

IF2224 TEORI BAHASA FORMAL DAN OTOMATA

IMPLEMENTASI LEXICAL ANALYSIS PADA PASCAL-S COMPILER BERBASIS DFA

Laporan Milestone 1

Disusun untuk memenuhi tugas mata kuliah Teori Bahasa Formal dan Otomata pada
Semester 5 (lima)
Tahun Akademik 2025/2026



Disusun Oleh:

| | |
|-----------------------|----------|
| Brian Ricardo Tamin | 13523126 |
| Jovandra Otniel P. S. | 13523141 |
| Andrew Tedjapratama | 13523148 |
| Theo Kurniady | 13523154 |

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

2025

DAFTAR ISI

| | |
|--|-----------|
| DAFTAR ISI..... | 2 |
| BAB I DESKRIPSI PERSOALAN..... | 3 |
| BAB II LANDASAN TEORI..... | 4 |
| 2.1 Deterministic Finite Automata (DFA)..... | 4 |
| 2.2 Lexical Analysis..... | 4 |
| 2.3 Token..... | 5 |
| BAB III ANALISIS DFA AUTOMATON..... | 7 |
| 3.1 State dan Transition Function..... | 7 |
| 3.2 Diagram DFA..... | 8 |
| BAB IV IMPLEMENTASI PROGRAM..... | 9 |
| 4.1 Lexer..... | 9 |
| 4.2 DFA Loader..... | 10 |
| 4.3 DFA Engine..... | 11 |
| 4.4 Token..... | 12 |
| 4.4 Errors..... | 12 |
| BAB V TEST CASE..... | 13 |
| BAB VI KESIMPULAN & SARAN..... | 22 |
| 6.1 Kesimpulan..... | 22 |
| 6.2 Saran..... | 22 |
| REFERENSI..... | 23 |

BAB I

DESKRIPSI PERSOALAN

Bahasa pemrograman Pascal-S merupakan sebuah subset dari bahasa Pascal yang dirancang dengan aturan sintaks yang lebih terbatas. Untuk dapat menerjemahkan kode Pascal-S menjadi struktur yang dapat diproses lebih lanjut oleh compiler, diperlukan tahap Lexical Analysis sebagai langkah pertama dalam proses kompilasi.

Pada tahap ini, program masih berupa rangkaian karakter mentah yang tidak memiliki makna terstruktur. Compiler tidak dapat langsung memahami karakter-karakter tersebut tanpa proses pemisahan (tokenisasi). Oleh karena itu, dibutuhkan sebuah komponen lexer yang membaca input karakter demi karakter, kemudian mengelompokkan karakter tersebut menjadi satuan bermakna yang disebut token, seperti KEYWORD, IDENTIFIER, NUMBER, OPERATOR, dan sebagainya.

Permasalahan yang muncul adalah bagaimana cara membangun sebuah lexer yang dapat mengenali token secara deterministik dan konsisten, sesuai dengan aturan dari bahasa Pascal-S. Proses ini harus dilakukan menggunakan Deterministic Finite Automata (DFA) yang didefinisikan secara eksplisit melalui file aturan .json atau .txt, sesuai ketentuan tugas besar. Lexer juga harus mampu mengabaikan karakter yang tidak relevan seperti spasi dan baris baru, serta mendeteksi karakter ilegal sejak awal tanpa menyebabkan program berhenti secara tidak terkontrol.

Dengan demikian, fokus utama dari milestone ini adalah merancang dan mengimplementasikan lexical analyzer berbasis DFA yang mampu:

1. Membaca input .pas
2. Menjalankan proses scanning berbasis state transition
3. Menghasilkan list token yang valid dan terstruktur
4. Menjadi pondasi awal untuk tahap parsing pada milestone berikutnya

BAB II

LANDASAN TEORI

2.1 Deterministic Finite Automata (DFA)

Deterministic Finite Automata adalah model matematis deterministik yang terdiri dari sejumlah *state* (keadaan), *input alphabet* (alfabet input/symbol masukan), dan fungsi transisi yang memetakan *state* dan alfabet input ke *state* lain. Berbeda dari Nondeterministic Finite Automata, DFA hanya memiliki satu transisi untuk setiap domain fungsi (*state*, *input alphabet*). DFA direpresentasikan sebagai 5-tuple sebagai berikut:

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

dimana:

Q : Himpunan semua state

Σ : Himpunan alfabet input

δ : Fungsi transisi

q_0 : State awal (*start state*)

F : Himpunan state akhir / diterima (*final/accepting states*)

DFA bekerja dengan cara membaca input secara berurutan dari kiri ke kanan. Ketika setiap karakter dibaca, DFA berpindah ke state berikutnya sesuai aturan fungsi transisi. Jika DFA berhenti di salah satu *final state*, string tersebut dikenali sebagai pola yang valid.

