Bab 1

Implementasi Lexer Sederhana (Hand-Written)

1.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep dan struktur hand-written lexer
2. Merancang state machine untuk token recognition
3. Mengimplementasikan lexer sederhana dalam C/C++ untuk subset bahasa C
4. Menangani whitespace, komentar (single-line dan multi-line), dan escape sequences
5. Mengimplementasikan error handling untuk token tidak valid
6. Membuat unit test untuk berbagai kasus input

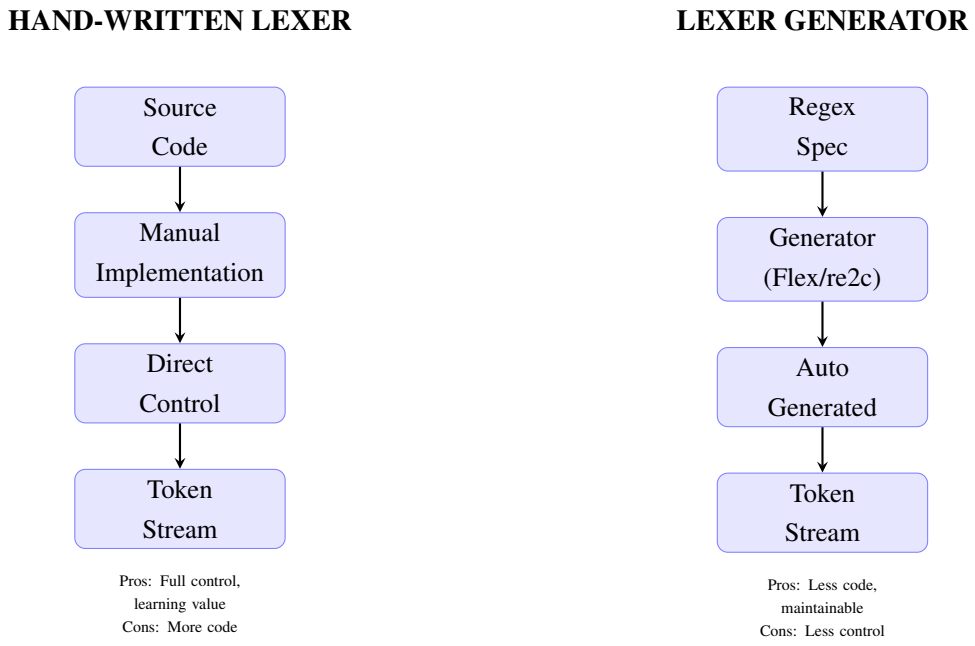
1.2 Pendahuluan

Setelah memahami konsep lexical analysis secara teori pada bab sebelumnya, pada bab ini kita akan mengimplementasikan lexer secara praktis menggunakan pendekatan **hand-written** (ditulis manual). Menurut sumber terbuka:

> "Hand-written lexers are possible: directly code a state machine, or use manual scanning logic. Requires careful handling of edge cases (e.g. unclosed strings/comments)."**?**

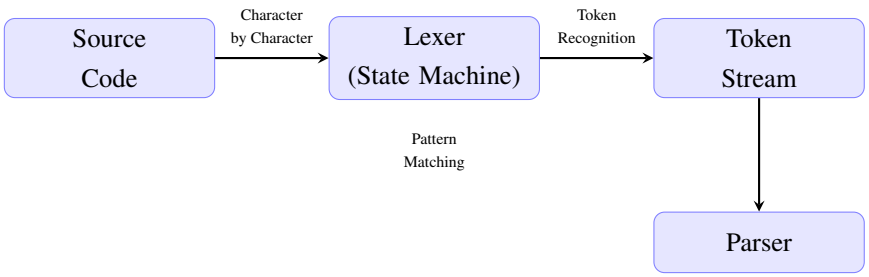
Pendekatan hand-written memberikan kontrol penuh terhadap implementasi dan sangat berguna untuk pembelajaran karena mahasiswa dapat memahami setiap detail proses tokenization. Meskipun lebih kompleks dibanding menggunakan generator seperti Flex atau re2c, hand-written lexer memberikan fleksibilitas dan pemahaman yang lebih dalam.

Gambar 1.1 menunjukkan perbandingan antara hand-written lexer dan lexer generator.



Gambar 1.1: Perbandingan hand-written lexer vs lexer generator

Gambar 1.2 menunjukkan alur umum proses tokenization dalam hand-written lexer.



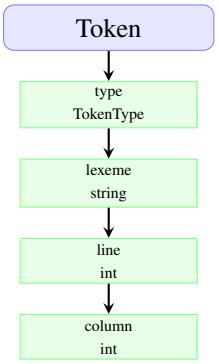
Gambar 1.2: Alur umum proses tokenization dalam hand-written lexer

1.3 Struktur Token

Sebelum mengimplementasikan lexer, kita perlu mendefinisikan struktur data untuk merepresentasikan token. Token minimal harus menyimpan:

1. **Token Type**: Jenis token (identifier, keyword, number, operator, dll.)
2. **Lexeme**: String aktual yang di-match dari source code
3. **Position Information**: Baris dan kolom untuk error reporting
4. **Value** (opsional): Nilai numerik untuk number literals

Gambar 1.3 menunjukkan struktur data token secara visual.

