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
18 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	: 18 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

Pada era rekayasa perangkat lunak modern, pemahaman mendalam tentang mekanisme internal kompilator bukan sekadar tentang menerjemahkan kode, melainkan tentang penguasaan seni perancangan sistem yang kompleks. 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 menjembatani kesenjangan antara teori ilmu komputer yang abstrak dengan praktik konstruksi perangkat lunak yang nyata, memastikan mahasiswa mampu membangun fondasi teoretis yang kuat sekaligus keterampilan praktis yang relevan dengan industri.

Kompilator adalah fondasi dari seluruh ekosistem komputasi, bertugas mentransformasi logika tingkat tinggi yang dipahami manusia menjadi instruksi mesin yang dapat dieksekusi oleh perangkat keras. Sebagaimana dijelaskan oleh [2, 3], proses kompilasi ini melibatkan serangkaian fase yang rumit namun terstruktur, mulai dari analisis leksikal hingga optimasi kode target. Dengan memahami setiap fase ini, mahasiswa akan memperoleh wawasan mendalam tentang bagaimana perangkat lunak berinteraksi dengan sumber daya perangkat

keras, sebuah kompetensi krusial untuk membangun aplikasi yang efisien, aman, dan berkinerja tinggi.

Pendekatan OBE yang diterapkan dalam buku ini memastikan bahwa setiap proses pembelajaran didorong oleh tujuan yang jelas dan hasil yang terukur (*Outcome*). Buku ini tidak hanya bertujuan untuk menyampaikan materi, tetapi untuk memastikan bahwa setiap bab berkontribusi langsung pada pembentukan kompetensi spesifik mahasiswa. Mahasiswa tidak hanya dituntut untuk mengetahui "apa" itu *parser* atau *lexer*, tetapi juga "bagaimana" merancang dan mengimplementasikannya dalam sebuah proyek kompilator yang memenuhi standar kualitas industri.

Tujuan spesifik dari buku ajar ini meliputi:

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 disusun dengan penyelarasan yang ketat terhadap Rencana Pembelajaran Semester (RPS) mata kuliah Teknik Kompilasi yang menerapkan kerangka kerja *Outcome-Based Education* (OBE). Keterkaitan ini menjamin bahwa setiap aktivitas kognitif yang dilakukan mahasiswa—mulai dari memahami definisi hingga merancang sistem—memiliki

kontribusi langsung terhadap pencapaian Capaian Pembelajaran Lulusan (CPL). Integrasi yang kuat antara konten buku dan CPL memastikan bahwa keterampilan yang dikuasai mahasiswa relevan dengan kebutuhan dan standar kompetensi di industri teknologi informasi modern [4].

Secara spesifik, buku ini dirancang untuk memenuhi CPL-1 (Pengetahuan Rekayasa) dengan menyajikan teori otoritatif mengenai bahasa formal, automata, dan tata bahasa bebas konteks. Di sisi lain, buku ini juga secara intensif menargetkan CPL-3 (Perancangan dan Pengembangan Solusi) dengan membimbing mahasiswa melalui proses iteratif pembangunan kompilator yang fungsional. Keseimbangan ini memastikan mahasiswa tidak hanya tumbuh sebagai teoretisi yang memahami konsep abstrak, tetapi juga sebagai insinyur perangkat lunak yang mampu menghasilkan solusi konkret untuk masalah yang kompleks.

Aktivitas pembelajaran dalam buku ini disusun secara bertingkat mengikuti hierarki Taksonomi Bloom untuk memandu perkembangan kognitif mahasiswa. Dimulai dari level dasar seperti memahami sintaksis dan semantik bahasa, mahasiswa dibimbing menuju level analisis untuk memecahkan konflik *parsing*, dan akhirnya mencapai level tertinggi yaitu menciptakan (*creating*) sebuah pemroses bahasa yang utuh. Progresi bertahap ini sangat krusial dalam pendidikan teknik untuk membantu mahasiswa menguasai permasalahan rekayasa yang rumit secara terstruktur dan percaya diri.

1.3 Arsitektur Kompilator Modern

Arsitektur kompilator modern umumnya terbagi menjadi front-end dan back-end [5].

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

Agar pembaca mendapatkan manfaat maksimal dari buku ini, disarankan untuk mengadopsi pendekatan pembelajaran aktif (*Active Learning*). Mahasiswa tidak cukup hanya membaca teori secara pasif, tetapi harus secara aktif terlibat dengan materi melalui eksperimen kode dan latihan pemecahan masalah. Keterlibatan langsung ini akan merangsang proses kognitif tingkat tinggi yang diperlukan untuk memahami abstraksi kompleks dalam teknik kompilasi.

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, terutama konsep pohon, graf, dan tabel hash.

Tahap Implementasi (Eksperimental) [6]:

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

Dalam paradigma OBE, peran dosen bergeser dari sekadar penyampai informasi menjadi fasilitator pembelajaran. Buku ini dirancang untuk mendukung transisi tersebut dengan menyediakan materi yang dapat memicu diskusi kritis dan eksplorasi mandiri. Dosen diharapkan dapat membimbing mahasiswa untuk menemukan hubungan antar konsep, bukan sekadar memberikan jawaban langsung.

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 pendidikan yang berpusat pada mahasiswa (*Student-Centered Learning*), di mana kurikulum dirancang berdasarkan apa yang seharusnya mampu dilakukan mahasiswa setelah lulus. Dalam konteks Teknik Kompilasi, filosofi ini mengubah fokus dari sekadar penyampaian materi algoritma menjadi pembentukan kemampuan rekayasa nyata. Tujuannya bukan hanya agar mahasiswa mengetahui teori *parsing*, tetapi agar mereka mampu membangun parser yang efisien, tangguh, dan sesuai standar industri.

Pendekatan ini juga menekankan pentingnya penyelarasan konstruktif (*Constructive Alignment*) antara aktivitas belajar, metode asesmen, dan capaian pembelajaran. Setiap tugas pemrograman dan ujian teori dalam buku ini dirancang secara spesifik untuk memvalidasi pencapaian kompetensi tertentu. Hal ini menjamin transparansi dalam proses penilaian,

sehingga mahasiswa memahami dengan jelas ekspektasi dan standar kualitas yang harus mereka penuhi.

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

Implementasi OBE dalam buku ini juga mencakup siklus perbaikan kualitas berkelanjutan (*Continuous Quality Improvement*). Melalui mekanisme *feedback* yang terintegrasi dalam setiap bab, mahasiswa diajak untuk terus mengevaluasi dan memperbaiki hasil kerja mereka. Proses iteratif ini meniru siklus hidup pengembangan perangkat lunak profesional, di mana kode selalu ditinjau, diuji, dan dioptimalkan secara berkala.

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, yang dapat dipandang sebagai sebuah "pipa transformasi" data. Proses dimulai dari aliran karakter mentah (*source code*), yang kemudian diubah menjadi token linear, disusun menjadi struktur pohon hierarkis (*Syntax Tree*), diperkaya dengan informasi semantik, diterjemahkan menjadi kode antara (*Intermediate Representation*) yang agnostik terhadap mesin, dan akhirnya diekspansi menjadi instruksi mesin spesifik (*Assembly*) yang optimal. Setiap bab dalam buku ini membedah satu ruas dari pipa tersebut secara mendalam.

Secara holistik, peta konsep ini tidak hanya mengajarkan teknik isolasi komponen, tetapi juga integrasi sistem. Mahasiswa diajak untuk melihat bagaimana keputusan desain di satu fase (misalnya, desain IR) berdampak signifikan pada fase berikutnya (optimasi dan generasi kode). Pemahaman lintas-fase ini adalah esensi dari pemikiran sistem (*systems thinking*) yang ingin dibangun melalui mata kuliah ini.

Berikut adalah rincian materi per bab:

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

Salah satu kekuatan utama buku ajar ini adalah penggunaan proyek pengembangan tunggal yang berkelanjutan (*continuous project*) berupa kompilator untuk subset bahasa C. Pendekatan ini dipilih karena bahasa C merupakan *lingua franca* sistem pemrograman yang memiliki karakteristik imperatif, prosedural, dan *statically typed* yang representatif. Dengan membangun kompilator untuk subset C, mahasiswa akan menghadapi tantangan nyata yang relevan dengan industri, namun dalam lingkup yang terkendali sehingga tetap dapat diselesaikan dalam satu semester.

Proyek ini dibangun menggunakan filosofi pengembangan inkremental (*incremental development*). Kita tidak mencoba membangun seluruh kompilator sekaligus. Sebaliknya, kita membangunnya lapis demi lapis—dimulai dari lexer sederhana, kemudian parser, lalu AST, dan seterusnya. Setiap bab menambahkan fungsionalitas baru ke atas fondasi yang telah dibangun sebelumnya. Metode ini tidak hanya memudahkan proses *debugging*, tetapi juga mengajarkan disiplin rekayasa perangkat lunak tentang bagaimana mengelola kompleksitas melalui modularitas.

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 ⋆ factor | term / factor

- **Factor**: literal | identifier | `(` ekspresi `)`

Precedence: ⋆ 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. Namun, pandangan ini terlalu menyederhanakan kompleksitas yang ada di dalamnya. Menurut [2], kompilasi sebenarnya adalah orkestrasi dari serangkaian proses transformasi data yang presisi, di mana makna program harus dipertahankan secara utuh melintasi berbagai representasi.

Secara formal, ***compiler*** adalah program perangkat lunak yang membaca program yang ditulis dalam satu bahasa (bahasa sumber) dan menerjemahkannya ke dalam program yang setara dalam bahasa lain (bahasa target). Output dari kompilator biasanya adalah kode mesin atau kode assembly yang dapat dieksekusi langsung oleh perangkat keras, namun bisa juga berupa kode sumber bahasa tingkat tinggi lain (source-to-source compiler/transpiler).

2.1.2 Karakteristik Kompilator

Kompilator memiliki beberapa karakteristik unik yang membedakannya dari alat pemroses bahasa lainnya:

- **Translasi Lengkap**: Kompilator menganalisis seluruh unit kompilasi (file atau modul) secara utuh sebelum menghasilkan output, memungkinkan validasi lintas baris kode.

- **Analisis Mendalam**: Melakukan verifikasi statis yang ketat meliputi pemeriksaan sintaksis, tipe data, dan aliran kontrol untuk menjamin keamanan program sebelum dijalankan.

- **Output Terpisah**: Hasil kompilasi disimpan sebagai artefak permanen (seperti file `.exe` atau `.o`) yang dapat didistribusikan dan dijalankan berulang kali tanpa memerlukan kode sumber asli.

- **Optimasi Matematis**: Menerapkan berbagai transformasi matematis untuk meningkatkan efisiensi eksekusi (kecepatan) dan efisiensi ruang (memori) dari kode hasil.

2.1.3 Interpreter vs Compiler dan Arsitektur Hibrida

Perbedaan klasik antara ***compiler*** dan ***interpreter*** terletak pada waktu eksekusi. Kompilator melakukan translasi sekali di awal (*ahead-of-time*), sedangkan interpreter melakukan translasi baris-per-baris saat program berjalan. Namun, di era komputasi modern, batas ini semakin kabur dengan munculnya arsitektur hibrida.

Sistem seperti Java Virtual Machine (JVM) dan .NET CLR menggunakan pendekatan dua langkah: kode sumber dikompilasi menjadi *bytecode* (bentuk antara yang kompak), yang kemudian dijalankan oleh interpreter atau dikompilasi ulang menjadi kode mesin asli saat runtime menggunakan teknik *Just-In-Time (JIT) Compilation*. Pendekatan ini menggabungkan portabilitas interpreter dengan performa tinggi *native code*.

Aspek	**Compiler**	**Interpreter**
Eksekusi	Compile lalu run (AOT)	Run langsung
Kecepatan	Cepat (native speed)	Lambat (overhead translasi)
Analisis Error	Menyeluruh sebelum run	Berhenti saat error terjadi
Portabilitas	Perlu re-compile per OS	Kode sumber jalan di mana saja
Contoh	C, C++, Rust, Go	Python, Ruby, PHP

Tabel 2.1: Perbandingan Karakteristik Compiler dan Interpreter

2.2 Tingkat Bahasa Pemrograman

Bahasa pemrograman dapat diklasifikasikan berdasarkan tingkat abstraksinya terhadap mesin. Semakin tinggi tingkat abstraksi, semakin besar peran translator untuk menjembatani kode sumber menjadi instruksi mesin yang dapat dieksekusi.

Level	Contoh	Translator
Bahasa Mesin	Kode biner spesifik arsitektur (x86-64, ARM)	– (langsung CPU)
Tingkat Rendah	Assembly, instruksi dekat hardware	Assembler
Tingkat Menengah	C dengan pointer dan kontrol memori	Compiler
Tingkat Tinggi	Python, Java, C# dengan abstraksi tinggi	Compiler/Interpreter

Tabel 2.2: Level bahasa pemrograman dan kebutuhan translator

2.2.1 Contoh Ringkas

- **Bahasa Mesin (konseptual)**: `10110000 00000001`
- **Assembly**: `mov eax, 1`
- **C**: `int x = 1;`
- **Python**: `print(1)`

2.2.2 Bahasa Mesin

Bahasa mesin adalah representasi biner yang dipahami langsung oleh CPU. Setiap instruksi dan data direpresentasikan sebagai bit (0 dan 1) yang spesifik terhadap arsitektur prosesor tertentu (misalnya x86-64 atau ARM).

2.2.3 Bahasa Tingkat Rendah

Bahasa tingkat rendah umumnya merujuk pada bahasa *assembly*. Instruksi assembly memiliki korespondensi yang sangat dekat dengan instruksi mesin, sehingga efisien tetapi sulit dipahami dan tidak portabel.

2.2.4 Bahasa Tingkat Menengah

Bahasa tingkat menengah (contoh: C) berada di antara kemudahan bahasa tingkat tinggi dan kendali bahasa tingkat rendah. Bahasa ini menyediakan abstraksi seperti fungsi dan tipe data, namun tetap memungkinkan akses memori langsung melalui pointer.

2.2.5 Bahasa Tingkat Tinggi

Bahasa tingkat tinggi (contoh: Python, Java, C#) lebih dekat dengan cara berpikir manusia. Bahasa ini memiliki abstraksi yang kaya (object, module, library) sehingga produktif, tetapi membutuhkan compiler atau interpreter untuk dieksekusi oleh mesin.

2.3 Translator: Assembler, Interpreter, dan Compiler

Translator adalah program yang menerjemahkan kode sumber dari satu bahasa ke bahasa lain. Dalam ekosistem pemrograman, terdapat tiga jenis translator utama yang bekerja pada level berbeda.

2.3.1 Assembler

Assembler menerjemahkan kode *assembly* ke bahasa mesin. Proses ini umumnya satu-ke-satu (setiap instruksi assembly dipetakan ke instruksi mesin yang setara) dan menghasilkan *object file* yang siap untuk di-link.

2.3.2 Interpreter

Interpreter menjalankan program secara langsung dengan membaca dan mengeksekusi instruksi baris demi baris. Interpreter tidak menghasilkan file biner permanen; kinerja biasanya lebih lambat, tetapi fleksibel dan cocok untuk prototyping cepat.

2.3.3 Compiler

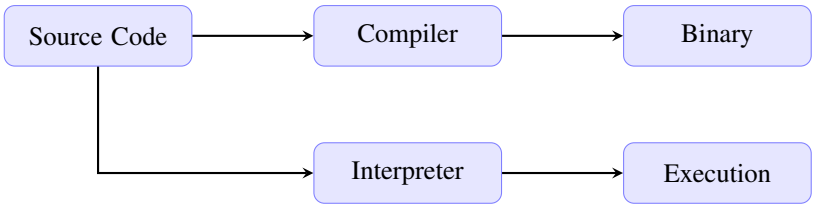
Compiler menerjemahkan seluruh program ke bentuk target (assembly atau kode mesin) sebelum dijalankan. Hasil kompilasi berupa artefak biner yang dapat dieksekusi berulang kali tanpa kode sumber. Compiler memungkinkan optimasi yang lebih agresif.

2.3.4 Perbandingan Singkat

- **Assembler**: input assembly $\rightarrow$ output machine code (sangat dekat dengan hardware).
- **Interpreter**: input source $\rightarrow$ eksekusi langsung (tanpa output biner permanen).
- **Compiler**: input source $\rightarrow$ output biner/assembly (dengan fase optimasi).

Translator	**Input**	**Output**	**Eksekusi**	**Contoh**
Assembler	Assembly	Machine code / object file	Setelah link	NASM, GAS
Interpreter	Source code	Eksekusi langsung	Saat run	Python, Ruby
Compiler	Source code	Assembly / biner	Setelah compile	GCC, Clang

Tabel 2.3: Perbandingan assembler, interpreter, dan compiler



Gambar 2.1: Alur eksekusi dengan compiler vs interpreter

2.4 Teori Utama Compiler

2.4.1 Teori Bahasa Formal dalam Desain Kompilator

Mengapa kita membutuhkan teori matematika untuk membangun teks processor? Jawabannya terletak pada presisi. Bahasa pemrograman harus didefinisikan secara tidak ambigu. Teori bahasa formal menyediakan kerangka kerja untuk mendefinisikan aturan-aturan tersebut secara matematis.

Dalam desain kompilator, kita menggunakan dua kelas utama dari hierarki Chomsky:

- **Regular Languages** (dikenali oleh Finite Automata): Digunakan untuk spesifikasi **token** dalam analisis leksikal. Ekspresi reguler dipilih karena efisien untuk mengenali pola linier sederhana seperti kata kunci, identifier, dan literal angka, namun tidak memiliki memori untuk mengenali struktur bersarang (nested).

- **Context-Free Languages** (dikenali oleh Pushdown Automata): Digunakan untuk spesifikasi **sintaksis** dalam analisis parsing. Tata bahasa bebas konteks (CFG) memiliki kemampuan "memori" melalui struktur tumpukan (*stack*), yang memungkinkannya mengenali struktur rekursif dan bersarang seperti blok `if` di dalam `while`, atau kurung matematis `((a+b)*c)`.

2.4.2 Evolusi Arsitektur: One-Pass vs Multi-Pass

Keputusan desain arsitektur kompilator sering kali dipengaruhi oleh keterbatasan perangkat keras pada zamannya dan kebutuhan optimasi.

One-Pass Compiler

Pada masa awal komputasi di mana memori sangat mahal dan terbatas (misalnya era PDP-11), kompilator dirancang sebagai *single-pass*: membaca kode sumber dan langsung menghasilkan kode mesin dalam satu lintasan tanpa menyimpan representasi antara yang besar di memori.

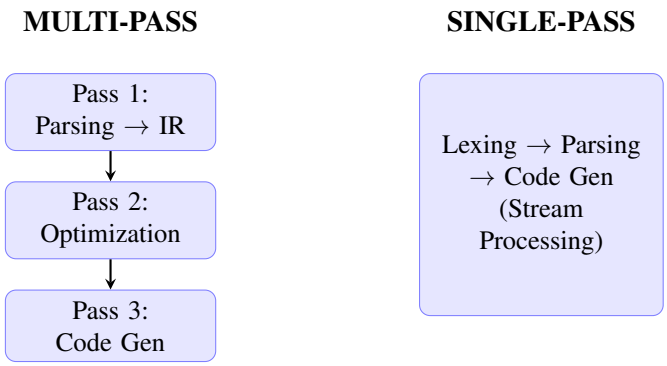
- **Kelebihan**: Kecepatan kompilasi sangat tinggi dan jejak memori minimal.

- **Kekurangan**: Tidak dapat melakukan optimasi yang memerlukan informasi global (misalnya: apakah fungsi di bawah dipanggil oleh fungsi di atas) dan struktur bahasa menjadi kaku (variabel harus dideklarasikan sebelum digunakan).
- **Legacy**: Bahasa Pascal dirancang spesifik agar bisa dikompilasi secara one-pass.

Multi-Pass Compiler

Kompilator modern hampir seluruhnya mengadopsi arsitektur *multi-pass*. Kode sumber dibaca dan diubah menjadi struktur pohon (AST) atau Intermediate Representation (IR), lalu diproses berulang kali oleh berbagai fase analisis dan optimasi.

- **Kelebihan**: Pemisahan perhatian (*separation of concerns*) yang bersih antar fase, memungkinkan optimasi global yang agresif, dan mendukung fitur bahasa yang kompleks (seperti *forward reference*).
- **Kekurangan**: Waktu kompilasi lebih lama dan konsumsi memori lebih besar.
- **Contoh**: GCC dan LLVM menjalankan puluhan "pass" optimasi pada IR sebelum menghasilkan kode mesin.



Gambar 2.2: Perbandingan arsitektur Multi-Pass dan Single-Pass Compiler

2.5 Grammar dan Hierarki Chomsky

Hierarki Chomsky mengklasifikasikan bahasa formal berdasarkan kekuatan ekspresinya dan jenis automata yang dapat mengenalinya. Klasifikasi ini penting karena fase kompilasi memanfaatkan kelas bahasa yang berbeda.

- **Type 0 (Unrestricted Grammar)**: Bahasa paling umum, setara dengan *Turing Machine*. Semua bahasa yang dapat dihitung (recursively enumerable) berada pada level ini.

- **Type 1 (Context-Sensitive Grammar)**: Dikenali oleh *Linear Bounded Automata*. Dapat memodelkan dependensi konteks yang tidak dapat ditangani oleh CFG.
- **Type 2 (Context-Free Grammar)**: Dikenali oleh *Pushdown Automata*. Digunakan untuk mendefinisikan struktur sintaksis bahasa pemrograman.
- **Type 3 (Regular Grammar)**: Dikenali oleh *Finite Automata*. Digunakan untuk mendefinisikan token pada analisis leksikal.

Dalam praktik kompilator, *regular language* digunakan pada lexer dan *context-free language* digunakan pada parser. Level di atasnya jarang dipakai secara langsung dalam konstruksi compiler dasar.

Tipe	Automata Pengenal	Penggunaan di Compiler
Type 0	Turing Machine	Teori komputasi umum, tidak dipakai langsung
Type 1	Linear Bounded Automata	Analisis kontekstual khusus (jarang)
Type 2	Pushdown Automata	Parsing dengan CFG (struktur sintaks)
Type 3	Finite Automata	Tokenization di lexer (regex)

Tabel 2.4: Ringkasan hierarki Chomsky dan peran di kompilator

2.5.1 Contoh Mini Grammar

Regular Grammar (Type 3):

```
R -> aR | bR | a | b
```

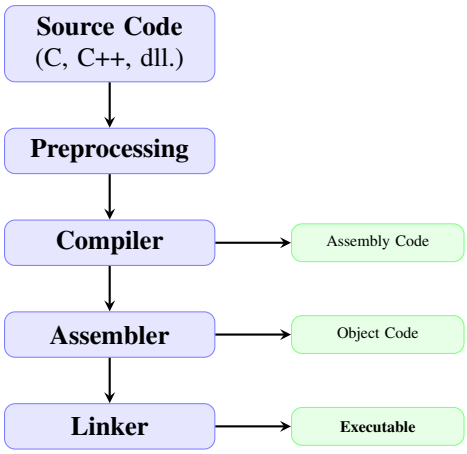
Context-Free Grammar (Type 2):

```
S -> a S b | ab
```

2.6 Alur Kerja dan Arsitektur Kompilator

2.6.1 Alur Kerja Kompilator: Dari Source ke Executable

Proses kompilasi bukanlah langkah tunggal, melainkan sebuah rantai produksi (*pipeline*) yang mentransformasi data dari satu bentuk ke bentuk lain. Gambar 2.3 mengilustrasikan perjalanan kode sumber: dimulai dari pre-processing yang menangani makro, masuk ke kompilator inti untuk diubah menjadi bahasa assembly tingkat rendah, lalu diteruskan ke assembler untuk menjadi kode objek biner, dan akhirnya disatukan oleh Linker dengan pustaka eksternal menjadi file eksekusi utuh. Pemahaman tentang rantai alat (*toolchain*) ini penting bagi seorang *systems programmer*.



Gambar 2.3: Alur kerja sistem kompilasi standar (GNU Toolchain model)

2.6.2 Arsitektur Tiga Tingkat: Front-end, Middle-end, Back-end

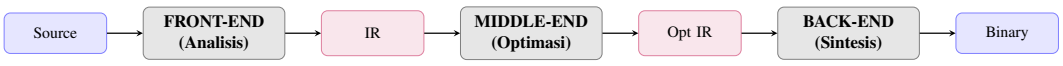
Meskipun secara tradisional kita membagi kompilator menjadi dua (Front-end dan Back-end), kompilator modern seperti LLVM memperkenalkan arsitektur tiga tingkat yang revolusioner.

- **Front-end (Analisis)**: Bertanggung jawab memahami kode sumber. Sifatnya spesifik terhadap bahasa (misal: Clang untuk C/C++, Rustc untuk Rust). Tugasnya memvalidasi sintaks/semantik dan menghasilkan *Intermediate Representation* (IR).

