

# LAPORAN MILESTONE 1

## TUGAS BESAR TEORI BAHASA FORMAL DAN AUTOMATA

IF2224 - Teori Bahasa Formal dan Automata

Kelompok ZZZ - JadiApaArtiHidup?



### Anggota Kelompok:

Ahmad Syafiq 13523135

Frederiko Eldad Mugiyono 13523147

Naufarrel Zhafif Abhista 13523159

Hasri Fayadh Muqaffa 13523156

I Made Wiweka Putera 13523160

**PROGRAM STUDI TEKNIK INFORMATIKA**

**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**

**INSTITUT TEKNOLOGI BANDUNG**

**2025**

## Daftar Isi

<b>Daftar Isi.....</b>	<b>2</b>
<b>Landasan Teori.....</b>	<b>3</b>
1. Compiler.....	3
2. Analisis Leksikal.....	3
3. Lexeme, Pattern, dan Token.....	4
4. Finite Automata.....	4
<b>Perancangan dan Implementasi.....</b>	<b>5</b>
1. Diagram.....	5
2. Struktur Proyek.....	5
3. Alur Kerja Lexical Analyzer.....	7
4. Penjelasan Modul Implementasi.....	8
<b>Pengujian.....</b>	<b>10</b>
1. Pengujian.....	10
Tabel 1.1. Hasil Pengujian TC1.....	10
Tabel 1.2. Hasil Pengujian TC2.....	12
Tabel 1.3. Hasil Pengujian TC3.....	13
Tabel 1.4. Hasil Pengujian TC4.....	14
Tabel 1.5. Hasil Pengujian TC5.....	16
2. Analisis Hasil Pengujian.....	16
<b>Kesimpulan dan Saran.....</b>	<b>18</b>
1. Kesimpulan.....	18
2. Saran.....	18
<b>Lampiran.....</b>	<b>20</b>
<b>Referensi.....</b>	<b>21</b>

## Landasan Teori

### 1. Compiler

Compiler adalah sebuah program komputer yang berfungsi untuk menerjemahkan kode sumber (*source code*) yang ditulis dalam suatu bahasa pemrograman (bahasa sumber) ke dalam bahasa pemrograman lain (bahasa target). Biasanya, bahasa target adalah *machine code* atau *assembly language* yang dapat dieksekusi langsung oleh prosesor komputer. Proses penerjemahan ini tidak terjadi dalam satu langkah, tetapi melalui serangkaian tahapan. Secara spesifik, tahapan dalam sebuah compiler adalah sebagai berikut:

- a. Analisis Leksikal (*Lexical Analysis*), yaitu fase pertama yang membaca aliran karakter dari kode sumber dan mengelompokkannya ke dalam unit bermakna yang disebut token.
- b. Analisis sintaks (*Syntax Analysis* atau *Parsing*), yaitu menggunakan token dari fase sebelumnya untuk membangun struktur hierarkis program, biasanya dalam bentuk *parse tree*.
- c. Analisis Semantik (*Semantic Analysis*), yaitu memeriksa konsistensi program secara semantik.
- d. Pembuatan Kode Perantara (*Intermediate Code Generation*), yaitu menghasilkan representasi program yang sederhana dan mirip dengan bahasa mesin.
- e. *Interpreter*

### 2. Analisis Leksikal

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

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

Proses ini dilakukan oleh komponen yang disebut lexer. Lexer membaca kode sumber dari kiri ke kanan dan menggunakan pola tertentu untuk mengenali jenis

token yang berbeda-beda. Ketika lexer menemukan urutan karakter yang cocok dengan salah satu pola tersebut, ia membuat sebuah token yang biasanya berisi dua hal utama: jenis token dan nilai token.

### **3. Lexeme, Pattern, dan Token**

Terdapat tiga konsep penting yang harus diperhatikan dari cara kerja analisis leksikal, yaitu:

- Lexeme, yaitu urutan karakter di dalam source code yang cocok dengan sebuah pattern.
- Pattern, yaitu aturan atau deskripsi yang mendefinisikan bentuk dari lexeme yang dapat membentuk sebuah token.
- Token, yaitu unit leksikal abstrak yang menjadi keluaran dari lexer dan masukan untuk parser.

