Milestone 1
IF2224 Teori Bahasa Formal dan Otomata

Lexical Analysis

Kelompok MOE

Disusun oleh:

Ivant Samuel Silaban	13523129
Rafa Abdussalam Danadyaksa	13523133
Muhamad Nazih Najmudin	13523144
Anas Ghazi Al Gifari	13523159
Muhammad Rizain Firdaus	13523164

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

2025

DAFTAR ISI

BAB I
LANDASAN TEORI

1.1 Lexical Analysis

Bahasa pemrograman adalah bahasa yang selalu digunakan untuk melakukan operasi-operasi yang melibatkan komputasi. Terkait bahasa pemrograman ini, proses untuk mengeksekusi bahasa pemrograman tersebut tidaklah mudah. Terdapat banyak proses yang perlu dilakukan oleh *compiler* supaya kode yang ditulis benar dan bisa memberikan output yang sesuai dengan bahasa pemrograman yang ditulis. Tahapan pertama yang akan dikerjakan dalam tugas besar mata kuliah IF2224 Teori Bahasa Formal dan Otomata ini adalah membuat *lexical analyzer* (lexer)

Analisis leksikal (lexical analysis) adalah tahap pertama dalam proses kompilasi atau interpretasi bahasa pemrograman. Pada tahap ini, *compiler* membaca kode sumber mentah yang pada dasarnya hanyalah rangkaian karakter dan mengubahnya menjadi rangkaian satuan makna yang disebut token. Token merupakan unit terkecil yang memiliki arti dalam sebuah program, misalnya kata kunci (keyword), nama variabel (identifier), operator, angka, atau tanda baca.

Tujuan utama dari analisis leksikal adalah untuk mempermudah tahap berikutnya, yaitu parsing. Alih-alih berurusan langsung dengan karakter mentah, tahap parsing bekerja dengan kumpulan token yang sudah terstruktur, sehingga lebih mudah memahami susunan logis dari program. Pada proses ini, analisis leksikal juga membuang bagian yang tidak diperlukan seperti spasi, tab, dan komentar, serta dapat mendeteksi kesalahan seperti simbol yang tidak dikenal sejak awal.

Proses ini dilakukan oleh komponen yang disebut lexer. Lexer membaca kode sumber dari kiri ke kanan dan menggunakan pola tertentu untuk mengenali jenis token yang berbeda-beda. Ketika lexer menemukan urutan karakter yang cocok dengan salah satu pola tersebut, ia membuat sebuah token yang biasanya berisi dua hal utama: jenis token dan nilai token.

1.2 Token

Token adalah unit terkecil dalam program yang memiliki makna. Setiap token memiliki dua komponen utama, yaitu jenis token dan nilai token. Jenis token menunjukkan kategori token, sedangkan nilai token menyimpan nilai aktual atau atribut dari *lexeme*.

Pada tahap analisis leksikal, kode sumber Pascal-S akan diuraikan menjadi token. Berikut ini merupakan daftar jenis token dan nilainya dalam bahasa Pascal-S.

a. Keywords
   Keyword adalah kata kunci yang sudah didefinisikan dan memiliki fungsi khusus dalam struktur program. Contohnya adalah program, var, begin, end, if, then, else, while, do, for, to, dan downto.

b. Identifiers
Identifier adalah nama yang diciptakan oleh pengguna untuk memberikan nama pada variabel, prosedur, fungsi, dan elemen pemrograman lainnya. *Identifier* sendiri terdiri dari rantai karakter (huruf, angka, *underscore*) yang harus berbeda dengan *keyword*. Contohnya adalah `x`, `y`, `z`, `sum`, `avg`, dan `count`.

c. Operators
Operator adalah simbol atau kata yang menunjukkan operasi tertentu pada data. Kategori operator sendiri mencakup
- Arithmetic Operator: `+`, `-`, `*`, `/`, `div`, `mod`.
- Relational Operator: `=`, `<>`, `<`, `<=`, `>`, `>=`.
- Logical Operator: `and`, `or`, `not`.
- Assignment Operator: `:=`.

d. Literals
Literal adalah nilai konstan yang secara langsung dituliskan dalam kode program. Kategori *literal* sendiri mencakup
- Number: `22`, `3`, `2018`, `3.14`, `0.001`.
- Character Literal: `'a'`, `'b'`, `'c'`.
- String Literal: `'tbfo'`, `'seru sekali'`, `'moe moe kyun'`.

e. Delimiters
Delimiter adalah simbol yang digunakan untuk memisahkan struktur dalam kode program, Simbol-simbol ini sendiri mencakup
Semicolon (`;`), Comma (`,`), Colon (`:`), Dot (`.`), Left Parenthesis `(`, Right Parenthesis `)`, Left Bracket `[`, Right Bracket `]`, Range Operator (`..`).

Contoh berikut menunjukkan hasil proses tokenisasi terhadap satu baris kode sumber dalam bahasa Pascal-S. Pada contoh ini, setiap komponen dalam kode dikenali dan dikategorikan sesuai dengan jenisnya.

```
program HelloWorld;
```

Baris kode di atas kemudian diproses oleh *lexer* untuk mengenali setiap bagian penyusunnya, sehingga menghasilkan daftar token sebagai berikut.

```
KEYWORD(program)
IDENTIFIER(HelloWorld)
SEMICOLON(;)
```

Dari hasil tersebut, terlihat bahwa *lexer* mengenali program sebagai *keyword*, HelloWorld sebagai *identifier*, dan tanda titik koma ; sebagai token *semicolon* yang berfungsi sebagai *delimiter* penutup pernyataan.

1.3 Deterministic Finite Automata (DFA)

DFA adalah automaton yang selalu berada tepat pada satu *state* tertentu setelah membaca urutan *input* apapun. Istilah *deterministic* menunjukkan bahwa untuk setiap input, automaton hanya dapat bertransisi ke tepat satu *state* tertentu dari *state*-nya sekarang. Hal ini berbeda dengan NFA yang dapat berada dalam beberapa *state* secara bersamaan.

DFA digunakan untuk mengenali bahasa-bahasa formal yang dapat dihasilkan oleh suatu tata bahasa tertentu. Secara formal, DFA didefinisikan sebagai 5-*tuple*

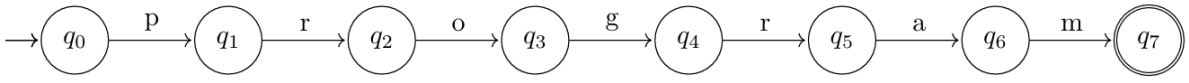
$$M = (Q, \Sigma, \delta, q_0, F)$$

dengan

- Q adalah himpunan berhingga dari *state*.
- Σ adalah himpunan berhingga dari simbol *input*.
- $\delta: Q \times \Sigma \rightarrow Q$ adalah fungsi transisi.
- $q_0 \in Q$ adalah *state* awal.
- $F \subseteq Q$ adalah himpunan *state* akhir.

DFA memulai proses dari *state* awal q_0, kemudian membaca simbol-simbol input di Σ satu per satu. DFA bergerak dari satu *state* ke *state* lain sesuai fungsi transisi δ. Setelah seluruh *input* dibaca, DFA akan berada pada salah satu *state*. Jika DFA berakhir pada salah satu *state* akhir di F, input diterima dan dianggap sebagai bagian dari bahasa yang dikenali DFA. Jika tidak, *input* ditolak.

Sebagai contoh, DFA dapat dirancang untuk mengenali *keyword* tertentu dalam kode sumber, misalnya keyword program. DFA tersebut memiliki tujuh transisi yang merepresentasikan setiap karakter dari kata tersebut. Proses dimulai dari state awal q_0, kemudian DFA membaca simbol demi simbol sesuai urutan input.