- **Middle-end (Optimasi)**: Jantung dari efisiensi kompilator. Bekerja sepenuhnya pada IR dan bersifat agnostik terhadap bahasa sumber maupun mesin target. Di sinilah optimasi matematika tingkat tinggi seperti *Dead Code Elimination* dan *Loop Unrolling* terjadi.

- **Back-end (Sintesis)**: Bertanggung jawab menerjemahkan IR yang sudah dioptimasi ke dalam instruksi mesin spesifik (misal: x86-64, ARM64, RISC-V). Sifatnya spesifik terhadap target hardware.

Keunggulan desain ini adalah **retargetability**. Untuk membuat kompilator bahasa baru ke 10 arsitektur mesin, kita hanya perlu membuat 1 Front-end, bukan 10 kompilator penuh. Masalah $N \times M$ (bahasa $\times$ target) direduksi menjadi $N + M$.



Gambar 2.4: Arsitektur Tiga Tingkat Kompilator Modern (Model LLVM)

2.7 Fase-Fase Kompilasi Secara Detail

Transformasi dari kode sumber menjadi kode target melibatkan serangkaian langkah logis yang presisi. Setiap langkah menerima input dari fase sebelumnya dan menghasilkan output intermediate untuk fase berikutnya. Kesalahan pada satu fase dapat menghentikan seluruh proses atau mempropagasi error ke fase selanjutnya.

2.7.1 Analisis Leksikal (Scanner)

Fase pertama ini bertugas membaca string karakter mentah dari kode sumber dan mengelompokkannya menjadi unit-unit bermakna yang disebut **token**. Scanner membuang elemen yang tidak relevan bagi mesin seperti spasi (*whitespace*), tab, baris baru, dan komentar. Scanner juga mendeteksi kesalahan leksikal sederhana, seperti karakter ilegal atau literal string yang tidak ditutup.

- **Input**: Stream karakter (`c, o, u, n, t,  , =,  , 1, 0`).
- **Output**: Stream token (`ID:count`, `ASSIGN`, `INT:10`).

2.7.2 Analisis Sintaksis (Parser)

Parser menerima stream token dan memverifikasi apakah urutan token tersebut membentuk struktur kalimat yang valid sesuai tata bahasa (*geometry*) dari bahasa pemrograman. Jika valid, parser membangun struktur data hierarkis visual yang disebut *Abstract Syntax Tree* (AST). Parser modern juga dilengkapi kemampuan *Error Recovery*, yang memungkinkannya melaporkan beberapa kesalahan sintaks sekaligus tanpa berhenti pada kesalahan pertama.

- **Input**: Stream token.
- **Output**: AST yang merepresentasikan struktur gramatikal (misal: node `Assignment` memiliki anak kiri `count` dan anak kanan `10`).

2.7.3 Analisis Semantik

Analisis semantik memberi makna pada struktur sintaksis yang telah dibangun. Fase ini memastikan bahwa program tersebut masuk akal secara logika, bukan hanya benar secara tata bahasa. Tugas utamanya meliputi pemeriksaan tipe (*type checking*), resolusi lingkup (*scope resolution*), dan verifikasi inisialisasi variabel. Semua informasi tentang identifier (variabel, fungsi) disimpan dan dilacak dalam struktur data sentral bernama **Symbol Table**.

- **Tugas**: Memastikan `count = "halo"` ditolak jika `count` bertipe `int`.

2.7.4 Generasi Intermediate Code (IR)

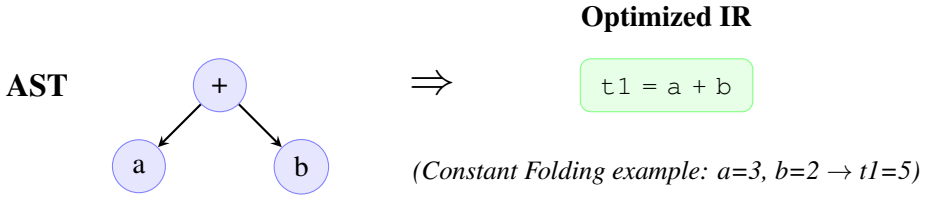
Setelah analisis selesai dan kode dinyatakan valid, kompilator menerjemahkan AST ke dalam representasi antara (*Intermediate Representation*). IR adalah bahasa pseudo-assembly yang sederhana, eksplisit, dan independen terhadap mesin target. Salah satu bentuk IR standar adalah *Three-Address Code* (TAC), di mana setiap instruksi memiliki paling banyak tiga operan, memudahkan optimasi dan translasi ke assembly nyata.

2.7.5 Optimasi Kode (Code Optimization)

Fase ini bertujuan meningkatkan kualitas kode IR tanpa mengubah perilaku program. Optimasi dapat berupa *local optimization* (seperti *constant folding*: mengubah `x = 3 + 5` menjadi `x = 8`) atau *global optimization* (seperti memindahkan perhitungan invarian keluar dari loop). Tujuannya adalah mengurangi jumlah instruksi, penggunaan memori, atau konsumsi daya.

2.7.6 Generasi Kode Target (Code Generation)

Fase terakhir memetakan instruksi IR yang telah dioptimasi ke set instruksi spesifik mesin target (misal: x86, ARM, MIPS). Fase ini harus membuat keputusan krusial mengenai alokasi register (*register allocation*) dan pemilihan instruksi (*instruction selection*) untuk memanfaatkan fitur spesifik prosesor secara maksimal. Output akhirnya adalah file assembly atau object file.



Gambar 2.5: Transformasi dari struktur pohon (AST) ke kode linear (IR) dan optimasi

2.8 Contoh Praktis: Alur Kompilasi Program

Untuk mengonkretkan teori fase-fase di atas, mari kita telusuri perjalanan satu baris kode C sederhana. Kita akan melihat bagaimana kode ini bermetamorfosis di setiap tahap.

Kode Sumber:

```
1 int sum = old_val + 10;
```

1. **Scanner (Lexical Analysis)**
   Scanner membaca aliran karakter dan memecahnya menjadi token:

- `<KEYWORD, "int">`
- `<ID, "sum">`
- `<OP, "=">`
- `<ID, "old_val">`
- `<OP, "+">`
- `<LITERAL, "10">`
- `<PUNCT, ";">`

2. **Parser (Syntax Analysis)**
   Parser menyusun token menjadi pohon sintaks. Ia mengenali pola deklarasi variabel:

```
Declaration
|-- Type: int
|-- Var: sum
`-- Init: Assignment
    |-- LHS: sum
    `-- RHS: BinaryOp (+)
        |-- Left: old_val
        `-- Right: 10
```

   Jika kita lupa menulis titik-koma (`;`), parser akan melaporkan *Syntax Error*.

3. **Semantic Analyzer**
   Di sini kompilator memeriksa konteks.

   - Apakah variabel `old_val` sudah dideklarasikan sebelumnya?
   - Apakah tipe data `old_val` kompatibel untuk dijumlahkan dengan integer `10`?

   Skenario Error: Jika `old_val` ternyata adalah string (`char*`), fase ini akan menghentikan proses dengan *Type Mismatch Error*, meskipun secara sintaksis kalimat tersebut benar.

4. **IR Generator & Optimizer**
   Pohon di atas diterjemahkan menjadi kode sementara (misal: Quadruples):

```
LOAD  t1, old_val  ; Muat nilai variabel ke temp
ADD   t2, t1, 10   ; Lakukan penjumlahan
STORE sum, t2      ; Simpan hasil ke sum
```

Optimasi: Jika `old_val` diketahui bernilai konstan 5, optimizer dapat langsung mengubahnya menjadi `STORE sum, 15`.

5. **Code Generator (x86-64)**
   IR akhirnya dipetakan ke register dan instruksi mesin nyata:

```
mov eax, [rbp-4]           ; Ambil old_val dari stack
add eax, 10                 ; Tambahkan 10
mov DWORD PTR [rbp-8], eax  ; Simpan ke lokasi sum
```

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

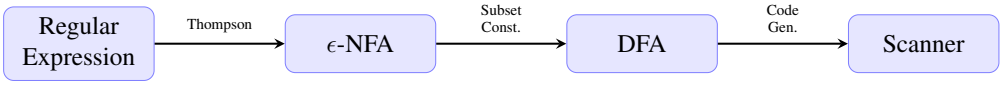
Lexical Analysis (atau ***Lexer/Scanner***) adalah garda terdepan dalam proses kompilasi. Tugas utamanya adalah membaca aliran karakter mentah dari kode sumber dan mengonversinya menjadi aliran *token* yang terstruktur. Bayangkan lexer sebagai seorang asisten yang membaca teks panjang, membuang spasi yang tidak perlu, dan mengelompokkan huruf-huruf menjadi kata-kata yang bermakna (seperti "run", "jump", "fast") sebelum diserahkan kepada ahli tata bahasa (Parser).

Mengapa memisahkan Lexer dari Parser? Alasan utamanya adalah efisiensi dan modularitas:

- **Efisiensi I/O**: Lexer menangani tugas berat membaca file karakter demi karakter. Dengan memisahkannya, kita bisa menerapkan teknik *buffering* yang agresif di level ini tanpa membebani parser.
- **Penyederhanaan Parser**: Parser tidak perlu khawatir tentang spasi atau komentar; ia hanya melihat aliran token bersih. Ini membuat tata bahasa (*grammar*) parser menjadi jauh lebih sederhana.

3.1.2 Alur Teoretis Lexical Analysis

Pembangunan lexer tidak dilakukan secara sembarangan, melainkan didasarkan pada fondasi matematika yang kokoh yaitu teori *Automata*. Alur transformasi dari spesifikasi pola hingga menjadi kode program yang berjalan adalah sebagai berikut:



Gambar 3.1: Alur konversi dari regular expression ke implementasi scanner

Kita mendeskripsikan pola token menggunakan *Regular Expression*, mengubahnya menjadi mesin abstrak non-deterministik (NFA), mengonversinya menjadi mesin deterministik (DFA) yang efisien, dan akhirnya menerjemahkannya menjadi kode (C/C++) [7].

3.1.3 Token, Lexeme, dan Pattern

Penting untuk membedakan tiga istilah ini:

- **Token**: Simbol abstrak yang mewakili jenis unit leksikal (misal: `Token::ID`, `Token::NUMBER`). Ini adalah output yang dilihat oleh parser.
- **Lexeme**: Deretan karakter aktual dalam kode sumber yang cocok dengan suatu pola token (misal: variabel `count`, angka `3.14`).
- **Attribute**: Karena banyak lexeme (seperti `10` dan `20`) bisa memiliki token yang sama (`INTEGER`), lexer perlu menyimpan informasi tambahan (nilai lexeme itu sendiri) agar fase semantik nanti tahu angka mana yang dimaksud.

3.2 Regular Expression

3.2.1 Definisi dan Keterbatasan

Regular expression (Regex) adalah notasi aljabar untuk mendeskripsikan himpunan string (*language*). Regex dipilih untuk analisis leksikal karena sifatnya yang deklaratif: kita cukup mendeskripsikan "seperti apa bentuk tokennya", bukan "bagaimana cara mencarinya".

Namun, Regex memiliki keterbatasan fundamental: ia **tidak memiliki memori**. Regex tidak bisa digunakan untuk mengenali struktur yang bersarang (*nested*) atau saling berpasangan, seperti:

- Menyeimbangkan kurung buka dan tutup: `(((...)))`.

- Mengenali string palindrome: `w w^R`.

Karena itulah Regex hanya digunakan untuk token linear sederhana, sedangkan struktur program yang kompleks ditangani oleh *Context-Free Grammar* di fase parsing.

3.2.2 Notasi dan Ekstensi Standar

Selain operator dasar (union, concat, kleene star), implementasi modern menyediakan "gula sintaksis" (*syntactic sugar*) untuk memudahkan penulisan:

Notasi	Arti	Ekuivalen Dasar
`[a-z]`	Character Class	$a\|b\|c\|...\|z$
`[a-zA-Z]`	Huruf apa saja	$a\|...\|z\|A\|...\|Z$
`\d`	Digit	`[0-9]`
`\w`	Word Character	`[a-zA-Z0-9_]`
`\s`	Whitespace	`[ \t\n\r]`
`.`	Any Character	Semua kecuali newline
`[^abc]`	Negasi	Karakter selain a, b, c

Tabel 3.1: Ekstensi notasi Regular Expression umum

3.2.3 Implementasi Leksikal dengan Regex

[8]

Dalam pembangunan kompilator, setiap elemen leksikal didefinisikan secara presisi:

1. **Identifier**: `[a-zA-Z_][a-zA-Z0-9_]*` — Harus dimulai huruf/_, boleh ada angka setelahnya.

2. **Integer**: `[0-9]+` — Satu digit atau lebih.

3. **Float**: `\d+\.\d+([eE][+-]?\d+)?` — Mendukung desimal dan notasi ilmiah.

4. **String**: `"([^"\\]|\\.)*"` — String dalam kutip, mendukung escape character.

3.3 Finite Automata

Finite Automata (FA) adalah mesin abstrak yang membaca string input dan memutuskan apakah string tersebut diterima (*accepted*) atau ditolak (*rejected*). FA adalah "mesin" yang menjalankan aturan Regular Expression.

3.3.1 NFA: The Parallel Multi-verse

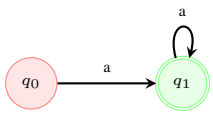
Nondeterministic Finite Automata (NFA) memiliki kemampuan "magis" untuk berada di beberapa state sekaligus. Jika ada dua transisi keluar dengan label yang sama (atau transisi ϵ), NFA seolah-olah membelah diri dan menelusuri semua jalur yang mungkin secara paralel. Jika salah satu "kloningan" mesin ini berhasil mencapai state akhir, maka input diterima.

- **Fleksibilitas**: Sangat mudah dibentuk langsung dari Regular Expression (misal: a|b langsung menjadi cabang).
- **Biaya**: Sulit diimplementasikan di komputer nyata karena komputer bekerja secara deterministik (serial). Simulasi NFA membutuhkan pelacakan himpunan state aktif yang kompleks.

3.3.2 DFA: The Realistic Machine

Deterministic Finite Automata (DFA) adalah mesin yang realistis. Untuk setiap input dan state saat ini, hanya ada **tepat satu** transisi ke state berikutnya. Tidak ada ambiguitas, tidak ada "tebakan".

- **Efisiensi**: Eksekusi DFA sangat cepat, linear terhadap panjang input ($O(n)$). Lexer komersial selalu menggunakan DFA (atau simulasi DFA) di balik layar.
- **Ukuran**: Jumlah state DFA bisa meledak secara eksponensial dibandingkan NFA asalnya, namun dalam praktik kompilator, ukurannya biasanya masih wajar.



DFA untuk $a+$ (menerima "a", "aa", ...)

Gambar 3.2: Contoh DFA sederhana yang deterministik

3.3.3 Konversi NFA ke DFA: Subset Construction

Karena komputer tidak bisa menjalankan NFA secara efisien, kita harus mengubahnya menjadi DFA. Algoritma *Subset Construction* bekerja dengan prinsip: "Setiap state di DFA mewakili himpunan state yang mungkin sedang aktif di NFA". Jadi, DFA mensimulasikan semua kemungkinan paralel NFA dalam satu langkah tunggal yang pasti.

3.4 Metode Konstruksi NFA: Algoritma Thompson

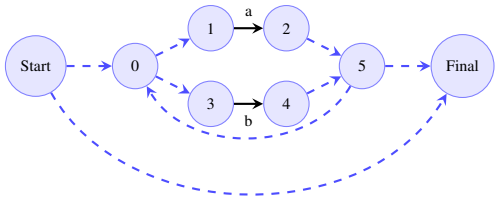
Algoritma Thompson menyediakan cara standar untuk mengubah pola Regular Expression menjadi NFA secara sistematis melalui template-template kecil yang digabungkan. Setiap konstruksi memiliki satu state awal dan satu state akhir, dan jumlah state hanya bertambah secara linear mengikuti panjang regex.

3.4.1 Template Dasar Thompson

1. **Literal ("a")**: Transisi langsung dari state s ke f dengan input 'a'.
2. **Concatenation** (RS): Menghubungkan state akhir NFA R ke state awal NFA S dengan transisi ϵ.
3. **Alternation** ($R|S$): Membuat state awal baru yang bercabang dengan transisi ϵ ke NFA R dan S, lalu state akhir keduanya digabung ke state akhir baru via ϵ.
4. **Kleene Closure** (R^*): Menambahkan loop ϵ dari akhir R ke awal R (pengulangan), serta jalur pintas ϵ dari awal R ke akhir R (skip).

Contoh Jejak Konstruksi untuk `(a|b)*` :

- **Langkah 1**: Buat NFA kecil untuk `a` dan `b`.
- **Langkah 2**: Gabungkan keduanya dengan template *Alternation* menjadi `(a|b)`. Ini melibatkan penambahan state awal baru yang membelah ke a dan b.
- **Langkah 3**: Bungkus hasilnya dengan template *Kleene Star*. Tambahkan transisi ϵ yang memungkinkan kita "melompat kembali" ke awal setelah selesai mengenali `(a|b)`, atau langsung "keluar" (match empty string).



Gambar 3.3: Visualisasi NFA untuk (a|b)*

3.5 Minimisasi DFA

Minimisasi DFA bertujuan menghasilkan automata dengan jumlah state paling sedikit tanpa mengubah bahasa yang dikenali. DFA minimal membuat implementasi lexer lebih ringkas dan efisien.

3.5.1 Langkah Umum Minimisasi

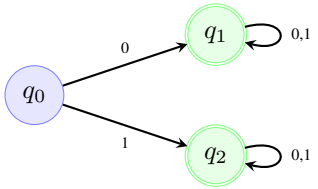
1. **Hapus state tak terjangkau**: Buang state yang tidak pernah dikunjungi dari start state.
2. **Partisi awal**: Pisahkan state final dan non-final karena keduanya pasti tidak ekuivalen.
3. **Refinement partisi**: Pecah kelompok state jika ada transisi yang menuju kelompok berbeda untuk simbol input yang sama.
4. **Gabungkan state ekuivalen**: State dalam partisi yang sama digabung menjadi satu state baru.

3.5.2 Intuisi Ekuivalensi State

Dua state dikatakan ekuivalen jika untuk setiap string input, keduanya selalu berakhir pada status terima/tolak yang sama. Algoritma minimisasi pada dasarnya mencari kelas-kelas ekuivalen ini melalui proses *partition refinement*.

3.5.3 Contoh DFA Sederhana

Gambar berikut menunjukkan DFA non-minimal dengan dua state final yang ekuivalen.



Gambar 3.4: DFA non-minimal: q_1 dan q_2 ekuivalen

3.5.4 Partisi dan Penggabungan

Iterasi	Partisi
P_0	$\{q_1, q_2\}$ (final), $\{q_0\}$ (non-final)
P_1	$\{q_1, q_2\}$, $\{q_0\}$ (tidak berubah)

Tabel 3.2: Refinement partisi untuk contoh DFA

Karena tidak ada pemisahan lebih lanjut, q_1 dan q_2 digabung menjadi satu state final.

3.6 Implementasi Lexer Hand-written

Meskipun alat bantu (*lexer generator*) seperti *re2c* [9] atau *RE/flex* [10] tersedia, menulis lexer secara manual memberikan kontrol penuh atas kinerja dan penanganan kesalahan.

3.6.1 Teknik Input Buffering

Membaca file satu karakter setiap kali (`fgetc`) merupakan operasi yang sangat lambat karena melibatkan *system call* ke OS. Solusi standar adalah menggunakan **Dual Buffer** dengan teknik **Sentinel**.

- **Buffer**: Kita memuat blok besar (misal 4KB) dari disk ke memori sekaligus.
- **Sentinel**: Alih-alih memeriksa apakah kita mencapai akhir buffer setiap kali maju satu langkah (yang membutuhkan dua perbandingan: `isEOF` dan `isEndBuf`), kita menaruh karakter spesial (`EOF sentinel`) di akhir buffer. Loop pembacaan bisa berjalan sangat cepat tanpa pengecekan batas buffer, sampai ia menabrak sentinel. Ini bisa meningkatkan kecepatan scanning hingga 30%.

3.6.2 Struktur Data Token

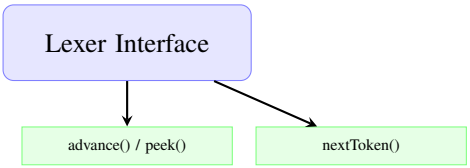
Token yang dihasilkan haruslah ringkas namun informatif:

```
enum class TokenType {
    ID, KW_IF, KW_ELSE, INT_LIT, OP_PLUS, EOF_TOKEN, ERROR
};

struct Token {
    TokenType type;
    std::string lexeme; // Teks asli ("count", "123")
    std::any value;     // Nilai semantik (123 untuk INT_LIT)
    int line, col;      // Lokasi untuk error reporting
};
```

3.6.3 Arsitektur Kelas Lexer

Kelas Lexer mempertahankan state posisi saat ini (`cursor`) dan menyediakan metode utama `nextToken()` yang dipanggil oleh parser.



Gambar 3.5: Komponen utama kelas Lexer

3.7 Logika State Machine Token

3.7.1 Prinsip Maximal Munch (Longest Match)

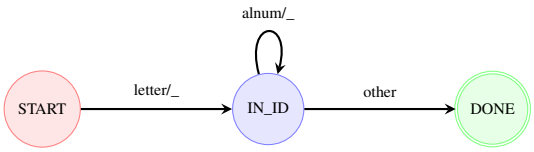
Dalam implementasi lexer, sering terjadi ambiguitas. Misalnya, input >= bisa dibaca sebagai > lalu =, atau sebagai satu token >=. Prinsip standar yang digunakan adalah **Maximal Munch**: *selalu ambil token terpanjang yang mungkin*.

- Jika input adalah >=, lexer akan melihat >. Jangan berhenti! Lihat karakter berikutnya =. Karena >= valid, ambil keduanya.
- Jika input adalah `>a`, lexer melihat >. Karakter berikutnya `a` tidak membentuk operator valid dengan >. Maka, kembalikan hanya > dan mundurkan kursor dari `a`.

3.7.2 Prinsip Prioritas: Keyword vs Identifier

Pola `if` cocok dengan aturan *Keyword IF*, tapi juga cocok dengan pola *Identifier*. Bagaimana membedakannya?

1. **Rule Order**: Dalam alat seperti Flex, aturan yang ditulis lebih awal menang. Tulis aturan `if` sebelum aturan Identifier.
2. **Lookup Table**: Dalam lexer manual, kita biasanya menganggap semuanya sebagai Identifier dulu. Setelah lexeme terkumpul ("if"), kita cek ke dalam tabel hash berisi daftar kata kunci. Jika ada, ubah tipe tokennya menjadi `KW_IF`. Ini lebih efisien daripada membuat state machine DFA yang sangat kompleks untuk setiap kata kunci.



Setelah DONE, cek HashMap Keywords

Gambar 3.6: State machine generik untuk Identifier/Keyword

3.7.3 State Machine untuk Angka

Menangani angka juga membutuhkan logika *lookahead*. Saat melihat titik (`.`), lexer harus memastikan karakter selanjutnya adalah digit sebelum memutuskannya sebagai `FLOAT_LITERAL`. Jika tidak (misal `1..10` di Pascal), titik tersebut mungkin token terpisah (*range operator*).

```
Token Lexer::scanNumber() {
    std::string text;
    while (isdigit(peek())) text += advance();

    if (peek() == '.' && isdigit(peekNext())) { // Lookahead 2 langkah
        text += advance(); // makan titik
        while (isdigit(peek())) text += advance();
        return Token(FLOAT, text);
    }
    return Token(INT, text);
}
```

3.8 Handling Komentar dan Kesalahan

3.8.1 Whitespace dan Pelacakan Posisi

Kompilator umumnya mengabaikan *whitespace* (spasi, tab, newline) kecuali pada bahasa sensitif-indentasi seperti Python. Namun, whitespace memiliki peran krusial: memperbarui penghitung baris (`line counter`). Informasi baris ini sangat vital untuk pesan kesalahan nanti.

```
void Lexer::skipWhitespace() {
    while (isspace(peek())) {
        if (peek() == '\n') {
            line++; col = 0; // Reset kolom
        } else {
            col++;
        }
        advance();
    }
}
```

3.8.2 Penanganan Komentar Bersarang

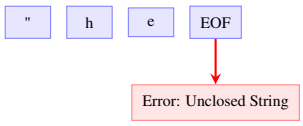
Komentar C standar (`/* ... */`) tidak mendukung *nesting*. Jika parser menemukan `/* A /* B */ C */`, ia akan berhenti pada `*/` pertama (setelah B), menyebabkan `C */` dianggap sebagai kode *syntax error*. Jika ingin mendukung *nested comments* (seperti di Swift), lexer harus menggunakan counter kedalaman.

3.8.3 Strategi Pemulihan Kesalahan Leksikal

Ketika lexer menemukan karakter ilegal (misal: `@` di C), ia tidak boleh langsung *crash*. Ada beberapa strategi pemulihan:

- **Panic Mode**: Abaikan karakter bermasalah dan coba lanjut memindai karakter berikutnya. Ini strategi paling umum.