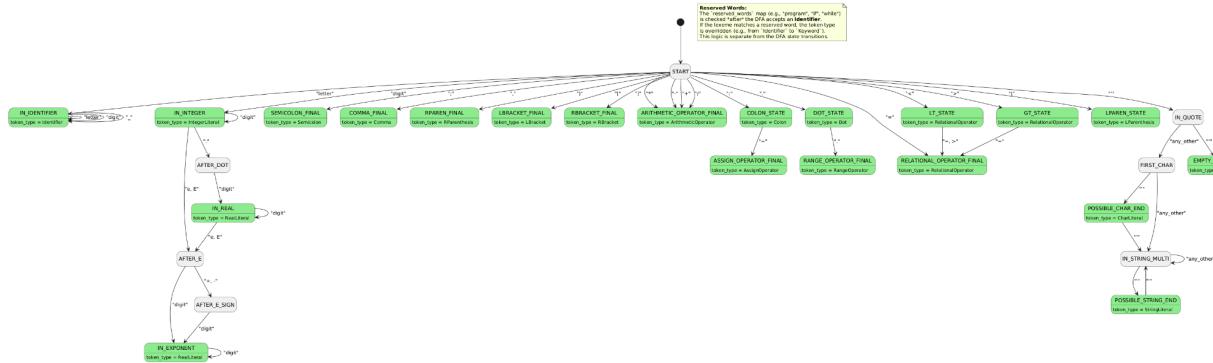
### **4. Finite Automata**

Finite Automata (FA) adalah model matematika yang memiliki sejumlah state (keadaan) dan transisi antar state yang dipicu oleh karakter masukan. Terdapat dua jenis Finite Automata, yaitu:

- a. Nondeterministic Finite Automata (NFA), yaitu FA yang dapat memiliki nol, satu, atau lebih transisi dari sebuah state untuk satu simbol masukan yang sama. NFA juga dapat memiliki transisi  $\epsilon$  (epsilon), yaitu transisi yang terjadi tanpa mengonsumsi karakter masukan.
- b. Deterministic Finite Automata (DFA), merupakan kasus khusus dari NFA yang membuat setiap state dan setiap simbol masukan, hanya ada tepat satu transisi ke state berikutnya. DFA tidak memiliki transisi  $\epsilon$  (epsilon)

# Perancangan dan Implementasi

## 1. Diagram



[https://github.com/reletz/ZZZ-Tubes-IF2224/blob/main/doc/dfa\\_diagram.png](https://github.com/reletz/ZZZ-Tubes-IF2224/blob/main/doc/dfa_diagram.png)

## 2. Struktur Proyek

Proyek ini dibuat menggunakan Rust dan struktur folder diatur mengikuti konvensi standar dari Cargo. Adapun struktur dari proyek ini adalah sebagai berikut.



```
|   └── milestone-4/  
|       └── milestone-5/  
└── Cargo.lock  
└── Cargo.toml  
└── README.md
```

Keterangan:

- config/: Berisi file dfa.json. File tersebut menyimpan aturan DFA yang akan dibaca oleh program rust untuk mengetahui cara mengenali token.
- doc/: Berisi dokumen spesifikasi dan laporan dari tiap milestone.
- examples/: Berisi contoh source code Pascal-S (.pas) yang digunakan sebagai masukan untuk compiler dan memastikan *lexical analyzer* berfungsi dengan benar.
- src\: Berisi source code utama dari tugas besar ini. Terdapat beberapa folder di dalamnya. Untuk milestone ini, fokus utamanya berada pada folder lexer. Selain itu, terdapat file main.rs yang menjadi *entry point* dari program.
- target/: Berisi hasil kompilasi dari program yang dibuat oleh Cargo.
- test/: Berisi file *input* dan *output* dari file uji.
- Cargo.toml: Berisi manifest dari proyek ini yang mendefinisikan proyek, versi dan dependensi atau *library* eksternal yang digunakan.
- README.md: Berisi dokumentasi dari proyek.

### 3. Alur Kerja Lexical Analyzer

Alur kerja dari program lexer ini dimulai dari eksekusi file main.rs sebagai *entry point* dan berakhir pada *output* yang berupa daftar token. Berikut rincian alur kerja dari lexer:

#### 1) Inisialisasi

Proses dimulai dari fungsi main() di dalam main.rs yang membaca *path* ke file source code Pascal-S (contohnya, examples/hello.pas). Isi dari source code tersebut akan dibaca dalam sebuah string di memori. Lalu, dibuat sebuah instansiasi dari struct PascalLexel. Ketika membuat instansiasi tersebut, lexer memanggil dfa::load\_dfa\_config("config/dfa.json"). Fungsi ini akan membaca file dfa.json, melakukan *parsing* isi dari file JSON tersebut.

