    for (int i = 0; i < n; i++) {
        vla[i] = i * 2;
    }
}
```

14.7.2 Alloca Implementation

```
void* alloca(size_t size) {
    void *result = (char*)current_frame_pointer - size;

    // Check for stack overflow
    if (result < stack_limit) {
        stack_overflow_error();
    }

    // Adjust stack pointer
    current_stack_pointer = result;
    return result;
}
```

Aktivitas Pembelajaran

1. **Stack Frame**: Implementasikan stack frame management system.

2. **Calling Convention**: Bangun procedure call mechanism dengan berbagai conventions.

3. **Nested Functions**: Implementasikan static links untuk nested functions.

4. **Exception Handling**: Buat exception handling dengan stack unwinding.

5. **Optimization**: Implementasikan leaf function dan tail call optimizations.

Latihan dan Refleksi

1. Gambarkan stack frame layout untuk function dengan multiple parameters dan locals!

2. Implementasikan calling convention untuk variadic functions!

3. Analisis overhead dari frame pointer vs stack pointer only!

4. Implementasikan exception handling dengan proper cleanup!

5. Optimasi function calls dengan tail recursion elimination!

6. **Refleksi**: Bagaimana activation records mempengaruhi performance function calls?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 5.3

A. Pilihan Ganda

1. Dynamic link menunjuk ke:

   (a) Enclosing function frame

   (b) Caller frame

   (c) Global frame

   (d) Stack base

2. Tail call optimization menghilangkan:

   (a) Parameter passing

   (b) Frame setup overhead

   (c) Return value

   (d) Function call

3. Static link digunakan untuk:

(a) Exception handling

(b) Nested functions

(c) Recursion

(d) Optimization

B. Essay

1. Jelaskan complete procedure call mechanism dengan activation records!
2. Implementasikan stack management system dengan support untuk nested functions dan exception handling!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat mengimplementasikan activation records untuk procedure calls

☐ Saya dapat mengelola stack operations dan frame management

☐ Saya dapat mengimplementasikan berbagai calling conventions

☐ Saya dapat menangani nested functions dengan static links

☐ Saya dapat mengimplementasikan exception handling

☐ Saya dapat melakukan stack-related optimizations

Rangkuman

Bab ini membahas activation records dan stack management, termasuk stack frame layout, procedure call mechanisms, nested functions, exception handling, and optimization techniques. Mahasiswa belajar mengimplementasikan efficient function call infrastructure.

Poin Kunci:

- Activation records menyimpan state untuk function execution
- Stack management mengelola dynamic allocation/deallocation
- Calling conventions menentukan parameter passing dan register saving
- Static links enable access to enclosing function variables

- Exception handling requires proper stack unwinding
- Optimizations reduce overhead of function calls

Kata Kunci: ***Activation Record***, ***Stack Frame***, ***Calling Convention***, ***Dynamic Link***, ***Static Link***, ***Tail Call Optimization***, ***Exception Handling***

Bab 15

Compiler Tools Analysis

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 6.1:** Menganalisis dan menggunakan compiler tools modern

15.1 Pengenalan Compiler Tools

15.1.1 Kategori Compiler Tools

Compiler Tools adalah software yang membantu pembuatan compiler. Ekosistem pengembangan kompilator modern melibatkan berbagai alat dan standar industri yang canggih untuk memastikan kualitas dan efisiensi kode yang dihasilkan [17].

- **Lexer Generators**: Flex, re2c
- **Parser Generators**: Bison, ANTLR
- **Optimization Frameworks**: LLVM, GCC
- **Build Systems**: Make, CMake
- **Debugging Tools**: GDB, Valgrind

15.1.2 Ekosistem Compiler Development

15.2 Lexer Generators

15.2.1 Flex (Fast Lexical Analyzer)

Flex adalah lexer generator yang populer:

Tool	Tipe	Platform
Flex	Lexer Generator	Cross-platform
Bison	Parser Generator	Cross-platform
ANTLR	Parser Generator	Java-based
LLVM	Compiler Framework	Cross-platform
GCC	Compiler Suite	Cross-platform

Tabel 15.1: Popular Compiler Tools

```
/* Example Flex specification */
%{
#include "y.tab.h"
%}

digit        [0-9]
letter       [a-zA-Z]
identifier   {letter}({letter}|{digit})*

%%

{digit}+         { yylval.ival = atoi(yytext); return NUMBER; }
{identifier}     { yylval.sval = strdup(yytext); return IDENTIFIER; }
"+"              { return PLUS; }
"*"              { return TIMES; }
[ \t\n]          { /* skip whitespace */ }
.                { fprintf(stderr, "Unknown character: %s\n", yytext); }

%%

int yywrap(void) {
    return 1;
}
```

15.2.2 re2c

re2c adalah alternatif yang lebih sederhana:

```
/* re2c specification */
/*!re2c
re2c:define:YYCTYPE        unsigned char
re2c:define:YYCURSOR       cursor
re2c:define:YYLIMIT         limit
re2c:define:YYMARKER        marker
re2c:define:YYFILL(n)      { memset(cursor, 0, n); cursor += n; }

NUMBER = [0-9]+;
IDENTIFIER = [a-zA-Z][a-zA-Z0-9]*;
*/

void scan(char *cursor, char *limit) {
    char *marker = cursor;

```

```
    /*!re2c
    NUMBER { return NUMBER; }
    IDENTIFIER { return IDENTIFIER; }
    "+" { return PLUS; }
    "*" { return TIMES; }
    [ \t\r\n]+ { continue; }
    . { return UNKNOWN; }
    */
}
```

15.3 Parser Generators

15.3.1 Bison (GNU Yacc)

Bison adalah LALR(1) parser generator:

```
/* Bison specification */
%{
#include <stdio.h>
#include <stdlib.h>
int yylex(void);
void yyerror(const char *s);
%}

%token NUMBER IDENTIFIER PLUS TIMES

%left PLUS
%left TIMES

%%
input:
    /* empty */
  | input line
  ;

line:
    expr '\n'          { printf("= %d\n", $1); }
  | '\n'               { /* empty line */ }
  ;

expr:
    expr PLUS expr     { $$ = $1 + $3; }
  | expr TIMES expr    { $$ = $1 * $3; }
  | NUMBER             { $$ = $1; }
  | IDENTIFIER         { $$ = symbol_lookup($1); }
  ;
%%

void yyerror(const char *s) {
    fprintf(stderr, "Error: %s\n", s);
}
```

15.3.2 ANTLR

ANTLR adalah parser generator modern:

```
// ANTLR grammar for arithmetic expressions
grammar Expr;

prog: stat+ ;

stat: expr NEWLINE
    | ID '=' expr NEWLINE { symbols.put($ID.text, $expr.v); }
    ;

expr: expr op=('*'|'/') expr      { $v = new BinOp($op.text, $expr.v,
↪ $expr2.v); }
    | expr op=('+'|'-') expr      { $v = new BinOp($op.text, $expr.v,
↪ $expr2.v); }
    | INT                         { $v = new Int($INT.text); }
    | ID                          { $v = symbols.get($ID.text); }
    | '(' expr ')'                { $v = $expr.v; }
    ;
```

15.4 Compiler Frameworks

15.4.1 LLVM (Low Level Virtual Machine)

LLVM adalah compiler framework modern:

```
// LLVM IR example
define i32 @add(i32 %a, i32 %b) {
entry:
  %sum = add i32 %a, %b
  ret i32 %sum
}

// Using LLVM C++ API
#include "llvm/IR/IRBuilder.h"

llvm::Value* createAdd(llvm::IRBuilder<> &builder,
                       llvm::Value *a, llvm::Value *b) {
    return builder.CreateAdd(a, b, "sum");
}
```

15.4.2 GCC (GNU Compiler Collection)

GCC adalah compiler suite lengkap:

- Frontends: C, C++, Objective-C, Fortran, Ada, Go
- Backends: x86, ARM, MIPS, PowerPC, RISC-V
- Middle-end: GIMPLE, RTL, Tree SSA

- Optimizations: -O0, -O1, -O2, -O3, -Os

15.5 Build Systems

15.5.1 Make

Make adalah build system tradisional:

```
# Makefile for simple compiler
CC = gcc
CFLAGS = -Wall -g

compiler: lexer.o parser.o main.o
  $(CC) $(CFLAGS) -o $@ $^

lexer.o: lexer.l
  flex lexer.l
  $(CC) $(CFLAGS) -c lex.yy.c -o $@

parser.o: parser.y
  bison -d parser.y
  $(CC) $(CFLAGS) -c parser.tab.c -o $@

clean:
  rm -f *.o compiler lex.yy.c parser.tab.c parser.tab.h
```

15.5.2 CMake

CMake adalah build system modern:

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.10)
project(MyCompiler)

set(CMAKE_C_STANDARD 11)

# Find required packages
find_package(FLEX REQUIRED)
find_package(BISON REQUIRED)

# Generate lexer and parser
flex_target(lexer lexer.l)
bison_target(parser parser.y parser.tab.h parser.tab.c)

# Create executable
add_executable(mycompiler
    main.c
    ${FLEX_lexer_OUTPUTS}
    ${BISON_parser_OUTPUTS}
)
```

15.6 Debugging Tools

15.6.1 GDB (GNU Debugger)

GDB untuk debugging compiler:

```
# Debugging compiler with GDB
gdb ./mycompiler
(gdb) break main
(gdb) run input.c
(gdb) step
(gdb) print token
(gdb) backtrace
(gdb) quit
```

15.6.2 Valgrind

Valgrind untuk memory debugging:

```
# Check for memory leaks
valgrind --leak-check=full ./mycompiler input.c

# Check for memory errors
valgrind --tool=memcheck ./mycompiler input.c

# Profile memory usage
valgrind --tool=massif ./mycompiler input.c
```

15.7 Performance Analysis

15.7.1 Profiling Tools

Tool	Fungsi
gprof	Function profiling
perf	System-wide profiling
valgrind/callgrind	Call graph profiling
time	Execution time measurement

Tabel 15.2: Profiling Tools

15.7.2 Benchmarking

```
// Simple benchmark framework
#include <time.h>
#include <stdio.h>

void benchmark_compiler(char *input_file) {
    clock_t start = clock();
```

```
7
8       // Run compiler
9       compile_file(input_file);
10
11      clock_t end = clock();
12      double elapsed = ((double)(end - start)) / CLOCKS_PER_SEC;
13
14      printf("Compilation time: %.3f seconds\n", elapsed);
15  }
```

Aktivitas Pembelajaran

1. **Flex/Bison**: Buat simple calculator menggunakan Flex dan Bison.
2. **ANTLR**: Implementasikan parser untuk bahasa ekspresi dengan ANTLR.
3. **LLVM**: Buat LLVM pass untuk optimasi sederhana.
4. **Build Systems**: Konversi Makefile ke CMake.
5. **Debugging**: Debug compiler crash dengan GDB.

Latihan dan Refleksi

1. Bandingkan Flex dan re2c untuk lexer generation!
2. Implementasikan parser untuk bahasa dengan control structures menggunakan Bison!
3. Analisis LLVM IR untuk program sederhana!
4. Konversi build system dari Make ke CMake untuk project compiler!
5. Debug memory leak dalam parser generator!
6. **Refleksi**: Bagaimana compiler tools mempengaruhi produktivitas pengembangan compiler?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 6.1

A. Pilihan Ganda

1. Flex adalah:
   (a) Parser generator
   (b) Lexer generator
   (c) Build system
   (d) Debugger
2. Bison menggunakan parsing algorithm:
   (a) LL(1)
   (b) LR(1)
   (c) LALR(1)
   (d) Recursive descent
3. LLVM adalah:
   (a) Compiler framework
   (b) Build system
   (c) Debugger
   (d) Lexer generator

B. Essay

1. Jelaskan ekosistem compiler tools dan peran masing-masing tool!
2. Implementasikan mini compiler menggunakan Flex, Bison, dan build system modern!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat menganalisis dan menggunakan compiler tools modern

☐ Saya dapat menggunakan lexer generators (Flex, re2c)

☐ Saya dapat menggunakan parser generators (Bison, ANTLR)

☐ Saya memahami compiler frameworks (LLVM, GCC)

☐ Saya dapat menggunakan build systems (Make, CMake)

☐ Saya dapat menggunakan debugging tools untuk compiler development

Rangkuman

Bab ini membahas compiler tools analysis, termasuk lexer generators, parser generators, compiler frameworks, build systems, dan debugging tools. Mahasiswa belajar menggunakan tools modern untuk pengembangan compiler yang efisien.

Poin Kunci:

- Compiler tools menyederhanakan pengembangan compiler
- Lexer generators otomatisasi token recognition
- Parser generators menghasilkan efficient parsers
- Compiler frameworks menyediakan infrastructure lengkap
- Build systems mengelola compilation dependencies
- Debugging tools membantu identifikasi dan perbaikan bugs

Kata Kunci: ***Compiler Tools***, ***Flex***, ***Bison***, ***ANTLR***, ***LLVM***, ***GCC***, ***Make***, ***CMake***, ***GDB***, ***Valgrind***

Bab 16

Performance Evaluation dan Benchmarking

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 6.2:** Mengevaluasi kinerja compiler dan optimasi

16.1 Studi Kasus: Proyek Compiler Subset C

Sebagai bentuk evaluasi kinerja dan integrasi seluruh fase, kita akan meninjau spesifikasi proyek ***Subset C*** yang telah kita bangun secara bertahap.

16.1.1 Spesifikasi Grammar dan AST

Proyek ini mengimplementasikan grammar *top-down* untuk mengakomodasi *recursive descent parser* serta grammar *bottom-up* untuk *Bison*. Token yang didukung meliputi tipe data `int`/`float`, kontrol `if`/`while`, dan fungsi `print`.

16.1.2 Manajemen Runtime dan Memori

Pada fase awal, proyek ini menggunakan alokasi statis untuk variabel sederhana. Saat fungsi ditambahkan, proyek mengadopsi *activation record* standar x86-64 untuk memastikan kompatibilitas dengan *runtime C library* (`libc`).

16.1.3 Analisis Kinerja

Mahasiswa diharapkan melakukan *benchmarking* terhadap kode yang dihasilkan, membandingkan antara kode tanpa optimasi dengan kode yang telah melalui fase *constant folding* dan *dead code elimination*.