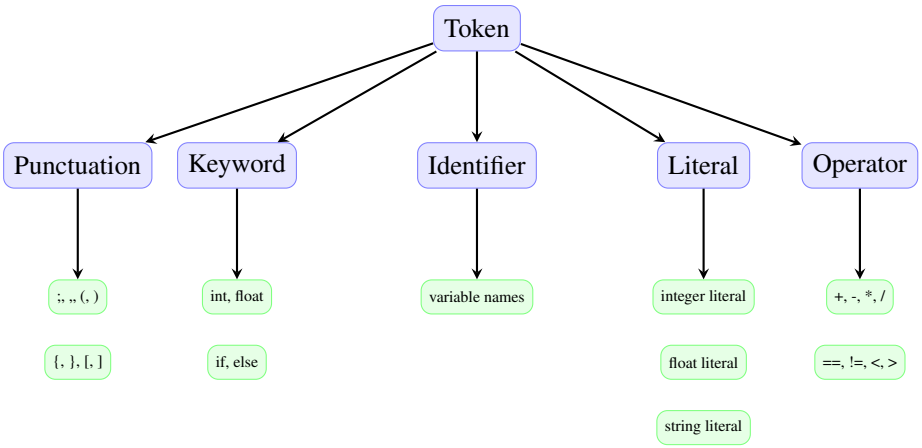


Gambar 1.3: Struktur data Token

1.3.1 Token Types

Token types dapat didefinisikan menggunakan enum. Berikut contoh untuk subset bahasa C:

Gambar 1.4 menunjukkan hierarki token types yang digunakan dalam lexer.



Gambar 1.4: Hierarki tipe token dalam lexer

Listing 1.1: Definisi Token Types

```
enum class TokenType {
    // Identifiers and Keywords
    IDENTIFIER,
    KEYWORD_INT, KEYWORD_FLOAT, KEYWORD_IF, KEYWORD_ELSE,
    KEYWORD_WHILE, KEYWORD_FOR, KEYWORD_RETURN,

    // Literals
    INTEGER_LITERAL,
    FLOAT_LITERAL,
    STRING_LITERAL,
    CHAR_LITERAL,
```

```
12
13      // Operators
14      OP_PLUS, OP_MINUS, OP_MULTIPLY, OP_DIVIDE,
15      OP_ASSIGN, OP_EQUAL, OP_NOT_EQUAL,
16      OP_LESS, OP_LESS_EQUAL, OP_GREATER, OP_GREATER_EQUAL,
17      OP_AND, OP_OR, OP_NOT,
18
19      // Punctuation
20      SEMICOLON, COMMA, DOT,
21      LPAREN, RPAREN,     // ( )
22      LBRACE, RBRACE,     // { }
23      LBRACKET, RBRACKET, // [ ]
24
25      // Special
26      END_OF_FILE,
27      INVALID
28  };
```

1.3.2 Token Structure

Struktur token dalam C++ dapat didefinisikan sebagai berikut:

Listing 1.2: Struktur Token

```
1  struct Token {
2      TokenType type;
3      std::string lexeme;
4      int line;
5      int column;
6      union {
7          int intValue;       // Untuk INTEGER_LITERAL
8          double floatValue; // Untuk FLOAT_LITERAL
9      };
10
11     Token(TokenType t, const std::string& lex, int l, int c)
12         : type(t), lexeme(lex), line(l), column(c) {}
13 };
```

1.4 Finite State Machine untuk Lexer

Lexical analysis secara fundamental adalah proses pattern matching yang dapat dimodelkan menggunakan **Finite State Machine (FSM)** atau **Finite Automata**. Menurut sumber dari Aoyama Gakuin University:

> "Lexical analysis breaks input text into lexemes which correspond to tokens. Usually implemented using regular languages → regex → NFA → DFA → (minimized) DFA for efficiency."**?**

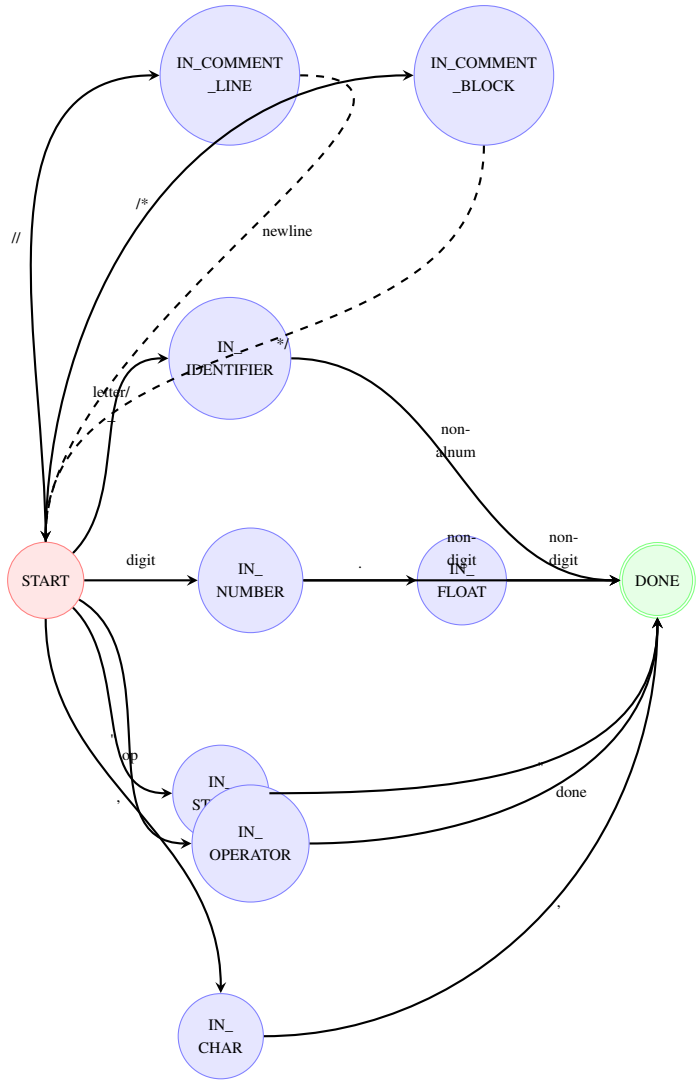
Dalam implementasi hand-written, kita tidak perlu membuat DFA secara eksplisit, tetapi kita menggunakan logika state machine dalam kode.

