

LAPORAN MILESTONE 1: LEXICAL ANALYZER

IF2224 Teori Bahasa Formal dan Automata



Disusun oleh:

Kelompok JYP, K-02

Noumisyifa Nabila Nareswari	13523058
M. Ghifary Komara Putra	13523066
Sabilul Huda	13523072
Diyah Susan Nugrahani	13523080
Henry Filberto Shinelu	13523108

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2025

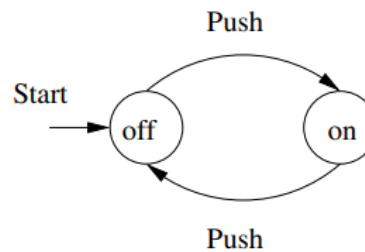
DAFTAR ISI

A. LANDASAN TEORI.....	3
1. Finite Automata.....	3
2. Deterministic Finite Automata (DFA).....	4
3. Lexical Analyzer.....	5
4. Pascal-S.....	7
B. PERANCANGAN DAN IMPLEMENTASI.....	8
1. Deskripsi Umum.....	8
2. Rancangan Model DFA.....	8
3. Rincian Implementasi.....	9
C. PENGUJIAN.....	13
E. LAMPIRAN.....	15
1. Link Repository Github.....	15
2. Link Workspace Diagram DFA.....	15
3. Pembagian Tugas.....	15

A. LANDASAN TEORI

1. *Finite Automata*

Finite automata atau *otomata hingga* merupakan salah satu model dasar dalam teori bahasa formal dan komputasi yang digunakan untuk merepresentasikan sistem dengan jumlah keadaan (*state*) yang terbatas. Setiap saat, automata berada pada satu keadaan tertentu dan berpindah ke keadaan lain berdasarkan masukan (*input*) yang diterima sesuai dengan fungsi transisi yang telah ditentukan. Model ini banyak dimanfaatkan dalam bidang perangkat keras dan perangkat lunak karena mampu menggambarkan perilaku sistem secara terstruktur dengan sumber daya yang tetap.



Gambar 1. Model *finite automaton* sebuah saklar hidup-mati

Sumber: Hopcroft et al., 2004

Secara umum, sebuah *finite automaton* terdiri atas lima komponen utama, yaitu: (1) himpunan keadaan (*states*), (2) himpunan simbol masukan (*input alphabet*), (3) fungsi transisi yang menentukan perpindahan antar-keadaan, (4) keadaan awal (*start state*), dan (5) satu atau lebih keadaan akhir (*accepting states*). Automata akan menerima suatu rangkaian masukan apabila setelah seluruh simbol diproses, sistem berakhir pada salah satu keadaan akhir.

Penerapan *finite automata* sangat luas, di antaranya pada perancangan dan verifikasi rangkaian digital, pembuatan *lexical analyzer* dalam proses kompilasi, pencarian pola dalam teks, serta verifikasi sistem dengan jumlah keadaan terbatas seperti protokol komunikasi atau sistem keamanan.

2. *Deterministic Finite Automata (DFA)*

Deterministic Finite Automata (DFA) merupakan bentuk formal dari *finite automata* yang bersifat deterministik, yaitu automata yang selalu berada tepat pada satu keadaan (*state*) tertentu setelah membaca setiap rangkaian simbol masukan (*input*). Sifat deterministik ini berarti bahwa untuk setiap keadaan dan setiap simbol masukan, hanya ada satu kemungkinan keadaan tujuan yang dapat dicapai. Hal ini membedakan DFA dengan *Nondeterministic Finite Automata (NFA)* yang dapat berada dalam lebih dari satu keadaan sekaligus.

Secara formal, sebuah DFA didefinisikan sebagai 5 *tuple*:

$$A = (Q, \Sigma, \delta, q_0, F)$$

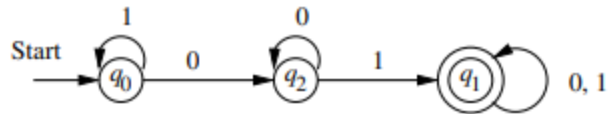
dengan:

1. Q adalah himpunan berhingga dari keadaan (*states*),
2. Σ adalah himpunan berhingga dari simbol masukan (*input alphabet*),
3. $\delta: Q \times \Sigma \rightarrow Q$ merupakan fungsi transisi yang memetakan pasangan keadaan dan simbol masukan ke keadaan berikutnya,
4. $q_0 \in Q$ adalah keadaan awal (*start state*), dan
5. $F \subseteq Q$ adalah himpunan keadaan akhir atau *accepting states*.

DFA memproses suatu string masukan dengan membaca simbol satu per satu mulai dari keadaan awal. Untuk setiap simbol, fungsi transisi δ menentukan keadaan berikutnya. Setelah seluruh simbol diproses, jika automata berakhir pada salah satu keadaan dalam himpunan F , maka string tersebut diterima, jika tidak, maka ditolak. Himpunan semua string yang diterima oleh DFA disebut sebagai bahasa (*language*) yang dikenali DFA tersebut.

Dalam praktiknya, DFA dapat direpresentasikan dalam dua bentuk utama:

1. Diagram transisi, berupa graf berarah di mana simpul menunjukkan keadaan dan busur menunjukkan fungsi transisi yang dilabeli simbol masukan.



Gambar 2. Contoh diagram transisi DFA

Sumber: Hopcroft et al., 2004

Keadaan awal ditandai dengan panah masuk bertuliskan *Start*, dan keadaan akhir ditandai dengan lingkaran ganda.