2.2 Lexical Analysis

Analisis leksikal adalah tahap pertama dalam kompilasi kode. Analisis ini bertujuan untuk membaca karakter mentah dari *source code* dan mengklasifikasikan kumpulan karakter tersebut menjadi token. Setiap token memiliki *type* dan *value*. Contohnya seperti berikut:

| Dari | Menjadi |
|---------------------------|---|
| <code>var n := 10;</code> | KEYWORD (var) IDENTIFIER (n) ASSIGN_OPERATOR (:=) NUMBER (10) SEMICOLON (;) |

Proses tersebut dilakukan oleh bagian dari compiler yang disebut lexer. Lexer diimplementasikan menggunakan model DFA untuk mendeteksi pola token dari karakter-karakter pada *source code*. Selain itu, lexer juga berperan dalam mengabaikan *whitespace*, mengabaikan komentar, dan melaporkan error jika terdapat simbol tak dikenal. Hasil dari analisis leksikal yang berupa daftar token kemudian diproses pada tahap selanjutnya, yaitu *syntax analysis (parser)*.

2.3 Token

Token merupakan unit bahasa terkecil yang memiliki arti dalam bahasa pemrograman. Umumnya, token terbentuk dari 2 bagian utama, yaitu tipe (*type*) dan nilai (*value*). Tipe menunjukkan kategori token seperti *identifier*, *keyword*, dan *operator*. Nilai berisi satu atau kumpulan simbol aktual dari token tersebut. Berikut ini contoh tabel token dalam Pascal-S:

Tabel 2.3.1 Daftar Token dalam Pascal-S

| 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 | .. | - |

BAB III

ANALISIS DFA AUTOMATON

3.1 State dan Transition Function

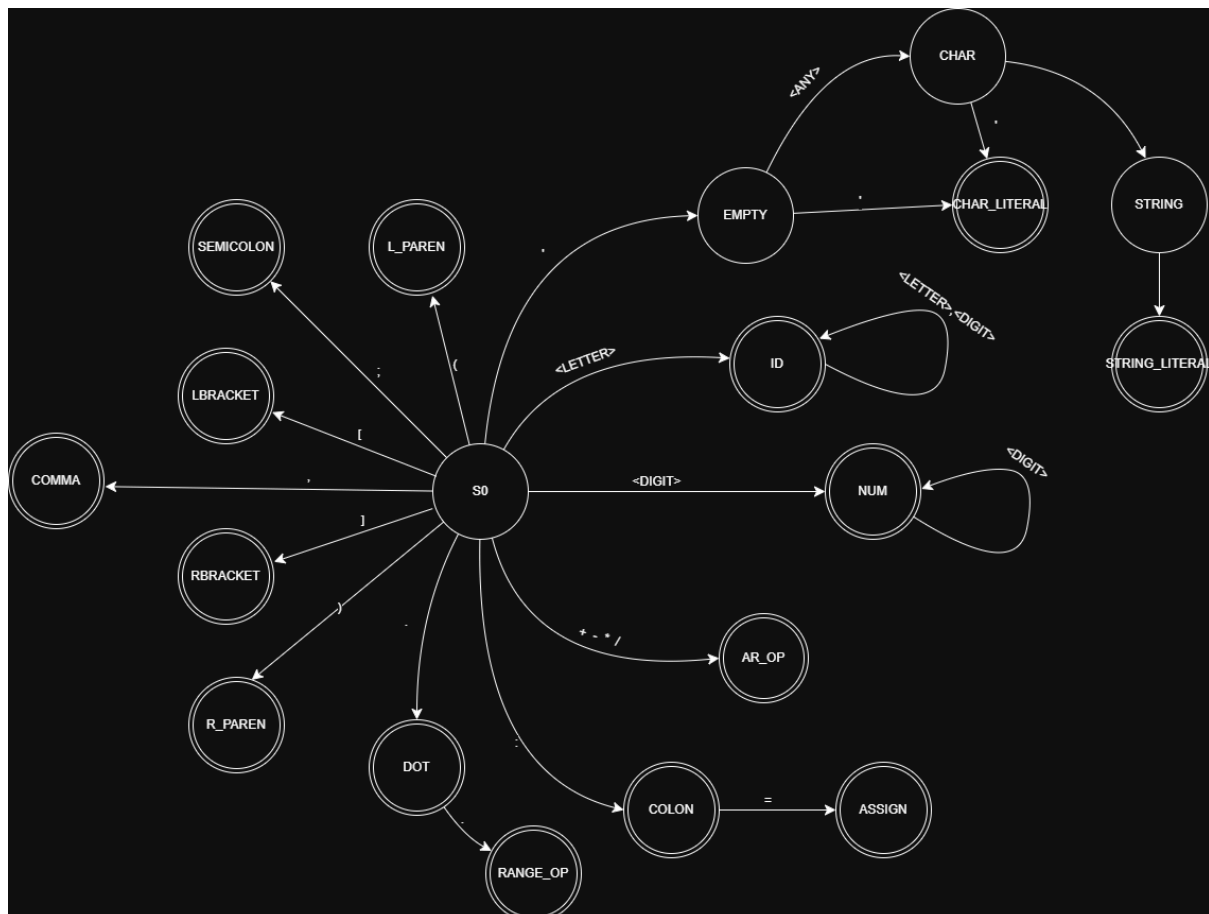
Kami merancang Deterministic Finite Automata dengan notasi JSON yang terdiri start_state (S0), himpunan final_states, dan fungsi transitions. Komponen-komponen tersebut dibuat berdasarkan list token yang legal dalam bahasa pemrograman Pascal-S.