- **Error Token**: Kembalikan token khusus `TK_ERROR` ke parser, biarkan parser yang memutuskan apakah akan berhenti atau mencoba pulih.
- **Deletion/Insertion**: Mencoba menghapus karakter aneh atau menyisipkan karakter yang hilang (misal: menutup string yang terbuka), tapi ini berisiko mengubah makna program.



Gambar 3.7: Ilustrasi unclosed string error

3.9 Integrasi dengan Parser

Tujuan akhir lexer adalah melayani parser. Oleh karena itu, antarmuka komunikasi keduanya harus disepakati.

3.9.1 Protokol Komunikasi Standard

Dalam ekosistem UNIX (Yacc/Bison), lexer dan parser berkomunikasi melalui variabel global:

1. **yylex()**: Fungsi utama yang dipanggil parser. Mengembalikan `int` yang merupakan ID Token (misal: 257 untuk `ID`, 258 untuk `INT`).
2. **yylval**: Variabel union global untuk menyimpan *atribut* token. Jika tokennya `INT`, `yylval.ival` diisi nilai integer-nya. Jika `ID`, `yylval.sval` diisi string namanya.
3. **yylloc**: Struktur untuk menyimpan lokasi (baris, kolom) token saat ini.

3.9.2 Best Practices Implementasi

- **Lookahead Minimal**: Usahakan lexer hanya butuh mengintip 1 karakter (`LL(1)`). Logika yang butuh lookahead jauh (misal membedakan deklarasi pointer vs perkalian di C++) sebaiknya ditunda hingga fase parsing.
- **Hindari State Kompleks**: Jangan menaruh logika sintaksis di lexer. Lexer tidak perlu tahu apakah `{` adalah awal fungsi atau awal blok. Ia hanya perlu melapor: "Ini kurung kurawal buka".
- **String Interning**: Untuk penghematan memori, semua identifier yang sama bisa menunjuk ke alamat string yang sama di memori (*string pool*).

3.10 Lexer Generator: Flex dan re2c

3.10.1 Flex: The Classic Table-Driven Generator

Flex (Fast Lexical Analyzer) adalah standar industri yang menghasilkan *State Machine* berbasis tabel.

- **Cara Kerja**: Flex mengonversi regex menjadi DFA, lalu menyimpannya sebagai array 2D integer besar (baris=state, kolom=input char).
- **Runtime**: Loop utama hanya berupa: `next_state = table[current_state][input_char]`.
- **Kelebihan**: Ukuran kode executable kecil, waktu kompilasi cepat.
- **Kekurangan**: Akses memori ke tabel bisa menyebabkan *cache miss* pada prosesor modern.

3.10.2 re2c: The High-Performance Code Generator

re2c mengambil pendekatan berbeda. Alih-alih tabel, ia menghasilkan kode *hard-coded goto* untuk setiap transisi.

- **Cara Kerja**: Mengubah DFA langsung menjadi sarang `if-else` dan `goto` dalam C++.
- **Runtime**: CPU dapat melakukan *branch prediction* dan kode instruksinya muat di *instruction cache*.
- **Performa**: Seringkali 2-3x lebih cepat dari Flex, digunakan oleh proyek performa tinggi seperti PHP dan Ninja Build.

Fitur	Hand-written	Table-Driven (Flex)	Direct-Coded (re2c)
Produktivitas	Rendah	Tinggi	Tinggi
Kecepatan Eksekusi	Tinggi	Sedang	Sangat Tinggi
Ukuran Kode Biner	Sedang	Kecil (tabel)	Besar (kode)
Fleksibilitas	Sangat Tinggi	Terbatas	Terbatas

Tabel 3.3: 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

4.1.1 Peran Parser

Parser adalah jantung dari front-end kompilator. Jika Lexer mengolah ”kata per kata”, Parser mengolah ”kalimat per kalimat” [11]. Parser mengubah deretan token datar menjadi struktur hierarkis (pohon) yang merepresentasikan struktur gramatikal program.

4.1.2 Context-Free Grammar (CFG)

Bahasa pemrograman umumnya didefinisikan menggunakan Context-Free Grammar (CFG). CFG cukup kuat untuk mengekspresikan struktur bersarang (*nested structures*) seperti kurung `(((...)))` dan blok `{...}` yang tidak bisa ditangani oleh Regular Expression [2].

Secara formal, CFG didefinisikan sebagai 4-tuple $G = (V, \Sigma, R, S)$:

- V (*Non-terminals*): Simbol variabel yang bisa diturunkan lebih lanjut (misal: *Statement*, *Expression*).
- Σ (*Terminals*): Simbol dasar atau token dari lexer (misal: `ID`, `IF`, `+`).
- R (*Production Rules*): Aturan substitusi berbentuk $A \rightarrow \alpha$, di mana $A \in V$ dan $\alpha \in (V \cup \Sigma)^*$.
- S (*Start Symbol*): Non-terminal awal dimulainya derivasi.

4.1.3 Derivasi: Dari Simbol ke String

Derivasi adalah proses penggantian non-terminal secara beruntun hingga menghasilkan string terminal. Contoh Grammar: $E \to E + E \mid \textbf{id}$ Derivasi string "**id + id**":

1. E (Start)
2. $\Rightarrow E + E$ (Gunakan aturan $E \to E + E$)
3. $\Rightarrow \textbf{id} + E$ (Ganti E pertama dengan **id**)
4. $\Rightarrow \textbf{id} + \textbf{id}$ (Ganti E kedua dengan **id**)

4.2 Top-Down Parsing dan LL(1)

4.2.1 Recursive Descent Parser

Metode ini paling intuitif untuk ditulis tangan. Idenya sederhana: *setiap non-terminal dalam grammar menjadi satu fungsi dalam kode program* [12]. Jika grammar memiliki aturan $S \to aBc$, maka fungsi `parseS()` akan memanggil `match('a')`, lalu `parseB()`, dan akhirnya `match('c')`.

4.2.2 Masalah pada Grammar Top-Down

Agar Recursive Descent bekerja, grammar harus memenuhi syarat **LL(1)**: membaca input dari Kiri (**L**eft), menghasilkan derivasi Kiri (**L**eftmost), dengan **1** token lookahead. Dua musuh utama LL(1) adalah:

1. Left Recursion

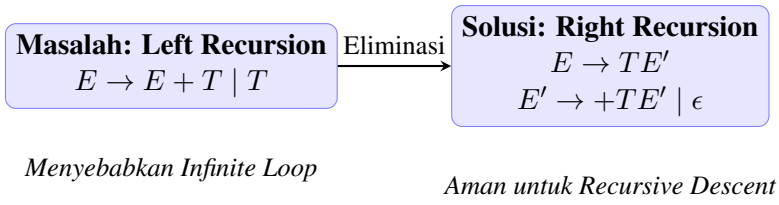
Aturan seperti $A \to A\alpha$ akan menyebabkan fungsi `parseA()` memanggil dirinya sendiri tanpa henti (*infinite loop*) sebelum membaca input apa pun. **Solusi:** Ubah menjadi *Right Recursion*.

$$A \to A\alpha \mid \beta \quad \Longrightarrow \quad A \to \beta A', \quad A' \to \alpha A' \mid \epsilon$$

2. Common Prefix (Butuh Left Factoring)

Jika parser melihat aturan $S \to \textbf{if} E \textbf{then} S \mid \textbf{if} E \textbf{then} S \textbf{else} S$, parser bingung memilih cabang mana saat melihat token **if**. **Solusi:** Faktorkan prefiks yang sama keluar.

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \quad \Longrightarrow \quad A \to \alpha A', \quad A' \to \beta_1 \mid \beta_2$$



Gambar 4.1: Transformasi Grammar untuk LL(1)

4.3 Struktur Derivasi dan Ambiguitas

4.3.1 Leftmost vs Rightmost Derivation

- **Leftmost Derivation**: Selalu memperluas non-terminal paling kiri. Ini yang dilakukan oleh parser *Top-Down* (LL). Parser "menebak" produksi apa yang dipakai sebelum melihat seluruh isi produksinya.
- **Rightmost Derivation**: Selalu memperluas non-terminal paling kanan. Jika urutannya dibalik (*Reverse Rightmost*), ini yang dilakukan parser *Bottom-Up* (LR). Parser menunggu sampai seluruh bagian produksi terlihat (*handle*) baru melakukan reduksi.

4.3.2 Bahaya Ambiguitas: The Dangling Else

Ambiguitas terjadi ketika satu string kode memiliki lebih dari satu pohon sintaks yang valid. Kasus klasik adalah *Dangling Else*:

```
1 if E1 then if E2 then S1 else S2
```

Apakah `else S2` milik `if E1` atau `if E2`?

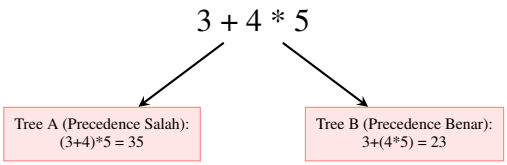
1. **Interpretasi 1**: `if E1 then (if E2 then S1 else S2)` (Else milik inner if).
2. **Interpretasi 2**: `if E1 then (if E2 then S1) else S2` (Else milik outer if).

Secara konvensi, bahasa pemrograman (C, Java, Pascal) memilih interpretasi 1 (*nearest-match*): else dipasangkan dengan if terdekat yang belum punya pasangan. Parser generator seperti Bison biasanya menyelesaikan konflik ini dengan aturan *Shift over Reduce*.

4.4 Parse Tree dan Abstract Syntax Tree (AST)

4.4.1 Parse Tree (Concrete Syntax Tree)

Parse Tree adalah representasi visual lengkap dari proses derivasi. Ia mencakup semua detail grammar, termasuk simbol-simbol non-terminal perantara yang mungkin tidak relevan untuk



Grammar ambigu $E \rightarrow E + E \mid E * E$ memungkinkan kedua struktur di atas.

Gambar 4.2: Ambiguitas pada ekspresi aritmatika

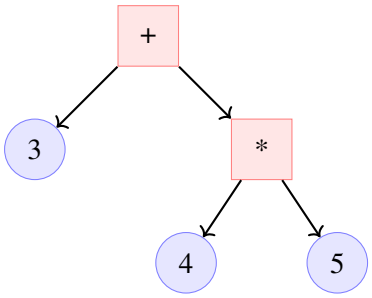
eksekusi program.

- **Kelebihan**: Merefleksikan grammar secara presisi.
- **Kekurangan**: Sangat boros memori dan sulit ditraversasi karena terlalu dalam.

4.4.2 Abstract Syntax Tree (AST)

AST adalah versi ringkas dan bersih dari Parse Tree. AST membuang semua token sintaksis yang tidak perlu (seperti kurung `(`, `)`, titik koma `;`, keyword `then`) dan hanya menyimpan struktur logis operasi.

- **Simpul Dalam**: Operator atau struktur kontrol (`+`, `if`, `while`).
- **Daun**: Operan atau nilai (`3.14`, `counter`).



Struktur Node:

```
struct Expr { ... };
struct BinOp : Expr {
 Expr* left;
 Token op;
 Expr* right;
};
```

Gambar 4.3: AST untuk ungkapan $3 + 4 * 5$

4.5 Integrasi Lexer dan Parser (Bison)

Dalam ekosistem GNU, **Bison** (Parser) adalah tuan dan **Flex** (Lexer) adalah pelayan. Parser meminta token, Lexer menyediakannya.

4.5.1 Mekanisme Komunikasi: yylval

Lexer dan Parser berbagi variabel global (atau anggota struct dalam mode *reentrant*) bernama `yylval`.

1. Lexer mencocokkan string "100".
2. Lexer mengonversi string "100" menjadi integer 100.
3. Lexer menyimpan nilai 100 ke dalam `yylval.ival`.
4. Lexer mengembalikan token ID `T_INT` ke parser.
5. Parser menerima token `T_INT`, dan mengambil nilai 100 dari `yylval` untuk membangun node AST.

```
// Di dalam file .l (Flex)
[0-9]+  { yylval.ival = atoi(yytext); return T_INT; }
[a-z]+  { yylval.sval = strdup(yytext); return T_ID; }
```

4.5.2 Struktur File Bison (.y)

File Bison memiliki tiga segmen yang dipisahkan oleh `%%`:

1. **Deklarasi**: Definisi token, tipe data union `yylval`, dan precedence.
2. **Rules**: Grammar dalam format BNF beserta *semantic actions* (kode C dalam `{ }`).
3. **User Code**: Fungsi `main()`, `yyerror()`, dll.

4.6 Implementasi: Hand-written Recursive Descent

Recursive Descent adalah teknik parsing paling populer untuk implementasi manual karena struktur kodenya sangat mirip dengan grammar EBNF-nya.

4.6.1 Pola Implementasi

Setiap aturan produksi diterjemahkan menjadi satu fungsi.

- **Pilihan** ($|$): Menjadi `if-else` atau `switch-case`.
- **Repetisi** ($*$ **atau** $+$): Menjadi `while` atau `do-while` loop.
- **Sequence**: Menjadi pemanggilan fungsi berurutan.

4.6.2 Pratt Parsing (Top-Down Operator Precedence)

Recursive Descent menjadi agak berbelit saat menangani ekspresi matematika dengan banyak level presedensi (misal: 10 level dari || hingga ()). Kita harus membuat 10 fungsi bersarang (`parseOr`, `parseAnd`, ..., `parsePrimary`).

Sebagai alternatif, teknik **Pratt Parsing** (digunakan di Python dan Rust) menggunakan tabel lookup untuk presedensi.

```
// Konsep Pratt Parsing
Expression parsePratt(int precedence) {
    Token t = advance();
    Expression left = prefixParselet[t.type].parse(t);
    while (precedence < getPrecedence(peek().type)) {
        t = advance();
        left = infixParselet[t.type].parse(left, t);
    }
    return left;
}
```

Teknik ini jauh lebih ringkas untuk ekspresi, namun Recursive Descent tetap lebih unggul untuk statement (`if`, `while`, `class`).

4.7 Precedence dan Error Recovery

4.7.1 Menangani Operator Precedence

Dalam *Recursive Descent*, precedence diatur secara implisit melalui struktur fungsi yang berlapis. Fungsi yang dipanggil paling dalam (misal: `parseFactor`) memiliki prioritas eksekusi paling tinggi (mengikat operan lebih kuat).

- **Lowest Precedence**: `parseExpression` (menangani `+ -`).
- **Medium Precedence**: `parseTerm` (menangani `* /`).
- **Highest Precedence**: `parseFactor` (menangani `( )`, angka, variabel).

4.7.2 Error Recovery: Panic Mode

Parser yang baik tidak boleh berhenti (*abort*) begitu menemukan satu kesalahan sintaks.

1. **Deteksi Error**: Parser menemukan token tak terduga (misal: `Expected ';'`).
2. **Masuk Mode Panik**: Parser membuang semua produksi yang sedang dikerjakan.
3. **Sinkronisasi**: Parser terus membuang token input (*eating tokens*) sampai menemukan token "jangkar" yang aman, biasanya titik koma (`;`) atau kurung kurawal tutup (`}`).

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

4.8 Bottom-Up Parsing: Shift-Reduce dan LR

4.8.1 Filosofi Bottom-Up

Jika Top-Down mencoba "menebak" struktur dari atas, Bottom-Up bekerja seperti menyusun puzzle dari kepingan kecil (token) menjadi gambar utuh (start symbol).

- **Shift**: Memindahkan token dari input ke stack parser.
- **Reduce**: Mengenali pola di puncak stack yang cocok dengan RHS aturan grammar, lalu menggantinya dengan LHS.

4.8.2 Contoh Trace Shift-Reduce

Grammar: $E \to E + E \mid$ **id** Input: **id** + **id** \$

Stack	**Input**	**Action**
\$	id + id \$	Shift **id**
\$ **id**	+ id \$	Reduce by $E \to$ **id**
\$ E	+ id \$	Shift **+**
\$ E **+**	id \$	Shift **id**
\$ E **+ id**	\$	Reduce by $E \to$ **id**
\$ E **+** E	\$	Reduce by $E \to E + E$
\$ E	\$	**Accept**

Tabel 4.1: Langkah-langkah Shift-Reduce Parsing

4.8.3 Keluarga LR Parser

1. **LR(0)**: Tidak melihat lookahead sama sekali. Hanya bisa menangani grammar sangat sederhana.
2. **SLR(1)**: Simple LR. Menggunakan himpunan *Follow* untuk menyelesaikan konflik Shift/Reduce dasar.

3. **LALR(1)**: Look-Ahead LR. Standar industri (Bison/Yacc). Menggabungkan state LR(1) yang memiliki inti (*core*) sama untuk menghemat memori.

4. **Canonical LR(1)**: Paling powerful, tapi tabel parsing-nya bisa sangat besar (ribuan state).

4.9 Parser Generator: Bison Deep Dive

4.9.1 Shift/Reduce Conflict

Konflik ini terjadi ketika parser bingung memilih antara menggeser (*shift*) token baru atau mereduksi (*reduce*) stack sekarang. Contoh klasik: `if E1 then if E2 then S1 else S2`. Di sini, parser memiliki `if E then S` di stack dan melihat `else`.

- **Shift**: Membawa `else` masuk (berarti `else` milik `if` dalam).
- **Reduce**: Mengubah `if E then S` menjadi *Stmt* (berarti `else` milik `if` luar).

Default Bison adalah **Shift** (yang benar untuk kasus ini).

4.9.2 Operator Precedence Declarations

Tanpa mengubah grammar menjadi rumit, kita bisa memberi tahu Bison prioritas operator:

```
%left '+' '-'    // Precedence rendah, asosiasi kiri
%left '*' '/'    // Precedence tinggi, asosiasi kiri
%right '^'       // Precedence lebih tinggi, asosiasi kanan
%nonassoc UMINUS // Untuk operator unary minus
```

Aturan yang dideklarasikan lebih bawah memiliki precedence lebih tinggi.

4.9.3 Error Handling dengan Token 'error'

Bison memiliki token spesial `error` untuk pemulihan.

```
stmt:
    expr ';'
  | error ';' { yyerror("Syntax error, skipping to semicolon"); yyerrok;
↪ }
;
```

Jika terjadi kesalahan dalam *stmt*, parser akan membuang token sampai menemukan titik koma, lalu mengeksekusi aksi pemulihan, dan melanjutkan parsing seolah-olah tidak ada masalah.

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

5.1.1 Definisi dan Fungsi

Symbol Table adalah struktur data sentral yang berfungsi sebagai kamus pintar bagi kompilator. Ia menjembatani jurang antara nama simbolik (`x`, `count`) yang dipahami manusia dengan entitas memori (`0x0040`, `R5`) yang dipahami mesin. Tanpa Symbol Table, kompilator ”buta” terhadap konteks variabel.

5.1.2 Informasi yang Disimpan (Attributes)

Entri dalam symbol table tidak sekadar menyimpan nama. Ia harus memuat metadata lengkap (*attributes*) yang relevan untuk semua fase kompilasi:

- **Basic Info**: Nama identifier (*Lexeme*).
- **Data Type**: Tipe primitif (`int`, `float`) atau komposit (`struct`, `array`, `pointer`).
- **Storage Class**: Sifat penyimpanan (`static`, `extern`, `const`).
- **Scope Level**: Kedalaman nesting (0 untuk global, 1 untuk main, dst).

- **Memory Offset**: Jarak relatif dari *Base Pointer* (penting untuk Code Generation).
- **Function Signature**: Jumlah dan tipe parameter (khusus untuk fungsi).

5.1.3 Fase Penggunaan

1. **Analysis**: Parser dan Semantic Analyzer mengisi tabel saat menemukan deklarasi.
2. **Synthesis**: Code Generator membaca tabel untuk menentukan alamat memori instruksi.

5.2 Struktur Data: Hash Table dan Scope Stack

5.2.1 Pilihan Struktur Data

Kinerja kompilator sangat bergantung pada kecepatan Symbol Table. Mengapa? Karena setiap kali parser menemukan identifier, ia harus melakukan *lookup*.

- **Linear List**: $O(N)$. Sangat lambat, hanya cocok untuk bahasa mainan.
- **Binary Search Tree (BST)**: $O(\log N)$. Cukup cepat, tapi butuh penyeimbangan (AVL/Red-Black) agar tidak terdegradasi menjadi linked list.
- **Hash Table**: $O(1)$ rata-rata. Ini adalah standar industri. Dengan fungsi hash yang baik, akses ke ribuan variabel tetap instan.

5.2.2 Manajemen Scope: The Cactus Stack

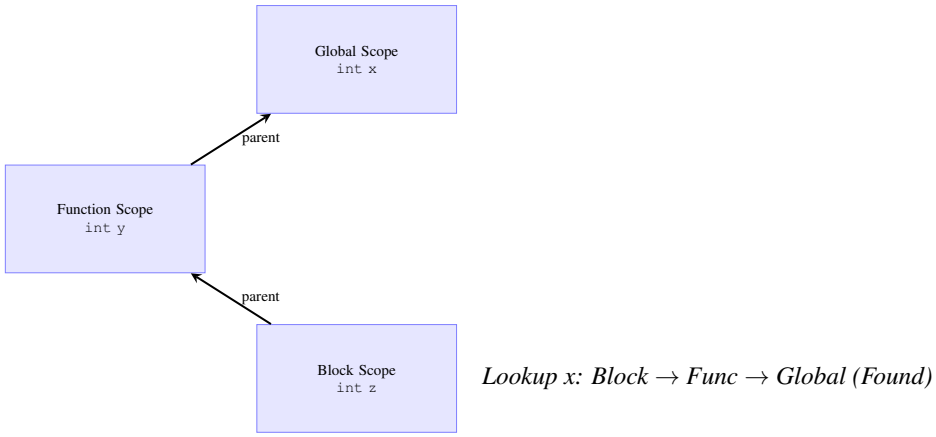
Bahasa modern mendukung *Nested Scopes* (blok di dalam blok). Struktur data yang paling tepat untuk ini adalah **Cactus Stack** (atau *Chained Symbol Tables*).

- Setiap scope memiliki Hash Table sendiri.
- Hash Table scope anak memiliki pointer `parent` ke Hash Table scope luar.
- Pencarian dimulai dari tabel saat ini. Jika tidak ketemu, lanjut ke `parent`, terus hingga `Global Scope` (yang parent-nya `NULL`).

5.3 Shadowing dan Resolusi Nama

5.3.1 Konsep Shadowing

Shadowing terjadi ketika deklarasi variabel di *nested scope* "menutupi" variabel dengan nama yang sama di *outer scope*. Kompilator perlu memberi peringatan (*warning*) jika shadowing tidak disengaja, namun harus mengizinkannya secara legal.



Gambar 5.1: Ilustrasi Chained Symbol Tables (Cactus Stack)

```
int x = 10; // Global x
void foo() {
    int x = 20; // x ini 'melindungi' foo dari akses ke global x
    {
        int x = 30; // x ini melindung blok ini
        print(x);   // Harus 30
    }
    print(x);       // Harus 20
}
```

5.3.2 Algoritma Resolusi Nama (Lookup)

Proses pencarian identifier dilakukan secara hierarkis:

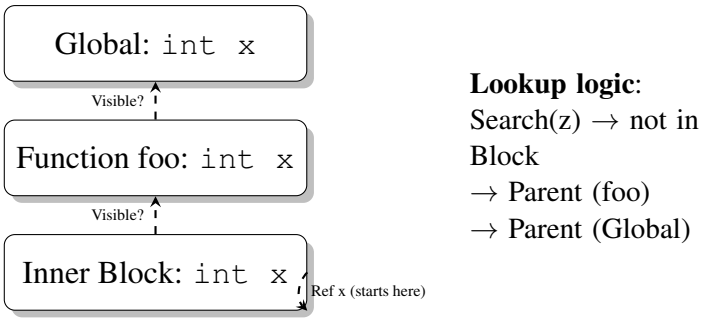
1. Mulai dari `current_scope`. Jika ditemukan, kembalikan object Symbol tersebut.
2. Jika tidak, pindah ke `current_scope->parent`.
3. Ulangi langkah 2 sampai menemukan `Global Scope`.
4. Jika sudah sampai puncak (Global) dan masih tidak ketemu, lemparkan error: `Undefined Variable 'x'`.

5.4 Scope Entry dan Exit: Static vs Dynamic

5.4.1 Static (Lexical) Scoping

Hampir semua bahasa modern (C, Java, Python) menggunakan **Static Scoping**.

- **Definisi**: Scope variabel ditentukan oleh struktur teks kode program saat kompilasi.



Gambar 5.2: Visualisasi Hierarki Scope untuk Resolusi Variabel

- **Sifat**: Identifier `x` di dalam fungsi `foo` akan selalu merujuk ke deklarasi `x` yang melingkupinya secara tekstual (misal di global), tidak peduli siapa yang memanggil `foo`.
- **Keuntungan**: Mudah dipahami oleh programmer hanya dengan membaca kode (*Predictable*).