Ketika simbol pertama p dibaca, DFA berpindah dari q_0 ke q_1. Selanjutnya, simbol r menyebabkan transisi dari q_1 ke q_2, diikuti oleh simbol o yang membawa DFA ke q_3. Simbol

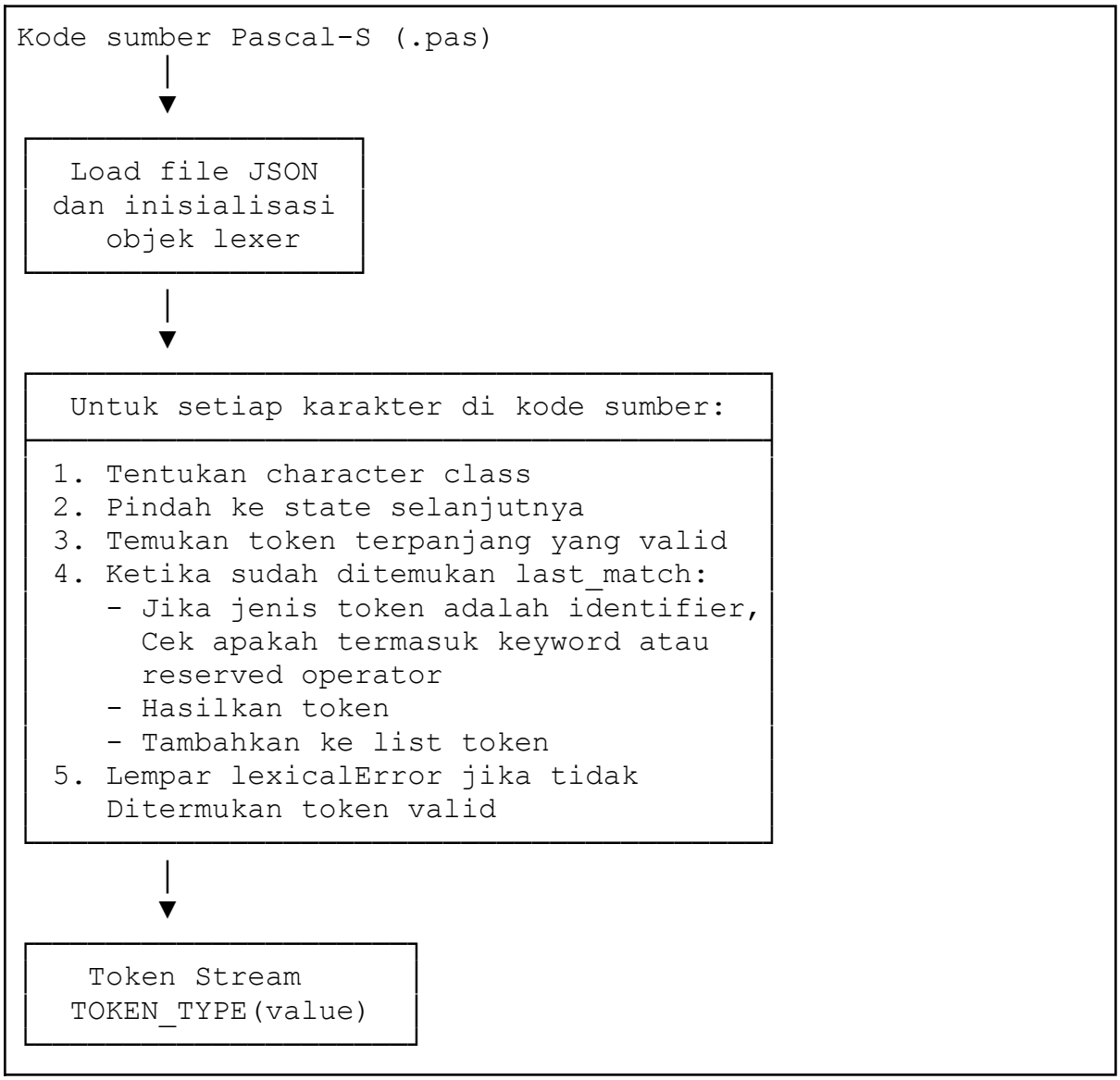
g kemudian mengubah state menjadi q_4, dan simbol r berikutnya membawa DFA ke q_5. Setelah membaca a, DFA berpindah ke q_6, dan terakhir simbol m membawa DFA ke q_7.

Jika DFA mencapai state q_7 yang merupakan *state* akhir ($q_7 \in F$), string program dikenali sebagai *keyword* yang valid. Apabila terdapat kesalahan dalam urutan simbol, DFA tidak akan mencapai *state* akhir dan string tersebut akan ditolak.

BAB II

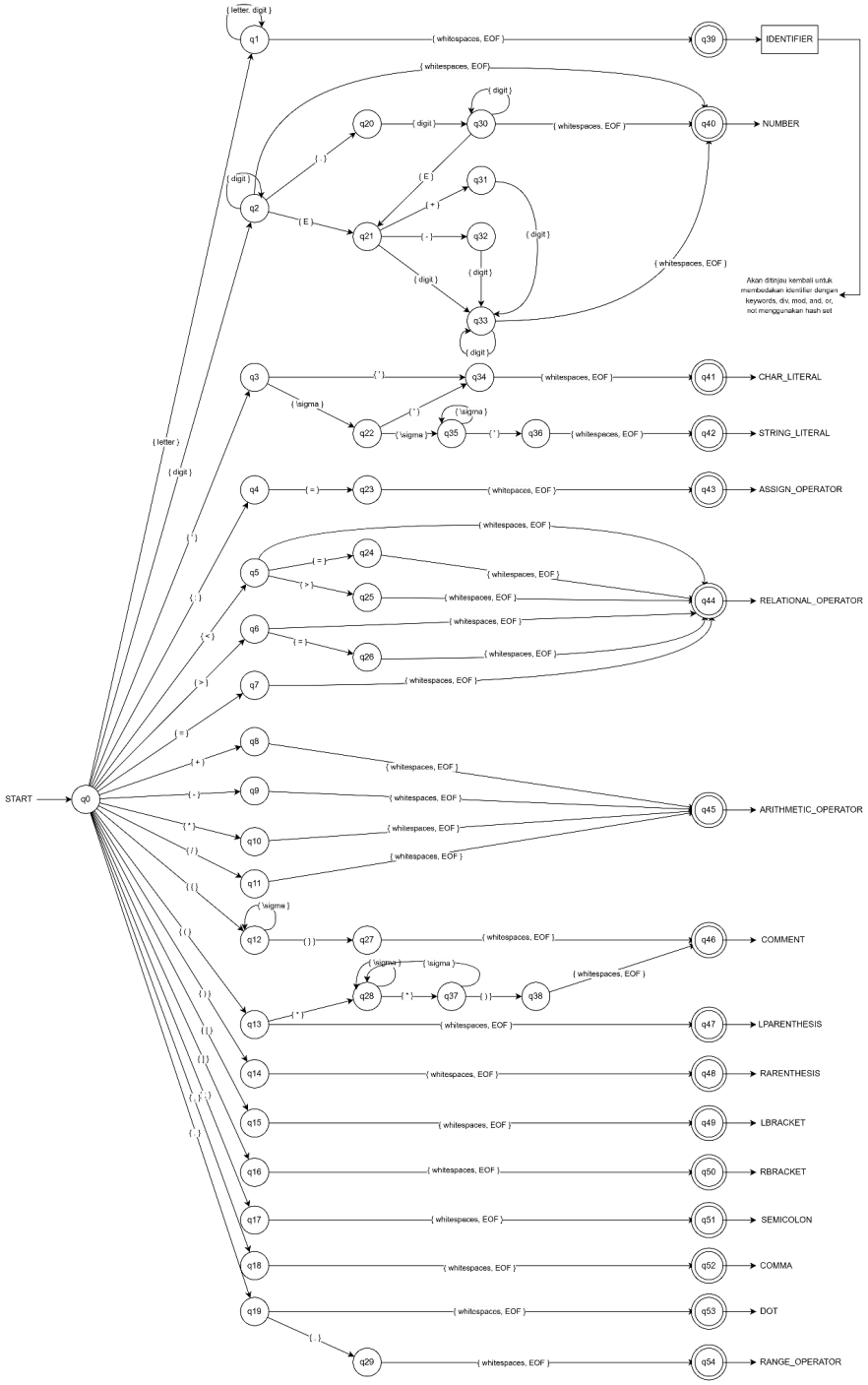
PERANCANGAN DAN IMPLEMENTASI

2.1 Arsitektur Sistem



2.2 Diagram DFA

2.2.1 Gambaran DFA



2.2.2 Definisi State

Proses dimulai dari *state* awal, yaitu S_START yang berperan sebagai titik masuk untuk semua *token recognition*. Dari *state* ini, DFA akan memeriksa karakter

pertama dari input, lalu bertransisi ke *state* berikutnya sesuai dengan aturan transisi yang telah ditetapkan. *Lexer* akan menelusuri urutan karakter untuk membentuk token terpanjang sebelum menentukan jenis token yang sesuai.

