Bab 2

Regular Expression dan Finite Automata untuk Lexical Analysis

2.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep regular expression dan regular language
2. Menjelaskan perbedaan antara NFA (Nondeterministic Finite Automata) dan DFA (Deterministic Finite Automata)
3. Mengkonversi regular expression ke NFA menggunakan algoritma Thompson
4. Mengkonversi NFA ke DFA menggunakan subset construction
5. Mengimplementasikan NFA dan DFA sederhana dalam C/C++
6. Membuat recognizer untuk pattern token sederhana menggunakan finite automata
7. Memahami hubungan antara regular expression, finite automata, dan lexical analysis

2.2 Pendahuluan

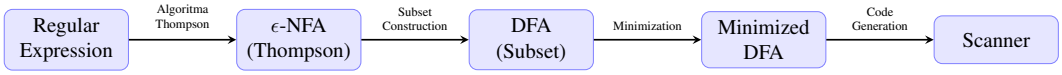
Spesifikasi token untuk proyek compiler subset C telah didefinisikan di Bab 1 (Bagian **??**). Dalam bab ini kita mempelajari teori formal yang mendasari lexical analysis—regular expression dan finite automata—dan mengaitkan contoh ke token-token proyek (identifier, number, keyword, operator, punctuator) bila relevan.

Sebagai landasan untuk memahami lexical analysis, kita perlu mempelajari teori formal yang mendasarinya. Menurut sumber dari Aoyama Gakuin University:

> "Lexical analysis breaks input text into lexemes which correspond to tokens. Usually implemented using regular languages $\rightarrow$ regex $\rightarrow$ NFA $\rightarrow$ DFA $\rightarrow$ (minimized) DFA for efficiency."[1]

Alur ini menunjukkan bahwa lexical analysis dalam kompilator modern menggunakan teori formal language, khususnya regular languages, yang direpresentasikan sebagai regular expressions dan kemudian diimplementasikan sebagai finite automata untuk efisiensi.

Gambar 2.1 menunjukkan alur lengkap dari regular expression hingga implementasi praktis dalam lexical analyzer.



Gambar 2.1: Alur konversi dari regular expression ke implementasi scanner

2.3 Regular Expression dan Regular Language

2.3.1 Definisi Regular Expression

Regular expression (regex) adalah notasi formal untuk mendeskripsikan pola string dalam suatu bahasa. Regular expression menggunakan operasi-operasi dasar untuk membangun pattern yang lebih kompleks.

Operasi-operasi dasar dalam regular expression meliputi:

1. **Literal**: Karakter tunggal, misalnya `a` mencocokkan string “a”
2. **Concatenation**: Penggabungan, misalnya `ab` mencocokkan string “ab”
3. **Union/Alternation**: Pilihan, misalnya `a|b` mencocokkan “a” atau “b”
4. **Kleene Star**: Nol atau lebih pengulangan, misalnya `a*` mencocokkan “”, “a”, “aa”, “aaa”, dll.
5. **Kleene Plus**: Satu atau lebih pengulangan, misalnya `a+` mencocokkan “a”, “aa”, “aaa”, dll.
6. **Optional**: Nol atau satu, misalnya `a?` mencocokkan “” atau “a”
7. **Character Class**: Set karakter, misalnya `[0-9]` mencocokkan digit 0-9

2.3.2 Contoh Regular Expression untuk Token

Dalam lexical analysis, setiap jenis token didefinisikan menggunakan regular expression. Berikut beberapa contoh:

- **Identifier**: `[a-zA-Z_][a-zA-Z0-9_]*`

  - Dimulai dengan huruf atau underscore
  - Diikuti oleh nol atau lebih huruf, digit, atau underscore

- **Integer Literal**: `[0-9]+`
  - Satu atau lebih digit
- **Floating Point**: `[0-9]+\.[0-9]+`
  - Digit, titik desimal, digit
- **String Literal**: `"([^"]|\\.)*"`
  - Dimulai dan diakhiri dengan tanda kutip
  - Berisi karakter apapun kecuali tanda kutip (atau escape sequence)
- **Whitespace**: `[ \t\n]+`
  - Satu atau lebih spasi, tab, atau newline
- **Operator**: `+|-|*|/|=|==|!=`
  - Operator aritmatika dan perbandingan