#### 2) Proses Tokenisasi

Langkah selanjutnya adalah pengambilan token melalui lexer.get\_all\_tokens(). Fungsi ini berfungsi untuk membaca seluruh input dan mengubahnya menjadi sebuah daftar yang berisi semua token. Di dalam fungsi tersebut, terdapat fungsi next\_token() yang akan melakukan hal berikut:

- Mengabaikan whitespace dengan memanggil self.skip\_whitespace().

- Melakukan pengecekan EoF.
- Menyimpan posisi dengan mencatat start\_line dan start\_col saat ini.
- Memanggil simulasi DFA dengan memanggil self.dfa(...).

### 3) Simulasi DFA

Fungsi self.dfa.dfa(...) ini berisi implementasi algoritma pengenalan token berdasarkan prinsip Longest Match, yaitu lexer akan selalu berusaha mencocokkan lexeme terpanjang yang mungkin. Adapun rinciannya sebagai berikut:

- Simulasi dimulai dari self.dfa.dfa(...).
- Setelah itu, fungsi akan melakukan *looping*, untuk mengintip karakter berikutnya tanpa memajukan iterator utama.
- Untuk setiap karakter, akan dicari transisi yang valid dari state saat ini. Transisi ini dicocokkan menggunakan transition\_matcher.
- Jika transisi ditemukan dan state tujuan baru (next\_state) ditandai sebagai is\_final: true di dfa.json, lexer akan menyimpan nama state ini (last\_final\_state) dan lexeme yang telah terbentuk hingga saat itu (lexeme\_at\_last\_final).
- Simulasi tidak berhenti di final state pertama, tetapi akan terus berlanjut selama masih ada transisi yang valid.
- Ketika *looping* berhenti, terdapat dua kemungkinan, yaitu:
  - Jika last\_final\_state berisi sebuah nama state, fungsi ini akan mengembalikan last\_final\_state dan lexeme\_at\_last\_final yang terakhir disimpan.
  - Jika last\_final\_state berhenti di non final state (DFA mengalami macet pada karakter tertentu dan tidak pernah sampai di final state), lexeme tidak akan dikonsumsi sebagai token yang valid.

### 4) Resolusi Token

Setelah memanggil self.dfa.dfa(...), fungsi next\_token() akan menerima output dari simulasi DFA tersebut berupa (final\_state\_name, lexeme). Kemudian, next\_token() akan memanggil self.resolve\_token\_type() yang akan mengambil token\_type dari final\_state\_name dan kemudian akan melakukan pengecekan sekunder, yaitu jika tipe dasarnya adalah Identifier, akan dicari lexeme tersebut di dalam self.dfa.reserved\_words. Jika ditemukan, token type akan diubah menjadi Keyword. Setelah token dikenali, iterator karakter utama akan maju sepanjang lexeme yang dikenali. Jika token yang dikenali ada komentar, maka akan di-skip. Terakhir akan dilakukan return yang berisi Struct Token.

## 4. Penjelasan Modul Implementasi

Implementasi kode dibagi menjadi beberapa file dalam direktori src/. Adapun rinciannya sebagai berikut.

### 1) lexer/token\_types.rs

File ini mendefinisikan struktur data yang digunakan di seluruh proses lexer.

Rinciannya sebagai berikut:

- enum TokenType: Mendefinisikan semua kemungkinan jenis token yang dapat dikenali.
- struct Token: Merepresentasikan sebuah token yang sudah selesai diproses.
- fmt::Display for Token: Mencetak token.

### 2) lexer/dfa.rs

File ini berisi semua implementasi yang berkaitan langsung dengan DFA.

Adapun rinciannya sebagai berikut.

- DfaConfig, State, Transition: Merepresentasikan DFA di dalam memori dan strukturnya sudah disesuaikan dengan format dari dfa.json.
- load\_dfa\_config(): Membaca file dfa.json, melakukan parsing, dan mengembalikan struct DfaConfig.
- dfa(): Menjalankan simulasi DFA dengan mengintip karakter-karakter berikutnya untuk menemukan lexeme terpanjang yang cocok (berdasarkan prinsip Longest Match) dan mengembalikan lexeme yang ditemukan dari final state setelah *looping* berhenti.