Kode	**State**	**Jenis Token**	**Deskripsi**
q39	S_IDENTIFIER	IDENTIFIER	State Final tercapai setelah membaca urutan huruf/angka yang diawali huruf
q40	S_INTEGER	NUMBER	State Final tercapai setelah membaca satu atau lebih digit
q40	S_REAL_FRAC	NUMBER	State Final tercapai setelah membaca angka dengan bagian desimal
q40	S_REAL_EXP_SIGNED	NUMBER	State Final tercapai setelah membaca
q40	S_REAL_EXP_DIGIT	NUMBER	State Final tercapai setelah membaca angka dalam notasi saintifik lengkap
q41	S_CHAR_QUOTE_END	CHAR_LITERAL	State Final tercapai setelah membaca literal karakter yang valid diapit kutip
q41	S_EMPTY_STRING_END	CHAR_LITERAL	State Final tercapai setelah membaca literal string kosong
q42	S_STRING_QUOTE_END	STRING_LITERAL	State Final tercapai setelah membaca literal string diakhiri kutip penutup
q43	S_ASSIGN	ASSIGN_OPERATOR	State Final tercapai setelah membaca operator penugasan
q45	S_PLUS	ARITHMETIC_OPERATOR	State Final tercapai setelah membaca operator tambah
q45	S_MINUS	ARITHMETIC_OPERATOR	State Final tercapai setelah membaca operator kurang
q45	S_STAR	ARITHMETIC_OPERATOR	State Final tercapai setelah membaca operator kali

q45	S_SLASH	ARITHMETIC_OPERATOR	State Final tercapai setelah membaca operator bagi
q44	S_EQUAL	RELATIONAL_OPERATOR	State Final tercapai setelah membaca operator sama dengan
q44	S_NOT_EQUAL	RELATIONAL_OPERATOR	State Final tercapai setelah membaca operator tidak sama dengan
q44	S_LESS	RELATIONAL_OPERATOR	State Final tercapai setelah membaca operator kurang dari
q44	S_LESS_EQUAL	RELATIONAL_OPERATOR	State Final tercapai setelah membaca operator kurang dari atau sama dengan
q44	S_GREATER	RELATIONAL_OPERATOR	State Final tercapai setelah membaca operator lebih dari
q44	S_GREATER_EQUAL	RELATIONAL_OPERATOR	State Final tercapai setelah membaca operator lebih dari atau sama dengan
q51	S_SEMICOLON	SEMICOLON	State Final tercapai setelah membaca tanda titik koma (;)
q52	S_COMMA	COMMA	State Final tercapai setelah membaca tanda koma (,)
q	S_COLON	COLON	State Final tercapai setelah membaca tanda titik dua (:)
q53	S_DOT	DOT	State Final tercapai setelah membaca tanda titik (.)
q54	S_RANGE	RANGE_OPERATOR	State Final tercapai setelah membaca operator rentang (..)
q47	S_LPAREN	LPARENTHESIS	State Final tercapai setelah membaca tanda kurung buka (.
q48	S_RPAREN	RPARENTHESIS	State Final tercapai setelah membaca tanda kurung tutup).
q49	S_LBRACKET	LBRACKET	State Final tercapai setelah membaca kurung siku buka

q50	S_RBRACKET	RBRACKET	State Final tercapai setelah membaca kurung siku tutup
q46	S_COMMENT_BLOCK_END	COMMENT	State Final tercapai setelah membaca kurung kurawal tutup } yang mengakhiri blok komentar
q46	S_COMMENT_LINE_END	COMMENT	State Final tercapai setelah membaca *) yang mengakhiri komentar baris

2.2.3 Aturan Transisi

	letter	digit	e/E	'	:	<	>	=	+	-	*	/	{	}	(	)	[	]	;	,	.	space	EOF
→q0	q1	q2		q3	q4	q5	q6	q7	q8	q9	q10	q11	q12		q13	q14	q15	q16	q17	q18	q19	q0	q0
q1	q1	q1																				q39	q39
q2		q2	q21																		q20	q40	q40
q3	q22	q22	q22	q24	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	q22	
q4								q23															
q5							q25	q24														q44	q44
q6								q26														q44	q44
q7																						q44	q44
q8																						q45	q45
q9																						q45	q45
q10																						q45	q45
q11																						q45	q45
q12	q12	q12	q12	q12	q12	q12	q12	q12	q12	q12	q12	q12	q12	q27	q12	q12	q12	q12	q12	q12	q12	q12	
q13											q28											q47	q47
q14																						q48	q48
q15																						q49	q49
q16																						q50	q50
q17																						q51	q51
q18																						q52	q52
q19																					q29	q53	q53
q20		q30																					
q21		q33							q31	q32													
q22	q35	q35	q35	q34	q35	q35	q35	q35	q35	q35	q35	q35	q35	q35	q35	q35	q35	q35	q35	q35	q35	q35	
q23																						q43	243

q24																					q44	q44
q25																					q44	q44
q26																					q44	q44
q27																					q46	q46
q28	q28	q28	q28	q28	q28	q28	q28	q28	q28	q28	q37	q28	q28	q28	q28	q28	q28	q28	q28	q28	q28	
q29																					q54	q54
q30		q30																			q40	q40
q31		q33																				
q32		q33																				
q33		q33																			q40	q40
q34																						q41
q35	q35	q35	q35	q36	q35	q35	q35	q35	q35	q35	q35	q35	q35	q35	q35	q35	q35	q35	q35	q35	q35	
q36																					q42	q42
q37	q28	q28	q28	q28	q28	q28	q28	q28	q28	q28	q28	q28	q28	q28	q28	q38	q28	q28	q28	q28	q28	
q38																					q46	q46
*q39																						q0
*q40																						q0
*q41																						q0
*q42																						q0
*q43																						q0
*q44																						q0
*q45																						q0
*q46																						q0
*q47																						q0
*q48																						q0
*q49																						q0
*q50																						q0
*q51																						q0
*q52																						q0
*q53																						q0
*q54																						q0

