Milestone 1 *Lexical Analysis*

IF2224 *Formal Language and Automata Theory*

Laboratorium Ilmu Rekayasa dan Komputasi

Oleh:

Kelompok HJE - HidupJalaninAje

Sebastian Enrico Nathanael (13523134)

Jonathan Kenan Budianto (13523139)

Mahesa Fadhillah Andre (13523140)

Muhammad Farrel Wibowo (13523153)

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika - Institut Teknologi Bandung

Jl. Ganesha 10, Bandung 40132

2025

Daftar Isi

BAB I
Landasan Teori

Proses kompilasi merupakan suatu rangkaian tahapan yang saling terhubung dan berurutan. Setiap tahapan menghasilkan keluaran yang menjadi input untuk tahap berikutnya. Oleh karena itu, sangat penting untuk memastikan bahwa hasil dari setiap langkah sudah benar agar tidak mengganggu kelancaran proses selanjutnya. Dalam tugas ini, kami memanfaatkan lima tahap utama dalam proses kompilasi, yaitu ***Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generation, dan Interpreter.*** Berikut adalah penjelasan lebih lanjut mengenai setiap tahapan tersebut:

1.1 *Lexical Analysis*

Tahapan ini bertujuan untuk mengubah *source code* Pascal-S dari kumpulan karakter mentah menjadi unit-unit bermakna yang disebut token. Setiap token memiliki jenis (*type*) dan nilai (*value*), serta akan menjadi masukan bagi tahap berikutnya dalam proses kompilasi. Analisis leksikal adalah tahap pertama dalam proses kompilasi atau interpretasi bahasa pemrograman. Pada tahap ini, compiler membaca source code mentah yang terdiri dari rangkaian karakter dan mengubahnya menjadi serangkaian satuan makna yang disebut **Token**. Proses ini dilakukan oleh komponen yang disebut Lexer.

Tujuan utama lexer adalah untuk mempermudah tahap berikutnya (*parsing*) dengan membuang karakter yang tidak diperlukan (seperti *whitespace* dan komentar) dan menyediakan input yang terstruktur berupa list token.

Masukan	Kode Pascal-S (.pas)
Keluaran	List Token (misalnya VAR(X), ASSIGN(=), OPERATOR(+))
Otomata	Deterministic finite automata (DFA)

Contoh input kode PASCAL-S

program Hello;
 	
var
  a, b: integer;
 	
begin
  a := 5;
  b := a + 10;
  writeln('Result = ', b);

```
end.
```

Contoh output list of tokens

```
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(.)
```

1.3 Token

Token adalah unit terkecil yang memiliki makna dalam suatu program atau kode sumber. Setiap token mewakili elemen-elemen dasar dalam bahasa pemrograman, seperti kata kunci (*keyword*), nama variabel (*identifier*), operator, angka, atau tanda baca. Token adalah hasil dari proses analisis leksikal yang dilakukan oleh lexer (penganalisis leksikal) dalam tahap pertama kompilasi.
Setiap token memiliki dua komponen utama:

1. Jenis Token (*Type*): Ini adalah kategori atau jenis token yang mengklasifikasikan token tersebut. Misalnya, token dapat berupa kata kunci (seperti *if, while*), identifier (seperti nama variabel atau fungsi), operator (seperti +, -, *), angka (seperti 5, 100), atau tanda baca (seperti titik koma ;, koma ,).
2. Nilai Token (*Value*): Ini adalah nilai spesifik yang terkait dengan token tersebut. Misalnya, jika token adalah NUMBER, nilai tokennya bisa berupa angka seperti 5 atau 10. Jika token adalah IDENTIFIER, nilai tokennya bisa berupa nama variabel, misalnya a, sum, atau count.

Contoh jenis-jenis token dalam bahasa Pascal-S (sebagaimana dicontohkan dalam dokumen Anda):

1. KEYWORD: Kata kunci seperti program, var, begin, dan sebagainya, yang memiliki makna khusus dalam bahasa pemrograman.
2. IDENTIFIER: Nama variabel, fungsi, atau prosedur yang didefinisikan oleh pengguna program, misalnya a, x, sum.
3. NUMBER: Bilangan yang dapat berupa integer atau bilangan riil (desimal), misalnya 5, 3.14.
4. OPERATOR: Simbol atau kata yang digunakan untuk operasi matematis atau logika, seperti +, -, *, div, mod, and, or.
5. SEPARATOR: Tanda baca yang digunakan untuk memisahkan elemen dalam program, seperti titik koma (;), koma (,), dan tanda kurung ((), []).

Fungsi Token dalam Kompilasi
Token memainkan peran penting dalam tahap parsing. Setelah kode sumber diubah menjadi serangkaian token, parser dapat lebih mudah menganalisis struktur program dengan bekerja dengan token daripada dengan karakter mentah. Dengan kata lain, token membantu menyederhanakan dan mengorganisir kode sumber untuk mempermudah analisis logika program pada tahap selanjutnya.

1.4 Lexeme

Lexeme adalah urutan karakter aktual dalam kode sumber yang cocok dengan pola atau aturan yang ditetapkan oleh lexer untuk sebuah token. Lexeme adalah representasi literal dari nilai token dalam kode sumber. Lexeme dapat dipahami sebagai "cangkang" atau "wujud" dari sebuah token yang terdapat dalam kode sumber. Sebagai contoh, jika sebuah token adalah KEYWORD(program), maka lexeme yang sesuai adalah kata "program" dalam kode sumber. Begitu juga dengan token NUMBER(5), lexeme yang sesuai adalah urutan karakter "5" yang muncul dalam kode.

Hubungan Antara Token dan Lexeme

Perbedaan utama antara token dan lexeme adalah sebagai berikut:

1. Token adalah konsep yang lebih abstrak dan terkait dengan jenis atau kategori elemen dalam program.
2. Lexeme adalah representasi konkrit atau literal dari token yang ditemukan dalam kode sumber.

Untuk lebih jelasnya, mari kita lihat contoh berikut:

Contoh Kode Sumber:

```
program Hello;
var
  a, b: integer;
begin
  a := 5;
  b := a + 10;
  writeln('Result = ', b);
End.
```

Contoh Token dan Lexeme:

- Token: KEYWORD(program) -> Lexeme: program
- Token: IDENTIFIER(Hello) -> Lexeme: Hello
- Token: SEMICOLON(;) -> Lexeme: ;
- Token: ASSIGN_OPERATOR(:=) -> Lexeme: :=
- Token: NUMBER(5) -> Lexeme: 5

1.3 Deterministic Finite Automata (DFA)

Deterministic Finite Automaton (DFA) adalah model matematis yang digunakan untuk mengenali pola dalam rangkaian karakter, terutama dalam tahap analisis leksikal kompilasi.