5.4.2 Dynamic Scoping

Bahasa lama seperti Lisp awal atau Perl (opsional) menggunakan ini.

- **Definisi**: Scope ditentukan oleh urutan pemanggilan fungsi (*Call Stack*) saat *runtime*.
- **Sifat**: Identifier `x` di dalam `foo` akan mencari deklarasi `x` di fungsi pemanggil (`caller`), lalu pemanggilnya lagi, dst.
- **Kekurangan**: Sangat sulit di-debug karena nilai variabel bergantung pada *siapa* yang memanggil fungsi tersebut.

5.4.3 Manajemen Scope Stack

Saat parser masuk ke blok `{` (`enterScope`):

1. Buat objek `Scope` baru.
2. Set `newScope->parent = currentScope`.
3. Update `currentScope = newScope`.

Saat parser keluar dari blok `}` (`exitScope`):

1. Simpan pointer `temp = currentScope`.
2. Update `currentScope = currentScope->parent`.
3. Hapus `temp` (kecuali jika bahasa mendukung *closure*, maka scope ini harus disimpan di heap).

5.5 Implementasi Symbol Table yang Lengkap

5.5.1 Arsitektur Kelas SymbolTable

Berikut adalah implementasi minimalis namun fungsional untuk Symbol Table dengan dukungan *nested scopes* menggunakan C++.

```cpp
#include <unordered_map>
#include <string>
#include <vector>
#include <memory>

struct Symbol {
    std::string name;
    std::string type;
    int offset; // Untuk Code Generation
};

class Scope {
public:
    std::unordered_map<std::string, std::shared_ptr<Symbol>> symbols;
    Scope* parent;

    Scope(Scope* p) : parent(p) {}

    void insert(std::string name, std::shared_ptr<Symbol> sym) {
        symbols[name] = sym;
    }

    std::shared_ptr<Symbol> lookup(std::string name) {
        if (symbols.count(name)) return symbols[name];
        if (parent) return parent->lookup(name);
        return nullptr;
    }
};

class SymbolTable {
    Scope* currentScope;
public:
    SymbolTable() { currentScope = new Scope(nullptr); } // Global Scope

    void enterScope() {
        currentScope = new Scope(currentScope);
    }

    void exitScope() {
        if (currentScope->parent) {
            Scope* temp = currentScope;
            currentScope = currentScope->parent;
            delete temp;
        }
    }

    void addSymbol(std::string name, std::string type) {
```

```
48          auto sym = std::make_shared<Symbol>();
49          sym->name = name;
50          sym->type = type;
51          currentScope->insert(name, sym);
52      }
53  };
```

5.5.2 Integrasi dalam Parser

Di dalam parser (misal: Recursive Descent), panggilan ke `enterScope` dan `exitScope` disisipkan pada grammar *Block*:

```
1  void Parser::parseBlock() {
2      match('{');
3      symbolTable.enterScope();   // Buat scope baru
4      parseDeclarations();        // Isi tabel dengan variabel lokal
5      parseStatements();          // Gunakan variabel (lookup)
6      symbolTable.exitScope();    // Hapus scope saat keluar
7      match('}');
8  }
```

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
- **Sub-CPMK 3.4:** Menerapkan attribute grammar (synthesized/inherited) dan syntax-directed definition (SDD) atau translation schemes

6.1 Abstract Syntax Tree (AST) Deep Dive

6.1.1 Filosofi Desain AST

AST adalah versi ”bersih” dari Parse Tree. Jika Parse Tree merekam *bagaimana* program diturunkan dari grammar (termasuk token-token sintaksis seperti `;` atau `()`), AST hanya merekam *apa* isi program tersebut.

- **Compact**: Tidak ada node untuk token pungtuasi.
- **Heterogen**: Setiap jenis node (misal `IfStmt`, `BinaryExpr`) bisa memiliki struktur C++ class yang berbeda, tidak harus seragam.

6.1.2 Implementasi Titik Temu: The Visitor Pattern

Tantangan utama AST adalah: "Bagaimana kita menambahkan operasi baru (seperti Type Checking, Code Gen, Pretty Print) tanpa mengacak-acak definisi class AST?" Solusinya adalah **Visitor Pattern**.

```
1 // Abstract Base Class
2 struct Node {
3     virtual void accept(Visitor* v) = 0;
```

```
};

// Concrete Node
struct BinaryExpr : Node {
    Node *left, *right;
    Token op;
    void accept(Visitor* v) override { v->visitBinary(this); }
};

// Visitor Interface
struct Visitor {
    virtual void visitBinary(BinaryExpr* node) = 0;
    virtual void visitNumber(NumberExpr* node) = 0;
};
```

Dengan pola ini, logika semantik terisolasi dalam class `SemanticAnalyzer` yang mewarisi `Visitor`, menjaga kode AST tetap bersih.

6.2 Sistem Tipe dan Type Checking

6.2.1 Formalisme Sistem Tipe

Dalam teori bahasa pemrograman, aturan tipe sering dituliskan menggunakan *Inference Rules*. Notasi $\Gamma \vdash e : T$ dibaca "Dalam konteks Γ, ekspresi e memiliki tipe T".

Contoh aturan untuk penjumlahan integer:

$$\frac{\Gamma \vdash e_1 : \textbf{int} \quad \Gamma \vdash e_2 : \textbf{int}}{\Gamma \vdash e_1 + e_2 : \textbf{int}}$$

Artinya: "Jika e_1 bertipe int DAN e_2 bertipe int, MAKA $e_1 + e_2$ bertipe int".

6.2.2 Type Checking vs Type Inference

- **Type Checking**: Programmer menulis tipe eksplisit (`int x = 5;`), kompilator memverifikasi kebenarannya.
- **Type Inference**: Programmer tidak menulis tipe (`auto x = 5;`), kompilator *menyimpulkan* tipe berdasarkan nilai inisialisasi ($5 \rightarrow$ `int`, maka $x \rightarrow$ `int`).

6.2.3 Static vs Dynamic Typing

- **Static Typing** (C, Java): Tipe variabel diketahui saat *compile-time* dan tidak berubah. Lebih aman dan cepat dieksekusi.
- **Dynamic Typing** (Python, JS): Tipe variabel ditentukan saat *run-time* berdasarkan nilai yang saat itu disimpannya. Lebih fleksibel tapi rentan *runtime error*.

6.3 Analisis Kontekstual dan Validasi

Selain tipe yang benar, kompilator harus memastikan validitas kontekstual lainnya:

1. **Control Flow**: `break` dan `continue` hanya boleh di dalam *loop*.
2. **Scope**: Identifier harus dideklarasikan sebelum digunakan.
3. **Const Correctness**: Variabel `const` tidak boleh diubah (assignment l-value check).

6.3.1 Implicit Type Conversion (Coercion)

Apakah `3 + 4.5` valid? Dalam banyak bahasa, ya. Kompilator secara otomatis memasukkan operasi konversi tipe (*cast*) pada AST:

$$3 \text{ (int)} + 4.5 \text{ (float)} \longrightarrow \text{to\_float}(3) + 4.5$$

Aturan ini disebut *Promotion*: tipe yang lebih ”kecil” (int) dikonversi ke tipe yang lebih ”besar”/presisi (float) untuk menghindari kehilangan data.

6.3.2 Error Recovery: The Poison Pili

Bagaimana jika `x` tidak dideklarasikan dalam `x + 5`?

1. Laporkan error: ”Undeclared identifier ’x”’.
2. Tandai node `x` sebagai **Poison** (atau `ErrorType`).
3. Saat mengecek `Poison + 5`, hasilnya juga **Poison**.
4. Jangan laporkan error lagi untuk operasi yang melibatkan Poison. Ini mencegah *cascade error messages* yang membingungkan programmer.

6.4 Attribute Grammar

Attribute Grammar memperluas CFG dengan menambahkan atribut dan aturan semantik pada setiap produksi. Dengan cara ini, struktur sintaksis dapat membawa informasi semantik seperti nilai, tipe, atau lokasi memori.

6.4.1 Synthesized Attributes

Synthesized attribute dihitung dari anak (child) ke parent dalam pohon sintaks. Contoh umum: menghitung nilai ekspresi aritmatika atau menentukan tipe hasil operasi berdasarkan tipe operand.

6.4.2 Inherited Attributes

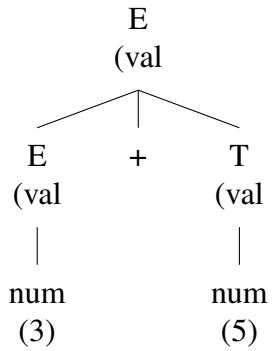
Inherited attribute diteruskan dari parent atau saudara (sibling) untuk memberikan konteks. Contoh umum: memberi informasi tipe variabel pada deklarasi atau memvalidasi scope.

6.4.3 Contoh Sederhana

Misalkan grammar: $E \rightarrow E + T \mid T$. Kita dapat mendefinisikan atribut `val` untuk menghitung nilai ekspresi:

$$E.val = E_1.val + T.val$$

Aturan seperti ini memungkinkan parser sekaligus membangun struktur dan menghitung makna program.



Gambar 6.1: Contoh atribut `val` pada pohon ekspresi

6.4.4 Aturan Atribut Ringkas

```
// Synthesized attribute
E.val = E1.val + T.val;

// Inherited attribute (contoh konteks)
T.inh = E.inh;
```

6.5 Syntax-Directed Definition dan Translation Schemes

6.5.1 Syntax-Directed Definition (SDD)

SDD adalah kombinasi grammar dengan atribut dan aturan semantik. Setiap produksi memiliki aturan yang menjelaskan bagaimana atribut dihitung. SDD menjembatani struktur sintaks dengan makna yang ingin dihasilkan, seperti tipe ekspresi atau kode intermediate.

6.5.2 Translation Schemes

Translation scheme menempatkan aksi semantik langsung di dalam produksi grammar sebagai *semantic actions*. Ini memudahkan implementasi karena aksi dapat dijalankan saat parsing berlangsung.

```
// Contoh skema sederhana untuk ekspresi penjumlahan
E -> E1 + T    { E.val = E1.val + T.val; }
T -> num       { T.val = num.lexval; }
```

SDD lebih bersifat deklaratif, sedangkan translation scheme lebih operasional. Keduanya digunakan untuk menghubungkan parsing dengan analisis semantik atau generasi kode.

Aspek	**SDD**	**Translation Scheme**
Sifat	Deklaratif, fokus pada aturan atribut	Operasional, aksi disisipkan di grammar
Waktu eksekusi	Dihitung berdasarkan dependensi atribut	Dieksekusi saat parsing berlangsung
Kegunaan	Spesifikasi semantik	Implementasi semantik praktis

Tabel 6.1: Perbandingan SDD dan Translation Schemes

6.5.3 Contoh SDD untuk Type Checking

```
// Contoh SDD untuk pengecekan tipe
E -> E1 + T {
  if (E1.type == T.type) E.type = E1.type;
  else E.type = error;
}
```

6.6 Praktikum: Implementasi Type Checker

Kita akan menggunakan **Visitor Pattern** untuk memisahkan logika pengecekan tipe dari struktur AST.

6.6.1 TypeChecker Visitor

```
class TypeChecker : public Visitor {
    Scope* currentScope;

public:
    void visitBinary(BinaryExpr* node) override {
        node->left->accept(this);  // Cek anak kiri
        node->right->accept(this); // Cek anak kanan

        Type leftType = node->left->type;
        Type rightType = node->right->type;
```

```
11
12          if (leftType != rightType) {
13              // Coba coercion
14              if (canPromote(leftType, rightType)) {
15                  insertCast(node->left, rightType);
16              } else {
17                  error(node->op.line, "Tipe tidak kompatibel");
18                  node->type = Type::Error; // Poisoning
19                  return;
20              }
21          }
22          node->type = leftType; // Propagasi tipe ke atas
23      }
24  };
```

6.6.2 Integrasi Symbol Table

Saat traversal, TypeChecker Visitor juga harus memelihara `SymbolTable` persis seperti Parser, agar bisa memvalidasi variabel:

- `visitBlock`: Panggil `enterScope()`, traverse anak-anak, lalu `exitScope()`.
- `visitVarDecl`: Masukkan variabel ke scope saat ini.
- `visitVariable`: Lookup nama variabel di scope. Jika tidak ketemu, set tipe ke `Error`.

6.7 Kesimpulan Analisis Semantik

Setelah fase semantik, kompilator memiliki dua aset utama:

1. **Annotated AST**: Pohon sintaks yang dilengkapi *tipe* dan konversi yang diperlukan (seperti `int-to-float`).
2. **Validated Symbol Table**: Tabel yang terisi penuh dengan informasi tentang fungsi, variabel, dan scope, yang siap digunakan oleh Code Generator untuk menghitung offset memori.

6.7.1 Menuju Intermediate Representation

Langkah selanjutnya (Chapter 7) adalah mengubah AST yang kaya makna ini menjadi bentuk yang lebih rendah tingkatannya (*Intermediate Representation*), seperti *Three-Address Code* (TAC), yang lebih dekat dengan mesin namun masih independen dari arsitektur CPU target.

Aktivitas Pembelajaran

1. **Semantic Rules**: Definisikan semantic rules untuk bahasa sederhana.
2. **Attribute Grammar**: Rancang attribute grammar untuk ekspresi aritmatika sederhana.
3. **SDD/Translation Scheme**: Susun aturan SDD untuk menghasilkan kode intermediate dasar.
4. **AST Builder**: Implementasikan AST construction dari parse tree.
5. **Type Checker**: Bangun type checker dengan error reporting informatif.
6. **Error Recovery**: Implementasikan error recovery untuk semantic errors.
7. **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. Rancang attribute grammar untuk ekspresi `E -> E + T | T`!
4. Tuliskan skema SDD untuk menghitung nilai ekspresi aritmatika!
5. Implementasikan type checker untuk expressions dengan multiple types!
6. Desain error messages yang informatif untuk berbagai semantic errors!
7. Analisis trade-off antara strict vs lenient semantic checking!
8. **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 merancang attribute grammar (synthesized/inherited)

☐ Saya dapat menyusun SDD atau translation schemes

☐ 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 attribute grammar, 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
- Attribute grammar memperkaya CFG dengan aturan semantik
- 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 salah satu bentuk Intermediate Representation (IR) yang paling populer dalam desain kompilator. Nama ini merujuk pada format instruksinya yang secara formal dibatasi memiliki maksimal tiga buah alamat (*addresses*) atau operan: satu tujuan (*result*) dan dua sumber (*arguments*).

7.1.1 Mengapa Kita Memerlukan IR?

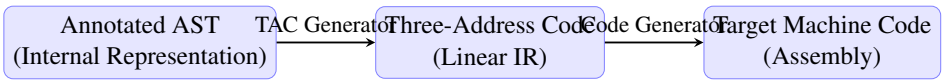
IR seperti TAC berfungsi sebagai jembatan penting antara *front-end* (yang memahami bahasa manusia) dan *back-end* (yang memahami mesin spesifik). Keuntungannya meliputi:

- **Retargetability**: Kita bisa menargetkan arsitektur CPU yang berbeda (x86, ARM, RISC-V) cukup dengan mengganti *back-end*, tanpa menyentuh *front-end*.
- **Machine-Independent Optimization**: Banyak optimasi seperti *Constant Folding* atau *Common Subexpression Elimination* jauh lebih mudah dilakukan pada level IR daripada pada kode mesin mentah.
- **Simplification**: Mengecilkan struktur program yang kompleks (seperti nested loops atau ekspresi rumit) menjadi urutan instruksi linier yang sederhana.

7.1.2 Format Instruksi dan Variabel Temporary

Format umum: `result = arg1 op arg2`. Ekspresi kompleks seperti `x + y * z / w` tidak bisa ditulis langsung dalam satu baris TAC. Kompilator mendekomposisinya menjadi urutan langkah kecil menggunakan variabel *temporary* (`t1, t2, ...`):

```
t1 = y * z
t2 = t1 / w
t3 = x + t2
```



Gambar 7.1: Posisi TAC dalam Alur Kompilasi

7.2 Representasi: Quadruples, Triples, dan Indirect Triples

Ada tiga cara utama untuk merepresentasikan instruksi TAC dalam struktur data memori. Pemilihan representasi ini sangat mempengaruhi performa fase optimasi.

7.2.1 1. Quadruples

Setiap instruksi disimpan dalam objek atau struct dengan empat *fields*: `(op, arg1, arg2, result)`.

- **Kelebihan**: Sangat fleksibel untuk optimasi. Kita bisa memindahkan atau menghapus instruksi tanpa merusak referensi instruksi lainnya karena tujuan (*result*) ditulis secara eksplisit sebagai nama variabel temporary.
- **Kekurangan**: Memerlukan memori lebih banyak untuk menyimpan nama variabel temporary di setiap entri.

7.2.2 2. Triples

Struktur ini hanya memiliki tiga *fields*: `(op, arg1, arg2)`. Hasil dari operasi tidak diberi nama variabel, melainkan dirujuk menggunakan ID atau indeks instruksi tersebut.

- **Kelebihan**: Lebih hemat memori karena tidak ada field *result*.
- **Kekurangan**: Sangat sulit untuk dioptimasi. Jika kita memindahkan baris #10 ke baris #15, semua instruksi lain yang merujuk pada hasil baris #10 (menggunakan indeks (10)) harus diperbarui secara manual.

7.2.3 3. Indirect Triples

Merupakan pengembangan dari Triples. Kita menyimpan Triples di satu tempat, dan memiliki array tambahan yang berisi *pointers* ke Triples tersebut dalam urutan eksekusi yang diinginkan.

- **Kelebihan**: Mendapatkan efisiensi memori Triples namun tetap mudah dioptimasi seperti Quadruples. Jika kita ingin menukar urutan kode, kita cukup menukar pointer di array tambahan tersebut.

Fitur	Quadruples	Triples	Indirect Triples
Field Result	Eksplisit	Implisit (Indeks)	Implisit (Indeks)
Optimasi	Sangat Mudah	Sulit	Mudah (via Pointer)
Konsumsi Memori	Tinggi	Rendah	Menengah
Standar Industri	Tinggi	Rendah	Menengah

Tabel 7.1: Perbandingan Representasi TAC

Gambar 7.2: Perbandingan Struktur Data Quadruples vs Triples

7.3 Postfix Notation

Postfix notation (Reverse Polish Notation) menuliskan operator setelah operand. Representasi ini mempermudah evaluasi dengan stack dan sering digunakan sebagai jembatan menuju three-address code.

7.3.1 Contoh Konversi

Ekspresi infix:

```
(a + b) * c
```

Postfix:

```
a b + c *
```

7.3.2 Evaluasi dengan Stack

Setiap operand didorong ke stack; saat operator ditemui, dua operand teratas di-pop, dioperasikan, lalu hasilnya di-push kembali. Model ini sejalan dengan proses pembentukan kode intermediate.

7.3.3 Keterkaitan dengan IR

Postfix notation mempermudah pembuatan TAC karena urutan operasinya sudah eksplisit dan tidak memerlukan penanganan prioritas operator secara khusus.

7.3.4 Contoh Evaluasi dengan Stack

Token	Stack Sebelum	Stack Sesudah
`a`	`[]`	`[a]`
`b`	`[a]`	`[a, b]`
`+`	`[a, b]`	`[t1]`
`c`	`[t1]`	`[t1, c]`
`*`	`[t1, c]`	`[t2]`

Tabel 7.2: Evaluasi postfix `a b + c *` dengan stack

7.3.5 Algoritma Konversi Infix ke Postfix

```
// Shunting-yard (ringkas)
for (token in input) {
  if (operand) output.push(token);
  else if (token == '(') opstack.push(token);
  else if (token == ')') {
    while (opstack.top() != '(') output.push(opstack.pop());
    opstack.pop(); // buang '('
  } else { // operator
    while (!opstack.empty() &&
           prec(opstack.top()) >= prec(token)) {
      output.push(opstack.pop());
    }
    opstack.push(token);
  }
}
while (!opstack.empty()) output.push(opstack.pop());
```

7.4 Translasi Ekspresi Boolean dan Backpatching

Penerjemahan ekspresi boolean (seperti `a < b || c > d`) seringkali memerlukan teknik khusus karena sifatnya yang mempengaruhi alur kontrol program, terutama jika bahasa mendukung *Short-Circuit Evaluation*.

7.4.1 Short-Circuit Evaluation

Kompilator tidak perlu mengevaluasi seluruh ekspresi jika hasil akhirnya sudah pasti.

- Pada `A || B`, jika `A` benar, maka `B` tidak perlu dieksekusi.
- Pada `A && B`, jika `A` salah, maka `B` diabaikan.

Ini diterjemahkan ke dalam TAC sebagai serangkaian *conditional jumps*.

7.4.2 Teknik Backpatching

Saat membangkitkan kode dalam satu fase (*one-pass*), seringkali kita belum tahu target alamat dari sebuah instruksi `goto`. Misalnya, saat melihat `if (cond)`, kita tahu harus lompat jika salah, tapi kita belum tahu baris mana yang merupakan akhir dari blok `if` tersebut. **Backpatching** menyelesaikan ini dengan meninggalkan target alamat kosong untuk sementara, lalu ”menambalnya” kemudian.

Fungsi utama dalam backpatching:

1. **makelist(i)**: Membuat list baru berisi indeks instruksi ke-i.
2. **merge(p1, p2)**: Menggabungkan dua list jump menjadi satu.
3. **backpatch(p, target)**: Mengisi semua instruksi jump dalam list p dengan alamat *target*.

```
// Contoh manipulasi list jump
struct BoolResult {
    list<int> truelist;  // Instruksi jump jika benar
    list<int> falselist; // Instruksi jump jika salah
};

BoolResult translateBoolean(ASTNode* node) {
    if (node->op == OR) {
        BoolResult left = translateBoolean(node->left);
        // Tandai titik masuk untuk evaluasi bagian kanan
        int M = nextInstructionID();
        backpatch(left.falselist, M);

        BoolResult right = translateBoolean(node->right);
        return { merge(left.truelist, right.truelist), right.falselist };
    }
}
```

Dengan teknik ini, list `truelist` dan `falselist` diteruskan ke atas hingga akhirnya di-backpatch oleh struktur kontrol (seperti `if` atau `while`).

7.5 Struktur Kontrol: Label dan Jump Alternatif

Penerjemahan struktur kontrol (*If*, *While*, *For*) menggunakan teknik backpatching menghasilkan kode TAC yang lebih efisien karena meminimalisir jumlah instruksi `goto` yang tidak perlu.

7.5.1 1. Translasi If-Then-Else

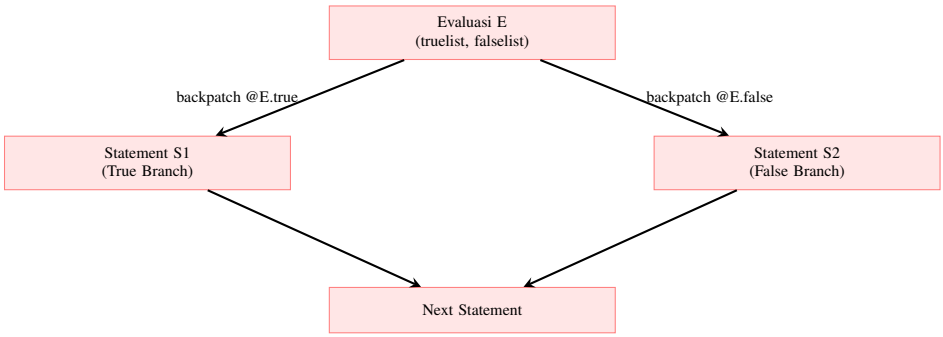
Skema translasi untuk statement `if (`E`)` S_1 `else` S_2:

1. Evaluasi ekspresi boolean E. Hasilnya adalah `truelist` dan `falselist`.
2. `backpatch(E.truelist, M1)`, di mana $M1$ adalah indeks awal kode untuk S_1.
3. `backpatch(E.falselist, M2)`, di mana $M2$ adalah indeks awal kode untuk S_2.
4. Tambahkan instruksi `goto` di akhir S_1 untuk melompati S_2. Alamat ini disimpan dalam list `nextlist`.

7.5.2 2. Translasi While Loop

Untuk statement `while (`M_1 E`)` M_2 S_1:

1. Simpan indeks M_1 (awal pengecekan kondisi).
2. Evaluasi E, menghasilkan `truelist` dan `falselist`.
3. `backpatch(E.truelist, M2)`, di mana $M2$ adalah awal tubuh loop S_1.
4. Di akhir S_1, tambahkan `goto M1` untuk pengulangan.
5. Hasil akhir dari statement `while` adalah `E.falselist` (kontrol keluar lewat sini).



Gambar 7.3: Alur Backpatching pada Struktur Kontrol If-Else

7.5.3 Tabel Simbol dan Label

Kompilator menyimpan pemetaan label ke alamat memori instruksi dalam tabel khusus. Ini memungkinkan optimasi *Jump-to-Jump* (menghapus lompatan yang menuju ke instruksi lompatan lain).