2.3 Format Konfigurasi DFA (JSON)

```
{
  "character_classes": {
    "letter": "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ",
    "digit": "0123456789",
    "whitespace": " \t\n\r"
  },
  "keywords": {
    "program": "KEYWORD",
    "var": "KEYWORD",
```

```
    "begin": "KEYWORD",
    "end": "KEYWORD",
    "if": "KEYWORD",
    "then": "KEYWORD",
    "else": "KEYWORD",
    "while": "KEYWORD",
    "do": "KEYWORD",
    "for": "KEYWORD",
    "to": "KEYWORD",
    "downto": "KEYWORD",
    "integer": "KEYWORD",
    "real": "KEYWORD",
    "boolean": "KEYWORD",
    "char": "KEYWORD",
    "array": "KEYWORD",
    "of": "KEYWORD",
    "procedure": "KEYWORD",
    "function": "KEYWORD",
    "const": "KEYWORD",
    "type": "KEYWORD",
    "true": "KEYWORD",
    "false": "KEYWORD"
  },
  "reserved_operators": {
    "div": "ARITHMETIC_OPERATOR",
    "mod": "ARITHMETIC_OPERATOR",
    "and": "LOGICAL_OPERATOR",
    "or": "LOGICAL_OPERATOR",
    "not": "LOGICAL_OPERATOR"
  },
  "start_state": "S_START",
  "final_states": {
    "S_IDENTIFIER": "IDENTIFIER",
    "S_INTEGER": "NUMBER",
    "S_REAL_FRAC": "NUMBER",
    "S_REAL_EXP_SIGNED": "NUMBER",
    "S_REAL_EXP_DIGIT": "NUMBER",
    "S_CHAR_QUOTE_END": "CHAR_LITERAL",
    "S_EMPTY_STRING_END": "CHAR_LITERAL",
    "S_STRING_QUOTE_END": "STRING_LITERAL",
    "S_ASSIGN": "ASSIGN_OPERATOR",
    "S_PLUS": "ARITHMETIC_OPERATOR",
    "S_MINUS": "ARITHMETIC_OPERATOR",
    "S_STAR": "ARITHMETIC_OPERATOR",
    "S_SLASH": "ARITHMETIC_OPERATOR",
    "S_EQUAL": "RELATIONAL_OPERATOR",
    "S_NOT_EQUAL": "RELATIONAL_OPERATOR",
    "S_LESS": "RELATIONAL_OPERATOR",
    "S_LESS_EQUAL": "RELATIONAL_OPERATOR",
    "S_GREATER": "RELATIONAL_OPERATOR",
    "S_GREATER_EQUAL": "RELATIONAL_OPERATOR",
    "S_SEMICOLON": "SEMICOLON",
    "S_COMMA": "COMMA",
    "S_COLON": "COLON",
    "S_DOT": "DOT",
    "S_RANGE": "RANGE_OPERATOR",
    "S_LPAREN": "LPARENTHESIS",
    "S_RPAREN": "RPARENTHESIS",
    "S_LBRACKET": "LBRACKET",
    "S_RBRACKET": "RBRACKET",
    "S_COMMENT_BLOCK_END": "COMMENT",
    "S_COMMENT_LINE_END": "COMMENT"
  },
  "transitions": {
    "S_START": {
      "letter": "S_IDENTIFIER",
      "digit": "S_INTEGER",
      "'": "S_STRING_CHAR_START",
      "{": "S_COMMENT_BLOCK_CONTENT",
      ":": "S_COLON",
      "<": "S_LESS",
      ">": "S_GREATER",
      ".": "S_DOT",
      "(": "S_LPAREN",
      ")": "S_RPAREN",
      "[": "S_LBRACKET",
      "]": "S_RBRACKET",
      ";": "S_SEMICOLON",
      ",": "S_COMMA",
```

```
    "=": "S_EQUAL",
    "+": "S_PLUS",
    "-": "S_MINUS",
    "*": "S_STAR",
    "/": "S_SLASH"
  },
  "S_IDENTIFIER": {
    "letter": "S_IDENTIFIER",
    "digit": "S_IDENTIFIER"
  },
  "S_INTEGER": {
    "digit": "S_INTEGER",
    ".": "S_REAL_DOT",
    "E": "S_REAL_EXP"
  },
  "S_REAL_DOT": {
    "digit": "S_REAL_FRAC"
  },
  "S_REAL_FRAC": {
    "digit": "S_REAL_FRAC",
    "E": "S_REAL_EXP"
  },
  "S_REAL_EXP": {
    "+": "S_REAL_EXP_SIGNED",
    "-": "S_REAL_EXP_SIGNED",
    "digit": "S_REAL_EXP_DIGIT"
  },
  "S_REAL_EXP_SIGNED": {
    "digit": "S_REAL_EXP_DIGIT"
  },
  "S_REAL_EXP_DIGIT": {
    "digit": "S_REAL_EXP_DIGIT"
  },
  "S_STRING_CHAR_START": {
    "'": "S_EMPTY_STRING_END",
    "any_except_'": "S_STRING_CHAR_CONTENT"
  },
  "S_STRING_CHAR_CONTENT": {
    "'": "S_CHAR_QUOTE_END",
    "any_except_'": "S_STRING_CONTENT"
  },
  "S_STRING_CONTENT": {
    "'": "S_STRING_QUOTE_END",
    "any_except_'": "S_STRING_CONTENT"
  },
  "S_CHAR_QUOTE_END": {},
  "S_STRING_QUOTE_END": {},
  "S_EMPTY_STRING_END": {},
  "S_COMMENT_BLOCK_CONTENT": {
    "}": "S_COMMENT_BLOCK_END",
    "any_except_}": "S_COMMENT_BLOCK_CONTENT"
  },
  "S_COMMENT_BLOCK_END": {},
  "S_LPAREN": {
    "*": "S_COMMENT_LINE_CONTENT"
  },
  "S_COMMENT_LINE_CONTENT": {
    "*": "S_COMMENT_LINE_STAR_END",
    "any_except_*": "S_COMMENT_LINE_CONTENT"
  },
  "S_COMMENT_LINE_STAR_END": {
    ")": "S_COMMENT_LINE_END",
    "any_except_)": "S_COMMENT_LINE_CONTENT"
  },
  "S_COMMENT_LINE_END": {},
  "S_COLON": {
    "=": "S_ASSIGN"
  },
  "S_LESS": {
    "=": "S_LESS_EQUAL",
    ">": "S_NOT_EQUAL"
  },
  "S_GREATER": {
    "=": "S_GREATER_EQUAL"
  },
  "S_DOT": {
    ".": "S_RANGE"
  },
  "S_ASSIGN": {},
```

```
    "S_PLUS": {},
    "S_MINUS": {},
    "S_STAR": {},
    "S_SLASH": {},
    "S_EQUAL": {},
    "S_NOT_EQUAL": {},
    "S_LESS_EQUAL": {},
    "S_GREATER_EQUAL": {},
    "S_SEMICOLON": {},
    "S_COMMA": {},
    "S_RANGE": {},
    "S_RPAREN": {},
    "S_LBRACKET": {},
    "S_RBRACKET": {}
  }
}
```

2.4 Implementasi Algoritma

2.4.1 Struktur File

```
MOE-Tubes-IF2224/
├── doc/
│   ├── Laporan-1-MOE.pdf
│   └── Diagram-1-MOE.png
├── src/
│   ├── dfa.json
│   └── lexer.py
├── test/
│   └── milestone-1/
│       ├── input-1.pas
│       ├── input-2.pas
│       ├── input-3.pas
│       ├── input-4.pas
│       ├── input-5.pas
│       ├── input-6.pas
│       ├── input-7.pas
│       ├── input-8.pas
│       ├── input-9.pas
│       ├── input-10.pas
│       ├── input-11.pas
│       ├── input-12.pas
│       ├── output-1.txt
│       ├── output-9.txt
│       ├── output-11.txt
│       └── output-12.txt
└── README.md
```

2.4.2 Kelas dan Fungsi

a. Kelas LexicalError

i. Fungsi __init__()

```
   def __init__(self, message, line, column):
        super().__init__(f"Lexical Error on line
{line}, column {column}: {message}")
        self.message = message
        self.line = line
        self.column = column
```

Ini adalah fungsi dari kelas turunan Exception yang digunakan untuk error leksikal saat proses tokenisasi kode sumber.

b. Kelas Lexer

i. Fungsi __init__

```
   def __init__(self, dfa_rules_path):
        """
        Inisialisasi lexer dengan memuat aturan DFA
dari file JSON.
        """
        try:
            with open(dfa_rules_path, 'r') as f:
                self.dfa = json.load(f)
        except FileNotFoundError:
            print(f"FATAL ERROR: dfa.json tidak
ditemukan di '{dfa_rules_path}'", file=sys.stderr)
            sys.exit(1)

        self.start_state = self.dfa['start_state']
        self.final_states = self.dfa['final_states']
        self.transitions = self.dfa['transitions']
        self.char_classes =
self.dfa['character_classes']

        # Daftar keywords dan reserved words dari DFA
(dalam lowercase)
        self.keywords = self.dfa.get('keywords', {})
        self.reserved_operators =
self.dfa.get('reserved_operators', {})
```

Ini adalah fungsi init dari kelas Lexer. Fungsi ini bertugas untuk menginisialisasi objek lexer dengan memuat aturan DFA dari file JSON. Tujuannya adalah agar objek lexer siap digunakan untuk proses tokenisasi berdasarkan aturan yang ada di file DFA. Program awalanya akan memabca file JSON yang berisi aturan DFA.

Jika file tidak ditemukan, program akan menampilkan pesan error dan langsung keluar. Program kemudian menyimpan komponen DFA ke atribut objek seperti:

- start_state untuk menyimpan state awal DFA.
- final_state untuk menyimpan daftar state final beserta tipe tokennya.
- transition untuk menyimapn aturan transisi antar state
- char_classes untuk pengelompokkan karakter seperti letter, digit, dan whitespace
- keywords dan reserved_operators sebagai mapping untuk kata kunci dan operator.

ii. Fungsi _get_char_class()

```
   def _get_char_class(self, char):
        """Mendapatkan kelas karakter (letter, digit,
dll.) dari sebuah karakter."""
        for class_name, chars in
self.char_classes.items():
            if char in chars:
                return class_name
        return char
```

Fungsi ini digunakan untuk menentukan kelas karakter dari sebuah karakter input, misalkan apakah karakter tersebut termasuk letter, digit, whitespace, dll.

iii. Fungsi tokenize()

1. Inisialisasi

```
   def tokenize(self, source_code):
        """
        Memproses source code dan mengubahnya
menjadi daftar token.
        Melempar LexicalError jika ada
kesalahan.
        """
        tokens = []
        position = 0
        line = 1
        column = 1
```

Bagian awal dari fungsi ini akan membuat list kosong tokens untuk menampung hasil tokenisasi, kemudian menyiapkan variabel posisi

(position), baris(line), dan kolom (column) untuk melacak lokasi karakter saat proses berjalan.