Berikut adalah penjelasan lebih rinci mengenai DFA. DFA adalah model yang deterministik, yang berarti untuk setiap input, DFA hanya memiliki satu transisi yang pasti ke state berikutnya. Tidak ada pilihan atau kebebasan untuk memilih jalur yang berbeda, yang membuatnya lebih efisien dibandingkan dengan model lainnya, seperti Non-deterministic Finite Automata (NFA).

Komponen utama DFA:

1. State: Merupakan kondisi yang menggambarkan status dari DFA pada saat tertentu. Setiap state memiliki nama unik dan berfungsi untuk melacak posisi saat memproses input.
2. Start State: State pertama tempat DFA mulai memproses input. Biasanya disebut sebagai S0.
3. Final States: State yang menandakan bahwa urutan karakter yang telah diproses berhasil dikenali sebagai token yang valid. Jika DFA berakhir di final state, maka input tersebut dianggap sah.
4. Transition Function (Fungsi Transisi): Mendefinisikan bagaimana DFA berpindah antar state berdasarkan simbol input yang dibaca. Fungsi transisi ini dapat digambarkan dengan tabel atau diagram

Cara kerja DFA:

1. DFA mulai di start state (S0) dan membaca karakter pertama dari input.
2. Berdasarkan karakter yang dibaca, DFA akan berpindah ke state berikutnya sesuai dengan aturan transisi.
3. Proses ini berlanjut untuk setiap karakter input. Jika pada akhirnya DFA berakhir di salah satu final state, maka urutan karakter tersebut dianggap sebagai token yang valid.
4. Jika DFA tidak berakhir di final state setelah membaca seluruh input, maka urutan karakter tersebut dianggap tidak valid.

Keunggulan DFA:

1. **Deterministik**: Setiap input hanya memicu satu transisi, yang menghindari ambiguitas dan meningkatkan efisiensi.
2. **Kecepatan**: DFA bekerja dengan waktu linear terhadap panjang input, yang berarti semakin panjang input, semakin cepat DFA memprosesnya.
3. **Kesederhanaan**: Dibandingkan dengan NFA, DFA lebih mudah diimplementasikan karena tidak ada keadaan paralel yang perlu ditangani.

Kekurangan DFA:

1. **Jumlah state yang besar**: Untuk bahasa yang sangat kompleks, jumlah state yang diperlukan oleh DFA bisa sangat banyak, yang dapat membebani memori dan membuat implementasinya menjadi lebih sulit.

2. **Kurang fleksibel**: DFA tidak cocok untuk bahasa yang membutuhkan fleksibilitas dalam hal transisi antar state. Setiap karakter input memiliki jalur transisi yang pasti, yang membatasi fleksibilitas dalam menangani aturan yang lebih dinamis.

BAB II
Perencanaan & Implementasi

2.1 Pemilihan bahasa dan Justifikasi

Proyek compiler Pascal-S ini diimplementasikan menggunakan Python 3 sebagai bahasa pemrograman utama. Pemilihan bahasa ini didasarkan pada berbagai pertimbangan yang mendukung kemudahan dalam pengembangan, efisiensi dalam implementasi struktur data, serta kecepatan dalam siklus pengembangan. Berikut adalah justifikasi terkait pemilihan Python:

1. **Kemudahan Implementasi Struktur Data Kompleks**
   Python menyediakan berbagai struktur data built-in yang sangat mendukung pengelolaan informasi dalam proyek compiler ini. Misalnya, dalam implementasi DFA (Deterministic Finite Automaton), struktur *dictionary* digunakan untuk menyimpan transisi antar state. Hal ini mempermudah pemodelan dan pemrosesan data dalam bentuk yang lebih mudah dipahami dan dikelola. Selain itu, penggunaan *list* untuk menyimpan urutan token serta *dictionary* untuk memetakan kata kunci dan identifier pada simbol tabel juga meningkatkan efisiensi dalam pengelolaan data yang diperlukan.
2. **Manipulasi String dan File yang Efisien**
   Python dilengkapi dengan berbagai library built-in yang mendukung manipulasi string dan pengelolaan file dengan mudah. Misalnya, penggunaan modul *os* untuk File I/O mempermudah pembacaan file kode Pascal-S dan penulisan hasil output.
3. **Penanganan JSON untuk Konfigurasi DFA**
   Dalam proyek ini, aturan DFA disimpan dalam file eksternal dalam format JSON. Python menyediakan dukungan native untuk parsing JSON, sehingga memudahkan pembacaan aturan DFA dan penggunaan aturan tersebut dalam pemrosesan kode sumber.

2.2 Struktur Program

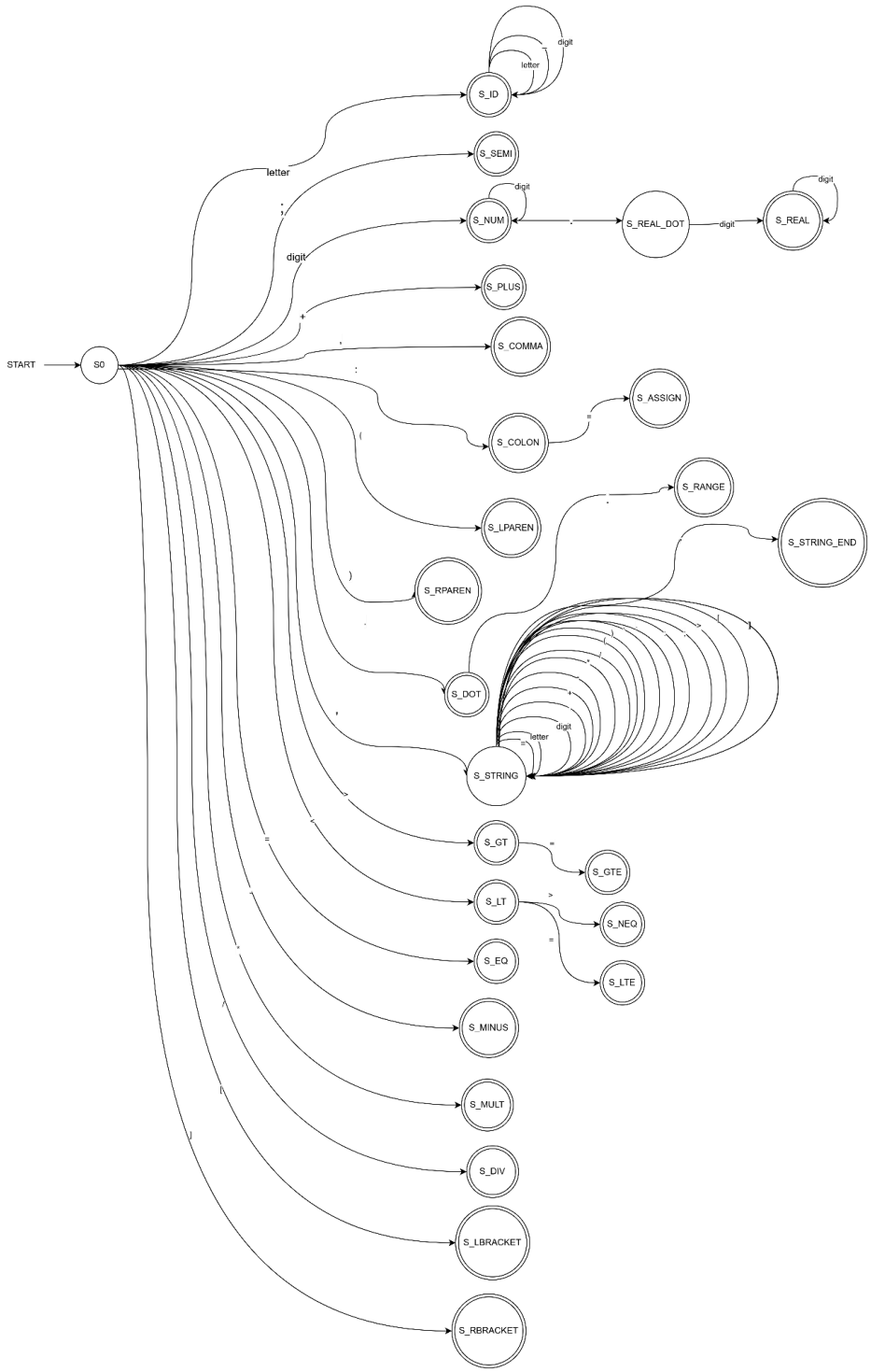
```
HJE-TUBES-IF2224/
|
|── doc/
|  └── header.png              # Gambar dokumentasi yang mungkin digunakan dalam laporan atau referensi visual
|
|── src/
|  ├── compiler.md            # Dokumentasi atau penjelasan tentang cara kerja compiler
|  ├── compiler.py            # File utama untuk menjalankan proses compiler
|  ├── dfa_rules.json         # Aturan DFA yang digunakan dalam analisis leksikal
|  ├── lexer.py               # Modul untuk lexical analysis (scanner) untuk menganalisis kode sumber
```

```
|   ├── pascal_token.py          # Modul untuk mendefinisikan tipe-tipe token dalam bahasa Pascal-S
|
├── test/
|   └── milestone-1/
|       ├── input/               # Folder untuk menyimpan file input kode Pascal-S yang akan diproses
|       └── output/              # Folder untuk menyimpan hasil output dari compiler
|
├── README.md                    # File dokumentasi umum mengenai proyek ini
```