1.4.1 State Machine Design

State machine untuk lexer sederhana dapat memiliki state-state berikut:

- **START**: State awal, menunggu karakter pertama dari token
- **IN_IDENTIFIER**: Sedang membaca identifier atau keyword
- **IN_NUMBER**: Sedang membaca angka (integer atau float)
- **IN_FLOAT**: Setelah menemukan titik desimal
- **IN_STRING**: Sedang membaca string literal
- **IN_CHAR**: Sedang membaca character literal
- **IN_COMMENT_LINE**: Sedang membaca single-line comment
- **IN_COMMENT_BLOCK**: Sedang membaca multi-line comment
- **IN_OPERATOR**: Sedang membaca operator (mungkin multi-character)
- **DONE**: Token selesai dibaca

Gambar 1.5 menunjukkan state machine untuk lexer sederhana.



Gambar 1.5: State machine untuk hand-written lexer

1.4.2 State Transitions

Transisi state terjadi berdasarkan karakter yang dibaca:

1. **START** → **IN_IDENTIFIER**: Jika karakter adalah huruf atau underscore
2. **START** → **IN_NUMBER**: Jika karakter adalah digit
3. **START** → **IN_STRING**: Jika karakter adalah double quote (`"`)
4. **START** → **IN_CHAR**: Jika karakter adalah single quote (`'`)
5. **START** → **IN_COMMENT_LINE**: Jika menemukan `//`
6. **START** → **IN_COMMENT_BLOCK**: Jika menemukan `/*`

7. **START** → **IN_OPERATOR**: Jika karakter adalah operator

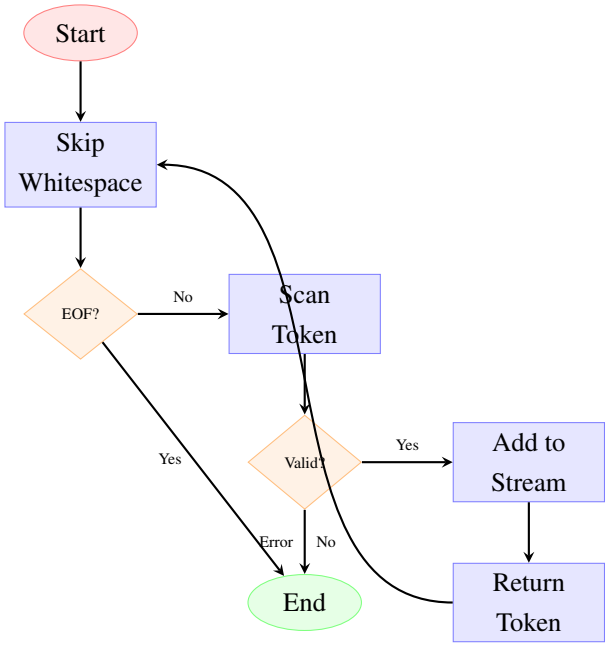
8. **IN_NUMBER** → **IN_FLOAT**: Jika menemukan titik desimal

9. **IN_IDENTIFIER** → **DONE**: Jika karakter bukan alphanumeric atau underscore

10. **IN_NUMBER** → **DONE**: Jika karakter bukan digit atau titik

11. **IN_STRING** → **DONE**: Jika menemukan closing quote (dengan handling escape)

Gambar 1.6 menunjukkan flowchart proses tokenization.



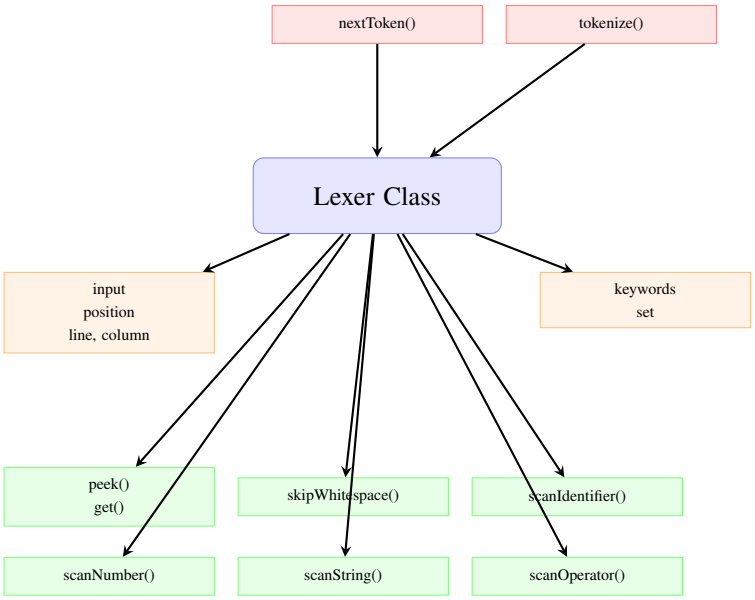
Gambar 1.6: Flowchart proses tokenization

1.5 Implementasi Lexer dalam C++

Berikut adalah implementasi lengkap lexer sederhana untuk subset bahasa C:

1.5.1 Kelas Lexer

Gambar 1.7 menunjukkan arsitektur kelas Lexer dan komponen-komponennya.



