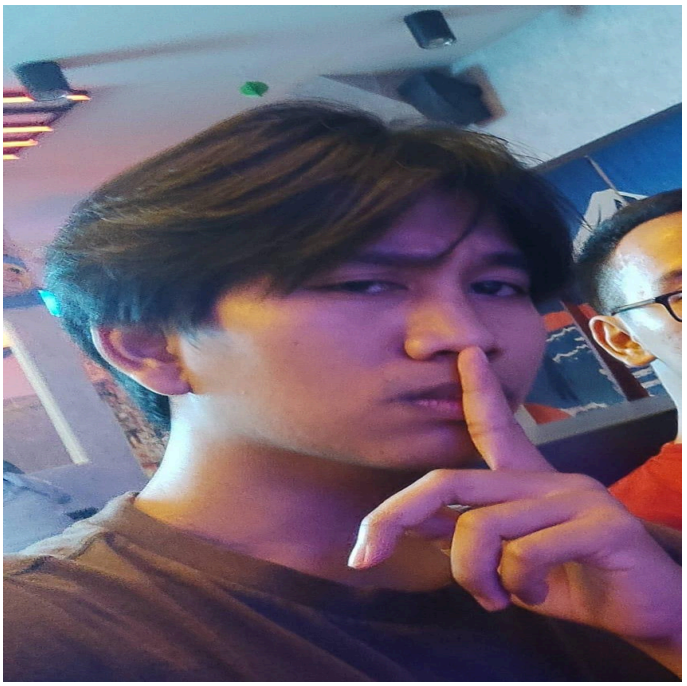
IF2224 - Teori Bahasa Formal dan Automata

Laporan Milestone 1 Tugas Besar 1

Lexical Analysis



Kelompok 10 - BetterCallPascal

Julius Arthur	13523030
Samuel Gerrard H. Girsang	13523064
Nadhif Al Rozin	13523076
Lutfi Hakim Yusra	13523084

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
JL. GANESHA 10, BANDUNG 40132

2025

DAFTAR ISI

LANDASAN TEORI

1. Finite Automata

Finite Automata adalah model matematika yang digunakan untuk pengenalan pola dari aliran karakter. Model ini punya sejumlah keadaan (states) yang terbatas dan dapat berpindah dari satu keadaan ke keadaan lain sebagai respons terhadap karakter input. Sebuah string masukan diterima oleh FA, jika setelah memproses seluruh simbol dalam string tersebut dari kiri ke kanan, FA berhenti di salah satu keadaan akhir. Jika tidak, string tersebut ditolak. Dalam mendesain compiler, FA memegang peran yang krusial dalam fase pertama kompilasi yaitu analisis leksikal.

Secara formal, sebuah FA didefinisikan sebagai tuple dari 5 variabel. M = (Q, Σ, δ, q0, F). Q adalah himpunan berhingga dari keadaan. Σ adalah himpunan berhingga simbol masukan, atau disebut juga alfabet. Δ adalah fungsi transisi (Q x Σ => Q), yang memetakan pasangan state dan simbol masukan ke keadaan berikutnya. q0 adalah keadaan awal, di mana q0 ∈ Q. F adalah himpunan keadaan akhir, di mana F subset Q.

Ada dua jenis utama Finite Automata yang relevan yaitu Deterministic Finite Automaton (DFA) dan Nondeterministic Finite Automaton (NFA). Dalam DFA, untuk setiap pasangan state dan simbol masukan, hanya ada satu kemungkinan transisi ke keadaan berikutnya. Fungsi transisinya bersifat tunggal. DFA lebih cepat dalam proses pengenalan pola tetapi bisa lebih kompleks untuk dibangun. Dalam NFA, dari sebuah keadaan, bisa terdapat nol, satu, atau lebih transisi untuk simbol masukan yang sama. NFA juga dapat memiliki transisi epsilon (ϵ), yaitu transisi yang terjadi tanpa memerlukan simbol masukan. NFA lebih mudah dirancang dari regex tetapi lebih lambat dalam eksekusi karena harus melacak beberapa kemungkinan jalur secara bersamaan. Meskipun berbeda, setiap NFA dapat dikonversi menjadi DFA yang ekuivalen melalui algoritma seperti subset construction.