2. Tabel transisi, yang menyajikan fungsi transisi δ dalam bentuk tabel, di mana baris mewakili keadaan dan kolom mewakili simbol masukan.

	0	1
$\rightarrow q_0$	q_2	q_0
$* q_1$	q_1	q_1
q_2	q_2	q_1

Gambar 3. Contoh tabel transisi DFA

Sumber: Hopcroft et al., 2004

Kedua representasi tersebut mengandung informasi yang sama dan digunakan untuk menggambarkan perilaku automata secara lebih mudah dibaca.

3. *Lexical Analyzer*

Lexical Analyzer atau lexer merupakan komponen utama pada tahap awal proses kompilasi yang bertugas mengubah *string* mentah dari *source code* menjadi satuan-satuan bermakna yang disebut token. Setiap token merupakan pasangan dari dua elemen, yaitu lexeme dan token type. *Lexeme* adalah sebuah set karakter aktual yang muncul dalam kode sumber. Sedangkan token adalah kategori atau kelas dari lexeme tersebut. Selain itu, pattern digunakan untuk menggambarkan bentuk umum dari suatu token, seperti ekspresi reguler, DFA, dan NFA yang menentukan pola karakter yang dikenali lexer.

Proses *lexical analysis* dilakukan dengan membaca karakter secara berurutan dan mencocokkannya terhadap pola yang didefinisikan. Ketika suatu *lexeme* cocok dengan pola tertentu, lexer menghasilkan token yang berisi jenis (*type*) dan nilai (*value*) dari lexeme tersebut. Token-token inilah yang kemudian menjadi masukan bagi tahap *syntax analysis*.

Dalam konteks tugas ini, daftar token yang digunakan mengikuti yang diberikan pada spesifikasi yang sudah disesuaikan untuk bahasa PASCAL-S, dengan daftar token sebagai berikut:

Daftar Token			
No	Tipe (type)	Nilai (value)	Keterangan
1	KEYWORD	program, var, begin, end, if, then, else, while, do, for, to, downto, integer, real, boolean, char, array, of, procedure, function, const, type	Kata kunci yang sudah didefinisikan oleh bahasa Pascal-S dan memiliki fungsi khusus dalam struktur program.
2	IDENTIFIER	x, y, z, sum, avg, count	Nama yang didefinisikan oleh pengguna, misalnya nama variabel, prosedur, atau fungsi.
3	ARITHMETIC_OPERATOR	+, -, *, /, div, mod	-
4	RELATIONAL_OPERATOR	=, <>, <, <=, >, >=	-
5	LOGICAL_OPERATOR	and, or, not	-
6	ASSIGN_OPERATOR	:=	Operator penugasan yang digunakan untuk memberi nilai ke variabel
7	NUMBER	22, 3, 2018	Bilangan berupa integer atau ril
8	CHAR_LITERAL	'a', 'b', 'c'	-

9	STRING_LITERAL	'tbfo', 'seru sekali'	-
10	SEMICOLON	;	-
11	COMMA	,	-
12	COLON	:	-
13	DOT	.	-
14	LPARENTHESIS	(-
15	RPARENTHESIS)	-
16	LBRACKET	[-
17	RBRACKET]	-
18	RANGE_OPERATOR	..	-

Selain itu, untuk tugas ini, proses analisis leksikal dibangun berdasarkan konsep *Deterministic Finite Automata (DFA)*. DFA digunakan sebagai basis aturan (*rule base*) untuk mengenali pola karakter dalam *lexeme*. Setiap keadaan (*state*) dalam DFA merepresentasikan kondisi pengenalan sebagian *lexeme*, dan ketika DFA mencapai keadaan akhir (*final state*), lexer mengidentifikasi *lexeme* yang valid dan mengubahnya menjadi token sesuai jenis yang telah ditentukan dalam spesifikasi.

4. Pascal-S

Pascal-S merupakan subset dari bahasa pemrograman Pascal yang dikembangkan oleh Niklaus Wirth di ETH Zürich. Bahasa ini dirancang khusus untuk tujuan pengajaran dasar pemrograman bagi mahasiswa teknik, fisika, dan matematika tingkat awal. Motivasi utama pembuatannya adalah untuk menyediakan sistem yang efisien dan mudah dipelajari, sekaligus tetap mendukung *error checking* yang menyeluruh.

Pascal-S dibuat sebagai versi yang lebih sederhana dari Pascal standar, dengan menghilangkan fitur-fitur kompleks yang tidak esensial untuk pembelajaran dasar, seperti *pointer*, *set*, *file structure*, *goto statement*, dan *variant record*. Namun, bahasa ini tetap

mempertahankan konsep fundamental dari pemrograman terstruktur seperti tipe data dasar (*integer*, *real*, *boolean*, *char*), array, record, serta struktur kontrol seperti *if*, *while*, dan *for*. Walaupun begitu, bahasa ini tetap kompatibel dengan Pascal standar.

B. PERANCANGAN DAN IMPLEMENTASI

1. Deskripsi Umum

Lexical analyzer dan *compiler* yang akan dibangun pada tugas ini akan diimplementasikan menggunakan bahasa C++. Justifikasi dari pemilihan bahasa tersebut adalah karena C++ bahasa yang relatif lebih *robust* apabila dibandingkan dengan performa *interpreted language* seperti Python dan Javascript, namun tetap menyediakan *library* yang cukup untuk pemrosesan *string*.