7.6 Translasi Array dan Pemanggilan Fungsi

7.6.1 1. Alamat Array Multidimensi

Mengakses elemen array seperti `A[i][j]` memerlukan perhitungan alamat memori pada saat *runtime*. Untuk array 2D berukuran $M \times N$ dengan urutan *Row-Major* (baris demi baris):

1. **Rumus Dasar**: $Address(A[i][j]) = Base_A + ((i \times N) + j) \times w$
2. **TAC Decompositon**:

```
t1 = i * N
t2 = t1 + j
t3 = t2 * w
t4 = base_A + t3
x = *t4  // Load value dari alamat t4
```

Di mana N adalah jumlah kolom dan w adalah ukuran tipe data (misal 4 byte untuk `int`). Perhitungan ini menjadi lebih kompleks untuk array 3D atau lebih, namun pola dekomposisi TAC tetap sama: hitung indeks linear, kalikan ukuran, tambahkan basis.

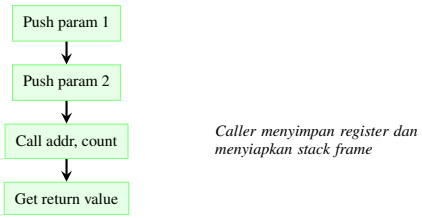
7.6.2 2. Pemanggilan Fungsi (*Function Calls*)

Translasi fungsi dalam TAC menggunakan instruksi `param`, `call`, dan `return`. Skema untuk `x = f(a, b+c):`

```
t1 = b + c
param a
param t1
t2 = call f, 2  // '2' adalah jumlah argumen
x = t2
```

7.6.3 Struktur Akhir TAC

Hasil akhir dari Chapter 7 adalah deretan instruksi TAC yang merepresentasikan logika program secara utuh. Kode ini kini siap untuk dioptimasi (Chapter 8) sebelum akhirnya diubah menjadi instruksi mesin yang sebenarnya.



Gambar 7.4: Urutan Operasi TAC untuk Pemanggilan Prosedur

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 unit atomik terkecil dalam analisis optimasi kompilator. Ia merupakan sekumpulan instruksi yang dieksekusi sebagai satu kesatuan: jika instruksi pertama dieksekusi, maka semua instruksi di bawahnya pasti dieksekusi hingga akhir blok.

8.1.1 Definisi dan Sifat Utama

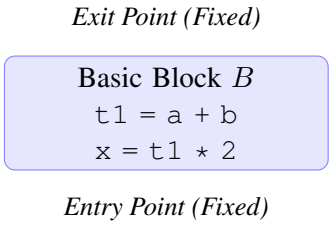
Sebuah blok instruksi disebut sebagai *Basic Block* jika memenuhi kriteria berikut:

- **Single Entry**: Kontrol hanya bisa masuk ke blok melalui instruksi pertama. Tidak ada instruksi di tengah blok yang boleh menjadi target lompatan (*jump label*) dari luar.
- **Single Exit**: Kontrol hanya bisa keluar dari blok melalui instruksi terakhir. Instruksi di tengah blok tidak boleh berupa *jump* atau *branch*.
- **High Cohesion**: Semua instruksi di dalamnya bersifat atomik dalam konteks kontrol alur.

8.1.2 Variabel Live on Exit

Dalam analisis blok, sangat penting untuk mengetahui variabel mana saja yang nilainya masih akan digunakan oleh blok lain setelah blok saat ini selesai dieksekusi. Variabel ini disebut **Live on Exit**.

- Jika sebuah variabel adalah *dead* (tidak live), maka instruksi penugasan terakhir ke variabel tersebut di dalam blok dapat dihapus karena hasilnya tidak akan pernah dibaca lagi.



Gambar 8.1: Abstraksi Basic Block sebagai Unit Terisolasi

8.2 Algoritma Identifikasi Leader

Proses dekomposisi kode TAC menjadi sekumpulan *Basic Blocks* diawali dengan menentukan **Leaders**.

8.2.1 Prosedur Penentuan Leader

Instruksi dalam deretan kode antara adalah *leader* jika memenuhi salah satu aturan berikut:

1. **Rule 1**: Instruksi pertama dalam kode program.
2. **Rule 2**: Instruksi yang merupakan target (*destination*) dari perintah lompatan (`goto`, `if...goto`).
3. **Rule 3**: Instruksi yang berada tepat setelah perintah lompatan (`goto`, `if...goto`).

8.2.2 Contoh Trace Identifikasi

Program TAC untuk menghitung faktorial:

```
1 (1) i = 1
2 (2) fact = 1
3 (3) if i > n goto (7)
4 (4) t1 = fact * i
5 (5) fact = t1
6 (6) i = i + 1
7 (7) goto (3)
8 (8) return fact
```

Leader Analysis:

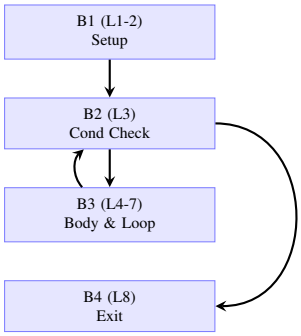
- **Line 1**: Leader (Rule 1).

- **Line 3**: Leader (Rule 2 - Target dari baris 7).
- **Line 4**: Leader (Rule 3 - Setelah `if` di baris 3).
- **Line 7**: Target dari baris 7 sendiri (Sudah tertutup Rule 2/3).
- **Line 8**: Leader (Rule 3 - Setelah `goto` di baris 7).

8.2.3 Hasil Pembentukan Blok

Berdasarkan leader tersebut, kita membagi kode menjadi:

- **Block 1**: Baris 1-2.
- **Block 2**: Baris 3.
- **Block 3**: Baris 4-7.
- **Block 4**: Baris 8.



Gambar 8.2: Visualisasi Segmentasi Kode menjadi Basic Blocks

8.3 Local Optimization: Folding dan Simplification

Optimasi lokal bekerja hanya di dalam satu *basic block*. Tujuannya adalah mengurangi beban komputasi instruksi aritmatika tanpa perlu menganalisis seluruh program.

8.3.1 1. Constant Folding

Jika semua operan bersifat konstan, kompilator melakukan perhitungan saat itu juga.

- **Sebelum**: `x = 2 * 3.14 * r`
- **Sesudah**: `x = 6.28 * r`

8.3.2 2. Algebraic Simplification

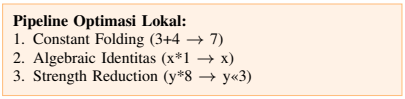
Menggunakan identitas matematika untuk menghapus operasi yang tidak perlu (*Identity Elimination*).

- `x + 0` → `x`
- `x * 1` → `x`
- `x / 1` → `x`

8.3.3 3. Strength Reduction

Mengganti operasi yang "mahal" (berat bagi CPU) dengan operasi yang "murah" namun ekuivalen. Biasanya dilakukan untuk perkalian dan pembagian dengan pangkat 2.

1. **Multiplication**: `x * 2` diganti `x + x` atau `x << 1`.
2. **Division**: `x / 4` diganti `x >> 2`.
3. **Exponent**: `pow(x, 2)` diganti `x * x`.



Gambar 8.3: Hierarki Optimasi Lokal pada Ekspresi Aritmatika

8.4 Representasi DAG untuk Optimasi Lokal

Directed Acyclic Graph (***DAG***) adalah struktur data yang sangat kuat untuk menganalisis aliran data di dalam sebuah *basic block*.

8.4.1 Manfaat DAG

Mengubah deretan instruksi ke dalam bentuk DAG memberikan beberapa keuntungan otomatis bagi kompilator:

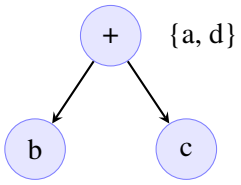
1. **Common Subexpression Elimination**: Jika sebuah perhitungan (misal `a + b`) muncul lebih dari sekali, DAG secara otomatis hanya akan membuat satu node untuk hasil tersebut.
2. **Dead Code Elimination**: Node yang tidak memiliki label variabel keluar (output) dan tidak digunakan oleh node lain dapat langsung dihapus.

3. **Instruction Reordering**: Struktur graf memungkinkan kompilator menata ulang instruksi tanpa merusak ketergantungan data.

8.4.2 Algoritma Konstruksi DAG

Setiap instruksi TAC `x = y op z` diproses sebagai berikut:

1. Cari node untuk operan `y` dan `z`. Jika belum ada, buat node daun (*leaf*).
2. Periksa apakah sudah ada node dengan operator `op` yang memiliki anak `y` dan `z`.
3. Jika ada, beri label `x` pada node tersebut (hasil penggunaan kembali).
4. Jika tidak ada, buat node baru dengan operator `op`, hubungkan ke `y` dan `z`, lalu beri label `x`.



Satu node '+' melayani variabel 'a' dan 'd'. Hilangkan redundansi.

Gambar 8.4: Representasi DAG untuk Eliminasi Subekspresi Umum

8.5 Control Flow Graph (CFG) dan Peephole Optimization

Setelah optimasi level blok (*local*), kompilator mulai memandang program secara keseluruhan melalui struktur graf dan jendela instruksi kecil.

8.5.1 1. Membangun Control Flow Graph yang Kompleks

CFG memetakan alur eksekusi antar blok. Terdapat dua jenis arah panah (*edges*):

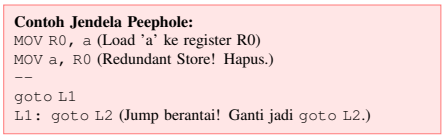
- **Fall-through**: Dari B_1 ke B_2 jika B_2 terletak tepat di bawah B_1 dan B_1 tidak berakhir dengan lompatan mutlak.
- **Jump**: Dari B_1 ke B_2 jika instruksi terakhir B_1 adalah lompatan ke label di awal B_2.

8.5.2 2. Peephole Optimization

Peephole Optimization adalah teknik machine-dependent yang menganalisis "jendela" kecil berisi 2-3 instruksi target (*assembly/machine code*) untuk mencari pola yang bisa dipercepat.

Teknik umum dalam Peephole:

1. **Redundant Load/Store Elimination**: Menghapus instruksi `STORE` variabel jika variabel tersebut langsung di-`LOAD` kembali tanpa perubahan.
2. **Unreachable Code**: Menghapus instruksi yang muncul setelah `goto` atau `return` mutlak yang tidak memiliki label.
3. **Flow of Control Optimization**: Mengganti `goto L1` di mana `L1` berisi `goto L2` menjadi langsung `goto L2`.
4. **Algebraic Hints**: Mengganti operasi matematika dengan instruksi mesin yang lebih efisien (misal: `INC` alih-alih `ADD 1`).



Gambar 8.5: Mekanisme Jendela (Peephole) pada Level Kode Mesin

8.5.3 Menuju Optimasi Global

Optimasi lokal dan peephole memberikan dasar yang kuat. Langkah berikutnya adalah *Data Flow Analysis* (Chapter 9) yang menggunakan CFG untuk melacak nilai variabel di seluruh fungsi, memungkinkan optimasi seperti *Global Dead Code Elimination*.

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 Kerangka Kerja Data-Flow Analysis

Data-Flow Analysis (DFA) adalah proses pengumpulan informasi tentang aliran data melalui graf kendali alir (*Control Flow Graph*) [13]. Informasi ini digunakan untuk menjawab pertanyaan global seperti: "Apakah variabel x pasti memiliki nilai 10 di baris ini?" atau "Apakah nilai y akan dibaca lagi di masa depan?".

9.1.1 Iterative Data-Flow Framework

Untuk menyelesaikan masalah DFA secara sistematis, kita menggunakan kerangka kerja iteratif. Setiap algoritma DFA dapat didefinisikan dengan empat komponen utama:

1. **Direction**: Arah aliran (*Forward* atau *Backward*).
2. **Domain**: Himpunan nilai yang dianalisis (misal: himpunan definisi variabel atau ekspresi).
3. **Transfer Function** (f_B): Aturan bagaimana sebuah blok mengubah informasi data-flow. Format umumnya: $OUT[B] = f_B(IN[B])$.
4. **Meet operator** ($\wedge$): Aturan penggabungan informasi ketika dua atau lebih jalur dalam CFG bertemu (biasanya berupa *Union* $\cup$ atau *Intersection* $\cap$).

9.1.2 Representasi Bit-Vector

Agar proses DFA berjalan cepat, kompilator biasanya menggunakan representasi **Bit-Vector**. Setiap elemen dalam domain direpresentasikan oleh satu bit dalam sebuah *array of bits*.

- Bit 1 berarti properti tersebut *true* atau ada dalam set.
- Bit 0 berarti *false* atau tidak ada.

Dengan bit-vector, operasi set seperti `Union` dapat dilakukan menggunakan instruksi bitwise `OR` (|), dan `Intersection` menggunakan bitwise `AND` (&), yang didukung sangat cepat oleh perangkat keras CPU.

Gambar 9.1: Model Aliran Data pada Satu Basic Block

9.2 Reaching Definitions Analysis

Analisis ***Reaching Definitions*** adalah analisis **Forward** dengan meet operator **Union** ($\cup$). Analisis ini menentukan definisi (penugasan nilai) mana yang mungkin masih aktif saat mencapai titik program tertentu.

9.2.1 Persamaan Aliran Data

Untuk setiap blok B, hubungannya didefinisikan sebagai:

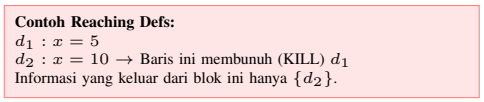
1. **Meet Operation**: $IN[B] = \bigcup_{P \in Pred(B)} OUT[P]$ (Definisi yang mencapai awal blok adalah gabungan dari semua yang keluar dari pendahulunya).
2. **Transfer Function**: $OUT[B] = GEN[B] \cup (IN[B] - KILL[B])$.

Di mana:

- $GEN[B]$: Himpunan definisi yang dibuat di dalam blok B dan mencapai akhir B.
- $KILL[B]$: Himpunan definisi di luar B yang variabelnya di-assign ulang di dalam B.

9.2.2 Konsep Fixed-Point

Kompilator menjalankan algoritma ini secara berulang di seluruh CFG. Karena himpunan data hanya bisa bertambah (monotonik) dan jumlah definisi terbatas, algoritma dijamin akan berhenti pada suatu titik di mana nilai IN dan OUT tidak lagi berubah. Titik stabil ini disebut **Fixed-Point**.



Gambar 9.2: Ilustrasi Operasi GEN dan KILL

9.3 Live Variable Analysis dan Alokasi Register

Analisis ***Live Variable*** adalah masalah **Backward** dengan meet operator **Union** ($\cup$). Analisis ini menjawab: "Apakah nilai variabel saat ini akan digunakan lagi di masa depan?".

9.3.1 Persamaan Aliran Data (Backward)

Kebalikan dari Reaching Defs, analisis ini merambat dari bawah ke atas:

1. **Meet Operation**: $OUT[B] = \bigcup_{S \in Succ(B)} IN[S]$.
2. **Transfer Function**: $IN[B] = USE[B] \cup (OUT[B] - DEF[B])$.

9.3.2 Peran dalam Alokasi Register

Ini adalah dasar dari **Register Allocation** yang efisien.

- Jika dua variabel tidak pernah "hidup" (*live*) secara bersamaan di titik mana pun, keduanya dapat menempati register fisik yang sama.
- Kompilator membangun **Interference Graph**: simpul adalah variabel, dan sisi ditarik antar variabel yang live bersamaan. Masalah alokasi register kemudian diselesaikan sebagai masalah *Graph Coloring*.

```
// Contoh Analisis Liveness
x = 10;
y = 20;
print(x); // x live di sini, y mati (dead)
// y = 20 adalah "Dead Code" karena y tidak live pada titik ini
```

Informasi ini memungkinkan **Dead Code Elimination** skala global yang lebih akurat daripada sekadar analisis di dalam satu blok.

9.4 Available Expressions dan Global CSE

Analisis ***Available Expressions*** adalah masalah **Forward** dengan meet operator **Intersection** ($\cap$). Analisis ini menentukan apakah sebuah perhitungan ekspresi sudah pernah dilakukan sebelumnya di semua jalur eksekusi yang memungkinkan.

9.4.1 Persamaan Aliran Data (Intersection)

Berbeda dengan Reaching Defs yang menggunakan Union, analisis ini menggunakan irisan karena sebuah ekspresi harus tersedia di **semua** jalur pendahulu:

1. **Meet Operation**: $IN[B] = \bigcap_{P \in Pred(B)} OUT[P]$.
2. **Transfer Function**: $OUT[B] = e\_gen[B] \cup (IN[B] - e\_kill[B])$.

9.4.2 Langkah Global CSE

Global Common Subexpression Elimination menggunakan informasi ini untuk:

1. Menemukan ekspresi $x + y$ yang *available* di awal blok B.
2. Melacak mundur ke titik-titik definisi ekspresi tersebut.
3. Mengganti perhitungan di titik-titik tersebut dengan variabel temporary t.
4. Mengganti penulisan ekspresi di blok B dengan pembacaan dari variabel t.

Local vs Global CSE:
Local: Hanya melihat di dalam satu blok sekuensial.
Global: Mengenali redundansi meski dipisahkan oleh `if` atau `loop`, asalkan operan tidak berubah di tengah jalan.

Gambar 9.3: Perbandingan Ruang Lingkup CSE

9.5 Global Constant Propagation

Jika pada optimasi lokal (Bab 8) kita hanya melihat penyebaran konstanta di dalam blok, ***Global Constant Propagation*** menyebarkan nilai konstanta melalui percabangan dan loop di seluruh program.

9.5.1 Mekanisme Aliran Data

Analisis ini menggunakan informasi dari *Reaching Definitions*. Jika di titik p, hanya terdapat satu definisi yang mencapai variabel x, dan definisi tersebut berupa konstanta (`x = 5`), maka semua penggunaan x di titik p dapat diganti dengan angka `5`.

9.5.2 Analisis Nilai Konstan

Beberapa alasan mengapa ini lebih kuat daripada versi lokal:

1. **Branch Pruning**: Jika kondisi `if (x > 0)` dievaluasi mendapati x selalu bernilai 10, maka blok *else* dapat dihapus sepenuhnya (*Unreachable Code Elimination*).

2. **Loop Invariant**: Membantu mengenali nilai yang tidak berubah selama iterasi loop.

```
void contohGlobal() {
    int x = 7;
    if (kondisi) {
        // x masih mencapai sini sebagai 7
    } else {
        // x masih mencapai sini sebagai 7
    }
    int y = x + 3; // Global Const Prop: y = 7 + 3 = 10
}
```

9.6 Pipeline dan Interaksi Optimasi Global

Optimasi tingkat global (*Data-Flow Based*) jarang berjalan secara mandiri. Sebaliknya, satu optimasi seringkali menjadi pemicu untuk optimasi lainnya dalam sebuah **Optimization Loop**.

9.6.1 Kaskade Optimasi

Perhatikan interaksi berikut:

1. **Global Constant Propagation** mengganti `x` dengan `5`.

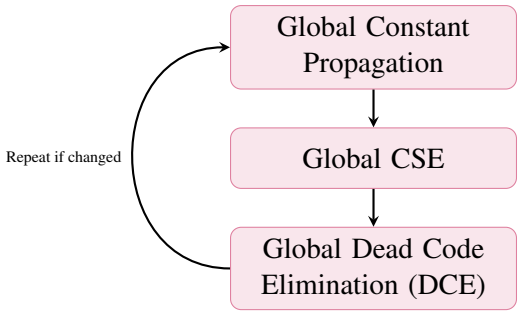
2. Hal ini memicu **Constant Folding** pada ekspresi `x + 10` menjadi `15`.

3. Hasil lipatan (folding) mungkin membuat sebuah kondisi `if` selalu bernilai benar, yang memicu **Dead Code Elimination** pada blok `else`.

4. Penghapusan blok tersebut menghapus definisi variabel lain, yang mungkin memicu **Global CSE** untuk ekspresi yang sebelumnya terhambat oleh definisi tersebut.

9.6.2 Iterasi Hingga Fixed-Point

Kompilator tidak hanya melakukan analisis data-flow hingga fixed-point, tetapi seluruh *optimizer pipeline* juga sering diulang beberapa kali hingga tidak ada lagi instruksi yang bisa disederhanakan.



Gambar 9.4: Loop Iteratif pada Optimization Pipeline

9.6.3 Kesimpulan

Data-flow analysis merubah pandangan kompilator dari deretan instruksi linear menjadi aliran informasi yang kaya. Dengan informasi global ini, kompilator dapat menghasilkan kode yang jauh lebih efisien daripada apa yang ditulis manusia secara manual.

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

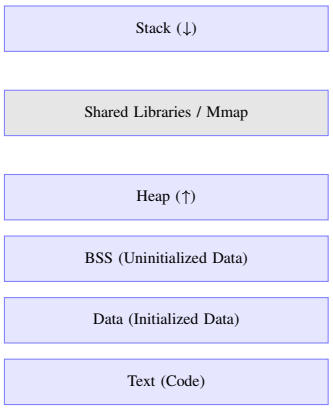
Runtime Environment mengelola bagaimana program menggunakan memori saat dieksekusi. Manajemen memori saat runtime merupakan salah satu aspek yang paling krusial dalam performa dan keamanan sebuah program hasil kompilasi [14].

10.1.1 Segmen Memori Utama

Secara tradisional, memori proses dibagi menjadi beberapa segmen:

- **Text/Code Segment**: Menyimpan instruksi biner hasil kompilasi. Biasanya bersifat *read-only* untuk mencegah perubahan kode secara tidak sengaja.
- **Data Segment (Initialized)**: Menyimpan variabel global dan statis yang sudah diberi nilai awal (misal: `int x = 10;`).
- **BSS (Block Started by Symbol)**: Menyimpan variabel global dan statis yang belum diinisialisasi (misal: `int y;`). Segmen ini biasanya diisi dengan nol oleh sistem operasi saat program dimuat.
- **Shared Libraries / Memory Mapping**: Area di antara stack dan heap yang digunakan untuk memetakan *shared objects* (.so atau .dll) dan file yang dipetakan ke memori.
- **Stack**: Segmen yang tumbuh ke bawah untuk mengelola pemanggilan fungsi dan variabel lokal secara LIFO.

- **Heap**: Segmen yang tumbuh ke atas untuk alokasi memori dinamis lewat instruksi seperti `malloc` atau `new`.



Gambar 10.1: Model organisasi memori runtime yang diperluas

10.2 Activation Records dan Skoping Bertingkat

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

10.2.1 Struktur Stack Frame

Sebuah stack frame biasanya menampung informasi kritis berikut:

1. **Return Address**: Alamat instruksi berikutnya di *caller* yang harus dieksekusi setelah fungsi selesai.

2. **Control Link (Dynamic Link)**: Pointer ke stack frame milik fungsi pemanggil (*caller*). Digunakan untuk menghapus frame saat ini dan kembali ke frame sebelumnya.

3. **Access Link (Static Link)**: Pointer ke stack frame milik fungsi yang secara leksikal (sintaksis) membungkus fungsi saat ini. Digunakan untuk mengakses variabel non-lokal dalam bahasa yang mendukung fungsi bersarang (*nested functions*).

4. **Saved Registers**: Nilai register prosesor yang harus dipulihkan sebelum kembali.

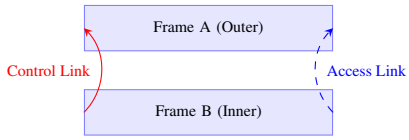
5. **Local Variables**: Ruang untuk data yang dideklarasikan di dalam fungsi tersebut.

10.2.2 Menangani Akses Variabel Non-Lokal

Dalam bahasa seperti Pascal atau Python, fungsi di dalam fungsi dapat mengakses variabel milik induknya. Ada dua teknik utama untuk mengelola ini:

1. **Access Links**: Membentuk rantai pointer statis. Jika variabel berada di tingkat bersarang ke-3 di atasnya, kompilator harus mengikuti rantai link tersebut 3 kali. Ini sederhana tapi lambat jika nesting terlalu dalam.

2. **Displays**: Menggunakan larik (*array*) global berisi pointer ke frame yang sedang aktif di setiap tingkat bersarang. Dengan *display*, akses ke tingkat mana pun selalu berbiaya konstan (satu kali akses larik).



Gambar 10.2: Perbedaan Aliran Control Link (Dinamis) vs Access Link (Statis)

10.3 Mekanisme Panggilan: Konvensi Register

Prosedur pemanggilan fungsi melibatkan pembagian tanggung jawab antara fungsi yang memanggil (*caller*) dan fungsi yang dipanggil (*callee*) dalam mengelola register CPU.

10.3.1 Caller-Saved vs Callee-Saved

Kompilator menggunakan konvensi berikut untuk menjaga integritas data:

1. **Caller-Saved Registers** (Volatile): Register yang nilainya boleh dirusak oleh fungsi yang dipanggil. Jika *caller* membutuhkan nilai di register ini setelah panggilan fungsi, *caller* harus menyimpannya ke stack sebelum perintah `call`. Contoh pada x86-64: `rax`, `rcx`, `rdx`.