2. Lexical Analyzer

Ini adalah fase pertama dalam proses kompilasi yang bertugas membaca *source code* sebagai aliran karakter dan mengubahnya menjadi unit yang bermakna (Token). Proses ini mengidentifikasi *lexeme* (Rangkaian karakter) dan mengklasifikasikannya ke dalam jenis token tertentu, seperti keyword, identifier, atau number berdasarkan sebuah *rule* yang disebut *pattern*. Tujuan fase ini adalah menyederhanakan input untuk fase selanjutnya dengan mengabaikan hal-hal yang tidak relevan dan mengelompokkannya menjadi unit logis.

Analisis lexical berakar kuat pada teori bahasa formal, Regex (Ekspresi Reguler) digunakan sebagai notasi matematis untuk mendefinisikan *pattern* dari tiap token. Selanjutnya setiap regex pasti dapat dikonversi menjadi sebuah mesin pengenal, Finite Automata (FA). Hal ini karena regex dan FA adalah cara yang setara untuk

mendeskripsikan sebuah bahasa. Hanya berbeda di penyajiannya, FA seperti jalur atau *flowchart* sedangkan regex adalah pola atau rumus singkat.

Secara praktis, pemisahan fase ini dari fase lain memberikan keuntungan yang signifikan dalam mendesain kompilator karena modularitas. Dimana tugas yang awalnya kompleks dibagi menjadi masalah yang lebih sederhana. Dengan demikian parser (fase selanjutnya) tidak perlu lagi dibebani oleh detail level karakter. Selain dari membantu maintenance dan pengembangan, hal ini juga meningkatkan efisiensi karena algoritma berbasis DFA sangat cepat dalam memproses input linear.

3. Pascal-S

Pascal-S adalah subset dari pascal yang dikembangkan oleh Niklaus Wirth fdi ETH Zurich, yang tujuannya untuk menyediakan sistem kompilasi-interpretasi yang tunggal untuk lingkungan pembelajaran. Bahasa ini menghilangkan sejumlah fitur penuh dari pascal seperti set, dynamic pointer, parameter prosedur dan fungsi.

Dalam aspek implementasi, Pascal-S menggunakan struktur arsitektur yang memadukan compiler yang menerjemahkan kode sumber ke suatu bentuk antara (misalnya p-code atau instruksi internal) dan kemudian interpreter yang menjalankan bentuk antara tersebut. Compiler menghasilkan array variabel internal yang kemudian dieksekusi oleh interpreter. Pendekatan ini memungkinkan pemisahan tahap-tahap compiler (analisis dan penerjemahan) dan tahap eksekusi, sekaligus menjadikan sistem cukup sederhana untuk dipahami dan dibangun ulang.