16.2 Benchmarking Methodology

16.2.1 Benchmark Design

Prinsip benchmark yang baik:

- **Reproducibility**: Hasil dapat direproduksi
- **Fairness**: Perbandingan yang adil
- **Representative**: Mewakili kasus nyata
- **Statistical significance**: Hasil statistik valid

16.2.2 Test Suite

```
// Benchmark test suite structure
typedef struct {
    char *name;
    char *description;
    char *input_file;
    int expected_time_ms;
    int expected_memory_kb;
} BenchmarkCase;

BenchmarkCase benchmarks[] = {
    {"small_file", "Small C file", "small.c", 100, 1024},
    {"medium_file", "Medium C file", "medium.c", 500, 4096},
    {"large_file", "Large C file", "large.c", 2000, 16384},
    {"complex_template", "Complex template", "template.cpp", 5000, 32768}
};
```

16.3 Measurement Tools

16.3.1 Time Measurement

```
#include <time.h>
#include <sys/time.h>

// High-resolution timer
double get_time_ms() {
    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    return ts.tv_sec * 1000.0 + ts.tv_nsec / 1000000.0;
}

// Measure compilation time
double measure_compilation_time(char *command) {
    double start = get_time_ms();

```

```
    int result = system(command);

    double end = get_time_ms();
    return end - start;
}
```

16.3.2 Memory Measurement

```
#include <sys/resource.h>
#include <unistd.h>

// Measure peak memory usage
size_t measure_peak_memory() {
    struct rusage usage;
    getrusage(RUSAGE_CHILDREN, &usage);
    return usage.ru_maxrss;  // Peak resident set size
}

// Measure current memory usage
size_t measure_current_memory() {
    FILE *status = fopen("/proc/self/status", "r");
    if (!status) return 0;

    char line[256];
    size_t memory = 0;

    while (fgets(line, sizeof(line), status)) {
        if (strncmp(line, "VmRSS:", 6) == 0) {
            sscanf(line + 7, "%zu", &memory);
            memory *= 1024;  // Convert KB to bytes
            break;
        }
    }

    fclose(status);
    return memory;
}
```

16.4 Compiler Comparison

16.4.1 Compiler Matrix

Compiler	Versi	Speed	Size	Memory
GCC	11.2	Fast	Medium	Medium
Clang	14.0	Fastest	Small	Low
MSVC	19.3	Medium	Large	High
ICC	2021.1	Fast	Medium	Medium

Tabel 16.1: Perbandingan Compiler Populer

16.4.2 Optimization Levels

```
# Benchmark different optimization levels
for level in O0 O1 O2 O3 Os; do
    echo "Testing with -$level optimization"
    time gcc -$level test.c -o test_$level
    ./test_$level
    echo "Size: $(stat -c%s test_$level)"
done
```

16.5 Performance Analysis

16.5.1 Profiling Results

```
# Profile with gprof
gcc -pg -O2 test.c -o test_profile
./test_profile
gprof test_profile gmon.out > profile_report.txt

# Profile with perf
perf stat -e cycles,instructions,cache-misses ./test_program

# Profile with valgrind
valgrind --tool=callgrind ./test_program
```

16.5.2 Statistical Analysis

```
import statistics
import numpy as np

def analyze_benchmark_results(times):
    """Analyze benchmark results statistically"""
    mean = np.mean(times)
    std_dev = np.std(times)
    median = np.median(times)

    # Remove outliers (beyond 2 standard deviations)
    filtered = [t for t in times if abs(t - mean) <= 2 * std_dev]

    filtered_mean = np.mean(filtered)
    filtered_std = np.std(filtered)

    return {
        'raw_mean': mean,
        'raw_std': std_dev,
        'filtered_mean': filtered_mean,
        'filtered_std': filtered_std,
        'median': median,
        'sample_size': len(times),
        'outliers_removed': len(times) - len(filtered)
    }
```

16.6 Optimization Impact

16.6.1 Optimization Techniques

Teknik	Speedup	Trade-off
Inline functions	1.2x - 2x	Code size increase
Loop unrolling	1.5x - 3x	Code size increase
Vectorization	4x - 8x	Limited to data parallel
Constant folding	1.1x - 1.3x	Compile time increase
Dead code elimination	1.05x - 1.2x	Compile time increase

Tabel 16.2: Impact Optimasi pada Performance

16.6.2 Case Studies

```
// Case study: Parser optimization
// Before optimization
int parse_slow(char *input) {
    // Multiple function calls
    token_t token1 = get_next_token(input);
    token_t token2 = get_next_token(token1->remaining);
    token_t token3 = get_next_token(token2->remaining);
    return process_tokens(token1, token2, token3);
}

// After optimization
int parse_fast(char *input) {
    // Inlined token processing
    token_t tokens[3];
    int count = 0;

    while (*input && count < 3) {
        tokens[count++] = get_next_token_inline(&input);
    }

    return process_tokens_inline(tokens[0], tokens[1], tokens[2]);
}
```

16.7 Benchmark Automation

16.7.1 Automated Testing

```
#!/usr/bin/env python3
import subprocess
import json
```

```python
import time

def run_benchmark(compiler, source_file, iterations=10):
    """Run automated benchmark"""
    results = []

    for i in range(iterations):
        start_time = time.time()

        # Compile
        compile_cmd = f"{compiler} -O2 {source_file} -o benchmark_test"
        subprocess.run(compile_cmd, shell=True, capture_output=True)

        # Execute
        start_exec = time.time()
        subprocess.run("./benchmark_test", shell=True, capture_output=
↪ True)
        end_exec = time.time()

        # Measure memory
        memory_usage = measure_memory_usage()

        results.append({
            'iteration': i,
            'compile_time': start_exec - start_time,
            'exec_time': end_exec - start_exec,
            'total_time': end_exec - start_time,
            'memory_kb': memory_usage
        })

    return results

def analyze_results(results):
    """Analyze benchmark results"""
    compile_times = [r['compile_time'] for r in results]
    exec_times = [r['exec_time'] for r in results]
    memory_usage = [r['memory_kb'] for r in results]

    return {
        'compile_stats': analyze_benchmark_results(compile_times),
        'exec_stats': analyze_benchmark_results(exec_times),
        'memory_stats': analyze_benchmark_results(memory_usage),
        'raw_data': results
    }
```

16.8 Real-World Performance

16.8.1 Industry Benchmarks

- **SPEC CPU2006**: CPU benchmark suite
- **SPEC CINT2006**: C integer benchmark

- **SPEC CFP2006**: C floating point benchmark
- **Compile-time benchmarks**: Compiler compilation speed

16.8.2 Cloud Compilation

```
# Cloud compilation benchmark
for cloud_provider in "aws" "gcp" "azure"; do
    echo "Testing $cloud_provider cloud compilation"

    # Measure compilation time
    start=$(date +%s%N)
    $cloud_provider compile build_project
    end=$(date +%s%N)

    echo "Compilation time: $((end - start)) seconds"

    # Measure cost
    cost=$($cloud_provider billing get-cost --project build_project)
    echo "Cost: $cost"
done
```

Aktivitas Pembelajaran

1. **Benchmark Setup**: Buat benchmark suite untuk compiler testing.
2. **Performance Analysis**: Analisis hasil benchmark dengan statistik.
3. **Optimization Testing**: Uji impact berbagai optimasi levels.
4. **Tool Comparison**: Bandingkan berbagai compiler tools.
5. **Automation**: Buat automated benchmarking pipeline.