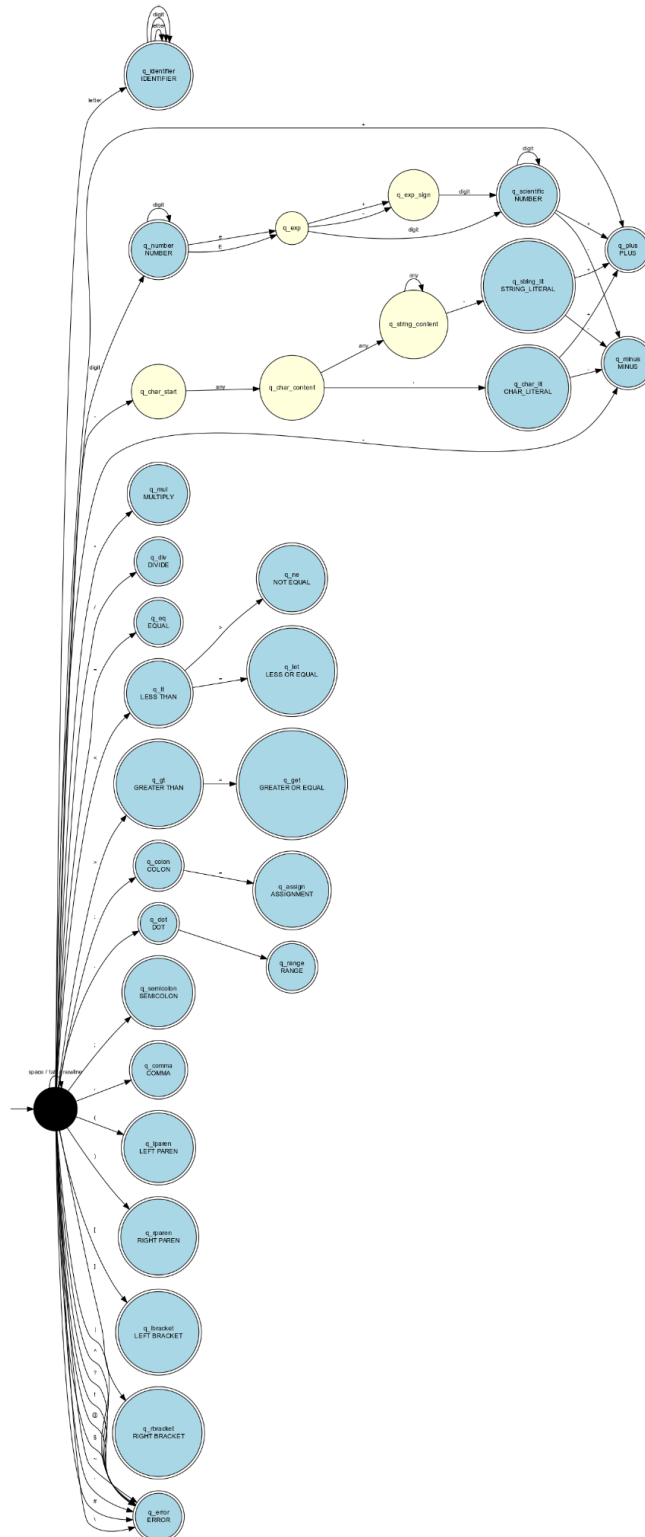
Sistem *lexer* akan dibangun berdasarkan aturan dengan struktur DFA sebagai dasar dalam mengenali pola *lexeme* pada *source code*. Aturan DFA akan didefinisikan dalam file JSON yang berisikan daftar *state* yang akan dibaca oleh *lexer* secara *runtime*.

Struktur file JSON terdiri atas beberapa bagian utama:

- *character_classes*: Mendefinisikan suatu karakter masuk ke jenis/kelas apa. Contoh dari kelas adalah karakter angka masuk ke kelas “digit”, lalu karakter alfabetikal masuk ke kelas “letter”.
- *dfa_config*: Berisi informasi terkait apa saja *start states* dan *final states* yang ada di DFA ini.
- *keyword_lookup*: Merepresentasikan *lookup table* yang kebanyakan berkaitan dengan token KEYWORD.
- *transitions*: mendefinisikan aturan perpindahan *state* dengan format {"from": "qi", "input": "x", "to": "qj"}
- *state_token_map*: memetakan setiap *final state* ke token yang sesuai.

2. Rancangan Model DFA

Berikut adalah rancangan model DFA untuk *lexer* bahasa Pascal-S



Gambar 4. Rancangan model DFA untuk *lexer compiler* bahasa Pascal-S

Sumber: [pranala ke workspace](#)

Diagram tersebut menunjukkan skema DFA untuk *lexer compiler*. Lingkaran hitam menggambarkan q0 atau initial state yang merupakan awal dari seluruh state yang ada. Lingkaran berwarna kuning menggambarkan transition state, kondisi saat automata sedang berpindah dari satu state ke state lain karena membaca input tertentu namun belum mencapai final state. Lingkaran berwarna biru menggambarkan final state, kondisi ketika input yang dibaca oleh automata dinyatakan valid dan menghasilkan token. Pada kasus input yang tidak valid maka akan langsung *terminate* dan menuju state q_error.