### 3) lexer/lexer.rs

File ini berisi implementasi utama untuk melakukan *lexical analysis*. Adapun rinciannya sebagai berikut.

- struct PascalLexer: Menyimpan state dari lexer selama proses analisis.
- get\_all\_tokens(): Mengambil semua token dari awal hingga akhir dengan berulang kali memanggil next\_token().
- next\_token(): Mengambil satu token berikutnya dengan menjalankan simulasi DFA untuk mendapatkan lexeme terpanjang.

# Pengujian

## 1. Pengujian

Pada bagian ini, dilakukan serangkaian pengujian terhadap lexer untuk memvalidasi kemampuannya dalam mengenali berbagai jenis token pada bahasa Pascal. Pengujian dilakukan dengan memberikan lima berkas masukan .pas yang berbeda, masing-masing dirancang untuk menguji aspek spesifik dari analisis leksikal. Setiap pengujian disajikan dalam tabel di bawah, lengkap dengan *input*, *output*, dan analisisnya.

Test Case 1	
Struktur Program Dasar test1_simple.pas	
Input	Output
	<code>KEYWORD(program) IDENTIFIER(Simple) SEMICOLON(;) KEYWORD(begin) KEYWORD(end) DOT(.) EOF()</code>
Analisis	
Lexer berhasil mengenali token-token fundamental (KEYWORD, IDENTIFIER, SEMICOLON, DOT) pada struktur program paling minimal. Pengujian berhasil.	

Tabel 1.1. Hasil Pengujian TC1

Test Case 2	
Operator Aritmetika & Relasional test2_operators.pas	
Input	Output

```
● ● ●  
1 program Operators;  
2 var x: integer;  
3 begin  
4   x := 5 + 3 - 2 * 4 div 2 mod 3;  
5   if x = 6 then  
6     if x < 7 then  
7       if x < 10 then  
8         if x > 0 then  
9           if x ≥ 0 then  
10          writeln('All operators work');  
11      end.  
12    end.
```

```
KEYWORD(program)  
IDENTIFIER(Operators)  
SEMICOLON(;)  
KEYWORD(var)  
IDENTIFIER(x)  
COLON(:)  
KEYWORD(integer)  
SEMICOLON(;)  
KEYWORD(begin)  
IDENTIFIER(x)  
ASSIGN_OPERATOR(:=)  
INT_LITERAL(5)  
ARITHMETIC_OPERATOR(+)  
INT_LITERAL(3)  
ARITHMETIC_OPERATOR(-)  
INT_LITERAL(2)  
ARITHMETIC_OPERATOR(*)  
INT_LITERAL(4)  
ARITHMETIC_OPERATOR(div)  
INT_LITERAL(2)  
ARITHMETIC_OPERATOR(mod)  
INT_LITERAL(3)  
SEMICOLON(;)  
KEYWORD(if)  
IDENTIFIER(x)  
RELATIONAL_OPERATOR(=)  
INT_LITERAL(6)  
KEYWORD(then)  
KEYWORD(if)  
IDENTIFIER(x)  
RELATIONAL_OPERATOR(<>)  
INT_LITERAL(7)  
KEYWORD(then)  
KEYWORD(if)  
IDENTIFIER(x)  
RELATIONAL_OPERATOR(<)  
INT_LITERAL(10)  
KEYWORD(then)  
KEYWORD(if)  
IDENTIFIER(x)  
RELATIONAL_OPERATOR(<=)  
INT_LITERAL(10)
```

	<pre> KEYWORD(then) KEYWORD(if) IDENTIFIER(x) RELATIONAL_OPERATOR(&gt;) INT_LITERAL(0) KEYWORD(then) KEYWORD(if) IDENTIFIER(x) RELATIONAL_OPERATOR(&gt;=) INT_LITERAL(0) KEYWORD(then) IDENTIFIER(writeln) LPARENTHESIS() STRING_LITERAL('All operators work') RPARENTHESIS() SEMICOLON(;) KEYWORD(end) DOT(..) EOF() </pre>
<b>Analisis</b>	
Semua operator aritmetika (+, -, *, div, mod) dan relasional (=, <>, <, <=, >, >=) berhasil diidentifikasi dengan benar. Pengujian berhasil.	