PERANCANGAN DAN IMPLEMENTASI

1. Diagram Planning

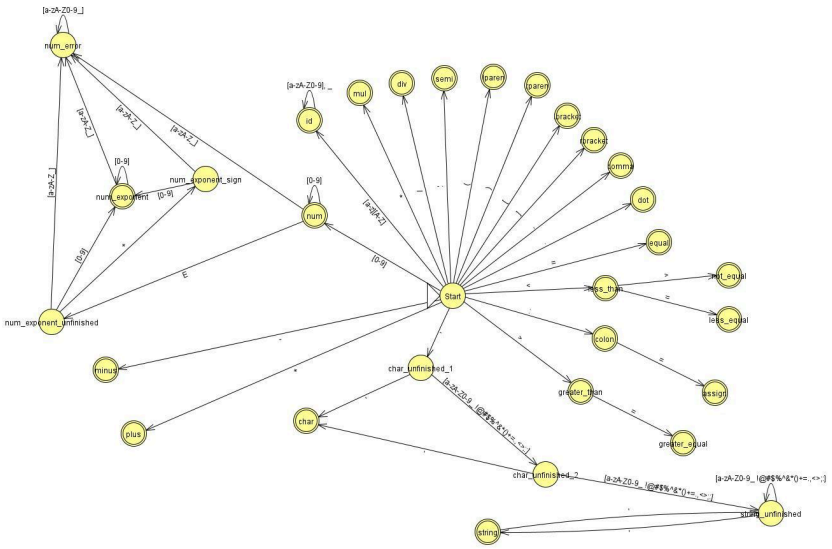


Diagram di atas merupakan DFA (Deterministic Finite Automata) yang digunakan untuk melakukan *lexical analysis* terhadap kode program Pascal-S. Kode program diproses simbol per simbol, mengikuti alur transisi sesuai dengan simbol yang dimasukkan. Ketika menerima sebuah simbol yang tidak memiliki alur transisi pada sebuah state, maka state tersebut akan menimbulkan token yang sesuai jika merupakan *accepting state*, dan menimbulkan error jika tidak. Setelah itu, pencarian akan diulang dari start state. Dapat dilihat bahwa terdapat empat jenis percabangan tokenisasi utama: *short symbols* (mul, div, semi, …), *string*, *identifier* (id), *number* (num, num_exponent).

Percabangan s*hort symbol* sederhana, hanya memeriksa satu atau dua simbol sebelum memunculkan token, dan tidak memiliki alur error. Sebuah start state yang membaca *short symbol* pasti akan memunculkan token tersebut.

Percabangan *string* mengikuti peraturan string PASCAL-S, yaitu sebuah string pasti dimulai dan diakhiri dengan petik tunggal (‘). Selain itu, untuk menulis petik tunggal pada sebuah string, harus menerima dua petik tunggal (‘’). String juga tidak bisa mengandung newline. Sebuah string dikatakan char jika jarak antara petik tunggal merupakan satu atau nol, sehingga percabangan ini dimulai dengan char sebelum masuk ke string.

Percabangan *identifier* menerima awalan alfabet, yang kemudian dapat menerima alfanumerik dan juga *underscore* (_). Sebuah identifier tidak boleh mengandung strip (-), dan simbol-simbol lainnya. Analisis *identifier* juga memanfaatkan sebuah *hash lookup* untuk mengklasifikasikan token keyword yang ditangkap dari state akhir *identifier*. Ketika terdapat sebuah string yang merupakan keyword dan berakhir di state akhir *identifier*, maka akan melepaskan token KEYWORD.

Percabangan *number* menerima awalan angka, yang dapat dilanjutkan dengan angka lagi. Jika dilanjutkan dengan ‘e’, maka harus dilanjutkan lagi dengan sebuah angka agar menjadi

sebuah eksponen, jika tidak maka error. Untuk setiap *accepting state* pada percabangan ini, terdapat transisi menuju num_error jika menambahkan huruf. Ini merupakan perubahan state penting agar analisis untuk kumpulan simbol '123abc' dianggap sebagai error, dan bukan menimbulkan number dan identifier.

2. Rule File Representation

Rule file memanfaatkan struktur file JSON dalam mengandung semua peraturan. Dengan JSON, kondisi keterbacaan semua transisi dari sebuah state menjadi lebih baik, dan *parsing* sebuah JSON dipermudah dengan modul import json dari Python.

Nama Atribut	Penjelasan
start_state	**string** state awal yang dilewati oleh lexer
final_states	**dict[string, string]** Accepting states beserta token yang bakal ditembakkan
keywords	**dict[string, string]** Keywords beserta token yang direpresentasikan oleh keyword tersebut
state	**dict[string,string]** Merepresentasikan sebuah state. Dictionary berguna untuk menampung transisi sebuah input