Transisi dari satu state ke state lain dihubungkan dengan panah berarah dengan simbol tertentu yang merepresentasikan input yang diterima. Simbol-simbol tersebut dapat berupa ikon, digit, letter, dan any. Ikon merepresentasikan lambang sesuai dengan fungsinya masing-masing, seperti ikon '+' merepresentasikan penjumlahan. Untuk digit dan letter merepresentasikan angka dan huruf sedangkan untuk any merepresentasikan seluruh karakter yang ada.

Pada tugas kali ini, digunakan konsep *lookup table* untuk memperringkas diagram DFA terutama pada bagian identifier dan keyword. Konsep dari *lookup table* ini adalah DFA membaca karakter per karakter hingga membentuk *keyword* atau *identifier* tertentu, lalu subset tersebut nantinya akan dicek ke *lookup table* untuk mencari token yang sesuai dengan subset yang diterima. Misalnya automata membaca letter p → r → o → g → r → a → m, maka nantinya subset 'program' ini akan dicocokkan ke *lookup table* dan apabila ada yang cocok maka akan menghasilkan token yang bersesuaian.

3. Rincian Implementasi

Terdapat satu struktur data dan tiga fungsi yang menjadi fokus utama dalam pembentukan *lexer* yang didefinisikan pada file `lexer.hpp` sebagai berikut:

```
struct Token {
    string type;
    string lexeme;
};

string classifyChar(char c);
vector<Token> runDFA(
    const string &input,
    const json &rules,
    const unordered_set<string> &keywords,
    const unordered_map<string, string> &singleCharTokens,
    const unordered_map<string, string> &multiCharTokens
```

```
);  
  
int lexer_main(int argc, char* argv[]);
```

- Struktur data Token berfungsi untuk merepresentasikan satu token yang menyimpan informasi tipe token tersebut (seperti KEYWORD, NUMBER, OPERATOR) sekaligus *lexeme* yang dikategorikan ke jenis token tersebut.
- Fungsi `ClassifyChar` berfungsi untuk mengkategorikan karakter saat ini masuk ke jenis karakter apa sesuai dengan apa yang didefinisikan pada JSON.
- Fungsi `runDFA` sebagai representasi *engine* DFA.
- Fungsi `lexer_main` sebagai representasi *main* program.

Berikut adalah implementasi fungsi `ClassifyChar`:

```
string classifyChar(char c, const unordered_map<char, string> &charMap)  
{  
    auto it = charMap.find(c);  
    if (it != charMap.end()) {  
        return it->second;  
    }  
    return "any";  
}
```

Fungsi ini akan mencocokkan apakah karakter `c` pada input cocok pada salah satu kelas karakter yang didefinisikan pada `charMap`.

Berikut adalah implementasi fungsi `runDFA`:

```
vector<Token> runDFA(  
    const string &input,  
    const json &rules,  
    const unordered_set<string> &keywords  
)  
{  
    // Build character classification map  
    auto charMap = buildCharMap(rules["character_classes"]);  
  
    // Build transition table  
    unordered_map<string, unordered_map<string, string>> transition;
```

```

for (auto &block : rules["transitions"]) {
    for (auto &r : block["rules"]) {
        string from = r["from"], inp = r["input"], to = r["to"];
        transition[from][inp] = to;
    }
}

unordered_set<string> finals;
for (auto &s : rules["dfa_config"]["final_states"])
    finals.insert(s);

unordered_map<string,string> stateToToken =
    rules["state_token_map"].get<unordered_map<string,string>>();

vector<Token> tokens;
string state = rules["dfa_config"]["start_state"];
string cur;

for (size_t i = 0; i <= input.size(); ++i) {
    char c = (i < input.size()) ? input[i] : '\0';
    string cls = classifyChar(c, charMap);

    if (state == "q0" && (cls == "space" || cls == "tab" || cls ==
"newline")) {
        continue;
    }

    // SPECIAL CASE: Handle 5..7 pattern (NON DFA)
    if (state == "q_number_dot" && c == '.' && i < input.size()) {
        // Emit the number (without the first dot)
        string numLexeme = cur.substr(0, cur.size() - 1); // Remove
trailing '.'
        tokens.push_back({"NUMBER", numLexeme});

        // Now process ".." as range operator
        // Start from q0, read first '.', then second '.'
        cur = "..";
        tokens.push_back({"RANGE_OPERATOR", ".."});

        cur.clear();
        state = "q0";
        continue; // Skip to next character after the second '.'
    }

    string next;
    bool transition_found = false;

    // Try exact character match first
    if (transition[state].count(string(1, c))) {
        next = transition[state][string(1, c)];
        transition_found = true;
    }
    // Try character class match
    else if (transition[state].count(cls)) {
        next = transition[state][cls];
        transition_found = true;
    }
    // Try 'any' transition

```

```

        else if (transition[state].count("any") && c != '\\0') {
            next = transition[state]["any"];
            transition_found = true;
        }

        if (transition_found) {
            cur += c;
            state = next;
        } else {
            // No valid transition -> check if we ended a token
            if (finals.count(state)) {
                string tokType = stateToToken[state];
                if (state == "q_identifier" && keywords.count(cur)) {
                    tokType = "KEYWORD";
                }
                tokens.push_back({tokType, cur});
                cur.clear();
                state = rules["dfa_config"]["start_state"];
                --i;
            }
        }
    }

    if (!cur.empty()) {
        if (finals.count(state)) {
            string tokType = stateToToken[state];
            if (state == "q_identifier" && keywords.count(cur)) {
                tokType = "KEYWORD";
            }
            tokens.push_back({tokType, cur});
        }
    }

    return tokens;
}