Gambar 1.7: Arsitektur kelas Lexer

Listing 1.3: Header File: lexer.h

```
#ifndef LEXER_H
#define LEXER_H

#include <string>
#include <unordered_set>
#include <vector>

enum class TokenType {
    IDENTIFIER,
    KEYWORD_INT, KEYWORD_FLOAT, KEYWORD_IF, KEYWORD_ELSE,
    KEYWORD_WHILE, KEYWORD_FOR, KEYWORD_RETURN,
    INTEGER_LITERAL, FLOAT_LITERAL,
    STRING_LITERAL, CHAR_LITERAL,
    OP_PLUS, OP_MINUS, OP_MULTIPLY, OP_DIVIDE,
    OP_ASSIGN, OP_EQUAL, OP_NOT_EQUAL,
    OP_LESS, OP_LESS_EQUAL, OP_GREATER, OP_GREATER_EQUAL,
    OP_AND, OP_OR, OP_NOT,
    SEMICOLON, COMMA, DOT,
    LPAREN, RPAREN, LBRACE, RBRACE,
    LBRACKET, RBRACKET,
    END_OF_FILE, INVALID
};

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

    Token(TokenType t, const std::string& lex, int l, int c)
        : type(t), lexeme(lex), line(l), column(c) {}
```

```
};

class Lexer {
private:
    std::string input;
    size_t position;
    int line;
    int column;
    std::unordered_set<std::string> keywords;

    char peek() const;
    char get();
    void skipWhitespace();
    void skipLineComment();
    void skipBlockComment();
    Token scanIdentifier();
    Token scanNumber();
    Token scanString();
    Token scanChar();
    Token scanOperator();
    TokenType getKeywordType(const std::string& lexeme) const;

public:
    Lexer(const std::string& source);
    Token nextToken();
    std::vector<Token> tokenize();
};

#endif
```

1.5.2 Implementasi Lexer

Listing 1.4: Implementasi: lexer.cpp (Bagian 1)

```
#include "lexer.h"
#include <cctype>
#include <stdexcept>

Lexer::Lexer(const std::string& source)
    : input(source), position(0), line(1), column(1) {
    // Initialize keywords
    keywords = {"int", "float", "if", "else",
                "while", "for", "return"};
}

char Lexer::peek() const {
    if (position >= input.length()) {
        return '\0';
    }
    return input[position];
}

char Lexer::get() {
```

```
    if (position >= input.length()) {
        return '\0';
    }
    char c = input[position++];
    if (c == '\n') {
        line++;
        column = 1;
    } else {
        column++;
    }
    return c;
}
```

1.5.3 Handling Whitespace dan Komentar

Listing 1.5: Implementasi: lexer.cpp (Bagian 2 - Whitespace dan Comments)

```
void Lexer::skipWhitespace() {
    while (position < input.length()) {
        char c = peek();
        if (std::isspace(c)) {
            get();
        } else if (c == '/' && position + 1 < input.length()
                   && input[position + 1] == '/') {
            skipLineComment();
        } else if (c == '/' && position + 1 < input.length()
                   && input[position + 1] == '*') {
            skipBlockComment();
        } else {
            break;
        }
    }
}

void Lexer::skipLineComment() {
    // Skip "//"
    get(); get();
    // Skip until newline or EOF
    while (peek() != '\n' && peek() != '\0') {
        get();
    }
}

void Lexer::skipBlockComment() {
    // Skip "/*"
    get(); get();
    while (position < input.length()) {
        if (peek() == '*' && position + 1 < input.length()
            && input[position + 1] == '/') {
            get(); get(); // Skip "*/"
            return;
        }
        get();
```

```
    }
    // Error: unclosed comment
    throw std::runtime_error("Unclosed block comment at line "
                             + std::to_string(line));
}
```

1.5.4 Scanning Identifier dan Keyword

Listing 1.6: Implementasi: lexer.cpp (Bagian 3 - Identifier)

```
Token Lexer::scanIdentifier() {
    int startLine = line;
    int startCol = column;
    std::string lexeme;

    // First character must be letter or underscore
    if (std::isalpha(peek()) || peek() == '_') {
        lexeme += get();
    }

    // Subsequent characters can be alphanumeric or underscore
    while (std::isalnum(peek()) || peek() == '_') {
        lexeme += get();
    }

    // Check if it's a keyword
    TokenType type = getKeywordType(lexeme);
    if (type != TokenType::IDENTIFIER) {
        return Token(type, lexeme, startLine, startCol);
    }

    return Token(TokenType::IDENTIFIER, lexeme, startLine, startCol);
}

TokenType Lexer::getKeywordType(const std::string& lexeme) const {
    if (lexeme == "int") return TokenType::KEYWORD_INT;
    if (lexeme == "float") return TokenType::KEYWORD_FLOAT;
    if (lexeme == "if") return TokenType::KEYWORD_IF;
    if (lexeme == "else") return TokenType::KEYWORD_ELSE;
    if (lexeme == "while") return TokenType::KEYWORD_WHILE;
    if (lexeme == "for") return TokenType::KEYWORD_FOR;
    if (lexeme == "return") return TokenType::KEYWORD_RETURN;
    return TokenType::IDENTIFIER;
}
```

