IF2124 Teori Bahasa Formal dan Otomata

Milestone 1
Lexical Analysis

Laporan Tugas Besar

Disusun untuk memenuhi tugas besar mata kuliah IF 2124 Teori Bahasa Formal dan Otomata
pada Semester I Tahun Akademik 2025/2026

Oleh

Raka Daffa Iftikhaar	13523018
Aliya Husna Fayyaza	13523062
Ahsan Malik Al Farisi	13523074
Bevinda Vivian	13523120

Kelompok Ahsan Et Al (NTB)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA - KOMPUTASI
PROGRAM STUDI TEKNIK INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

2025

Daftar Isi

BAB I

Landasan Teori

1.1. Compiler dan Fase-Fase Kompilasi

Compiler adalah sebuah program komputer yang menerjemahkan *source code* yang ditulis dalam bahasa pemrograman tingkat tinggi menjadi bahasa mesin atau *object code* yang dapat dieksekusi langsung oleh komputer. Proses kompilasi merupakan serangkaian transformasi yang kompleks dan terstruktur, yang secara umum dibagi menjadi enam fase utama.

1) Fase pertama adalah analisis leksikal (*lexical analysis*), di mana kode sumber dibaca karakter demi karakter dan dikelompokkan menjadi unit-unit leksikal yang disebut token.
2) Fase kedua adalah analisis sintaks (*syntax analysis atau parsing*), yang memeriksa apakah urutan token membentuk struktur yang sesuai dengan tata bahasa (*grammar*) bahasa pemrograman.
3) Fase ketiga adalah analisis semantik (*semantic analysis*), yang memeriksa konsistensi makna dari program, seperti pengecekan tipe data dan deklarasi variabel.
4) Fase keempat adalah pembangkitan kode antara (*intermediate code generation*), yang menghasilkan representasi abstrak dari program yang lebih mudah dioptimasi.
5) Fase kelima adalah optimisasi kode (*code optimization*), yang bertujuan memperbaiki kode antara agar lebih efisien tanpa mengubah fungsionalitas program.
6) Fase terakhir adalah pembangkitan kode *(code generation*), yang menghasilkan kode mesin atau assembly yang dapat dieksekusi.

Pada Milestone 1 ini, fokus utama adalah pada fase pertama, yaitu analisis leksikal, yang merupakan fondasi dari seluruh proses kompilasi. Tanpa analisis leksikal yang akurat, fase-fase selanjutnya tidak dapat berjalan dengan baik karena input yang diterima tidak terstruktur dengan benar.

1.2. Analisis Leksikal dan Peran Lexer

Analisis leksikal adalah tahap awal dalam proses kompilasi yang bertugas membaca aliran karakter dari kode sumber dan mengelompokkan karakter-karakter tersebut menjadi unit-unit bermakna yang disebut token atau lexeme. Komponen perangkat lunak yang melakukan tugas ini disebut *lexical analyzer*, lexer, atau

scanner. Lexer berperan sebagai jembatan antara representasi tekstual dari program dan representasi struktural yang lebih abstrak yang digunakan oleh parser.

Fungsi utama lexer meliputi beberapa aspek penting. Pertama, lexer membaca karakter demi karakter dari aliran input kode sumber secara berurutan. Kedua, lexer mengenali pola-pola karakter yang membentuk token berdasarkan aturan leksikal bahasa pemrograman. Ketiga, lexer mengabaikan elemen-elemen yang tidak relevan untuk analisis sintaks, seperti whitespace (spasi, tab, newline) dan komentar. Keempat, lexer menghasilkan urutan token yang akan menjadi input bagi parser pada fase analisis sintaks. Kelima, lexer melaporkan kesalahan leksikal jika menemukan karakter atau urutan karakter yang tidak valid menurut aturan bahasa pemrograman.

Token sendiri merupakan pasangan yang terdiri dari dua komponen. Komponen pertama adalah token type atau token name, yang merupakan kategori atau klasifikasi dari lexeme tersebut, seperti KEYWORD, IDENTIFIER, NUMBER, OPERATOR, atau PUNCTUATION. Komponen kedua adalah token value atau lexeme, yang merupakan string aktual atau sekuens karakter dari kode sumber yang membentuk token tersebut.

1.3. Teori Bahasa Formal dan Hierarki Chomsky

Untuk memahami bagaimana lexer bekerja, kita perlu memahami konsep bahasa formal. Bahasa formal adalah himpunan string yang dibentuk dari alfabet tertentu menurut aturan-aturan yang terdefinisi dengan jelas. Dalam konteks kompilasi, bahasa formal digunakan untuk mendefinisikan struktur leksikal dan sintaksis dari bahasa pemrograman.

Noam Chomsky mengklasifikasikan bahasa formal menjadi empat tingkatan dalam Hierarki Chomsky, dari yang paling umum hingga yang paling terbatas: Type-0 (*Unrestricted Grammar*), Type-1 (*Context-Sensitive Grammar*), Type-2 (*Context-Free Grammar*), dan Type-3 (*Regular Grammar*). Setiap tingkat memiliki kekuatan ekspresif yang berbeda dan dapat dikenali oleh mesin abstrak yang berbeda pula.

Untuk analisis leksikal, kita menggunakan Regular Grammar (Type-3), yang merupakan tingkat paling sederhana namun cukup powerful untuk mendeskripsikan pola-pola token dalam bahasa pemrograman. Regular Grammar dapat diekspresikan dalam tiga bentuk ekuivalen: Regular Expression (regex), Finite Automaton (FA), dan Regular Grammar dalam bentuk produksi. Ketiga representasi ini memiliki kekuatan ekspresif yang sama dan dapat dikonversi satu sama lain.

Regular Expression adalah notasi matematis yang ringkas untuk mendeskripsikan pola string. Contohnya, regex `[a-zA-Z][a-zA-Z0-9]*` mendeskripsikan identifier yang dimulai dengan huruf dan diikuti oleh nol atau lebih

huruf atau digit. Regular Grammar, di sisi lain, menggunakan aturan produksi untuk mendefinisikan bahasa. Finite Automaton adalah mesin abstrak yang dapat mengenali bahasa reguler dengan membaca input dan melakukan transisi antar state.

1.4. Finite Automaton

Finite Automaton (FA) adalah model komputasi abstrak yang terdiri dari sejumlah terbatas state dan dapat melakukan transisi antar state berdasarkan input yang dibaca. FA dibagi menjadi dua jenis utama, yaitu Nondeterministic Finite Automaton (NFA) dan Deterministic Finite Automaton (DFA).

Nondeterministic Finite Automaton (NFA) memiliki karakteristik di mana dari satu state dengan input tertentu, automaton dapat melakukan transisi ke beberapa state sekaligus, atau bahkan melakukan transisi tanpa membaca input (ε-transition). NFA lebih mudah dirancang dari regex karena lebih fleksibel, namun lebih sulit diimplementasikan dalam program karena sifat nondeterministiknya yang memerlukan backtracking atau eksplorasi cabang secara paralel.