Penjelasan Struktur:

1. **doc/:**
    a. Berisi file dokumentasi dan referensi seperti gambar yang dapat digunakan untuk penjelasan dalam laporan atau presentasi.
2. **src/:**
    a. compiler.py: Merupakan file utama untuk menjalankan proses kompilasi kode Pascal-S.
    b. dfa_rules.json: Berisi aturan DFA yang diperlukan oleh lexer untuk melakukan analisis leksikal.
    c. lexer.py: Modul yang menangani pemindaian dan analisis leksikal dari kode sumber untuk menghasilkan token.
    pascal_token.py: Berisi definisi dan representasi token yang digunakan dalam kode Pascal-S.
3. **test/:**
    a. Folder ini digunakan untuk menyimpan file input (kode Pascal-S) dan output yang dihasilkan oleh compiler untuk pengujian.
4. **README.md:**
    a. File dokumentasi utama yang menjelaskan tentang proyek ini, cara menjalankannya, dan dependensi yang diperlukan.

2.3 Penjelasan Diagram DFA



Link: [dfa diagram]

2.3.1. Start State (S0)

Diagram DFA dimulai pada start state, yaitu S0. Ini adalah titik awal dalam proses scanning kode sumber. Pada state ini, DFA menunggu untuk menerima input pertama dari kode sumber, seperti huruf, angka, atau simbol lainnya. Berdasarkan input pertama, DFA akan bergerak ke state berikutnya sesuai dengan aturan transisi yang sudah didefinisikan.

2.3.2. Final States

Final states dalam diagram DFA adalah state yang menandakan bahwa token tertentu telah dikenali. Setiap final state ini memiliki makna spesifik berdasarkan jenis token yang terdeteksi. Misalnya:

- S_ID: Menandakan token IDENTIFIER, yaitu variabel atau nama fungsi.
- S_NUM: Menandakan token NUMBER, yang bisa berupa bilangan bulat.
- S_REAL: Menandakan token REAL NUMBER atau bilangan desimal.
- S_SEMI: Menandakan token SEMICOLON, yaitu tanda titik koma.
- S_ASSIGN: Menandakan token ASSIGN_OPERATOR seperti :=.

Final states ini digunakan untuk memberi tahu DFA bahwa sebuah token telah berhasil dikenali dan siap untuk diproses lebih lanjut dalam langkah-langkah berikutnya.

2.3.3. Transisi antar State

Transisi antar state dalam DFA ini sangat tergantung pada simbol input yang diterima. Setiap simbol input (huruf, angka, operator, atau tanda baca) akan memicu transisi dari satu state ke state berikutnya. Misalnya:

- Dari S0 (start state), jika karakter yang dibaca adalah letter (huruf), DFA akan bergerak ke S_ID, yang menandakan bahwa input berikutnya adalah sebuah identifier.
- Jika karakter yang dibaca adalah digit (angka), DFA akan bergerak ke S_NUM, untuk mengenali bilangan.
- Jika karakter yang dibaca adalah operator +, DFA akan bergerak ke S_PLUS untuk mengenali operator aritmatika.

Setiap state dalam diagram DFA ini memiliki transisi yang jelas, yang memungkinkan DFA untuk memindai dan mengidentifikasi token dalam kode dengan efisien.

2.3.4. State Khusus untuk Literals dan Operators

Diagram DFA ini juga memiliki transisi khusus untuk mengenali berbagai jenis simbol dalam kode Pascal-S. Misalnya:

- S_STRING dan S_STRING_END: Digunakan untuk mengenali string literal dalam kode Pascal-S. Ketika DFA membaca tanda kutip tunggal ('), DFA akan memasuki state S_STRING dan melanjutkan pemindaian karakter string. Ketika ditemukan tanda kutip penutup ('), DFA akan bertransisi ke S_STRING_END, yang menandakan akhir dari string literal.
- S_REAL_DOT: Setelah membaca angka (S_NUM), jika ditemukan simbol titik ("."), DFA akan bertransisi ke S_REAL_DOT, yang menunjukkan bahwa input tersebut adalah bagian dari angka desimal.

2.3.5. Relational dan Arithmetic Operators

Diagram DFA ini juga menangani berbagai relational operators (seperti >, <, =, >=, <=, !=) dan arithmetic operators (seperti +, -, *, /). Misalnya:

- S_GT, S_LT, dan S_EQ: Menandakan operator relasional seperti greater than, less than, dan equal to.
- S_PLUS, S_MINUS, S_MULT, S_DIV: Menandakan operator aritmatika.

