Bab 1

Kumpulan Latihan

1.1 Soal

1.1.1 Latihan Bab 1: Pengenalan Kompilator dan Fase-Fase Kompilasi

1. Jelaskan perbedaan antara kompilator dan interpreter. Berikan contoh bahasa pemrograman yang menggunakan masing-masing pendekatan.

2. Buatlah diagram yang menunjukkan alur kerja kompilator dari source code C sampai menjadi executable. Sertakan semua fase yang telah dipelajari.

3. Identifikasi fase kompilasi yang bertanggung jawab untuk:

    - Menghapus whitespace dan comments
    - Memeriksa apakah variabel dideklarasi sebelum digunakan
    - Mengoptimasi kode dengan menghilangkan kode yang tidak pernah dieksekusi
    - Mengkonversi ekspresi `a + b * c` menjadi three-address code

4. Jelaskan mengapa kompilator modern menggunakan pendekatan multi-pass. Apa keuntungannya dibanding single-pass?

5. Carilah informasi tentang satu kompilator open source (misalnya GCC, Clang, atau TinyCC) dan identifikasi:

    - Bahasa sumber dan target yang didukung
    - Tools yang digunakan untuk lexical dan syntax analysis
    - Format intermediate representation yang digunakan

1.1.2 Latihan Bab 2: Regular Expression dan Finite Automata

1. Buatlah regular expression untuk:

- Email address sederhana (format: `user@domain.com`)
- Phone number (format: `+62-812-3456-7890`)
- C-style comment (`/* ... */`)

2. Konstruksi NFA untuk regular expression berikut menggunakan algoritma Thompson:
   - `a*b+`
   - `(a|b)*ab`
   - `[0-9]+(\.[0-9]+)?`
3. Konversi NFA dari soal nomor 2 menjadi DFA menggunakan subset construction. Gambarkan state diagram untuk DFA yang dihasilkan.
4. Implementasikan kelas `NFA` dan `DFA` dalam C++ dengan fungsi:
   - Konstruksi NFA dari regular expression (sederhana)
   - Konversi NFA ke DFA
   - Simulasi DFA untuk string input
5. Buat program recognizer yang dapat mengenali token-token berikut:
   - Identifier: `[a-zA-Z_][a-zA-Z0-9_]*`
   - Integer: `[0-9]+`
   - Float: `[0-9]+\.[0-9]+`
   - Operator: `+|-|*|/|=|==|!=`
6. Jelaskan mengapa DFA lebih efisien untuk simulasi dibanding NFA. Berikan contoh kompleksitas waktu untuk keduanya.
7. Implementasikan algoritma minimisasi DFA (dapat menggunakan versi sederhana). Uji dengan DFA yang dihasilkan dari soal nomor 3.

1.1.3 Latihan Bab 3: Implementasi Lexer

1. **Implementasi Dasar**: Implementasikan lexer sederhana yang dapat mengenali:
   - Identifier (huruf, angka, underscore)
   - Integer literals
   - Operator dasar: `+`, `-`, `*`, `/`, `=`
   - Punctuation: `;`, `,`, `(`, `)`

2. **Handling Comments**: Tambahkan support untuk:
   - Single-line comments (`//`)
   - Multi-line comments (`/* */`)
   - Error handling untuk unclosed block comments
3. **String Literals**: Implementasikan scanning untuk string literals dengan:
   - Support escape sequences: `\n`, `\t`, `\"`, `\\`
   - Error handling untuk unclosed strings
4. **Float Literals**: Extend number scanning untuk mendukung:
   - Float dengan decimal point: `3.14`
   - Scientific notation: `1.5e10`, `2.3E-5`
5. **Unit Testing**: Buat test suite yang mencakup:
   - Valid tokens dari semua kategori
   - Edge cases (unclosed strings, invalid characters, dll.)
   - Position tracking (line dan column)
6. **Error Recovery**: Implementasikan error recovery strategy:
   - Skip invalid characters dan lanjut scanning
   - Report multiple errors dalam satu pass jika memungkinkan
7. **Performance**: Analisis dan optimasi:
   - Bandingkan performa dengan lexer generator (jika tersedia)
   - Identifikasi bottleneck dalam implementasi

1.1.4 Latihan Bab 4: Lexer Generator

1. Buatlah specification file Flex untuk mengenali token-token berikut:
   - Keywords: `for`, `break`, `continue`
   - String literals (dengan escape sequences: `\n`, `\t`, `\"`)
   - Character literals (dalam single quotes)
   - Operators: `++`, `--`, `+=`, `-=`
2. Implementasikan lexer menggunakan re2c untuk bahasa yang sama seperti soal 1. Bandingkan kompleksitas specification antara Flex dan re2c.

3. Jelaskan kapan sebaiknya menggunakan hand-written lexer dan kapan menggunakan generator. Berikan contoh kasus untuk masing-masing.

4. Buatlah program yang mengintegrasikan Flex lexer dengan Bison parser sederhana untuk:

   - Parsing ekspresi aritmatika: `a + b * c`
   - Parsing assignment: `x = 42;`
   - Parsing conditional: `if (x > 0) { ... }`

5. Analisis performa: Buat benchmark untuk membandingkan:

   - Hand-written lexer (dari Bab 3)
   - Flex-generated lexer
   - re2c-generated lexer

   Gunakan input file yang besar (misalnya 10MB) dan ukur waktu eksekusi.

6. Jelaskan bagaimana Flex menangani longest match dan rule priority. Berikan contoh yang menunjukkan perbedaan hasil ketika urutan rule diubah.

1.1.5 Latihan Bab 5: Context-Free Grammar dan Parsing

1. Tuliskan grammar dalam BNF untuk:

   - Ekspresi boolean dengan operator AND, OR, NOT
   - Assignment statement: `variable = expression;`
   - For loop: `for (init; condition; update) statement`

2. Konversi grammar berikut ke EBNF:

```
<list> ::= <item> | <list> , <item>
<item> ::= <number> | <string>
```

3. Buatlah leftmost dan rightmost derivation untuk ekspresi `(2 + 3) * 4` menggunakan grammar ekspresi aritmatika yang telah dipelajari.

4. Gambarkan parse tree untuk ekspresi berikut menggunakan grammar yang sesuai:

   - `1 + 2 * 3`
   - `10 / 2 - 3`
   - `(5 + 3) * 2`

5. Identifikasi apakah grammar berikut ambiguous. Jika ya, berikan contoh string yang dapat di-parse dengan lebih dari satu cara:

```
S → a S a | b S b | a | b | epsilon
```

6. Eliminasi left recursion dari grammar berikut:

```
A → A + B | A - B | B
B → B * C | B / C | C
C → ( A ) | number
```

7. Lakukan left factoring pada grammar berikut:

```
S → if E then S else S
  | if E then S
  | while E do S
  | id = E
```

8. Rancang grammar untuk bahasa sederhana yang mendukung:

   - Variable declarations: `int x;`
   - Assignments: `x = 5;`
   - If-else statements
   - While loops
   - Arithmetic expressions

   Tulis grammar dalam EBNF dan berikan contoh program valid.

1.1.6 Latihan Bab 6: Top-Down Parsing dan Recursive Descent

1. Implementasikan recursive descent parser untuk grammar berikut:

```
S -> if E then S else S | id := E | while E do S
E -> E + T | T
T -> T * F | F
F -> ( E ) | id | num
```

   Perhatikan: Grammar ini memiliki left recursion. Anda perlu mengeliminasi left recursion terlebih dahulu.

2. Modifikasi parser ekspresi aritmatika untuk menambahkan operator unary minus (misalnya `-5` atau `-(3 + 4)`).

3. Implementasikan error recovery pada parser yang telah Anda buat. Test dengan input yang mengandung multiple errors.

4. Buatlah parser yang dapat mengevaluasi ekspresi boolean dengan operator `AND`, `OR`, dan `NOT`. Perhatikan precedence: `NOT` > `AND` > `OR`.

5. Bandingkan recursive descent parser dengan table-driven LL parser. Apa keuntungan dan kerugian masing-masing pendekatan?

6. Implementasikan parser untuk ekspresi dengan right-associative operator (misalnya operator assignment =).

1.1.7 Latihan Bab 7: Bottom-Up Parsing dan Parser Generator

1. Jelaskan perbedaan antara top-down dan bottom-up parsing. Berikan contoh situasi di mana masing-masing lebih cocok digunakan.

2. Untuk grammar berikut:

```
E -> E + T | T
T -> T * F | F
F -> ( E ) | id
```

   Lakukan shift-reduce parsing manual untuk input `id + id * id`. Tunjukkan stack, input, dan action pada setiap langkah.

3. Jelaskan perbedaan antara SLR(1), CLR(1), dan LALR(1). Mengapa LALR(1) menjadi pilihan populer untuk parser generator?

4. Buatlah file Bison sederhana untuk grammar ekspresi aritmatika dengan:

   - Operator +, -, *, / dengan precedence yang benar
   - Parentheses untuk grouping
   - Semantic actions yang mencetak ekspresi yang di-parse

5. Integrasikan Flex lexer dengan Bison parser untuk membuat calculator sederhana yang dapat mengevaluasi ekspresi aritmatika.

6. Implementasikan error recovery dalam Bison untuk menangani syntax error dengan baik. Test dengan input yang mengandung error.

7. Bandingkan performa dan kompleksitas implementasi antara recursive descent parser (top-down) dan parser yang di-generate oleh Bison (bottom-up) untuk grammar yang sama.