```

Logika yang terjadi pada fungsi ini adalah sebagai berikut:

- Fungsi menerima input berupa *string* yang ingin ditokenisasi, *rules* yang merupakan isi dari JSON DFA, dan *keywords* yang merupakan representasi *lookup table* untuk beberapa *keyword* dan *operator*.
- Ekstraksi semua informasi yang ada di JSON terkait aturan DFA dan juga inisialisasi variabel awal seperti mengatur variabel *state* menjadi *state awal* pada file JSON, menyiapkan *vector of Token*, dan inisiasi variabel *cur* bertipe *string*.
- Fungsi akan mulai iterasi satu per satu karakter yang ada pada string &input pada dan di dalamnya akan dicocokkan dengan berbagai kondisi sebagai berikut:

- Skip Whitespace di State Awal
- Mencari Transisi yang Valid
- Jika Transisi Ditemukan, karakter ditambahkan ke *lexeme* saat ini, state DFA berpindah ke *state* berikutnya, lalu lanjut membaca karakter selanjutnya.
- Jika tidak ada transisi valid, terdapat dua kemungkinan:
 - Token berhasil dikenali karena berada di *final state*, khusus untuk identifier, cek apakah *lexeme*-nya adalah keyword. Simpan token ke vector hasil lalu reset *lexeme* dan state ke awal. Lalu backtrack satu karakter untuk memproses ulang karakter yang menyebabkan tidak ada transisi (karakter tersebut mungkin awal dari token baru).
 - Terjadi error karena tidak ada transisi valid dan state bukan final state, yang berarti, *lexeme* yang dibaca tidak membentuk token yang valid lalu print pesan error dan masukkan token ERROR ke vector. Setelah itu, return vector token (proses tokenisasi berhenti).
- Setelah semua karakter diproses, cek apakah masih ada *lexeme* tersisa. Jika state terakhir adalah *final state*, simpan sebagai token valid. Jika tidak, laporkan sebagai error
- Program melakukan *error handling* leksikal
 - Ada dua cara identifikasi:
 - Transisi ke state error Jika DFA memiliki state *q_error* dan token map menandainya sebagai "ERROR", Ketika state mencapai final state dan *stateToToken[state]* menghasilkan "ERROR", atau ketika state saat ini adalah *q_error*
 - Stuck di Non-Final State jika tidak ada transisi yang valid dari state saat ini dan state saat ini bukan termasuk final state dengan buffer (*cur*) tidak kosong
 - Ketika error terdeteksi, pesan error ditampilkan ke *stderr*, token dengan tipe ERROR dimasukkan ke *vector* hasil, proses tokenisasi langsung dihentikan (return).

Berikut adalah implementasi fungsi `lexer_main`:

```

int lexer_main(int argc, char* argv[]) {
    if (argc < 2) {
        cerr << "Usage: " << argv[0] << " <source_file.pas>\n";
        return 1;
    }

    const string ruleFile = "test/milestone-1/rule.json";
    ifstream jfile(ruleFile);
    if (!jfile) {
        cerr << "Error: cannot open " << ruleFile << "\n";
        return 1;
    }
    json rules;
    jfile >> rules;

    // Load keyword/operator sets
    unordered_set<string> keywords;
    for (auto &kw : rules["keyword_lookup"]["keywords"])
        keywords.insert(kw);
    for (auto &kw : rules["keyword_lookup"]["logical_operators"])
        keywords.insert(kw);
    for (auto &kw :
        rules["keyword_lookup"]["arithmetic_word_operators"])
        keywords.insert(kw);

    // Read Pascal
    ifstream f(argv[1]);
    if (!f) { cerr << "Cannot open file\n"; return 1; }
    stringstream buf; buf << f.rdbuf(); string input = buf.str();

    // Run DFA
    vector<Token> toks = runDFA(input, rules, keywords);

    // Print tokens
    for (auto &t : toks)
        cout << "<" << t.type << "(" << t.lexeme << ">\n";

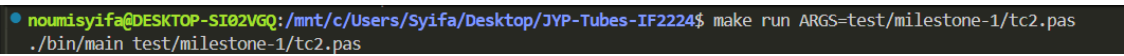
    return 0;
}

```

Apa yang terjadi pada fungsi ini adalah sebagai berikut:

- Fungsi melakukan berbagai validasi terlebih dahulu seperti validasi *command* untuk *compile* dan ketersediaan *file* terkait.
- Setelah itu, fungsi load semua *keyword* dan *operator* yang terdaftar di *lookup table* pada JSON.
- Setelah itu fungsi mengekstrak isi dari file Pascal yang ingin di-*compile* lalu menjalankan fungsi *runDFA* dan menyimpan hasil fungsi tersebut ke *vector of Token*.
- Setelah itu fungsi *print* semua token yang dihasilkan

C. PENGUJIAN

Pengujian 1: tc2.pas
<i>Test Case</i>
<pre>program tc2; var x, y, result: integer; average: real; begin x := 10; y := 20; result := x + y; average := result / 2; if result > 25 then writeln('Damn bro that is huge') else writeln('Damn bro that is kinda small ..'); end.</pre>
Bukti Input