Deterministic Finite Automaton, sebaliknya, memiliki karakteristik di mana dari setiap state dengan input tertentu, automaton hanya dapat melakukan transisi ke tepat satu state. DFA lebih mudah diimplementasikan dalam program karena tidak memerlukan backtracking dan dapat dijalankan dalam waktu linear terhadap panjang input. Meskipun DFA mungkin memiliki lebih banyak state daripada NFA ekuivalen, efisiensi eksekusinya membuat DFA menjadi pilihan yang lebih baik untuk implementasi lexer

Secara formal, DFA didefinisikan sebagai 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, di mana:

Q adalah himpunan terbatas dari states (keadaan)
Σ adalah himpunan terbatas dari alphabet atau simbol input
δ adalah transition function atau fungsi transisi yang memetakan pasangan (state, input) ke state berikutnya: $\delta : Q \times \Sigma \rightarrow Q$
$q_0 \in Q$ adalah start state atau state awal
$F \subseteq Q$ adalah himpunan final states atau accept states yang menandakan bahwa input yang dibaca membentuk string yang valid

Cara kerja DFA dapat dijelaskan melalui algoritma berikut:

a) Inisialisasi, diulai dari start state q_0
b) Baca input, ambil simbol input berikutnya dari aliran input
c) Transisi, gunakan fungsi transisi δ untuk berpindah ke state berikutnya berdasarkan state saat ini dan simbol input yang dibaca
d) Iterasi, ulangi langkah 2-3 hingga seluruh input habis dibaca

e) Akseptasi, jika setelah input habis, automaton berada di salah satu final state (state ∈ F), maka input diterima (*accepted*); jika tidak, input ditolak (*rejected*)

Jika di tengah proses tidak ada transisi yang valid dari state saat ini dengan input yang dibaca (fungsi transisi tidak terdefinisi), maka automaton berhenti dan input ditolak.

1.5. Keywords dan Reserved Words dalam Pascal

Dalam bahasa pemrograman Pascal, terdapat sejumlah kata yang memiliki makna khusus dalam bahasa tersebut dan tidak dapat digunakan sebagai identifier (nama variabel, fungsi, atau prosedur). Kata-kata ini disebut keywords atau reserved words. Keywords merupakan bagian integral dari sintaks bahasa dan memiliki fungsi spesifik dalam struktur program.

Keywords dalam Pascal dapat dikelompokkan berdasarkan fungsinya, yaitu:

a) Control Flow Keywords
   Kata kunci yang mengatur alur eksekusi program, meliputi if, then, else, while, do, for, to, downto, repeat , dan until.
b) Program Structure Keywords
   Kata kunci yang mendefinisikan struktur program, meliputi program, begin, end, procedure, dan function.
c) Data Type Keywords
   Kata kunci yang mendeklarasikan tipe data, meliputi integer, real, boolean, char, string, array, dan of.
d) Declaration Keywords
   Kata kunci untuk deklarasi, meliputi var, const, dan type.
e) Logical Operator Keywords
   Operator logika yang ditulis sebagai kata, meliputi and, or, dan not.
f) Arithmetic Operator Keywords
   Operator aritmatika yang ditulis sebagai kata, meliputi div, dan mod.

Dalam implementasi lexer, penanganan keywords dilakukan dengan pendekatan khusus. Secara leksikal, keywords memiliki struktur yang sama dengan identifier. Oleh karena itu, lexer pertama-tama mengenali token sebagai identifier menggunakan DFA, kemudian melakukan pengecekan apakah string yang dikenali termasuk dalam daftar keywords. Jika ya, tipe token diubah dari IDENTIFIER menjadi KEYWORD. Pendekatan ini lebih efisien daripada membuat state terpisah dalam DFA untuk setiap keyword.

1.6. Representasi dan Pemrosesan Token

Setelah lexer mengenali pola karakter yang membentuk token, informasi token tersebut perlu disimpan dalam struktur data yang terorganisir. Setiap token biasanya direpresentasikan sebagai struktur data yang berisi beberapa atribut penting.

Token Type adalah kategori atau jenis dari token, yang menentukan peran leksikal token dalam bahasa pemrograman. Contoh tipe token meliputi KEYWORD, IDENTIFIER, INTEGER, REAL, STRING, OPERATOR, ASSIGNMENT, PUNCTUATION, dan sebagainya. Tipe token ini akan digunakan oleh parser pada fase analisis sintaks untuk memverifikasi struktur program. Token Value atau Lexeme adalah string aktual dari kode sumber yang membentuk token. Nilai ini penting terutama untuk nama variabel yang akan digunakan dalam symbol table, nilai yang akan digunakan dalam evaluasi ekspresi, dan teks yang akan disimpan dalam memori.

Line Number adalah nomor baris di mana token ditemukan dalam kode sumber. Informasi ini sangat penting untuk pelaporan error yang informatif, sehingga programmer dapat dengan cepat menemukan lokasi masalah dalam kode mereka. Column Number adalah nomor kolom atau posisi karakter dalam baris di mana token dimulai. Bersama dengan line number, informasi ini memberikan lokasi yang presisi untuk setiap token dan sangat membantu dalam debugging.

Sebagai contoh, untuk kode var x bertipe integer di baris 1, token yang dihasilkan akan memiliki struktur sebagai berikut:

Token 1: type=KEYWORD, value="var", line=1, column=1

Token 2: type=IDENTIFIER, value="x", line=1, column=5

Token 3: type=COLON, value=":", line=1, column=7

Token 4: type=KEYWORD, value="integer", line=1, column=9

Token 5: type=SEMICOLON, value=";", line=1, column=16

Analisis leksikal merupakan fondasi dari proses kompilasi yang memanfaatkan teori bahasa formal, khususnya regular language dan finite automaton. Dengan memahami konsep-konsep seperti DFA, character classes, dan token representation, dapat dibangun lexer yang robust, efisien, dan mudah dipelihara. Implementasi lexer yang baik tidak hanya menghasilkan token yang benar, tetapi juga memberikan pesan error yang informatif dan memiliki performa yang optimal. Teori yang diaplikasikan dalam bentuk program penganalisis leksikal dengan menggunakan bahasa Pascal-S. Selanjutnya akan dibahas terkait perancangan dan implementasi dari *lexical analysis* yang kelompok kami kerjakan.