| DFA States & Transitions | Penjelasan Singkat |
|--|---|
| <pre> { "start_state": "S0", "final_states": ["ID", "NUM", "ARITHMETIC_OPERATOR", "ASSIGN", "COLON", "STRING_LITERAL", "CHAR_LITERAL", "SEMICOLON", "COMMA", "DOT", "RANGE_OPERATOR", "LPARENTHESIS", "RPARENTHESIS"], "transitions": { "S0": { "<LETTER>": "ID", "<DIGIT>": "NUM", "+": "ARITHMETIC_OPERATOR", "-": "ARITHMETIC_OPERATOR", "*": "ARITHMETIC_OPERATOR", "/": "ARITHMETIC_OPERATOR", ":": "COLON", ";": "SEMICOLON", ",": "COMMA", ".": "DOT", "(": "LPARENTHESIS", ")": "RPARENTHESIS", " ": "EMPTY" }, "ID": { "<LETTER>": "ID", "<DIGIT>": "ID" }, "NUM": { "<DIGIT>": "NUM" }, "COLON": { "=": "ASSIGN" }, "DOT": { ".": "RANGE_OPERATOR" }, "EMPTY": { "<ANY>": "CHAR", "' ': "CHAR_LITERAL" }, "CHAR": { "<ANY>": "STRING", "' ': "CHAR_LITERAL" }, "STRING": { "<ANY>": "STRING", "' ': "STRING_LITERAL" } } }</pre> | <p>Berikut merupakan json yang mendefinisikan aturan transisi DFA yang digunakan oleh lexer untuk mengenali token pada bahasa Pascal-S.</p> <p>State S0 berfungsi sebagai start state yang menangani karakter awal dari setiap token. Jika karakter pertama termasuk dalam kelas <LETTER> atau <DIGIT>, lexer berpindah ke state ID atau NUM untuk membentuk identifier atau angka dengan kemungkinan lebih dari satu karakter (loopback transition).</p> <p>Karakter tunggal seperti +, -, *, /, ;, ,, (,), [, dan] langsung dipetakan menuju final state masing-masing, karena token tersebut bersifat single-character token. Untuk operator :=, digunakan state COLON yang hanya akan berpindah ke state ASSIGN jika diikuti oleh karakter =. Untuk operator range .., digunakan state DOT yang hanya akan berpindah ke state RANGE_OPERATOR jika diikuti oleh karakter '.'.</p> <p>Literal string ditandai dengan token pembuka ' dan langsung dialihkan ke state EMPTY. Pada state ini, ada dua pilihan, yaitu karakter apapun atau tanda petik ('). Memilih karakter apapun akan memindahkan ke state CHAR, sedangkan tanda petik memindahkan ke state CHAR_LITERAL yang merupakan final state. Pada state char, ada dua pilihan yang sama seperti sebelumnya dengan</p> |

| | |
|--|--|
| | <p>perbedaan karakter apapun memindahkan ke state STRING. Dari state STRING, dapat diperpanjang dan tetap di state STRING hingga mendeteksi tanda petik yang memindahkan ke state STRING_LITERAL yang merupakan final state.</p> |
|--|--|

3.2 Diagram DFA

Perancangan DFA fokus mengenali token deterministik (IDENTIFIER, NUMBER, operator, simbol tunggal). Untuk literal string/karakter, DFA hanya menandai awal dan akhir kutip. Isi string tidak dimodelkan eksplisit agar DFA minimalis dan mudah diverifikasi. Pembacaan isi literal dialihkan ke lexer Python setelah transisi awal. Pendekatan ini memisahkan peran DFA sebagai pengenali token struktural dari penanganan konten literal oleh lexer, memungkinkan perluasan fitur tanpa modifikasi automata.



Tabel 3.2.1 Diagram DFA lexer

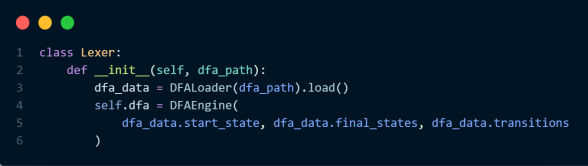
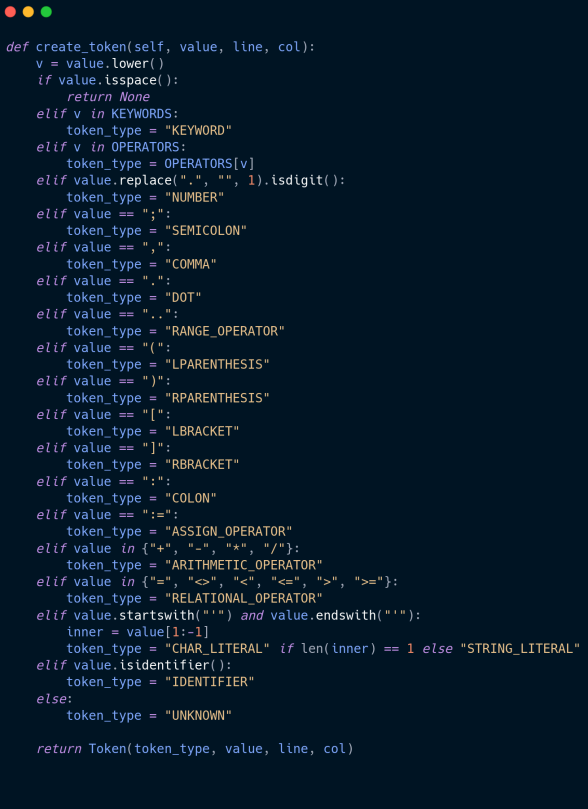
- <LETTER> : [A-Za-z]
- <DIGIT> : [0-9]
- <ANY> : [^]