Bukti Output


```

noumisyifa@DESKTOP-SI02VGQ:/mnt/c/Users/Syifa/Desktop/JYP-Tubes-IF2224$ make run ARGS=test/milestone-1/tc2.pas
./bin/main test/milestone-1/tc2.pas
<KEYWORD(program)>
<IDENTIFIER(tc2)>
<SEMICOLON(;)>
<KEYWORD(var)>
<IDENTIFIER(x)>
<COMMA(,)>
<IDENTIFIER(y)>
<COMMA(,)>
<IDENTIFIER(result)>
<COLON(:)>
<KEYWORD(integer)>
<SEMICOLON(;)>
<IDENTIFIER(average)>
<COLON(:)>
<KEYWORD(real)>
<SEMICOLON(;)>
<KEYWORD(begin)>
<IDENTIFIER(x)>
<ASSIGN_OPERATOR(=)>
<NUMBER(10)>
<SEMICOLON(;)>
<IDENTIFIER(y)>
<ASSIGN_OPERATOR(=)>
<NUMBER(20)>
<SEMICOLON(;)>
<IDENTIFIER(result)>
<ASSIGN_OPERATOR(=)>
<IDENTIFIER(x)>
<ARITHMETIC_OPERATOR(+)>
<IDENTIFIER(y)>
<SEMICOLON(;)>
<IDENTIFIER(average)>
<ASSIGN_OPERATOR(=)>
<IDENTIFIER(result)>
<ARITHMETIC_OPERATOR(/)>
<NUMBER(2)>
<SEMICOLON(;)>
<KEYWORD(if)>
<IDENTIFIER(result)>
<RELATIONAL_OPERATOR(>)>
<NUMBER(25)>
<KEYWORD(then)>
<IDENTIFIER(writeln)>
<LPARENTHESIS(<)>
<STRING_LITERAL('Damn bro that is huge')>
<RPARENTHESIS(>)>
<STRING_LITERAL('Damn bro that is kinda small ..')>
<RPARENTHESIS(>)>
<SEMICOLON(;)>
<KEYWORD(end)>
<DOT(.)>

```

Pengujian 2: tc3.pas

Test Case

```

program loopdeloop;
var
    numbers: array[1 .. 10] of integer;
    i, total: integer;
begin
    total := 0;
    for i := 1 to 10 do

```

```
begin
    numbers[i] := i * 2;
    total := total + numbers[i];
end;

i := 10;
while i >= 1 do
begin
    writeln('Element ', i, ' = ', numbers[i]);
    i := i - 1;
end;
end.
```

Bukti Input

```
noumisyifa@DESKTOP-SI02VGQ:/mnt/c/Users/Syifa/Desktop/JYP-Tubes-IF2224$ make run ARGS=test/milestone-1/tc3.pas
./bin/main test/milestone-1/tc3.pas
```

Bukti Output

```
noumisyaifa@DESKTOP-SI02VGQ: /mnt/c/Users/Syifa/Desktop/JYP-Tubes-IF2224$ make run ARGS=test/milestone-1/tc3.pas
./bin/main test/milestone-1/tc3.pas
<KEYWORD(program)>
<IDENTIFIER(loopde loop)>
<SEMICOLON(;>
<KEYWORD(var)>
<IDENTIFIER(numbers)>
<COLON(:)>
<KEYWORD(array)>
<LBRACKET([>
<NUMBER(1)>
<RANGE_OPERATOR(..)>
<NUMBER(10)>
<RBRACKET]>
<KEYWORD(of)>
<KEYWORD(integer)>
<SEMICOLON(;>
<IDENTIFIER(i)>
<COMMA,>
<IDENTIFIER(total)>
<COLON(:)>
<KEYWORD(integer)>
<SEMICOLON(;>
<KEYWORD(begin)>
<IDENTIFIER(total)>
<ASSIGN_OPERATOR(:=)>
<NUMBER(0)>
<SEMICOLON(;>
<KEYWORD(for)>
<IDENTIFIER(i)>
<ASSIGN_OPERATOR(:=)>
<NUMBER(1)>
<KEYWORD(to)>
<NUMBER(10)>
<KEYWORD(do)>
<KEYWORD(begin)>
<IDENTIFIER(numbers)>
<LBRACKET([>
<IDENTIFIER(i)>
<RBRACKET]>
<ASSIGN_OPERATOR(:=)>
<IDENTIFIER(i)>
<ARITHMETIC_OPERATOR(*)>
<NUMBER(2)>
<SEMICOLON(;>
<IDENTIFIER(total)>
<ASSIGN_OPERATOR(:=)>
<IDENTIFIER(total)>
<ARITHMETIC_OPERATOR(+)>
<IDENTIFIER(numbers)>
```

```

<LBRACKET([]>
<IDENTIFIER(i)>
<RBRACKET(]>
<SEMICOLON(;)>
<KEYWORD(end)>
<SEMICOLON(;)>
<IDENTIFIER(i)>
<ASSIGN_OPERATOR(:=)>
<NUMBER(10)>
<SEMICOLON(;)>
<KEYWORD(while)>
<IDENTIFIER(i)>
<RELATIONAL_OPERATOR(>=)>
<NUMBER(1)>
<KEYWORD(do)>
<KEYWORD(begin)>
<IDENTIFIER(writeIn)>
<LPARENTHESIS((>
<STRING_LITERAL('Element '>
<COMMA(>)>
<IDENTIFIER(i)>
<COMMA(>)>
<STRING_LITERAL(' = '>
<COMMA(>)>
<IDENTIFIER(numbers)>
<LBRACKET([]>
<IDENTIFIER(i)>
<RBRACKET(]>
<RPARENTHESIS(>)>
<SEMICOLON(;)>
<IDENTIFIER(i)>
<ASSIGN_OPERATOR(:=)>
<IDENTIFIER(i)>
<ARITHMETIC_OPERATOR(-)>
<NUMBER(1)>
<SEMICOLON(;)>
<KEYWORD(end)>
<SEMICOLON(;)>
<KEYWORD(end)>
<DOT(.)>

```

Pengujian 3

Test Case

```

program erroridentifier;
var
    2abc: integer;
    valid_name: integer;
    9total: real;
    my$var: integer;
    numbers: array[1..10] of integer ;
begin
    2abc :  = 100;
    valid_name := 50;
    9total := 2abc + valid_name;
    my$var := 10;
end.

```

Bukti Input

```
noumisyifa@DESKTOP-SI02VGQ:/mnt/c/Users/Syifa/Desktop/JYP-Tubes-IF2224$ make run ARGS=test/milestone-1/tc4.pas
./bin/main test/milestone-1/tc4.pas
```

Bukti Output

```
<body>
noumisyifa@DESKTOP-SI02VGQ:/mnt/c/Users/Syifa/Desktop/JYP-Tubes-IF2224$ make run ARGS=test/milestone-1/tc4.pas
./bin/main test/milestone-1/tc4.pas
Lexical Error: Invalid token '2a' at position 34
noumisyifa@DESKTOP-SI02VGQ:/mnt/c/Users/Syifa/Desktop/JYP-Tubes-IF2224$
```

Pengujian 4

Test Case

```
program EdgeCaseNumbersAndStrings;
var
    zero: integer;
    negative: integer;
    huge: integer;
    pi: real;
    epsilon: real;
    empty: char;
    quote: char;
begin
    zero := 0;
    negative := -999;
    huge := 2147483647;
    pi := 3.14159265359;
    epsilon := 0.00001;
    numbers: array[1..10] of integer    ;

    empty := ' ';
    quote := ''';

    if zero = 0 then
        writeln('Zero works');

    if negative < 0 then
        writeln('Negative: ', negative);

    writeln('Pi = ', pi);
    writeln('String with spaces:  multiple  spaces');
    writeln('Special: ', quote);
    writeln('Numbers:', 1, 22, 333, 4444);
end.
```

Bukti Input

```
● noumisyifa@DESKTOP-SI02VGQ:/mnt/c/Users/Syifa/Desktop/JYP-Tubes-IF2224$ make run ARGS=test/milestone-1/tc5.pas
./bin/main test/milestone-1/tc5.pas
```

Bukti Output

```
● noumisyifa@DESKTOP-SI02VGQ:/mnt/c/Users/Syifa/Desktop/JYP-Tubes-IF2224$ make run ARGS=test/milestone-1/tc5.pas
./bin/main test/milestone-1/tc5.pas
<KEYWORD(program)>
<IDENTIFIER(EdgeCaseNumbersAndStrings)>
<SEMICOLON(;)>
<KEYWORD(var)>
<IDENTIFIER(zero)>
<COLON(:)>
<KEYWORD(integer)>
<SEMICOLON(;)>
<IDENTIFIER(negative)>
<COLON(:)>
<KEYWORD(integer)>
<SEMICOLON(;)>
<IDENTIFIER(huge)>
<COLON(:)>
<KEYWORD(integer)>
<SEMICOLON(;)>
<IDENTIFIER(pi)>
<COLON(:)>
<KEYWORD(real)>
<SEMICOLON(;)>
<IDENTIFIER(epsilon)>
<COLON(:)>
<KEYWORD(real)>
<SEMICOLON(;)>
<IDENTIFIER(empty)>
<COLON(:)>
<KEYWORD(char)>
<SEMICOLON(;)>
<IDENTIFIER(quote)>
<COLON(:)>
<KEYWORD(char)>
<SEMICOLON(;)>
<KEYWORD(begin)>
<IDENTIFIER(zero)>
<ASSIGN_OPERATOR(:=)>
<NUMBER(0)>
<SEMICOLON(;)>
<IDENTIFIER(negative)>
<ASSIGN_OPERATOR(:=)>
<ARITHMETIC_OPERATOR(-)>
<NUMBER(999)>
<SEMICOLON(;)>
<IDENTIFIER(huge)>
<ASSIGN_OPERATOR(:=)>
<NUMBER(2147483647)>
<SEMICOLON(;)>
<IDENTIFIER(pi)>
<ASSIGN_OPERATOR(:=)>
```

```
noumisyifa@DESKTOP-SI02VGQ:/mnt/c/Users/Syifa/Desktop/JYP-Tubes-IF2224$ make run ARGS=test/milestone-1/tc5.pas
<NUMBER(3)>
<DOT(.)>
<NUMBER(14159265359)>
<SEMICOLON(;)>
<IDENTIFIER(epsilon)>
<ASSIGN_OPERATOR(=)>
<NUMBER(0)>
<DOT(.)>
<NUMBER(00001)>
<SEMICOLON(;)>
<IDENTIFIER(numbers)>
<COLON(:)>
<KEYWORD(array)>
<LBRACKET([)>
<NUMBER(1)>
<RANGE_OPERATOR(..)>
<NUMBER(10)>
<RBRACKET(]>
<KEYWORD(of)>
<KEYWORD(integer)>
<SEMICOLON(;)>
<IDENTIFIER(empty)>
<ASSIGN_OPERATOR(=)>
<CHAR_LITERAL(' '>
<SEMICOLON(;)>
<IDENTIFIER(quote)>
<ASSIGN_OPERATOR(=)>
<CHAR_LITERAL('')>
<CHAR_LITERAL('')>
<SEMICOLON(;)>
<KEYWORD(if)>
<IDENTIFIER(zero)>
<RELATIONAL_OPERATOR(=)>
<NUMBER(0)>
<KEYWORD(then)>
<IDENTIFIER(writeIn)>
<LPARENTHESIS(()>
<STRING_LITERAL('Zero works')>
<RPARENTHESIS())>
<SEMICOLON(;)>
<KEYWORD(if)>
<IDENTIFIER(negative)>
<RELATIONAL_OPERATOR(<)>
<NUMBER(0)>
<KEYWORD(then)>
<IDENTIFIER(writeIn)>
<LPARENTHESIS(()>
<STRING_LITERAL('Negative: '>
<COMMA(,)>
```

```

<IDENTIFIER(negative)>
<RPARENTHESIS(<)>
<SEMICOLON(<)>
<IDENTIFIER(writeln)>
<LPARENTHESIS(<)>
<STRING_LITERAL('Pi = '>
<COMMA(<)>
<IDENTIFIER(pi)>
<RPARENTHESIS(<)>
<SEMICOLON(<)>
<IDENTIFIER(writeln)>
<LPARENTHESIS(<)>
<STRING_LITERAL('String with spaces:  multiple  spaces')>
<RPARENTHESIS(<)>
<SEMICOLON(<)>
<IDENTIFIER(writeln)>
<LPARENTHESIS(<)>
<STRING_LITERAL('Special: '>
<COMMA(<)>
<IDENTIFIER(quote)>
<RPARENTHESIS(<)>
<SEMICOLON(<)>
<IDENTIFIER(writeln)>
<LPARENTHESIS(<)>
<STRING_LITERAL('Numbers:')>
<COMMA(<)>
<NUMBER(1)>
<COMMA(<)>
<NUMBER(22)>
<COMMA(<)>
<NUMBER(333)>
<COMMA(<)>
<NUMBER(4444)>
<RPARENTHESIS(<)>
<SEMICOLON(<)>
<KEYWORD(end)>
<DOT(<)>