Pada diagram ini, transisi antara state disesuaikan dengan simbol yang terdeteksi, memastikan bahwa operator-operator ini dikenali dan diproses dengan benar.

2.4 Implementasi

compiler.py

```
import sys
import os
from lexer import Lexer
from pascal_token import Token

# KEYWORD Pascal-S
PASCAL_S_KEYWORDS = [
    # Keyword
    "program", "var", "begin", "end", "if", "then", "else", "while", "do",
    "for", "to", "downto", "integer", "real", "boolean", "char", "array",
    "of", "procedure", "function", "const", "type", "true", "false"
]

def main():
    #Penerimaan Input File
    if len(sys.argv) != 2:
        print("Penggunaan: python compiler.py [Kode Pascal]")
        return

    pascal_file = sys.argv[1]

    if not os.path.exists(pascal_file) or not os.path.isfile(pascal_file):
            print(f"File input '{pascal_file}' tidak ditemukan atau bukan file yang valid.")
            raise SystemExit(1)

    #Membaca kode Pascal-S
    try:
        with open(pascal_file, 'r') as f:
            source_code = f.read()
    except Exception as e:
        print(f"Gagal membaca file input: {e}")
        return

    #Inisialisasi Lexer
    dfa_path = os.path.join(os.path.dirname(__file__), "dfa_rules.json")
    try:
        lexer = Lexer(dfa_path, PASCAL_S_KEYWORDS)
    except SystemExit:
        return

    # 4. Melakukan Scanning
    tokens = lexer.run_scanner(source_code)

    # 5. Penghasilan Output
    if tokens:
        input_path = pascal_file
```

```
        input_filename = os.path.basename(input_path)

        milestone_dir = os.path.dirname(os.path.dirname(input_path))

        test_number = "".join(filter(str.isdigit, input_filename))
        output_filename = f"output-{test_number}.txt"

        output_dir = os.path.join(milestone_dir, "output")
        os.makedirs(output_dir, exist_ok=True)

        output_path = os.path.join(output_dir, output_filename)

        try:
            with open(output_path, 'w') as f:
                for token in tokens:
                    f.write(str(token) + '\n')
            print(f"Output berhasil ditulis ke: {output_path}")
        except Exception as e:
            print(f"Gagal menulis file output: {e}")

    else:
        print("Tidak ada token yang dihasilkan atau terjadi error.")

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

Function	*Class/Entity*	Deskripsi

main()	*Global Constant*	Fungsi utama yang dieksekusi. Sebagai alur program dari input hingga output. Input: Memeriksa argumen command line (sys.argv), memverifikasi path file Pascal-S, dan membaca konten file menjadi string source_code. Inisialisasi: Menginisialisasi objek Lexer dengan path aturan DFA dan list PASCAL_S_KEYWORDS. Eksekusi: Memanggil lexer.run_scanner(source_code). Output: Menulis daftar token yang dihasilkan ke file output (output-[n].txt) di direktori yang sesuai, dan menampilkan *comment* ketika berhasil/gagal
PASCAL_S_KEYWORDS		Fungsi utamanya adalah sebagai kamus leksikal yang digunakan oleh method get_token_type di class Lexer untuk menyelesaikan ambiguitas DFA (*Deterministic Finite Automata*). Dengan kata lain, karena DFA tidak dapat membedakan antara keyword dan identifier hanya dari pola karakter, Lexer memeriksa lexeme yang ditemukan terhadap daftar ini untuk mengklasifikasikannya dengan benar.

lexer.py

```
import json
import os
from pascal_token import Token

class Lexer:
    """
    Melakukan analisis leksikal menggunakan DFA yang dimuat dari file.
    """

    def __init__(self, dfa_file_path, keyword_list):
        self.keywords = keyword_list
        self.dfa = self.load_dfa(dfa_file_path)

        self.current_index = 0
        self.current_line = 1
        self.current_coloumn = 1

    def load_dfa(self, file_path):
        """
        Membaca dan memparsing file aturan DFA (JSON atau TXT).
        """
        #print(f"Menginisialisasi DFA dari: {file_path}")
        try:
            with open(file_path, 'r') as f:
                dfa_data = json.load(f)

                if 'start_state' not in dfa_data or 'final_states' not in dfa_data or
'transitions' not in dfa_data:
                    print("File aturan DFA tidak lengkap")
                    return None
            return dfa_data

        except json.JSONDecodeError as e:
            print(f"Format File DFA tidak valid {e}")
            return None

        except FileNotFoundError:
            print(f"File tidak ditemukan di {file_path}")
            return None

        except ValueError as e:
            print(f"Error pada konfigurasi DFA {e}")
            return None

    def get_token_type(self, lexeme, final_state):
        """
        Menentukan tipe token, termasuk mengecek apakah lexeme adalah KEYWORD atau
IDENTIFIER.
```

```
        """
        token_type = self.dfa['final_states'].get(final_state)
        lexeme_lower = lexeme.lower()

        if token_type == "IDENTIFIER_CANDIDATE":
            if lexeme_lower in ["and", "or", "not"]:
                return "LOGICAL_OPERATOR"
            elif lexeme_lower in ["div", "mod"]:
                return "ARITHMETIC_OPERATOR"
            elif lexeme_lower in [k.lower() for k in self.keywords]:
                return "KEYWORD"
            else:
                return "IDENTIFIER"

        if token_type == "STRING_LITERAL":
            # Jika panjangnya 3 (contoh: 'a'), itu adalah CHAR_LITERAL
            if len(lexeme) == 3 or len(lexeme) == 2:
                return "CHAR_LITERAL"
            else:
                return "STRING_LITERAL"

        return token_type if token_type else "UNKNOWN_TOKEN"

    def advance_past_comment(self, source_code, index):
        """
        Melakukan scanning untuk melewati seluruh blok komentar (misalnya: (*...*) atau
{...}).
        Mengembalikan index setelah akhir komentar dan penyesuaian baris/kolom.
        """
        return index, 0, 0

    def classify_char_input(self, char):
        if char.isalpha():
            return "letter"
        if char.isdigit():
            return "digit"
        return char

    def run_scanner(self, source_code):
        """
        Melakukan scanning kode sumber huruf demi huruf menggunakan logika DFA.
        """
        tokens = []


        while self.current_index < len(source_code):
            token_start_line = self.current_line
            token_start_coloumn = self.current_coloumn
```

```
            char = source_code[self.current_index]

            #Buat handle whitespace
            if char.isspace():
                if char == '\n':
                    self.current_line += 1
                    self.current_coloumn = 1
                else:
                    self.current_coloumn += 1
                self.current_index += 1
                continue