2. **Callee-Saved Registers** (Non-volatile): Register yang nilainya harus tetap sama bagi *caller* sebelum dan sesudah panggilan. Jika *callee* ingin menggunakan register ini, ia harus menyimpan isinya di awal fungsi (*prologue*) dan memulihkannya di akhir fungsi (*epilogue*). Contoh pada x86-64: `rbx`, `rbp`, `r12-r15`.

10.3.2 Penyaluran Parameter

Pada arsitektur modern (seperti System V AMD64 ABI), enam parameter pertama biasanya dikirim melalui register (`rdi, rsi, rdx, rcx, r8, r9`) alih-alih stack untuk meningkatkan performa. Parameter ke-7 dan seterusnya baru dikirim melalui stack.

Konvensi Umum (x86-64):
- Return Value: `rax`
- Stack Pointer: `rsp`
- Frame Pointer: `rbp` (Callee-saved)
- Arguments: rdi, rsi, rdx, rcx, r8, r9

Gambar 10.3: Peta Penggunaan Register dalam Prosedur Panggilan

10.4 Manajemen Dinamis dan Strategi Heap

Berbeda dengan stack yang memiliki struktur kaku, ***Heap*** adalah area memori bebas di mana blok-blok dapat dialokasikan dan dibebaskan dalam urutan apa pun.

10.4.1 Strategi Alokasi Pemuatan

Kompilator dan pustaka runtime menggunakan algoritma tertentu untuk mencari blok kosong yang cukup besar:

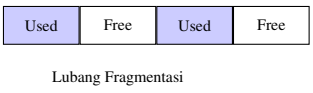
1. **First-Fit**: Mencari dari awal heap dan mengambil blok kosong pertama yang ukurannya cukup. Algoritma ini sangat cepat namun cenderung meninggalkan fragmentasi di awal heap.

2. **Best-Fit**: Mencari ke seluruh heap untuk menemukan blok kosong terkecil yang masih cukup menampung permintaan. Ini meminimalkan sisa ruang (*leftover*) namun pencariannya lambat.

3. **Next-Fit**: Mirip First-fit, tapi pencarian dimulai dari titik terakhir alokasi dilakukan. Ini mendistribusikan alokasi lebih merata di seluruh heap.

10.4.2 Masalah Fragmentasi

Tujuan utama pengelola memori adalah menekan **External Fragmentation** (adanya banyak lubang kecil yang terpisah-pisah sehingga tidak bisa melayani alokasi satu blok besar). Solusinya adalah *Coalescing* (menggabungkan blok kosong yang bertetangga menjadi satu) atau *Compaction* (menggeser blok-blok yang terisi ke satu sisi).



Lubang Fragmentasi

Gambar 10.4: Ilustrasi Fragmentasi Eksternal pada Heap

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 Metode Garbage Collection Lanjutan

Setelah memahami mekanisme dasar *Reference Counting* dan *Mark-and-Sweep*, kompilator modern sering menggunakan teknik tambahan untuk menangani fragmentasi dan meningkatkan performa.

10.6.1 1. Mark-Compact Garbage Collection

Kelemahan *Mark-and-Sweep* murni adalah ia meninggalkan "lubang" (*holes*) di seluruh heap. ***Mark-Compact*** menyelesaikan ini dalam tiga fase:

1. **Mark**: Menandai semua objek yang masih bisa dijangkau (*reachable*) dari *GC Roots* (Stack, Register, Variabel Global).

2. **Compact**: Memindahkan semua objek yang ditandai ke satu sisi heap, sehingga semua memori kosong terkumpul menjadi satu blok besar yang kontinu.

3. **Update**: Memperbarui semua pointer di program agar merujuk ke lokasi baru objek yang telah dipindah.

10.6.2 2. Copying Garbage Collection (Semi-space)

Algoritma ini membagi heap menjadi dua bagian berukuran sama: **From-Space** dan **To-Space**.

1. Alokasi hanya dilakukan di *From-Space*.
2. Saat penuh, GC menyalin hanya objek yang *live* ke *To-Space* secara berurutan (kontinu).
3. Seluruh isi *From-Space* yang tersisa (sampah) langsung dibuang secara massal.
4. Peran *From-Space* dan *To-Space* ditukar untuk siklus berikutnya.

Keuntungan: Alokasi menjadi sangat cepat (cukup menaikkan pointer *top of heap*) dan fragmentasi otomatis hilang. **Kerugian**: Membutuhkan kapasitas memori dua kali lipat dari yang sebenarnya bisa digunakan.

Ringkasan Strategi GC:
- *Reference Counting*: Real-time tapi tidak bisa menangani siklus.
- *Mark-Sweep*: Menangani siklus tapi menyebabkan fragmentasi.
- *Copying GC*: Sangat cepat tapi memboroskan memori.
- *Mark-Compact*: Hemat memori dan bebas fragmentasi tapi lambat.

Gambar 10.5: Perbandingan Algoritma Manajemen Memori Otomatis

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 [15].

11.1.1 Jenis Pengalamatan Umum

- **Register Addressing**: Operan berada langsung di register (`ADD R1, R2`). Sangat cepat karena tidak ada akses memori.
- **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 Complex Addressing Modes (CISC)

Arsitektur seperti x86 mendukung mode yang lebih kompleks untuk mendukung abstraksi bahasa tingkat tinggi secara langsung di perangkat keras:

$$\text{Alamat Efektif} = \text{Base} + (\text{Index} \times \text{Scale}) + \text{Displacement}$$

Di mana *Scale* biasanya bernilai 1, 2, 4, atau 8—cocok untuk ukuran data dasar (`char`, `short`, `int`, `double`).

11.1.3 Instruksi LEA (Load Effective Address)

Instruksi `LEA` pada x86 adalah "trik" yang sering digunakan kompilator. Meskipun secara teknis merupakan instruksi pengalamatan, `LEA` sebenarnya melakukan kalkulasi aritmatika tanpa mengakses memori.

- Contoh: `LEA EAX, [EBX + ECX*4]` menghitung $EBX + 4 \times ECX$ dan menyimpan hasilnya di EAX.
- Kompilator menggunakan ini untuk perkalian cepat (misal: kali 5, kali 9) dalam satu instruksi.

Aplikasi LEA untuk Array:
`A[i]` di mana tiap elemen 4 byte.
Instruksi: `LEA EAX, [EBX + ECX*4]`
EBX: Alamat Dasar A; ECX: Indeks i.

Gambar 11.1: Efisiensi Pengalamatan Kompleks pada Kompiler

11.2 Filosofi Pengalamatan: RISC vs CISC

Arsitektur target sangat mempengaruhi bagaimana kompilator menghasilkan kode untuk mengakses data. Dua filosofi utama adalah ***CISC*** (x86) dan ***RISC*** (RISC-V, ARM).

11.2.1 CISC: Kaya dan Kompleks

Pada x86, filosofinya adalah menyediakan instruksi yang mampu melakukan banyak hal sekaligus.

- **Memory-to-Memory**: Memungkinkan operan diambil dari memori, diproses, dan hasilnya disimpan kembali ke memori dalam satu instruksi.
- **Variable Length**: Instruksi bisa berukuran mulai dari 1 hingga 15 byte, tergantung kompleksitas pengalamatannya.

11.2.2 RISC: Sederhana dan Cepat

Pada RISC-V atau ARM, filosofinya adalah "Load/Store Architecture".

- **Terisolasi**: Hanya instruksi khusus (`LOAD` dan `STORE`) yang boleh mengakses memori. Instruksi aritmatika hanya boleh bekerja pada register.

- **Decomposisi**: Kompilator harus memecah akses kompleks menjadi urutan instruksi sederhana.

11.2.3 Perbandingan Dekomposisi Kode

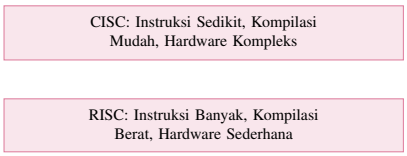
Akses ke `x = a[i]` di mana `a` adalah array global:

x86 (CISC):

```
MOV EAX, [base_a + ECX*4] ; 1 instruksi langsung
```

RISC-V (RISC):

```
SLLI t1, t0, 2      # t1 = i * 4 (shift left logical)
LA   t2, base_a     # Muat alamat dasar a ke t2
ADD  t3, t1, t2     # t3 = alamat elemen yang dituju
LW   a0, 0(t3)      # Muat nilai dari memori ke register
```



Gambar 11.2: Trade-off antara Kompleksitas Kompiler dan Desain CPU

11.3 Layout Memori Array Multidimensi

Penyimpanan array multidimensi dalam memori linear (satu dimensi) memerlukan pemetaan indeks yang konsisten. Ada dua pendekatan utama yang digunakan oleh berbagai bahasa pemrograman.

11.3.1 Row-Major vs Column-Major

- **Row-Major Order** (digunakan di C, C++, Java): Elemen-elemen dalam satu baris disimpan secara berturutan. Indeks paling kanan berubah paling cepat.
- **Column-Major Order** (digunakan di Fortran, MATLAB, Julia): Elemen-elemen dalam satu kolom disimpan secara berturutan. Indeks paling kiri berubah paling cepat.

Rumus pemetaan alamat untuk `A[i][j]` pada array $M \times N$:

- **Row-Major**: Base$(A) + (i \times N + j) \times$ ukuran_elemen
- **Column-Major**: Base$(A) + (j \times M + i) \times$ ukuran_elemen

11.3.2 Larik Petunjuk (Iliffe Vectors)

Alih-alih menyatukan baris dalam satu blok kontinu, beberapa bahasa (seperti Python untuk *list of lists*) menggunakan *Iliffe Vectors*.

- Array utama berisi sekumpulan pointer.
- Setiap pointer merujuk ke blok memori baris yang terpisah.
- **Keuntungan**: Mendukung *Jagged Arrays* (panjang baris bisa berbeda-beda).
- **Kerugian**: Memerlukan akses memori ganda (dereferensi pointer baris, baru kemudian ambil datanya).

11.3.3 Rumus Umum N-Dimensi (Row-Major)

Untuk array $A[d_1][d_2]...[d_n]$, alamat elemen $A[i_1][i_2]...[i_n]$ dihitung sebagai:

$$\text{Offset} = i_1 \cdot (d_2 \cdot d_3 \cdots d_n) + i_2 \cdot (d_3 \cdot d_4 \cdots d_n) + \cdots + i_n$$

Gambar 11.3: Visualisasi Struktur Kontinu vs Layar Petunjuk

11.4 Layout Struktur dan Perataan Memori

Kompilator tidak selalu menyimpan anggota struktur secara berdampingan tanpa celah. Ada aturan perangkat keras yang memaksa data tertentu berada di alamat yang "selaras" (*aligned*).

11.4.1 Perataan Memori (Memory Alignment)

Kebanyakan prosesor modern bekerja paling efisien jika data diakses pada alamat yang merupakan kelipatan dari ukuran data tersebut.

- **Natural Alignment**: `int` (4 byte) harus berada di alamat yang habis dibagi 4; `double` (8 byte) di alamat yang habis dibagi 8.
- Jika data tidak selaras (*misaligned*), CPU mungkin perlu melakukan dua kali akses memori untuk satu data, atau bahkan menyebabkan *exception*.

11.4.2 Padding pada Struct

Untuk memenuhi syarat perataan, kompilator menyelipkan byte kosong yang disebut *padding*.

```
struct Contoh {
    char a;     // 1 byte
    // Padding: 3 byte (agar 'b' mulai di offset 4)
    int b;      // 4 byte
    short c;    // 2 byte
    // Padding: 2 byte (agar total struct kelipatan 4 atau 8)
};
```

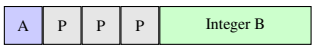
11.4.3 32-bit vs 64-bit

Ukuran pointer dan tipe data dasar dapat berubah antar arsitektur:

- **Pointer**: 4 byte pada sistem 32-bit, 8 byte pada sistem 64-bit.
- **Alignment Requirement**: Sistem 64-bit seringkali lebih ketat, mengharuskan lebih banyak padding untuk tipe data `long` atau pointer.

11.4.4 Packed Structures

Kompilator menyediakan direktif (seperti `#pragma pack(1)`) untuk menghilangkan padding. Ini berguna untuk protokol jaringan atau format file, namun berisiko menurunkan performa akses memori.



Gambar 11.4: Visualisasi Padding (P) untuk Menyelaraskan Integer B

11.5 Aritmatika Pointer dan Skalasi

Aritmatika pointer dalam bahasa tingkat tinggi seperti C diterjemahkan ke dalam kalkulasi alamat otomatis oleh kompilator yang menyesuaikan ukuran tipe data.

11.5.1 Hubungan Pointer dan Skala

Saat kita melakukan `p + i` di mana `p` adalah pointer ke tipe data T, alamat fisik yang dihitung adalah:

$$\text{Alamat Baru} = \text{Alamat Lama} + (i \times \text{sizeof}(T))$$

Kompilator secara otomatis menyisipkan faktor skala (*scale factor*) ini ke dalam instruksi mesin.

11.5.2 Kaitan dengan Addressing Mode

Pada arsitektur mesin, operasi pointer seringkali dipetakan langsung ke mode pengalamatan terindex.

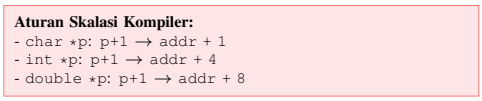
- **High-Level**: `int val = *(ptr + i);`
- **Machine-Level (x86)**: `MOV EAX, [EBX + ECX*4]`

Di sini, `EBX` adalah `ptr`, `ECX` adalah `i`, dan faktor skala 4 adalah `sizeof(int)`.

11.5.3 Pointer vs Array

Bagi kompilator, array dan pointer seringkali diperlakukan dengan logika pengalamatan yang identik. Nama array bertindak sebagai alamat dasar (*base address*) konstanta, sedangkan pointer adalah alamat dasar variabel.

- `A[i]` $\equiv$ `*(A + i)`
- Kompilator menggunakan informasi tipe data untuk menentukan apakah ia harus melompat 1 byte (`char`), 4 byte (`int`), atau 8 byte (`double`) untuk setiap penambahan indeks.



Gambar 11.5: Skalasi Otomatis dalam Aritmatika Pointer

11.6 Translasi Alamat dan Performa

Pada sistem operasi modern, alamat yang dihasilkan oleh kompilator adalah alamat logis atau *virtual*. Perangkat keras (*MMU*) harus memetakan ini ke alamat fisik di RAM.

11.6.1 Virtual to Physical Address

Proses translasi ini biasanya melibatkan pembagian alamat menjadi *Page Number* dan *Offset*. Kompilator harus memastikan data yang sering diakses bersama-sama berada dalam satu *page* (halaman) yang sama untuk meminimalkan *page fault*.

11.6.2 TLB (Translation Lookaside Buffer)

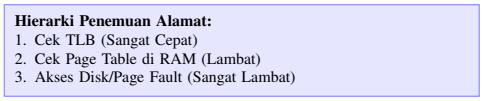
TLB adalah cache khusus di dalam CPU yang menyimpan pemetaan alamat virtual ke fisik yang baru saja digunakan.

- **TLB Hit**: Alamat fisik ditemukan di TLB, akses sangat cepat.
- **TLB Miss**: CPU harus mencari di tabel halaman (*page table*) di RAM, yang jauh lebih lambat.

11.6.3 Pentingnya Lokalitas Spasial

Manajemen memori oleh kompilator (seperti pemilihan *Row-Major order*) sangat krusial karena mendukung **Lokalitas Spasial**.

- Jika elemen array diakses secara berurutan sesuai layout memori, maka data tersebut kemungkinan besar sudah ada di cache CPU dan pemetaan halamannya sudah ada di TLB.
- Lompatan alamat yang acak atau besar dapat menyebabkan *TLB Miss* dan *Cache Miss* berulang kali, menurunkan performa program secara drastis meskipun algoritma di tingkat tinggi tampak efisien.



Gambar 11.6: Alur Penemuan Alamat Fisik di Perangkat Keras

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)

Target Machine adalah arsitektur komputer tujuan di mana kode hasil kompilasi akan dijalankan. Memahami karakteristik perangkat keras sangat krusial bagi *code generator* untuk menghasilkan instruksi yang optimal [15].

12.1.1 Arsitektur RISC (Reduced Instruction Set Computer)

Contoh: RISC-V, ARM.

- **Load-Store Architecture**: Hanya instruksi `Load` dan `Store` yang bisa mengakses memori. Instruksi aritmatika hanya bekerja pada register.
- **Register Pressure**: Karena semua operan harus berada di register, RISC membutuhkan lebih banyak register fisik dan temporer, yang meningkatkan "tekanan" pada pengalokasi register.
- **Fixed-Length**: Instruksi selalu berukuran tetap (32-bit), memudahkan *pipelining*.

12.1.2 Arsitektur CISC (Complex Instruction Set Computer)

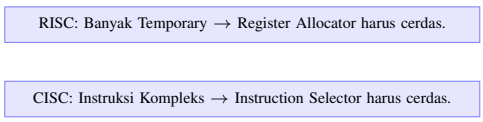
Contoh: x86 (Intel/AMD).

- **Orthogonality**: Kemampuan instruksi untuk menggunakan berbagai mode pengalamatan secara bebas. Misalnya, instruksi `ADD` pada x86 bisa menjumlahkan register dengan memori secara langsung.
- **Variable-Length**: Instruksi berukuran 1-15 byte, menghemat ruang memori tapi mempersulit pendekodean instruksi (*decoding*).

12.1.3 Dampak pada Code Generation

Code generator harus memilih strategi yang sesuai dengan arsitektur:

1. Pada **RISC**, fokus pada jadwal instruksi (*scheduling*) untuk menghindari *pipeline stall* dan manajemen register yang agresif.
2. Pada **CISC**, fokus pada pemilihan instruksi kompleks yang dapat menggabungkan beberapa operasi TAC menjadi satu instruksi mesin untuk mengurangi ukuran kode (*code density*).



Gambar 12.1: Prioritas Optimasi berdasarkan Arsitektur Target

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 Tiling (Pengubinan)

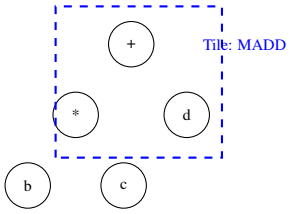
Proses pemilihan instruksi sering divisualisasikan sebagai "menutupi" pohon ekspresi (*Expression Tree*) dengan "ubin" (*tiles*). Setiap ubin mewakili satu instruksi mesin yang dapat menggantikan satu atau lebih simpul pada pohon tersebut.

12.2.2 Algoritma Pilihan

Ada dua strategi populer untuk melakukan *tiling*:

1. **Maximal Munch**: Strategi *greedy* (rakus). Dimulai dari akar pohon, pilih ubin terbesar yang cocok (*match*). Jika ada ubin berukuran 3 simpul dan 1 simpul, ubin 3 simpul akan dipilih. Sangat efektif untuk arsitektur RISC.

2. **Dynamic Programming**: Strategi optimal. Menghitung biaya (*cost*) minimum untuk menutupi setiap sub-pohon. Algoritma ini memastikan total biaya seluruh pohon adalah yang terendah. Sangat berguna jika CPU memiliki banyak instruksi kompleks dengan biaya yang bervariasi.



Gambar 12.2: Representasi Tiling: Satu instruksi MADD menutupi operasi Multiply dan Add

12.3 Register Allocation dan Graph Coloring

Register adalah sumber daya paling berharga dalam CPU karena kecepatannya. ***Register Allocation*** berupaya menempatkan sebanyak mungkin variabel ke dalam register fisik dan meminimalkan *spilling* (pemindahan data ke memori).

12.3.1 Masalah Pewarnaan Graf (Graph Coloring)

Alokasi register global sering dimodelkan sebagai masalah pewarnaan graf.

1. **Interference Graph**: Setiap simpul mewakili sebuah variabel. Sebuah sisi menghubungkan dua variabel jika mereka "hidup" (*live*) secara bersamaan. Variabel yang terhubung tidak boleh menggunakan register yang sama.

2. **Warna (Register)**: Jika CPU memiliki k register, tantangannya adalah mewarnai graf tersebut dengan maksimal k warna sehingga tidak ada dua simpul bertetangga yang memiliki warna yang sama.

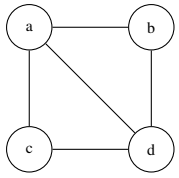
12.3.2 Algoritma Chaitin-Briggs

Langkah-langkah umum algoritma ini adalah:

- **Build**: Bangun graf interferensi berdasarkan analisis *liveness*.

- **Simplify**: Cari simpul dengan derajat $< k$, hapus dari graf, dan masukkan ke dalam stack.

- **Spill**: Jika semua simpul berderajat $\geq k$, pilih satu variabel untuk disimpan di memori (*spilling*), hapus dari graf, dan lanjutkan.
- **Select**: Ambil simpul dari stack satu per satu dan berikan warna yang belum digunakan oleh tetangganya.



Gambar 12.3: Graf Interferensi: Interaksi antar variabel yang tidak boleh berbagi register

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 Algoritma Code Generation Lokal

Untuk menghasilkan kode di dalam sebuah *Basic Block*, kompilator menggunakan algoritma yang memaksimalkan pemanfaatan register melalui informasi lokal.

12.5.1 Informasi Penggunaan Berikutnya (Next-Use Info)

Kompilator melakukan pemindaian mundur (*backward scan*) pada blok tersebut untuk menentukan kapan sebuah variabel akan digunakan lagi. Informasi ini disimpan di dalam tabel simbol atau anotasi pada instruksi.

- Jika variabel `x` digunakan di baris i dan tidak digunakan lagi setelah itu di dalam blok tersebut, maka register yang menampung `x` dapat langsung dibebaskan atau digunakan untuk menampung hasil operasi di baris i.
- Informasi ini sangat vital bagi fungsi `GetReg()` untuk meminimalkan pengosongan register ke memori (*spilling*).

12.5.2 Fungsi Pembantu GetReg

Fungsi `GetReg(I)` dipanggil untuk setiap instruksi $I : x = y \; op \; z$ untuk menentukan register mana yang akan menampung x. Aturannya antara lain:

1. Jika variabel y sudah berada di register R, dan y tidak memiliki penggunaan lagi setelah instruksi ini (*no next use*), maka R dapat digunakan untuk menampung hasil x.
2. Jika tidak ada register kosong, pilih register yang variabel di dalamnya memiliki "jarak penggunaan berikutnya" paling jauh (*Heuristic: Farthest Next Use*).

Logika Pembangkitan:
1. Muat operan ke register (jika belum).
2. Pilih register hasil lewat GetReg.
3. Hasilkan instruksi mesin.
4. Mutakhirkan lokasi variabel di Tabel Simbol.

Gambar 12.4: Alur Kerja Code Generator di tingkat Basic Block

12.6 Optimasi dan Penjadwalan Instruksi

Tahap akhir dari pembangkitan kode adalah memastikan instruksi berjalan seefisien mungkin pada perangkat keras yang memiliki *pipeline*.

12.6.1 Penjadwalan Instruksi (Instruction Scheduling)

CPU modern mengeksekusi instruksi dalam beberapa tahap (*fetch, decode, execute, write-back*). Jika sebuah instruksi membutuhkan hasil dari instruksi sebelumnya yang belum selesai, terjadi hambatan yang disebut *stall* atau *bubble*. ***Instruction Scheduling*** berupaya menukar urutan instruksi agar *stall* terminimalisir.

12.6.2 Bahaya Jalur Pipa (Pipeline Hazards)

Kompilator harus mendeteksi dan mengatasi tiga jenis *hazard*:

1. **Data Hazard**: Instruksi bergantung pada data dari instruksi sebelumnya (misal: *Read-After-Write*).
2. **Control Hazard**: Terjadi saat ada instruksi percabangan (*jump/branch*), di mana CPU tidak tahu instruksi mana yang harus diambil selanjutnya.
3. **Structural Hazard**: Dua instruksi mencoba menggunakan sumber daya perangkat keras yang sama (misal: unit *floating point*) secara bersamaan.

12.6.3 Peephole Optimization

Merupakan teknik optimasi lokal yang memindai jendela kecil instruksi (misal: 2-3 instruksi) dan menggantinya dengan urutan yang lebih efisien. Contoh:

- *Redundant Load/Store Elimination*: Menghapus `STORE R1, x` diikuti `LOAD x, R1`.
- *Strength Reduction*: Mengganti `MUL R1, 8` dengan `SHL R1, 3`.
- *Algebraic Simplification*: Menghapus `ADD R1, 0`.

Contoh Penjadwalan:
1. LD R1, a
2. LD R2, b
3. ADD R3, R1, R2 → *Stall* jika LD belum selesai.

Solusi: Sisipkan instruksi independen di antara 2 dan 3.