1.1.8 Latihan Bab 8: Semantic Actions dan AST Construction

1. Buatlah parser untuk ekspresi boolean yang mendukung:
   - Operasi AND (`&&`), OR (`||`), NOT (`!`)
   - Perbandingan (<, >, ==, !=)
   - Precedence yang benar
2. Modifikasi calculator untuk mendukung:
   - Variabel (assignment dan penggunaan)
   - Fungsi matematika dasar (sin, cos, sqrt)
   - Error handling yang lebih baik
3. Buatlah parser untuk konfigurasi file sederhana dengan format:

```
key1 = value1
key2 = value2
section {
    key3 = value3
}
```

4. Implementasikan semantic actions untuk membangun AST lengkap dari subset C, kemudian buat fungsi untuk:
   - Print AST dalam format tree
   - Evaluate AST (jika semua nilai diketahui)
   - Optimize AST (constant folding)
5. Pelajari dokumentasi Bison dan jelaskan perbedaan antara:
   - LALR(1) parsing (default Bison)
   - GLR parsing
   - Kapan masing-masing digunakan?

1.1.9 Latihan Bab 9: Abstract Syntax Tree (AST)

1. Buatlah AST untuk program berikut:

```
int x = 10;
int y = 20;
int z = x + y * 2;
```

Gambarkan struktur AST-nya.

2. Implementasikan AST node untuk:
   - For loop statement
   - Array access expression
   - String literal
   - Boolean literal
3. Buatlah visitor yang menghitung jumlah node dalam AST.
4. Buatlah visitor yang melakukan constant folding sederhana (misalnya: `3 + 5` $\rightarrow$ `8`).
5. Modifikasi PrettyPrinter untuk menghasilkan output yang lebih mirip dengan source code asli.
6. Implementasikan AST visualizer yang menghasilkan output dalam format Graphviz DOT untuk visualisasi grafis.
7. Jelaskan mengapa AST lebih disukai daripada parse tree untuk fase-fase selanjutnya dalam kompilator.
8. Buatlah contoh program yang menunjukkan perbedaan antara pre-order, post-order, dan in-order traversal pada AST ekspresi `(a + b) * (c - d)`.

1.1.10 Latihan Bab 10: Symbol Table dan Scope Management

1. Implementasikan symbol table lengkap dengan fitur-fitur berikut:
   - Insert, lookup, delete operations
   - Nested scopes support
   - Shadowing detection dengan warning
   - Visualisasi symbol table
2. Buatlah test cases untuk berbagai skenario:
   - Nested scopes dengan shadowing
   - Duplicate declaration dalam scope yang sama
   - Undeclared identifier usage
   - Function parameters dalam function scope
3. Analisis program berikut dan buat visualisasi symbol table-nya:

```
int a = 1;
int b = 2;

void func1(int x) {
    int a = 10;
    int c = 20;

    {
        int b = 30;
        int d = 40;
    }
}

void func2() {
    int x = 100;
    int y = 200;
}

```

4. Implementasikan fitur tambahan:
   - Tracking semua usages dari setiap identifier (bukan hanya deklarasi)
   - Support untuk forward declaration
   - Scope numbering untuk debugging
5. Bandingkan implementasi symbol table menggunakan:
   - Stack of hash tables (seperti yang dibahas)
   - Single hash table dengan per-name stacks

   Apa kelebihan dan kekurangan masing-masing pendekatan?

1.1.11 Latihan Bab 11: Type Checking dan Semantic Analysis

1. Jelaskan perbedaan antara static type checking dan dynamic type checking. Berikan contoh bahasa pemrograman untuk masing-masing.
2. Implementasikan type checker sederhana untuk ekspresi aritmatika dengan mendukung:
   - Operasi +, -, *, / untuk int dan float
   - Operasi perbandingan ==, !=, <, > yang menghasilkan boolean
   - Type promotion (int $\rightarrow$ float)
3. Buatlah fungsi untuk mengecek type compatibility dengan aturan:

- Exact match selalu kompatibel
- int dapat di-assign ke float (implicit conversion)
- float tidak dapat di-assign ke int tanpa explicit cast

4. Implementasikan semantic error detection untuk:
   - Undeclared variable
   - Type mismatch pada assignment
   - Wrong number of arguments pada function call
5. Buatlah test cases untuk type checker yang mencakup:
   - Valid expressions (harus pass type checking)
   - Invalid expressions (harus menghasilkan error yang sesuai)
   - Edge cases (null, empty, boundary values)
6. Jelaskan mengapa annotated AST diperlukan. Bagaimana informasi tipe pada AST digunakan pada fase code generation?
7. Implementasikan type checking untuk if statement dan while loop. Pastikan kondisi harus bertipe boolean.
8. Bandingkan nominal typing dan structural typing. Berikan contoh situasi di mana masing-masing pendekatan lebih menguntungkan.

1.1.12 Latihan Bab 12: Intermediate Code Generation

1. Buatlah TAC untuk ekspresi berikut:
   - `a = b + c * d - e`
   - `x = (a + b) * (c - d)`
   - `y = -a + b * -c`
2. Implementasikan generator TAC untuk:
   - Unary operations (negation, logical NOT)
   - Array access (`array[index]`)
   - Member access (`object.member`)
3. Buatlah TAC untuk program berikut:

```
int i, sum;
sum = 0;
for (i = 1; i <= 10; i = i + 1) {
    sum = sum + i;
}
```

4. Implementasikan common subexpression elimination untuk basic block. Test dengan contoh:

```
x = a + b * c
y = a + b * c
z = (a + b) * c
w = (a + b) * c
```

5. Jelaskan perbedaan antara three-address code dan quadruples. Kapan quadruples lebih menguntungkan?

6. Buatlah generator TAC untuk switch statement. Bagaimana cara menangani multiple cases?

7. Implementasikan optimasi constant folding pada TAC generator. Contoh: `x = 3 + 5` langsung menjadi `x = 8`

8. Bandingkan pendekatan top-down (generate sambil traverse) dan bottom-up (build IR structure dulu) untuk TAC generation. Apa keuntungan masing-masing?

1.1.13 Latihan Bab 13: Runtime Environment dan Memory Management

1. Jelaskan perbedaan antara static, stack, dan heap allocation. Berikan contoh penggunaan masing-masing.

2. Buatlah diagram activation records untuk program berikut saat `factorial(3)` sedang dieksekusi:

```
1  int factorial(int n) {
2      if (n <= 1) return 1;
3      return n * factorial(n - 1);
4  }
5
6  int main() {
7      int result = factorial(3);
8      return 0;
9  }
10
```

3. Implementasikan runtime stack simulator yang dapat menangani:
   - Function calls dengan parameters
   - Local variables
   - Return values
   - Nested function calls
4. Jelaskan calling sequence untuk fungsi dengan 3 parameters. Sertakan assembly code (simplified).
5. Bandingkan manual memory management dengan garbage collection. Apa keuntungan dan kekurangan masing-masing?
6. Implementasikan mark-and-sweep garbage collector sederhana yang dapat:
   - Mark reachable objects dari root set
   - Sweep dan free unreachable objects
   - Handle cyclic references
7. Analisis memory layout untuk program berikut dan identifikasi di mana setiap variabel dialokasikan:

```
    int global;
    static int static_var;

    void func() {
        int local;
        static int static_local;
        int* ptr = new int;
        // ...
    }

```

8. Jelaskan apa yang terjadi pada stack saat terjadi recursive call yang sangat dalam (misalnya 1000 level). Apa masalah yang mungkin terjadi?

1.1.14 Latihan Bab 14: Code Generation

1. Implementasikan code generator sederhana untuk operasi aritmatika dasar (+, -, *, /) yang menghasilkan RISC-V assembly.
2. Buatlah code generator untuk assignment statement. Test dengan berbagai skenario:
   - Assignment dari constant ke variabel
   - Assignment dari variabel ke variabel

- Assignment dari ekspresi ke variabel

3. Implementasikan local register allocation dengan algoritma sederhana. Test dengan basic block yang memiliki lebih banyak variabel aktif daripada jumlah register yang tersedia.

4. Buatlah code generator untuk function call sederhana dengan 2-3 parameter. Implementasikan calling convention RISC-V.

5. Test workflow lengkap: compile → assemble → link → run untuk program sederhana yang menghitung `a * b + c`.

6. Bandingkan code yang dihasilkan untuk ekspresi `a + b * c` dengan dan tanpa optimasi register allocation. Hitung jumlah instruksi dan memory access.

7. Implementasikan code generator untuk conditional statement (if-else) dengan branch instructions.

8. Buatlah code generator untuk loop (for/while) dengan label dan jump instructions.

1.1.15 Latihan Bab 15: Optimasi Kompilator

1. Identifikasi basic blocks dari kode three-address berikut:

```
t1 = a + b
t2 = c * d
if t1 > t2 goto L1
t3 = t1 - t2
goto L2
L1: t4 = t1 * t2
L2: t5 = t3 + t4
return t5
```

2. Lakukan constant folding pada kode berikut:

```
t1 = 5 + 3
t2 = t1 * 2
t3 = 10 / 2
x = t2 + t3
```

3. Lakukan constant propagation dan folding pada kode berikut:

```
x = 10
y = 20
t1 = x + 5
t2 = y * 2
```

```
z = t1 + t2
```

4. Identifikasi dan hapus dead code dari kode berikut:

```
x = 10
y = 20
t1 = x + y
t2 = 5 + 3      // Dead
t3 = t2 - 2     // Dead
z = t1 * 2
return z
```

5. Implementasikan optimizer sederhana yang melakukan:

   - Constant folding
   - Constant propagation (local)
   - Dead code elimination (basic)

   Uji dengan beberapa contoh program dan evaluasi hasilnya.