3. Automata Implementation

automata.py

```
class StateMachine:
    def __init__(self):
        self.transition: dict[tuple[str, str], str] = {}
        self.final_states: dict[str, str] = {} # state ->
token type
        self.start_state: str = "start"
        self.reserved_keywords: dict[str, str] = {}

    def add_transition(self, from_state: str, input_char:
str, to_state: str):
        input_char = input_char.lower()
        if (from_state, input_char) in self.transition:
            return
```

```
        self.transition[(from_state, input_char)] = to_state

    def add_final_state(self, state: str, token_type: str =
"IDENTIFIER"):
        self.final_states[state] = token_type

    def set_start_state(self, start_state: str):
        self.start_state = start_state

    def add_reserved_keyword(self, keyword: str, token_type:
str = "KEYWORD"):
        self.reserved_keywords[keyword] = token_type

    def get_next_state(self, current_state: str, input_char:
str) -> str | None:
        return self.transition.get((current_state,
input_char))

    def get_reserved_keywords(self) -> dict[str, str]:
        return self.reserved_keywords

    def is_accepting_state(self, state: str) -> bool:
        return state in self.final_states

    def get_token_type(self, state: str) -> str | None:
        return self.final_states.get(state)

    def get_start_state(self) -> str:
        return self.start_state

class Automata:
    def __init__(self, state_machine: StateMachine):
        self.state_machine: StateMachine = state_machine
        self.current_state: str = state_machine.start_state
        self.buffer: str = ""
        self.token_type: str = ""
        self.tokens: list[tuple[str, str]] = [] # List of
(token_type, token_value)f
        self.errors: list[str] = []

    def reset(self):
        self.current_state = self.state_machine.start_state
```

```
        self.buffer = ""
        self.token_type = ""

    def process_char(self, char: str) -> bool:
        char = char.lower()
        next_state: str | None =
self.state_machine.get_next_state(self.current_state, char)
        if next_state:
            # Valid transition
            self.current_state = next_state
            if next_state != self.state_machine.start_state:
                self.buffer += char # ignore chars leading to
start state
            if
self.state_machine.is_accepting_state(self.current_state):
                self.token_type =
self.state_machine.get_token_type(self.current_state)
            return True
        elif
self.state_machine.is_accepting_state(self.current_state):
            # No valid transition, but in accepting state.
Record token and reset, then reprocess char
            if self.buffer:
                if self.buffer in
self.state_machine.get_reserved_keywords():
                    Self.token_type =
self.state_machine.get_reserved_keywords()[self.buffer]
                self.tokens.append((self.token_type,
self.buffer))
            self.reset()
            return self.process_char(char)
        # No valid transition and not in accepting state:
LEXICAL ERROR
        self.errors.append(f"Unexpected character '{char}' in
state '{self.current_state}' with buffer '{self.buffer}'")
        self.reset()
        return False

    def finalize(self):
        if
self.state_machine.is_accepting_state(self.current_state) and
self.buffer:
```

```
            self.tokens.append((self.token_type,
self.buffer))
        elif self.buffer:
            self.errors.append(f"Incomplete token
'{self.buffer}' in state '{self.current_state}' at end of
input")
        self.reset()

    def print_tokens(self):
        for token_type, token_value in self.tokens:
            print(f"{token_type}({token_value})")

    def save_tokens(self, file_path: str =
"../test/milestone-1/tokens.txt"):
        with open(file_path, 'w') as file:
            for token_type, token_value in self.tokens:
                file.write(f"{token_type}({token_value})\n")

    def get_tokens(self) -> list[tuple[str, str]]:
        return self.tokens

    def get_errors(self) -> list[str]:
        return self.errors

    def analyze(self, input_string: str): # Use this method
to process an entire string
        for char in input_string:
            self.process_char(char)
        self.finalize()
```