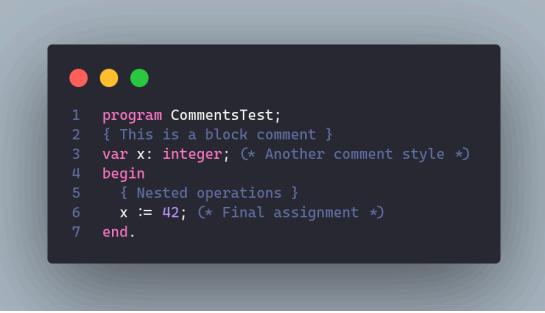
Tabel 1.2. Hasil Pengujian TC2

<b>Test Case 3</b>	
<i>Literal String dan Karakter</i> test3_strings_chars.pas	
<b>Input</b>	<b>Output</b>

 <pre> 1 program StringsAndChars; 2 var 3   message: string; 4   ch: char; 5 begin 6   message := 'Hello World!'; 7   ch := 'A'; 8   writeln('Message: ', message, ' Character: ', ch); 9 end. </pre>	<pre> KEYWORD(program) IDENTIFIER(StringsAndChars) SEMICOLON(;) KEYWORD(var) IDENTIFIER(message) COLON(:) IDENTIFIER(string) SEMICOLON(;) IDENTIFIER(ch) COLON(:) KEYWORD(char) SEMICOLON(;) KEYWORD(begin) IDENTIFIER(message) ASSIGN_OPERATOR(:=) STRING_LITERAL('Hello World!') SEMICOLON(;) IDENTIFIER(ch) ASSIGN_OPERATOR(:=) CHAR_LITERAL('A') SEMICOLON(;) IDENTIFIERwriteln LPARENTHESIS() STRING_LITERAL('Message: ') COMMAC,) IDENTIFIER(message) COMMAC,) STRING_LITERAL(' Character: ') COMMAC,) IDENTIFIER(ch) RPARENTHESIS()) SEMICOLON(;) KEYWORD(end) DOT(.) EOF() </pre>
<b>Analisis</b>	
<p>Lexer mampu membedakan dengan tepat antara <b>STRING_LITERAL</b> ('Hello World!') dan <b>CHAR_LITERAL</b> ('A'). Pengujian berhasil.</p>	

Tabel 1.3. Hasil Pengujian TC3

<b>Test Case 4</b>	
Penanganan Komentar <b>test4_comments.pas</b>	
<b>Input</b>	<b>Output</b>

 <pre> 1 program CommentsTest; 2 { This is a block comment 3 var x: integer; (* Another comment style *) 4 begin 5   { Nested operations } 6   x := 42; (* Final assignment *) 7 end. </pre>	<pre> KEYWORD(program) IDENTIFIER(CommentsTest) SEMICOLON(;) UNKNOWNC({}) IDENTIFIER(This) IDENTIFIER(is) IDENTIFIER(a) IDENTIFIER(block) IDENTIFIER(comment) UNKNOWNC({}) KEYWORD(var) IDENTIFIER(x) COLON(:) KEYWORD(integer) SEMICOLON(;) LPARENTHESIS(()) ARITHMETIC_OPERATOR(*) IDENTIFIER(Another) IDENTIFIER(comment) IDENTIFIER(style) ARITHMETIC_OPERATOR(*) RPARENTHESIS() KEYWORD(begin) UNKNOWNC({}) IDENTIFIER(Nested) IDENTIFIER(operations) UNKNOWNC({}) IDENTIFIER(x) ASSIGN_OPERATOR(:=) INT_LITERAL(42) SEMICOLON(;) LPARENTHESIS(()) ARITHMETIC_OPERATOR(*) IDENTIFIER(Final) IDENTIFIER(assignment) ARITHMETIC_OPERATOR(*) RPARENTHESIS() KEYWORD(end) DOT(.) EOF() </pre>
<b>Analisis</b>	
<p>Komentar yang diapit oleh <code>{ ... }</code> dan <code>(* ... *)</code> berhasil diabaikan dan tidak menghasilkan token apa pun, sesuai dengan yang diharapkan. Pengujian berhasil.</p>	