            # Handle comment
            if char == '{' or (char == '(' and self.current_index + 1 < len(source_code) and
source_code[self.current_index + 1] == '*'):
                # Skip comment content
                if char == '{':
                    self.current_index += 1
                    self.current_coloumn += 1
                    while self.current_index < len(source_code) and
source_code[self.current_index] != '}':
                        if source_code[self.current_index] == '\n':
                            self.current_line += 1
                            self.current_coloumn = 1
                        else:
                            self.current_coloumn += 1
                        self.current_index += 1
                    if self.current_index < len(source_code):
                        self.current_index += 1
                        self.current_coloumn += 1
                else:  # Handle (* ... *) comments
                    self.current_index += 2 #skip (*
                    self.current_coloumn += 2
                    while self.current_index + 1 < len(source_code):
                        if (self.current_index + 1 < len(source_code) and
                            source_code[self.current_index:self.current_index+2] == '*)'):
                            self.current_index += 2
                            self.current_coloumn += 2
                            break
                        if source_code[self.current_index] == '\n':
                            self.current_line += 1
                            self.current_coloumn = 1
                        else:
                            self.current_coloumn += 1
                        self.current_index += 1
                continue

            #Scanning Token
            current_state = self.dfa['start_state']
```

```
            lexeme =""
            longest_finalstate = None
            longest_lexeme = ""
            last_valid_index = self.current_index

            temp_index = self.current_index

            while temp_index < len(source_code):
                temp_char = source_code[temp_index]
                input_symbol = self.classify_char_input(temp_char)

                # Cek ada transisi ga
                if current_state in self.dfa['transitions'] and input_symbol in
self.dfa['transitions'][current_state]:
                    current_state = self.dfa['transitions'][current_state][input_symbol]
                    lexeme += temp_char
                    temp_index += 1

                    # Cek current state nya final atau ga
                    if current_state in self.dfa['final_states']:
                        longest_lexeme = lexeme
                        longest_finalstate = current_state
                        last_valid_index = temp_index
                else:
                    # No valid transition, stop
                    break

            #Buat tokennya
            if longest_lexeme:
                token_type = self.get_token_type(longest_lexeme, longest_finalstate)
                tokens.append(Token(token_type, longest_lexeme, token_start_line,
token_start_coloumn))

                #terus majuin poinnya ke posisi setelah token found
                self.current_coloumn += len(longest_lexeme)
                self.current_index = last_valid_index
            else:
                # Lexical error unknown symbol
                tokens.append(Token("LEXICAL_ERROR", char, token_start_line,
token_start_coloumn))
                print(f"Simbol unknown '{char}' pada baris {token_start_line}")
                self.current_index += 1
                self.current_coloumn += 1

        return tokens
```

Function	Class	Deskripsi

__init__(...)	Lexer	*Constructor* untuk *class*. Inisiasi state untuk lexer Ia akan memanggil load_dfa, menyimpan list keyword, dan mengatur variabel scanning awal.
load_dfa(file_path)		Untuk memuat dan memverifikasi file aturan DFA. Ia akan membaca file JSON, memeriksa kunci-kunci wajib (start_state, final_states, transitions), dan mengembalikan struktur data DFA. Juga menangani error seperti file tidak ditemukan.
get_token_type(...)		Menentukan tipe token yang sebenarnya dari kandidat Lexer. Jika mencapai state IDENTIFIER_CANDIDATE, ia akan memeriksa lexeme untuk menentukan apakah itu LOGICAL_OPERATOR, ARITHMETIC_OPERATOR (mod/div), KEYWORD, atau fallback ke IDENTIFIER. Juga membedakan CHAR_LITERAL dari STRING_LITERAL berdasarkan panjang lexeme.
classify_char_input(char)		Mengubah karakter aktual menjadi simbol input yang digunakan dalam tabel transisi DFA . Mengubah 'A', 'B',... menjadi "letter". Mengubah '1', '2',... menjadi "digit". Karakter lain dikembalikan apa adanya (misalnya '+', ';').
advance_past_comment(...)		*Comment handler*, melewati

		konten di dalam blok komentar yang valid ({...} atau (*...*)). Menemukan karakter penutup dan memperbarui current_index, current_line, dan current_coloumn tanpa menghasilkan token apa pun.
run_scanner(source_code)		Fungsi utama yang menjalankan analisis leksikal. 1. Main Loop: Iterasi melalui source_code. 2. Pembersihan: Menangani dan melewati whitespace serta komentar. 3. *Greedy Match*: Menggunakan DFA untuk mencari *longest valid* lexeme. 4. Tokenisasi: Memanggil get_token_type dan menambahkan objek Token ke list tokens. 5. *Error Handling*: Melaporkan dan melanjutkan setelah menemukan simbol yang tidak dikenal.

pascal_token.py

```
class Token:
    """
    Merepresentasikan satu unit makna tunggal (Token).
    """
    def __init__(self, type, value, line=None, column=None):
        # inisiasi objek token
        self.type = type
        self.value = value
        self.line = line
        self.column = column

    def __str__(self):
        """
        print representasi string Token sesuai format output yang diminta
        """
```

```
        return f"{self.type}({self.value})"
```

Function	Class	Deskripsi
__init__(...)	Token	Konstruktor, Menyimpan detail token (tipe, nilai, posisi).
__str__()		Format Output, Mendefinisikan bagaimana objek dicetak. Mengembalikan representasi string token dalam format TYPE(VALUE) (misalnya, KEYWORD(program)), yang digunakan untuk output akhir.

dfa_rules.json

```
{
    "start_state": "S0",
    "final_states": {
        "S_ID": "IDENTIFIER_CANDIDATE",
        "S_SEMI": "SEMICOLON",
        "S_NUM": "NUMBER",
        "S_REAL": "NUMBER",
        "S_PLUS": "ARITHMETIC_OPERATOR",
        "S_COMMA": "COMMA",
        "S_COLON": "COLON",
        "S_ASSIGN": "ASSIGN_OPERATOR",
        "S_LPAREN": "LPARENTHESIS",
        "S_RPAREN": "RPARENTHESIS",
    "S_DOT": "DOT",
        "S_STRING_END": "STRING_LITERAL",
        "S_GT": "RELATIONAL_OPERATOR",
        "S_GTE": "RELATIONAL_OPERATOR",
        "S_LT": "RELATIONAL_OPERATOR",
        "S_LTE": "RELATIONAL_OPERATOR",
        "S_NEQ": "RELATIONAL_OPERATOR",
        "S_EQ": "RELATIONAL_OPERATOR",
        "S_MINUS": "ARITHMETIC_OPERATOR",
        "S_MULT": "ARITHMETIC_OPERATOR",
        "S_DIV": "ARITHMETIC_OPERATOR",
        "S_LBRACKET": "LBRACKET",
```