1.5.5 Scanning Number Literals

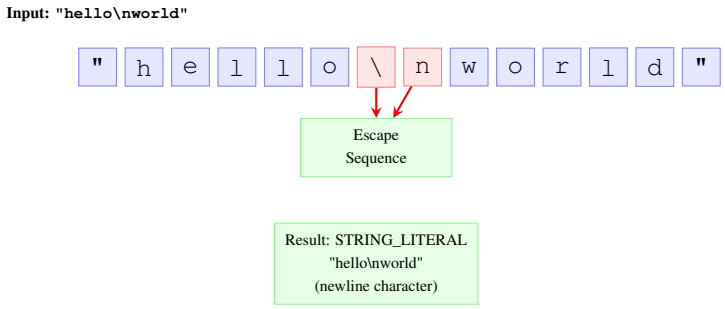
Listing 1.7: Implementasi: lexer.cpp (Bagian 4 - Numbers)

```
Token Lexer::scanNumber() {
    int startLine = line;
    int startCol = column;
```

```
    std::string lexeme;
    bool isFloat = false;

    // Read integer part
    while (std::isdigit(peek())) {
        lexeme += get();
    }

    // Check for decimal point
    if (peek() == '.') {
        lexeme += get();
        isFloat = true;

        // Read fractional part
        while (std::isdigit(peek())) {
            lexeme += get();
        }
    }

    // Check for exponent (optional, for future enhancement)
    if (peek() == 'e' || peek() == 'E') {
        lexeme += get();
        if (peek() == '+' || peek() == '-') {
            lexeme += get();
        }
        while (std::isdigit(peek())) {
            lexeme += get();
        }
        isFloat = true;
    }

    TokenType type = isFloat ? TokenType::FLOAT_LITERAL
                             : TokenType::INTEGER_LITERAL;
    return Token(type, lexeme, startLine, startCol);
}
```

1.5.6 Scanning String dan Character Literals

Gambar 1.8 menunjukkan contoh handling escape sequences dalam string literal.



Gambar 1.8: Handling escape sequences dalam string literal

Listing 1.8: Implementasi: lexer.cpp (Bagian 5 - Strings)

```
Token Lexer::scanString() {
    int startLine = line;
    int startCol = column;
    std::string lexeme;

    // Consume opening quote
    get(); // Skip opening "

    while (peek() != '"' && peek() != '\0') {
        if (peek() == '\\') {
            // Handle escape sequences
            get(); // Skip backslash
            char escaped = get();
            switch (escaped) {
                case 'n': lexeme += '\n'; break;
                case 't': lexeme += '\t'; break;
                case 'r': lexeme += '\r'; break;
                case '\\': lexeme += '\\'; break;
                case '"': lexeme += '"'; break;
                default: lexeme += '\\'; lexeme += escaped; break;
            }
        } else {
            lexeme += get();
        }
    }

    if (peek() == '\0') {
        // Unclosed string
        return Token(TokenType::INVALID, lexeme, startLine, startCol);
    }

    get(); // Consume closing "
    return Token(TokenType::STRING_LITERAL, lexeme, startLine, startCol);
}

Token Lexer::scanChar() {
    int startLine = line;
    int startCol = column;
    std::string lexeme;

    get(); // Skip opening '

    if (peek() == '\\') {
        // Escape sequence
        get(); // Skip backslash
        lexeme += get();
    } else {
        lexeme += get();
    }

    if (peek() != '\'') {
        return Token(TokenType::INVALID, lexeme, startLine, startCol);
    }
```

```
54
55      get(); // Consume closing '
56      return Token(TokenType::CHAR_LITERAL, lexeme, startLine, startCol);
57  }
```

1.5.7 Scanning Operators

Listing 1.9: Implementasi: lexer.cpp (Bagian 6 - Operators)

```
1  Token Lexer::scanOperator() {
2      int startLine = line;
3      int startCol = column;
4      char first = get();
5      std::string lexeme(1, first);
6
7      // Check for multi-character operators
8      char next = peek();
9
10     switch (first) {
11         case '=':
12             if (next == '=') {
13                 lexeme += get();
14                 return Token(TokenType::OP_EQUAL, lexeme, startLine,
   ↪ startCol);
15             }
16             return Token(TokenType::OP_ASSIGN, lexeme, startLine,
   ↪ startCol);
17
18         case '!':
19             if (next == '=') {
20                 lexeme += get();
21                 return Token(TokenType::OP_NOT_EQUAL, lexeme, startLine,
   ↪ startCol);
22             }
23             return Token(TokenType::OP_NOT, lexeme, startLine, startCol);
24
25         case '<':
26             if (next == '=') {
27                 lexeme += get();
28                 return Token(TokenType::OP_LESS_EQUAL, lexeme, startLine,
   ↪  startCol);
29             }
30             return Token(TokenType::OP_LESS, lexeme, startLine, startCol)
   ↪ ;
31
32         case '>':
33             if (next == '=') {
34                 lexeme += get();
35                 return Token(TokenType::OP_GREATER_EQUAL, lexeme,
   ↪ startLine, startCol);
36             }
37             return Token(TokenType::OP_GREATER, lexeme, startLine,
   ↪ startCol);
```

```
38
39          case '&':
40              if (next == '&') {
41                  lexeme += get();
42                  return Token(TokenType::OP_AND, lexeme, startLine,
↪ startCol);
43              }
44              return Token(TokenType::INVALID, lexeme, startLine, startCol)
↪ ;
45
46          case '|':
47              if (next == '|') {
48                  lexeme += get();
49                  return Token(TokenType::OP_OR, lexeme, startLine,
↪ startCol);
50              }
51              return Token(TokenType::INVALID, lexeme, startLine, startCol)
↪ ;
52
53          case '+':
54              return Token(TokenType::OP_PLUS, lexeme, startLine, startCol)
↪ ;
55          case '-':
56              return Token(TokenType::OP_MINUS, lexeme, startLine, startCol
↪ );
57          case '*':
58              return Token(TokenType::OP_MULTIPLY, lexeme, startLine,
↪ startCol);
59          case '/':
60              return Token(TokenType::OP_DIVIDE, lexeme, startLine,
↪ startCol);
61
62          default:
63              return Token(TokenType::INVALID, lexeme, startLine, startCol)
↪ ;
64      }
65  }
```