2.3.3 Regular Language

Bahasa yang dapat dinyatakan dengan regular expression disebut **regular language**. Regular language memiliki sifat-sifat penting:

- Dapat dikenali oleh finite automata (NFA atau DFA)
- Tertutup terhadap operasi union, concatenation, dan Kleene star
- Tidak dapat mengekspresikan struktur nested (seperti matching parentheses)
- Cukup untuk mendeskripsikan sebagian besar token dalam bahasa pemrograman

2.4 Finite Automata

Finite automata adalah model matematika yang digunakan untuk mengenali string dalam suatu bahasa. Terdapat dua jenis utama: NFA (Nondeterministic Finite Automata) dan DFA (Deterministic Finite Automata).

2.4.1 Definisi Formal

Finite Automaton didefinisikan sebagai tuple $(Q, \Sigma, \delta, q_0, F)$ dimana:

- Q: Himpunan state (keadaan) yang terbatas
- Σ: Alphabet (himpunan simbol input)
- δ: Fungsi transisi (transition function)
- q_0: Start state (state awal)
- F: Himpunan accept states (final states)

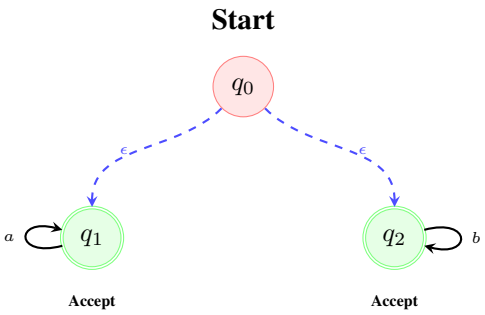
2.4.2 NFA (Nondeterministic Finite Automata)

NFA memiliki karakteristik:

- Untuk suatu state dan input symbol, dapat memiliki **nol, satu, atau lebih** transisi
- Dapat memiliki ϵ**-transitions** (epsilon transitions) yang tidak mengonsumsi input
- Lebih mudah dikonstruksi dari regular expression
- Simulasi memerlukan backtracking atau multiple states tracking

Contoh NFA untuk pattern `a | b`:

Gambar 2.2 menunjukkan NFA untuk pattern `a | b` yang menggunakan ϵ-transitions untuk branching.



Gambar 2.2: NFA untuk pattern `a | b` dengan ϵ-transitions

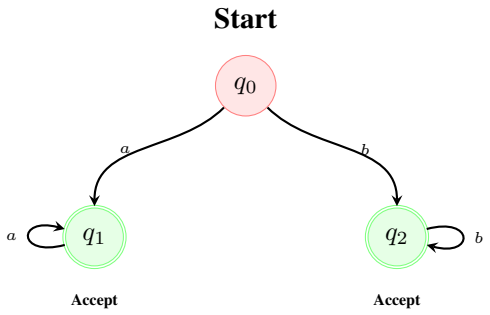
State q_0 adalah start state, q_1 dan q_2 adalah accept states. Dari q_0, dengan ϵ-transition dapat menuju q_1 atau q_2, kemudian dari q_1 dapat menerima 'a', dan dari q_2 dapat menerima 'b'.

2.4.3 DFA (Deterministic Finite Automata)

DFA memiliki karakteristik:

- Untuk setiap state dan input symbol, terdapat **tepat satu** transisi
- Tidak memiliki ϵ-transitions
- Lebih efisien untuk simulasi (deterministic)
- Setiap NFA dapat dikonversi menjadi DFA yang ekuivalen

Contoh DFA untuk pattern `a|b`:
Gambar 2.3 menunjukkan DFA yang ekuivalen dengan NFA di atas, tetapi deterministik.