Gambar 12.5: Ilustrasi Penghindaran Pipeline Stall lewat Penjadwalan

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 Kompleksitas Pewarnaan Graf

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

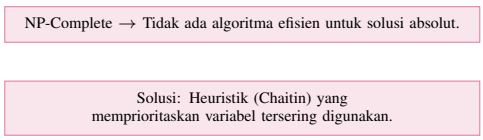
- **Node**: Variabel yang aktif (*live variables*).
- **Edge**: Saling berpotongan (*interfere*), artinya dua variabel hidup di waktu yang sama.
- **Warna (k)**: Jumlah register fisik yang tersedia.

Secara teoretis, menentukan apakah sebuah graf dapat diwarnai dengan k warna adalah masalah yang bersifat **NP-Complete**. Ini berarti untuk fungsi yang besar dengan banyak variabel, menemukan solusi optimal secara matematis akan memakan waktu yang sangat lama.

13.1.2 Pendekatan Heuristik

Kompilator praktis menggunakan algoritma heuristik seperti [15] yang memberikan hasil "cukup baik" dalam waktu linear atau polinomial rendah.

1. **Chaitin-Briggs**: Menggunakan penyederhanaan graf secara iteratif.
2. **Linear Scan**: Digunakan untuk kompilasi yang cepat (*JIT compilers*) seperti pada V8 (JavaScript) atau JVM HotSpot, karena lebih sederhana dibanding *graph coloring* namun tetap memberikan performa memadai.



Gambar 13.1: Tantangan Teoretis dalam Alokasi Register

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
```

```
19      InterferenceGraph *graph = create_graph(num_vars);
20
21      for (int i = 0; i < num_vars; i++) {
22          for (int j = i + 1; j < num_vars; j++) {
23              if (ranges_interfere(ranges[i], ranges[j])) {
24                  add_interference_edge(graph, i, j);
25              }
26          }
27      }
28
29      return graph;
30  }
```

13.3 Algoritma Graph Coloring (Chaitin-Briggs)

Algoritma ini adalah standar dalam kompilator produksi karena mampu memberikan alokasi global yang efisien melalui pendekatan terstruktur.

13.3.1 Simpul Terwarna-Awal (Pre-colored Nodes)

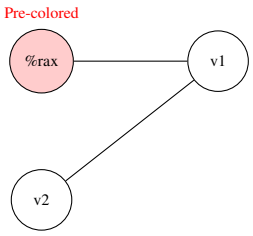
Dalam graf interferensi, terdapat simpul yang mewakili register fisik mesin (seperti %rax, %rbx pada x86). Simpul-simpul ini disebut ***Pre-colored Nodes***.

- **Alasan**: Beberapa instruksi mengharuskan penggunaan register tertentu (contoh: hasil fungsi selalu di %rax).
- **Penanganan**: Simpul terwarna-awal tidak boleh "disederhanakan" (*Simplify*) atau "dipindahkan ke memori" (*Spill*). Mereka bertindak sebagai kendala tetap dalam proses pewarnaan simpul variabel lainnya.

13.3.2 Fase Select dan Backtracking

Setelah fase *Simplify* mengosongkan graf (atau memindahkan beberapa simpul ke *spill stack*), algoritma memasuki fase ***Select***:

1. Ambil simpul dari stack satu per satu.
2. Assign warna (register) yang tidak digunakan oleh tetangganya.
3. **Optimistic Coloring**: Briggs mengusulkan agar simpul yang awalnya ditandai *spill* tetap dicoba diwarnai pada fase ini. Terkadang, setelah tetangganya diwarnai, ternyata masih ada warna tersisa yang bisa digunakan, sehingga *spill* nyata bisa dihindari.



Gambar 13.2: Kendala Register Fisik: v1 tidak boleh menggunakan %rax karena interferensi

13.4 Linear Scan Allocation

13.4.1 Linear Scan Algorithm

Alokasi register yang lebih efisien:

```
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 Dampak Konvensi Panggilan (Calling Conventions)

Alokasi register tidak terjadi dalam ruang hampa; ia harus mematuhi aturan interaksi antar fungsi yang didefinisikan oleh ***Application Binary Interface (ABI)***.

13.5.1 Caller-Saved vs Callee-Saved

ABI membagi register menjadi dua kategori utama yang memaksa kompilator melakukan penghematan (*saving*) secara strategis:

1. **Caller-Saved** (Volatile): Register yang nilainya boleh dirusak oleh fungsi yang dipanggil. Jika pemanggil (*caller*) masih membutuhkan data tersebut setelah panggilan fungsi, ia harus menyimpannya ke *stack* sebelumnya.

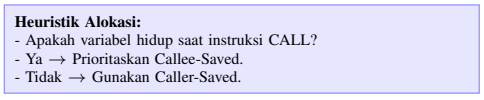
2. **Callee-Saved** (Non-volatile): Register yang nilainya harus dijaga. Jika fungsi yang dipanggil (*callee*) ingin menggunakannya, ia wajib menyimpan nilai aslinya di awal fungsi dan mengembalikannya sebelum selesai.

13.5.2 Strategi Penempatan Variabel

Kompilator yang cerdas menggunakan informasi ini untuk optimasi:

- **Variabel Berumur Panjang** (melewati banyak panggilan fungsi): Sebaiknya diletakkan di **Callee-Saved registers**. Ini menghindari biaya *save/restore* di setiap titik pemanggilan fungsi.

- **Variabel Berumur Pendek** (tidak melewati panggilan fungsi): Sebaiknya diletakkan di **Caller-Saved registers** karena tidak ada risiko data tersebut dihancurkan oleh fungsi lain.



Gambar 13.3: Strategi Pemilihan Register Berbasis Konvensi Panggilan

13.6 Penggabungan (Coalescing)

Coalescing adalah teknik untuk menghilangkan instruksi salin (`MOVE`) yang tidak perlu dengan memberikan register yang sama kepada sumber dan target salinan.

13.6.1 Heuristik Penggabungan yang Aman

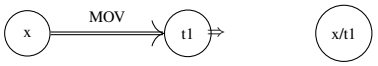
Penggabungan agresif dapat membuat graf yang awalnya bisa diwarnai menjadi tidak bisa diwarnai (*uncolorable*). Oleh karena itu, kompilator menggunakan dua kriteria utama:

1. **Kriteria Briggs**: Dua simpul dapat digabungkan jika simpul hasil penggabungan memiliki kurang dari k tetangga yang memiliki derajat $\geq k$. Ini menjamin fase *Simplify* masih bisa berjalan lancar.

2. **Kriteria George**: Simpul U dan V dapat digabungkan jika untuk setiap tetangga T dari V, T sudah bertetangga dengan U atau T memiliki derajat rendah ($< k$).

13.6.2 Manfaat

Selain mengurangi jumlah instruksi, *coalescing* juga mengurangi kebutuhan register secara keseluruhan jika banyak variabel temporer yang sebenarnya hanya merupakan salinan dari variabel lain.



Gambar 13.4: Coalescing: Menghapus MOVE dengan menyatukan node dalam graf interferensi

13.7 Optimasi Lanjut dan Peran SSA

Bentuk ***Static Single Assignment (SSA)*** telah merevolusi cara kompilator melakukan optimasi, termasuk alokasi register.

13.7.1 Dampak SSA terhadap Graf Interferensi

Dalam bentuk SSA, setiap variabel hanya didefinisikan satu kali. Hal ini memiliki beberapa properti menarik bagi pengalokasi register:

- **Chordal Graphs**: Graf interferensi yang dihasilkan dari kode berbasis SSA seringkali bersifat *chordal*. Untuk graf jenis ini, masalah pewarnaannya tidak lagi NP-Complete dan dapat diselesaikan dalam waktu polinomial untuk hasil yang optimal.

- **Liveness Analysis**: Properti SSA menyederhanakan perhitungan rentang hidup variabel (*live ranges*) karena tidak ada ambiguitas definisi variabel.

13.7.2 Penamaan Ulang Register (Register Renaming)

Register Renaming pada tingkat perangkat lunak (kompilator) bertujuan menghilangkan ketergantungan semu (*WAR - Write-After-Read* dan *WAW - Write-After-Write*).

- Dengan menggunakan nama variabel yang berbeda untuk setiap definisi (seperti pada SSA), kompilator memberi kebebasan lebih bagi pengalokasi register untuk menggunakan register yang berbeda, sehingga meningkatkan potensi eksekusi paralel.

Gambar 13.5: Peran SSA sebagai pondasi optimasi dan alokasi register

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

Activation Record (disebut juga *stack frame*) adalah struktur data yang dibuat setiap kali sebuah fungsi dipanggil. Struktur ini menyimpan semua informasi yang diperlukan untuk eksekusi fungsi tersebut dan cara kembali ke fungsi pemanggil (*caller*).

14.1.1 Komponen Utama

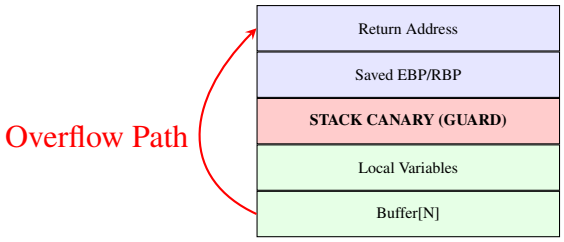
Secara umum, sebuah *activation record* menyimpan:

- **Local Variables**: Variabel yang dideklarasikan di dalam fungsi.
- **Parameters**: Argumen yang dikirimkan oleh pemanggil.
- **Return Address**: Alamat instruksi selanjutnya yang harus dijalankan setelah fungsi selesai.
- **Saved Registers**: Nilai register yang harus dijaga agar tidak rusak oleh fungsi ini.
- **Dynamic Link**: Penunjuk ke basis *stack frame* fungsi pemanggil (*previous FP*).

14.1.2 Keamanan: Stack Canaries

Untuk mencegah serangan ***Buffer Overflow*** yang dapat menimpa alamat kembalian (*return address*), kompilator modern menyisipkan variabel penjaga yang disebut **Stack Canary** [15].

1. **Placement**: Kenari (nilai acak) diletakkan di antara variabel lokal dan data kendali (*return address*).
2. **Verification**: Sebelum kembali (*ret*), kompilator memeriksa apakah nilai kenari masih sama.
3. **Action**: Jika nilai berubah (indikasi *stack smashing*), program akan segera dihentikan (*aborted*).



Gambar 14.1: Penempatan Stack Canary untuk melindungi Return Address

14.2 Mekanisme Panggilan Prosedur (Procedure Call)

Kompilator harus mengikuti standar *Application Binary Interface* (ABI) agar kode yang dihasilkannya dapat berinteraksi dengan perpustakaan (*libraries*) lain.

14.2.1 Konvensi System V x86-64 ABI

Pada sistem Linux x86-64, argumen pertama dikirimkan melalui register untuk performa maksimal [15]:

- `RDI, RSI, RDX, RCX, R8, R9` (untuk argumen bilangan bulat/pointer).
- `XMM0 – XMM7` (untuk argumen *floating-point*).
- Argumen selanjutnya baru diletakkan di dalam *stack*.

14.2.2 Perataan Stack (16-byte Alignment)

ABI mengharuskan *Stack Pointer* (`RSP`) selaras pada kelipatan 16 byte sebelum instruksi `CALL`.

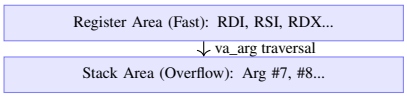
- **Alasan**: Mendukung instruksi vektor modern (SSE/AVX) yang akan mengalami *crash* jika data di stack tidak selaras 16-byte.
- Kompilator menyisipkan *padding* di dalam *activation record* untuk menjamin keselarasan ini.

14.2.3 Fungsi Variadik (va_list)

Fungsi seperti `printf(...)` yang menerima jumlah argumen fleksibel memerlukan penanganan khusus.

1. **Register Save Area**: Di awal fungsi, semua register argumen (`RDI-R9`) disimpan ke area khusus di stack.

2. **va_list**: Adalah struktur data yang menyimpan *offset* ke area penyimpanan register dan pointer ke area stack untuk argumen tambahan.



Gambar 14.2: Layout Argumen untuk Fungsi Variadik

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

```
// 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 Fungsi Bersarang (Nested Functions)

Untuk bahasa seperti Pascal, Ada, atau ekstensi GCC pada C, fungsi dapat dideklarasikan di dalam fungsi lain. Hal ini memerlukan mekanisme untuk mengakses variabel dari lingkup (*scope*) luar.

14.4.1 Static Links (Access Links)

Setiap *activation record* menyimpan pointer ke *frame* fungsi yang secara leksikal membungkusnya.

- **Mekanisme**: Untuk mengakses variabel di level N tingkat ke atas, kompilator menelusuri rantai *static link* sebanyak N kali.
- **Performa**: Semakin dalam penyarangan fungsi, semakin banyak dereferensi memori yang diperlukan.

14.4.2 Display (Tabel Array)

Alternatif performa tinggi menggunakan array global atau register khusus untuk menyimpan pointer ke *activation record* aktif di setiap level penyarangan.

- **Mekanisme**: Akses variabel di level L dilakukan dengan indeks langsung: `Display[L] + offset`.
- **Performa**: Akses sangat cepat (O(1)), namun memiliki overhead lebih besar saat masuk/keluar fungsi karena harus memperbarui entri tabel *Display*.

14.4.3 Lambda Lifting

Banyak kompilator modern (terutama untuk bahasa fungsional) menghindari penyarangan fisik dengan teknik ***Lambda Lifting***: variabel lingkup luar yang digunakan oleh fungsi dalam diubah menjadi parameter eksplisit tambahan.

Fitur	**Static Links**	**Display**
Akses Non-Lokal	O(N) traversal	O(1) direct index
Overhead Call	Rendah (set 1 pointer)	Tinggi (update tabel)
Implementasi	Sederhana	Kompleks

Tabel 14.1: Perbandingan Mekanisme Akses Lingkup Luar

14.5 Penanganan Eksepsi (Exception Handling)

Saat terjadi kesalahan (*error/exception*), program harus dapat "melompat" keluar dari banyak fungsi ke *handler* yang sesuai tanpa merusak struktur data stack.

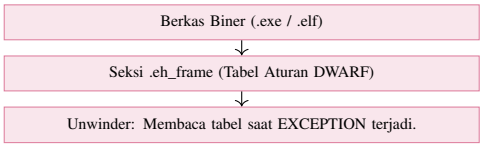
14.5.1 Stack Unwinding

Proses membersihkan *activation records* satu per satu dari atas ke bawah untuk mencari penangkap (*catch block*) yang cocok disebut ***Stack Unwinding***.

14.5.2 Metadata DWARF dan CFI

Kompilator modern seringkali tidak menggunakan *Frame Pointer* (`EBP`) untuk menghemat satu register. Tanpa `EBP`, bagaimana kita bisa menelusuri stack?

- **CFI (Call Frame Information)**: Kompilator menyisipkan tabel metadata di bagian khusus berkas biner (`.eh_frame` atau `.debug_frame`).
- **Aturan Unwinding**: Metadata ini memberikan instruksi kepada *runtime library* (seperti `libgcc`) tentang bagaimana cara menghitung alamat kembalian dan mengembalikan register hanya dengan menggunakan informasi lokasi kode (`Program Counter`).



Gambar 14.3: Penanganan Eksepsi Tanpa Frame Pointer menggunakan DWARF

14.6 Teknik Optimasi Stack

Optimasi pada tingkat manajemen stack sangat krusial karena setiap panggilan fungsi memiliki biaya overhead (*prologue* dan *epilogue*).

14.6.1 Optimasi Fungsi Daun (Leaf Function)

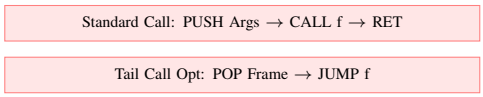
Fungsi daun (*leaf function*) adalah fungsi yang tidak memanggil fungsi lain.

- **Optimasi**: Kompilator seringkali tidak membuat *activation record* sama sekali jika variabel lokal dapat ditampung sepenuhnya dalam register. Instruksi `push` dan `pop` dihilangkan sepenuhnya.

14.6.2 Tail Call vs Tail Recursion

Sebuah *panggilan ekor* ([15]) terjadi jika sebuah fungsi memanggil fungsi lain sebagai tindakan terakhirnya.

- **Tail Recursion**: Fungsi memanggil dirinya sendiri di akhir. Kompilator mengubah rekursi menjadi *loop* biasa (melompat ke awal fungsi).
- **Tail Call**: Fungsi memanggil fungsi BERBEDA di akhir. Kompilator dapat menghapus *frame* saat ini sebelum melakukan `JUMP` (bukan `CALL`) ke fungsi target. Hal ini mencegah pertumbuhan stack yang tidak perlu.



Gambar 14.4: Efisiensi Tail Call Optimization

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

Compiler Tools adalah perangkat lunak pendukung yang mengotomatisasi berbagai tahapan dalam pembuatan kompilator. Penggunaan alat bantu ini memungkinkan pengembang fokus pada logika bahasa daripada rincian implementasi mesin yang repetitif.

15.1.1 Kategori Alat Bantu

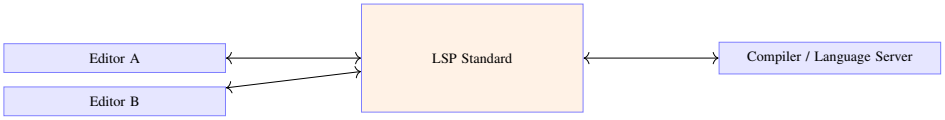
Ekosistem pengembangan kompilator modern melibatkan berbagai standar industri:

- **Lexer Generators (Flex, re2c)**: Mengubah spesifikasi ekspresi reguler menjadi kode C/C++ untuk pengenalan token.
- **Parser Generators (Bison, ANTLR)**: Mengubah tata bahasa bebas konteks (CFG) menjadi penganalisis sintaksis.
- **Frameworks (LLVM, GCC)**: Menyediakan infrastruktur untuk optimasi dan pembuatan kode mesin (*back-end*).
- **Build Systems (Make, CMake)**: Mengelola proses kompilasi kode sumber yang kompleks.

15.1.2 Language Server Protocol (LSP)

Salah satu evolusi paling signifikan dalam sepuluh tahun terakhir adalah munculnya ***Language Server Protocol (LSP)*** [16].

- **Masalah M x N**: Dahulu, setiap editor (M) harus mengimplementasikan dukungan untuk setiap bahasa (N) secara manual.
- **Solusi LSP**: Kompilator bertindak sebagai "Server" yang menyediakan informasi cerdas (*autocompletion, diagnostics, go to definition*) melalui protokol standar berbasis JSON-RPC.
- **Dampak**: Satu *Language Server* dapat digunakan oleh editor mana pun yang mendukung LSP (VS Code, Vim, Emacs), memisahkan kecerdasan bahasa dari antarmuka pengguna.



Gambar 15.1: Efisiensi LSP: Menghubungkan berbagai editor ke satu otak kompilator

15.2 Lexer Generators

15.2.1 Flex (Fast Lexical Analyzer)

Flex adalah lexer generator yang populer:

```
/* 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); }

%%
```

```
20
21 int yywrap(void) {
22     return 1;
23 }
```

15.2.2 re2c

re2c adalah alternatif yang lebih sederhana:

```
1  /* re2c specification */
2  /*!re2c
3  re2c:define:YYCTYPE         unsigned char
4  re2c:define:YYCURSOR        cursor
5  re2c:define:YYLIMIT          limit
6  re2c:define:YYMARKER         marker
7  re2c:define:YYFILL(n)       { memset(cursor, 0, n); cursor += n; }
8
9  NUMBER = [0-9]+;
10 IDENTIFIER = [a-zA-Z][a-zA-Z0-9]*;
11 */
12
13 void scan(char *cursor, char *limit) {
14     char *marker = cursor;
15
16     /*!re2c
17     NUMBER { return NUMBER; }
18     IDENTIFIER { return IDENTIFIER; }
19     "+" { return PLUS; }
20     "*" { return TIMES; }
21     [ \t\r\n]+ { continue; }
22     . { return UNKNOWN; }
23     */
24 }
```

15.3 Parser Generators

15.3.1 Bison (GNU Yacc)

Bison adalah standar industri untuk pembuatan parser berbasis LALR(1). Ia sangat efisien untuk pengolahan tumpukan (*batch processing*) kode sumber yang lengkap.

15.3.2 Tree-sitter: Parsing Inkremental

Baru-baru ini, ***Tree-sitter*** telah menjadi pilihan utama untuk alat bantu pengembang (*editor, linter*). Berbeda dengan Bison, Tree-sitter dirancang untuk:

- **Incremental Parsing**: Hanya memperbarui bagian pohon sintaksis yang berubah saat pengguna mengetik, bukan membedah ulang seluruh berkas.

- **Error Tolerance**: Mampu memberikan struktur pohon yang berguna meskipun kode sedang dalam keadaan tidak lengkap atau memiliki kesalahan sintaksis.

- **CST vs AST**: Tree-sitter menghasilkan *Concrete Syntax Tree* (CST) yang menyimpan setiap detail termasuk spasi dan komentar.

15.3.3 ANTLR (Another Tool for Language Recognition)

ANTLR adalah parser generator modern yang sangat populer di ekosistem Java dan C# karena kemudahannya dalam membangun pohon sintaksis abstrak (AST) secara otomatis menggunakan algoritma ALL(*).

Fitur	**Bison**	**Tree-sitter**
Tujuan	Kompilator (Batch)	Editor/IDE (Real-time)
Kecepatan	Sangat Cepat	Cepat (Incremental)
Penanganan Error	Terhenti/Terbatas	Robust (Tetap jalan)

Tabel 15.1: Bison vs Tree-sitter

15.4 Compiler Frameworks

15.4.1 LLVM (Low Level Virtual Machine)

LLVM adalah kumpulan teknologi kompilator modular yang menyediakan *intermediate representation* (IR) yang sangat dioptimalkan.

15.4.2 MLIR (Multi-Level Intermediate Representation)

MLIR adalah sub-proyek LLVM yang dirancang untuk mengatasi fragmentasi dalam representasi kode [15].

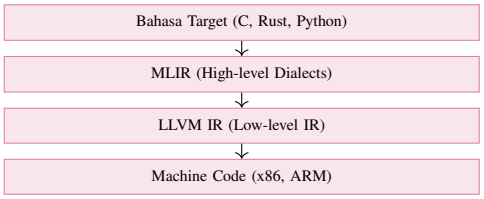
- **Abstraksi Bertingkat**: MLIR memungkinkan kompilator merepresentasikan kode pada berbagai tingkat (dari tingkat tinggi seperti operasi matriks hingga tingkat instruksi mesin).

- **Dialects**: MLIR menggunakan sistem "dialek" yang memungkinkan bahasa baru mendefinisikan operasi kustom mereka sendiri (contoh: dialek *Linalg* untuk aljabar linear atau dialek *TensorFlow*).

- **Efisiensi**: Dengan optimasi yang terjadi pada level dialek yang tepat, informasi semantik tingkat tinggi tidak hilang saat proses penurunan ke kode mesin.

15.4.3 GCC (GNU Compiler Collection)

Meskipun LLVM sangat populer untuk proyek riset, GCC tetap menjadi standar emas untuk sistem Linux tradisional karena kestabilannya dan dukungan arsitektur perangkat keras yang sangat luas [17].



Gambar 15.2: Alur Kompilasi Modern: Dari Dialek MLIR ke Kode Mesin

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 adalah standar industri untuk penelusuran *runtime* kesalahan logika. Sangat berguna untuk melihat jejak tumpukan (*backtrace*) saat kompilator mengalami *segmentation fault*.

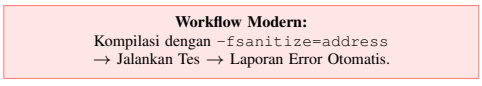
15.6.2 Sanitizers: Pembersihan Otomatis

Bagi pengembang kompilator modern, menggunakan ***Sanitizers*** (seperti ASan dan MSan) seringkali lebih efisien daripada debugging manual [15].

1. **AddressSanitizer (ASan)**: Mendeteksi penggunaan memori setelah dibebaskan (*use-after-free*) dan luapan buffer (*buffer overflow*) secara otomatis dengan *slowdown* minimal.
2. **MemorySanitizer (MSan)**: Mendeteksi pembacaan nilai dari memori yang belum diinisialisasi.
3. **ThreadSanitizer (TSan)**: Membantu mendeteksi *race conditions* jika kompilator mendukung pemindaian paralel.

15.6.3 Valgrind

Meskipun lebih lambat dari ASan, Valgrind tetap menjadi alat utama untuk deteksi kebocoran memori (*memory leaks*) karena tidak memerlukan kompilasi ulang kode sumber.



Gambar 15.3: Debugging Cepat menggunakan Sanitizers

15.7 Performance Analysis

15.7.1 FlameGraphs: Visualisasi Bottleneck

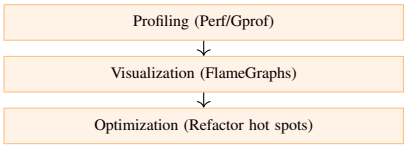
Kompilator adalah aplikasi yang sangat haus sumber daya. Untuk mengidentifikasi fungsi mana yang paling lambat, pengembang menggunakan ***FlameGraphs***.

- **Visual**: Memberikan representasi visual dari pohon panggilan (*call stacks*). Semakin lebar sebuah kotak, semakin banyak waktu CPU yang dihabiskan dalam fungsi tersebut.
- **Manfaat**: Memungkinkan identifikasi instan terhadap bagian optimasi kompilator yang terlalu membebani performa.

15.7.2 Infrastruktur: Reproducible Builds

Dalam proyek kompilator skala besar, konsistensi lingkungan sangat penting.

- **Docker**: Digunakan untuk membungkus seluruh *toolchain* kompilator agar setiap pengembang dan server CI/CD menggunakan versi GCC/LLVM, Flex, dan Bison yang identik. Hal ini menjamin biner yang dihasilkan selalu konsisten (*reproducible*).



Gambar 15.4: Siklus Optimasi Performa Kompilator

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 Metodologi Benchmarking

16.2.1 Desain Pengujian (Benchmark Design)

Benchmarking bukan sekadar menjalankan program dan mencatat waktu. Hasil yang akurat memerlukan metodologi yang ketat untuk meminimalkan anomali data [15].

Prinsip-prinsip benchmark yang baik:

- **Reproducibility**: Jika dijalankan berkali-kali di mesin yang sama, hasilnya harus konsisten.
- **Fairness**: Semua kandidat harus diuji dengan beban kerja dan lingkungan yang identik.
- **Representative**: Kasus uji harus mencerminkan beban kerja yang akan dihadapi pengguna di dunia nyata.
- **Statistical Significance**: Menggunakan rata-rata dari banyak iterasi daripada satu kali percobaan.

16.2.2 Siklus Pemanasan (Warm-up Cycle)

Salah satu kesalahan pemula adalah langsung mengukur eksekusi pertama.

- **Masalah**: Eksekusi pertama seringkali lebih lambat karena *cold caches* (data belum ada di L1/L2), *branch predictor* yang belum terlatih, dan *disk I/O* yang lambat.
- **Solusi**: Jalankan program beberapa kali sebelum mulai mengambil data (*warm-up phase*) agar performa mencapai kondisi stabil (*steady state*).

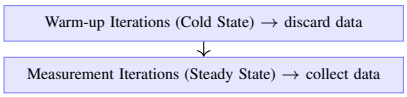
16.2.3 Derau Sistem (System Noise)

Gangguan dari latar belakang dapat merusak data benchmark.

- **Penyebab**: Proses latar belakang (OS *updates*, peramban web), *frequency scaling* (CPU yang menurunkan kecepatan karena panas), dan interupsi jaringan.
- **Mitigasi**: Matikan aplikasi yang tidak perlu, gunakan *performance governor* yang statis pada Linux, dan jalankan benchmark di lingkungan terisolasi.

16.3 Alat Ukur Kinerja (Measurement Tools)

Untuk mendapatkan analisis yang mendalam, kita memerlukan alat yang dapat melihat ke dalam mesin (perangkat keras) maupun perangkat lunak.



Gambar 16.1: Alur Pengukuran yang Akurat

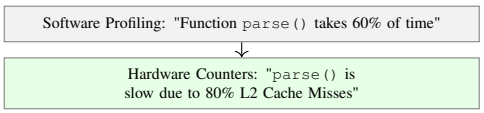
16.3.1 Pengukuran Waktu dan Memori

Kompilator diukur berdasarkan seberapa cepat ia memproses kode (*compilation speed*) dan seberapa cepat kode yang dihasilkannya berjalan (*runtime speed*). Penggunaan memori puncak (*Peak RSS*) sangat penting untuk memastikan kompilasi tidak menyebabkan sistem kehabisan RAM.

16.3.2 Hardware Performance Counters (HPC)

Alat profil perangkat lunak standar memberi tahu kita fungsi mana yang lambat, tetapi ***Hardware Performance Counters (HPC)*** memberi tahu kita **mengapa** fungsi tersebut lambat [15]. Alat seperti `perf` pada Linux memungkinkan akses ke register perangkat keras di dalam CPU untuk mengukur:

- **Cache Misses**: Seberapa sering CPU harus menunggu data dari RAM karena tidak ada di cache L1/L2.
- **Branch Mispredictions**: Seberapa sering unit prediksi cabang CPU salah menebak alur instruksi (sangat berguna untuk mengoptimalkan *switch-case* pada parser).
- **IPC (Instructions Per Cycle)**: Menunjukkan efisiensi penggunaan instruksi per siklus detak CPU.



Gambar 16.2: Keunggulan Analisis Perangkat Keras (HPC)

16.4 Perbandingan Kompilator dan Tingkat Optimasi

16.4.1 Tingkat Optimasi Standar

Kompilator modern memiliki bendera (*flags*) standar:

- `-O0`: Tanpa optimasi (untuk debugging cepat).

- `-O2`: Optimasi moderat tanpa kompromi ukuran kode.
- `-O3`: Optimasi agresif (vektorisasi, unrolling).
- `-Os`: Optimasi untuk ukuran biner terkecil.

16.4.2 Link Time Optimization (LTO)

Kompilasi tradisional bekerja secara independen per berkas sumber. Hal ini mencegah optimasi lintas berkas (*cross-module*). ***Link Time Optimization (LTO)*** menunda pembuatan kode final hingga tahap penautan (*linking*):

- **Whole-Program Analysis**: Kompilator dapat melihat seluruh program sebagai satu kesatuan.
- **Cross-file Inlining**: Fungsi dari `file_a.c` dapat disisipkan (*inlined*) ke dalam `file_b.c`.
- **Dead Code Elimination**: Menghapus kode yang benar-benar tidak pernah dipanggil di seluruh proyek.

Gambar 16.3: Alur LTO untuk Optimasi Seluruh Program

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

