

# Bab 3

## Lexical Analysis dan Regular Expression

### Sub-CPMK yang Dicakup dalam Bab Ini:

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

### 3.1 Pengenalan Lexical Analysis

#### 3.1.1 Peran Lexical Analyzer

*Lexical Analysis* (atau *Lexer/Scanner*) adalah tahap pertama dalam kompilasi yang bertugas membaca aliran karakter dari program sumber dan mengelompokkannya ke dalam unit-unit bermakna yang disebut *token* [1, 2].

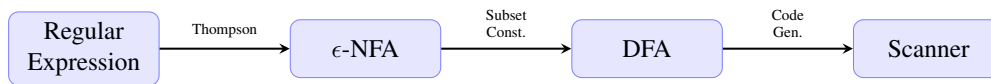
- Menghapus whitespace dan komentar.
- Melaporkan lexical error.

#### 3.1.2 Alur Teoretis Lexical Analysis

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

#### 3.1.3 Token dan Lexeme

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



Gambar 3.1: Alur konversi dari regular expression ke implementasi scanner

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

## 3.2 Regular Expression

### 3.2.1 Definisi dan Operasi Dasar

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

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

Tabel 3.1: Operator dasar Regular Expression

### 3.2.2 Implementasi Leksikal dengan Regex

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

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

### 3.2.3 Sifat Regular Language

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

### 3.3 Finite Automata

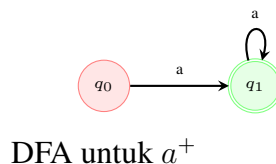
Finite automata adalah model matematika yang digunakan untuk mengenali string dalam suatu bahasa.

#### 3.3.1 NFA (Nondeterministic Finite Automata)

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

#### 3.3.2 DFA (Deterministic Finite Automata)

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



Gambar 3.2: Contoh DFA sederhana

#### 3.3.3 Konversi NFA ke DFA: Subset Construction

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

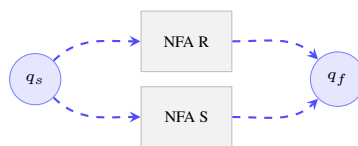
1. **Start state DFA** adalah  $\epsilon$ -closure dari start state NFA.
2. Untuk setiap state DFA, hitung transisi ke set NFA states berikutnya berdasarkan simbol input.
3. Ambil  $\epsilon$ -closure dari setiap set tersebut untuk menjadi state DFA baru.
4. State DFA menjadi **accept state** jika mengandung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

### 3.4 Metode Konstruksi NFA: Algoritma Thompson

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

### 3.4.1 Template Dasar Thompson

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



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

## 3.5 Implementasi Lexer Hand-written

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

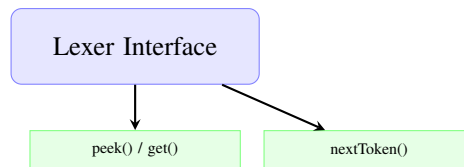
### 3.5.1 Struktur Data Token

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

```
1 enum class TokenType {  
2     IDENTIFIER, KEYWORD, INTEGER_LITERAL, FLOAT_LITERAL,  
3     OP_PLUS, OP_ASSIGN, SEMICOLON, END_OF_FILE, INVALID  
4 };  
5  
6 struct Token {  
7     TokenType type;  
8     std::string lexeme;  
9     int line;  
10    int column;  
11 };
```

### 3.5.2 Arsitektur Kelas Lexer

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



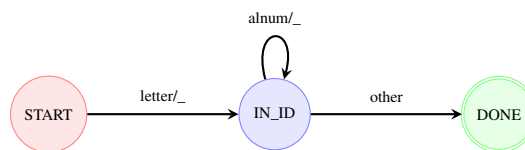
Gambar 3.4: Komponen utama kelas Lexer

### 3.6 Logika State Machine Token

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

#### 3.6.1 State Machine untuk Identifier dan Keywords

Transisi dimulai dari karakter alfabet atau `_`, kemudian diikuti oleh karakter alfanumerik. Setelah lexeme terkumpul, dilakukan pengecekan apakah ia termasuk kata kunci (*keyword*). Selain Flex, terdapat berbagai generator lexer modern lainnya seperti `re2c` yang berorientasi pada kecepatan [3] dan `RE/flex` yang mendukung C++ dengan lebih baik [4].



Gambar 3.5: State machine untuk identifikasi identifier

#### 3.6.2 State Machine untuk Angka (Literals)

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

```

1 Token Lexer::scanNumber() {
2     bool isFloat = false;
3     std::string lexeme;
4     while (isdigit(peek())) lexeme += get();
5     if (peek() == '.') {
6         isFloat = true;
7         lexeme += get();
8         while (isdigit(peek())) lexeme += get();
9     }
10    return Token(isFloat ? FLOAT : INT, lexeme);
11 }
  
```

## 3.7 Handling Komentar dan Kesalahan

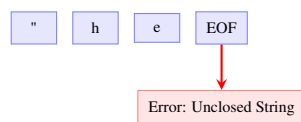
### 3.7.1 Whitespace dan Komentar

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

```
1 void Lexer::skipLineComment() {  
2     get(); get(); // skip //  
3     while (peek() != '\n' && peek() != '\0') get();  
4 }
```

### 3.7.2 Penanganan Kesalahan Leksikal

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



Gambar 3.6: Ilustrasi unclosed string error

## 3.8 Lexer Generator dan Best Practices

### 3.8.1 Penggunaan Flex

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

### 3.8.2 Best Practices

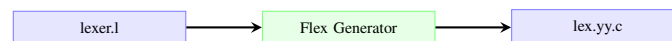
1. **Lookahead:** Gunakan fungsi `peek()` untuk mengintip karakter berikutnya tanpa menghabiskannya.
2. **Position Tracking:** Simpan baris dan kolom untuk mempermudah *debugging* bagi pemrogram.
3. **Longest Match:** Selalu ambil token terpanjang yang cocok (misal: `==` lebih prioritas daripada `=`).

### 3.9 Lexer Generator: Flex dan re2c

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

#### 3.9.1 Flex (Fast Lexical Analyzer)

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



Gambar 3.7: Alur kerja Flex

#### 3.9.2 re2c

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

#### 3.9.3 Perbandingan Pendekatan

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

Tabel 3.2: Perbandingan metode konstruksi lexer

#### Aktivitas Pembelajaran

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

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

### Latihan dan Refleksi

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

### Asesmen (Evaluasi Kinerja)

#### Instrumen Penilaian untuk Sub-CPMK 2.1-2.2

##### A. Pilihan Ganda

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

(c) Make

(d) GDB

### B. Essay

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

**Rubrik Penilaian:** Lihat Lampiran A

### Checklist Pencapaian Kompetensi

*Centang item berikut setelah Anda yakin telah menguasainya:*

- ☐ Saya dapat membuat regular expression untuk token spesifik bahasa
- ☐ Saya dapat mengimplementasikan NFA untuk token recognition
- ☐ Saya dapat mengkonversi NFA ke DFA
- ☐ Saya dapat mengimplementasikan DFA lexer
- ☐ Saya memahami perbedaan hand-coded vs generated lexer
- ☐ Saya dapat menggunakan tools modern untuk lexical analysis

### Rangkuman

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

#### Poin Kunci:

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

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

# Daftar Pustaka

- [1] Aoyama Gakuin University. *Compiler Lecture 5: Lexical Analysis*. Lecture notes. 2024. URL: <https://www.sw.it.aoyama.ac.jp/2025/Compiler/lecture5.html>.
- [2] OpenGenus. *Build Lexer*. Tutorial on hand-written lexers. 2024. URL: <https://iq.opengenus.org/build-lexer/>.
- [3] Wikipedia. *re2c*. Encyclopedia entry. 2024. URL: <https://en.wikipedia.org/wiki/Re2c>.
- [4] Wikipedia. *RE/flex*. Encyclopedia entry. 2024. URL: <https://en.wikipedia.org/wiki/Draft:RE/flex>.