6. Jelaskan perbedaan antara:

   - Local optimization vs global optimization
   - Machine-independent vs machine-specific optimization
   - Constant folding vs constant propagation

7. Buatlah benchmark untuk mengevaluasi efektivitas optimasi. Bandingkan:

   - Ukuran kode sebelum dan sesudah optimasi
   - Waktu eksekusi sebelum dan sesudah optimasi
   - Waktu kompilasi sebelum dan sesudah optimasi

1.1.16 Latihan Bab 16: Evaluasi dan Project Final

1. **Prepare Presentation**:

   - Buat outline presentasi untuk project final Anda
   - Siapkan demo script dengan minimal 5 test cases
   - Latih presentasi untuk memastikan sesuai waktu

2. **Tool Evaluation**:

- Buat tabel perbandingan hand-written vs generator tools yang Anda gunakan
- Identifikasi trade-off yang Anda hadapi
- Justifikasi pilihan tools yang Anda buat

3. **Benchmarking**:
   - Siapkan test suite dengan berbagai ukuran program
   - Ukur metrik: compilation time, code size, execution time
   - Buat laporan benchmark dengan tabel dan analisis
4. **Documentation**:
   - Tulis README.md yang komprehensif
   - Buat design document
   - Dokumentasikan API jika ada
5. **Reflection**:
   - Tulis refleksi pembelajaran (minimal 500 kata)
   - Identifikasi 3 challenges utama yang dihadapi
   - Rangkum 5 lessons learned yang paling penting
   - Identifikasi 3 area untuk improvement
6. **Peer Review**:
   - Review project final dari teman sekelas
   - Berikan feedback konstruktif
   - Bandingkan pendekatan yang berbeda

1.2 Jawaban

1.2.1 Jawaban Latihan Bab 1: Pengenalan Kompilator dan Fase-Fase Kompilasi

1. **Perbedaan antara kompilator dan interpreter:**

   Kompilator:
   - Menerjemahkan seluruh program sekaligus sebelum dieksekusi
   - Hasil translasi disimpan dalam file terpisah (executable) yang kemudian dapat dieksekusi langsung oleh sistem operasi atau mesin virtual

- Sekali compile, berkali-kali run
- Contoh bahasa: C, C++, Rust, Go

Interpreter:

- Menerjemahkan dan mengeksekusi program baris demi baris secara langsung tanpa menghasilkan file terpisah untuk eksekusi
- Interpret setiap kali run
- Contoh bahasa: Python, JavaScript, Ruby

Hybrid Approach: Beberapa bahasa modern menggunakan pendekatan hybrid, seperti Java yang dikompilasi menjadi bytecode, kemudian diinterpretasi oleh JVM (Java Virtual Machine), atau menggunakan JIT (Just-In-Time compilation). Contoh: Java, C#, Python (dengan bytecode).

2. **Diagram alur kerja kompilator dari source code C sampai menjadi executable:**

   Alur kerja kompilator meliputi fase-fase berikut:

   (a) **Preprocessing**: Memproses directive khusus seperti `#include`, `#define`

   (b) **Lexical Analysis**: Memecah source code menjadi token-token (identifiers, keywords, operators, literals)

   (c) **Syntax Analysis**: Menganalisis struktur grammar dan membangun parse tree atau Abstract Syntax Tree (AST)

   (d) **Semantic Analysis**: Memeriksa aturan semantik bahasa, type checking, scope resolution

   (e) **Intermediate Code Generation**: Mengubah AST menjadi intermediate representation (misalnya three-address code)

   (f) **Code Optimization**: Mengoptimasi kode intermediate untuk meningkatkan efisiensi

   (g) **Code Generation**: Menghasilkan target code (assembly atau machine code)

   (h) **Assembling**: Mengubah assembly code menjadi object code (file .o atau .obj)

   (i) **Linking**: Menyatukan object files dan library files menjadi satu executable file

3. **Identifikasi fase kompilasi:**

   - **Menghapus whitespace dan comments**: **Lexical Analysis** - Fase ini membaca source code karakter demi karakter, mengelompokkan karakter menjadi token-token bermakna, dan mengeliminasi whitespace, comments, serta karakter yang tidak relevan.

- **Memeriksa apakah variabel dideklarasi sebelum digunakan**: **Semantic Analysis** - Fase ini melakukan scope resolution dan name resolution untuk memastikan setiap identifier merujuk ke deklarasi yang valid.
- **Mengoptimasi kode dengan menghilangkan kode yang tidak pernah dieksekusi**: **Code Optimization** - Fase ini melakukan dead code elimination untuk menghapus kode yang tidak memiliki efek pada perilaku program yang dapat diamati.
- **Mengkonversi ekspresi `a + b * c` menjadi three-address code**: **Intermediate Code Generation** - Fase ini mengubah AST menjadi intermediate representation seperti three-address code. Contoh hasilnya:

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

4. **Alasan kompilator modern menggunakan pendekatan multi-pass:**

   Keuntungan multi-pass:

   - **Pemisahan tanggung jawab**: Setiap pass memiliki tugas spesifik, membuat implementasi lebih modular dan mudah di-maintain
   - **Optimasi yang lebih baik**: Informasi yang dikumpulkan dalam pass pertama dapat digunakan untuk optimasi di pass berikutnya
   - **Portabilitas**: Front-end menghasilkan IR, back-end mengkonsumsi IR. Perubahan pada satu sisi tidak mempengaruhi sisi lain
   - **Retargeting**: Untuk menambahkan dukungan target baru, cukup menambahkan code generator untuk IR tersebut tanpa mengubah front-end
   - **Analisis yang lebih mendalam**: Multi-pass memungkinkan analisis lintas fase yang lebih komprehensif

   Keuntungan dibanding single-pass:

   - Single-pass memiliki keterbatasan dalam melakukan analisis yang memerlukan informasi dari seluruh program
   - Multi-pass memungkinkan optimasi yang lebih agresif dan akurat
   - Multi-pass memudahkan debugging karena setiap fase dapat diuji secara terpisah

5. **Contoh kompilator open source: GCC (GNU Compiler Collection)**

   - **Bahasa sumber dan target yang didukung**:
     - Bahasa sumber: C, C++, Fortran, Ada, Go, dan lainnya
     - Target: x86, ARM, RISC-V, MIPS, dan banyak arsitektur lainnya

- **Tools untuk lexical dan syntax analysis**:
  - GCC menggunakan hand-written recursive descent parser untuk C dan C++ (bukan generator tools)
  - Alasan: Better error messages, easier maintenance, better handling of complex grammar
- **Format intermediate representation**:
  - GCC menggunakan GIMPLE (Generic Intermediate Representation) sebagai IR utama
  - Juga menggunakan RTL (Register Transfer Language) untuk optimasi tingkat rendah
  - GIMPLE adalah three-address code yang lebih high-level

1.2.2 Jawaban Latihan Bab 2: Regular Expression dan Finite Automata

1. **Regular expression untuk:**
   - **Email address sederhana** (`user@domain.com`):

     ```
     [a-zA-Z0-9_]+@[a-zA-Z0-9]+\.[a-zA-Z]{2,}
     ```

     Penjelasan: `[a-zA-Z0-9_]+` untuk user, `@` literal, `[a-zA-Z0-9]+` untuk domain, `\.` untuk titik literal, `[a-zA-Z]{2,}` untuk ekstensi domain (minimal 2 karakter).
   - **Phone number** (`+62-812-3456-7890`):

     ```
     \+62-[0-9]{3,4}-[0-9]{4}-[0-9]{4}
     ```

     Penjelasan: `\+` untuk tanda plus literal, `62` untuk kode negara, − untuk separator, `[0-9]{3,4}` untuk operator (3-4 digit), `[0-9]{4}` untuk nomor (4 digit).
   - **C-style comment** (`/* ... */`):

     ```
     /\*([^*]|\*+[^*/])*\*+/
     ```

     Penjelasan: `/\*` untuk awal comment, pola untuk konten berupa karakter selain asterisk atau sekuens asterisk yang tidak diikuti slash, `*+/` untuk akhir comment.
2. **Konstruksi NFA menggunakan algoritma Thompson:**

   Algoritma Thompson menggunakan template untuk setiap operasi regex:
   - **Literal**: State awal dengan transisi ke state akhir menggunakan karakter tersebut

- **Concatenation (RS)**: Menghubungkan NFA untuk R dan S menggunakan ϵ-transition
- **Union (R|S)**: Menggunakan ϵ-transitions untuk branching dari state awal ke NFA R dan S, kemudian menggabungkan ke state akhir
- **Kleene Star (R*)**: Membuat loop dengan ϵ-transitions yang memungkinkan nol atau lebih pengulangan R

Untuk `a*b+`:

- Buat NFA untuk `a*` dengan loop menggunakan ϵ-transitions
- Concatenate dengan NFA untuk `b+` (satu atau lebih b)

Untuk `(a|b)*ab`:

- Buat NFA untuk `(a|b)*` dengan union dan Kleene star
- Concatenate dengan NFA untuk literal `a` dan `b`

Untuk `[0-9]+(\.[0-9]+)?`:

- Buat NFA untuk `[0-9]+` (satu atau lebih digit)
- Buat optional group `(\.[0-9]+)?` dengan ϵ-transition untuk kasus opsional

3. **Konversi NFA ke DFA menggunakan subset construction:**

   Algoritma subset construction:

   (a) Mulai dengan ϵ-closure dari start state NFA sebagai state awal DFA

   (b) Untuk setiap state DFA dan setiap input symbol:
   - Hitung ϵ-closure dari semua state yang dapat dicapai dengan symbol tersebut
   - Jika hasilnya non-empty dan belum ada, tambahkan sebagai state baru DFA

   (c) State DFA adalah accept state jika mengandung accept state dari NFA

   (d) Ulangi sampai tidak ada state baru

   State diagram DFA akan menunjukkan transisi deterministik (setiap state memiliki tepat satu transisi untuk setiap input symbol).