Penjelasan	
StateMachine	StateMachine merupakan representasi DFA dalam bentuk sebuah dictionary yang menerima key berbentuk (State, next_symbol) untuk mengarah ke State berikutnya. Start state merupakan awal. Keywords ditampung di reserved_keywords.
	add_transition() merupakan fungsi yang menambahkan transisi dari sebuah state menuju state lainnya add_final_state() merupakan fungsi yang menambahkan state akhir dan juga token yang direpresentasikannya
	get_next_state() merupakan fungsi untuk memanfaatkan

	lookup dictionary untuk menentukan arah dari sebuah input. Ini dapat mengembalikan output kosong.
	get_token_type() merupakan fungsi untuk memanfaatkan lookup final_states untuk mengetahui token yang dimunculkan dari state tersebut. Ini dapat mengembalikan output kosong.
Automata	Automata merupakan sistem lexer untuk mengolah simbol menjadi token dengan memanfaatkan DFA. Kelas ini menampung kumpulan tokens dan errors, dan juga state.
	process_char() menerima symbol berikutnya, dan mengolahnya berdasarkan state saat ini. Jika terdapat transisi, gerak ke state tersebut. Jika tidak, cek apakah ini merupakan *accepting state*. State yang merupakan *accepting state* akan menimbulkan token, dan mereset lexer, sebelum memproses simbol berikutnya lagi dari start_state. Jika bukan *accepting state*, maka error akan ditimbulkan, dan balik ke start_state.
	save_tokens() menyimpan semua token pada sebuah file
	analyze() menerima sebuah string untuk dikonsumsi.

rule_reader.py

```
class RuleReader:
    @staticmethod
    def expand_char_range(char_range: str) -> list[str]:
        list_chars: list[str] = []
        i = 0
        while i < len(char_range):
            if i + 2 < len(char_range) and char_range[i + 1]
== '-':
                start_char, end_char = char_range[i],
char_range[i + 2]

                list_chars.extend([chr(c) for c in
range(ord(start_char), ord(end_char) + 1)])

                i += 3
            else:
                list_chars.append(char_range[i])
```

```
                i += 1

        return list_chars

    @staticmethod
    def from_file(file_path: str) -> StateMachine:
        with open(file_path, 'r') as file:
            rules: dict = json.load(file)
        state_machine = StateMachine()

state_machine.set_start_state(rules.get("start_state"))
        for final_state, token_type in
rules.get("final_states", {}).items():
            state_machine.add_final_state(final_state,
token_type)

        for keyword, token_type in rules.get("keywords",
{}).items():
            state_machine.add_reserved_keyword(keyword,
token_type)

        for state, transitions in rules.items():
            if state in ["start_state", "final_states",
"keywords"]:
                continue
            for input_char, next_state in
transitions.items():
                if len(input_char) > 1:
                    expanded_chars =
RuleReader.expand_char_range(input_char)
                    for char in expanded_chars:
                        state_machine.add_transition(state,
char, next_state)
                else:
                    state_machine.add_transition(state,
input_char, next_state)

        return state_machine

if __name__ == "__main__":
    sm =
RuleReader.from_file("../test/milestone-1/input.json")
```

```
    automata = Automata(sm)
    test_code = "../test/milestone-1/test.pas"
    with open(test_code, 'r') as f:
        test_code = f.read()


    for char in test_code:
        automata.process_char(char)
    automata.finalize()
    automata.print_tokens()
    automata.errors and print("Errors:", automata.errors)
    automata.save_tokens()
```

Penjelasan	
RuleReader	RuleReader merupakan sebuah kelas pembantu untuk pemrosesan rule file yang akan menghasilkan automata.
	from_file() mengubah sebuah rule file menjadi sebuah state machine. Untuk sebuah rule yang overlap, utamakan pertama yang muncul.
	expand_char_range() mengubah sebuah range yang mirip pada regex ([a-z]) menjadi kumpulan char (a, b, c … , y, z) untuk menyingkatkan rule file.

main.py

```
from rule_reader import RuleReader
from automata import Automata
def main():
    sm = RuleReader.from_file("../test/milestone-1/input.json")
    automata = Automata(sm)
    test_code = "../test/milestone-1/"
    filename = input("Enter test file name (e.g., test.pas): ")
    test_code += filename
    with open(test_code, 'r') as f:
        test_code = f.read()


    for char in test_code:
```

```
            automata.process_char(char)
        automata.finalize()
        automata.print_tokens()
        for error in automata.errors:
            print("LEXICAL ERROR:", error)
        automata.save_tokens()

if __name__ == "__main__":
    main()
```

main	Main merupakan fungsi yang dijalankan untuk menjalankan program secara keseluruhan.