```

Pengujian 5

Test Case

```

program InvalidOperators;
var
    x, y: integer;
    result: real;
    message: string;
begin
    x := 10;
    y := 5;
    result := x @ y;
    x := x & 3;

    if x != y then
        writeln('Not equal');

    message := "This is double quote";
    result := 3.14.159;

```


<pre> x := 100%; y := #invalid; end. </pre>
Bukti Input
<pre> noumisyifa@DESKTOP-SI02VGQ:/mnt/c/Users/Syifa/Desktop/JYP-Tubes-IF2224\$ make run ARGS=test/milestone-1/tc6.pas ./bin/main test/milestone-1/tc6.pas </pre>
Bukti Output
<pre> noumisyifa@DESKTOP-SI02VGQ:/mnt/c/Users/Syifa/Desktop/JYP-Tubes-IF2224\$ make run ARGS=test/milestone-1/tc6.pas ./bin/main test/milestone-1/tc6.pas Lexical Error: Invalid token '@' at position 136 </pre>

D. KESIMPULAN DAN SARAN

1. Kesimpulan

Deterministic Finite Automata (DFA) berhasil diimplementasikan sebagai basis aturan dari *lexical analyzer* untuk bahasa Pascal-S. DFA terbukti mampu mengenali pola karakter dan mengubah source code menjadi token-token yang telah ditentukan, terkecuali proses *error handling* yang membutuhkan tambahan proses dalam kodenya secara langsung. Selain itu, Penggunaan representasi DFA dalam bentuk file JSON memungkinkan fleksibilitas dalam modifikasi aturan tanpa perlu mengubah kode program secara langsung.

Sistem error handling yang diimplementasikan menggunakan pendekatan *bail-out strategy*, di mana lexer mendeteksi error leksikal pertama dan langsung menghentikan proses tokenisasi. Meskipun pendekatan ini sederhana, namun efektif. Error yang dapat dideteksi meliputi invalid identifiier (dimulai dengan angka atau mengandung karakter ilegal) dan operator/symbol yang tidak dikenal.

2. Saran

E. LAMPIRAN

3. Link *Repository* Github

Berikut terlampir link *repository* github untuk tugas compiler Pascal-S kelompok JYP K-02:

<https://github.com/numshv/JYP-Tubes-IF2224>

4. Link *Workspace* Diagram DFA

Berikut terlampir link *workspace* diagram DFA tugas pembuatan *lexer* compiler Pascal-S kelompok JYP K-02:

[pranala ke workspace](#)

5. Pembagian Tugas

Berikut terlampir pembagian tugas tiap anggota kelompok JYP K-02 pada *milestone 1*:

NIM	Nama	Tugas	Persentase
13523058	Noumisyifa Nabila Nareswari	- Membantu dalam menyusun DFA - Membuat laporan	20%
13523066	M. Ghifary Komara Putra	- Mengerjakan implementasi kode bagian engine DFA	20%
13523072	Sabilul Huda	- Memimpin penyusunan rule DFA dan graft	20%
13523080	Diyah Susan Nugrahani	- Memimpin penyusunan rule DFA dan graft	20%
13523108	Henry Filbert Shinelu	- Mengerjakan implementasi kode bagian engine DFA	20%