4. **Implementasi kelas NFA dan DFA dalam C++:**

   Struktur dasar:

   - **Kelas NFA**: Menyimpan states, transitions (termasuk ϵ-transitions), start state, dan accept states

- **Kelas DFA**: Menyimpan states, transition table (deterministik), start state, dan accept states
- **Fungsi konstruksi NFA dari regex**: Menggunakan algoritma Thompson secara rekursif
- **Fungsi konversi NFA ke DFA**: Mengimplementasikan subset construction algorithm
- **Fungsi simulasi DFA**: Mengikuti transisi berdasarkan input symbol, menerima jika berakhir di accept state

5. **Program recognizer untuk token-token:**

   Implementasi menggunakan DFA atau state machine:

   - **Identifier**: State machine yang mulai dengan huruf/underscore, kemudian menerima huruf/digit/underscore
   - **Integer**: State machine yang menerima satu atau lebih digit
   - **Float**: State machine yang menerima digit, titik desimal, kemudian digit
   - **Operator**: State machine yang mengenali operator single-character atau multi-character (==, !=)

6. **Mengapa DFA lebih efisien untuk simulasi dibanding NFA:**

   DFA:

   - Kompleksitas waktu: O(n) dimana n adalah panjang input string
   - Setiap state memiliki tepat satu transisi untuk setiap input symbol
   - Tidak memerlukan backtracking atau multiple states tracking
   - Implementasi sederhana: hanya perlu mengikuti transisi berdasarkan input symbol saat ini

   NFA:

   - Kompleksitas waktu: O(n * m) atau lebih buruk, dimana n adalah panjang input dan m adalah jumlah states
   - Memerlukan backtracking atau simulasi multiple states secara bersamaan
   - Perlu menangani ϵ-transitions yang tidak mengonsumsi input
   - Implementasi lebih kompleks: perlu mempertahankan set of states yang mungkin

   Kesimpulan: DFA lebih efisien untuk simulasi karena deterministik dan tidak memerlukan backtracking, meskipun konstruksi DFA dari regex lebih kompleks.

7. **Implementasi algoritma minimisasi DFA:**

   Algoritma minimisasi DFA (menggunakan partition refinement):

   (a) Partisi awal: Pisahkan accept states dan non-accept states

   (b) Untuk setiap partisi:

   - Cek apakah states dalam partisi memiliki transisi ke partisi yang berbeda untuk setiap input symbol
   - Jika ya, pisahkan menjadi partisi baru

   (c) Ulangi sampai tidak ada partisi yang dapat dipecah lagi

   (d) Gabungkan states dalam partisi yang sama menjadi satu state

   Hasilnya adalah DFA minimal yang ekuivalen dengan DFA asli tetapi dengan jumlah states yang lebih sedikit.

1.2.3 Jawaban Latihan Bab 3: Implementasi Lexer

1. **Implementasi lexer dasar:**

   Struktur dasar lexer:

   - **Token Types**: Enum untuk berbagai jenis token (IDENTIFIER, INTEGER_LITERAL, OPERATOR, PUNCTUATION)
   - **Token Structure**: Struktur data yang menyimpan type, lexeme, line, dan column
   - **State Machine**: Menggunakan finite state machine untuk mengenali token
   - **Methods**: `nextToken()`, `scanIdentifier()`, `scanNumber()`, `scanOperator()`, dll.

2. **Handling comments:**

   Single-line comments (`//`):

   - Ketika menemukan `//`, masuk ke state IN_COMMENT_LINE
   - Baca karakter sampai menemukan newline atau EOF
   - Skip semua karakter dalam comment (tidak menghasilkan token)

   Multi-line comments (`/* */`):

   - Ketika menemukan `/*`, masuk ke state IN_COMMENT_BLOCK
   - Baca karakter sampai menemukan `*/`
   - Error handling: Jika mencapai EOF sebelum menemukan `*/`, report error "unclosed block comment"

3. **String literals dengan escape sequences:**

   Implementasi:

   - Ketika menemukan `"`, masuk ke state IN_STRING
   - Baca karakter sampai menemukan closing `"`
   - Handle escape sequences:
     - `\n`: Newline
     - `\t`: Tab
     - `\"`: Quote literal
     - `\\`: Backslash literal
   - Error handling: Jika mencapai EOF atau newline sebelum closing quote, report error "unclosed string"

4. **Float literals:**

   Extend number scanning:

   - Setelah membaca digit, cek apakah karakter berikutnya adalah titik desimal
   - Jika ya, masuk ke state IN_FLOAT
   - Baca digit setelah titik desimal
   - Untuk scientific notation (`1.5e10`, `2.3E-5`):
     - Setelah membaca float, cek apakah ada `e` atau `E`
     - Baca optional + atau −
     - Baca eksponen (digit)

5. **Unit testing:**

   Test cases yang harus dicakup:

   - **Valid tokens**: Semua kategori token (identifiers, numbers, operators, punctuation)
   - **Edge cases**:
     - Unclosed strings
     - Unclosed block comments
     - Invalid characters
     - Empty input
     - Whitespace-only input
   - **Position tracking**: Verifikasi bahwa line dan column dihitung dengan benar untuk error reporting

6. **Error recovery:**

   Strategi error recovery:

   - **Skip invalid characters**: Ketika menemukan karakter yang tidak valid, skip karakter tersebut dan lanjutkan scanning
   - **Report multiple errors**: Kumpulkan semua error dalam satu pass, jangan berhenti pada error pertama
   - **Error messages yang informatif**: Sertakan line dan column number untuk setiap error

7. **Performance analysis:**

   Optimasi yang dapat dilakukan:

   - **Lookahead buffer**: Gunakan buffer untuk mengurangi jumlah system calls
   - **String interning**: Simpan lexeme sebagai interned strings untuk mengurangi memory usage
   - **Table-driven approach**: Gunakan lookup table untuk karakter classification
   - **Comparison dengan generator**: Benchmark dengan Flex-generated lexer untuk melihat perbedaan performa

1.2.4 Jawaban Latihan Bab 4: Lexer Generator

1. **Flex specification file:**

   Contoh specification untuk token-token yang diminta:

```
%{
#include "parser.tab.h"
%}

%%
"for"       { return FOR; }
"break"     { return BREAK; }
"continue"  { return CONTINUE; }

\"([^"\\]|\\.)*\"  {
    // String literal dengan escape sequences
    yylval.string = processString(yytext);
    return STRING_LITERAL;
}

'([^'\\]|\\.)'  {
    // Character literal
```

```
    yylval.char = processChar(yytext);
    return CHAR_LITERAL;
}

"++"        { return PLUS_PLUS; }
"--"        { return MINUS_MINUS; }
"+="        { return PLUS_EQ; }
"-="        { return MINUS_EQ; }
%%
```

2. **Implementasi dengan re2c:**

   Perbandingan kompleksitas:

   - **Flex**: Specification lebih declarative, menggunakan regular expressions
   - **re2c**: Specification lebih embedded dalam kode C/C++, menggunakan state machine yang lebih eksplisit
   - **Kompleksitas**: re2c biasanya menghasilkan kode yang lebih efisien tetapi specification lebih verbose

3. **Kapan menggunakan hand-written vs generator:**

   Hand-written lexer cocok untuk:

   - Error messages yang sangat informatif dan customizable
   - Kasus edge yang kompleks yang sulit diekspresikan dalam regex
   - Kontrol penuh atas implementasi
   - Project kecil yang tidak memerlukan generator overhead

   Generator cocok untuk:

   - Development yang cepat
   - Grammar yang well-defined dan standard
   - Project yang memerlukan proven algorithms
   - Ketika ingin fokus pada logic bukan implementasi detail

4. **Integrasi Flex dengan Bison:**

   Langkah-langkah:

   (a) Buat Flex specification file (`lexer.l`) yang meng-include header dari Bison

   (b) Buat Bison specification file (`parser.y`) yang mendefinisikan tokens

   (c) Compile: `flex lexer.l`, `bison -d parser.y`

(d) Link: `gcc lex.yy.c parser.tab.c -o program`

Interface antara Flex dan Bison:

- Bison menghasilkan `parser.tab.h` dengan definisi token
- Flex menggunakan token definitions dari header tersebut
- Flex mengisi `yylval` dengan semantic value sebelum return token
- Parser memanggil `yylex()` untuk mendapatkan token berikutnya

5. **Performance benchmark:**

   Metodologi:

   - Siapkan input file besar (10MB)
   - Ukur waktu eksekusi untuk setiap implementasi
   - Ukur memory usage
   - Bandingkan hasil

     Hasil yang diharapkan:

     – Flex-generated lexer biasanya lebih cepat karena menggunakan optimized DFA
     – re2c-generated lexer juga sangat efisien
     – Hand-written lexer mungkin lebih lambat tetapi lebih fleksibel

6. **Longest match dan rule priority dalam Flex:**

   Longest match:

   - Flex selalu mencocokkan pattern terpanjang yang mungkin
   - Contoh: Input `"if"` akan cocok dengan keyword `"if"` bukan identifier `"i"` diikuti `"f"`

   Rule priority:

   - Jika beberapa pattern cocok dengan panjang yang sama, Flex menggunakan rule yang muncul pertama dalam specification
   - Contoh: Jika keyword `"if"` didefinisikan sebelum identifier pattern, maka `"if"` akan dikenali sebagai keyword
   - Urutan rule penting: keyword harus didefinisikan sebelum identifier pattern

1.2.5 Jawaban Latihan Bab 5: Context-Free Grammar dan Parsing

1. **Grammar dalam BNF:**

   Ekspresi boolean:

   ```
   <boolean_expr> ::= <boolean_expr> AND <boolean_term>
                    | <boolean_expr> OR <boolean_term>
                    | <boolean_term>

   <boolean_term> ::= NOT <boolean_term>
                   | <boolean_factor>

   <boolean_factor> ::= TRUE | FALSE | ( <boolean_expr> )
   ```

   Assignment statement:

   ```
   <assignment> ::= <identifier> = <expression> ;
   ```

   For loop:

   ```
   <for_loop> ::= for ( <init> ; <condition> ;
             <update> ) <statement>

   <init> ::= <declaration> | <expression> | epsilon
   <condition> ::= <expression>
   <update> ::= <expression>
   ```

2. **Konversi ke EBNF:**

   Grammar asli:

   ```
   <list> ::= <item> | <list> , <item>
   <item> ::= <number> | <string>
   ```

   Dalam EBNF:

   ```
   list = item { "," item }
   item = number | string
   ```

   Penjelasan: EBNF menggunakan `{ }` untuk repetition (nol atau lebih), sehingga `<list> , <item>` menjadi `{ "," item }`.