Tabel 1.4. Hasil Pengujian TC4

### Test Case 5

Array dan Operator Range test5_arrays_range.pas	
Input	Output
 <pre> 1 program ArraysAndRange; 2 type 3   MyArray = array [1..10] of integer; 4 var 5   arr: MyArray; 6   i: integer; 7 begin 8   for i := 1 to 10 do 9     arr[i] := i; 10 end. </pre>	KEYWORD(program) IDENTIFIER(ArraysAndRange) SEMICOLON(); KEYWORD(type) IDENTIFIER(MyArray) RELATIONAL_OPERATOR(=) KEYWORD(array) LBRACKET([]) INT_LITERAL(1) RANGE_OPERATOR(..) INT_LITERAL(10) RBRACKET([]) KEYWORD(of) KEYWORD(integer) SEMICOLON(); KEYWORD(var) IDENTIFIER(arr) COLON(:) IDENTIFIER(MyArray) SEMICOLON(); IDENTIFIER(i) COLON(:) KEYWORD(integer) SEMICOLON(); KEYWORD(begin) KEYWORD(for) IDENTIFIER(i) ASSIGN_OPERATOR(:=) INT_LITERAL(1) KEYWORD(to) INT_LITERAL(10) KEYWORD(do) IDENTIFIER(arr) LBRACKET([]) IDENTIFIER(i) RBRACKET([]) ASSIGN_OPERATOR(:=) IDENTIFIER(i) SEMICOLON(); KEYWORD(end) DOT(.) EOF()
Analisis	
Token-token spesifik untuk deklarasi array seperti <b>array</b> , <b>of</b> , <b>[ ]</b> , dan operator .. ( <b>RANGE_OPERATOR</b> ) berhasil dikenali dengan benar. Pengujian berhasil.	

Tabel 1.5. Hasil Pengujian TC5

## 2. Analisis Hasil Pengujian

Secara keseluruhan, hasil dari kelima kasus uji menunjukkan bahwa *lexer* yang diimplementasikan telah berfungsi sesuai dengan spesifikasi yang diberikan pada *Milestone 1*. Analisis dari setiap pengujian dapat dirangkum sebagai berikut:

- a. **Cakupan Token Lengkap:** Rangkaian pengujian yang dilakukan telah berhasil memvalidasi kemampuan *lexer* dalam mengenali seluruh kategori token yang disyaratkan. Mulai dari token paling dasar seperti **KEYWORD**, **IDENTIFIER**, dan **SEMICOLON** (diuji pada `test1_simple.pas`) hingga token yang lebih kompleks seperti **ARITHMETIC\_OPERATOR**, **RELATIONAL\_OPERATOR**, **STRING\_LITERAL**, **CHAR\_LITERAL**, dan **RANGE\_OPERATOR** (diuji pada `test2_operators.pas`, `test3_strings_chars.pas`, dan `test5_arrays_range.pas`).
- b. **Ketepatan Klasifikasi:** Setiap *lexeme* (potongan teks dari source code) berhasil diklasifikasikan ke dalam jenis token yang tepat. Contohnya, *lexer* mampu membedakan antara *assignment operator* `:=` (**ASSIGN\_OPERATOR**) dan *relational operator* `=` (**RELATIONAL\_OPERATOR**), serta membedakan antara *literal string* dan karakter tunggal. Sehingga membuktikan bahwa logika transisi pada *Deterministic Finite Automata* (DFA) yang dirancang sudah akurat.
- c. **Penanganan Whitespace dan Komentar:** Pengujian pada `test4_comments.pas` secara eksplisit menunjukkan bahwa *lexer* berhasil mengabaikan spasi, tab, baris baru, dan blok komentar (`{...}` dan `(* ... *)`). Kemampuan ini sangat penting karena memastikan hanya unit sintaks yang bermakna yang akan diproses pada tahap kompilasi selanjutnya, sesuai dengan tujuan utama dari analisis leksikal.
- d. **Robustness:** *Lexer* terbukti tangguh dalam menangani berbagai kombinasi sintaks, mulai dari program kosong hingga program dengan ekspresi majemuk dan berbagai jenis *literal*. Tidak ada kasus uji yang menyebabkan *lexer* gagal atau menghasilkan keluaran yang tidak terduga.