BAB IV

IMPLEMENTASI PROGRAM

4.1 Lexer

Pada aplikasi lexer Pascal-S, lexer merupakan komponen utama dalam memecah *source code* menjadi token-token yang valid.

| Class & Functions | Penjelasan Singkat |
|--|--|
|  <pre> 1 class Lexer: 2 def __init__(self, dfa_path): 3 dfa_data = DFALoader(dfa_path).load() 4 self.dfa = DFAEngine(5 dfa_data.start_state, dfa_data.final_states, dfa_data.transitions 6) </pre> | <p>Inisialisasi dari kelas lexer yang berisi data aturan transisi dan engine dari DFA.</p> |
|  <pre> def create_token(self, value, line, col): v = value.lower() if value.isspace(): return None elif v in KEYWORDS: token_type = "KEYWORD" elif v in OPERATORS: token_type = OPERATORS[v] elif value.replace(".", "", 1).isdigit(): token_type = "NUMBER" elif value == ";": token_type = "SEMICOLON" elif value == ",": token_type = "COMMA" elif value == ".": token_type = "DOT" elif value == "..": token_type = "RANGE_OPERATOR" elif value == "(": token_type = "LPARENTHESIS" elif value == ")": token_type = "RPARENTHESIS" elif value == "[": token_type = "LBRACKET" elif value == "]": token_type = "RBRACKET" elif value == ":": token_type = "COLON" elif value == "=:": token_type = "ASSIGN_OPERATOR" elif value in {"+", "-", "*", "/"}: token_type = "ARITHMETIC_OPERATOR" elif value in {"=", "<", "<=", ">", ">="}: token_type = "RELATIONAL_OPERATOR" elif value.startswith("'") and value.endswith("'"): inner = value[1:-1] token_type = "CHAR_LITERAL" if len(inner) == 1 else "STRING_LITERAL" elif value.isidentifier(): token_type = "IDENTIFIER" else: token_type = "UNKNOWN" return Token(token_type, value, line, col) </pre> | <p>Fungsi untuk mengklasifikasi suatu char input menjadi suatu predefined state berdasarkan token yang sudah ditentukan dari awal.</p> |

```

def tokenize(self, text):
    tokens = []
    current = ""
    line, col = 1, 0
    i = 0
    self.dfa.reset()

    while i < len(text):
        char = text[i]
        col += 1

        # newline
        if char == "\n":
            line += 1
            col = 0
            i += 1
            continue

        success = self.dfa.next_state(char)

        if success:
            current += char
            i += 1
        else:
            if current:
                token = self.create_token(current, line, col)
                if token:
                    tokens.append(token)
                    current = ""
                    self.dfa.reset()
            else:
                token = self.create_token(char, line, col)
                if token:
                    tokens.append(token)
                    self.dfa.reset()
                i += 1

    if current:
        tokens.append(self.create_token(current, line, col))

    return tokens


```

Fungsi utama untuk melakukan parsing input kode teks pascal menjadi kumpulan token. Fungsi akan membaca kode per baris dan maju per kolom. Fungsi ini memanfaatkan kelas lexer yang telah diinisialisasi sebelumnya untuk menghasilkan token dengan fungsi `create_token` sesuai aturan dengan DFA.

4.2 DFA Loader





DFA Loader mengambil aturan DFA dari file JSON. Loader membuka dan mem-parsing file JSON, kemudian mengembalikan hasilnya ke kelas lexer.

| Class & Functions | Penjelasan Singkat |
|--|--|
| <pre> 1 class DFALoader: 2 def __init__(self, path): 3 self.path = path 4 self.start_state = None 5 self.final_states = set() 6 self.transitions = {} </pre> | <p>Class untuk membaca <code>dfa.json</code>, dimana json menyimpan seluruh informasi mengenai state dan fungsi transisi, serta start dan final state.</p> |

| | |
|--|--|
|  <pre> 1 def load(self): 2 with open(self.path, "r") as f: 3 data = json.load(f) 4 self.start_state = data["start_state"] 5 self.final_states = set(data["final_states"]) 6 self.transitions = data["transitions"] 7 return self </pre> | <p>Fungsi untuk melakukan load pada dfa.json dan parsing berdasarkan klasifikasinya start state/ final state/ fungsi transisi.</p> |
|--|--|

4.3 DFA Engine

DFA Engine merupakan komponen utama dalam lexer. Ia berperan sebagai mesin DFA yang mengatur state dan transisi ketika digunakan lexer untuk meng-generate token dari *source code*.