3. **Leftmost dan rightmost derivation untuk `(2 + 3) * 4`:**

   Grammar:

```
E → E + T | E - T | T
T → T * F | T / F | F
F → ( E ) | number
```

Leftmost derivation:

$$
\begin{aligned}
E &\Rightarrow T \\
&\Rightarrow T * F \\
&\Rightarrow F * F \\
&\Rightarrow (E) * F \\
&\Rightarrow (E + T) * F \\
&\Rightarrow (T + T) * F \\
&\Rightarrow (F + T) * F \\
&\Rightarrow (2 + T) * F \\
&\Rightarrow (2 + F) * F \\
&\Rightarrow (2 + 3) * F \\
&\Rightarrow (2 + 3) * 4
\end{aligned}
$$

Rightmost derivation:

$$
\begin{aligned}
E &\Rightarrow T \\
&\Rightarrow T * F \\
&\Rightarrow T * 4 \\
&\Rightarrow F * 4 \\
&\Rightarrow (E) * 4 \\
&\Rightarrow (E + T) * 4 \\
&\Rightarrow (E + F) * 4 \\
&\Rightarrow (E + 3) * 4 \\
&\Rightarrow (T + 3) * 4 \\
&\Rightarrow (F + 3) * 4 \\
&\Rightarrow (2 + 3) * 4
\end{aligned}
$$

4. **Parse tree untuk ekspresi:**

   Parse tree menunjukkan struktur hierarkis ekspresi berdasarkan grammar. Untuk `1 + 2 * 3`, parse tree menunjukkan bahwa `*` memiliki precedence lebih tinggi daripada `+`, sehingga `2 * 3` dievaluasi terlebih dahulu.

5. **Identifikasi ambiguity:**

   Grammar:

   ```
   S → a S a | b S b | a | b | epsilon
   ```

   Grammar ini menghasilkan bahasa palindrome dengan panjang ganjil. Contoh string yang dapat di-parse dengan lebih dari satu cara: `aba` dapat dihasilkan dengan:

   - `S → a S a → a b a` (menggunakan `S → b`)
   - Atau dengan cara lain tergantung pada urutan aplikasi rules

6. **Eliminasi left recursion:**

   Grammar asli:

   ```
   A → A + B | A - B | B
   B → B * C | B / C | C
   C → ( A ) | number
   ```

   Setelah eliminasi left recursion:

   ```
   A → B A'
   A' → + B A' | - B A' | epsilon

   B → C B'
   B' → * C B' | / C B' | epsilon

   C → ( A ) | number
   ```

7. **Left factoring:**

   Grammar asli:

   ```
   S → if E then S else S
     | if E then S
     | while E do S
     | id = E
   ```

   Setelah left factoring:

   ```
   S → if E then S S'
     | while E do S
     | id = E

   S' → else S | epsilon
   ```

8. **Grammar untuk bahasa sederhana:**

   Dalam EBNF:

```
program = { declaration | statement }

declaration = type identifier ";"

statement = assignment
          | if_statement
          | while_statement
          | block

assignment = identifier "=" expression ";"

if_statement = "if" "(" expression ")" statement
              [ "else" statement ]

while_statement = "while" "(" expression ")" statement

block = "{" { statement } "}"

expression = term { ("+" | "-") term }
term = factor { ("*" | "/") factor }
factor = identifier | number | "(" expression ")"

type = "int" | "float"
identifier = letter { letter | digit }
number = digit { digit }
```

   Contoh program valid:

```
int x;
x = 5;
if (x > 0) {
    int y;
    y = 10;
    while (y > 0) {
        y = y - 1;
    }
}
```

1.2.6 Jawaban Latihan Bab 6: Top-Down Parsing dan Recursive Descent

1. **Implementasi recursive descent parser:**

Langkah 1: Eliminasi left recursion

Grammar asli memiliki left recursion pada E dan T. Setelah eliminasi:

```
S → if E then S else S | id := E | while E do S
E → T E'
E' → + T E' | epsilon
T → F T'
T' → * F T' | epsilon
F → ( E ) | id | num
```

Langkah 2: Implementasi dalam C++

Setiap nonterminal menjadi fungsi yang sesuai:

- `parseS()` untuk S
- `parseE()`, `parseEPrime()` untuk E dan E'
- `parseT()`, `parseTPrime()` untuk T dan T'
- `parseF()` untuk F

2. **Modifikasi untuk operator unary minus:**

   Tambahkan production untuk unary minus di level factor:

   ```
   F → ( E ) | id | num | - F
   ```

   Implementasi dalam `parseF()`:

   - Cek apakah token berikutnya adalah −
   - Jika ya, consume token dan parse F secara rekursif
   - Return negated value

3. **Error recovery:**

   Strategi:

   - **Synchronization points**: Token-token yang dapat digunakan untuk recovery (misalnya `;`, `}`, keywords)
   - **Panic mode**: Skip token sampai menemukan synchronization point
   - **Error reporting**: Kumpulkan semua error, jangan berhenti pada error pertama
   - **Insertion/deletion recovery**: Coba insert token yang diharapkan atau skip token yang tidak diharapkan

4. **Parser untuk ekspresi boolean:**

   Grammar dengan precedence:

```
E → T E'          (OR level, lowest precedence)
E' → OR T E' | epsilon

T → F T'          (AND level)
T' → AND F T' | epsilon

F → NOT F | ( E ) | comparison
```

Precedence: NOT > AND > OR

5. **Perbandingan recursive descent vs table-driven LL:**

   Recursive Descent:

   - **Keuntungan**: Kode lebih readable, error recovery lebih baik, mudah di-debug
   - **Kerugian**: Hanya cocok untuk LL(1) grammar, lebih banyak kode manual

   Table-driven LL:

   - **Keuntungan**: Grammar dapat diubah tanpa mengubah kode parser, lebih compact
   - **Kerugian**: Error messages kurang informatif, lebih sulit di-debug, memerlukan table construction

6. **Parser untuk right-associative operator:**

   Untuk operator assignment = yang right-associative:

   ```
   assignment → identifier = assignment | expression
   ```

   Implementasi: Parse identifier dan =, kemudian parse assignment secara rekursif (right-associative).

1.2.7 Jawaban Latihan Bab 7: Bottom-Up Parsing dan Parser Generator

1. **Perbedaan top-down dan bottom-up parsing:**

   Top-down parsing:

   - Mulai dari start symbol, turun ke terminal
   - Menggunakan leftmost derivation
   - Contoh: Recursive descent, LL parsers

- Cocok untuk: Grammar yang tidak memiliki left recursion, grammar yang lebih sederhana

Bottom-up parsing:

- Mulai dari terminal, naik ke start symbol
- Menggunakan rightmost derivation dalam reverse
- Contoh: LR, LALR, SLR parsers
- Cocok untuk: Grammar yang lebih kompleks, grammar dengan left recursion

2. **Shift-reduce parsing manual untuk `id + id * id`:**

   Grammar:

   ```
   E → E + T | T
   T → T * F | F
   F → ( E ) | id
   ```

Stack	Input	Action	Production
$	id + id * id $	Shift	
$ id	+ id * id $	Reduce	F → id
$ F	+ id * id $	Reduce	T → F
$ T	+ id * id $	Reduce	E → T
$ E	+ id * id $	Shift	
$ E +	id * id $	Shift	
$ E + id	* id $	Reduce	F → id
$ E + F	* id $	Reduce	T → F
$ E + T	* id $	Shift	
$ E + T *	id $	Shift	
$ E + T * id	$	Reduce	F → id
$ E + T * F	$	Reduce	T → T * F
$ E + T	$	Reduce	E → E + T
$ E	$	Accept	

3. **Perbedaan SLR(1), CLR(1), dan LALR(1):**

   SLR(1) - Simple LR:

   - Menggunakan LR(0) item sets
   - Reduce hanya jika lookahead dalam Follow set
   - Tabel lebih kecil, tetapi kurang powerful

   CLR(1) - Canonical LR:

- Menggunakan LR(1) items dengan lookahead spesifik
- Paling powerful, dapat menangani lebih banyak grammar
- Tabel sangat besar

LALR(1) - Look-Ahead LR:

- Merge states dari CLR(1) yang memiliki LR(0) core sama
- Kompromi praktis: tabel lebih kecil dari CLR(1), lebih powerful dari SLR(1)
- Digunakan oleh Bison dan Yacc

Mengapa LALR(1) populer:

- Balance yang baik antara power dan table size
- Dapat menangani sebagian besar grammar praktis
- Tabel cukup kecil untuk efisiensi

4. **File Bison untuk ekspresi aritmatika:**

```
%{
#include <stdio.h>
%}

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

%%
expr: expr '+' expr { printf("+ "); }
   | expr '-' expr { printf("- "); }
   | expr '*' expr { printf("* "); }
   | expr '/' expr { printf("/ "); }
   | '(' expr ')' { /* no action */ }
   | NUMBER { printf("%s ", $1); }
%%
```

5. **Integrasi Flex dan Bison untuk calculator:**

   Langkah-langkah:

   (a) Buat Flex lexer yang menghasilkan NUMBER token

   (b) Buat Bison parser dengan semantic actions untuk evaluasi

   (c) Link kedua file dan compile

   Semantic actions mengevaluasi ekspresi dan mencetak hasil.