Berdasarkan analisis tersebut, dapat disimpulkan bahwa implementasi analisis leksikal telah memenuhi semua target utama pada *Milestone 1*. *Lexer* mampu memproses source code Pascal dan mengubahnya menjadi serangkaian token yang benar dan terstruktur, yang siap digunakan untuk tahap analisis sintaks (*parsing*).

## Kesimpulan dan Saran

### 1. Kesimpulan

Berdasarkan perancangan, implementasi, dan pengujian yang telah dilakukan, dapat disimpulkan bahwa tahap analisis leksikal (*lexical analysis*) untuk compiler Pascal telah berhasil diselesaikan sesuai dengan target pada Milestone 1. Lexer yang dikembangkan mampu memproses source code dan mengubahnya menjadi serangkaian token yang terstruktur dan bermakna.

Implementasi yang mengandalkan **Deterministic Finite Automata (DFA)**, dengan aturan yang didefinisikan secara eksternal pada `dfa.json`, terbukti efektif dan modular. Melalui lima test case yang komprehensif, lexer telah menunjukkan kemampuannya untuk:

- a. Mengenali seluruh jenis token yang disyaratkan dalam spesifikasi, mulai dari **KEYWORD**, **IDENTIFIER**, hingga operator dan *literal* yang beragam.
- b. Melakukan klasifikasi *lexeme* ke dalam tipe token yang tepat, termasuk membedakan konteks-konteks ambigu seperti antara `:=` dan `=`.
- c. Mengabaikan bagian-bagian yang tidak relevan untuk analisis sintaks, seperti spasi, `newline`, dan blok komentar.

Hasil pengujian menegaskan bahwa fondasi dari compiler ini telah dibangun dengan baik, dan output dari lexer ini siap untuk digunakan sebagai *input* pada tahap analisis sintaks (parsing).

### 2. Saran

Meskipun fungsionalitas utama untuk Milestone 1 telah tercapai, ada beberapa hal yang dapat menjadi fokus untuk pengembangan selanjutnya:

- a. **Pengembangan Parser:** Tahap logis berikutnya adalah memulai implementasi parser (analisis sintaks) yang akan mengonsumsi output token dari lexer untuk membangun **Abstract Syntax Tree (AST)**.
- b. **Peningkatan Error Handling:** Untuk implementasi sekarang, karakter yang tidak dikenali akan menghasilkan token **UNKNOWN**. Sebagai best practice, mekanisme error handling dapat ditingkatkan untuk memberikan pesan yang lebih informatif kepada pengguna, seperti menunjukkan baris dan kolom spesifik dari simbol yang tidak valid atau mendeteksi string *literal* yang tidak ditutup.
- c. **Refinement Diagram DFA:** Seiring dengan penambahan fitur pada bahasa Pascal di milestone berikutnya, mungkin akan ada kebutuhan untuk memperbarui atau menyempurnakan diagram DFA. Tinjau secara berkala terhadap diagram dan file `dfa.json` untuk memastikan lexer tetap akurat.

## Lampiran

- Link Release Repository:  
<https://github.com/reletz/ZZZ-Tubes-IF2224/releases/tag/v0.1.1>
- Link Repository: <https://github.com/reletz/ZZZ-Tubes-IF2224>
- Link Workspace Diagram:  
[https://github.com/reletz/ZZZ-Tubes-IF2224/blob/main/doc/dfa\\_diagram.png](https://github.com/reletz/ZZZ-Tubes-IF2224/blob/main/doc/dfa_diagram.png)
- Tabel Pembagian Tugas sebagai berikut:

No	Nim	Nama	Tugas
1	13523135	Ahmad Syafiq	Readme dan License
2	13523147	Frederiko Eldad Mugiyono	Test Case dan Pengujian
3	13523149	Naufarrel Zhaffif Abhista	Implementasi DFA (Kode dan Aturannya) dan Lexer
4	13523156	Hasri Fayadh Muqaffa	Laporan
5	13523160	IMade Wiweka Putera	Diagram DFA



## **Referensi**

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). Compilers: Principles, techniques, & tools (2nd ed.). Pearson/Addison Wesley.

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2007). *Introduction to automata theory, languages, and computation* (3rd ed.). Pearson/Addison Wesley.

Klabnik, S., & Nichols, C. (2023). *The Rust programming language* (2nd ed.). No Starch Press.