PENGUJIAN

1. **test1.pas**

Source Code

```
program Test1;
begin
  x := 5..9
  y := x-9
  char := 'h'
  string := 'wowwww'
end.
```

Tokens

KEYWORD(program)
IDENTIFIER(test1)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(x)
ASSIGN_OPERATOR(:=)
NUMBER(5)
DOT(.)
DOT(.)
NUMBER(9)
IDENTIFIER(y)
ASSIGN_OPERATOR(:=)
IDENTIFIER(x)
ARITHMETIC_OPERATOR(-)
NUMBER(9)
KEYWORD(char)
ASSIGN_OPERATOR(:=)
CHAR_LITERAL('h')
IDENTIFIER(string)
ASSIGN_OPERATOR(:=)
STRING_LITERAL('wowwww')
KEYWORD(end)
DOT(.)

Errors (jika ada)

2. **test2.pas**

Source Code

```
program test2;
begin
  2Times := 4;
  Tax-Rate := 0.05;
  BadString := 'Error string ;
end.
```

Tokens

KEYWORD(program)
IDENTIFIER(test2)
SEMICOLON(;)
KEYWORD(begin)
ASSIGN_OPERATOR(:=)
NUMBER(4)
SEMICOLON(;)
IDENTIFIER(tax)
ARITHMETIC_OPERATOR(-)
IDENTIFIER(rate)
ASSIGN_OPERATOR(:=)
NUMBER(0)
DOT(.)
NUMBER(05)
SEMICOLON(;)
IDENTIFIER(badstring)
ASSIGN_OPERATOR(:=)
KEYWORD(end)
DOT(.)

Errors (jika ada)

```
LEXICAL ERROR: Unexpected character ' ' in state 'num_error' with buffer '2times'
LEXICAL ERROR: Unexpected character '
' in state 'string_unfinished' with buffer ''error string ;'
```

3. **test3.pas**

Source Code

```
program test3;
begin
  is_equal := 10 = 10;
  is_less := 5 < 10;
```

```
  is_not_equal := 5 <> 10;

  result_bool := is_equal and is_less;
  final_bool := is_not_equal or result_bool;
end.
```

Tokens

KEYWORD(program)
IDENTIFIER(test3)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(is_equal)
ASSIGN_OPERATOR(:=)
NUMBER(10)
RELATIONAL_OPERATOR(=)
NUMBER(10)
SEMICOLON(;)
IDENTIFIER(is_less)
ASSIGN_OPERATOR(:=)
NUMBER(5)
RELATIONAL_OPERATOR(<)
NUMBER(10)
SEMICOLON(;)
IDENTIFIER(is_not_equal)
ASSIGN_OPERATOR(:=)
NUMBER(5)
RELATIONAL_OPERATOR(<>)
NUMBER(10)
SEMICOLON(;)
IDENTIFIER(result_bool)
ASSIGN_OPERATOR(:=)
IDENTIFIER(is_equal)
LOGICAL_OPERATOR(and)
IDENTIFIER(is_less)
SEMICOLON(;)
IDENTIFIER(final_bool)
ASSIGN_OPERATOR(:=)
IDENTIFIER(is_not_equal)
LOGICAL_OPERATOR(or)
IDENTIFIER(result_bool)
SEMICOLON(;)
KEYWORD(end)
DOT(.)