Latihan dan Refleksi

1. Desain benchmark suite untuk compiler dengan multiple test cases!
2. Implementasikan high-resolution timer untuk akurasi pengukuran!
3. Analisis hasil benchmark dengan statistical methods!
4. Bandingkan performance GCC vs Clang untuk berbagai optimasi levels!

5. Identifikasi bottleneck dalam compilation pipeline!

6. **Refleksi**: Bagaimana performance evaluation mempengaruhi pengembangan compiler?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian untuk Sub-CPMK 6.2

A. Pilihan Ganda

1. Metrik yang TIDAK diukur dalam performance evaluation:
   (a) Code quality
   (b) Developer productivity
   (c) User satisfaction
   (d) Compilation speed
2. Tool untuk profiling memory usage:
   (a) GDB
   (b) Valgrind
   (c) /proc filesystem
   (d) System monitor
3. Optimasi yang memberikan speedup terbesar:
   (a) Loop unrolling
   (b) Vectorization
   (c) Inlining
   (d) Constant folding

B. Essay

1. Jelaskan metodologi benchmarking yang komprehensif untuk compiler evaluation!
2. Implementasikan automated benchmarking system dengan statistical analysis!

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya dapat mengevaluasi kinerja compiler dan optimasi

☐ Saya dapat merancang benchmark suite yang efektif

☐ Saya dapat menggunakan profiling tools untuk analisis

☐ Saya dapat melakukan statistical analysis pada hasil benchmark

☐ Saya dapat membandingkan performance berbagai compiler

☐ Saya dapat mengidentifikasi optimization opportunities

Rangkuman

Bab ini membahas performance evaluation dan benchmarking, termasuk metodologi, measurement tools, compiler comparison, optimization analysis, dan automated testing. Mahasiswa belajar mengevaluasi dan mengoptimasi kinerja compiler.

Poin Kunci:

- Performance evaluation mengukur efektivitas compiler
- Benchmarking menyediakan pengukuran yang sistematis
- Profiling tools membantu identifikasi bottleneck
- Statistical analysis memastikan validitas hasil
- Optimizations memiliki trade-off yang perlu dipertimbangkan
- Automation mempermudah testing berulang

Kata Kunci: ***Performance Evaluation***, ***Benchmarking***, ***Profiling***, ***Optimization***, ***Statistical Analysis***, ***Compiler Comparison***

Bab 17

Evaluasi dan Refleksi Kompetensi

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 1.1-6.2:** Mengevaluasi seluruh capaian pembelajaran mata kuliah Teknik Kompilasi

17.1 Latihan Mandiri Komprehensif

Bagian ini menyajikan kumpulan soal latihan untuk menguji pemahaman mahasiswa terhadap seluruh fase kompilasi yang telah dipelajari.

17.1.1 Analisis Leksikal dan Sintaksis

Identifikasi token dan gambarkan *parse tree* untuk ekspresi berikut: `x = (10 + y) * 2;`.

17.1.2 Semantik dan Tabel Simbol

Gambarkan hierarki tabel simbol untuk kode program yang memiliki fungsi di dalam lingkup global dan variabel lokal dengan nama yang sama (*shadowing*).

17.1.3 Generasi IR dan Optimasi

Tuliskan *Three-Address Code* (TAC) untuk struktur `while` loop dan lakukan *Constant Folding* jika terdapat ekspresi statis.

17.2 Evaluasi Capaian Pembelajaran Mata Kuliah

17.2.1 CPMK-1: Arsitektur Kompilator

Sub-CPMK	Teori	Praktik	Proyek	Nilai
1.1: Fase kompilasi	20%	15%	10%	45%
1.2: Struktur kompilator	20%	15%	10%	45%
1.3: Intermediate code	20%	15%	10%	45%

Tabel 17.1: Komponen Penilaian CPMK-1

17.2.2 CPMK-2: Lexer dan Parser

Sub-CPMK	Teori	Praktik	Proyek	Nilai
2.1: Regular expression	15%	25%	15%	55%
2.2: Finite automata	15%	25%	15%	55%
2.3: Parsing techniques	20%	20%	15%	55%
2.4: Parser generators	15%	25%	15%	55%

Tabel 17.2: Komponen Penilaian CPMK-2

17.2.3 CPMK-3: Semantic Analysis

Sub-CPMK	Teori	Praktik	Proyek	Nilai
3.1: Symbol table	15%	25%	15%	55%
3.2: Scope management	15%	25%	15%	55%
3.3: Type checking	20%	20%	15%	55%

Tabel 17.3: Komponen Penilaian CPMK-3

17.2.4 CPMK-4: Intermediate Code dan Optimasi

17.2.5 CPMK-5: Code Generation dan Runtime

17.2.6 CPMK-6: Tools dan Evaluasi

17.3 Refleksi Pembelajaran

17.3.1 Pertanyaan Reflektif

Bagian 1: Pemahaman Konsep

1. Konsep kompilator mana yang paling menant bagi Anda? Mengapa?

Sub-CPMK	Teori	Praktik	Proyek	Nilai
4.1: Three-address code	15%	25%	15%	55%
4.2: Basic blocks	15%	25%	15%	55%
4.3: Local optimization	20%	20%	15%	55%

Tabel 17.4: Komponen Penilaian CPMK-4

Sub-CPMK	Teori	Praktik	Proyek	Nilai
5.1: Runtime environment	15%	25%	15%	55%
5.2: Memory layout	15%	25%	15%	55%
5.3: Code generation	20%	20%	15%	55%

Tabel 17.5: Komponen Penilaian CPMK-5

2. Bagaimana pemahaman Anda tentang kompilator berkembang selama semester ini?

3. Konsep mana yang paling relevan dengan pengembangan software modern?

Bagian 2: Keterampilan Praktis

1. Implementasi mana yang memberikan insight terbesar tentang kompilasi?

2. Kesulitan teknis apa yang Anda hadapi dan bagaimana mengatasinya?

3. Keterampilan mana yang paling berharga untuk karir Anda?

Bagian 3: Pengembangan Diri

1. Bagaimana mata kuliah ini mengubah cara Anda berpikir tentang programming?

2. Topik mana yang ingin Anda pelajari lebih dalam?

3. Bagaimana Anda akan menerapkan pengetahuan ini di proyek masa depan?

17.3.2 Self-Assessment Checklist

17.4 Portofolio Proyek

17.4.1 Struktur Portofolio

Portofolio proyek kompilator harus mencakup:

1. **Deskripsi Proyek**: Tujuan, scope, dan requirements

2. **Arsitektur**: Desain komponen dan interaksi

3. **Implementasi**: Code snippets dan penjelasan

4. **Testing**: Test cases dan hasil

Sub-CPMK	Teori	Praktik	Proyek	Nilai
6.1: Compiler tools	15%	25%	15%	55%
6.2: Performance evaluation	15%	25%	15%	55%

Tabel 17.6: Komponen Penilaian CPMK-6

Kompetensi	Belum	Sedang	Menguasai
Memahami fase kompilasi	☐	☐	☐
Mengimplementasikan lexer	☐	☐	☐
Membangun parser	☐	☐	☐
Melakukan semantic analysis	☐	☐	☐
Menghasilkan intermediate code	☐	☐	☐
Melakukan optimasi	☐	☐	☐
Menghasilkan target code	☐	☐	☐
Menggunakan compiler tools	☐	☐	☐

