Bab 16

Project Final Presentation dan Review

16.1 Tujuan Pembelajaran

Bab ini memfokuskan presentasi dan review **compiler subset C** yang telah dibangun secara bertahap dari Bab 2 hingga Bab 15: spesifikasi token dan grammar (Bab 1, 2, 5), lexer (Bab 3, 4), parser (Bab 6, 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). Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Mempresentasikan project final (compiler subset C) dengan baik
2. Mendemonstrasikan kemampuan compiler subset C yang telah dibangun
3. Mengevaluasi dan membandingkan tools kompilator (parser generators vs hand-written parsers)
4. Menganalisis trade-off antara waktu kompilasi, kualitas kode, dan efisiensi runtime
5. Melakukan refleksi pembelajaran dan evaluasi diri terhadap seluruh materi
6. Menyusun dokumentasi proyek yang komprehensif

16.2 Pendahuluan

Bab ini merupakan puncak dari pembelajaran mata kuliah Teknik Kompilasi. Setelah mempelajari semua fase kompilasi dari Analisis Leksikal (Lexical Analysis) hingga Optimasi (Optimization), mahasiswa diharapkan telah membangun sebuah kompilator lengkap yang dapat mengkompilasi bahasa sederhana menjadi executable code.

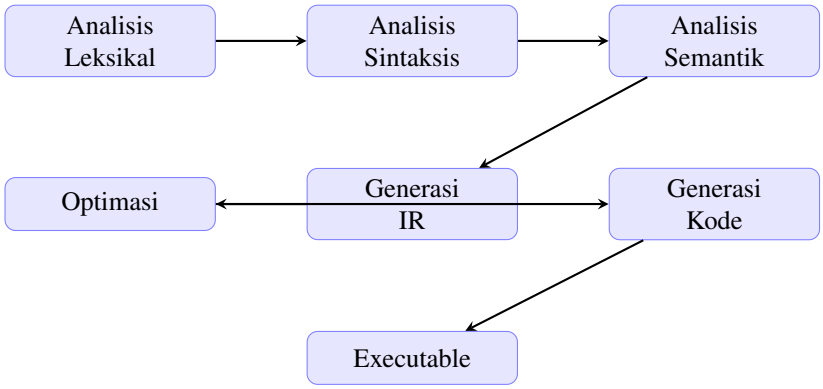
Menurut sumber dari University of Oxford:

> "Evaluate and compare compiler tools (like parser generators) and optimization approaches, analyze trade-offs between compilation time, code quality, and runtime efficiency."[1]

Project final ini tidak hanya menguji kemampuan teknis, tetapi juga kemampuan untuk:

- Mengintegrasikan semua komponen yang telah dipelajari
- Membuat keputusan desain yang tepat
- Mengevaluasi tools dan teknik yang digunakan
- Berkomunikasi secara efektif tentang hasil kerja

Gambar 16.1 menunjukkan arsitektur lengkap kompilator yang harus dibangun dalam project final.



Gambar 16.1: Arsitektur lengkap kompilator untuk project final

16.3 Persiapan Presentasi Project Final

Presentasi project final adalah kesempatan untuk menunjukkan hasil kerja keras selama satu semester. Berikut adalah panduan untuk mempersiapkan presentasi yang efektif:

16.3.1 Struktur Presentasi

Presentasi sebaiknya mengikuti struktur berikut:

1. Pendahuluan (5 menit)

- Perkenalan tim dan project
- Tujuan dan scope bahasa yang dikompilasi
- Overview arsitektur compiler

2. Demo Compiler (10 menit)

- Live demonstration: compile dan run program contoh

- Menunjukkan berbagai fitur bahasa yang didukung
- Menampilkan error handling yang baik

3. Arsitektur dan Implementasi (15 menit)

- Penjelasan setiap fase kompilasi
- Pilihan desain dan justifikasinya
- Tools dan teknik yang digunakan
- Tantangan yang dihadapi dan solusinya

4. Evaluasi dan Analisis (10 menit)

- Perbandingan hand-written vs generator tools
- Trade-off analysis
- Benchmark hasil kompilasi
- Evaluasi kualitas kode yang dihasilkan