1.5.8 Main Tokenization Function

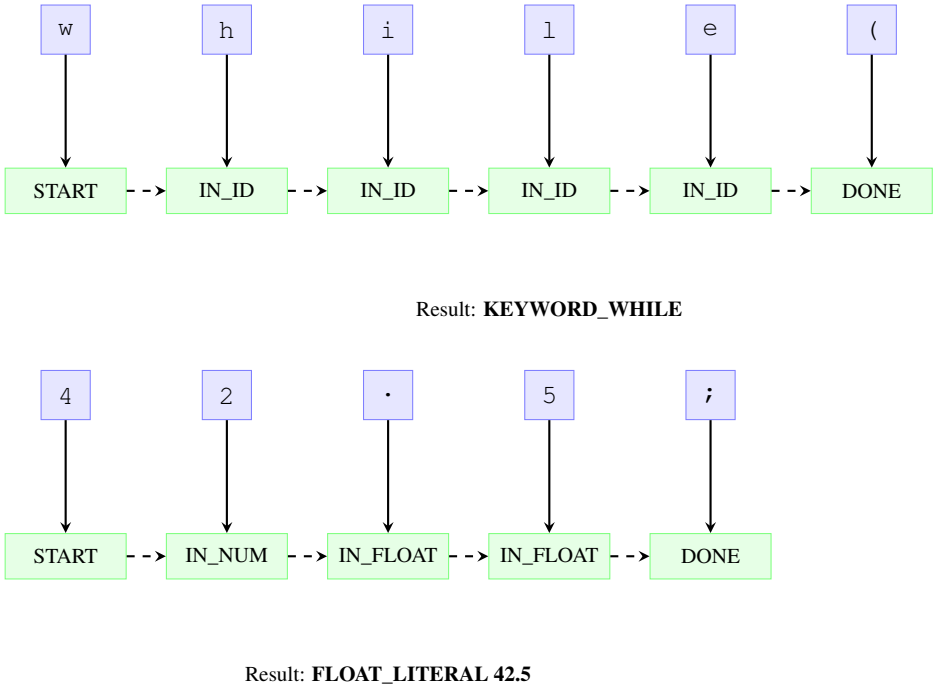
Listing 1.10: Implementasi: lexer.cpp (Bagian 7 - Main Function)

```
1  Token Lexer::nextToken() {
2      skipWhitespace();
3
4      if (position >= input.length()) {
5          return Token(TokenType::END_OF_FILE, "", line, column);
6      }
7
8      char c = peek();
9
10     // Identifier or keyword
11     if (std::isalpha(c) || c == '_') {
```

```
12          return scanIdentifier();
13      }
14
15      // Number
16      if (std::isdigit(c)) {
17          return scanNumber();
18      }
19
20      // String literal
21      if (c == '"') {
22          return scanString();
23      }
24
25      // Character literal
26      if (c == '\'') {
27          return scanChar();
28      }
29
30      // Operators and punctuation
31      if (c == '+' || c == '-' || c == '*' || c == '/' ||
32          c == '=' || c == '!' || c == '<' || c == '>' ||
33          c == '&' || c == '|') {
34          return scanOperator();
35      }
36
37      // Punctuation
38      if (c == ';') {
39          get();
40          return Token(TokenType::SEMICOLON, ";", line, column - 1);
41      }
42      if (c == ',') {
43          get();
44          return Token(TokenType::COMMA, ",", line, column - 1);
45      }
46      if (c == '.') {
47          get();
48          return Token(TokenType::DOT, ".", line, column - 1);
49      }
50      if (c == '(') {
51          get();
52          return Token(TokenType::LPAREN, "(", line, column - 1);
53      }
54      if (c == ')') {
55          get();
56          return Token(TokenType::RPAREN, ")", line, column - 1);
57      }
58      if (c == '{') {
59          get();
60          return Token(TokenType::LBRACE, "{", line, column - 1);
61      }
62      if (c == '}') {
63          get();
64          return Token(TokenType::RBRACE, "}", line, column - 1);
65      }
```

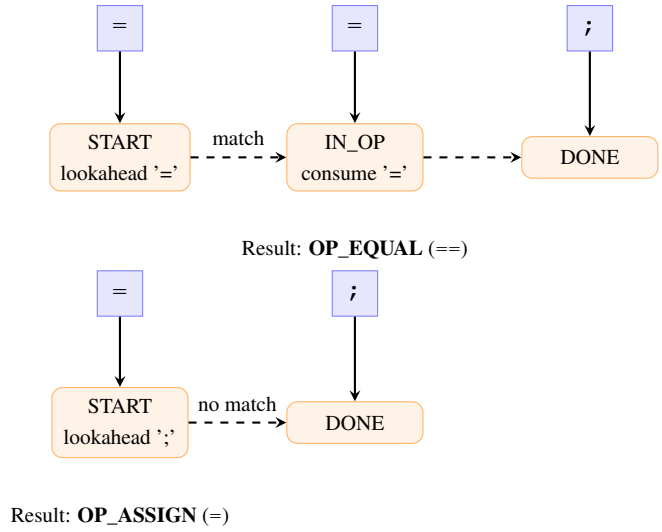
```cpp
    if (c == '[') {
        get();
        return Token(TokenType::LBRACKET, "[", line, column - 1);
    }
    if (c == ']') {
        get();
        return Token(TokenType::RBRACKET, "]", line, column - 1);
    }

    // Unknown character
    get();
    return Token(TokenType::INVALID, std::string(1, c), line, column - 1)
↪ ;
}

std::vector<Token> Lexer::tokenize() {
    std::vector<Token> tokens;
    Token token = nextToken();
    while (token.type != TokenType::END_OF_FILE) {
        tokens.push_back(token);
        token = nextToken();
    }
    tokens.push_back(token); // Add EOF token
    return tokens;
}
```

Gambar 1.9 menunjukkan proses scanning untuk identifier dan keyword.