Tabel 17.7: Self-Assessment Kompetensi

5. **Evaluasi**: Performance analysis dan benchmarking

6. **Refleksi**: Pembelajaran dan improvement

17.4.2 Kriteria Penilaian Portofolio

Aspek	Bobot
Koreksi dan kelengkapan	20%
Kualitas implementasi	25%
Pengujian dan validasi	20%
Dokumentasi	15%
Presentasi	10%
Refleksi dan pembelajaran	10%

Tabel 17.8: Kriteria Penilaian Portofolio

17.5 Feedback dan Continuous Improvement

17.5.1 Mekanisme Feedback

- **Formative Assessment**: Feedback selama pembelajaran
- **Peer Review**: Feedback dari teman sekelas
- **Self-Reflection**: Evaluasi diri
- **Instructor Feedback**: Feedback dari dosen

17.5.2 Action Plan for Improvement

1. **Identifikasi Gap**: Area yang perlu improvement
2. **Set Goals**: Target pencapaian yang spesifik
3. **Action Steps**: Langkah konkret untuk mencapai goals
4. **Timeline**: Jadwal implementasi
5. **Evaluation**: Cara mengukur progress

17.6 Kesimpulan dan Rekomendasi

17.6.1 Pencapaian Pembelajaran

Setelah menyelesaikan mata kuliah Teknik Kompilasi, mahasiswa diharapkan:

- Memahami prinsip-prinsip fundamental kompilasi
- Mampu mengimplementasikan compiler components
- Dapat menggunakan tools modern untuk pengembangan compiler
- Memiliki kemampuan analisis dan evaluasi performance
- Siap untuk pengembangan software yang lebih kompleks

17.6.2 Rekomendasi Lanjutan

- **Study Lanjut**: Advanced compiler design, optimization techniques
- **Aplikasi Praktis**: Bahasa programming, system programming
- **Research**: Compiler optimization, language design
- **Industry**: Compiler development, tool development

17.6.3 Final Reflection

Teknik Kompilasi adalah mata kuliah fundamental yang menghubungkan teori computer science dengan implementasi praktis. Pemahaman yang baik tentang kompilator memberikan fondasi kuat untuk pengembangan software yang efisien dan optimal.

"The best way to learn compiler construction is to build a compiler." - Andrew Appel

Aktivitas Pembelajaran

1. **Self-Assessment**: Isi checklist kompetensi mandiri.
2. **Portfolio Construction**: Susun portofolio proyek semester.
3. **Reflective Essay**: Tulis refleksi pembelajaran satu semester.
4. **Peer Review**: Evaluasi portofolio rekan sekelas.
5. **Action Planning**: Buat rencana pengembangan diri pasca mata kuliah.

Latihan dan Refleksi

1. Identifikasi Sub-CPMK yang paling sulit dicapai!
2. Susun bukti-bukti pencapaian kompetensi dalam portofolio!
3. Analisis kaitan antar modul dalam pengembangan compiler!
4. Evaluasi perkembangan keterampilan programming Anda!
5. Rencanakan langkah belajar selanjutnya di bidang software engineering!
6. **Refleksi**: Apa insight terbesar yang Anda dapatkan dari mata kuliah ini?

Asesmen (Evaluasi Kinerja)

Instrumen Penilaian Akhir Semester

A. Evaluasi Portofolio

1. Kelengkapan komponen compiler (Lexer, Parser, Semantic, dsb)
2. Kualitas dokumentasi dan analisis
3. Hasil pengujian dan benchmarking

B. Presentasi Proyek

1. Demonstrasi fungsionalitas compiler
2. Penjelasan arsitektur dan design decisions
3. Tanya jawab teknis

Rubrik Penilaian: Lihat Lampiran A dan B

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya memahami seluruh fase kompilasi dan interaksinya

☐ Saya dapat mengimplementasikan front-end dan back-end compiler

☐ Saya dapat melakukan optimasi dan performance evaluation

☐ Saya mahir menggunakan compiler tools modern

☐ Saya dapat mendokumentasikan proyek pengembangan compiler dengan baik

☐ Saya siap menerapkan prinsip kompilasi dalam rekayasa perangkat lunak

Rangkuman

Bab ini merangkum seluruh perjalanan pembelajaran Teknik Kompilasi, melalui evaluasi CPMK, refleksi diri, dan penyusunan portofolio. Mahasiswa belajar mengevaluasi pencapaian kompetensi secara komprehensif.

Poin Kunci:

- Evaluasi kompetensi mencakup teori dan praktik implementasi
- Refleksi membantu internalisasi pemahaman konsep
- Portofolio proyek menjadi bukti nyata pencapaian Sub-CPMK
- Continuous improvement penting untuk pengembangan karir professional
- Mata kuliah ini memberikan fondasi kuat dalam computer science

Kata Kunci: ***Evaluasi Kompetensi***, ***Refleksi***, ***Portofolio***, ***CPMK***, ***Self-Assessment***, ***Continuous Improvement***

Bab 18

Lampiran dan Rubrik Penilaian

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 1.1-6.2:** Menyediakan instrumen penilaian dan panduan praktis untuk seluruh capaian pembelajaran

18.1 Koleksi Quiz dan Uji Kompetensi

Uji pemahaman cepat untuk setiap topik utama:

18.1.1 Quiz 1: Arsitektur Kompilator

1. Apa perbedaan utama antara *front-end* dan *back-end* kompilator?
2. Mengapa kita memerlukan *Intermediate Representation*?

18.1.2 Quiz 2: Optimasi dan Code Gen

1. Apa resiko melakukan optimasi yang terlalu agresif?
2. Jelaskan istilah *Register Spilling*!

18.2 Template Dokumentasi Kode

18.2.1 Template Laporan Praktikum

```
NAMA MAHASISWA: [Nama Lengkap]
NIM: [Nomor Induk Mahasiswa]
KELAS: [Kelas]
MATA KULIAH: Teknik Kompilasi
TUGAS: [Judul Tugas]
TANGGAL: [Tanggal]

1. TUJUAN
   [Jelaskan tujuan praktikum]

2. TEORI DASAR
   [Jelaskan teori yang relevan]

3. IMPLEMENTASI
   [Jelaskan implementasi yang dilakukan]

4. HASIL DAN ANALISIS
   [Tampilkan hasil dan analisis]

5. KESULITAN DAN SOLUSI
   [Jelaskan kesulitan yang dihadapi dan solusinya]

6. KESIMPULAN
   [Buat kesimpulan dari praktikum]

7. LAMPIRAN
   [Lampirkan kode sumber lengkap]
```

18.2.2 Template Dokumentasi Kode

```c
/**
 * @file [nama_file].c
 * @brief [deskripsi singkat file]
 * @author [nama author]
 * @date [tanggal]
 *
 * [deskripsi detail file]
 */

/**
 * @brief [deskripsi fungsi]
 * @param [param1] [deskripsi parameter 1]
 * @param [param2] [deskripsi parameter 2]
 * @return [deskripsi return value]
 *
 * [detail implementasi fungsi]
 */
int function_name(int param1, char *param2) {
    // Implementation here
}
```