```python
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 Analisis Dampak Optimasi

Optimasi tingkat lanjut tidak hanya bergantung pada logika kode, tetapi juga pada bagaimana kode tersebut dieksekusi di dunia nyata.

16.6.1 Profile Guided Optimization (PGO)

PGO (disebut juga *Feedback-Directed Optimization*) menggunakan data performa dari ekosistem nyata untuk memandu kompilator [18]. Siklus PGO:

1. **Instrumentation**: Kompilasi program dengan penanda (*probes*) untuk mencatat jalur mana yang sering dilewati (*hot paths*).
2. **Profiling**: Menjalankan program yang telah ditandai dengan beban kerja representatif.
3. **Optimized Build**: Kompilasi ulang menggunakan data profil tersebut untuk memprioritaskan optimasi pada fungsi yang paling sering digunakan.

16.6.2 BOLT (Binary Optimization and Layout Tool)

Setelah kode ditautkan (*post-link*), alat seperti ***BOLT*** dapat mengoptimalkan biner yang sudah jadi.

- **Reorganisasi Biner**: BOLT mengatur ulang tata letak instruksi fisik di dalam berkas eksekusi agar fungsi-fungsi yang sering memanggil satu sama lain diletakkan berdekatan.
- **Manfaat**: Mengurangi *Instruction Cache Misses* dan meningkatkan akurasi prediksi cabang secara signifikan melampaui kemampuan kompilator standar.

Gambar 16.4: Optimasi Berbasis Umpan Balik (Feedback-Driven)

16.7 Otomasi Benchmark

Dalam pengembangan kompilator modern, performa harus dipantau secara otomatis (CI/CD) untuk mencegah degradasi kinerja secara tidak sengaja.

16.7.1 Pelacakan Regresi Performa

Setiap kali ada perubahan kode sumber (*commit*), sistem otomatis harus menjalankan *benchmark suite*. Jika performa turun di bawah ambang batas (*threshold*) tertentu, sistem akan menandainya sebagai ***Performance Regression***.

16.7.2 Teknik Bisection untuk Performa

Jika terjadi penurunan kinerja yang signifikan, kita menggunakan teknik **Bisection** [19].

- **Prinsip**: Melakukan pencarian biner (*binary search*) pada riwayat perubahan kode untuk menemukan *commit* tunggal mana yang menyebabkan perlambatan.
- **Automated Bisect**: Kompilator seperti LLVM menggunakan alat otomatis yang membangun versi kompilator di tengah rentang waktu tertentu, menjalankan benchmark, dan mengulangi proses hingga penyebab utama ditemukan.

Gambar 16.5: Proses Bisection untuk Mencari Penurunan Performa

16.8 Metrik Kinerja Dunia Nyata

16.8.1 Benchmark Industri: SPEC CPU2017

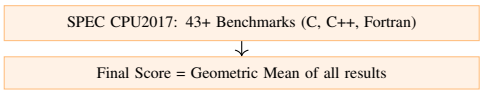
Untuk mengevaluasi kompilator secara profesional, industri menggunakan ***SPEC CPU2017***. Ini adalah kumpulan aplikasi dunia nyata (ditulis dalam C, C++, dan Fortran) yang dirancang untuk menguras kemampuan komputasi CPU dan memori [20].

- **SPECspeed**: Mengukur seberapa cepat sistem menyelesaikan satu tugas instruksi intensif.
- **SPECrate**: Mengukur *throughput* (seberapa banyak tugas yang dapat diselesaikan secara paralel).
- **Integer vs Floating Point**: Memisahkan pengujian berdasarkan jenis operasi matematika untuk akurasi profil.

16.8.2 Interpretasi Data: Geometric Mean

Saat membandingkan performa di banyak kasus uji (seperti pada SPEC), skor akhir tidak dihitung dengan rata-rata aritmatika biasa, melainkan dengan **Geometric Mean**.

- **Alasan**: Rata-rata geometris tidak mudah dipengaruhi oleh satu hasil yang sangat ekstrem (*outliers*).
- **Keadilan**: Menjamin bahwa peningkatan performa 10% pada program kecil memiliki bobot yang sama dengan peningkatan 10% pada program besar, sehingga data perbandingan menjadi lebih adil dan representatif.



Gambar 16.6: Standar Evaluasi Kinerja Kompilator Profesional

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

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

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

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 Teknis

Bagian 1: Arsitektur dan Trade-offs

1. Setelah mempelajari interpretasi dan kompilasi, dalam skenario apa Anda akan memilih *Just-In-Time* (JIT) compiler dibandingkan kompilasi AOT (*Ahead-of-Time*)?

2. Analisis *trade-off* antara kecepatan kompilasi (`-O0`) dan performa waktu eksekusi (`-O3`). Kapan kecepatan kompilasi menjadi lebih penting bagi pengembang?

3. Mengapa transisi dari komponen buatan tangan (*hand-written*) ke generator otomatis (seperti Flex/Bison) dianggap sebagai langkah krusial dalam produktivitas pengembangan kompilator?

Bagian 2: Keterampilan Praktis dan Implementasi

1. Implementasi fase mana (Lexer, Parser, atau Semantik) yang memberikan wawasan terdalam bagi Anda tentang cara kerja bahasa pemrograman?

2. Bagaimana pengalaman menangani kesalahan (*error handling*) di tingkat parser mengubah cara Anda menulis kode yang "aman" secara umum?

3. Hubungan apa yang paling berharga antara teori *Computer Science* (seperti Finite Automata) dengan praktik rekayasa yang Anda lakukan di laboratorium?

Bagian 3: Pandangan Visioner

1. Bagaimana pemahaman tentang representasi intermediet (IR) seperti SSA membantu Anda memahami cara kerja alat bantu modern seperti *Linters* atau *Static Analyzers*?

2. Apa manfaat terbesar dari mempelajari manajemen stack dan register bagi Anda saat melakukan *debugging* aplikasi tingkat sistem?

17.3.2 Self-Assessment Checklist Kompetensi

Kompetensi Teknis	Belum	Sedang	Menguasai
Memahami siklus hidup instruksi dari sumber ke biner	☐	☐	☐
Membangun Parser yang mampu melakukan *error recovery*	☐	☐	☐
Melakukan *Static Type Checking* pada AST	☐	☐	☐
Menghasilkan IR dan menerapkan *Optimization Passes*	☐	☐	☐
Menganalisis performa biner menggunakan `perf` atau `valgrind`	☐	☐	☐

Tabel 17.7: Self-Assessment Kompetensi Akhir Semester

17.4 Portofolio Proyek

Portofolio bukan hanya sekadar kumpulan kode, melainkan bukti kemampuan rekayasa (*engineering capability*) Anda di hadapan dunia profesional atau akademis.

17.4.1 Struktur Portofolio Teknis

Untuk proyek ***Subset C***, portofolio Anda sebaiknya mencakup elemen mendalam berikut:

1. **Analisis Kedalaman Teknis**: Jelaskan satu algoritma kompleks yang Anda implementasikan (misalnya: *Register Coloring* atau *Liveness Analysis*). Mengapa algoritma itu dipilih?

2. **Benchmarking Performa**: Sertakan grafik perbandingan antara biner yang dihasilkan dengan optimasi *-O0* vs *-O3*. Jelaskan mengapa biner yang satu lebih cepat (misal: karena berkurangnya instruksi memori).

3. **Dokumentasi Arsitektur**: Gunakan diagram blok untuk menunjukkan bagaimana data mengalir dari *Flex*, ke *Bison*, melalui penganalisis semantik, hingga ke generator kode LLVM/Assembly.

4. **Metodologi Pengujian**: Tunjukkan suite pengujian otomatis Anda, termasuk kasus uji untuk (*Edge Cases*) seperti pembagian dengan nol atau *scoping* yang tumpang tindih.

Aspek	Bobot
Koreksi Output (Kepatuhan terhadap Spesifikasi)	30%
Kualitas Kode (Modularitas, Kerapuhan, Dokumentasi)	20%
Analisis Performa (Data Benchmarking Nyata)	25%
Kedalaman Refleksi (Pemahaman atas Keputusan Desain)	25%

Tabel 17.8: Kriteria Penilaian Portofolio Akhir

17.4.2 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 Langkah Selanjutnya

17.6.1 Pencapaian Dasar

Teknik Kompilasi telah memberikan Anda fondasi untuk memahami bagaimana abstraksi tingkat tinggi diterjemahkan menjadi realitas fisik di dalam prosesor. Kemampuan ini adalah "kekuatan super" bagi pengembang perangkat lunak karena Anda tidak lagi melihat kompilator sebagai Kotak Hitam (*Black Box*).

17.6.2 Rekomendasi Lanjutan dan Tren Masa Depan

Ekosistem pengembangan kompilator berkembang sangat pesat. Beberapa area yang layak untuk Anda jelajahi lebih lanjut meliputi:

- **AI dalam Kompilator**: Penggunaan *Large Language Models* (LLM) untuk membantu pembuatan kode otomatis, *bug fixing*, dan penemuan strategi optimasi baru yang tidak terpikirkan oleh algoritma heuristik tradisional.
- **WebAssembly (WASM)**: Memahami bagaimana kompilator memungkinkan bahasa seperti C++ atau Rust berjalan di dalam peramban web dengan kecepatan mendekati aplikasi asli (*near-native*).
- **GraalVM dan JIT modern**: Mempelajari bagaimana kompilator dapat melakukan optimasi dinamis saat program SEDANG berjalan untuk hasil yang bahkan lebih baik daripada kompilasi statis.
- **Verified Compilers (Contoh: CompCert)**: Bagi Anda yang tertarik pada keamanan kritis (*mission-critical*), pelajari bagaimana kompilator dapat dibuktikan secara matematis tidak akan pernah menghasilkan kode biner yang salah atau tidak sesuai dengan spesifikasi aslinya.

"The best way to learn compiler construction is to build a compiler." - Andrew Appel. Semoga perjalanan Anda di dunia *Computer Science* semakin cemerlang setelah menaklukkan mata kuliah ini.

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 Instrumen Penilaian OBE

18.2.1 Rubrik Proyek Kompilator (Berdasarkan Sub-CPMK)

Penilaian proyek akhir menggunakan standar *Outcome-Based Education* (OBE) untuk mengukur kompetensi teknis mahasiswa secara spesifik [15].

Kriteria	Sangat Baik (A)	Baik (B)	Cukup (C)	Kurang (D/E)
Lexer/Parser (Sub-CPMK 2.1-2.4)	Grammar kompleks, *error recovery* sempurna	Grammar benar, *error reporting* jelas	Grammar sederhana, *recovery* minim	Banyak error, tidak bisa parsing
Analisis Semantik (Sub-CPMK 3.1-3.3)	*Type checking* dan *Inference* lengkap	*Scoping* dan *Type checking* benar	Hanya *symbol table* dasar	Tidak ada *contextual validation*
Optimasi & IR (Sub-CPMK 4.1-4.3)	Optimasi lokal & global (DCE, Folding)	Menggunakan TAC standar	IR dasar tanpa optimasi	Langsung ke target tanpa IR
Kualitas Kode	Modular, terdokumentasi, *pipeline* CI/CD	Bersih, terdokumentasi manual	Sedikit berantakan, dokumentasi minim	Kode sulit dibaca, tanpa komentar

Tabel 18.1: Rubrik Penilaian Proyek Berbasis OBE

18.2.2 Lembar Penilaian Rekan (*Peer Review*)

Mahasiswa diharapkan memberikan umpan balik konstruktif terhadap portofolio rekan sekelas.

Checklist Pencapaian Kompetensi

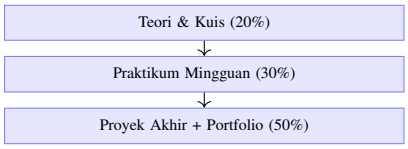
Centang item berikut setelah Anda yakin telah menguasainya:

☐ **Alur Logika**: Apakah desain parser mudah diikuti?

☐ **Penanganan Kesalahan**: Apakah sistem memberikan pesan error yang bermakna bagi pengguna?

☐ **Efisiensi**: Apakah penggunaan memori dan register dipertimbangkan?

☐ **Dokumentasi**: Apakah *README* cukup jelas untuk menjalankan proyek dari nol?



Gambar 18.1: Distribusi Bobot Penilaian Kompetensi

18.3 Checklist Kualitas Teknis

Checklist ini dirancang untuk membantu mahasiswa dan asisten praktikum dalam memverifikasi kedalaman teknis dari implementasi kompilator.

18.3.1 Checklist Analisis Semantik (Sub-CPMK 3)

☐ **Symbol Table Hierarchy**: Apakah tabel simbol menangani lingkup berjenjang (*nested scopes*) dengan benar?

- ☐ **Type Coercion**: Apakah ada aturan untuk konversi otomatis (misal: `int` ke `float`) atau penolakan tipe yang tidak kompatibel?
- ☐ **Recursive Definitions**: Apakah fungsi rekursif terdaftar di tabel simbol sebelum tubuh fungsinya dianalisis?
- ☐ **Contextual Constraints**: Verifikasi apakah variabel digunakan sebelum dideklarasikan dan apakah fungsi dipanggil dengan jumlah argumen yang tepat.

18.3.2 Checklist Optimasi dan IR (Sub-CPMK 4)

- ☐ **Semantics Preserving**: Apakah kode hasil optimasi menghasilkan output yang identik dengan kode asli untuk input yang sama?
- ☐ **Constant Folding**: Apakah ekspresi seperti `3 + 4` dievaluasi pada saat kompilasi menjadi `7`?
- ☐ **Dead Code Elimination**: Apakah blok kode yang tidak terjangkau (*unreachable*) atau variabel yang tidak digunakan berhasil dihapus?
- ☐ **Loop Invariant**: Verifikasi apakah perhitungan yang tidak berubah di dalam loop berhasil dipindahkan ke luar loop.

18.3.3 Checklist Generasi Kode (Sub-CPMK 5)

- ☐ **Activation Records**: Apakah alokasi stack frame sesuai dengan ABI (misal: perataan 16-byte pada x86-64)?
- ☐ **Register Handling**: Apakah kompilator menangani *Register Spilling* jika jumlah variabel melebihi jumlah register fisik yang tersedia?
- ☐ **System Calls**: Apakah instruksi `print` atau `input` terhubung dengan benar ke standar `libc` atau *syscall* sistem operasi?

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

18.7 Panduan Praktis: Debugging dengan Sanitizers

Mencari kesalahan memori atau perilaku tidak terdifinisi (*Undefined Behavior*) secara manual sangatlah sulit. Kompilator modern menyediakan fitur ***Sanitizers*** untuk mendeteksi masalah ini saat program dijalankan.

18.7.1 Mengaktifkan Sanitizer

Tambahkan bendera berikut saat mengompilasi kode C Anda menggunakan GCC atau Clang:

```
# ASan (AddressSanitizer) untuk kesalahan memori
# UBSan (UndefinedBehaviorSanitizer) untuk kesalahan logika standar
gcc -fsanitize=address,undefined -g source.c -o my_compiler
```

18.7.2 Kesalahan yang Terdeteksi

- **Buffer Overflow**: Mengakses array di luar batas yang ditentukan.
- **Use-After-Free**: Mengakses memori yang telah dibebaskan (*free*).
- **Memory Leaks**: Alokasi memori (*malloc*) yang tidak pernah dibebaskan.
- **Integer Overflow**: Hasil operasi aritmatika yang melebihi kapasitas tipe data.

18.7.3 Interpretasi Laporan

Jika terjadi kesalahan, program akan berhenti dan mencetak *stack trace* yang menunjukkan baris kode sumber yang bermasalah.

```
==1234==ERROR: AddressSanitizer: heap-buffer-overflow on address ...
READ of size 4 at 0x603000000010 thread T0
    #0 0x400c47 in main (source.c:15)
```

Laporan ini memberi tahu Anda bahwa baris 15 pada `source.c` mencoba membaca di luar batas memori yang dialokasikan.

18.8 Glosarium Istilah Teknik Kompilasi

Kumpulan istilah penting yang digunakan di sepanjang buku ini beserta definisinya.

Activation Record (*Stack Frame*): Struktur data yang menyimpan informasi pemanggilan fungsi (parameter, variabel lokal, alamat kembali).

AST (Abstract Syntax Tree) : Representasi pohon dari struktur sintaksis kode sumber, mengabaikan detail tanda baca yang tidak perlu.

Basic Block : Urutan instruksi yang hanya memiliki satu titik masuk (di awal) dan satu titik keluar (di akhir).

Calling Convention : Standar yang mengatur bagaimana parameter dikirimkan dan register dikelola saat fungsi dipanggil (misal: System V x86-64 ABI).

Constant Folding : Optimasi yang mengevaluasi ekspresi nilai konstan pada saat kompilasi.

Dead Code Elimination : Proses penghapusan kode yang tidak pernah dieksekusi atau hasilnya tidak pernah digunakan.

Liveness Analysis : Analisis untuk menentukan variabel mana yang masih dibutuhkan (*live*) di setiap titik dalam program.

Peep-hole Optimization (*Lubang Intip*): Optimasi lokal pada jendela kecil instruksi target untuk mengganti urutan instruksi yang tidak efisien.

Register Spilling : Proses menyimpan nilai dari register ke memori (*stack*) karena keterbatasan jumlah register fisik.

SSA (Static Single Assignment) : Bentuk antara di mana setiap variabel hanya ditetapkan nilainya tepat satu kali.

Three-Address Code (TAC) : Representasi antara di mana setiap instruksi memiliki maksimal tiga alamat (dua sumber, satu tujuan).

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

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

- **MLIR: A Compiler Infrastructure for the End of Moore's Law**
  Lattner, Chris, et al.
  arXiv preprint arXiv:2002.11054, 2020

- **Large Language Models for Compiler Optimization**
  Cummins, Chris, et al.
  arXiv preprint arXiv:2309.07062, 2023

- **Formal Verification of a Realistic Compiler**
  Leroy, Xavier
  Communications of the ACM, 2009

- **Otomata Bahasa dan Teknik Kompilasi**
  Vulandari, Retno Tri
  Penerbit Informatika, 2017
  ISBN: 978-6026232410

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/

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

- **Compiler Explorer**
  Matt Godbolt
  https://godbolt.org/

- **Wasmtime: A small and efficient runtime for WebAssembly**
  Bytecode Alliance
  https://wasmtime.dev/

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/

- **WebAssembly Core Specification**
  W3C Working Group - Living Standard
  https://www.w3.org/TR/wasm-core/

- **Language Server Protocol Specification**
  Microsoft
  https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/

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 et al. *Compilers: Principles, Techniques, and Tools*. 2nd. Dragon Book. Pearson Education, 2006.

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

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

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

[6] 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/.

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

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

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

[14] 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.

[15] 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.

[16] 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/.

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

[18] Fedora Project. *Changes/PGO_by_default*. Documentation on Profile Guided Optimization. 2024. URL: https://fedoraproject.org/wiki/Changes/PGO_by_default.

[19] Asher Mancinelli. *Bisecting Performance Regressions in LLVM*. Technical blog post. 2024. URL: https://ashermancinelli.com/bisecting-llvm-performance.

[20] Standard Performance Evaluation Corporation (SPEC). *SPEC CPU 2017*. Industry standard benchmark suite. 2024. URL: https://www.spec.org/cpu2017/.