Gambar 2.3: DFA untuk pattern `a|b` (deterministik)

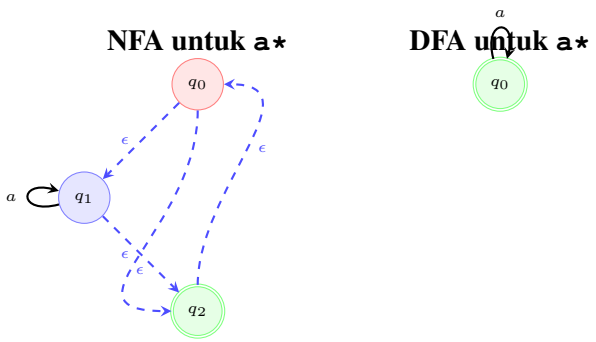
DFA ini deterministik: dari q_0, input 'a' selalu menuju q_1, input 'b' selalu menuju q_2. Tidak ada ϵ-transitions dan setiap state memiliki tepat satu transisi untuk setiap input symbol.

2.4.4 Perbedaan NFA dan DFA

Perbedaan utama antara NFA dan DFA dapat dilihat pada Tabel 2.1 dan perbandingan visual pada Gambar 2.4.

Aspek	**NFA**	**DFA**
Transisi per state	Bisa 0, 1, atau lebih	Tepat 1
ϵ-transitions	Diizinkan	Tidak diizinkan
Efisiensi simulasi	Perlu backtracking	Linear time
Jumlah states	Biasanya lebih sedikit	Bisa lebih banyak
Konstruksi dari regex	Lebih mudah	Lebih kompleks

Tabel 2.1: Perbandingan NFA dan DFA



Gambar 2.4: Perbandingan visual NFA dan DFA untuk pattern `a*` (keduanya ekuivalen)

2.5 Konversi Regular Expression ke NFA: Algoritma Thompson

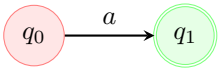
Algoritma Thompson adalah metode sistematis untuk mengkonversi regular expression menjadi ϵ-NFA. Algoritma ini menggunakan pendekatan rekursif dengan template untuk setiap operasi regex.

2.5.1 Template Dasar

Algoritma Thompson menggunakan template untuk setiap operasi regex. Template-template berikut menunjukkan konstruksi dasar yang digunakan.

Literal (Karakter Tunggal)

Untuk regex `a`, NFA-nya adalah:



Gambar 2.5: Template NFA untuk literal `a`

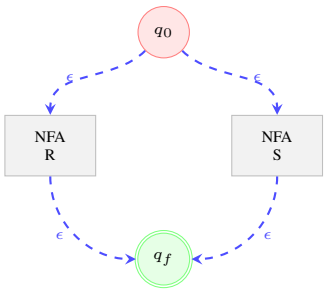
Concatenation (RS)

Untuk regex `RS`, NFA-nya dibangun dengan menghubungkan NFA untuk R dan S menggunakan ϵ-transition:

Gambar 2.6: Template NFA untuk concatenation `RS`

Union (R|S)

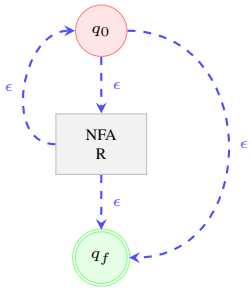
Untuk regex `R|S`, NFA-nya menggunakan ϵ-transitions untuk branching:



Gambar 2.7: Template NFA untuk union `R|S`

Kleene Star (R*)

Untuk regex `R*`, NFA-nya memiliki loop dengan ϵ-transitions:

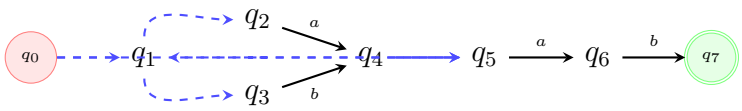


Gambar 2.8: Template NFA untuk Kleene star `R*`

2.5.2 Contoh: Konversi `(a|b)*abb`

Mari kita konstruksi NFA untuk regex `(a|b)*abb` menggunakan algoritma Thompson. Gambar 2.9 menunjukkan proses konstruksi langkah demi langkah.

1. **Literal 'a' dan 'b'**: Buat NFA untuk masing-masing
2. **Union (a|b)**: Gabungkan dengan ϵ-transitions
3. **Kleene Star ((a|b)*)**: Tambahkan loop dengan ϵ-transitions
4. **Concatenation dengan 'a'**: Tambahkan NFA untuk 'a'
5. **Concatenation dengan 'b'**: Tambahkan NFA untuk 'b' (dua kali)



Gambar 2.9: NFA untuk regex `(a|b)*abb` menggunakan algoritma Thompson

Hasil akhirnya adalah NFA yang dapat mengenali string seperti “abb”, “aabb”, “babb”, “ababb”, dll. NFA ini memiliki ϵ-transitions yang memungkinkan multiple paths untuk input yang sama.

2.6 Konversi NFA ke DFA: Subset Construction

Karena NFA tidak deterministik dan simulasi NFA bisa tidak efisien, kita perlu mengkonversi NFA menjadi DFA yang ekuivalen menggunakan algoritma **subset construction**.

2.6.1 Konsep ϵ-Closure

Sebelum subset construction, kita perlu memahami konsep **ϵ-closure**:

- **ϵ-closure** dari suatu state adalah himpunan semua state yang dapat dicapai dari state tersebut melalui ϵ-transitions saja
- **ϵ-closure** dari suatu set states adalah union dari ϵ-closure setiap state dalam set tersebut

2.6.2 Algoritma Subset Construction

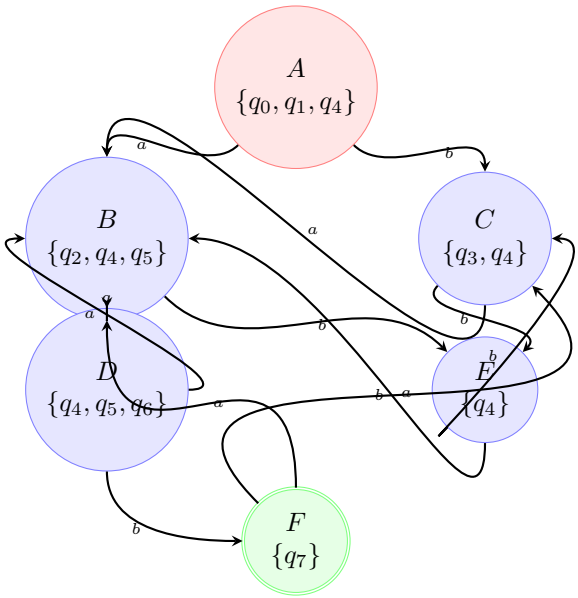
Algoritma subset construction bekerja sebagai berikut:

1. **Start State DFA**: ϵ-closure dari start state NFA
2. **Untuk setiap state DFA dan setiap input symbol**:
   (a) Hitung semua NFA states yang dapat dicapai dengan input symbol tersebut
   (b) Ambil ϵ-closure dari set states tersebut
   (c) Jika hasilnya belum ada sebagai state DFA, buat state baru
   (d) Tambahkan transisi dari state DFA saat ini ke state hasil
3. **Accept States DFA**: Setiap state DFA yang mengandung accept state NFA

2.6.3 Contoh: Konversi NFA `(a|b)*abb` ke DFA

Mari kita ikuti langkah-langkah subset construction. Gambar 2.10 menunjukkan proses konversi dan hasil DFA yang dihasilkan.

1. **Start State DFA**:
   - Mulai dari start state NFA, ambil ϵ-closure
   - Misalkan hasilnya adalah set $\{q_0, q_1, q_4\} \rightarrow$ ini menjadi state DFA A
2. **Transisi dari State A dengan input 'a'**:
   - Dari semua NFA states dalam A, cari yang dapat menerima 'a'
   - Ambil ϵ-closure dari hasilnya $\rightarrow$ misalkan $\{q_2, q_4, q_5\} \rightarrow$ state DFA B
3. **Transisi dari State A dengan input 'b'**:
   - Dari semua NFA states dalam A, cari yang dapat menerima 'b'
   - Ambil ϵ-closure dari hasilnya $\rightarrow$ misalkan $\{q_3, q_4\} \rightarrow$ state DFA C
4. **Lanjutkan untuk state B dan C** dengan cara yang sama
5. **Accept States**: State DFA yang mengandung accept state NFA (misalnya state yang mengandung q_7)



Gambar 2.10: DFA hasil subset construction dari NFA `(a|b)*abb`

Hasilnya adalah DFA yang ekuivalen dengan NFA asli, tetapi deterministik dan lebih efisien untuk simulasi. Setiap state DFA merepresentasikan set states NFA yang dapat dicapai dengan input tertentu.

2.7 Implementasi NFA dan DFA dalam C/C++

Untuk memahami konsep secara praktis, kita akan melihat struktur data dan algoritma dasar untuk mengimplementasikan NFA dan DFA.

2.7.1 Struktur Data NFA

Listing 2.1: Struktur Data untuk NFA

```
#include <vector>
#include <set>
#include <map>

struct NFATransition {
    int from_state;
    char symbol;  // '\0' untuk epsilon transition
    int to_state;
};

class NFA {
private:
    int num_states;
    int start_state;
    std::set<int> accept_states;
    std::vector<NFATransition> transitions;

public:
    // Konstruktor
    NFA(int states, int start);

    // Menambahkan transisi
    void addTransition(int from, char symbol, int to);

    // Menghitung epsilon closure
    std::set<int> epsilonClosure(const std::set<int>& states);

    // Simulasi NFA
    bool simulate(const std::string& input);
};
```

2.7.2 Struktur Data DFA

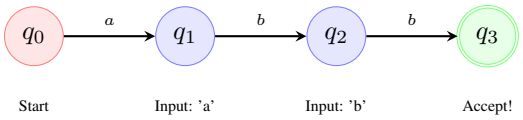
Listing 2.2: Struktur Data untuk DFA

```
class DFA {
private:
    int num_states;
    int start_state;
    std::set<int> accept_states;
    std::map<std::pair<int, char>, int> transition_table;

public:
```

```
    // Konstruktor
    DFA(int states, int start);

    // Menambahkan transisi (deterministic)
    void addTransition(int from, char symbol, int to);

    // Simulasi DFA (lebih sederhana dari NFA)
    bool simulate(const std::string& input);
};
```

2.7.3 Implementasi Simulasi DFA

Simulasi DFA lebih sederhana karena deterministik. Gambar 2.11 menunjukkan proses simulasi DFA untuk input string.



Gambar 2.11: Contoh simulasi DFA untuk input “abb”

Listing 2.3: Simulasi DFA

```
bool DFA::simulate(const std::string& input) {
    int current_state = start_state;

    for (char c : input) {
        auto key = std::make_pair(current_state, c);
        if (transition_table.find(key) == transition_table.end()) {
            return false;  // Tidak ada transisi
        }
        current_state = transition_table[key];
    }

    return accept_states.find(current_state) != accept_states.end();
}
```

2.7.4 Implementasi Subset Construction

Berikut adalah pseudocode untuk subset construction:

Listing 2.4: Subset Construction Algorithm

```
DFA NFA::toDFA() {
    DFA dfa(0, 0);
    std::map<std::set<int>, int> state_mapping;
    std::queue<std::set<int>> work_queue;

    // Start state DFA = epsilon closure dari start state NFA
    std::set<int> start_set = epsilonClosure({start_state});
```

```
    int dfa_start = dfa.addState();
    state_mapping[start_set] = dfa_start;
    work_queue.push(start_set);

    while (!work_queue.empty()) {
        std::set<int> nfa_states = work_queue.front();
        work_queue.pop();
        int dfa_state = state_mapping[nfa_states];

        // Untuk setiap input symbol
        for (char symbol : alphabet) {
            if (symbol == '\0') continue;  // Skip epsilon

            // Hitung move dengan symbol
            std::set<int> next_nfa_states;
            for (int state : nfa_states) {
                // Cari semua transisi dengan symbol ini
                for (auto& trans : transitions) {
                    if (trans.from_state == state &&
                        trans.symbol == symbol) {
                        next_nfa_states.insert(trans.to_state);
                    }
                }
            }

            // Ambil epsilon closure
            std::set<int> closure = epsilonClosure(next_nfa_states);

            if (!closure.empty()) {
                int next_dfa_state;
                if (state_mapping.find(closure) == state_mapping.end()) {
                    // State baru
                    next_dfa_state = dfa.addState();
                    state_mapping[closure] = next_dfa_state;
                    work_queue.push(closure);
                } else {
                    next_dfa_state = state_mapping[closure];
                }

                dfa.addTransition(dfa_state, symbol, next_dfa_state);
            }
        }
    }

    // Set accept states
    for (auto& pair : state_mapping) {
        for (int nfa_accept : accept_states) {
            if (pair.first.find(nfa_accept) != pair.first.end()) {
                dfa.setAcceptState(pair.second);
                break;
            }
        }
    }

```

```
62      return dfa;
63  }
```

2.8 Aplikasi dalam Lexical Analysis

Gambar 2.12 menunjukkan alur lengkap dari regular expression hingga token recognition dalam lexical analysis.

Gambar 2.12: Alur proses dari regular expression ke token scanner

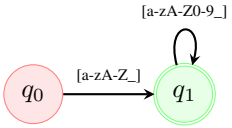
2.8.1 Token Recognition dengan DFA

Dalam lexical analysis, kita menggunakan DFA untuk mengenali token. Prosesnya:

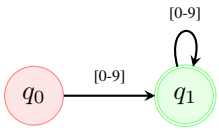
1. **Definisi Token**: Setiap jenis token didefinisikan dengan regular expression
2. **Kombinasi Regex**: Semua regex untuk token digabungkan dengan union
3. **Konversi ke DFA**: Regex gabungan dikonversi menjadi satu DFA
4. **Scanning**: Input dibaca karakter demi karakter, DFA dijalankan
5. **Longest Match**: Ambil token terpanjang yang cocok
6. **Token Classification**: Tentukan jenis token berdasarkan accept state yang dicapai

2.8.2 Contoh: Recognizer untuk Identifier dan Number

Mari kita buat recognizer sederhana untuk identifier dan number. Gambar 2.13 dan 2.14 menunjukkan DFA untuk masing-masing pattern.



Gambar 2.13: DFA untuk identifier: `[a-zA-Z_][a-zA-Z0-9_]*`



Gambar 2.14: DFA untuk number: `[0-9]+`

Listing 2.5: Token Recognizer menggunakan DFA

```
enum TokenType {
    IDENTIFIER,
    NUMBER,
    UNKNOWN
};

class TokenRecognizer {
private:
    DFA identifier_dfa;  // DFA untuk [a-zA-Z_][a-zA-Z0-9_]*
    DFA number_dfa;      // DFA untuk [0-9]+

public:
    TokenRecognizer() {
        // Konstruksi DFA untuk identifier dan number
        // (dari regex menggunakan Thompson + subset construction)
        buildIdentifierDFA();
        buildNumberDFA();
    }

    TokenType recognize(const std::string& lexeme) {
        if (identifier_dfa.simulate(lexeme)) {
            return IDENTIFIER;
        } else if (number_dfa.simulate(lexeme)) {
            return NUMBER;
        } else {
            return UNKNOWN;
        }
    }

private:
    void buildIdentifierDFA() {
        // Implementasi konstruksi DFA untuk identifier
        // Regex: [a-zA-Z_][a-zA-Z0-9_]*
    }

    void buildNumberDFA() {
        // Implementasi konstruksi DFA untuk number
        // Regex: [0-9]+
    }
};
```

Contoh penggunaan:

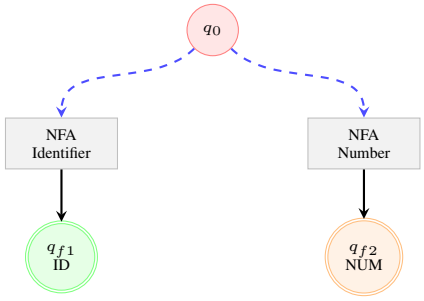
Listing 2.6: Contoh penggunaan TokenRecognizer

```
int main() {
```

```
    TokenRecognizer recognizer;

    std::vector<std::string> test_inputs = {
        "variable123",  // IDENTIFIER
        "42",           // NUMBER
        "_private",     // IDENTIFIER
        "3.14",         // UNKNOWN (belum support float)
        "123abc"        // UNKNOWN (mixed)
    };

    for (const auto& input : test_inputs) {
        TokenType type = recognizer.recognize(input);
        std::cout << input << " -> ";
        switch(type) {
            case IDENTIFIER: std::cout << "IDENTIFIER\n"; break;
            case NUMBER: std::cout << "NUMBER\n"; break;
            case UNKNOWN: std::cout << "UNKNOWN\n"; break;
        }
    }

    return 0;
}
```

2.8.3 Handling Multiple Tokens

Ketika kita memiliki multiple token types, kita perlu menggabungkan semua pattern menjadi satu DFA. Gambar 2.15 menunjukkan proses penggabungan NFA untuk multiple token types.



Gambar 2.15: Penggabungan NFA untuk multiple token types dengan labeling accept states

Proses handling multiple tokens:

1. Membuat NFA terpisah untuk setiap token type
2. Menggabungkan semua NFA dengan union, tetapi **label setiap accept state** dengan token type-nya
3. Konversi ke DFA (setiap DFA state mungkin mengandung multiple NFA accept states dengan label berbeda)

4. Saat scanning, jika mencapai accept state dengan multiple labels, gunakan **priority** atau **longest match**

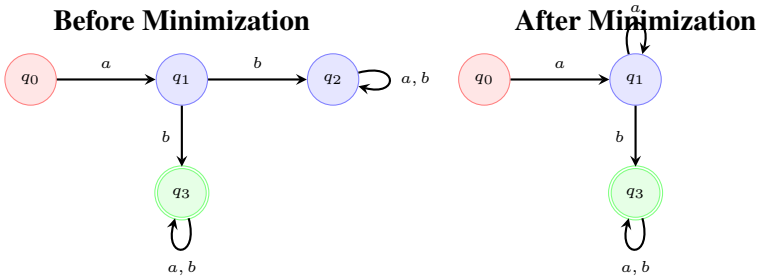
Contoh implementasi untuk multiple tokens:

Listing 2.7: Handling Multiple Token Types

```
class MultiTokenRecognizer {
private:
    DFA combined_dfa;
    std::map<int, TokenType> state_to_token;

public:
    TokenType recognize(const std::string& lexeme) {
        int final_state = combined_dfa.simulate(lexeme);
        if (final_state == -1) return UNKNOWN;

        // Jika state memiliki multiple labels, gunakan priority
        if (state_to_token.find(final_state) != state_to_token.end()) {
            return state_to_token[final_state];
        }
        return UNKNOWN;
    }

    // Longest match: scan sampai tidak bisa lanjut
    Token scanLongestMatch(std::istream& input) {
        std::string lexeme;
        int last_accept_state = -1;
        int last_accept_pos = -1;
        int current_state = start_state;
        int pos = 0;

        char c;
        while (input.get(c)) {
            lexeme += c;
            // Update state dengan input c
            // ...

            if (isAcceptState(current_state)) {
                last_accept_state = current_state;
                last_accept_pos = pos;
            }
            pos++;
        }

        // Kembalikan ke posisi terakhir yang accept
        input.seekg(last_accept_pos);
        return Token(state_to_token[last_accept_state],
                     lexeme.substr(0, last_accept_pos + 1));
    }
};
```

2.9 Optimasi: DFA Minimization

Setelah subset construction, DFA yang dihasilkan mungkin memiliki states yang redundan. Kita dapat meminimalkan DFA menggunakan algoritma seperti **Hopcroft's algorithm** atau **Moore's algorithm**. Gambar 2.16 menunjukkan contoh DFA sebelum dan sesudah minimisasi.



Gambar 2.16: Contoh DFA sebelum dan sesudah minimisasi (state q_2 dihapus karena equivalent dengan q_1)

2.9.1 Konsep State Equivalence

Dua states dalam DFA dikatakan **equivalent** jika:

- Keduanya accept states ATAU keduanya bukan accept states
- Untuk setiap input symbol, transisi dari kedua states menuju ke states yang equivalent

2.9.2 Algoritma Minimization

Algoritma minimisasi bekerja dengan:

1. Partisi states menjadi dua grup: accept states dan non-accept states
2. Untuk setiap partisi, periksa apakah states dalam partisi tersebut equivalent
3. Jika tidak equivalent, pisahkan menjadi partisi baru
4. Ulangi sampai tidak ada partisi yang dapat dipisah lagi
5. Merge states dalam partisi yang sama

DFA yang sudah diminimalkan memiliki jumlah states minimum yang masih ekuivalen dengan DFA asli.

2.10 Contoh Praktis: Implementasi Lengkap

Sebagai contoh praktis, berikut adalah implementasi lengkap DFA sederhana untuk mengenali identifier dan number:

Listing 2.8: Implementasi Lengkap DFA untuk Identifier dan Number

```
#include <iostream>
#include <string>
#include <map>
#include <set>

class SimpleDFA {
private:
    int start_state;
    std::set<int> accept_states;
    std::map<std::pair<int, char>, int> transitions;

    bool isLetter(char c) {
        return (c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z') || c == '
↪ _';
    }

    bool isDigit(char c) {
        return c >= '0' && c <= '9';
    }

    bool isAlphanumeric(char c) {
        return isLetter(c) || isDigit(c);
    }

public:
    SimpleDFA() {
        // DFA untuk identifier: [a-zA-Z_][a-zA-Z0-9_]*
        start_state = 0;
        accept_states.insert(1);

        // State 0 -> State 1 dengan letter atau underscore
        for (char c = 'a'; c <= 'z'; c++) {
            transitions[{0, c}] = 1;
            transitions[{1, c}] = 1;  // Loop di state 1
        }
        for (char c = 'A'; c <= 'Z'; c++) {
            transitions[{0, c}] = 1;
            transitions[{1, c}] = 1;
        }
        transitions[{0, '_'}] = 1;
        transitions[{1, '_'}] = 1;

        // State 1 -> State 1 dengan digit
        for (char c = '0'; c <= '9'; c++) {
            transitions[{1, c}] = 1;
        }
    }

```

```
48      bool simulate(const std::string& input) {
49          int current_state = start_state;
50
51          for (char c : input) {
52              auto key = std::make_pair(current_state, c);
53              if (transitions.find(key) == transitions.end()) {
54                  return false;
55              }
56              current_state = transitions[key];
57          }
58
59          return accept_states.find(current_state) != accept_states.end();
60      }
61  };
62
63  int main() {
64      SimpleDFA dfa;
65
66      std::vector<std::string> test_cases = {
67          "variable",
68          "var123",
69          "_private",
70          "123var",       // Invalid (dimulai dengan digit)
71          "var_name",
72          "VarName123"
73      };
74
75      std::cout << "Testing DFA untuk Identifier:\n";
76      std::cout << "============================\n";
77      for (const auto& test : test_cases) {
78          bool result = dfa.simulate(test);
79          std::cout << "\"" << test << "\" -> "
80                    << (result ? "ACCEPT" : "REJECT") << "\n";
81      }
82
83      return 0;
84  }
```

Output program di atas:

```
Testing DFA untuk Identifier:
============================
"variable" -> ACCEPT
"var123" -> ACCEPT
"_private" -> ACCEPT
"123var" -> REJECT
"var_name" -> ACCEPT
"VarName123" -> ACCEPT
```

2.11 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Regular expression adalah notasi formal untuk mendeskripsikan pola token
2. Finite automata (NFA dan DFA) adalah model matematika untuk mengenali regular language
3. Algoritma Thompson mengkonversi regular expression menjadi ϵ-NFA
4. Subset construction mengkonversi NFA menjadi DFA yang ekuivalen
5. DFA lebih efisien untuk simulasi dan digunakan dalam lexical analysis
6. Implementasi praktis memerlukan struktur data yang tepat dan algoritma yang efisien

Pemahaman tentang regular expression dan finite automata ini menjadi dasar penting untuk implementasi lexical analyzer yang akan dipelajari dalam bab-bab selanjutnya.

2.12 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang regular expression dan finite automata, mahasiswa disarankan membaca:

- **Dragon Book**: Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* [2] - Bab 3: Lexical Analysis
- **Engineering a Compiler**: Cooper & Torczon (2011) [3] - Bab 2: Scanners
- **Aoyama Gakuin University**: Lecture notes tentang lexical analysis dan finite automata [1]
- **OpenGenus**: Tutorial tentang membangun lexer [4]
- **GeeksforGeeks**: Artikel tentang regular expression to NFA dan NFA to DFA conversion

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

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

[4] OpenGenus. *Build Lexer*. Tutorial on hand-written lexers. 2024. URL: https://iq.opengenus.org/build-lexer/.