| Class & Functions | Penjelasan Singkat |
|--|--|
|  <pre> 1 class DFAEngine: 2 def __init__(self, start_state, final_states, transitions): 3 self.start_state = start_state 4 self.final_states = final_states 5 self.transitions = transitions 6 self.current_state = start_state </pre> | <p>Class untuk menyimpan seluruh finite set of start state, final state, transition functions dan current state.</p> |
|  <pre> 1 def reset(self): 2 self.current_state = self.start_state 3 </pre> | <p>Untuk mereset suatu current state kembali ke S0 atau ke start state.</p> |
|  <pre> def next_state(self, char): trans = self.transitions.get(self.current_state, {}) if char in trans: self.current_state = trans[char] return True elif "<LETTER>" in trans and char.isalpha(): self.current_state = trans["<LETTER>"] return True elif "<DIGIT>" in trans and char.isdigit(): self.current_state = trans["<DIGIT>"] return True if "<ANY>" in trans: self.current_state = trans["<ANY>"] return True else: return False </pre> | <p>Untuk melakukan next state pada suatu current state , berdasarkan character terbaca dari pascal file. Selain itu, fungsi ini membantu parsing makna dari <LETTER>, <DIGIT>, dan <ANY> pada dfa_rules.json</p> |
|  <pre> 1 def is_accepting(self): 2 return self.current_state in self.final_states 3 </pre> | <p>Function untuk menentukan apakah sebuah state adalah accepting atau final state.</p> |

4.4 Token

Bagian ini mendefinisikan token-token yang akan digunakan pada lexer sebagai category identifier berdasarkan hasil parsing dari pascal file.

| Class & Functions | Penjelasan Singkat |
|---|--|
| <pre>1 class Token: 2 def __init__(self, type_, value, line, column): 3 self.type = type_ 4 self.value = value 5 self.line = line 6 self.column = column 7 8 def __repr__(self): 9 if self.type in {"STRING_LITERAL", "CHAR_LITERAL"} and not self.value.startswith('"'): 10 return f"{self.type}('{self.value}')" 11 return f"{self.type}({self.value})"</pre> | Constructor untuk pembuatan token dari hasil lexical analyzer dfa, juga menyimpan token-token yang akan digunakan nantinya |
| <pre>1 KEYWORDS = { 2 "program", "var", "begin", "end", "if", "then", "else", "while", "do", "for", 3 "to", "downto", "integer", "real", "boolean", "char", "array", "of", 4 "procedure", "function", "const", "type" 5 } 6 7 OPERATORS = { 8 "+": "ARITHMETIC_OPERATOR", 9 "-": "ARITHMETIC_OPERATOR", 10 "*": "ARITHMETIC_OPERATOR", 11 "/": "ARITHMETIC_OPERATOR", 12 "div": "ARITHMETIC_OPERATOR", 13 "mod": "ARITHMETIC_OPERATOR", 14 ":=": "ASSIGN_OPERATOR", 15 "=": "RELATIONAL_OPERATOR", 16 "<": "RELATIONAL_OPERATOR", 17 ">": "RELATIONAL_OPERATOR", 18 "<=": "RELATIONAL_OPERATOR", 19 ">=": "RELATIONAL_OPERATOR", 20 "<>": "RELATIONAL_OPERATOR", 21 "and": "LOGICAL_OPERATOR", 22 "or": "LOGICAL_OPERATOR", 23 "not": "LOGICAL_OPERATOR", 24 } 25 26 COMMENTS = { 27 "{": "COMMENT_START", 28 "}": "COMMENT_END", 29 "(*": "COMMENT_START", 30 "*)": "COMMENT_END", 31 } 32</pre> | Set of state identifier yang mencakup keywords, operators, dan comments (comments tidak dipakai), berdasarkan hasil parsing lexical. |

4.4 Errors

Error handler untuk lexical analyzer.

| Class & Functions | Penjelasan Singkat |
|--|---|
| <pre>1 class LexicalError(Exception): 2 def __init__(self, message, line, column): 3 self.message = message 4 self.line = line 5 self.column = column 6 7 def __str__(self): 8 return f"[LexicalError] {self.message} at line {self.line}, column {self.column}" 9</pre> | Output error untuk spesifik line dan column dengan input error message tertentu |

BAB V

TEST CASE

Tabel 5.1 Tabel test case

| No. | Input | Output |
|-----|---|---|
| 1 | <pre> program Hello; var a, b: integer; begin a := 5; b := a + 10; writeln('Result = ', b); end. </pre> | KEYWORD(program) IDENTIFIER(Hello) SEMICOLON(;) KEYWORD(var) IDENTIFIER(a) COMMA(, IDENTIFIER(b) COLON(:) KEYWORD(integer) SEMICOLON(;) KEYWORD(begin) IDENTIFIER(a) ASSIGN_OPERATOR(:=) NUMBER(5) SEMICOLON(;) IDENTIFIER(b) ASSIGN_OPERATOR(:=) IDENTIFIER(a) ARITHMETIC_OPERATOR(+) NUMBER(10) SEMICOLON(;) IDENTIFIER(writeln) LPARENTHESIS(STRING_LITERAL('Result = ' COMMA(, IDENTIFIER(b) RPARENTHESIS()) SEMICOLON(;) KEYWORD(end) DOT(.) |
| 2 | <pre> program Hello2; var a, b, i: integer; arr: array[1..5] of integer; begin a := 5; b := a + 10; writeln('Result = ', b); for i := 1 to 5 do begin arr[i] := i + 1; writeln(arr[i]); end; end. </pre> | KEYWORD(program) IDENTIFIER(Hello2) SEMICOLON(;) KEYWORD(var) IDENTIFIER(a) COMMA(, IDENTIFIER(b) COMMA(, IDENTIFIER(i) COLON(:) KEYWORD(integer) SEMICOLON(;) IDENTIFIER(arr) COLON(:) KEYWORD(array) LBRACKET([|