Errors (jika ada)

4. **test4.pas**

Source Code

```
program test4;
begin
    i := 0;
    j := 10;
    k := Max(i, j);

    x := 1.5;
    y := x * 2.0 + 3.14 / (x + 1.0);
    message := 'Hello, Lexer!';
    done := (x > y) or (not (x < y));
end.
```

Tokens

KEYWORD(program)
IDENTIFIER(test4)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
NUMBER(0)
SEMICOLON(;)
IDENTIFIER(j)
ASSIGN_OPERATOR(:=)
NUMBER(10)
SEMICOLON(;)
IDENTIFIER(k)
ASSIGN_OPERATOR(:=)
IDENTIFIER(max)
LPAREN(()
IDENTIFIER(i)
COMMA(,)
IDENTIFIER(j)
RPAREN())
SEMICOLON(;)
IDENTIFIER(x)
ASSIGN_OPERATOR(:=)
NUMBER(1)
DOT(.)
NUMBER(5)
SEMICOLON(;)
IDENTIFIER(y)

ASSIGN_OPERATOR(:=)
IDENTIFIER(x)
ARITHMETIC_OPERATOR(*)
NUMBER(2)
DOT(.)
NUMBER(0)
ARITHMETIC_OPERATOR(+)
NUMBER(3)
DOT(.)
NUMBER(14)
ARITHMETIC_OPERATOR(/)
LPAREN(()
IDENTIFIER(x)
ARITHMETIC_OPERATOR(+)
NUMBER(1)
DOT(.)
NUMBER(0)
RPAREN())
SEMICOLON(;)
IDENTIFIER(message)
ASSIGN_OPERATOR(:=)
STRING_LITERAL('hello, lexer!')
SEMICOLON(;)
IDENTIFIER(done)
ASSIGN_OPERATOR(:=)
LPAREN(()
IDENTIFIER(x)
RELATIONAL_OPERATOR(>)
IDENTIFIER(y)
RPAREN())
LOGICAL_OPERATOR(or)
LPAREN(()
LOGICAL_OPERATOR(not)
LPAREN(()
IDENTIFIER(x)
RELATIONAL_OPERATOR(<)
IDENTIFIER(y)
RPAREN())
RPAREN())
SEMICOLON(;)
KEYWORD(end)
DOT(.)

Errors (jika ada)

5. **test5.pas**

Source Code

```
program test5;

var
    ifx, theny, z123, _hidden : integer;
    a, b : real;
    s : string;

begin
    a := 0.00123E+4;
    s := 'hello ''world''';
    ifx := 10;
    theny := ifx + 5;
    _hidden := theny - 3*2/4;

    if (a >= b) and not (ifx = 0) then
        writeln('a >= b')
    else
        writeln('a < b');
end.
```

Tokens

KEYWORD(program)
IDENTIFIER(test5)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(ifx)
COMMA(,)
IDENTIFIER(theny)
COMMA(,)
IDENTIFIER(z123)
COMMA(,)
IDENTIFIER(hidden)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(a)
COMMA(,)
IDENTIFIER(b)
COLON(:)
KEYWORD(real)
SEMICOLON(;)

```
IDENTIFIER(s)
COLON(:)
IDENTIFIER(string)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(a)
ASSIGN_OPERATOR(:=)
NUMBER(0)
DOT(.)
NUMBER(00123e+4)
SEMICOLON(;)
IDENTIFIER(s)
ASSIGN_OPERATOR(:=)
STRING_LITERAL('hello "world"')
SEMICOLON(;)
IDENTIFIER(ifx)
ASSIGN_OPERATOR(:=)
NUMBER(10)
SEMICOLON(;)
IDENTIFIER(theny)
ASSIGN_OPERATOR(:=)
IDENTIFIER(ifx)
ARITHMETIC_OPERATOR(+)
NUMBER(5)
SEMICOLON(;)
IDENTIFIER(hidden)
ASSIGN_OPERATOR(:=)
IDENTIFIER(theny)
ARITHMETIC_OPERATOR(-)
NUMBER(3)
ARITHMETIC_OPERATOR(*)
NUMBER(2)
ARITHMETIC_OPERATOR(/)
NUMBER(4)
SEMICOLON(;)
KEYWORD(if)
LPAREN(()
IDENTIFIER(a)
RELATIONAL_OPERATOR(>=)
IDENTIFIER(b)
RPAREN())
LOGICAL_OPERATOR(and)
LOGICAL_OPERATOR(not)
LPAREN(()
IDENTIFIER(ifx)
RELATIONAL_OPERATOR(=)
NUMBER(0)
RPAREN())
KEYWORD(then)
```

IDENTIFIER(writeln)
LPAREN(()
STRING_LITERAL('a >= b')
RPAREN())
KEYWORD(else)
IDENTIFIER(writeln)
LPAREN(()
STRING_LITERAL('a < b')
RPAREN())
SEMICOLON(;)
KEYWORD(end)
DOT(.)