2. Main Loop

```
while position < len(source_code):
    char = source_code[position]
    # Jika karakter adalah whitespace, lewati
dan perbarui posisi
    if char in
self.char_classes.get('whitespace', ''):
        if char == '\n':
            line += 1
            column = 1
        else:
            column += 1
        position += 1
        continue
    # Mulai dari state awal DFA
    current_state = self.start_state
    current_lexeme = ""
    last_match = None
    start_line, start_col = line, column
    temp_pos = position
    temp_line, temp_col = line, column
    # Inner Loop: Simulasi DFA untuk menemukan
token terpanjang
    while True:
        # Cek apakah EOF?
        if temp_pos >= len(source_code):
            # Error handling untuk string yang
tidak ditutup
            if current_state in
["S_STRING_CONTENT", "S_STRING_QUOTE_END"]:
                ...
```

Bagian dari kode ini akan melakukan iterasi satu per satu karakter dari source_code. Jika karakter adalah whitespace, posisi dan baris/kolom diperbarui, lalu lanjut ke karakter selanjutnya

3. Handle Komentar

```
            # Handle komentar blok { ... }
            if char == '{':
```

```
                position += 1
                column += 1
                while position <
len(source_code) and source_code[position] !=
'}':
                    if source_code[position] ==
'\n':
                        line += 1
                        column = 1
                    else:
                        column += 1
                    position += 1
                # Skip closing }
                if position < len(source_code):
                    position += 1
                    column += 1
                continue

            # Handle komentar line (* ... *)
            if char == '(' and position + 1 <
len(source_code) and source_code[position + 1]
== '*':
                position += 2
                column += 2
                while position + 1 <
len(source_code):
                    if source_code[position] ==
'*' and source_code[position + 1] == ')':
                        position += 2
                        column += 2
                        break
                    if source_code[position] ==
'\n':
                        line += 1
                        column = 1
                    else:
                        column += 1
                    position += 1
                continue
```

Saat menemukan karakter { atau (*, lexer langsung memajukan posisi dan melakukan loop untuk melewati semua karakter sampai menemukan } atau *) dan kemudian dimajukan lagi dan lanjut ke karakter selanjutnya.

4. Simulasi DFA (Inner Loop)

```
while True:
    # Cek apakah EOF?
    if temp_pos >= len(source_code):
        ...
    # Ambil karakter dan tentukan kelasnya
    char = source_code[temp_pos]
    char_class = self._get_char_class(char)
    next_state = None
    # Pencarian transisi DFA
    if current_state in self.transitions:
        possible_transitions =
self.transitions[current_state]
        if char_class in possible_transitions:
            next_state =
possible_transitions[char_class]
        elif char in possible_transitions:
            next_state =
possible_transitions[char]
        else:
            for rule, target_state in
possible_transitions.items():
                if
rule.startswith("any_except_"):
                    excluded_chars =
rule.split('_')[-1]
                    if char not in
excluded_chars:
                        next_state =
target_state
                        break
    # Pemrosesan hasil transisi
    if next_state:
        current_state = next_state
        current_lexeme += char
        if char == '\n':
            temp_line += 1
            temp_col = 1
        else:
            temp_col += 1
        temp_pos += 1
        if current_state in self.final_states:
            last_match = (current_lexeme,
self.final_states[current_state], temp_line,
temp_col)
    else:
```

```
        break
```

Ini adalah proses tokenisasi. Dimulai dari state awal DFA, kemudian program menyiapkan variabel untuk menyimpan lexeme (potongan karakter yang diproses) dan token valid terakhir (last_match). Program menggunakan pointer sementara (temp_pos, temp_line, temp_col) untuk simulasi DFA (Inner Loop) tanpa mengubah pointer utama. Pointer sementara ini akan digunakan untuk menjelajahi source code, sedangkan pointer yang diinisialisasi di awal tadi hanya untuk menandai satu token yang valid.
Simulasi DFA (Inner Loop) ini bertujuan untuk menemukan token terpanjang yang valid. Program akan mengecek transisi DFA berdasarkan kelas karakter atau karakter literal. Jika ada transisi, pindah ke state berikutnya, menambahkan karakter ke lexeme, dan memajukan pointer sementara. Jika state yang dicapai adalah final, simpan lexeme dan tipe token di last_match. Jika tidak ada transisi valid, keluar dari loop.

5. Finalisasi dan Error Handling

```
# Finalisasi token setelah inner loop
if last_match:
    lexeme, token_type, end_line, end_col =
last_match
    if token_type == "IDENTIFIER":
        lexeme_lower = lexeme.lower()
        if lexeme_lower in self.keywords:
            token_type =
self.keywords[lexeme_lower]
        elif lexeme_lower in
self.reserved_operators:
            token_type =
self.reserved_operators[lexeme_lower]
    tokens.append((token_type, lexeme))
    position += len(lexeme)
    line, column = end_line, end_col
else:
    raise LexicalError(f"Invalid character
'{source_code[position]}'", line, column)
```

Jika ditemukan token valid (last_match), cek apakah token tersebut adalah identifier. Jika ya, cek apakah lexeme tersebut adalah keyword atau reserved operator, dan update tipe token jika harus diupdate. Kemudian, tambahkan token ke list tokens dan perbarui pointer utama ke posisi

setelah token yang ditemukan, serta baris dan kolom. Jika tidak ditemukan token valid, lempar lexicalError dengan informasi lokasi error.

6. Return

```
return tokens
```

Setelah semua karakter diproses, fungsi mengembalikan list token yang ditemukan.

iv. Fungsi main()

```
def main():
    """
    Fungsi utama untuk menjalankan lexer.
    Dapat menerima 1 argumen (output ke terminal) atau
2 argumen (output ke file):
    - python lexer.py input.pas -> output ke terminal
    - python lexer.py input.pas output.txt -> output
ke file
    """
    if len(sys.argv) < 2 or len(sys.argv) > 3:
        print("Usage:", file=sys.stderr)
        print("  python lexer.py
<source_file_path.pas> -> output ke terminal",
file=sys.stderr)
        print("  python lexer.py
<source_file_path.pas> <output_file_path.txt> ->
output ke file", file=sys.stderr)
        sys.exit(1)

    source_file_path = sys.argv[1]
    output_file_path = sys.argv[2] if len(sys.argv) ==
3 else None

    if not source_file_path.lower().endswith('.pas'):
        print(f"Input Error: Source file harus
memiliki ekstensi .pas. Diberikan:
'{source_file_path}'", file=sys.stderr)
        sys.exit(1)

    if output_file_path and not
output_file_path.lower().endswith('.txt'):
        print(f"Output Error: Output file harus
```

```
memiliki ekstensi .txt. Diberikan:
'{output_file_path}'", file=sys.stderr)
        sys.exit(1)

    script_dir =
os.path.dirname(os.path.abspath(__file__))
    dfa_path = os.path.join(script_dir, "dfa.json")

    try:
        with open(source_file_path, 'r') as f:
            source_code = f.read()
    except FileNotFoundError:
        print(f"Error: Input file tidak ditemukan di
'{source_file_path}'", file=sys.stderr)
        sys.exit(1)

    lexer = Lexer(dfa_path)

    try:
        tokens = lexer.tokenize(source_code)

        if output_file_path:
            # Output ke file
            output_dir =
os.path.dirname(output_file_path)
            if output_dir:
                os.makedirs(output_dir, exist_ok=True)

            with open(output_file_path, 'w') as f:
                for token_type, lexeme in tokens:

f.write(f"{token_type}({lexeme})\n")

            print(f"Tokenization successful. Output
written to '{output_file_path}'")
        else:
            # Output ke terminal
            print("Tokenization successful. Daftar
token:")
            print("=" * 40)
            for token_type, lexeme in tokens:
                print(f"{token_type}({lexeme})")

    except LexicalError as e:
        print(str(e), file=sys.stderr)
        sys.exit(1)
```

```
    except IOError as e:
        print(f"Error writing to file
'{output_file_path}': {e}", file=sys.stderr)
        sys.exit(1)
```

Fungsi main() adalah fungsi masuk utama program lexer. Fungsi ini menerima argumen dari command line untuk menentukan file source code pascal yang akan diproses dan opsional, yaitu file output untuk hasil tokenisasi. Jika argumen tidak sesuai, fungsi akan menampilkan petunjuk penggunaan dan keluar. Setelah validasi, fungsi akan membaca file input lalu membuat objek lexer dengan aturan DFA yang diambil dari file JSON. Selanjutnya, fungsi akan menjalankan tokenisasi. Jika berhasil, hasil token akan ditampilkan di terminal atau ke file output.