| | | |
|--|--|--|
| | | NUMBER(1) RANGE_OPERATOR(..) NUMBER(5) RBRACKET() KEYWORD(of) KEYWORD(integer) SEMICOLON(;) KEYWORD(begin) IDENTIFIER(a) ASSIGN_OPERATOR(:=) NUMBER(5) SEMICOLON(;) IDENTIFIER(b) ASSIGN_OPERATOR(:=) IDENTIFIER(a) ARITHMETIC_OPERATOR(+) NUMBER(10) SEMICOLON(;) IDENTIFIER(writeln) LPARENTHESIS() STRING_LITERAL('Result = ') COMMA(,) IDENTIFIER(b) RPARENTHESIS()) SEMICOLON(;) KEYWORD(for) IDENTIFIER(i) ASSIGN_OPERATOR(:=) NUMBER(1) KEYWORD(to) NUMBER(5) KEYWORD(do) KEYWORD(begin) IDENTIFIER(arr) LBRACKET([) IDENTIFIER(i) RBRACKET() ASSIGN_OPERATOR(:=) IDENTIFIER(i) ARITHMETIC_OPERATOR(+) NUMBER(1) SEMICOLON(;) IDENTIFIER(writeln) LPARENTHESIS() IDENTIFIER(arr) LBRACKET([) IDENTIFIER(i) RBRACKET() RPARENTHESIS()) SEMICOLON(;) KEYWORD(end) SEMICOLON(;) KEYWORD(end) DOT(.) |
|--|--|--|

| | | |
|---|---|--|
| 3 | <pre> program RangesLoops; var i: integer; a: array[0..3] of integer; begin a[0] := 1; for i := 0 to 3 do a[i] := i; for i := 3 downto 0 do a[i] := a[i] + 1; if a[1] < a[2] then a[1] := a[2]; while i > 0 do i := i - 1; writeln('ok', a[1]); end. </pre> | <pre> KEYWORD(program) IDENTIFIER(RangesLoops) SEMICOLON(;) KEYWORD(var) IDENTIFIER(i) COLON(:) KEYWORD(integer) SEMICOLON(;) IDENTIFIER(a) COLON(:) KEYWORD(array) LBRACKET([) NUMBER(0) RANGE_OPERATOR(..) NUMBER(3) RBRACKET(]) KEYWORD(of) KEYWORD(integer) SEMICOLON(;) KEYWORD(begin) IDENTIFIER(a) LBRACKET([) NUMBER(0) RBRACKET(]) ASSIGN_OPERATOR(:=) NUMBER(1) SEMICOLON(;) KEYWORD(for) IDENTIFIER(i) ASSIGN_OPERATOR(:=) NUMBER(0) KEYWORD(to) NUMBER(3) KEYWORD(do) IDENTIFIER(a) LBRACKET([) IDENTIFIER(i) RBRACKET(]) ASSIGN_OPERATOR(:=) IDENTIFIER(i) SEMICOLON(;) KEYWORD(for) IDENTIFIER(i) ASSIGN_OPERATOR(:=) NUMBER(3) KEYWORD(downto) NUMBER(0) KEYWORD(do) IDENTIFIER(a) LBRACKET([) IDENTIFIER(i) RBRACKET(]) ASSIGN_OPERATOR(:=) IDENTIFIER(a) </pre> |
|---|---|--|

| | | |
|---|--|--|
| | | LBRACKET([IDENTIFIER(i) RBRACKET(]) ARITHMETIC_OPERATOR(+) NUMBER(1) SEMICOLON(;) KEYWORD(if) IDENTIFIER(a) LBRACKET([NUMBER(1) RBRACKET(]) RELATIONAL_OPERATOR(<) IDENTIFIER(a) LBRACKET([NUMBER(2) RBRACKET(]) KEYWORD(then) IDENTIFIER(a) LBRACKET([NUMBER(1) RBRACKET(]) ASSIGN_OPERATOR(:=) IDENTIFIER(a) LBRACKET([NUMBER(2) RBRACKET(]) SEMICOLON(;) KEYWORD(while) IDENTIFIER(i) RELATIONAL_OPERATOR(>) NUMBER(0) KEYWORD(do) IDENTIFIER(i) ASSIGN_OPERATOR(:=) IDENTIFIER(i) ARITHMETIC_OPERATOR(-) NUMBER(1) SEMICOLON(;) IDENTIFIER(writeIn) LPARENTHESIS(STRING_LITERAL('ok') COMMA(, IDENTIFIER(a) LBRACKET([NUMBER(1) RBRACKET(]) RPARENTHESIS()) SEMICOLON(;) KEYWORD(end) DOT(.) |
| 4 | <pre> program Ops; var x, y: integer; </pre> | KEYWORD(program) IDENTIFIER(Ops) SEMICOLON(;) |