18.3 Checklist Kualitas Kode

18.3.1 Checklist untuk Lexer

- ☐ Regular expressions terdefinisi dengan benar
- ☐ Token recognition akurat
- ☐ Error handling untuk invalid input
- ☐ Memory management yang benar
- ☐ Kode style konsisten
- ☐ Dokumentasi lengkap
- ☐ Testing coverage memadai

18.3.2 Checklist untuk Parser

- ☐ Grammar definition benar
- ☐ Parsing algorithm efisien
- ☐ Syntax error reporting jelas
- ☐ AST construction benar
- ☐ Error recovery mechanism
- ☐ Testing dengan valid dan invalid input

18.3.3 Checklist untuk Code Generator

- ☐ Instruction selection optimal
- ☐ Register allocation efisien
- ☐ Target code generation benar
- ☐ Optimization terimplementasi
- ☐ Performance testing dilakukan
- ☐ Cross-platform compatibility

18.4 Format Submisi Tugas

18.4.1 Struktur Folder Proyek

```
[NIM]_[Nama]_[Tugas]/
|-- src/
|   |-- lexer.c
|   |-- parser.c
|   |-- ast.c
|   |-- symtab.c
|   |-- main.c
|   `-- util.h
|-- include/
|   |-- ast.h
|   `-- symtab.h
|-- tests/
|   |-- test1.c
|   `-- test2.c
|-- docs/
|   |-- README.md
|   `-- laporan.pdf
|-- Makefile
`-- README.md
```

18.4.2 Format Penamaan File

- Source code: `[nim]_[nama]_[komponen].c`
- Header files: `[nim]_[nama]_[komponen].h`
- Test files: `test_[komponen].c`
- Documentation: `laporan_[nim]_[nama].pdf`
- Executable: `[nim]_[nama]_[proyek]`

18.5 Best Practices Pengembangan Compiler

18.5.1 Coding Standards

- Gunakan consistent naming conventions
- Comment code yang kompleks
- Modular design dengan single responsibility
- Error handling yang robust
- Memory management yang safe

- Testing yang komprehensif

18.5.2 Version Control

```
# Git workflow untuk proyek compiler
git init
git add .
git commit -m "Initial commit: lexer implementation"
git branch feature-parser
git checkout feature-parser
# Implement parser
git add .
git commit -m "Add parser with LL(1) algorithm"
git checkout main
git merge feature-parser
git tag v1.0.0
```

18.5.3 Testing Strategies

- Unit testing untuk setiap komponen
- Integration testing untuk komponen interaksi
- End-to-end testing untuk complete compiler
- Performance testing untuk optimization validation
- Regression testing untuk maintenance

18.6 Resources Tambahan

18.6.1 Online Resources

- **Compiler Design**: https://www.cs.cornell.edu/courses/cs4120/2018fa/
- **LLVM Tutorial**: https://llvm.org/docs/tutorial/
- **Flex/Bison Manual**: https://www.gnu.org/software/flex/manual/
- **ANTLR**: https://www.antlr.org/

18.6.2 Recommended Books

- **Compilers: Principles, Techniques, and Tools** - Aho, Lam, Sethi, Ullman
- **Modern Compiler Implementation** - Andrew Appel

- **Engineering a Compiler** - Cooper and Torczon
- **Programming Language Pragmatics** - Michael Scott

18.6.3 Useful Tools

- **IDE**: VS Code, CLion, Eclipse CDT
- **Debugging**: GDB, Valgrind, AddressSanitizer
- **Profiling**: Perf, gprof, Intel VTune
- **Documentation**: Doxygen, Sphinx

Aktivitas Pembelajaran

1. **Rubrik Review**: Pelajari kriteria penilaian untuk setiap jenis tugas.
2. **Template Usage**: Gunakan template dokumentasi untuk laporan praktikum.
3. **Code Quality Check**: Lakukan self-audit menggunakan checklist kualitas kode.
4. **Workflow Implementation**: Terapkan best practices dalam pengembangan tugas.
5. **Resource Exploration**: Telusuri referensi tambahan untuk pendalaman materi.

Latihan dan Refleksi

1. Buat draf laporan praktikum menggunakan template yang disediakan!
2. Lakukan review kode menggunakan checklist kualitas untuk Lexer!
3. Simulasikan Git workflow untuk proyek kecil!
4. Susun struktur folder proyek sesuai standar submisi!
5. Analisis kriteria penilaian untuk mendapatkan nilai maksimal (Grade A)!
6. **Refleksi**: Bagaimana rubrik dan standar dokumentasi membantu efektivitas belajar Anda?

Asesmen (Evaluasi Kinerja)

Instrumen Evaluasi Kualitas Dokumentasi dan Praktik

A. Audit Dokumentasi

1. Kesesuaian dengan template laporan
2. Kejelasan penjelasan implementasi
3. Kualitas dokumentasi dalam kode (comments/doxygen)

B. Audit Standar Kode

1. Kepatuhan terhadap coding standards
2. Penggunaan version control (commit messages/history)
3. Struktur folder dan penamaan file

Rubrik Penilaian: Lihat Section 18.1

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya memahami kriteria penilaian untuk setiap komponen evaluasi

☐ Saya mahir menggunakan template dokumentasi yang disediakan

☐ Saya dapat menerapkan standar kualitas kode dalam pengembangan compiler

☐ Saya memahami format submisi tugas yang benar

☐ Saya dapat menerapkan best practices dalam version control dan testing

☐ Saya dapat memanfaatkan resources tambahan untuk belajar mandiri

Rangkuman

Bab ini menyediakan berbagai instrumen pendukung pembelajaran, mulai dari rubrik penilaian, template dokumentasi, checklist kualitas, hingga best practices dan referensi tambahan. Mahasiswa menggunakannya sebagai panduan standar dalam menyelesaikan tugas dan proyek.

Poin Kunci:

- Rubrik memberikan transparansi dalam penilaian kompetensi

- Template dan checklist memastikan konsistensi dan kualitas output
- Standar submisi mempermudah integrasi dan manajemen proyek
- Best practices meningkatkan profesionalisme dalam pengembangan software
- Resources tambahan membuka peluang eksplorasi lebih dalam

Kata Kunci: ***Rubrik Penilaian***, ***Template Dokumentasi***, ***Checklist Kualitas***, ***Best Practices***, ***Standard Submisi***, ***Learning Resources***

Bab 19

Daftar Referensi

Sub-CPMK yang Dicakup dalam Bab Ini:

- **Sub-CPMK 1.1-6.2:** Menunjukkan kemampuan literasi informasi dan penggunaan referensi ilmiah dalam bidang teknik kompilasi

19.1 Buku Referensi Utama

19.1.1 Compiler Design Fundamentals

- **Compilers: Principles, Techniques, and Tools** (2nd Edition)
  Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman
  Pearson/Addison-Wesley, 2007
  ISBN: 978-0321486813

- **Modern Compiler Implementation in C**
  Appel, Andrew W.
  Cambridge University Press, 1998
  ISBN: 978-0521583909

- **Engineering a Compiler** (2nd Edition)
  Cooper, Keith D. and Linda Torczon
  Morgan Kaufmann, 2011
  ISBN: 978-0120884788