Gambar 1.9: Proses scanning keyword/identifier dan literal numerik pada lexer

Gambar 1.10 menunjukkan proses scanning untuk operator multi-character.



Gambar 1.10: Proses scanning operator: perbandingan == dan =

1.6 Error Handling

Error handling dalam lexer harus menangani berbagai kasus edge case:

1.6.1 Unclosed Strings dan Comments

- **Unclosed String**: Jika string literal tidak ditutup sebelum EOF, lexer harus mengembalikan token INVALID dengan informasi posisi yang tepat.

- **Unclosed Block Comment**: Jika komentar blok tidak ditutup, dapat di-handle dengan exception atau mengembalikan error token.

1.6.2 Invalid Characters

Karakter yang tidak valid (tidak termasuk dalam kategori token manapun) harus dikembalikan sebagai token INVALID dengan informasi posisi untuk error reporting yang baik.

Tabel 1.1 menunjukkan contoh-contoh token yang valid dan tidak valid.

Input	Token Type	Keterangan
`int`	KEYWORD_INT	Keyword valid
`hello`	IDENTIFIER	Identifier valid
`42`	INTEGER_LITERAL	Integer valid
`3.14`	FLOAT_LITERAL	Float valid
`"hello"`	STRING_LITERAL	String valid
`==`	OP_EQUAL	Operator multi-character
`=`	OP_ASSIGN	Operator single-character
	INVALID	Karakter tidak valid
`"unclosed`	INVALID	String tidak tertutup
`/* comment`	Error	Comment tidak tertutup

Tabel 1.1: Contoh token valid dan tidak valid

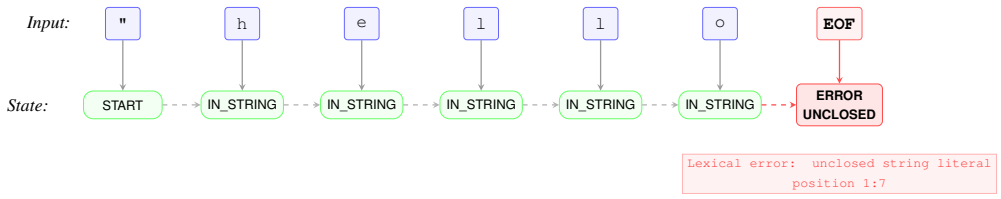
1.6.3 Malformed Numbers

Contoh kasus malformed:

- `123.` (titik tanpa digit setelahnya)
- `.456` (titik tanpa digit sebelumnya) - dapat di-handle sebagai valid float
- `12.34.56` (multiple decimal points)

Implementasi dapat memilih untuk menerima atau menolak format tertentu sesuai kebutuhan.

Gambar 1.11 menunjukkan contoh error handling untuk unclosed string.



Gambar 1.11: Transisi state lexer dan penanganan kesalahan pada string literal yang tidak tertutup

1.7 Testing Lexer

Unit testing sangat penting untuk memastikan lexer bekerja dengan benar. Berikut contoh test cases:

1.7.1 Test Cases untuk Identifier dan Keyword

Listing 1.11: Test Cases: Identifiers dan Keywords

```
void testIdentifiers() {
    Lexer lexer("int x = 42;");
    Token t1 = lexer.nextToken(); // Should be KEYWORD_INT
    Token t2 = lexer.nextToken(); // Should be IDENTIFIER "x"
    Token t3 = lexer.nextToken(); // Should be OP_ASSIGN
    // ...
}
```

1.7.2 Test Cases untuk Numbers

- `42` → INTEGER_LITERAL
- `3.14` → FLOAT_LITERAL
- `123.456` → FLOAT_LITERAL
- `0` → INTEGER_LITERAL

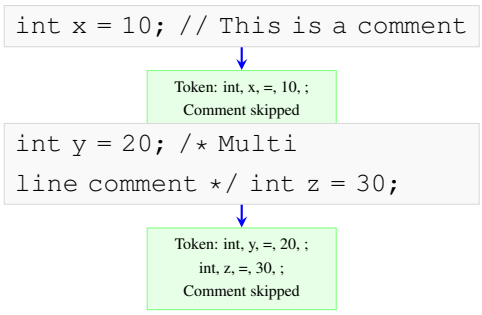
1.7.3 Test Cases untuk Strings

- `"hello"` → STRING_LITERAL dengan value "hello"
- `"hello\nworld"` → STRING_LITERAL dengan escape sequence
- `"unclosed` → INVALID (unclosed string)

1.7.4 Test Cases untuk Comments

- `// single line comment` → Di-skip, tidak menghasilkan token
- `/* multi-line comment */` → Di-skip
- `/* unclosed comment` → Error atau exception

Gambar 1.12 menunjukkan proses handling komentar dalam lexer.