| | |
|--|---|
| <pre> r: real; ch: char; ok: boolean; begin x := 10; y := 3; r := 12.5; ch := 'Z'; ok := (x > y) and not (x = y) x := x + y - 2 * (y div 2) / 1; x := x mod y; if x <= y then x := y else x := y + 1; writeln('done ', x); end. </pre> | <pre> KEYWORD(var) IDENTIFIER(x) COMMA(,) IDENTIFIER(y) COLON(:) KEYWORD(integer) SEMICOLON(;) IDENTIFIER(r) COLON(:) KEYWORD(real) SEMICOLON(;) IDENTIFIER(ch) COLON(:) KEYWORD(char) SEMICOLON(;) IDENTIFIER(ok) COLON(:) KEYWORD(boolean) SEMICOLON(;) KEYWORD(begin) IDENTIFIER(x) ASSIGN_OPERATOR(:=) NUMBER(10) SEMICOLON(;) IDENTIFIER(y) ASSIGN_OPERATOR(:=) NUMBER(3) SEMICOLON(;) IDENTIFIER(r) ASSIGN_OPERATOR(:=) NUMBER(12) DOT(.) NUMBER(5) SEMICOLON(;) IDENTIFIER(ch) ASSIGN_OPERATOR(:=) CHAR_LITERAL('Z') SEMICOLON(;) IDENTIFIER(ok) ASSIGN_OPERATOR(:=) LPARENTHESIS(() IDENTIFIER(x) RELATIONAL_OPERATOR(>) IDENTIFIER(y) RPARENTHESIS() LOGICAL_OPERATOR(and) LOGICAL_OPERATOR(not) LPARENTHESIS(() IDENTIFIER(x) RELATIONAL_OPERATOR(=) IDENTIFIER(y) RPARENTHESIS() IDENTIFIER(x) ASSIGN_OPERATOR(:=) </pre> |
|--|---|

| | | |
|---|--|---|
| | | IDENTIFIER(x) ARITHMETIC_OPERATOR(+) IDENTIFIER(y) ARITHMETIC_OPERATOR(-) NUMBER(2) ARITHMETIC_OPERATOR(*) LPARENTHESIS() IDENTIFIER(y) ARITHMETIC_OPERATOR(di v) NUMBER(2) RPARENTHESIS()) ARITHMETIC_OPERATOR(/) NUMBER(1) SEMICOLON(;) IDENTIFIER(x) ASSIGN_OPERATOR(:=) IDENTIFIER(x) ARITHMETIC_OPERATOR(m od) IDENTIFIER(y) SEMICOLON(;) KEYWORD(if) IDENTIFIER(x) RELATIONAL_OPERATOR(<) RELATIONAL_OPERATOR(=) IDENTIFIER(y) KEYWORD(then) IDENTIFIER(x) ASSIGN_OPERATOR(:=) IDENTIFIER(y) KEYWORD(else) IDENTIFIER(x) ASSIGN_OPERATOR(:=) IDENTIFIER(y) ARITHMETIC_OPERATOR(+) NUMBER(1) SEMICOLON(;) IDENTIFIER(writeln) LPARENTHESIS() STRING_LITERAL('done ') COMMA(,) IDENTIFIER(x) RPARENTHESIS()) SEMICOLON(;) KEYWORD(end) DOT(.) |
| 5 | <pre> program DeclsProcFunc; const K = 10; type Index = 1..5; var </pre> | KEYWORD(program) IDENTIFIER(DeclsProcFunc) SEMICOLON(;) KEYWORD(const) IDENTIFIER(K) RELATIONAL_OPERATOR(=) |

| | |
|---|---|
| <pre> i: integer; r: real; ch: char; ok: boolean; arr: array[Index] of integer; procedure IncVar(var z: integer); begin z := z + 1; end; function AddOne(t: integer): integer; begin AddOne := t + 1; end; begin r := 3.14; ch := 'A'; ok := true and not false; {comment} arr[1] := 2; if (arr[1] >= K) then arr[1] := AddOne(arr[1]) else IncVar(arr[1]); writeln('Value: ', arr[1], '.'); end </pre> | <pre> NUMBER(10) SEMICOLON(;) KEYWORD(type) IDENTIFIER(Index) RELATIONAL_OPERATOR(=) NUMBER(1) RANGE_OPERATOR(..) NUMBER(5) SEMICOLON(;) KEYWORD(var) IDENTIFIER(i) COLON(:) KEYWORD(integer) SEMICOLON(;) IDENTIFIER(r) COLON(:) KEYWORD(real) SEMICOLON(;) IDENTIFIER(ch) COLON(:) KEYWORD(char) SEMICOLON(;) IDENTIFIER(ok) COLON(:) KEYWORD(boolean) SEMICOLON(;) IDENTIFIER(arr) COLON(:) KEYWORD(array) LBRACKET([) IDENTIFIER(Index) RBRACKET(]) KEYWORD(of) KEYWORD(integer) SEMICOLON(;) KEYWORD(procedure) IDENTIFIER(IncVar) LPARENTHESIS(() KEYWORD(var) IDENTIFIER(z) COLON(:) KEYWORD(integer) RPARENTHESIS()) SEMICOLON(;) KEYWORD(begin) IDENTIFIER(z) ASSIGN_OPERATOR(:=) IDENTIFIER(z) ARITHMETIC_OPERATOR(+) NUMBER(1) SEMICOLON(;) KEYWORD(end) SEMICOLON(;) KEYWORD(function) </pre> |
|---|---|