BAB 2

Perancangan & Implementasi

2.1. Deterministic Finite Automata

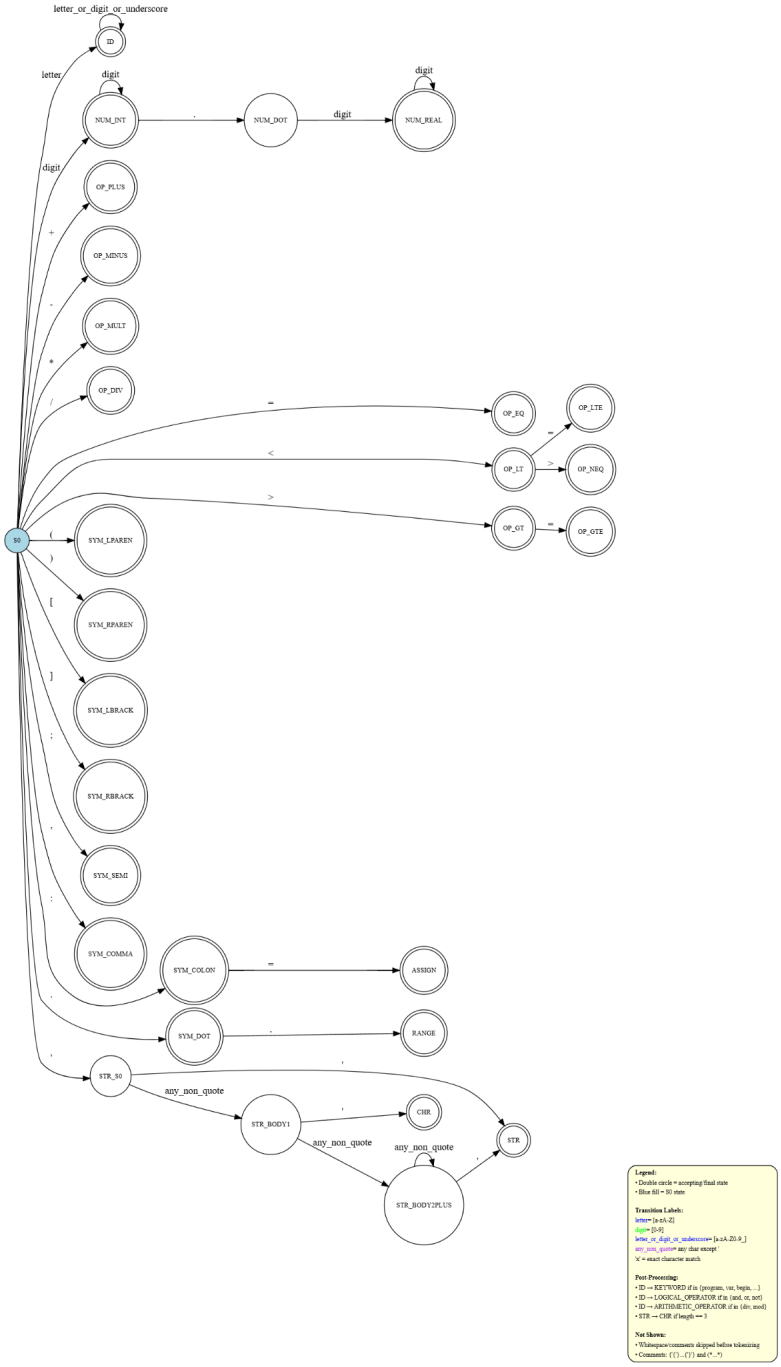
Diagram di bawah merupakan DFA dari lexical analysis untuk tokenisasi input program. Adapun alur state dari DFA, seperti berikut.

a. State Awal (S0)
   State S0 adalah titik awal tokenisasi. Dari state ini, automata akan berpindah ke state lain berdasarkan karakter pertama yang dibaca.
b. Identifier (ID)
   Identifier terbentuk dari pola [a-zA-Z][a-zA-Z0-9]*. Pada tahap *post-processing,* jika terdapat kata yang cocok dengan daftar kunci Pascal-S (program, begin, dll) maka akan diklasifikasikan sebagai KEYWORD.
c. Number (NUM_INT & NUM REAL)
   Number dapat terbentuk dari karakter angka saja untuk NUM_INt dan jika mengandung titik diikuti digit makan akan sebagai NUM_REAL.
d. Assignment dan Range Operator
   Karakter ‘:=’ akan di tokenisasi menjadi ASSIGN_OPERATOR, ‘..’ menjadi RANGE_OPERATOR, ‘:’ dan ‘.’ menjadi SYM_COLON dan SYM_DOT. Namun terdapat limitasi dalam pengklasifikasian RANGE_OPERATOR, karena DFA tidak memiliki memori sehingga tidak mungkin untuk mengklasifikasikan NUMBER, RANGE_OPERATOR, NUMBER pada diagram. Namun, hal ini dapat ditangani di dalam program, karena program memiliki memori yang dapat mengecek dan mentokenisasi ulang dengan benar dari RANGE_OPERATOR.
e. String dan Character Literal
   Jika kode program diapit oleh (‘) tanda petik maka ini akan menjadi CHAR_LITERAL, namun ketika panjangnya melebihi satu maka akan menjadi STRING_LITERAL.
f. Operator Aritmetika
   Automata mengenali berbagai operator tunggal seperti tanda ‘+’, ‘-’, ‘*’, dan ‘/’. Selain itu terdapat kata kunci div dan mod yang akan diubah menjadi operator aritmetika pada *post-processing* karena dikenali oleh token IDENTIFIER.
g. Operator Relasional
   Terdapat berbagai karakter relasional yang dikenali seperti ‘=’, ‘<’, ‘>’. Selain itu terdapat juga berbagai kombinasi dari token relasional seperti ‘<=’, ‘>=’, dan ‘<>’.
h. Delimiter dan Simbol

Karakter seperti ‘(‘, ‘)’, ‘[‘, ‘]’, ‘;’, dan ‘,’ akan langsung menuju state finalnya masing-masing (SYM_LPAREN, SYM_RPAREN, dll) tanpa state perantara.

i. Whitespace dan Komentar

Whitespace, spasi, tab, newline, dan juga komentar tidak akan ditokenisasi, sehingga tidak termasuk di dalam diagram.



Gambar 1. Diagram DFA Lexical Analysis

2.2. Arsitektur Program

Struktur direktori tugas besar ini adalah sebagai berikut:

```
NTB-Tubes-IF2224/
├── src/
│   ├── main.cpp
│   ├── lexer.cpp
│   ├── dfa_loader.cpp
│   └── char_classes.cpp
├── include/
│   ├── lexer.hpp
│   ├── dfa_loader.hpp
│   ├── char_classes.hpp
│   ├── token.hpp
│   └── nlohmann/
│       └── json.hpp
├── dfa/
│   └── dfa.json
├── test/
│   └── milestone-1/
│       └── input/
│       └── output/
├── Makefile
└── README.md
```

a. DFA Loader Module (dfa_loader.hpp dan dfa_loader.cpp)
   Bertanggung jawab untuk membaca dan mem-parse file DFA JSON, kemudian membangun struktur data internal untuk DFA yang dapat digunakan oleh lexer.

b. Character Classification Module (char_classes.hpp dan char_classes.cpp)
   Menyediakan fungsi untuk mengklasifikasikan karakter input ke dalam character classes seperti letter, digit, whitespace, operator, dan lainnya. Module ini juga menyimpan daftar keywords dan reserved words.

c. Token Module (token.hpp)
   Mendefinisikan struktur data Token yang menyimpan informasi tentang tipe token, nilai lexeme, nomor baris, dan nomor kolom.

d. Lexer Module (lexer.hpp dan lexer.cpp)
Implementasi algoritma tokenisasi menggunakan DFA. Lexer membaca karakter dari input, melakukan transisi state, dan menghasilkan token.

e. Main Program (main.cpp)
Entry point program yang menangani argument parsing, loading DFA, membaca file input, memanggil lexer, dan menampilkan hasil tokenisasi.

f. JSON Parser Library (nlohmann/json.hpp)
Library digunakan untuk parsing file JSON

2.3. Implementasi DFA Loader

DFA Loader bertugas membaca definisi DFA dari file JSON dan membangun struktur data yang dapat digunakan oleh lexer.

2.3.1 Struktur Data DFA

Class DFA didefinisikan dalam dfa_loader.hpp dengan komponen-komponen berikut:

```
class DFA {
public:
    DFA() = default;
    DFA(std::string start,
        std::unordered_set<std::string> finals,
        std::unordered_map<std::pair<std::string,std::string>,
                           std::string, PairHash> trans);

    std::string next_state(const std::string& state,
                           const std::string& inp) const;
    const std::string& get_start() const;
    bool is_final(const std::string& s) const;

private:
    std::string start_state;
    std::unordered_set<std::string> final_states;
    std::unordered_map<std::pair<std::string, std::string>,
                       std::string, PairHash> transitions;
};
```

start_state: String yang menyimpan nama state awal DFA.
final_states: Unordered set yang berisi nama-nama state final, memungkinkan pengecekan keanggotaan dalam waktu O(1).
transitions: Hash map yang memetakan pasangan (state, input_class) ke next_state. Penggunaan std::pair sebagai key memerlukan custom hash function PairHash.

PairHash diimplementasikan untuk memungkinkan std::pair<std::string, std::string> digunakan sebagai key dalam unordered_map

```
struct PairHash {
    template <class T1, class T2>
    std::size_t operator()(const std::pair<T1,T2>& p) const noexcept {
        std::hash<T1> h1;
        std::hash<T2> h2;
        return h1(p.first) ^ (h2(p.second) + 0x9e3779b97f4a7c15ULL +
                              (h1(p.first) << 6) + (h1(p.first) >> 2));
    }
};
```

Hash function ini mengkombinasikan hash dari kedua elemen pair dengan teknik bit shifting dan XOR untuk menghasilkan distribusi hash yang baik.

2.3.2 Fungsi next_state

Fungsi next_state adalah inti dari operasi DFA, mengimplementasikan fungsi transisi δ

```
std::string DFA::next_state(const std::string& state,
                            const std::string& inp) const {
    auto it = transitions.find({state, inp});
    if (it != transitions.end())
        return it->second;
    return std::string();
}
```

Fungsi ini mencari pasangan (state, inp) dalam transition table. Jika transisi ditemukan, mengembalikan next_state; jika tidak, mengembalikan string kosong yang menandakan tidak ada transisi valid (dead end).

2.3.3 Loading dari JSON

Fungsi load_dfa_json dalam dfa_loader.cpp membaca file JSON dan membangun object DFA:

```
DFA load_dfa_json(const std::string& path) {
    std::ifstream f(path);
    if (!f)
        throw std::runtime_error("Cannot open DFA file: " + path);

    json j;
    f >> j;

    std::string start = j["start_state"];
    std::unordered_set<std::string> finals;
    for (const auto& fs : j["final_states"])
        finals.insert(fs);

    std::unordered_map<std::pair<std::string,std::string>,
                       std::string, PairHash> trans;

    for (auto& [state, edges] : j["transitions"].items()) {
        for (auto& [inp, next] : edges.items()) {
            trans[{state, inp}] = next;
```

```
            }
        }

        return DFA(start, finals, trans);
    }
```

Proses loading meliputi

- Membuka dan membaca file JSON menggunakan library nlohmann/json
- Ekstraksi start_state dari field "start_state"
- Iterasi array "final_states" dan memasukkan setiap state ke set finals
- Iterasi nested object "transitions" untuk membangun transition table, di mana level pertama adalah current state dan level kedua adalah input class yang memetakan ke next state

2.4. Implementasi Character Classification

Character Classification Module bertanggung jawab untuk memetakan setiap karakter input ke satu atau lebih character classes. Ini penting karena DFA menggunakan character classes, bukan karakter individual, dalam transisinya.

2.4.1 Definisi Keywords dan Reserved Words

File char_classes.cpp mendefinisikan tiga set global untuk keywords

```
const std::unordered_set<std::string> KEYWORDS = {
    "program", "var", "begin", "end", "if", "then", "else",
    "while", "do", "for", "to", "downto", "integer", "real",
    "boolean", "char", "array", "of", "procedure", "function",
    "const", "type"
};

const std::unordered_set<std::string> LOGICAL_WORDS = {
    "and", "or", "not"
};

const std::unordered_set<std::string> ARITH_WORDS = {
    "div", "mod"
};
```

Set-set ini digunakan oleh lexer untuk membedakan identifier biasa dari keywords setelah token dikenali.

2.4.2 Fungsi classify_char

Fungsi classify_char memetakan satu karakter ke set character classes

```
std::unordered_set<std::string> classify_char(char ch) {
    std::unordered_set<std::string> classes;

    // Newline: special handling for line tracking
```

```
    if (ch == '\n') {
        classes.insert("newline");
        return classes;
    }

    // Whitespace: space, tab, carriage return
    if (ch == ' ' || ch == '\t' || ch == '\r') {
        classes.insert("whitespace");
        return classes;
    }

    // Letter: alphabetic characters
    if (std::isalpha(static_cast<unsigned char>(ch))) {
        classes.insert("letter");
    }

    // Digit: numeric characters
    if (std::isdigit(static_cast<unsigned char>(ch))) {
        classes.insert("digit");
    }

    // Underscore: identifier component
    if (ch == '_') {
        classes.insert("underscore");
    }

    // Quote: for string/char literals
    if (ch == '\'') {
        classes.insert("quote");
    }

    // Single character class: exact match for operators/symbols
    std::string single_char(1, ch);
    classes.insert(single_char);

    // "any" class: for string content (not newline, not quote)
    if (ch != '\n' && ch != '\'') {
        classes.insert("any");
    }

    return classes;
}
```

Strategi klasifikasi yang kami terapkan adalah sebagai berikut:

- Prioritas khusus: Newline dan whitespace langsung return dengan satu class saja, karena mereka diabaikan oleh lexer.
- Multiple classes: Karakter dapat masuk beberapa class sekaligus. Contoh: 'a' masuk class "letter" dan juga "a" (single char), serta "any".
- Single character class: Setiap karakter juga dimasukkan sebagai class dengan nama karakter itu sendiri (untuk operator seperti '+', '-', dll).
- "any" class: Digunakan untuk matching karakter apapun dalam string literal, kecuali newline dan quote.

- Penggunaan static_cast<unsigned char>(ch) diperlukan karena std::isalpha dan std::isdigit mensyaratkan nilai non-negative atau EOF.

2.5. Implementasi Lexer

Lexer adalah komponen inti yang mengimplementasikan algoritma tokenisasi menggunakan DFA. Class Lexer didefinisikan dalam lexer.hpp dan diimplementasikan dalam lexer.cpp.

2.5.1 Struktur Class Lexer

```
class Lexer {
public:
    Lexer(const DFA& dfa, std::string source)
        : dfa(dfa), src(std::move(source)), i(0), line(1), col(1) {}

    std::vector<Token> tokenize();

private:
    const DFA& dfa;
    std::string src;
    size_t i;      // current position in source
    int line, col; // current line and column

    char peek(int k = 0) const;
    char advance();
    void skip_ws_comment();
    std::string map_state_to_type(const std::string& state,
                                  const std::string& lex) const;
};
```

2.5.2 Fungsi Helper: peek dan advance

peek() memungkinkan lexer melihat karakter ke depan tanpa mengkonsumsinya

```
char Lexer::peek(int k) const {
    size_t idx = i + (k < 0 ? 0 : static_cast<size_t>(k));
    if (idx < src.size())
        return src[idx];
    return '\0'; // EOF marker
}
```

advance() mengkonsumsi karakter saat ini dan memajukan posisi

```
char Lexer::advance() {
    if (i >= src.size())
        return '\0';

    char ch = src[i++];
    if (ch == '\n') {
        line++;
        col = 1;
    } else {
        col++;
    }
    return ch;
```

```
}
```

2.5.3 Skip Whitespace dan Comment

Fungsi skip_ws_comment() bertanggung jawab untuk melewati karakter-karakter yang tidak relevan

```
void Lexer::skip_ws_comment() {
    while (true) {
        char ch = peek();

        // Skip whitespace
        if (ch == ' ' || ch == '\t' || ch == '\r' || ch == '\n') {
            advance();
            continue;
        }

        // Skip { ... } comment
        if (ch == '{') {
            advance();
            while (peek() != '}' && peek() != '\0')
                advance();
            if (peek() == '}')
                advance();
            continue;
        }

        // Skip (* ... *) comment
        if (ch == '(' && peek(1) == '*') {
            advance(); // (
            advance(); // *
            while (true) {
                if (peek() == '\0')
                    break;
                if (peek() == '*' && peek(1) == ')') {
                    advance(); // *
                    advance(); // )
                    break;
                }
                advance();
            }
            continue;
        }

        break; // no more whitespace or comment
    }
}
```

Fungsi ini menangani tiga kasus yaitu

- Whitespace: Karakter spasi, tab, carriage return, dan newline diabaikan
- Curly brace comment { ... }: Skip hingga menemukan }
- Parenthesis-asterisk comment (* ... *): Skip hingga menemukan *)

Loop akan terus berjalan hingga tidak ada lagi whitespace atau comment yang ditemukan.

2.5.4 Algoritma Tokenisasi Utama

Fungsi tokenize() mengimplementasikan algoritma utama lexer

```
std::vector<Token> Lexer::tokenize() {
    std::vector<Token> tokens;
    while (i < src.size()) {
        skip_ws_comment();

        if (i >= src.size())
            break;
        // Save starting position
        int start_line = line;
        int start_col = col;
        // DFA simulation
        std::string current_state = dfa.get_start();
        std::string buffer;
        std::string last_final_state;
        size_t last_final_pos = i;

        while (i < src.size()) {
            char ch = peek();
            auto classes = classify_char(ch);

            bool found_transition = false;
            std::string next;

            // Try each class for this character
            for (const auto& cls : classes) {
                next = dfa.next_state(current_state, cls);
                if (!next.empty()) {
                    found_transition = true;
                    break;
                }
            }

            if (!found_transition)
                break; // dead end

            // Valid transition found
            advance();
            buffer += ch;
            current_state = next;

            // Track last final state (longest match)
            if (dfa.is_final(current_state)) {
                last_final_state = current_state;
                last_final_pos = i;
            }
        }

        // Check if we have a valid token
        if (!last_final_state.empty()) {
            // Backtrack to last final position
            while (i > last_final_pos) {
                i--;
                if (src[i] == '\n') {
                    line--;
                    // Recompute col (simplified)
                    col = 1;
                } else {
                    col--;
                }
```

```
            }

            // Trim buffer to match last final position
                size_t valid_len = last_final_pos - (last_final_pos -
buffer.size());
            buffer = buffer.substr(0, valid_len);

            // Create token
                std::string type = map_state_to_type(last_final_state,
buffer);
            tokens.push_back({type, buffer, start_line, start_col});
        } else {
            // Lexical error
            throw LexerError("Unexpected character at line " +
                             std::to_string(line) + ", column " +
                             std::to_string(col));
        }
    }

    return tokens;
}
```

Alur algoritma yang diterapkan adalah sebagai berikut:

- Loop, iterasi selama masih ada karakter yang belum diproses
- Skip whitespace dan comment sebelum memproses token baru
- Inisialisasi dengan menyimpan posisi awal (line, col), set current_state ke start state DFA, siapkan buffer kosong
- DFA simulation loop, dilakukan dengan peek karakter berikutnya lalu klasifikasikan karakter ke character classes, setelah itu dicoba setiap class untuk mencari transisi valid. Jika transisi ditemukan lakukan advance, append ke buffer, update state. Sementara itu juga dilakukan track last final state untuk longest match
- Token generation, jika last_final_state tidak kosong backtrack ke posisi final state terakhir, trim buffer, generate token, jika tidak ada final state tercapai throw lexical error. Algoritma menggunakan strategi longest match dengan tracking last_final_state dan last_final_pos, sehingga token yang dihasilkan adalah yang terpanjang yang valid.

2.5.5 Mapping State ke Token Type

Fungsi map_state_to_type memetakan final state DFA ke tipe token:

```
std::string Lexer::map_state_to_type(const std::string& state,
                                     const std::string& lex) const {
    // Identifier or Keyword
    if (state == "ID") {
        std::string lower_lex = lex;
        std::transform(lower_lex.begin(), lower_lex.end(),
                       lower_lex.begin(), ::tolower);

        if (KEYWORDS.find(lower_lex) != KEYWORDS.end())
            return "KEYWORD";
        if (LOGICAL_WORDS.find(lower_lex) != LOGICAL_WORDS.end())
```

```
            return "LOGICAL_OP";
        if (ARITH_WORDS.find(lower_lex) != ARITH_WORDS.end())
            return "ARITH_OP";
        return "IDENTIFIER";
    }

    // Numbers
    if (state == "NUM_INT") return "INTEGER";
    if (state == "NUM_REAL") return "REAL";

    // String and Character Literals
    if (state == "STR") return "STRING";
    if (state == "CHR") return "CHAR";

    // Arithmetic Operators
    if (state == "OP_PLUS") return "PLUS";
    if (state == "OP_MINUS") return "MINUS";
    if (state == "OP_MUL") return "MULTIPLY";
    if (state == "OP_DIV") return "DIVIDE";

    // Relational Operators
    if (state == "OP_EQ") return "EQUAL";
    if (state == "OP_LT") return "LESS_THAN";
    if (state == "OP_GT") return "GREATER_THAN";
    if (state == "OP_LTE") return "LESS_EQUAL";
    if (state == "OP_GTE") return "GREATER_EQUAL";
    if (state == "OP_NEQ") return "NOT_EQUAL";

    // Assignment
    if (state == "ASSIGN") return "ASSIGNMENT";

    // Symbols
    if (state == "SYM_LPAREN") return "LPAREN";
    if (state == "SYM_RPAREN") return "RPAREN";
    if (state == "SYM_LBRACK") return "LBRACK";
    if (state == "SYM_RBRACK") return "RBRACK";
    if (state == "SYM_SEMI") return "SEMICOLON";
    if (state == "SYM_COMMA") return "COMMA";
    if (state == "COLON") return "COLON";
    if (state == "DOT") return "DOT";
    if (state == "RANGE") return "RANGE";

    return "UNKNOWN";
}
```

Fungsi ini melakukan beberapa hal penting seperti

- Case-insensitive keyword check
  Pascal case-insensitive, jadi identifier di-lowercase dulu sebelum dicek apakah termasuk keyword
- Prioritas keyword
  Cek KEYWORDS dulu, kemudian LOGICAL_WORDS, lalu ARITH_WORDS, baru IDENTIFIER
- Direct mapping
  Untuk state lain (numbers, operators, symbols), langsung map ke tipe token yang sesuai

2.6. Implementasi Main Program

File main.cpp adalah entry point program yang mengintegrasikan seluruh komponen bersama-sama.

2.6.1 Path Resolution

Fungsi helper resolve_from_here menangani path resolution untuk file input

```
static std::string resolve_from_here(const std::string& p) {
    fs::path cand(p);
    if (fs::exists(cand))
        return p;

    // Try ../p for relative path from build directory
    fs::path alt = fs::path("..") / p;
    if (fs::exists(alt))
        return alt.string();

    return p; // let open() fail later with clear error
}
```

Fungsi ini mencoba dua kemungkinan lokasi file: path asli dan path relatif dari parent directory (untuk kasus di mana executable berada di subdirectory build/).

2.6.2 Main Function

```
int main(int argc, char** argv) {
    if (argc < 2) {
            std::cerr << "Usage: " << argv[0] << " <source.pas> [--dfa
path/to/dfa.json]\n";
        return 1;
    }

    std::string source = argv[1];
    std::string dfa_path = "dfa/dfa.json";

    // Parse optional --dfa flag
    for (int i = 2; i < argc; i++) {
        if (std::string(argv[i]) == "--dfa" && i+1 < argc) {
            dfa_path = argv[i+1];
            break;
        }
    }

    // Resolve paths
    source = resolve_from_here(source);
    dfa_path = resolve_from_here(dfa_path);

    try {
        // Load DFA
        DFA dfa = load_dfa_json(dfa_path);

        // Read source file
        std::ifstream src_file(source);
        if (!src_file) {
             std::cerr << "Error: Cannot open source file: " << source <<
"\n";
```

```
            return 1;
        }
                                                                    std::string
    src_content((std::istreambuf_iterator<char>(src_file)),
                                    std::istreambuf_iterator<char>());

        // Tokenize
        Lexer lexer(dfa, src_content);
        std::vector<Token> tokens = lexer.tokenize();

        // Output tokens
        for (const auto& tok : tokens) {
            std::cout << tok.toString() << " ";
        }
        std::cout << "\n";

    } catch (const std::exception& e) {
        std::cerr << "Error: " << e.what() << "\n";
        return 1;
    }

    return 0;
}
```

Alur eksekusi:

- Argument parsing
  Cek jumlah argumen minimal, parse source file path dan optional --dfa flag
- Path resolution
  Resolve path untuk source file dan DFA file
- Load DFA
  Panggil load_dfa_json untuk membangun DFA dari JSON
- Read source
  Baca seluruh isi file sumber ke string
- Tokenization
  Buat Lexer object dan panggil tokenize()
- Output
  Cetak setiap token menggunakan method toString()
- Error handling
  Catch seluruh exception dan tampilkan error message

2.7. Error Handling

Program didesain dengan error handling untuk menangani berbagai kasus error.

2.7.1 File Error

```
if (!f)
    throw std::runtime_error("Cannot open DFA file: " + path);
```

```
if (!src_file) {
    std::cerr << "Error: Cannot open source file: " << source << "\n";
    return 1;
}
```

Program memeriksa keberhasilan pembukaan file dan memberikan pesan error yang jelas jika gagal.

2.7.2 Lexical Error

```
class LexerError : public std::runtime_error {
public:
    explicit LexerError(const std::string& msg)
        : std::runtime_error(msg) {}
};

throw LexerError("Unexpected character at line " +
                 std::to_string(line) + ", column " +
                 std::to_string(col));
```

Custom exception class LexerError digunakan untuk lexical errors dengan informasi lokasi yang presisi.

2.7.3 JSON Parsing Error

Library nlohmann/json secara otomatis throw exception jika format JSON tidak valid. Program menangkap exception ini di main function dan menampilkan pesan error.

2.7.4 Defensive Programming

```
if (i >= src.size())
    return '\0';

size_t idx = i + (k < 0 ? 0 : static_cast<size_t>(k));
if (idx < src.size())
    return src[idx];
```

Fungsi-fungsi helper melakukan bounds checking untuk mencegah buffer overflow dan undefined behavior.

BAB III

Pengujian

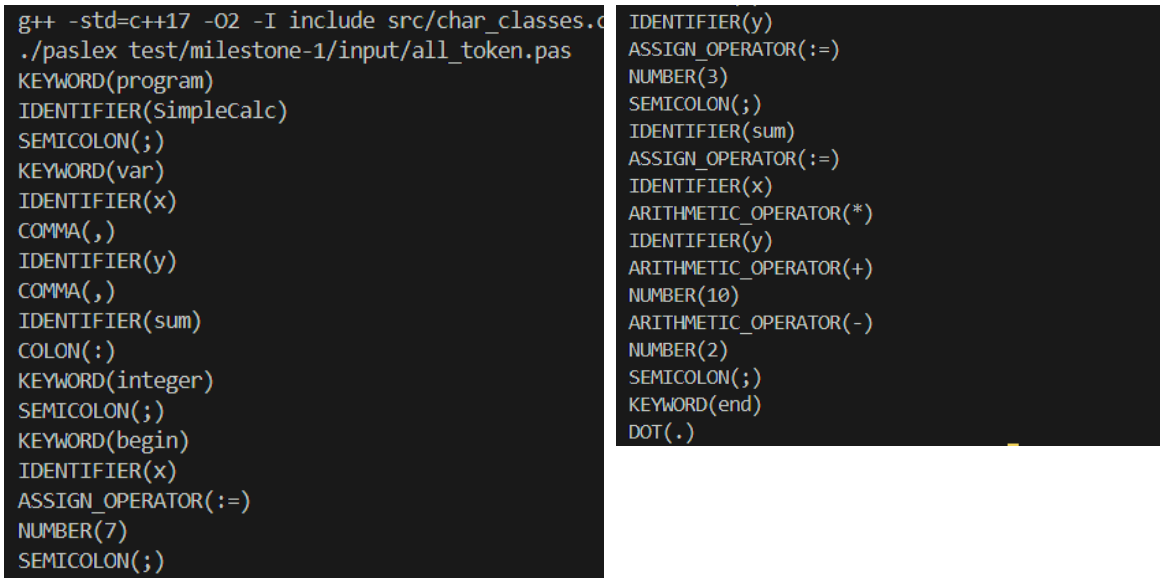
3.1. Test Case 1 - SimpleCalc

Test case ini bertujuan untuk menguji kemampuan lexer dalam mengenali token-token dasar dalam bahasa Pascal-S, seperti keyword, identifier, operator aritmatika, literal numerik, dan tanda baca. Kasus ini berfungsi sebagai dasar bahwa DFA sudah dapat melakukan scan urutan token paling umum dengan benar tanpa ambiguitas.

all_token.pas (input)

```
program SimpleCalc;
var
  x, y, sum: integer;
begin
  x := 7;
  y := 3;
  sum := x * y + 10 - 2;
end.
```

Output:



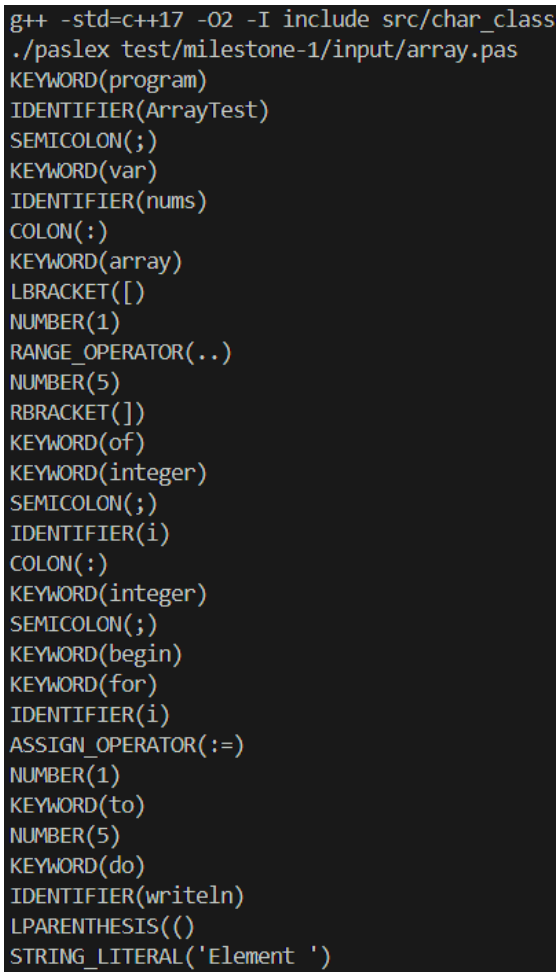
3.2. Test Case 2 - ArrayTest

Test case ini bertujuan untuk menguji kompleksitas struktur deklaratif, yaitu deklarasi array dengan range operator, lalu penggunaan of dan tipe data kompleks, serta akses elemen array di dalam statement. Kasus ini berfungsi untuk memastikan lexer mengenali variasi token lain dan string literal.

array.pas (Input)

```
program ArrayTest;
var
  nums: array [1..5] of integer;
  i: integer;
begin
  for i := 1 to 5 do
    writeln('Element ', i, ': ', nums[i]);
end.
```

Output:



3.3. Test Case 3 - TestComment

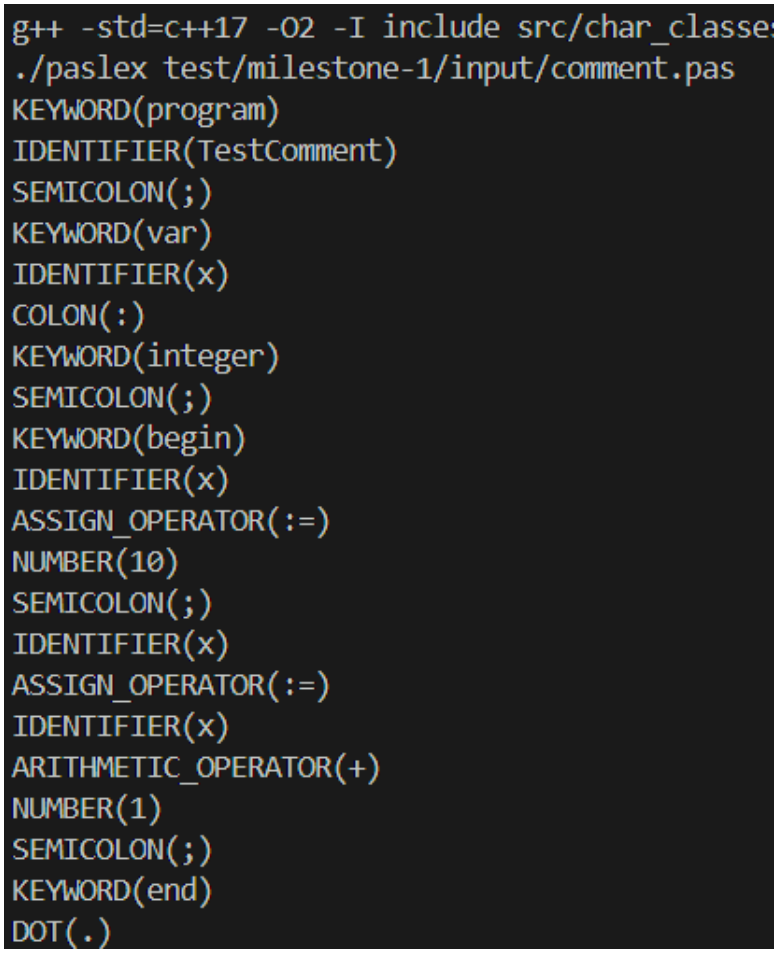
Test case ini bertujuan untuk menguji penanganan komentar dengan mengenali dan mengabaikan komentar namun tetap memproses token valid di luar komentar. Kasus ini berfungsi untuk memastikan DFA memisahkan ignored states.

comment.pas (Input)

```
program TestComment;

var x: integer;  { ini komentar rakdaf }
(* komentar lain ahsan*)
begin
  x := 10;   { assign nilai bevinda }
  x := x + 1; (* ini alhen *)
end.
```