6. **Error recovery dalam Bison:**

   Menggunakan token `error`:

   ```
   expr: expr '+' expr
      | error { yyerror("Syntax error in expression"); }
      | error ';' { /* recover by skipping to semicolon */ }
   ```

   Strategi: Skip token sampai menemukan synchronization point (misalnya `;`).

7. **Perbandingan performa:**

   Recursive descent:

   - Lebih cepat untuk grammar sederhana
   - Overhead function calls

   Bison-generated:

   - Table lookup overhead
   - Lebih efisien untuk grammar kompleks
   - Dapat dioptimasi dengan table compression

1.2.8 Jawaban Latihan Bab 8: Semantic Actions dan AST Construction

1. **Parser untuk ekspresi boolean:**

   Grammar dengan semantic actions untuk membangun AST:

   ```
   %union {
       ASTNode* node;
   }

   %type <node> expr

   %%
   expr: expr AND expr { $$ = createBinaryNode(AND, $1, $3); }
      | expr OR expr  { $$ = createBinaryNode(OR, $1, $3); }
      | NOT expr      { $$ = createUnaryNode(NOT, $2); }
      | comparison    { $$ = $1; }

   comparison: expr '<' expr
       { $$ = createBinaryNode(LT, $1, $3); }
            | expr '>' expr
       { $$ = createBinaryNode(GT, $1, $3); }
            | expr EQ expr
   ```

```
    { $$ = createBinaryNode(EQ, $1, $3); }
         | expr NE expr
    { $$ = createBinaryNode(NE, $1, $3); }
%%
```

2. **Modifikasi calculator:**

   Fitur tambahan:

   - **Variabel**: Symbol table untuk menyimpan nilai variabel
   - **Fungsi matematika**: Built-in functions (sin, cos, sqrt)
   - **Error handling**: Type checking, undefined variable detection

3. **Parser untuk konfigurasi file:**

   Grammar:

```
config: { /* init */ } config_items

config_items: config_items config_item | config_item

config_item: key '=' value '\n'
           | section

section: IDENTIFIER '{' config_items '}'

key: IDENTIFIER
value: STRING | NUMBER
```

   Semantic actions membangun struktur data konfigurasi.

4. **Semantic actions untuk AST lengkap:**

   Fungsi-fungsi yang diperlukan:

   - **Print AST**: Traverse AST dan print dalam format tree (pre-order atau in-order)
   - **Evaluate AST**: Jika semua nilai diketahui, evaluasi ekspresi
   - **Optimize AST**: Constant folding - ganti ekspresi konstanta dengan hasilnya

5. **Perbedaan LALR(1), GLR parsing:**

   LALR(1) - Default Bison:

   - Deterministic parsing
   - Satu parse tree untuk setiap input

- Tidak dapat menangani ambiguous grammar
- Lebih cepat dan efisien

GLR - Generalized LR:

- Dapat menangani ambiguous grammar
- Menjaga multiple parse trees aktif secara bersamaan
- Merge stack prefixes yang mungkin
- Lebih lambat tetapi lebih powerful

Kapan digunakan:

- LALR(1): Grammar yang unambiguous, sebagian besar kasus praktis
- GLR: Grammar yang ambiguous, natural language processing, extensible syntax

1.2.9 Jawaban Latihan Bab 9: Abstract Syntax Tree (AST)

1. **AST untuk program:**

   Program:

   ```
   int x = 10;
   int y = 20;
   int z = x + y * 2;
   ```

   Struktur AST:

   - Root: Program node dengan tiga children (declarations)
   - Setiap declaration: VarDecl node dengan type, identifier, dan optional initializer
   - Ekspresi `x + y * 2`: BinaryExpr node dengan operator `+`, left adalah Identifier `x`, right adalah BinaryExpr dengan operator `*`

2. **Implementasi AST node:**

   For loop statement:

   - Node: ForStmt dengan fields: init (optional), condition, update (optional), body

   Array access expression:

   - Node: ArrayAccess dengan fields: array (expression), index (expression)

   String literal:

   - Node: StringLiteral dengan field: value (string)

Boolean literal:

- Node: BoolLiteral dengan field: value (bool)

3. **Visitor untuk menghitung jumlah node:**

   Implementasi menggunakan visitor pattern:

   - Visitor memiliki counter
   - Setiap node memanggil `accept(visitor)`
   - Visitor increment counter dan traverse children
   - Hasil: total jumlah node dalam AST

4. **Visitor untuk constant folding:**

   Algoritma:

   - Traverse AST secara post-order
   - Jika menemukan BinaryExpr dengan kedua operan adalah konstanta:
     - Evaluasi ekspresi (misalnya 3 + 5 = 8)
     - Ganti BinaryExpr dengan Literal node yang berisi hasil
   - Ulangi sampai tidak ada perubahan

5. **Modifikasi PrettyPrinter:**

   Improvements:

   - Preserve whitespace dan formatting sebisa mungkin
   - Handle operator precedence dengan parentheses yang tepat
   - Format indentation yang konsisten
   - Preserve comments jika ada

6. **AST visualizer dengan Graphviz DOT:**

   Implementasi:

   - Traverse AST dan generate DOT format
   - Setiap node menjadi node dalam graph
   - Edges menunjukkan parent-child relationships
   - Output dapat di-render dengan `dot` command

7. **Mengapa AST lebih disukai daripada parse tree:**

   Alasan:

- **Lebih kompak**: Menghilangkan detail sintaksis yang tidak relevan
- **Fokus semantik**: Hanya menyertakan informasi yang diperlukan untuk fase selanjutnya
- **Mudah di-manipulasi**: Struktur yang lebih sederhana memudahkan transformasi
- **Efisien**: Ukuran lebih kecil, traversal lebih cepat

8. **Traversal orders untuk `(a + b) * (c - d)`:**

   Pre-order: `* + a b - c d` (operator sebelum operan)

   In-order: `(a + b) * (c - d)` (mirip dengan ekspresi asli)

   Post-order: `a b + c d - *` (operator setelah operan, cocok untuk evaluasi)

1.2.10 Jawaban Latihan Bab 10: Symbol Table dan Scope Management

1. **Implementasi symbol table lengkap:**

   Fitur-fitur:

   - **Insert**: Menambahkan symbol ke scope saat ini
   - **Lookup**: Mencari symbol mulai dari scope saat ini ke parent scopes
   - **Delete**: Menghapus symbol dari scope (biasanya saat exit scope)
   - **Nested scopes**: Stack of hash tables, setiap scope memiliki parent pointer
   - **Shadowing detection**: Warning ketika identifier shadow outer declaration
   - **Visualisasi**: Print symbol table dengan indentasi untuk menunjukkan hierarchy

2. **Test cases:**

   Nested scopes dengan shadowing:

```
int x = 1;
{
    int x = 2;  // Shadows outer x
    // x refers to inner x (2)
}
// x refers to outer x (1)
```

   Duplicate declaration:

```
int x = 1;
int x = 2;  // Error: duplicate declaration
```

Undeclared identifier:

```
x = 10;  // Error: x not declared
```

Function parameters:

```
void func(int x) {
    // x is in function scope
    int y = x + 1;  // OK: x is parameter
}
```

3. **Visualisasi symbol table untuk program:**

   Program:

```
int a = 1;        // Global scope
int b = 2;

void func1(int x) {  // Function scope
    int a = 10;      // Shadows global a
    int c = 20;

    {                // Block scope
        int b = 30;  // Shadows func1 b
        int d = 40;
    }
}

void func2() {       // Function scope
    int x = 100;
    int y = 200;
}
```

   Symbol table structure:

```
Level 0 (Global):
  a -> int (line 1)
  b -> int (line 2)
  func1 -> function
  func2 -> function

Level 1 (func1):
  x -> int (parameter)
  a -> int (line 4) [shadows global a]
  c -> int (line 5)

Level 2 (block in func1):
```

```
    b -> int (line 8) [shadows func1 b]
    d -> int (line 9)

  Level 1 (func2):
    x -> int (line 14)
    y -> int (line 15)
```

4. **Fitur tambahan:**

   Tracking usages:

   - Simpan list semua lokasi dimana identifier digunakan
   - Berguna untuk dead code elimination dan optimasi

   Forward declaration:

   - Simpan declaration tanpa body
   - Resolve saat body ditemukan

   Scope numbering:

   - Assign unique number untuk setiap scope
   - Memudahkan debugging dan visualisasi

5. **Perbandingan implementasi:**

   Stack of hash tables:

   - **Kelebihan**: Simple, efficient lookup dalam scope saat ini
   - **Kekurangan**: Lookup di parent scope memerlukan traversal

   Single hash table dengan per-name stacks:

   - **Kelebihan**: Fast lookup untuk name tertentu
   - **Kekurangan**: Lebih kompleks, perlu manage multiple stacks

1.2.11 Jawaban Latihan Bab 11: Type Checking dan Semantic Analysis

1. **Perbedaan static dan dynamic type checking:**

   Static type checking:

   - Type checking dilakukan pada waktu kompilasi
   - Error ditemukan sebelum program dijalankan

- Contoh bahasa: C, C++, Java, Rust
- Keuntungan: Early error detection, better performance

Dynamic type checking:

- Type checking dilakukan pada waktu runtime
- Error ditemukan saat program dijalankan
- Contoh bahasa: Python, JavaScript, Ruby
- Keuntungan: Lebih fleksibel, rapid prototyping

2. **Type checker untuk ekspresi aritmatika:**

   Implementasi:

   - **Operasi +, -, *, /**: Cek bahwa kedua operan adalah int atau float
   - **Type promotion**: Jika salah satu operan float, hasilnya float
   - **Operasi perbandingan**: Cek kompatibilitas tipe, hasilnya boolean