| | | |
|--|--|--|
| | | IDENTIFIER(AddOne) LPARENTHESIS() IDENTIFIER(t) COLON(:) KEYWORD(integer) RPARENTHESIS() COLON(:) KEYWORD(integer) SEMICOLON(;) KEYWORD(begin) IDENTIFIER(AddOne) ASSIGN_OPERATOR(:=) IDENTIFIER(t) ARITHMETIC_OPERATOR(+) NUMBER(1) SEMICOLON(;) KEYWORD(end) SEMICOLON(;) KEYWORD(begin) IDENTIFIER(r) ASSIGN_OPERATOR(:=) NUMBER(3) DOT(.) NUMBER(14) SEMICOLON(;) IDENTIFIER(ch) ASSIGN_OPERATOR(:=) CHAR_LITERAL('A') SEMICOLON(;) IDENTIFIER(ok) ASSIGN_OPERATOR(:=) IDENTIFIER(true) LOGICAL_OPERATOR(and) LOGICAL_OPERATOR(not) IDENTIFIER(false) SEMICOLON(;) IDENTIFIER(arr) LBRACKET([) NUMBER(1) RBRACKET(]) ASSIGN_OPERATOR(:=) NUMBER(2) SEMICOLON(;) KEYWORD(if) LPARENTHESIS() IDENTIFIER(arr) LBRACKET([) NUMBER(1) RBRACKET(]) RELATIONAL_OPERATOR(>) RELATIONAL_OPERATOR(=) IDENTIFIER(K) RPARENTHESIS() KEYWORD(then) |
|--|--|--|

| | | |
|--|--|---|
| | | IDENTIFIER(arr) LBRACKET([) NUMBER(1) RBRACKET(]) ASSIGN_OPERATOR(:=) IDENTIFIER(AddOne) LPARENTHESIS(IDENTIFIER(arr) LBRACKET([) NUMBER(1) RBRACKET(]) RPARENTHESIS()) KEYWORD(else) IDENTIFIER(IncVar) LPARENTHESIS(IDENTIFIER(arr) LBRACKET([) NUMBER(1) RBRACKET(]) RPARENTHESIS()) SEMICOLON(;) IDENTIFIER(writeln) LPARENTHESIS(STRING_LITERAL('Value: ') COMMA(, IDENTIFIER(arr) LBRACKET([) NUMBER(1) RBRACKET(]) COMMA(, CHAR_LITERAL('.') RPARENTHESIS()) SEMICOLON(;) KEYWORD(end) DOT(.) |
|--|--|---|

BAB VI

KESIMPULAN & SARAN

6.1 Kesimpulan

Implementasi lexical analyzer Pascal-S berbasis DFA berhasil membuktikan bahwa proses tokenisasi dapat dilakukan secara deterministik dan terstruktur. Dengan mendefinisikan state, final state, dan fungsi transisi secara eksplisit melalui file JSON, lexer mampu mengenali token-token Pascal-S secara konsisten tanpa ambiguitas. Pendekatan ini juga memisahkan logika pengenalan pola (via DFA) dan parsing karakter (via kode Python), sehingga sistem menjadi modular dan mudah diperluas untuk tahap parsing berikutnya. Keputusan desain untuk tidak memasukkan string literal penuh ke dalam DFA dan menanganinya langsung di lexer terbukti membuat automata lebih sederhana dan tetap efisien.

6.2 Saran

Untuk pengembangan selanjutnya, integrasi antara lexer dan parser sebaiknya dilakukan dengan tetap mempertahankan pemisahan logika agar arsitektur tetap bersih. DFA dapat diperluas untuk mendukung pengenalan token tambahan seperti komentar multiline atau literal kompleks jika dibutuhkan pada tahap lanjutan. Selain itu, pengujian otomatis berbasis batch input .pas dapat ditambahkan untuk memverifikasi konsistensi tokenisasi dan mengurangi potensi bug sebelum masuk ke milestone parsing.

REFERENSI

- [1] Edunex ITB. (n.d.). *Deterministic Finite Automata*. ITB Lecture Module. Accessed: October 19, 2025. [Online]. Available: <http://edunex.itb.ac.id>
- [2] N. Wirth, *PASCAL-S: A Subset and its implementation*. Accessed: October 19, 2025. [Online]. Available: <http://pascal.hansotten.com/uploads/pascals/PASCAL-S%20A%20subset%20and%20its%20Implementation%20012.pdf>
- [3] Tutorials Point. (n.d.). *Compiler Design - Lexical Analysis*. Accessed: October 19, 2025. [Online]. Available: https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm
- [4] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Accessed: October 19, 2025. [Online]. Available: https://cdn-edunex.itb.ac.id/29161-Formal-Language-Theory-and-Automata/1629640939613_Introduction-to-Automata-Theory,-Languages,-and-Computation-Edition-3.pdf