```
        "S_RBRACKET": "RBRACKET"
    },
    "transitions": {
        "S0": {
            "letter": "S_ID",
            ";": "S_SEMI",
            "digit": "S_NUM",
            "+": "S_PLUS",
            ",": "S_COMMA",
            ":": "S_COLON",
            "(": "S_LPAREN",
            ")": "S_RPAREN",
            ".": "S_DOT",
            "'": "S_STRING",
            ">": "S_GT",
            "<": "S_LT",
            "=": "S_EQ",
            "-": "S_MINUS",
            "*": "S_MULT",
            "/": "S_DIV",
            "[": "S_LBRACKET",
            "]": "S_RBRACKET"
        },
        "S_ID": {
            "letter": "S_ID",
            "digit": "S_ID",
            "_": "S_ID"
        },
        "S_SEMI": {},
        "S_NUM": {
            "digit": "S_NUM",
            ".": "S_REAL_DOT"
        },
        "S_REAL_DOT": {
            "digit": "S_REAL"
        },
        "S_REAL": {
            "digit": "S_REAL"
        },
        "S_PLUS": {},
        "S_COMMA": {},
        "S_COLON": {
            "=": "S_ASSIGN"
        },
        "S_ASSIGN": {},
        "S_LPAREN": {},
        "S_RPAREN": {},
        "S_DOT": {},
        "S_STRING": {
```

```
            "letter": "S_STRING",
            "digit": "S_STRING",
            " ": "S_STRING",
            "=": "S_STRING",
            "+": "S_STRING",
            "-": "S_STRING",
            "*": "S_STRING",
            "/": "S_STRING",
            "(": "S_STRING",
            ")": "S_STRING",
            ",": "S_STRING",
            ".": "S_STRING",
            ":": "S_STRING",
            ";": "S_STRING",
            ">": "S_STRING",
            "<": "S_STRING",
            "[": "S_STRING",
            "]": "S_STRING",
            "'": "S_STRING_END"
        },
        "S_STRING_END": {},
        "S_GT": {
            "=": "S_GTE"
        },
        "S_GTE": {},
        "S_LT": {
            "=": "S_LTE",
            ">": "S_NEQ"
        },
        "S_LTE": {},
        "S_NEQ": {},
        "S_EQ": {},
        "S_MINUS": {},
        "S_MULT": {},
        "S_DIV": {},
        "S_LBRACKET": {},
        "S_RBRACKET": {}
    }
}
```

1. start_state
Nilai: "S0” Mendefinisikan state awal. Setiap kali Lexer selesai mengidentifikasi satu token atau mengabaikan whitespace/komentar, ia akan kembali ke S0 untuk memulai pengenalan token berikutnya.

2. final_states
Mendefinisikan semua state yang, jika dicapai, Lexer dapat mengklaim telah berhasil menemukan lexeme yang valid dan mengeluarkan Token. Beberapa state dipetakan ke tipe Token yang sama (misalnya, S_NUM dan S_REAL keduanya dipetakan ke NUMBER) karena keduanya merupakan Literal numerik.

3. transitions
Tabel logika utama DFA. Setiap kunci di tingkat pertama adalah State Awal, dan isinya adalah aturan yang menentukan perpindahan state berdasarkan Input Simbol (letter, digit, atau karakter spesifik).

State Awal	Input Dilihat	State Selanjutnya	Tipe Token yang Dibentuk	Konsep Utama
S0	letter	S_ID	IDENTIFIER_CANDIDATE	Semua kata (identifier/key word) mulai dari sini.
S0	digit	S_NUM	NUMBER	Semua bilangan dimulai dari sini.
S_NUM	.	S_REAL_DOT	Intermediate	Pengenalan Bilangan Real: Jika setelah angka, Lexer melihat . (dot), Lexer beralih dari integer sederhana ke

				real (angka desimal).
S_REAL_DOT	digit	S_REAL	NUMBER	Setelah dot, harus ada digit agar valid. S_REAL adalah final state yang menunjukkan bilangan real.
S0	:	S_COLON	COLON	Pola Majemuk (:=): Lexer berada pada state final (S_COLON).
S_COLON	=	S_ASSIGN	ASSIGN_OPERATOR	Greedy Match: Jika setelah : dilihat =, Lexer pindah ke state yang lebih panjang (S_ASSIGN), memprioritaskan := daripada :.
S0	<	S_LT	RELATIONAL_OPERATOR	Pola Majemuk (<=, <>, <): Lexer mulai dari <.

S_LT	=	S_LTE	RELATIONAL_OPERATOR	Jika diikuti =, membentuk <=.
S_LT	>	S_NEQ	RELATIONAL_OPERATOR	Jika diikuti >, membentuk <>.
S0	'	S_STRING	Intermediate	Pengenalan String/Char Literal: Lexer mulai dari tanda kutip pembuka.
S_STRING	any_char	S_STRING	Intermediate	Tetap dalam state ini hingga menemukan '.
S_STRING	'	S_STRING_END	STRING_LITERAL	Menemukan tanda kutip penutup, Token selesai.

BAB III
Pengujian

3.1 Testing 1

Input (Kode Paskal)

```
program Hello;

var
  a, b: integer;

begin
  a := 5;
  b := a + 10;
  writeln('Result = ', b);
end.
```

Output (File.txt)

```
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(.)
```

3.2 Testing 2

Input (Kode Paskal)

```
program BasicTest;

var
   x, y, total: integer;

begin
   x := 5;
   y := x + 10 mod 3;

   if y > x then
      total := y
   else
      total := x;

   writeln('Total:', total);
end.
```

Output (File.txt)

```
KEYWORD(program)
IDENTIFIER(BasicTest)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(x)
COMMA(,)
IDENTIFIER(y)
COMMA(,)
IDENTIFIER(total)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(x)
```

```
ASSIGN_OPERATOR(:=)
NUMBER(5)
SEMICOLON(;)
IDENTIFIER(y)
ASSIGN_OPERATOR(:=)
IDENTIFIER(x)
ARITHMETIC_OPERATOR(+)
NUMBER(10)
ARITHMETIC_OPERATOR(mod)
NUMBER(3)
SEMICOLON(;)
KEYWORD(if)
IDENTIFIER(y)
RELATIONAL_OPERATOR(>)
IDENTIFIER(x)
KEYWORD(then)
IDENTIFIER(total)
ASSIGN_OPERATOR(:=)
IDENTIFIER(y)
KEYWORD(else)
IDENTIFIER(total)
ASSIGN_OPERATOR(:=)
IDENTIFIER(x)
SEMICOLON(;)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Total:')
COMMA(,)
IDENTIFIER(total)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(end)
DOT(.)
```

3.3 Testing 3

Input (Kode Paskal)

```
program ArrayLogic;

const
   SIZE = 1..10;

var
```

```
    A: array [SIZE] of char;
    flag: boolean;

begin
    flag := TRUE and NOT FALSE; (* Logika dan assignment *)

    if (A[1] <> 'Z') or flag then
    begin
        A[1] := 'a'; (* Char Literal *)
    end;

    writeln('Done'); (* String Literal *)
end.
```

Output (File.txt)