5. Kesimpulan dan Refleksi (5 menit)

- Lesson learned
- Area untuk improvement
- Kesimpulan

6. Q&A (5 menit)

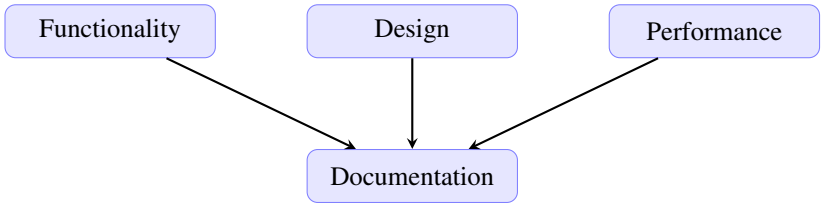
16.3.2 Tips Presentasi yang Efektif

1. **Persiapan Demo**: Pastikan demo berjalan lancar dengan test cases yang sudah dipersiapkan
2. **Visual Aids**: Gunakan diagram arsitektur, flowchart, dan contoh kode yang jelas
3. **Time Management**: Latih presentasi untuk memastikan sesuai waktu yang dialokasikan
4. **Anticipate Questions**: Siapkan jawaban untuk pertanyaan umum tentang desain dan implementasi
5. **Show Passion**: Tunjukkan antusiasme terhadap project yang telah dibangun

16.4 Demonstrasi Compiler

Demo adalah bagian penting dari presentasi. Demo yang baik menunjukkan bahwa compiler benar-benar berfungsi dan dapat digunakan secara praktis.

Gambar 16.2 menunjukkan aspek-aspek evaluasi project final.



Gambar 16.2: Aspek-aspek evaluasi project final

16.4.1 Preparing for Demo

1. Test Cases yang Komprehensif

- Program sederhana (hello world, arithmetic)
- Program dengan kontrol flow (if-else, loops)
- Program dengan fungsi dan scope
- Program dengan error (untuk menunjukkan error handling)
- Program yang lebih kompleks (menunjukkan kemampuan compiler)

2. Environment Setup

- Pastikan semua dependencies terinstall
- Compile compiler terlebih dahulu
- Siapkan backup plan jika ada masalah teknis
- Test di environment yang sama dengan presentasi

3. Demo Script

- Buat script demo yang terstruktur
- Siapkan penjelasan untuk setiap langkah
- Antisipasi kemungkinan error atau masalah

16.4.2 Contoh Demo Flow

Berikut adalah contoh alur demo yang efektif:

1. **Hello World**: Menunjukkan compiler dapat menghasilkan executable sederhana

```
// hello.lang
print("Hello, World!");
```

2. **Arithmetic Operations**: Menunjukkan kemampuan menangani ekspresi

```
// calc.lang
int x = 10;
int y = 20;
int result = x + y * 2;
print(result);
```

3. **Control Flow**: Menunjukkan if-else dan loops

```
// control.lang
int i = 0;
while (i < 10) {
    if (i % 2 == 0) {
        print(i);
    }
    i = i + 1;
}
```

4. **Error Handling**: Menunjukkan kualitas error messages

```
// error.lang
int x = 10;
int y = "string";  // Type error
x = undefined_var; // Undefined variable
```

5. **Complex Program**: Menunjukkan kemampuan compiler dengan program yang lebih kompleks

16.5 Review Materi: Fase-Fase Kompilasi

Sebelum melakukan evaluasi tools dan teknik, penting untuk mereview kembali semua fase kompilasi yang telah dipelajari:

16.5.1 Front-End Phases

1. Lexical Analysis

- Memecah source code menjadi tokens
- Implementasi: hand-written atau menggunakan Flex/re2c
- Output: stream of tokens

2. Syntax Analysis

- Memverifikasi struktur grammar
- Implementasi: recursive descent, LR parser, atau Bison
- Output: Abstract Syntax Tree (AST)

3. Semantic Analysis

- Type checking, scope resolution, name resolution
- Implementasi: tree traversal dengan symbol table
- Output: Annotated AST dengan type information

16.5.2 Back-End Phases

4. Intermediate Code Generation

- Mengkonversi AST menjadi IR (three-address code)
- Output: Intermediate Representation

5. Code Optimization

- Optimasi lokal dan global
- Output: Optimized IR

6. Code Generation

- Mengkonversi IR menjadi target code (assembly)
- Output: Assembly code atau machine code

16.5.3 Integration Points

Setiap fase harus terintegrasi dengan baik:

- Lexer → Parser: Token stream
- Parser → Semantic Analyzer: AST
- Semantic Analyzer → IR Generator: Annotated AST
- IR Generator → Optimizer: IR
- Optimizer → Code Generator: Optimized IR
- Code Generator → Assembler: Assembly code

16.6 Evaluasi Tools: Hand-Written vs Generator-Based

Salah satu aspek penting dalam project final adalah evaluasi tools yang digunakan. Mahasiswa perlu membandingkan pendekatan hand-written dengan generator-based tools.