Output:



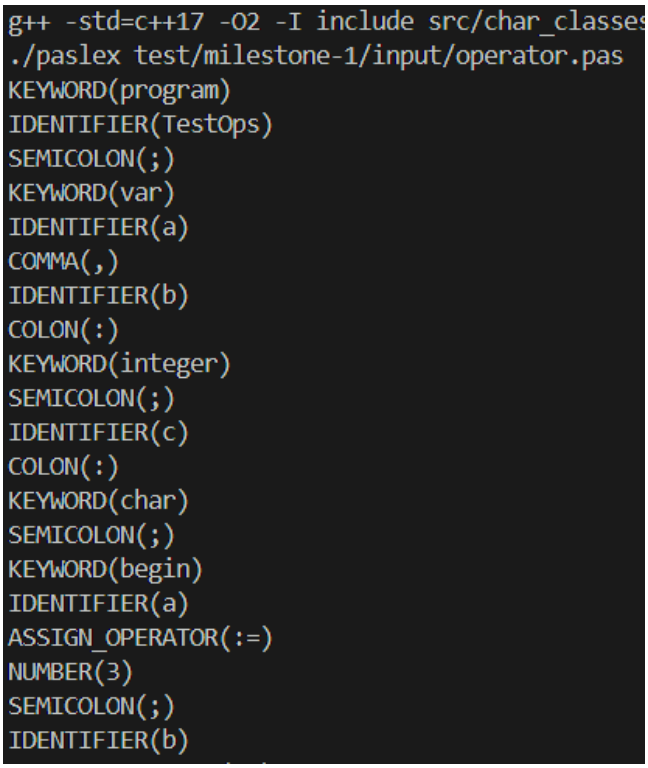
3.4. Test Case 4 - TestOps

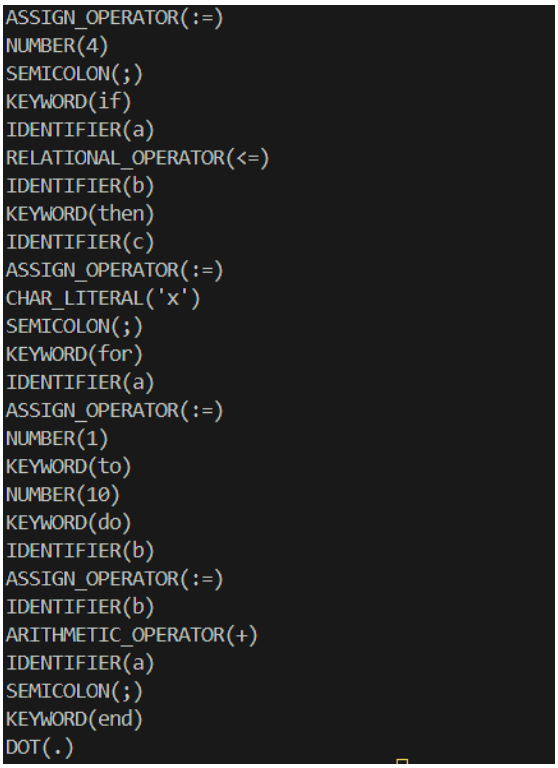
Test case ini bertujuan untuk menguji pengenalan operator multi karakter dan literal karakter tunggal seperti relational operator, assign operator, range operator, dan char literal.

operator.pas

```
program TestOps;
var
  a, b: integer;
  c: char;
begin
  a := 3;
  b := 4;
  if a <= b then
    c := 'x';
  for a := 1 to 10 do
    b := b + a;
end.
```

Output:





3.5. Test Case 5 - LogicTest

Test case ini bertujuan untuk menguji operator logika dan ekspresi kompleks seperti logical operator, relational operator, tanda kurung bertingkat, dan string literal.

string_and_logical.pas

```
program LogicTest;
var
  flag: boolean;
begin
  flag := (5 < 10) and not (3 > 1);
  writeln('Check: ', flag);
end.
```

Output:

```
g++ -std=c++17 -O2 -I include src/char_classes.cpp src/
./paslex test/milestone-1/input/string_and_logical.pas
KEYWORD(program)
IDENTIFIER(LogicTest)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(flag)
COLON(:)
KEYWORD(boolean)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(flag)
ASSIGN_OPERATOR(:=)
LPARENTHESIS(()
NUMBER(5)
RELATIONAL_OPERATOR(<)
NUMBER(10)
RPARENTHESIS())
```

```
LOGICAL_OPERATOR(and)
LOGICAL_OPERATOR(not)
LPARENTHESIS(()
NUMBER(3)
RELATIONAL_OPERATOR(>)
NUMBER(1)
RPARENTHESIS())
SEMICOLON(;)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Check: ')
COMMA(,)
IDENTIFIER(flag)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(end)
DOT(.)
```

BAB IV

KESIMPULAN DAN SARAN

4.1. Kesimpulan

Berdasarkan pengerjaan milestone 1 yang telah kami lakukan, kami telah berhasil membuat sebuah *lexical analyzer* atau lexer untuk bahasa Pascal-S yang berfungsi untuk mengubah kode sumber menjadi rangkaian token bermakna menggunakan pendekatan Deterministic Finite Automata atau DFA. Lexer yang telah dibuat mampu mengenali seluruh jenis token yang tercantum dalam spesifikasi, termasuk keyword, identifier, literal, operator aritmatika dan logika, tanda baca, serta struktur seperti array dan komentar.

Pengujian dilakukan pada lima kasus uji unik yang mencakup berbagai aspek yaitu token dasar, komentar, operator, ekspresi logika, serta struktur kompleks. Hasil seluruh pengujian menunjukkan bahwa program berhasil mengklasifikasikan token secara benar dan mengabaikan elemen non-esensial seperti komentar dan spasi. Dengan demikian, tahap analisis leksikal dapat dikatakan telah berjalan sesuai spesifikasi yang telah diberikan.

4.2. Saran

Tidak ada saran yang bisa kami berikan namun kami ingin mengapresiasi diri kami sendiri karena telah berhasil menyelesaikan milestone 1 tubes TBFO ini dalam kurun waktu yang cukup cepat dan dalam tekanan tinggi dari tugas lain serta persiapan UTS.

LAMPIRAN

Link Release Repository Github.

https://github.com/ahsuunn/NTB-Tubes-IF2224/releases/tag/v0.1.1

Link diagram.

https://s.hmif.dev/Diagram-1-NTB

Pembagian Tugas (menunjukan persentase kontribusi)

NIM	Persentase Kerja	Tugas
13523018	25%	Melakukan pengujian dan membuat kesimpulan pada laporan
13523062	25%	Mengimplementasikan lexer, dfa_loader, dan membuat diagram DFA
13523074	25%	Membuat diagram DFA, penjelasan DFA pada laporan, dan membuat makefile
13523120	25%	Membuat landasan teori, membuat penjelasan implementasi, dan membuat file README.md