- **Programming Language Pragmatics** (4th Edition)
  Scott, Michael L.
  Morgan Kaufmann, 2015
  ISBN: 978-0124104099

19.1.2 Advanced Compiler Topics

- **Advanced Compiler Design and Implementation**
  Muchnick, Steven S.
  Morgan Kaufmann, 1997
  ISBN: 978-1558603204

- **Optimizing Compilers for Modern Architectures**
  Allen, Randy and Ken Kennedy
  Morgan Kaufmann, 2001
  ISBN: 978-1558602863

- **The Dragon Book: Compilers**
  Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman
  Addison-Wesley, 1986
  ISBN: 978-0201100884

19.2 Buku Praktis dan Implementasi

19.2.1 Lexical Analysis and Parsing

- **flex & bison: Text Processing Tools**
  Levine, John R.
  O'Reilly Media, 2009
  ISBN: 978-0596155971

- **The Definitive ANTLR 4 Reference**
  Parr, Terence
  Pragmatic Bookshelf, 2013
  ISBN: 978-1934356599

- **Building a Parser with JavaCC**
  Grune, Dick and Ceriel J.H. Jacobs
  Addison-Wesley, 2008
  ISBN: 978-0321316324

19.2.2 Code Generation and Optimization

- **The LLVM Compiler Infrastructure**
  Lattner, Chris and Vikram Adve
  ACM SIGPLAN Notices, 2004

- **Linkers and Loaders**
  Levine, John R.
  Morgan Kaufmann, 2000
  ISBN: 978-1558604964

- **Computer Systems: A Programmer's Perspective**
  Bryant, Randal E. and David R. O'Hallaron
  Pearson, 2015
  ISBN: 978-0134092669

19.3 Jurnal dan Paper Akademik

19.3.1 Seminal Papers

- **A Fast Algorithm for Finding Dominators in a Flowgraph**
  Lengauer, Thomas and Robert E. Tarjan
  ACM Transactions on Programming Languages and Systems, 1979

- **Register Allocation via Graph Coloring**
  Chaitin, Gregory J.
  ACM SIGPLAN Notices, 1982

- **SSA-Based Optimizations**
  Cytron, Ron, et al.
  ACM Transactions on Programming Languages and Systems, 1991

- **Linear Scan Register Allocation**
  Poletto, Massimiliano and Vivek Sarkar
  ACM SIGPLAN Notices, 1999

19.3.2 Recent Research

- **LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation**
  Lattner, Chris and Vikram Adve
  International Symposium on Code Generation and Optimization, 2004

- **The LLVM Instruction Set and Compilation Strategy**
  Lattner, Chris
  University of Illinois at Urbana-Champaign, 2002

- **Automatic Vectorization for Free**
  Nuzman, Dorit, et al.
  ACM SIGPLAN Notices, 2006

19.4 Dokumentasi Teknis

19.4.1 Compiler Tools Documentation

- **GNU Compiler Collection Internals**
  Free Software Foundation
  https://gcc.gnu.org/onlinedocs/gccint/

- **LLVM Language Reference Manual**
  LLVM Project
  https://llvm.org/docs/LangRef.html

- **Flex Manual**
  Free Software Foundation
  https://westes.github.io/flex/manual/

- **Bison Manual**
  Free Software Foundation
  https://www.gnu.org/software/bison/manual/

19.4.2 Architecture Specifications

- **x86-64 Architecture Programmer's Manual**
  Intel Corporation
  https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html

- **ARM Architecture Reference Manual**
  ARM Limited
  https://developer.arm.com/documentation/ddi0487/latest

- **RISC-V ISA Manual**
  RISC-V Foundation
  https://riscv.org/technical/specifications/

19.5 Online Resources

19.5.1 Course Materials

- **CS4120/CS4121: Compilers**
  Cornell University
  https://www.cs.cornell.edu/courses/cs4120/2018fa/

- **CS143: Compilers**
  Stanford University
  https://www.stanford.edu/class/cs143/

- **15-213: Introduction to Computer Systems**
  Carnegie Mellon University
  https://www.cs.cmu.edu/~213/

19.5.2 Tutorials and Examples

- **LLVM Tutorial: My First Language Frontend**
  LLVM Project
  https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/index.html

- **Let's Build a Compiler**
  Jack Crenshaw
  https://compilers.iecc.com/crenshaw/

- **Writing a Simple Compiler**
  Carl Burch
  https://www.cburch.com/cs/330/

19.6 Standar dan Spesifikasi

19.6.1 Programming Language Standards

- **ISO/IEC 9899:2018 - C Programming Language**
  International Organization for Standardization

- **ISO/IEC 14882:2020 - C++ Programming Language**
  International Organization for Standardization

- **ISO/IEC 23660:2020 - Programming Language Specifications**
  International Organization for Standardization

19.6.2 Compiler Standards

- **DWARF Debugging Information Format**
  DWARF Standards Committee
  http://dwarfstd.org/

- **ELF: Executable and Linkable Format**
  UNIX System Laboratories
  https://refspecs.linuxfoundation.org/elf/

- **COFF: Common Object File Format**
  Microsoft Corporation

19.7 Software dan Tools

19.7.1 Open Source Compilers

- **GCC (GNU Compiler Collection)**
  Free Software Foundation
  https://gcc.gnu.org/

- **Clang**
  LLVM Project
  https://clang.llvm.org/

- **TinyCC (Tiny C Compiler)**
  Fabrice Bellard
  https://bellard.org/tcc/

19.7.2 Compiler Development Tools

- **ANTLR**
  University of San Francisco
  https://www.antlr.org/

- **Flex**
  Free Software Foundation
  https://github.com/westes/flex

- **Bison**
  Free Software Foundation
  https://www.gnu.org/software/bison/

19.8 Historical References

19.8.1 Classic Papers

- **Recursive Functions of Symbolic Expressions and Their Computation by Machine**
  McCarthy, John
  Communications of the ACM, 1960

- **A Method for Translating Algol 60**
  Bauer, Friedrich L. and Klaus Samelson
  Communications of the ACM, 1960

- **A Formal System for Specification and Verification of Compilers**
  McCarthy, John and James Painter
  Communications of the ACM, 1967

19.8.2 Historical Books

- **The Design of an Optimizing Compiler**
  Aho, Alfred V. and Jeffrey D. Ullman
  Elsevier, 1977

- **Compiler Construction: Theory and Practice**
  Gries, David
  Wiley, 1971

- **A Compiler Generator**
  Feldman, Joel and David Gries
  Prentice-Hall, 1970

19.9 Catatan Penggunaan Referensi

Referensi-referensi ini disusun untuk mendukung pembelajaran Teknik Kompilasi secara komprehensif:

- **Buku Utama**: Digunakan sebagai referensi fundamental untuk konsep dan teori
- **Buku Praktis**: Memberikan panduan implementasi dan penggunaan tools
- **Paper Akademik**: Menyajikan penelitian terkini dan algoritma modern
- **Dokumentasi Teknis**: Referensi spesifik untuk tools dan arsitektur

- **Online Resources**: Materi tambahan dan tutorial interaktif
- **Standar**: Spesifikasi resmi untuk bahasa dan format
- **Software**: Tools yang dapat digunakan untuk praktikum
- **Historical**: Konteks perkembangan compiler construction

Semua referensi ini dapat diakses melalui perpustakaan universitas, online repositories, atau pembelian langsung. Mahasiswa disarankan untuk memilih referensi yang sesuai dengan kebutuhan pembelajaran dan proyek yang sedang dikerjakan.