16.6.1 Perbandingan Lexer: Hand-Written vs Flex/re2c

Hand-Written Lexer

Keuntungan:

- Kontrol penuh atas implementasi
- Error messages yang lebih informatif dan customizable
- Tidak ada dependency eksternal
- Dapat dioptimasi untuk kasus khusus
- Lebih mudah di-debug karena kode lebih readable

Kekurangan:

- Lebih banyak waktu development
- Lebih banyak kode boilerplate
- Lebih mudah terjadi error manual
- Perlu implementasi state machine secara manual

Generator-Based Lexer (Flex/re2c)

Keuntungan:

- Development lebih cepat
- Grammar specification lebih declarative
- Automatically generates efficient DFA
- Less boilerplate code
- Proven algorithms (Thompson's construction, subset construction)

Kekurangan:

- Generated code sulit di-debug
- Error messages kurang informatif
- Dependency pada tool eksternal
- Kurang fleksibel untuk kasus edge yang kompleks
- Build process lebih kompleks

16.6.2 Perbandingan Parser: Hand-Written vs Bison/Yacc

Hand-Written Parser (Recursive Descent)

Keuntungan:

- Kode lebih readable dan mudah di-maintain
- Error recovery yang lebih baik dan customizable
- Tidak ada dependency eksternal
- Dapat menangani grammar yang kompleks dengan mudah
- Lebih mudah di-debug

Kekurangan:

- Hanya cocok untuk LL(1) grammar
- Lebih banyak kode manual
- Lebih mudah terjadi error dalam implementasi

Generator-Based Parser (Bison/Yacc)

Keuntungan:

- Mendukung LR, LALR, GLR parsing

- Automatic table generation
- Grammar specification lebih declarative
- Proven algorithms
- Mendukung grammar yang lebih kompleks

Kekurangan:

- Generated code sulit di-debug
- Error messages kurang informatif
- Dependency pada tool eksternal
- Build process lebih kompleks
- Kurang fleksibel untuk error recovery yang kompleks

16.6.3 Case Study: Real-World Compilers

Banyak compiler production menggunakan pendekatan hybrid atau beralih dari generator ke hand-written:

GCC (GNU Compiler Collection)

- Menggunakan hand-written recursive descent parser untuk C dan C++
- Alasan: Better error messages, easier maintenance, better handling of complex grammar
- Trade-off: Lebih banyak kode manual, tetapi lebih maintainable

Clang (LLVM Compiler)

- Menggunakan hand-written parser
- Alasan: Superior error diagnostics, better recovery
- Hasil: Error messages yang sangat informatif dan helpful

Many Educational Compilers

- Menggunakan Flex + Bison untuk pembelajaran
- Alasan: Lebih cepat untuk development, fokus pada konsep bukan implementasi detail
- Cocok untuk: Prototyping, learning, small languages

16.7 Analisis Trade-Off

Menurut sumber dari University of Oxford[1], evaluasi compiler tools harus mempertimbangkan trade-off antara berbagai aspek:

16.7.1 Compilation Time vs Code Quality

Pendekatan	Compilation Time	Code Quality
Minimal Optimization	Fast	Lower
Aggressive Optimization	Slow	Higher
Selective Optimization	Medium	Medium-High

Tabel 16.1: Trade-off compilation time vs code quality

Pertimbangan:

- Development phase: Prioritize fast compilation untuk iterasi cepat
- Production build: Prioritize code quality untuk performa runtime
- Selective optimization: Balance antara keduanya

16.7.2 Development Time vs Maintainability

- **Generator Tools**: Faster initial development, tetapi mungkin lebih sulit di-maintain untuk grammar yang kompleks
- **Hand-Written**: Slower initial development, tetapi lebih maintainable dalam jangka panjang