```
KEYWORD(program)
IDENTIFIER(ArrayLogic)
SEMICOLON(;)
KEYWORD(const)
IDENTIFIER(SIZE)
RELATIONAL_OPERATOR(=)
NUMBER(1)
DOT(.)
DOT(.)
NUMBER(10)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(A)
COLON(:)
KEYWORD(array)
LBRACKET([)
IDENTIFIER(SIZE)
RBRACKET(])
KEYWORD(of)
KEYWORD(char)
SEMICOLON(;)
IDENTIFIER(flag)
COLON(:)
KEYWORD(boolean)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(flag)
ASSIGN_OPERATOR(:=)
IDENTIFIER(TRUE)
```

```
LOGICAL_OPERATOR(and)
LOGICAL_OPERATOR(NOT)
IDENTIFIER(FALSE)
SEMICOLON(;)
KEYWORD(if)
LPARENTHESIS(()
IDENTIFIER(A)
LBRACKET([)
NUMBER(1)
RBRACKET(])
RELATIONAL_OPERATOR(<>)
CHAR_LITERAL('Z')
RPARENTHESIS())
LOGICAL_OPERATOR(or)
IDENTIFIER(flag)
KEYWORD(then)
KEYWORD(begin)
IDENTIFIER(A)
LBRACKET([)
NUMBER(1)
RBRACKET(])
ASSIGN_OPERATOR(:=)
CHAR_LITERAL('a')
SEMICOLON(;)
KEYWORD(end)
SEMICOLON(;)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Done')
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(end)
DOT(.)
```

3.4 Testing 4

Input (Kode Paskal)

```
program ErrorTest;

var
   z: real;

begin
   { Ini adalah komentar gaya curly brace
```

```
    yang bisa multi-baris }

   z := 5.25; (* Bilangan Real *)

   if z >= 0 then
      z := z / 2;

   writeln(z); $

   (* Ini juga komentar, menggunakan kurung dan bintang *)
end.
```

Output (File.txt)

```
KEYWORD(program)
IDENTIFIER(ErrorTest)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(z)
COLON(:)
KEYWORD(real)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(z)
ASSIGN_OPERATOR(:=)
NUMBER(5.25)
SEMICOLON(;)
KEYWORD(if)
IDENTIFIER(z)
RELATIONAL_OPERATOR(>=)
NUMBER(0)
KEYWORD(then)
IDENTIFIER(z)
ASSIGN_OPERATOR(:=)
IDENTIFIER(z)
ARITHMETIC_OPERATOR(/)
NUMBER(2)
SEMICOLON(;)
IDENTIFIER(writeln)
LPARENTHESIS(()
IDENTIFIER(z)
RPARENTHESIS())
SEMICOLON(;)
LEXICAL_ERROR($)
KEYWORD(end)
```

DOT(.)

3.5 Testing 5

Input (Kode Paskal)

```
program TestAllTokens; {1, 2, 10}

const
   PiValue = 3.14; {1, 2, 7, 10}

type
   MyArrayType = array [1..10] of integer; {1, 2, 18, 16, 17, 1}

var
   sum, count: integer; {2, 11, 12, 1}
   flag: boolean; {2, 12, 1}
   ch: char; {2, 12, 1}

begin
   sum := 0; {1, 2, 6, 7, 10}
   count := 5; {2, 6, 7, 10}

   ch := 'A'; {2, 6, 8, 10}

   if (sum <> 0) and (count > 0) then {1, 14, 2, 4, 7, 15, 5, 14, 2, 4, 7, 15, 1}
      flag := TRUE
   else
      flag := FALSE;

   while flag do {1}
   begin
      count := count div 2; {2, 6, 2, 3, 7, 10}
   end;

   for count := 1 to 10 do {1, 2, 6, 7, 1, 7, 1}
      sum := sum + count; {2, 6, 2, 3, 2, 10}

   writeln('Final Sum: ', sum); {2, 14, 9, 11, 2, 15, 10}

   end. {1, 13}
```

Output (File.txt)

```
KEYWORD(program)
IDENTIFIER(TestAllTokens)
SEMICOLON(;)
KEYWORD(const)
IDENTIFIER(PiValue)
RELATIONAL_OPERATOR(=)
NUMBER(3.14)
SEMICOLON(;)
KEYWORD(type)
IDENTIFIER(MyArrayType)
RELATIONAL_OPERATOR(=)
KEYWORD(array)
LBRACKET([)
NUMBER(1)
DOT(.)
DOT(.)
NUMBER(10)
RBRACKET(])
KEYWORD(of)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(sum)
COMMA(,)
IDENTIFIER(count)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(flag)
COLON(:)
KEYWORD(boolean)
SEMICOLON(;)
IDENTIFIER(ch)
COLON(:)
KEYWORD(char)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(sum)
ASSIGN_OPERATOR(:=)
NUMBER(0)
SEMICOLON(;)
IDENTIFIER(count)
ASSIGN_OPERATOR(:=)
NUMBER(5)
SEMICOLON(;)
IDENTIFIER(ch)
```

```
ASSIGN_OPERATOR(:=)
CHAR_LITERAL('A')
SEMICOLON(;)
KEYWORD(if)
LPARENTHESIS(()
IDENTIFIER(sum)
RELATIONAL_OPERATOR(<>)
NUMBER(0)
RPARENTHESIS())
LOGICAL_OPERATOR(and)
LPARENTHESIS(()
IDENTIFIER(count)
RELATIONAL_OPERATOR(>)
NUMBER(0)
RPARENTHESIS())
KEYWORD(then)
IDENTIFIER(flag)
ASSIGN_OPERATOR(:=)
IDENTIFIER(TRUE)
KEYWORD(else)
IDENTIFIER(flag)
ASSIGN_OPERATOR(:=)
IDENTIFIER(FALSE)
SEMICOLON(;)
KEYWORD(while)
IDENTIFIER(flag)
KEYWORD(do)
KEYWORD(begin)
IDENTIFIER(count)
ASSIGN_OPERATOR(:=)
IDENTIFIER(count)
ARITHMETIC_OPERATOR(div)
NUMBER(2)
SEMICOLON(;)
KEYWORD(end)
SEMICOLON(;)
KEYWORD(for)
IDENTIFIER(count)
ASSIGN_OPERATOR(:=)
NUMBER(1)
KEYWORD(to)
NUMBER(10)
KEYWORD(do)
IDENTIFIER(sum)
ASSIGN_OPERATOR(:=)
IDENTIFIER(sum)
```

```
ARITHMETIC_OPERATOR(+)
IDENTIFIER(count)
SEMICOLON(;)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Final Sum: ')
COMMA(,)
IDENTIFIER(sum)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(end)
DOT(.)
```

3.6 Testing 6

Input (Kode Paskal)

```
program NegativeNumberTest;

const
  MAX_SIZE = 100;

var
  x, y: integer;
  isActive: boolean;
  testString: char;
  emptyString: char;

begin

  x := -10;

  y := x + (-5);

  isActive := true;

  testString := 'true';

  emptyString := ''; // Baris baru ditambahkan di sini

  if y < -12 then
  begin
    writeln('y is very small');
  end;
```

```
end.
```

Output (File.txt)