BAB III
PENGUJIAN

3.1 Rencana Pengujian

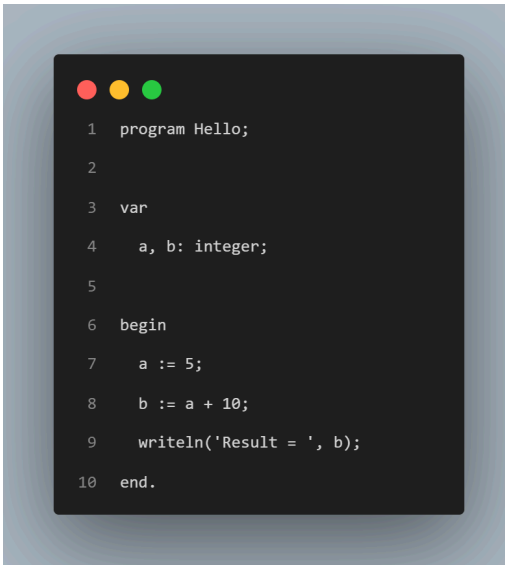
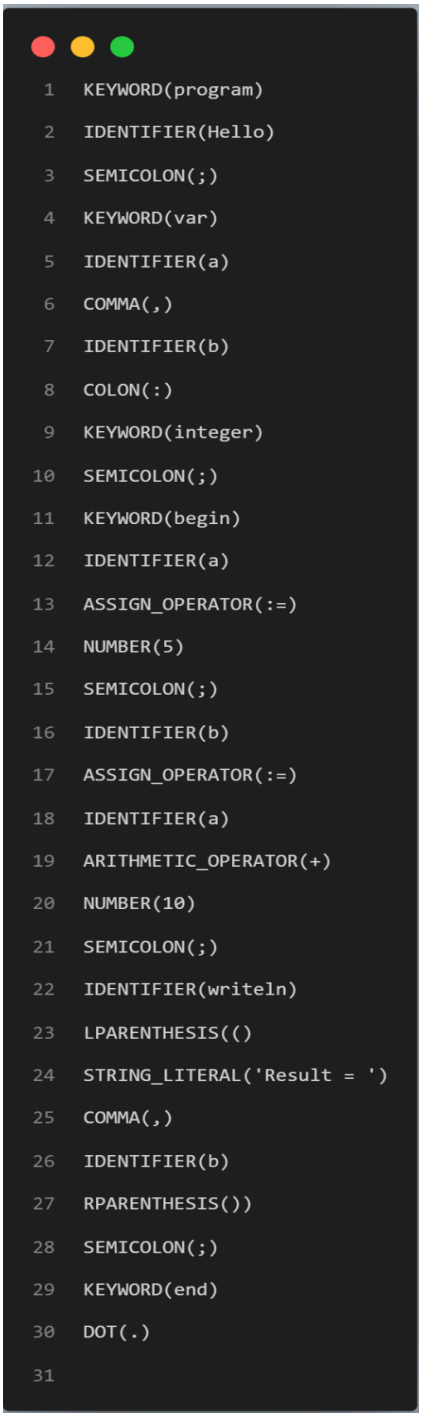
Pengujian dilakukan untuk memastikan bahwa sistem lexer berfungsi sesuai dengan spesifikasi yang telah ditetapkan. Setiap pengujian dirancang untuk mengamati bagaimana sistem mengenali berbagai jenis token berdasarkan kategori tertentu.

Secara keseluruhan, terdapat 9 *test case* yang disusun untuk mencakup seluruh aspek utama dari proses analisis leksikal. Berikut ini adalah masing-masing test case yang difokuskan pada area tertentu agar dapat mengevaluasi kemampuan sistem dalam mengenali pola token yang berbeda.

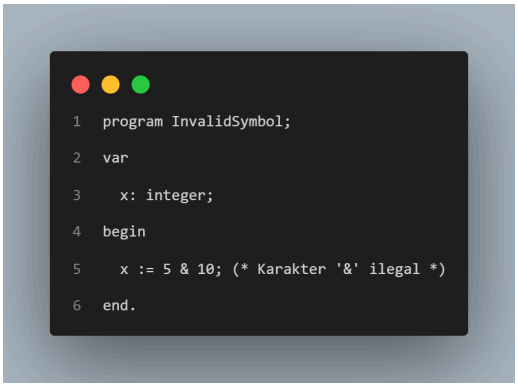
No	Test Case	Fokus Pengujian
1	Hello World	Keywords, identifiers, assign operator, number
2	Invalid Symbol	Identifikasi penggunaan simbol yang tidak ada
3	Start With Error	Error handling pada awal program yang tidak memenuhi (simbol/karakter tidak ada)
4	Unterminated	Kasus unterminated string (string belum tertutup)
5	E of In String	Kasus error end-of-file terutama pada deklarasi string/char yang belum tertutup
6	Bad Real	Number (syntax error belum dibuat karena tidak bersinggungan dengan milestone 1 ini)
7	Bad Exponent	Kasus invalid karakter (karakter “#” tidak dikenali)
8	Comprehensive Test	Keywords, identifiers, assign operators, numbers, arithmetic operators, logical operators, relational operators, string literals, character literals, parentheses, brackets, semicolons, commas, colons, dots, dan range operators
9	Boolean Test	Keywords, identifiers, assign operators, logical operators, string literals, parentheses, semicolons, commas, colons, dan dots

3.2 Test Case

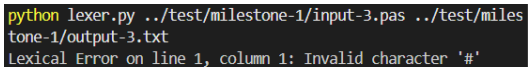
3.2.1 Test Case 1: Hello World

Input.pas	Output.txt
	

3.2.2 Test Case 2: Invalid Symbol

Input.pas	Output.txt
	

3.2.3 Test Case 3: Start With Error

Input.pas	Output.txt
	

3.2.4 Test Case 4: Unterminated

Input.pas	Output.txt

	python lexer.py ../test/milestone-1/input-4.pas ../test/milestone-1/output-4.txt Lexical Error on line 3, column 11: Unterminated string literal

3.2.5 Test Case 5: E of In String

Input.pas	Output.txt
	python lexer.py ../test/milestone-1/input-5.pas ../test/milestone-1/output-5.txt Lexical Error on line 5, column 8: Invalid character '''

3.2.6 Test Case 6: Bad Real

Input.pas	Output.txt

<table>
<tr>
<td>

```
1  program BadReal;
2  var
3    r: real;
4  begin
5    r := 1.2.3;
6  end.
```

</td>
<td>

```
1  KEYWORD(program)
2  IDENTIFIER(BadReal)
3  SEMICOLON(;)
4  KEYWORD(var)
5  IDENTIFIER(r)
6  COLON(:)
7  KEYWORD(real)
8  SEMICOLON(;)
9  KEYWORD(begin)
10 IDENTIFIER(r)
11 ASSIGN_OPERATOR(:=)
12 NUMBER(1.2)
13 DOT(.)
14 NUMBER(3)
15 SEMICOLON(;)
16 KEYWORD(end)
17 DOT(.)
18
```

</td>
</tr>
</table>

3.2.7 Test Case 7: Bad Exponent

Input.pas	Output.txt

<table>
<tr>
<td>

```
program BadExponent;
begin
  x := 1.2E#5; (* Karakter '#' ilegal setelah 'E' *)
end.
```

</td>
<td>

```
python lexer.py ../test/milestone-1/input-10.pas ../test/mi
lestone-1/output-10.txt
Lexical Error on line 3, column 12: Invalid character '#'
```

</td>
</tr>
</table>

3.2.8 Test Case 8: comprehensiveTest

<table>
<tr><th>Input.pas</th><th>Output.txt</th></tr>
<tr>
<td>

```
program comprehensiveTest;
{ Comprehensive Test Case for All PASCAL-S Tokens }
var
myInt, loopCounter: integer;
myReal            : real;
isReady           : boolean;
myInitial         : char;
myArray           : array[1..10] of char;

begin
(* Assignment dan semua operator aritmatika *)
myInt := (100 mod 3) + (20 * 3) div 2 - 1;
myReal := 123.456 / 2.0E-1;
isReady := true;
myInitial := 'Z';

(* Contoh string literal dan pemanggilan identifier 'writeln' *)
writeln('--- Starting Loop ---');

(* Loop dengan operator relasional dan logika lengkap *)
if (isReady = true) and (myInt >= 0) then
begin
loopCounter := 10;
while (loopCounter > 0) or (not isReady) do
begin
if (loopCounter <= 5) and (loopCounter <> 3) then
writeln('Looping...')
else
writeln('Still looping...');
loopCounter := loopCounter - 1;
end;
end;

(* Cek kondisi akhir, < digunakan di sini *)
if loopCounter < 1 then
myArray[1] := 'A';

end.
```

</td>
<td>

KEYWORD(program)
IDENTIFIER(comprehensiveTest)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(myInt)
COMMA(,)
IDENTIFIER(loopCounter)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(myReal)
COLON(:)
KEYWORD(real)
SEMICOLON(;)
IDENTIFIER(isReady)
COLON(:)
KEYWORD(boolean)
SEMICOLON(;)
IDENTIFIER(myInitial)
COLON(:)
KEYWORD(char)
SEMICOLON(;)
IDENTIFIER(myArray)
COLON(:)
KEYWORD(array)
LBRACKET([)
NUMBER(1)
RANGE_OPERATOR(..)
NUMBER(10)
RBRACKET(])
KEYWORD(of)
KEYWORD(char)
SEMICOLON(;)
KEYWORD(begin)

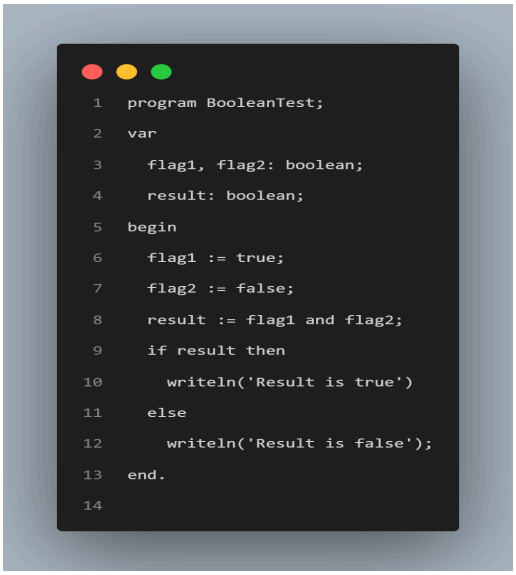
</td>
</tr>
</table>

	IDENTIFIER(myInt) ASSIGN_OPERATOR(:=) LPARENTHESIS(() NUMBER(100) ARITHMETIC_OPERATOR(mod) NUMBER(3) RPARENTHESIS()) ARITHMETIC_OPERATOR(+) LPARENTHESIS(() NUMBER(20) ARITHMETIC_OPERATOR(*) NUMBER(3) RPARENTHESIS()) ARITHMETIC_OPERATOR(div) NUMBER(2) ARITHMETIC_OPERATOR(-) NUMBER(1) SEMICOLON(;) IDENTIFIER(myReal) ASSIGN_OPERATOR(:=) NUMBER(123.456) ARITHMETIC_OPERATOR(/) NUMBER(2.0E-1) SEMICOLON(;) IDENTIFIER(isReady) ASSIGN_OPERATOR(:=) IDENTIFIER(true) SEMICOLON(;) IDENTIFIER(myInitial) ASSIGN_OPERATOR(:=) CHAR_LITERAL('Z') SEMICOLON(;) IDENTIFIER(writeln) LPARENTHESIS(() STRING_LITERAL('--- Starting Loop ---') RPARENTHESIS()) SEMICOLON(;) KEYWORD(if) LPARENTHESIS(() IDENTIFIER(isReady) RELATIONAL_OPERATOR(=) IDENTIFIER(true) RPARENTHESIS()) LOGICAL_OPERATOR(and) LPARENTHESIS(() IDENTIFIER(myInt) RELATIONAL_OPERATOR(>=) NUMBER(0) RPARENTHESIS())

	KEYWORD(then) KEYWORD(begin) IDENTIFIER(loopCounter) ASSIGN_OPERATOR(:=) NUMBER(10) SEMICOLON(;) KEYWORD(while) LPARENTHESIS(() IDENTIFIER(loopCounter) RELATIONAL_OPERATOR(>) NUMBER(0) RPARENTHESIS()) LOGICAL_OPERATOR(or) LPARENTHESIS(() LOGICAL_OPERATOR(not) IDENTIFIER(isReady) RPARENTHESIS()) KEYWORD(do) KEYWORD(begin) KEYWORD(if) LPARENTHESIS(() IDENTIFIER(loopCounter) RELATIONAL_OPERATOR(<=) NUMBER(5) RPARENTHESIS()) LOGICAL_OPERATOR(and) LPARENTHESIS(() IDENTIFIER(loopCounter) RELATIONAL_OPERATOR(<>) NUMBER(3) RPARENTHESIS()) KEYWORD(then) IDENTIFIER(writeln) LPARENTHESIS(() STRING_LITERAL('Looping...') RPARENTHESIS()) KEYWORD(else) IDENTIFIER(writeln) LPARENTHESIS(() STRING_LITERAL('Still looping...') RPARENTHESIS()) SEMICOLON(;) IDENTIFIER(loopCounter) ASSIGN_OPERATOR(:=) IDENTIFIER(loopCounter) ARITHMETIC_OPERATOR(-) NUMBER(1) SEMICOLON(;) KEYWORD(end)

	SEMICOLON(;) KEYWORD(end) SEMICOLON(;) KEYWORD(if) IDENTIFIER(loopCounter) RELATIONAL_OPERATOR(<) NUMBER(1) KEYWORD(then) IDENTIFIER(myArray) LBRACKET([) NUMBER(1) RBRACKET(]) ASSIGN_OPERATOR(:=) CHAR_LITERAL('A') SEMICOLON(;) KEYWORD(end) DOT(.)

3.2.9 Test Case 9: BooleanTest

Input.pas	Output.txt
	KEYWORD(program) IDENTIFIER(BooleanTest) SEMICOLON(;) KEYWORD(var) IDENTIFIER(flag1) COMMA(,) IDENTIFIER(flag2) COLON(:) KEYWORD(boolean) SEMICOLON(;) IDENTIFIER(result) COLON(:) KEYWORD(boolean) SEMICOLON(;) KEYWORD(begin) IDENTIFIER(flag1) ASSIGN_OPERATOR(:=) KEYWORD(true) SEMICOLON(;) IDENTIFIER(flag2) ASSIGN_OPERATOR(:=) KEYWORD(false) SEMICOLON(;) IDENTIFIER(result) ASSIGN_OPERATOR(:=)

	IDENTIFIER(flag1) LOGICAL_OPERATOR(and) IDENTIFIER(flag2) SEMICOLON(;) KEYWORD(if) IDENTIFIER(result) KEYWORD(then) IDENTIFIER(writeln) LPARENTHESIS(() STRING_LITERAL('Result is true') RPARENTHESIS()) KEYWORD(else) IDENTIFIER(writeln) LPARENTHESIS(() STRING_LITERAL('Result is false') RPARENTHESIS()) SEMICOLON(;) KEYWORD(end) DOT(.)

Bukti Screenshot untuk test case 8 dan 9 terlampir di bagian lampiran

3.3 Ringkasan Hasil Pengujian

Pengujian program lexical analyzer dilakukan menggunakan serangkaian test case untuk memvalidasi setiap aspek dari aturan DFA.

1. Pengujian Fungsionalitas Dasar
   Program berhasil memproses kode PASCAL-S yang valid (Dapat dilihat pada test case 1, 11, 12). Pengujian ini mengonfirmasi akurasi pengenalan semua kategori token seperti KEYWORD, IDENTIFIER, NUMBER, OPERATOR, dan LITERAL.

2. Pengujian Penanganan Kesalahan
   Program berhasil memproses kode PASCAL-S yang tidak valid. Penanganan *Illegal Symbol* (test case 1, 3, dan 6) dan *Unterminated Constructs* (Test case 4 dan 5). Program berhasil mendeteksi dan melaporkan lexical error ketika menemukan karakter yang tidak terdefinisi dalam aturan DFA.

BAB IV
KESIMPULAN DAN SARAN

4.1 Kesimpulan

Berdasarkan pengujian yang telah dilakukan, dapat disimpulkan bahwa *lexer* untuk bahasa Pascal-S berhasil diimplementasikan dengan baik menggunakan pendekatan berbasis DFA. Implementasi tersebut mencakup 54 *state* untuk mengenali 18 jenis token yang berbeda. Struktur formal DFA memberikan dasar yang kuat dalam proses *lexer*, sehingga mampu mengenali pola token secara deterministik. Penggunaan model *state machine* yang formal memberikan kejelasan dalam perancangan dan implementasi pada *pattern matching* dan *token recognition*.

Dari sisi perancangan, penerapan pendekatan berbasis konfigurasi memberikan fleksibilitas dalam pengelolaan aturan DFA dan konfigurasi sistem. Pemisahan ini dicapai dengan memanfaatkan berkas eksternal berformat JSON untuk mendefinisikan aturan DFA. Selain itu, penggunaan character classes dalam aturan DFA sangat membantu dalam menyederhanakan sistem. Dengan mengelompokkan karakter yang memiliki sifat serupa, seperti huruf, digit, dan spasi, aturan transisi dapat dibuat lebih ringkas dan mudah dipahami.

Secara keseluruhan, sistem *lexer* yang dibangun dengan pendekatan formal berbasis DFA terbukti efisien dalam menjalankan proses analisis leksikal. Integrasi antara lexer dan konfigurasi eksternal DFA menghasilkan sistem yang bersifat modular serta mudah dikembangkan lebih lanjut. Hasil ini dapat dijadikan dasar untuk tahapan berikutnya dalam mekanisme kompilator, yaitu *syntax analysis*.

4.2 Saran

Untuk pengembangan lebih lanjut untuk program lexical analyzer, terdapat beberapa saran yang dapat dipertimbangkan :

1. Integrasi dengan parser (Syntax Analyzer). Output dari program lexer ini digunakan sebagai input untuk sebuah parser. Sehingga, rangkaian token yang dihasilkan dapat divalidasi strukturnya sesuai dengan tata bahasa formal dari bahasa PASCAL-S.

LAMPIRAN

Link Repository : https://github.com/ivant8k/MOE-Tubes-IF2224/tree/main
Link Diagram :
https://drive.google.com/file/d/1pTG5Xr3bADhWrdvgtx4SeR4Geufj5gj8/view?usp=sharing

Nama	NIM	Pembagian Tugas	Persentase
Ivant Samuel Silaban	13523129	• Mengimplementasikan kelas Lexer pada lexer.py. • Membuat logika looping untuk simulasi DFA dan prinsip Longest Match. • Mengimplementasikan kelas LexicalError dan logika error handling.	25%
Rafa Abdussalam Danadyaksa	13523133	• Merancang Keseluruhan Diagram DFA • Mendefinisikan character classes, keywords, dan operator pada dfa.json • Melakukan pengujian dan Laporan Hasil Pengujian	19%
Muhamad Nazih Najmudin	13523144	• Merancang Keseluruhan Diagram DFA • Mendefinisikan character classes, keywords, dan operator pada dfa.json • Menulis Laporan (BAB 2)	19%
Anas Ghazi Al Gifari	13523159	• Mendefinisikan character classes, keywords, dan operator pada dfa.json • Menulis Landasan Teori (BAB I) • Menulis laporan bagian implementasi algortima	19%
Muhammad Rizain Firdaus	13523164	• Mendefinisikan character classes, keywords, dan operator pada dfa.json • Menulis laporan metodologi pengujian dan analisis hasil • Laporan Kesimpulan dan Saran	18%


```
KEYWORD(program)
IDENTIFIER(comprehensiveTest)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(myInt)
COMMA(,)
IDENTIFIER(loopCounter)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(myReal)
COLON(:)
KEYWORD(real)
SEMICOLON(;)
IDENTIFIER(isReady)
COLON(:)
KEYWORD(boolean)
SEMICOLON(;)
IDENTIFIER(myInitial)
COLON(:)
KEYWORD(char)
SEMICOLON(;)
IDENTIFIER(myArray)
COLON(:)
KEYWORD(array)
LBRACKET([)
NUMBER(1)
RANGE_OPERATOR(..)
NUMBER(10)
RBRACKET(])
KEYWORD(of)
KEYWORD(char)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(myInt)
ASSIGN_OPERATOR(:=)
LPARENTHESIS(()
NUMBER(100)
ARITHMETIC_OPERATOR(mod)
NUMBER(3)
RPARENTHESIS())
ARITHMETIC_OPERATOR(+)
LPARENTHESIS(()
NUMBER(20)
ARITHMETIC_OPERATOR(*)
NUMBER(3)
```

```
KEYWORD(program)
IDENTIFIER(BooleanTest)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(flag1)
COMMA(,)
IDENTIFIER(flag2)
COLON(:)
KEYWORD(boolean)
SEMICOLON(;)
IDENTIFIER(result)
COLON(:)
KEYWORD(boolean)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(flag1)
ASSIGN_OPERATOR(:=)
KEYWORD(true)
SEMICOLON(;)
IDENTIFIER(flag2)
ASSIGN_OPERATOR(:=)
KEYWORD(false)
SEMICOLON(;)
IDENTIFIER(result)
ASSIGN_OPERATOR(:=)
IDENTIFIER(flag1)
LOGICAL_OPERATOR(and)
IDENTIFIER(flag2)
SEMICOLON(;)
KEYWORD(if)
IDENTIFIER(result)
KEYWORD(then)
IDENTIFIER(writeln)
LPARENTHESIS(()
```

```
RPARENTHESIS())
ARITHMETIC_OPERATOR(div)
NUMBER(2)
ARITHMETIC_OPERATOR(-)
NUMBER(1)
SEMICOLON(;)
IDENTIFIER(myReal)
ASSIGN_OPERATOR(:=)
NUMBER(123.456)
ARITHMETIC_OPERATOR(/)
NUMBER(2.0E-1)
SEMICOLON(;)
IDENTIFIER(isReady)
ASSIGN_OPERATOR(:=)
IDENTIFIER(true)
SEMICOLON(;)
IDENTIFIER(myInitial)
ASSIGN_OPERATOR(:=)
CHAR_LITERAL('Z')
SEMICOLON(;)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('--- Starting Loop ---')
RPARENTHESIS())
```

```
SEMICOLON(;)
KEYWORD(if)
LPARENTHESIS(()
IDENTIFIER(isReady)
RELATIONAL_OPERATOR(=)
IDENTIFIER(true)
RPARENTHESIS())
LOGICAL_OPERATOR(and)
LPARENTHESIS(()
IDENTIFIER(myInt)
RELATIONAL_OPERATOR(>=)
NUMBER(0)
RPARENTHESIS())
KEYWORD(then)
KEYWORD(begin)
IDENTIFIER(loopCounter)
ASSIGN_OPERATOR(:=)
NUMBER(10)
SEMICOLON(;)
KEYWORD(while)
LPARENTHESIS(()
IDENTIFIER(loopCounter)
RELATIONAL_OPERATOR(>)
NUMBER(0)
```

```
STRING_LITERAL('Result is true')
RPARENTHESIS())
KEYWORD(else)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Result is false')
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(end)
DOT(.)
```

```
RPARENTHESIS())
LOGICAL_OPERATOR(or)
LPARENTHESIS(()
LOGICAL_OPERATOR(not)
IDENTIFIER(isReady)
RPARENTHESIS())
KEYWORD(do)
KEYWORD(begin)
KEYWORD(if)
LPARENTHESIS(()
IDENTIFIER(loopCounter)
RELATIONAL_OPERATOR(<=)
NUMBER(5)
RPARENTHESIS())
LOGICAL_OPERATOR(and)
LPARENTHESIS(()
IDENTIFIER(loopCounter)
RELATIONAL_OPERATOR(<>)
NUMBER(3)
RPARENTHESIS())
KEYWORD(then)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Looping...')
```

```
RPARENTHESIS())
KEYWORD(else)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Still looping...')
RPARENTHESIS())
SEMICOLON(;)
IDENTIFIER(loopCounter)
ASSIGN_OPERATOR(:=)
IDENTIFIER(loopCounter)
ARITHMETIC_OPERATOR(-)
NUMBER(1)
SEMICOLON(;)
KEYWORD(end)
SEMICOLON(;)
KEYWORD(end)
SEMICOLON(;)
KEYWORD(if)
IDENTIFIER(loopCounter)
RELATIONAL_OPERATOR(<)
NUMBER(1)
KEYWORD(then)
IDENTIFIER(myArray)
LBRACKET([)
```

```
143  NUMBER(1)
144  RBRACKET(])
145  ASSIGN_OPERATOR(:=)
146  CHAR_LITERAL('A')
147  SEMICOLON(;)
148  KEYWORD(end)
149  DOT(.)
```

REFERENSI

[1] N. Wirth, “PASCAL-S: A Subset and its Implementation,” dalam *Pascal: The Language and its Implementation*, disunting oleh D. W. Barron. New York, NY, USA: John Wiley & Sons, 1981, hlm. 199–259. [Daring]. Tersedia di: http://pascal.hansotten.com/uploads/pascals/PASCAL-S%20A%20subset%20and%20its%20Implementation%20012.pdf

[2] J. E. Hopcroft, R. Motwani, dan J. D. Ullman, Automata Theory, Languages, and Computation, edisi ke-3. Boston: Pearson Education, 2006

[3]