16.7.3 Error Messages Quality

- **Hand-Written**: Dapat menghasilkan error messages yang sangat informatif dan helpful
- **Generator-Based**: Error messages cenderung generic, perlu custom handling untuk improvement

16.7.4 Flexibility vs Correctness

- **Generator Tools**: Enforce grammar constraints, mengurangi human error
- **Hand-Written**: Lebih fleksibel, tetapi lebih mudah terjadi error manual

16.8 Benchmarking dan Evaluasi Kinerja

Evaluasi compiler tidak hanya tentang correctness, tetapi juga tentang performa. Berikut adalah metrik yang dapat digunakan:

16.8.1 Metrik Compiler Performance

1. Compilation Speed

- Waktu kompilasi untuk berbagai ukuran program
- Throughput (lines/second atau tokens/second)
- Memory usage selama kompilasi

2. Generated Code Quality

- Execution time dari program yang dikompilasi
- Code size (executable size)
- Memory footprint
- Instruction count

3. Compiler Correctness

- Test coverage
- Number of bugs found
- Error detection rate

16.8.2 Contoh Benchmark Results

Berikut adalah contoh format untuk melaporkan hasil benchmark:

Metric	Baseline	Optimized	Improvement
Compilation Time (s)	2.5	3.1	-24% (slower)
Generated Code Size (KB)	64	48	25% reduction
Execution Time (ms)	100	75	25% faster
Memory Usage (MB)	8	6	25% reduction

Tabel 16.2: Contoh hasil benchmark compiler

16.8.3 Test Suite

Untuk evaluasi yang komprehensif, siapkan test suite yang mencakup:

1. **Unit Tests**: Test setiap fase secara terpisah
    - Lexer tests: Valid/invalid tokens
    - Parser tests: Valid/invalid syntax
    - Semantic tests: Type checking, scope resolution
    - Code generation tests: Correctness of generated code
2. **Integration Tests**: Test seluruh pipeline
    - End-to-end compilation
    - Error propagation through phases
    - Optimization correctness
3. **Performance Tests**: Test dengan program besar
    - Compilation time
    - Memory usage
    - Generated code performance
4. **Regression Tests**: Test untuk memastikan tidak ada regresi

16.9 Dokumentasi Proyek

Dokumentasi yang baik adalah bagian penting dari project final. Dokumentasi harus mencakup:

16.9.1 README.md

README harus berisi:

- **Overview**: Deskripsi singkat tentang compiler
- **Features**: Fitur-fitur yang didukung
- **Build Instructions**: Cara mengkompilasi compiler
- **Usage**: Cara menggunakan compiler
- **Examples**: Contoh program dan cara mengkompilasinya
- **Architecture**: Overview arsitektur compiler
- **Testing**: Cara menjalankan test suite

16.9.2 Design Document

Design document mencakup:

- **Language Specification**: Grammar, syntax, semantics
- **Architecture Overview**: Diagram arsitektur compiler
- **Component Design**: Desain setiap fase kompilasi
- **Data Structures**: AST nodes, symbol table, IR format
- **Algorithm Choices**: Justifikasi pilihan algoritma
- **Trade-offs**: Diskusi tentang trade-off yang dibuat

16.9.3 API Documentation

Jika compiler menyediakan library atau API:

- Function signatures
- Parameter descriptions
- Return values
- Usage examples

16.10 Refleksi Pembelajaran

Refleksi adalah bagian penting dari proses pembelajaran. Mahasiswa diharapkan melakukan refleksi terhadap:

16.10.1 Technical Skills Acquired

- **Lexical Analysis**: Kemampuan mengimplementasikan lexer
- **Parsing**: Pemahaman tentang grammar dan parsing techniques
- **Semantic Analysis**: Kemampuan melakukan type checking dan scope resolution
- **Code Generation**: Kemampuan menghasilkan target code
- **Optimization**: Pemahaman tentang optimasi kompilator
- **Software Engineering**: Kemampuan mengintegrasikan komponen-komponen besar

16.10.2 Challenges Faced

Identifikasi tantangan yang dihadapi:

- Technical challenges (implementasi, debugging)
- Design challenges (trade-offs, architecture decisions)
- Time management challenges
- Team collaboration challenges (jika project team-based)

16.10.3 Lessons Learned

Rangkum pembelajaran:

- Apa yang bekerja dengan baik?
- Apa yang tidak bekerja seperti yang diharapkan?
- Apa yang akan dilakukan berbeda jika memulai lagi?
- Insight tentang compiler design dan implementation

16.10.4 Areas for Improvement

Identifikasi area untuk improvement:

- Fitur yang belum diimplementasikan
- Optimasi yang dapat ditambahkan
- Error handling yang dapat ditingkatkan
- Dokumentasi yang dapat diperbaiki
- Testing yang dapat diperluas

16.11 Best Practices untuk Project Final

Berdasarkan pengalaman dan best practices dari berbagai compiler projects:

16.11.1 Code Quality

- **Clean Code**: Kode yang readable dan well-structured
- **Modularity**: Komponen yang terpisah dengan jelas
- **Error Handling**: Comprehensive error handling dan reporting
- **Comments**: Dokumentasi yang adequate dalam kode
- **Consistency**: Konsistensi dalam coding style

16.11.2 Testing

- **Test Coverage**: Test coverage yang komprehensif
- **Edge Cases**: Test untuk edge cases dan error conditions
- **Automated Testing**: Automated test suite
- **Regression Testing**: Test untuk mencegah regresi

16.11.3 Documentation

- **Completeness**: Dokumentasi yang lengkap
- **Clarity**: Dokumentasi yang jelas dan mudah dipahami
- **Examples**: Contoh yang membantu
- **Maintenance**: Dokumentasi yang up-to-date

16.11.4 Presentation

- **Preparation**: Persiapan yang matang
- **Clarity**: Presentasi yang jelas dan terstruktur
- **Demo**: Demo yang berjalan lancar
- **Time Management**: Manajemen waktu yang baik

16.12 Kesimpulan

Bab ini telah membahas:

1. **Project Final Presentation**: Struktur dan tips untuk presentasi yang efektif
2. **Demonstrasi Compiler**: Cara mempersiapkan dan melakukan demo yang baik
3. **Review Materi**: Review semua fase kompilasi yang telah dipelajari
4. **Evaluasi Tools**: Perbandingan hand-written vs generator-based tools
5. **Analisis Trade-Off**: Trade-off antara berbagai aspek compiler design
6. **Benchmarking**: Metrik dan cara melakukan evaluasi kinerja
7. **Dokumentasi**: Best practices untuk dokumentasi proyek
8. **Refleksi Pembelajaran**: Cara melakukan refleksi yang konstruktif

Project final adalah kesempatan untuk mengintegrasikan semua pengetahuan yang telah diperoleh selama semester. **Compiler subset C** yang dibangun dari Bab 2 hingga Bab 15—dengan komponen lexer proyek (`simplec.l`), parser proyek (`simplec.y`), AST, symbol table, type checking, IR, code generation, dan optimasi—menjadi bukti nyata penguasaan fase-fase kompilasi. Melalui project ini, mahasiswa tidak hanya menunjukkan kemampuan teknis, tetapi juga kemampuan untuk membuat keputusan desain, mengevaluasi tools dan teknik, dan berkomunikasi secara efektif tentang hasil kerja.

16.13 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang evaluasi compiler dan best practices:

- **University of Oxford Compilers Course**[1]: Materials tentang evaluasi tools dan trade-off analysis
- **Dragon Book**[2]: Bab tentang compiler optimization dan code generation evaluation
- **Engineering a Compiler**[3]: Best practices untuk compiler design dan implementation
- **Compiler Construction Resources**:
  - UC San Diego CSE 231[4]: Course materials dengan project examples
  - Northeastern University CS 4410[5]: Comprehensive compiler design course

  - Johns Hopkins University EN.601.428[6]: Course tentang compilers dan interpreters

- **Open Source Compiler Projects**:
  - TinyCC (TCC): https://bellard.org/tcc/ - Contoh compiler sederhana
  - AnjaneyaTripathi's C Compiler: https://github.com/AnjaneyaTripathi/c-compiler - Educational compiler dengan Flex & Bison
  - LLVM: https://llvm.org/ - Production compiler infrastructure

Selamat! Anda telah menyelesaikan perjalanan pembelajaran Teknik Kompilasi. Project final adalah kesempatan untuk menunjukkan semua yang telah Anda pelajari dan bangun. Semoga sukses dengan presentasi dan project final Anda!

Daftar Pustaka

[1] 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/.

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

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

[4] 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.

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

[6] 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.