3. **Fungsi type compatibility:**

   Aturan:

   - Exact match selalu kompatibel
   - int dapat di-assign ke float (implicit conversion)
   - float tidak dapat di-assign ke int tanpa explicit cast

4. **Semantic error detection:**

   Undeclared variable:

   - Saat menemukan identifier, cek di symbol table
   - Jika tidak ditemukan, report error

   Type mismatch pada assignment:

   - Cek tipe left-hand side dan right-hand side
   - Gunakan fungsi compatibility check

   Wrong number of arguments:

   - Cek jumlah argument dengan jumlah parameter fungsi
   - Cek tipe setiap argument dengan parameter yang sesuai

5. **Test cases:**

   Valid expressions:

- `int x = 42;`
- `float y = 3.14;`
- `int z = x + 10;`
- `bool b = x > 0;`

Invalid expressions:

- `int x = "string";` // Type mismatch
- `x = 10;` // Undeclared variable
- `add(5);` // Wrong number of arguments

6. **Mengapa annotated AST diperlukan:**

   Alasan:

   - **Type information**: Setiap node menyimpan tipe hasil ekspresi
   - **Code generation**: Informasi tipe digunakan untuk memilih instruksi yang tepat
   - **Optimization**: Type information memungkinkan optimasi yang lebih baik

7. **Type checking untuk if dan while:**

   Implementasi:

   - Cek bahwa kondisi adalah boolean
   - Jika bukan boolean, report type error

8. **Perbandingan nominal dan structural typing:**

   Nominal typing:

   - Tipe kompatibel jika memiliki nama yang sama
   - Contoh: Java, C++
   - Keuntungan: Explicit, clear contracts

   Structural typing:

   - Tipe kompatibel jika memiliki struktur yang sama
   - Contoh: TypeScript, Go interfaces
   - Keuntungan: Lebih fleksibel, duck typing

1.2.12 Jawaban Latihan Bab 12: Intermediate Code Generation

1. **TAC untuk ekspresi:**

   `a = b + c * d - e:`

   ```
   t1 = c * d
   t2 = b + t1
   t3 = t2 - e
   a = t3
   ```

   `x = (a + b) * (c - d):`

   ```
   t1 = a + b
   t2 = c - d
   t3 = t1 * t2
   x = t3
   ```

   `y = -a + b * -c:`

   ```
   t1 = -a
   t2 = -c
   t3 = b * t2
   t4 = t1 + t3
   y = t4
   ```

2. **Generator TAC untuk operasi khusus:**

   Unary operations:

   - Negation: `t1 = -operand`
   - Logical NOT: `t1 = !operand`

   Array access:

   - `t1 = index * element_size`
   - `t2 = base + t1`
   - `t3 = mem[t2]`

   Member access:

   - `t1 = object + offset`
   - `t2 = mem[t1]`

3. **TAC untuk for loop:**

   Program:

```
int i, sum;
sum = 0;
for (i = 1; i <= 10; i = i + 1) {
    sum = sum + i;
}
```

TAC:

```
sum = 0
i = 1
L1: t1 = i <= 10
    jmpf t1, L2
    t2 = sum + i
    sum = t2
    t3 = i + 1
    i = t3
    jmp L1
L2:
```

4. **Common subexpression elimination:**

   Kode asli:

```
x = a + b * c
y = a + b * c
z = (a + b) * c
w = (a + b) * c
```

   Setelah CSE:

```
t1 = b * c
t2 = a + b
x = a + t1
y = x        // atau y = a + t1
z = t2 * c
w = z        // atau w = t2 * c
```

5. **Perbedaan TAC dan quadruples:**

   Three-address code:

   - Format text: `t1 = a + b`
   - Mudah dibaca manusia
   - Sequential representation

Quadruples:

- Format struktural: `(op: +, arg1: a, arg2: b, result: t1)`
- Mudah di-reorder untuk optimasi
- Lebih mudah untuk manipulasi programmatic

Quadruples lebih menguntungkan untuk:

- Optimasi yang memerlukan reordering
- Common subexpression elimination
- Code generation yang memerlukan manipulasi struktural

6. **Generator TAC untuk switch statement:**

   Strategi:

   - Generate comparison untuk setiap case
   - Gunakan jump table jika case values consecutive
   - Atau gunakan chain of if-else dengan labels

7. **Constant folding pada TAC generator:**

   Implementasi:

   - Saat generate TAC, cek apakah kedua operan adalah konstanta
   - Jika ya, evaluasi langsung dan generate assignment konstanta hasil
   - Contoh: `x = 3 + 5` langsung menjadi `x = 8`

8. **Perbandingan pendekatan TAC generation:**

   Top-down (generate sambil traverse):

   - Generate TAC langsung saat traverse AST
   - Keuntungan: Simple, langsung menghasilkan output
   - Kerugian: Sulit untuk optimasi lintas ekspresi

   Bottom-up (build IR structure dulu):

   - Build struktur IR lengkap dulu, kemudian generate TAC
   - Keuntungan: Memungkinkan optimasi sebelum code generation
   - Kerugian: Lebih kompleks, memerlukan lebih banyak memory

1.2.13 Jawaban Latihan Bab 13: Runtime Environment dan Memory Management

1. **Perbedaan static, stack, dan heap allocation:**

   Static allocation:

   - Alokasi pada waktu kompilasi
   - Lifetime: Selama program berjalan
   - Contoh: Global variables, static variables
   - Lokasi: Data segment

   Stack allocation:

   - Alokasi pada waktu runtime (function call)
   - Lifetime: Selama fungsi aktif
   - Contoh: Local variables, function parameters
   - Lokasi: Stack segment
   - Management: Automatic (dikelola oleh compiler)

   Heap allocation:

   - Alokasi dinamis pada waktu runtime
   - Lifetime: Sampai secara eksplisit di-deallocate
   - Contoh: Dynamic arrays, objects dengan lifetime fleksibel
   - Lokasi: Heap segment
   - Management: Manual (programmer) atau garbage collection

2. **Diagram activation records untuk `factorial(3)`:**

   Stack saat `factorial(3)` sedang dieksekusi (recursive call):

```
High Address
+---------------------+
| main() AR           |
|   result            |
+---------------------+
| factorial(3) AR     | <- Current (top)
|   n = 3             |
|   return address    |
|   control link      | -> main AR
+---------------------+
| factorial(2) AR     |
```

```
|   n = 2            |
|   return address   |
|   control link     | -> factorial(3) AR
+--------------------+
| factorial(1) AR    |
|   n = 1            |
|   return address   |
|   control link     | -> factorial(2) AR
Low Address
```

3. **Runtime stack simulator:**

   Implementasi:

   - **Stack structure**: Linked list atau array of activation records
   - **Push frame**: Saat function call, buat AR baru dan push ke stack
   - **Pop frame**: Saat function return, pop AR dari stack
   - **Parameter passing**: Simpan parameters dalam AR
   - **Local variables**: Simpan dalam AR
   - **Return value**: Simpan dalam AR sebelum pop

4. **Calling sequence untuk fungsi dengan 3 parameters:**

   Assembly code (simplified):

```
; Caller sequence
push param3      ; Push parameter 3
push param2      ; Push parameter 2
push param1      ; Push parameter 1
call function    ; Call function
add esp, 12      ; Clean up parameters (3 * 4 bytes)

; Callee sequence (function prologue)
push ebp         ; Save frame pointer
mov ebp, esp     ; Set new frame pointer
sub esp, N       ; Allocate space for locals

; Function body
...

; Return sequence (function epilogue)
mov eax, return_value  ; Set return value
mov esp, ebp     ; Restore stack pointer
pop ebp          ; Restore frame pointer
ret              ; Return
```

5. **Perbandingan manual memory management vs garbage collection:**

   Manual memory management:

   - **Keuntungan**: Kontrol penuh, predictable performance, no GC overhead
   - **Kekurangan**: Mudah terjadi memory leak, use-after-free bugs, lebih kompleks
   - Contoh: C, C++

   Garbage collection:

   - **Keuntungan**: Tidak ada memory leak, lebih aman, programmer tidak perlu manage memory
   - **Kekurangan**: GC pause, overhead, unpredictable timing
   - Contoh: Java, Python, Go

6. **Implementasi mark-and-sweep garbage collector:**

   Algoritma:

   (a) **Mark phase**: Traverse dari root set (stack, global variables, registers), mark semua reachable objects

   (b) **Sweep phase**: Scan semua objects, free yang tidak ter-mark

   (c) **Handle cyclic references**: Mark phase secara otomatis menangani cycles karena menggunakan graph traversal

7. **Memory layout untuk program:**

```
int global;           // Static/Global segment
static int static_var; // Static/Global segment

void func() {
    int local;              // Stack segment
    static int static_local; // Static (meskipun dalam fungsi)
    int* ptr = new int;     // Heap segment (yang ditunjuk ptr)
    // ptr sendiri ada di stack
}
```

8. **Recursive call yang sangat dalam:**

   Masalah yang mungkin terjadi:

   - **Stack overflow**: Stack segment terbatas, recursive call yang sangat dalam dapat melebihi batas
   - **Memory exhaustion**: Setiap call memerlukan activation record

- **Performance degradation**: Overhead function calls

Solusi:

- Tail call optimization (jika compiler mendukung)
- Iterative implementation
- Increase stack size (jika memungkinkan)

1.2.14 Jawaban Latihan Bab 14: Code Generation

1. **Code generator untuk operasi aritmatika RISC-V:**

   Implementasi:

   - **Addition**: `ADD rd, rs1, rs2`
   - **Subtraction**: `SUB rd, rs1, rs2`
   - **Multiplication**: `MUL rd, rs1, rs2`
   - **Division**: `DIV rd, rs1, rs2`

2. **Code generator untuk assignment:**

   Constant ke variabel:

   ```
   LI t1, 42        ; Load immediate
   SW t1, x         ; Store to variable
   ```

   Variabel ke variabel:

   ```
   LW t1, x         ; Load from x
   SW t1, y         ; Store to y
   ```

   Ekspresi ke variabel:

   ```
   LW t1, a
   LW t2, b
   ADD t3, t1, t2
   SW t3, result
   ```

3. **Local register allocation:**

   Algoritma sederhana:

   - Scan basic block, assign register untuk setiap variabel aktif
   - Jika register penuh, spill variabel yang paling lama tidak digunakan ke memory

- Load dari memory saat diperlukan kembali

4. **Code generator untuk function call:**

   RISC-V calling convention:

   - Parameters: a0-a7 (first 8 parameters), kemudian stack
   - Return value: a0
   - Caller-saved: t0-t6, a0-a7
   - Callee-saved: s0-s11

   Code generation:

```
; Caller
ADDI sp, sp, -16    ; Allocate stack space
SW ra, 12(sp)       ; Save return address
MV a0, param1       ; Pass parameter 1
MV a1, param2       ; Pass parameter 2
CALL function       ; Call function
LW ra, 12(sp)       ; Restore return address
ADDI sp, sp, 16     ; Deallocate
MV result, a0       ; Get return value

; Callee
function:
    ADDI sp, sp, -16  ; Allocate frame
    SW ra, 12(sp)     ; Save return address
    SW s0, 8(sp)      ; Save callee-saved
    ADDI s0, sp, 16   ; Set frame pointer
    ; Function body
    MV a0, return_val ; Set return value
    LW s0, 8(sp)      ; Restore
    LW ra, 12(sp)     ; Restore
    ADDI sp, sp, 16   ; Deallocate
    RET               ; Return
```

5. **Test workflow lengkap:**

   Program: `a * b + c`

   Compile:

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

   Assemble:

```
LW t0, a
LW t1, b
MUL t2, t0, t1
LW t3, c
ADD t4, t2, t3
```

Link: Menghasilkan executable

Run: Eksekusi program

6. **Perbandingan dengan dan tanpa optimasi register allocation:**
   Tanpa optimasi:

```
LW t0, a          ; Load a
LW t1, b          ; Load b
MUL t2, t0, t1    ; t2 = a * b
SW t2, temp       ; Spill to memory
LW t3, c          ; Load c
LW t4, temp       ; Reload from memory
ADD t5, t4, t3    ; t5 = t2 + c
```

   Instruksi: 7, Memory access: 5
   Dengan optimasi:

```
LW t0, a
LW t1, b
MUL t2, t0, t1    ; Keep in register
LW t3, c
ADD t4, t2, t3    ; Use register directly
```

   Instruksi: 5, Memory access: 3

   Improvement: Mengurangi instruksi dan memory access.

7. **Code generator untuk if-else:**

```
; Condition evaluation
LW t0, x
LI t1, 0
BGT t0, t1, then_label  ; Branch if x > 0

; Else branch
else_label:
    LI t2, 0
    SW t2, y
    J end_label
```

```
; Then branch
then_label:
    LI t2, 1
    SW t2, y

end_label:
```

8. **Code generator untuk loop:**

```
loop_start:
    ; Condition check
    LW t0, i
    LI t1, 10
    BGE t0, t1, loop_end  ; Branch if i >= 10

    ; Loop body
    LW t2, sum
    ADD t2, t2, t0
    SW t2, sum

    ; Increment
    ADDI t0, t0, 1
    SW t0, i

    J loop_start          ; Jump back

loop_end:
```

1.2.15 Jawaban Latihan Bab 15: Optimasi Kompilator

1. **Identifikasi basic blocks:**

   Kode:

```
t1 = a + b
t2 = c * d
if t1 > t2 goto L1
t3 = t1 - t2
goto L2
L1: t4 = t1 * t2
L2: t5 = t3 + t4
return t5
```

   Basic blocks:

- **Block 1**: `t1 = a + b`, `t2 = c * d`, `if t1 > t2 goto L1`
- **Block 2**: `t3 = t1 - t2`, `goto L2`
- **Block 3 (L1)**: `t4 = t1 * t2`
- **Block 4 (L2)**: `t5 = t3 + t4`, `return t5`

2. **Constant folding:**

   Kode asli:

```
t1 = 5 + 3
t2 = t1 * 2
t3 = 10 / 2
x = t2 + t3
```

   Setelah constant folding:

```
t1 = 8          ; 5 + 3 = 8
t2 = 16         ; 8 * 2 = 16
t3 = 5          ; 10 / 2 = 5
x = 21          ; 16 + 5 = 21
```

   Atau lebih optimal:

```
x = 21          ; Semua di-fold menjadi satu nilai
```

3. **Constant propagation dan folding:**

   Kode asli:

```
x = 10
y = 20
t1 = x + 5
t2 = y * 2
z = t1 + t2
```

   Setelah constant propagation:

```
x = 10
y = 20
t1 = 10 + 5    ; x diganti dengan 10
t2 = 20 * 2    ; y diganti dengan 20
z = t1 + t2
```

   Setelah constant folding:

```
x = 10
y = 20
t1 = 15        ; 10 + 5 = 15
t2 = 40        ; 20 * 2 = 40
z = 55         ; 15 + 40 = 55
```

4. **Dead code elimination:**

   Kode asli:

```
x = 10
y = 20
t1 = x + y
t2 = 5 + 3      // Dead: t2 tidak digunakan
t3 = t2 - 2     // Dead: t3 tidak digunakan
z = t1 * 2
return z
```

   Setelah dead code elimination:

```
x = 10
y = 20
t1 = x + y
z = t1 * 2
return z
```

5. **Implementasi optimizer sederhana:**

   Algoritma:

   (a) Constant folding: Evaluasi ekspresi konstanta

   (b) Constant propagation: Ganti variabel konstanta dengan nilainya

   (c) Dead code elimination: Hapus assignment yang tidak digunakan

   (d) Iterate sampai tidak ada perubahan

6. **Perbedaan optimasi:**

   Local vs Global:

   - Local: Dalam satu basic block
   - Global: Lintas basic blocks, memerlukan data-flow analysis

   Machine-independent vs Machine-specific:

   - Machine-independent: Tidak bergantung pada target architecture (constant folding, dead code elimination)

- Machine-specific: Bergantung pada target (register allocation, instruction selection)

Constant folding vs Constant propagation:

- Constant folding: Evaluasi ekspresi konstanta langsung
- Constant propagation: Ganti penggunaan variabel konstanta dengan nilainya

7. **Benchmark untuk evaluasi optimasi:**

   Metrik:

   - **Code size**: Ukuran executable sebelum dan sesudah optimasi
   - **Execution time**: Waktu eksekusi program yang dikompilasi
   - **Compilation time**: Waktu kompilasi dengan optimasi

   Hasil yang diharapkan:

   - Code size: Biasanya lebih kecil setelah optimasi
   - Execution time: Biasanya lebih cepat setelah optimasi
   - Compilation time: Biasanya lebih lama karena optimasi memerlukan analisis tambahan

1.2.16 Jawaban Latihan Bab 16: Evaluasi dan Project Final

1. **Prepare Presentation:**

   Outline presentasi:

   (a) Introduction: Overview project

   (b) Architecture: Komponen utama compiler

   (c) Implementation: Tools dan teknik yang digunakan

   (d) Demo: Live demonstration dengan test cases

   (e) Evaluation: Benchmark results dan analisis

   (f) Challenges: Masalah yang dihadapi dan solusi

   (g) Conclusion: Lessons learned dan future work

   Demo script dengan 5 test cases:

   - Test case 1: Simple arithmetic expressions
   - Test case 2: Control flow (if-else, loops)
   - Test case 3: Function calls

- Test case 4: Error handling
- Test case 5: Complex program

2. **Tool Evaluation:**

   Tabel perbandingan:

   - Hand-written lexer vs Flex: Development time, maintainability, error messages
   - Recursive descent vs Bison: Grammar support, error recovery, performance
   - Trade-offs: Flexibility vs correctness, development time vs maintainability

   Justifikasi pilihan:

   - Pilih tools berdasarkan requirements project
   - Pertimbangkan complexity, performance, dan maintainability

3. **Benchmarking:**

   Test suite:

   - Small programs (10-100 lines)
   - Medium programs (100-1000 lines)
   - Large programs (1000+ lines)

   Metrik:

   - Compilation time
   - Code size (executable)
   - Execution time

   Laporan:

   - Tabel hasil benchmark
   - Analisis perbandingan
   - Identifikasi bottleneck

4. **Documentation:**

   README.md:

   - Overview project
   - Build instructions
   - Usage examples

- Test instructions

Design document:

- Architecture overview
- Component descriptions
- Data structures
- Algorithms used

API documentation:

- Public interfaces
- Function signatures
- Usage examples

5. **Reflection:**

   Refleksi pembelajaran:

   - Pemahaman tentang compiler phases
   - Pengalaman implementasi
   - Challenges yang dihadapi
   - Lessons learned

   3 challenges utama:

   (a) Error recovery yang baik

   (b) Optimasi yang efektif

   (c) Integrasi antar fase

   5 lessons learned:

   (a) Importance of good data structures

   (b) Error messages sangat penting untuk user experience

   (c) Testing adalah kunci untuk correctness

   (d) Documentation membantu maintenance

   (e) Trade-offs selalu ada dalam engineering decisions

   3 area untuk improvement:

   (a) Advanced optimizations

(b) Better error recovery

(c) Support untuk lebih banyak language features

6. **Peer Review:**

   Aspects to review:

   - Code quality dan organization
   - Correctness dari compiler
   - Performance dan optimizations
   - Documentation quality
   - Test coverage

   Feedback konstruktif:

   - Identifikasi strengths
   - Suggest improvements
   - Compare different approaches