Gambar 1.12: Handling komentar dalam lexer

1.8 Contoh Penggunaan

Berikut contoh penggunaan lexer untuk tokenize source code sederhana:

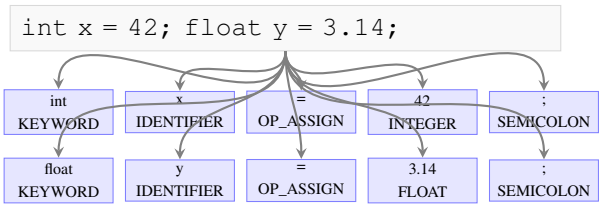
Listing 1.12: Contoh Penggunaan Lexer

```
#include "lexer.h"
#include <iostream>

int main() {
    std::string source = R"(
        int x = 42;
        float y = 3.14;
        if (x > 10) {
            return y;
        }
    )";

    Lexer lexer(source);
    std::vector<Token> tokens = lexer.tokenize();

    for (const auto& token : tokens) {
        std::cout << "Token: " << token.lexeme
                  << " Type: " << static_cast<int>(token.type)
                  << " Line: " << token.line
                  << " Column: " << token.column << std::endl;
    }

    return 0;
}
```

Output yang diharapkan:

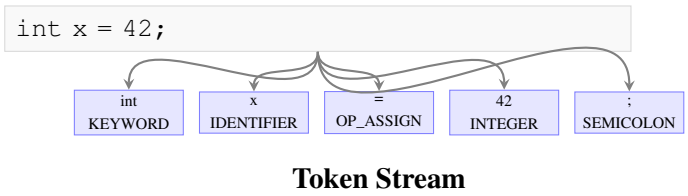
```
Token: int Type: 1 Line: 2 Column: 9
Token: x Type: 0 Line: 2 Column: 13
Token: = Type: 13 Line: 2 Column: 15
Token: 42 Type: 8 Line: 2 Column: 17
Token: ; Type: 20 Line: 2 Column: 19
...
```

Gambar 1.13 menunjukkan contoh lengkap tokenization untuk program sederhana.



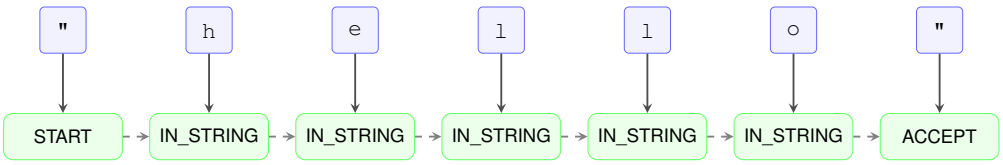
Gambar 1.13: Contoh lengkap tokenization untuk program sederhana

Gambar 1.14 menunjukkan visualisasi token stream untuk contoh kode sederhana.



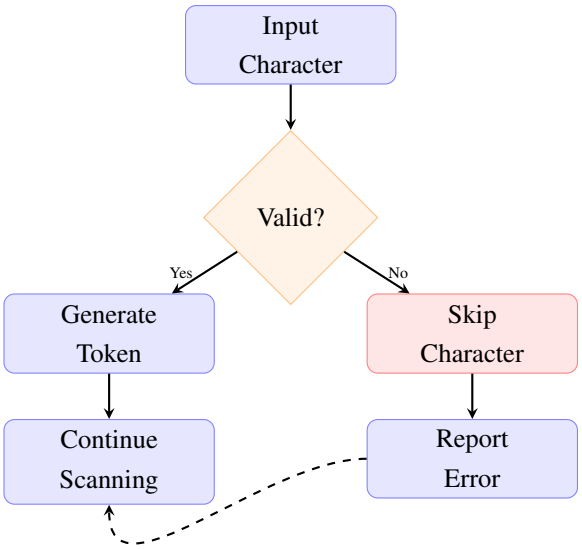
Gambar 1.14: Contoh tokenization: `int x = 42;` menjadi token stream

Gambar 1.15 menunjukkan proses scanning secara detail untuk string `"hello"`.



Gambar 1.15: Representasi formal proses analisis leksikal pada string literal `"hello"`

Gambar 1.16 menunjukkan strategi error recovery dalam lexer.



Gambar 1.16: Strategi error recovery dalam lexer

1.9 Best Practices

Beberapa best practices dalam implementasi hand-written lexer:

1. **Separation of Concerns**: Pisahkan logika untuk setiap jenis token ke fungsi terpisah
2. **Position Tracking**: Selalu track line dan column untuk error reporting yang baik
3. **Lookahead**: Gunakan `peek()` untuk lookahead tanpa mengkonsumsi karakter
4. **Error Recovery**: Rancang strategi error recovery (misalnya skip invalid character dan lanjut)
5. **Testing**: Buat comprehensive test suite untuk semua edge cases
6. **Documentation**: Dokumentasikan token types dan format yang didukung

1.10 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Struktur token dan token types untuk subset bahasa C
2. Konsep finite state machine dalam konteks lexical analysis
3. Implementasi hand-written lexer dalam C++ dengan handling:

- Identifier dan keyword recognition
- Number literals (integer dan float)
- String dan character literals dengan escape sequences
- Operators (single dan multi-character)
- Whitespace dan komentar (single-line dan multi-line)

4. Error handling untuk edge cases
5. Testing strategies untuk lexer

Implementasi hand-written lexer memberikan pemahaman mendalam tentang proses tokenization dan menjadi dasar untuk memahami bagaimana lexer generator seperti Flex bekerja di belakang layar.

1.11 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang implementasi lexer, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* **?** - Bab 3: Lexical Analysis
- **Engineering a Compiler**: Cooper & Torczon (2011) **?** - Bab 2: Scanning
- **OpenGenus - Build Lexer**: Tutorial tentang hand-written lexer **?**
- **Aoyama Gakuin University**: Lecture notes tentang lexical analysis **?**
- **GeeksforGeeks**: Contoh implementasi lexical analyzer dalam C++ [1]
- **Programming Notes**: Tutorial tentang simple lexer menggunakan finite state machine [2]

[1] https://www.geeksforgeeks.org/cpp/lexical-analyzer-in-cpp/
[2] https://www.programmingnotes.org/4699/cpp-simple-lexer-using-a-finite-state-machine/