Errors (jika ada)

```
LEXICAL ERROR: Unexpected character '_' in state 'start' with buffer ''
LEXICAL ERROR: Unexpected character '_' in state 'start' with buffer ''
```

KESIMPULAN DAN SARAN

Lexer merupakan analisis tingkat terendah dari proses *compiling* yang menghasilkan token-token untuk kemudian diproses di tahap berikutnya. Dari pengujian yang telah dilalui, *lexer* berbasis DFA sudah cukup kompeten dalam memproses kode program berbahasa Pascal-S menjadi token-token yang sesuai. Walaupun begitu, dalam proses pengerjaan terlihat jelas batasan dari lexer berbasis DFA (*Deterministic Finite Automata*) murni pada beberapa *edge case*. Dihilangkannya tokenisasi bilangan negatif, bilangan desimal, bilangan eksponen, komen, range merupakan hasil dari ambiguitas yang muncul akibat ketidakmampuannya *lexer* DFA murni untuk melakukan *look-ahead* atau *context*. Oleh karena itu, tanggung jawab untuk mengolah lebih lanjut token-token yang ada merupakan tanggung jawab *parser*, yaitu tahap berikutnya.

Pengerjaan Tugas Besar Milestone 1 Teori Berbahasa Formal Otomata ini berhasil memperluas wawasan mengenai pemanfaatan DFA dalam lexer. Dalam segi teknis, kami menjadi paham tentang tahap pertama dalam proses kompilasi, yang sebelumnya merupakan hal yang hanya diketahui secara samar-samar. Dalam segi non-teknis, dari kami, sebaiknya pengerjaan dilakukan dengan pembagian tugas yang jelas dan memperbanyak pertemuan antara anggota kelompok agar lancar dalam pengerjaan. Selain itu, mengenai spesifikasi pengerjaan tugas, terdapat momen-momen yang dapat diperbaiki. Kebingungan mengenai ambiguitas pada *lexer* merupakan hal yang banyak dipertanyakan tapi cukup lama untuk dijawab, sehingga menghambat dalam penyelesaian. Selain itu, resolusi dari permasalahannya sebaiknya dicapai dengan memberlakukan sebuah revisi secepatnya terhadap dokumen spesifikasi jika terdapat perubahan yang drastis, supaya dokumen spesifikasi tidak *outdated* dan menyesatkan orang yang tidak dikabarkan mengenai beberapa perubahan spesifik.

LAMPIRAN

- Link Github Release Repository: https://github.com/pixelatedbus/BCP-Tubes-IF2224
- Diagram memanfaatkan Jflaps, sehingga tidak ada *workspace*.

Pembagian Tugas	
Julius Arthur (13523030)	Rule planning, Diagram creation
Samuel Gerrard Hamonangan Girsang (13523064)	Rule planning, document
Nadhif Al-Rozin (13523076)	Rule planning, document
Lutfi Hakim Yusra (13523084)	Rule planning, document, automata implementation, rule file implementation, lexer implementation

-

REFERENSI

- “Compilers: Principles, Techniques, and Tools” Buku dari Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. Link : https://repository.unikom.ac.id/48769/1/Compilers%20-%20Principles,%20Techniques,%20and%20Tools%20(2006).pdf
- "Engineering a Compiler" oleh Keith D. Cooper and Linda Torczon. Link : https://www.r-5.org/files/books/computers/compilers/writing/Keith_Cooper_Linda_Torczon-Engineering_a_Compiler-EN.pdf
- https://www.moorecad.com/standardpascal/pascals.html?utm_source=chatgpt.com
-