Aktivitas Pembelajaran

1. **Reference Deep Dive**: Pilih satu buku referensi utama dan rangkum satu bab.
2. **Paper Analysis**: Baca paper seminal tentang register allocation dan analisis algoritmanya.
3. **Tool Investigation**: Eksplorasi dokumentasi LLVM atau GCC untuk fitur optimasi tertentu.
4. **Bibliography Management**: Buat database referensi pribadi menggunakan BibTeX.
5. **Resource Mapping**: Petakan referensi ke setiap Sub-CPMK yang ada dalam mata kuliah.

Latihan dan Refleksi

1. Temukan perbedaan pendekatan antara Dragon Book dan Engineering a Compiler!
2. Cari paper terbaru (3 tahun terakhir) tentang compiler optimization di ACM Digital Library!
3. Bandingkan spesifikasi x86 dan ARM dalam hal calling conventions dari manual teknis!
4. Identifikasi kontribusi John McCarthy dalam sejarah teknik kompilasi!
5. Susun rencana membaca referensi untuk mendukung proyek portofolio Anda!
6. **Refleksi**: Bagaimana keragaman referensi membantu Anda memahami kompleksitas teknik kompilasi?

Asesmen (Evaluasi Kinerja)

Instrumen Evaluasi Kemampuan Literasi

A. Review Literatur

1. Kualitas sitasi dalam laporan proyek
2. Kedalaman analisis perbandingan antar referensi
3. Ketepatan penggunaan standar teknis

B. Presentasi Referensi

1. Kemampuan menjelaskan konsep dari paper akademik
2. Relevansi referensi yang dipilih dengan masalah teknis yang dihadapi

Rubrik Penilaian: Lihat Lampiran A

Checklist Pencapaian Kompetensi

Centang item berikut setelah Anda yakin telah menguasainya:

☐ Saya mengenal buku-buku referensi utama dalam bidang teknik kompilasi

☐ Saya dapat menelusuri paper akademik untuk algoritma spesifik

☐ Saya mahir membaca dan mengikuti dokumentasi teknis compiler tools

☐ Saya memahami standar dan spesifikasi arsitektur serta format file

☐ Saya dapat menggunakan online resources dan tutorial secara efektif

☐ Saya memahami konteks sejarah dan perkembangan teknologi kompilator

Rangkuman

Bab ini menyajikan daftar referensi komprehensif yang mencakup buku teks, paper akademik, dokumentasi teknis, dan online resources. Mahasiswa belajar memanfaatkan berbagai sumber informasi untuk memperdalam pemahaman teknik kompilasi.

Poin Kunci:

- Literatur klasik memberikan fondasi teori yang kuat
- Dokumentasi modern memberikan panduan praktis implementasi

- Research papers menunjukkan arah perkembangan teknologi masa depan
- Standar teknis memastikan interoperabilitas dan kepatuhan sistem
- Pembelajaran mandiri didorong melalui eksplorasi berbagai media belajar

Kata Kunci: ***Referensi***, ***Literatur***, ***Paper Akademik***, ***Dokumentasi***, ***Standar***, ***Historical Context***

Lampiran

Lampiran A: Rubrik Penilaian Proyek Kompilator

Kriteria	Sangat Baik	Baik	Perlu Perbaikan
Analisis Leksikal	Tokenisasi akurat, penanganan error baik	Tokenisasi akurat, error handling terbatas	Tokenisasi kurang akurat
Analisis Sintaksis	Parser efisien, AST terbentuk benar	Parser berfungsi, AST ada sedikit error	Parser sering gagal
Manajemen Tabel Simbol	Implementasi hashtable/tree efisien	Implementasi linear search	Tidak ada tabel simbol
Kualitas Kode	Modular, efisien, naming konsisten	Cukup modular, ada duplikasi	Tidak modular, sulit dibaca
Dokumentasi	Lengkap dan membantu	Cukup lengkap	Minim dokumentasi

Tabel 19.1: Rubrik Penilaian Proyek Kompilator

Lampiran B: Glosarium Istilah Teknik Kompilasi

- **Compiler**: Program penerjemah bahasa tingkat tinggi ke bahasa mesin.
- **Interpreter**: Program pengeksekusi kode sumber secara langsung.
- **Token**: Unit leksikal terkecil dalam bahasa pemrograman.
- **Lexeme**: String aktual yang membentuk token.
- **AST (Abstract Syntax Tree)**: Representasi pohon dari struktur sintaksis kode.
- **Intermediate Representation**: Bentuk antara antara source dan target code.

Daftar Pustaka

[1] StudyLib. *Outcomes-Based Education Curricula*. Materials on runtime environment and activation records. 2024. URL: https://studylib.net/doc/14111770/outcomes-based-education-curricula--academic-year-2015-.

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

[3] Northeastern University. *CS 4410/6410: Compiler Design*. Course syllabus and materials, taught by Benjamin Lerner. 2024. URL: https://course.ccs.neu.edu/cs4410sp25/.

[4] Keith D. Cooper **and** Linda Torczon. *Engineering a Compiler*. 2nd. Morgan Kaufmann, 2011.

[5] 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/.

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

[7] Aoyama Gakuin University. *Compiler Lecture 5: Lexical Analysis*. Lecture notes. 2024. URL: https://www.sw.it.aoyama.ac.jp/2025/Compiler/lecture5.html.

[8] OpenGenus. *Build Lexer*. Tutorial on hand-written lexers. 2024. URL: https://iq.opengenus.org/build-lexer/.

[9] Wikipedia. *re2c*. Encyclopedia entry. 2024. URL: https://en.wikipedia.org/wiki/Re2c.

[10] Wikipedia. *RE/flex*. Encyclopedia entry. 2024. URL: https://en.wikipedia.org/wiki/Draft:RE/flex.

[11] John R. Levine. *flex & bison: Text Processing Tools*. O'Reilly Media, 2009.

[12] GNU Project. *GNU Bison Manual*. Official Bison documentation. 2024. URL: https://www.gnu.org/software/bison/manual/.

[13] IT Trip. *C Parser Flex Bison*. Tutorial. 2024. URL: https://en.ittrip.xyz/c-language/c-parser-flex-bison.

[14] Nguyen Thanh Vu. *Compiler Class Notes: Semantic Analysis*. Class notes. 2024. URL: https://nguyenthanhvuh.github.io/class-compilers/notes/sem.html.

[15] UC San Diego CSE Department. *CSE 231: Compiler Construction / Advanced Compiler Design*. Course materials and syllabus. 2024. URL: https://catalog.ucsd.edu/archive/2024-25/courses/CSE.html.

[16] Johns Hopkins University. *EN.601.428/628: Compilers and Interpreters*. Course syllabus and materials, taught by David Hovemeyer. 2024. URL: https://jhucompilers.github.io/fall2025/syllabus.html.

[17] University of Oxford. *Compilers Course*. Course materials, Michaelmas Term 2024, taught by Matty Hoban. 2024. URL: https://www.cs.ox.ac.uk/teaching/courses/2024-2025/com/.