```
KEYWORD(program)
IDENTIFIER(NegativeNumberTest)
SEMICOLON(;)
KEYWORD(const)
IDENTIFIER(MAX_SIZE)
RELATIONAL_OPERATOR(=)
NUMBER(100)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(x)
COMMA(,)
IDENTIFIER(y)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(isActive)
COLON(:)
KEYWORD(boolean)
SEMICOLON(;)
IDENTIFIER(testString)
COLON(:)
KEYWORD(char)
SEMICOLON(;)
IDENTIFIER(emptyString)
COLON(:)
KEYWORD(char)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(x)
ASSIGN_OPERATOR(:=)
ARITHMETIC_OPERATOR(-)
NUMBER(10)
SEMICOLON(;)
IDENTIFIER(y)
ASSIGN_OPERATOR(:=)
IDENTIFIER(x)
ARITHMETIC_OPERATOR(+)
LPARENTHESIS(()
ARITHMETIC_OPERATOR(-)
NUMBER(5)
RPARENTHESIS())
```

```
SEMICOLON(;)
IDENTIFIER(isActive)
ASSIGN_OPERATOR(:=)
IDENTIFIER(true)
SEMICOLON(;)
IDENTIFIER(testString)
ASSIGN_OPERATOR(:=)
STRING_LITERAL('true')
SEMICOLON(;)
IDENTIFIER(emptyString)
ASSIGN_OPERATOR(:=)
CHAR_LITERAL('')
SEMICOLON(;)
ARITHMETIC_OPERATOR(/)
ARITHMETIC_OPERATOR(/)
IDENTIFIER(Baris)
IDENTIFIER(baru)
IDENTIFIER(ditambahkan)
IDENTIFIER(di)
IDENTIFIER(sini)
KEYWORD(if)
IDENTIFIER(y)
RELATIONAL_OPERATOR(<)
ARITHMETIC_OPERATOR(-)
NUMBER(12)
KEYWORD(then)
KEYWORD(begin)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('y is very small')
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(end)
SEMICOLON(;)
KEYWORD(end)
DOT(.)
```

BAB IV
Kesimpulan dan Saran

4.1 Kesimpulan

Tujuan utama Lexical Analysis untuk mengubah source code dari rangkaian karakter mentah menjadi unit-unit bermakna yang disebut Token telah tercapai. Setiap Token yang dihasilkan (terdapat 20 tipe Token) memiliki jenis (type) dan nilai (value) yang akurat. Lexer diimplementasikan berdasarkan model teoretis *Deterministic Finite Automata* (DFA). Logika DFA dimodelkan secara terstruktur dalam file konfigurasi dfa_rules.json dan diproses oleh class Lexer , yang memberikan kerangka kerja deterministik dan non-ambigu untuk *scanning* kode.

Poin-poin keberhasilan utama dari implementasi Lexer ini adalah:

1. DFA Berbasis Konfigurasi: Lexer berhasil diimplementasikan berdasarkan model Deterministic Finite Automata (DFA) , dengan semua aturan transisi dimuat secara dinamis dari file dfa_rules.json, yang menunjukkan modularitas yang baik.
2. Kelengkapan Token: DFA yang dirancang berhasil mengenali dan mengklasifikasikan 20 tipe token Pascal-S , termasuk penanganan Literal Numerik (integer dan real) dan Literal String/Char.
3. Greedy Match dan Ambiguitas: Logika Greedy Match dan method klasifikasi Token berhasil mengatasi ambiguitas token multikarakter (misalnya, membedakan : dari := dan . dari ..). Selain itu, keyword operator (seperti mod, div, and, or, not) berhasil dibedakan dari Identifiers.
4. Pembersihan Kode: Lexer secara efektif membersihkan source code dengan melewati whitespace dan semua jenis komentar yang valid ({...} dan (*...*)).

Secara keseluruhan, output yang dihasilkan berupa list Token sudah benar

4.2 Saran

Untuk meningkatkan kualitas proyek dan mempersiapkan Milestone berikutnya, berikut adalah beberapa saran pengembangan:

1. Integrasi dengan Syntax Analysis: Disarankan untuk segera memulai perancangan Parser (Analisis Sintaksis), yang akan menggunakan list Token yang dihasilkan oleh Lexer ini sebagai input untuk memverifikasi struktur program. Hal ini akan menguji kapabilitas Lexer secara penuh.
2. Peningkatan Error Handling: Meskipun Lexer sudah menangani error simbol yang tidak dikenal, sistem dapat ditingkatkan untuk menangani error leksikal lainnya, seperti string

atau komentar yang tidak ditutup hingga End-of-File (EOF), dan memberikan pesan error yang lebih informatif (misalnya, menyertakan lexeme yang menyebabkan error).

3. Optimalisasi DFA: Untuk proyek yang lebih besar, DFA dapat dioptimalisasi lebih lanjut untuk mengurangi jumlah state secara signifikan , yang dapat meningkatkan efisiensi penggunaan memori, meskipun DFA saat ini sudah terbukti efisien dalam waktu linear

BAB V
Lampiran

5.1 Link Repository Github

https://github.com/iannn23/HJE-Tubes-IF2224.git

5.2 Link Workspace Diagram DFA

https://drive.google.com/file/d/1rB25CckCak9_7L3TSquK9DLXx0BWqna2/view?usp=sharing

5.3 Pembagian Tugas

Nama	NIM	Tugas
Sebastian Enrico Nathanael	13523134	Implementasi code, DFA, testing, Laporan
Jonathan Kenan Budianto	13523139	Implementasi code, DFA, testing
Mahesa Fadhillah Andre	13523140	Implementasi DFA, testing, Laporan
Muhammad Farrel Wibowo	13523153	Implementasi DFA, Pembuatan Diagram, testing

BAB VI
Referensi

- Aho, Alfred V., Sethi, Ravi, dan Ullman, Jeffrey D. Compilers. Principles, Techniques, and Tools (2006). Prentice Hall.
- Hopcroft, John E., Motwani, Rajeev, dan Ullman, Jeffrey D. Introduction to Automata Theory, Languages, and Computation,. Edisi ke-3.
- Jensen, Kathleen, dan Wirth, Niklaus. PASCAL-S: A Subset and its implementation. Tutorialspoint. Compiler Design - Lexical Analysis.
- Institut Teknologi Bandung. Spesifikasi Tugas Besar IF2224 - Formal Language and Automata Theory, Milestone 1. https://docs.google.com/document/d/1w0GmHW5L0gKZQWbgmtJPFmOzlpSWBknNPdugucn4eII/edit?tab=t.0
- GeeksforGeeks. "Introduction of Lexical Analysis.". https://www.geeksforgeeks.org/compiler-design/introduction-of-lexical-analysis/
- TutorialsPoint. "Compiler Design - Lexical Analysis.". https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm