Laporan Tugas Besar Milestone 1

IF2224 - Teori Bahasa Formal Otomata 2025/2026

Disusun oleh :

Kelompok AutoRejectAtas

Muhammad Syarafi Akmal	13522076
Kenneth Poenadi	13523040
Bob Kunanda	13523086
M. Zahran Ramadhan Ardiana	13523104

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

JL. GANESHA 10, BANDUNG 40132

2025

Daftar Isi

BAB I
LANDASAN TEORI

1.1. Finite Automata (FA)

Finite Automata (FA) adalah model matematika yang digunakan untuk merepresentasikan sistem dengan jumlah keadaan (state) terbatas dan bekerja dengan membaca input simbol satu per satu untuk berpindah antar state berdasarkan aturan transisi yang telah ditentukan. FA terdiri atas himpunan state (Q), himpunan alfabet (Σ), fungsi transisi (δ), state awal (q_0), dan himpunan state akhir (F). Finite Automata berfungsi sebagai alat untuk mengenali bahasa formal dan pola tertentu dalam teks atau data. Dalam konteks komputasi, FA digunakan untuk memodelkan sistem yang memiliki perilaku diskrit seperti pengenalan pola, pemeriksaan string, serta pengenalan token dalam proses kompilasi bahasa pemrograman.

1.2. Deterministic Finite Automata (DFA)

Deterministic Finite Automata (DFA) adalah bentuk khusus dari Finite Automata di mana setiap kombinasi state dan simbol input hanya memiliki satu transisi yang pasti, tanpa adanya ε-transition. Dengan demikian, DFA bekerja secara deterministik dan mampu menentukan secara pasti apakah suatu string diterima atau ditolak oleh sistem. Representasi formal DFA terdiri dari lima komponen yaitu $(Q, \Sigma, \delta, q_0, F)$, di mana $\delta(q, a) = q'$ menunjukkan perpindahan dari state q ke q′ ketika membaca simbol a.

Fungsi transisi yang diperluas, yang dilambangkan dengan $\hat{\delta}$. Berbeda dengan fungsi transisi biasa (δ) yang memetakan sebuah status dan sebuah simbol input, fungsi $\hat{\delta}$ memetakan sebuah status dan sebuah string ke status tujuan. Fungsi ini didefinisikan secara rekursif sebagai berikut:

Secara formal, fungsi transisi yang diperluas δ* didefinisikan secara rekursif sebagai berikut:

1. **Basis:**

$$\delta^*(q, \epsilon) = q$$

Artinya, jika DFA berada pada state *q* dan membaca string kosong (ε), maka ia tetap berada di state yang sama.

2. **Rekursif:**

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a)$$

- *w* adalah string dari alfabet input Σ*,
- *a* adalah simbol terakhir dari *w*,
- dan *δ* adalah fungsi transisi biasa.

3. Artinya, untuk mengetahui kemana DFA berpindah saat membaca string *wa*, kita pertama-tama mencari state setelah membaca *w* (yakni $\delta^*(q, w)$), kemudian dari state itu kita lihat ke mana DFA berpindah ketika membaca simbol terakhir *a*.

DFA banyak digunakan dalam implementasi lexical analyzer untuk mengenali pola atau token pada bahasa pemrograman, seperti kata kunci, operator, identifier, dan konstanta, karena sifatnya yang efisien dan tidak ambigu.

1.3. Analisis Leksikal

Lexer atau analisis leksikal merupakan tahap awal dalam proses kompilasi yang bertugas untuk membaca dan memproses kode sumber menjadi unit-unit terkecil yang bermakna yang disebut token. Token adalah representasi dari elemen-elemen dasar bahasa pemrograman seperti kata kunci, identifier, operator, konstanta, dan simbol khusus. Fungsi utama lexer adalah untuk memisahkan karakter-karakter dalam kode sumber menjadi token yang valid berdasarkan pola tertentu, serta mengabaikan elemen yang tidak relevan seperti spasi dan komentar.

Dalam konteks kompilasi, lexer berperan penting sebagai penghubung antara tahap pembacaan kode dengan tahap parsing, karena output dari lexer berupa deretan token yang

akan dianalisis lebih lanjut oleh parser untuk membentuk struktur sintaksis program. Proses kerja lexer didasarkan pada teori Regular Expression dan Finite Automata, khususnya DFA. Setiap pola token dapat didefinisikan sebagai regular expression yang kemudian dikonversi menjadi DFA agar dapat dikenali secara efisien dan deterministik. Dengan menggunakan DFA, lexer mampu menentukan dengan pasti jenis token dari setiap rangkaian karakter yang dibaca. Oleh karena itu, lexer berperan penting dalam memastikan bahwa setiap bagian kode sumber dikenali secara tepat sebelum diteruskan ke tahap analisis sintaksis dan semantik dalam proses kompilasi.

1.4. Bahasa Pemrograman

Bahasa pemrograman adalah sistem notasi formal yang dirancang untuk menulis instruksi yang dapat dieksekusi oleh komputer, sehingga memungkinkan programmer untuk mengkomunikasikan logika, algoritma, serta struktur data kepada mesin. Bahasa pemrograman memiliki tiga komponen utama yaitu sintaks, semantik, dan pragmatik. Berdasarkan tingkatannya, bahasa pemrograman dibedakan menjadi bahasa tingkat rendah dan bahasa tingkat tinggi. Dalam teori bahasa formal, struktur bahasa pemrograman dapat dimodelkan menggunakan automata dan grammar seperti Context-Free Grammar (CFG), di mana automata berperan dalam mengenali pola sintaksis yang valid untuk proses analisis kode.

BAB II
PERANCANGAN & IMPLEMENTASI

2.1. Penjelasan Diagram

Diagram DFA ini dirancang berdasarkan aturan leksikal bahasa Pascal-S, yang mendefinisikan himpunan simbol dan pola karakter yang dapat membentuk token. Setiap token merepresentasikan unit dasar yang akan dikenali oleh lexer untuk proses analisis sintaksis berikutnya.

- **Identifier**

  Identifier adalah rangkaian karakter yang dimulai dengan huruf (A-Z, a-z) atau underscore (‘_’), dan dapat diikuti oleh huruf, angka, atau underscore tanpa spasi. Beberapa identifier tertentu yang cocok dengan entri dalam tabel kata kunci (keyword table) akan diklasifikasikan ulang sebagai:
  - Keyword, misalnya program, var, begin, end
  - Arithmetic Operator Keyword, misalnya div, mod
  - Logical Operator Keyword, misalnya and, or, not

- **Number / Float**

  Token NUMBER diawali oleh satu atau lebih digit numerik (0-9).

  Jika setelahnya terdapat titik (.) dan diikuti lagi oleh angka, maka terbentuk NUMBER_FLOAT.

  Selain itu, baik bilangan bulat maupun float dapat diikuti dengan notasi ilmiah (scientific notation), yaitu e atau E dengan opsional tanda +, -, atau spasi, kemudian diikuti oleh satu atau lebih digit angka.

  Contoh Valid: 43.7E5, 6E-5, 53E+4

- **Operator**

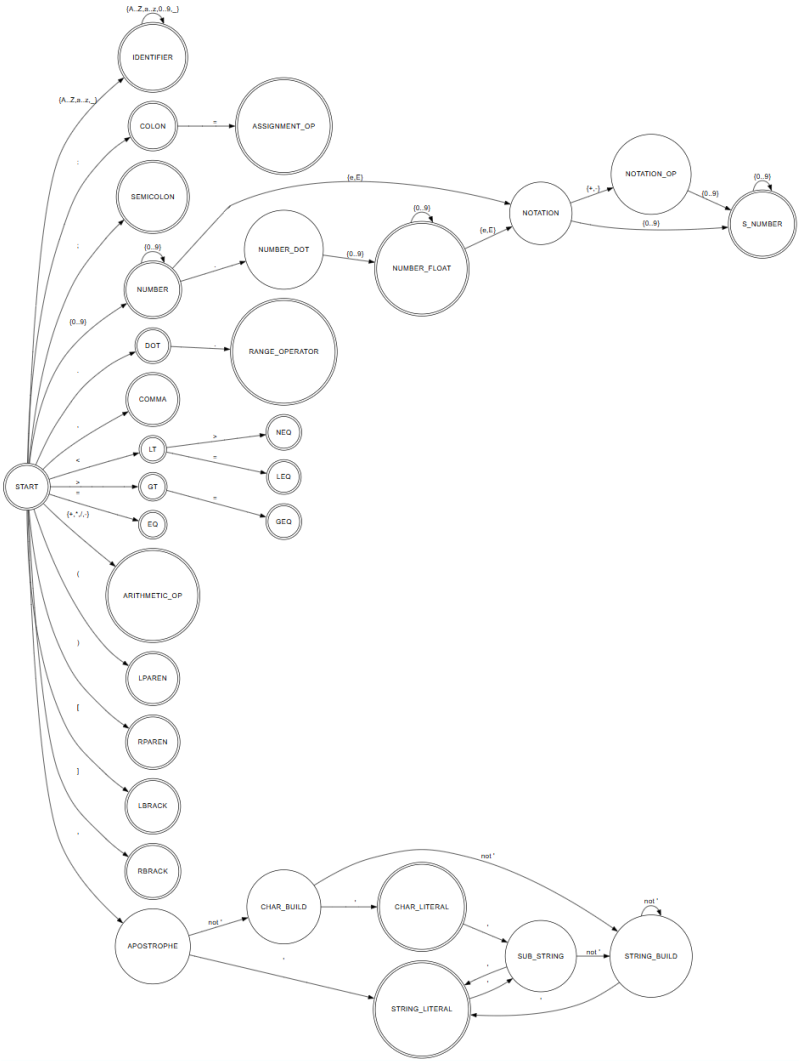
  Operator digunakan untuk melakukan operasi aritmetika dan relasional

- Operator aritmetika: +, -, *, /
- Operator relasional: <, >, <=, >=, =, <>
- Operator assignment: :=

- **String/Char literal**

  Literal karakter dan string ditulis di antara tanda petik tunggal (').

  - Jika hanya berisi satu karakter, maka menjadi CHAR_LITERAL.
  - Jika berisi lebih dari satu karakter, maka menjadi STRING_LITERAL.



Gambar 1. State Diagram Transisi DFA

Transisi DFA:

a. Identifier

Current State	Input Symbol	Destination State
START	A-Z a-z _	IDENTIFIER
IDENTIFIER	A-Z a-z 0-9_	IDENTIFIER

b. Number, Float, & Size

Current State	Input Symbol	Destination State
START	0-9	NUMBER
NUMBER	0-9	NUMBER
NUMBER	.	NUMBER_DOT
NUMBER	E e	NOTATION
NUMBER_DOT	0-9	NUMBER_FLOAT
NUMBER_FLOAT	0-9	NUMBER_FLOAT
NUMBER_FLOAT	E e	NOTATION
NOTATION	+ -	NOTATION_OP
NOTATION	0-9	S_NUMBER
NOTATION	0-9	S_NUMBER
NOTATION_OP	0-9	S_NUMBER
S_NUMBER	0-9	S_NUMBER

c. Operator

Current State	Input Symbol	Destination State
START	<	LT
START	>	GT
START	=	EQ
START	+ * / -	ARITHMETIC_OP
START	:	COLON
START	.	DOT
LT	=	LEQ
LT	>	NEQ
GT	=	GEQ
COLON	=	ASSIGNMENT_OP
DOT	.	RANGE_OPERATOR

d. String dan Char

Current State	Input Symbol	Destination State
START	'	APOSTROPHE
APOSTROPHE	Not '	CHAR_BUILD
APOSTROPHE	'	STRING_LITERAL
CHAR_BUILD	Not '	STRING_BUILD
CHAR_BUILD	'	CHAR_LITERAL
CHAR_LITERAL	'	SUB_STRING
STRING_BUILD	Not '	STRING_BUILD
STRING_BUILD	'	STRING_LITERAL
SUB_STRING	Not '	STRING_BUILD
SUB_STRING	'	STRING_LITERAL
STRING_LITERAL	'	SUB_STRING

Keterangan:

Pada pascal, untuk menuliskan sebuah Apostrophe (') dalam sebuah string memerlukan 2 buah apostrophe agar bekerja. Misal dalam code pascal tertulis ‘that''s’ maka outputnya adalah ‘that's, tetapi tokenizer yang kami buat tetap menyimpan apostrophe sebelumnya agar untuk dihandle ditahap selanjutnya. Kasus substring tersebut memiliki edge case ketika apostrophe dalam string ditaruh di awal atau di bagian akhir seperti ('''a'' is less than ''b''').

e. Token Lain

Current State	Input Symbol	Destination State
START	;	SEMICOLON
START	:	COLON
START	,	COMMA
START	(	LPAREN
START	)	RPAREN
START	[	LBRACK
START	]	RBRACK
START	.	DOT

2.2. Implementasi dfa_rules_final.json

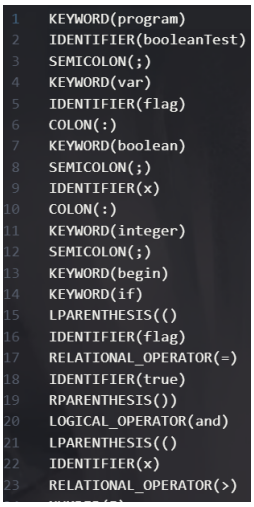
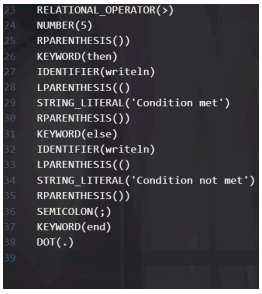
```
{
  "Start_state": "START",
  "Final_states": [
    "START",
    "IDENTIFIER",
    "COLON",
    "SEMICOLON",
    "NUMBER",
    "DOT",
    "COMMA",
    "LT",
    "LEQ",
    "NEQ",
    "GT",
    "EQ",
    "ARITHMETIC_OP",
    "LPAREN",
    "RPAREN",
    "LBRACK",
    "RBRACK",
    "GEQ",
    "ASSIGNMENT_OP",
    "NUMBER_FLOAT",
    "S_NUMBER",
    "CHAR_LITERAL",
    "STRING_LITERAL",
    "RANGE_OPERATOR"
  ],
  "Transitions": [
    ["START","A..Z,a..z,_","IDENTIFIER"],
    ["START",":","COLON"],
    ["START",";","SEMICOLON"],
    ["START","0..9","NUMBER"],
    ["START",".","DOT"],
    ["START",",","COMMA"],
    ["START","'","APOSTROPHE"],
    ["START","<","LT"],
    ["START",">","GT"],
    ["START","=","EQ"],
    ["START","+,*,/,-","ARITHMETIC_OP"],
    ["START","(","LPAREN"],
    ["START",")","RPAREN"],
    ["START","[","LBRACK"],
    ["START","]","RBRACK"],
    ["LT","=","LEQ"],
    ["LT",">","NEQ"],
    ["GT","=","GEQ"],
    ["DOT", ".", "RANGE_OPERATOR"],
```

```
    ["COLON","=","ASSIGNMENT_OP"],
    ["IDENTIFIER","A..Z,a..z,0..9,_","IDENTIFIER"],
    ["NUMBER","0..9","NUMBER"],
    ["NUMBER",".","NUMBER_DOT"],
    ["NUMBER_DOT","0..9","NUMBER_FLOAT"],
    ["NUMBER_FLOAT","0..9","NUMBER_FLOAT"],
    ["NUMBER","e,E","NOTATION"],
    ["NUMBER_FLOAT","e,E","NOTATION"],
    ["NOTATION","+,-","NOTATION_OP"],
    ["NOTATION","0..9","S_NUMBER"],
    ["NOTATION_OP","0..9","S_NUMBER"],
    ["S_NUMBER","0..9","S_NUMBER"],
    ["APOSTROPHE","ALL_EXCEPT '","CHAR_BUILD"],
    ["APOSTROPHE","'","STRING_LITERAL"],
    ["CHAR_BUILD","ALL_EXCEPT '","STRING_BUILD"],
    ["CHAR_BUILD","'","CHAR_LITERAL"],
    ["CHAR_LITERAL","'","SUB_STRING"],
    ["SUB_STRING", "ALL_EXCEPT '", "STRING_BUILD"],
    ["SUB_STRING", "'", "STRING_LITERAL"],
    ["STRING_BUILD","ALL_EXCEPT '","STRING_BUILD"],
    ["STRING_BUILD","'","STRING_LITERAL"],
    ["STRING_LITERAL","'","SUB_STRING"]
  ],

  "Token_mapping": {
    "START": "START",
    "IDENTIFIER": "IDENTIFIER",
    "COLON": "COLON",
    "SEMICOLON": "SEMICOLON",
    "NUMBER": "NUMBER",
    "DOT": "DOT",
    "COMMA": "COMMA",
    "LT": "RELATIONAL_OPERATOR",
    "LEQ": "RELATIONAL_OPERATOR",
    "NEQ": "RELATIONAL_OPERATOR",
    "GT": "RELATIONAL_OPERATOR",
    "EQ": "RELATIONAL_OPERATOR",
    "ARITHMETIC_OP": "ARITHMETIC_OPERATOR",
    "LPAREN": "LPARENTHESIS",
    "RPAREN": "RPARENTHESIS",
    "LBRACK": "LBRACKET",
    "RBRACK": "RBRACKET",
    "GEQ": "RELATIONAL_OPERATOR",
    "ASSIGNMENT_OP": "ASSIGN_OPERATOR",
    "NUMBER_FLOAT": "NUMBER",
    "S_NUMBER": "NUMBER",
    "CHAR_LITERAL": "CHAR_LITERAL",
    "STRING_LITERAL": "STRING_LITERAL",
    "RANGE_OPERATOR": "RANGE_OPERATOR"
  },
```

```
  "Error_states": {
    "STRING_BUILD": "String not closed",
    "SUB_STRING": "String not closed",
    "CHAR_BUILD": "Character not closed",
    "NOTATION": "Invalid scientific notation",
    "NOTATION_OP": "Invalid scientific notation"
  }
}
```

BAB III
PENGUJIAN

3.1. input-boolean.pas

Deskripsi	Input	Output
Output tidak memiliki error. Input-halo.pas bertujuan untuk testing negative, arithmetic, scientific notation, sub-string serta tipe2 comment (terutama ada edge case 1 apostrophe dalam comment)		 

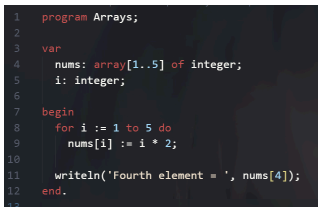
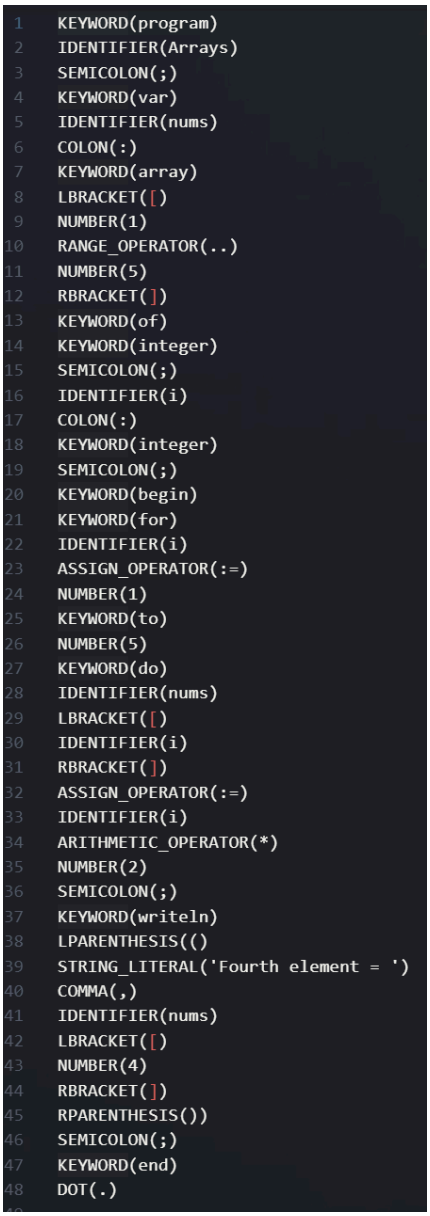
```
program booleanTest;

var
  flag : boolean;
  x : integer;

begin
  if ( flag = true ) and ( x > 5 ) then
    writeln('Condition met')
  else
    writeln('Condition not met');
end.
```

```
KEYWORD(program)
IDENTIFIER(booleanTest)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(flag)
COLON(:)
KEYWORD(boolean)
SEMICOLON(;)
IDENTIFIER(x)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(begin)
KEYWORD(if)
LPARENTHESIS(()
IDENTIFIER(flag)
RELATIONAL_OPERATOR(=)
IDENTIFIER(true)
RPARENTHESIS())
LOGICAL_OPERATOR(and)
LPARENTHESIS(()
IDENTIFIER(x)
RELATIONAL_OPERATOR(>)
```

```
RELATIONAL_OPERATOR(>)
NUMBER(5)
RPARENTHESIS())
KEYWORD(then)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Condition met')
RPARENTHESIS())
KEYWORD(else)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Condition not met')
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(end)
DOT(.)
```

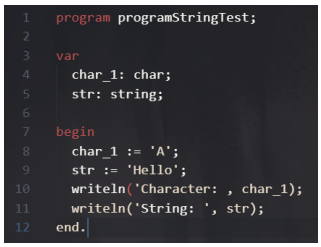
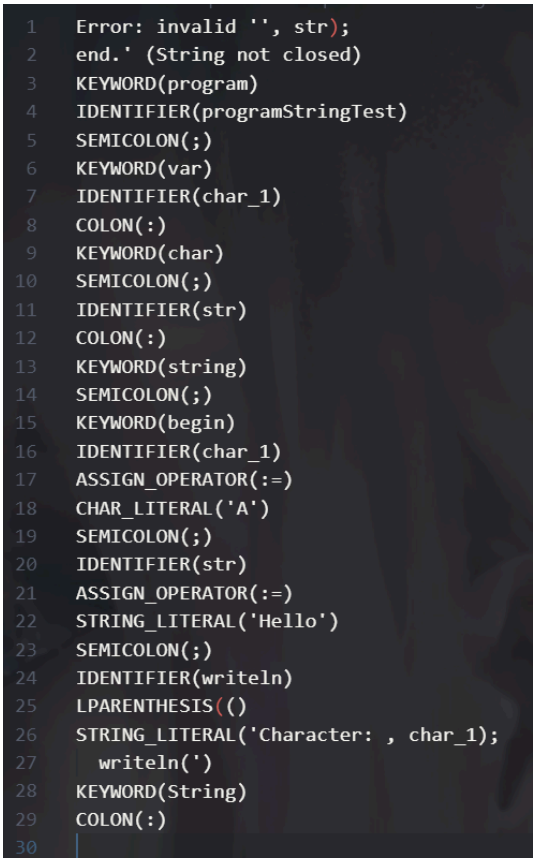
3.2. input-arrays-index.pas

Deskripsi	Input	Output

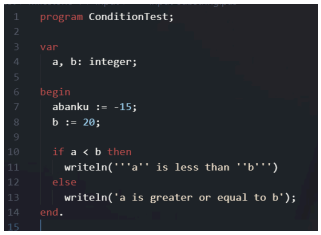
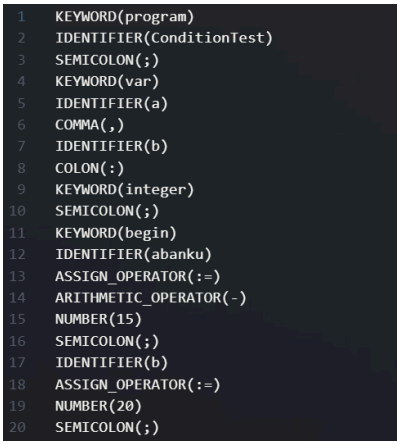
Output tidak memiliki error. Menguji range operator setelah number serta penggunaan bracket untuk array		

3.3. input-unclose-strings.pas

Deskripsi	Input	Output

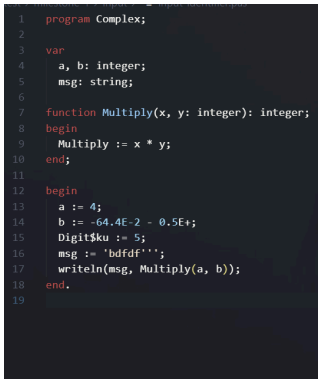
Output error string tidak tertutup. State mengambil apostrophe pada `'Character` dan ditutup pada `'String`, sehingga Apostrophe pada `',str` tidak memiliki pasangan penutup. Note* kalau ada error bakal di output duluan, lalu diberhentikan. Kasus disini dianggap terdeteksi errornya 2 baris terakhir (karena string menjadi 2 baris dan tidak tertutup)	 ``` program programStringTest; var char_1: char; str: string; begin char_1 := 'A'; str := 'Hello'; writeln('Character: , char_1); writeln('String: ', str); end. ```	 ``` Error: invalid '', str); end.' (String not closed) KEYWORD(program) IDENTIFIER(programStringTest) SEMICOLON(;) KEYWORD(var) IDENTIFIER(char_1) COLON(:) KEYWORD(char) SEMICOLON(;) IDENTIFIER(str) COLON(:) KEYWORD(string) SEMICOLON(;) KEYWORD(begin) IDENTIFIER(char_1) ASSIGN_OPERATOR(:=) CHAR_LITERAL('A') SEMICOLON(;) IDENTIFIER(str) ASSIGN_OPERATOR(:=) STRING_LITERAL('Hello') SEMICOLON(;) IDENTIFIER(writeln) LPARENTHESIS(() STRING_LITERAL('Character: , char_1); writeln(') KEYWORD(String) COLON(:) ```

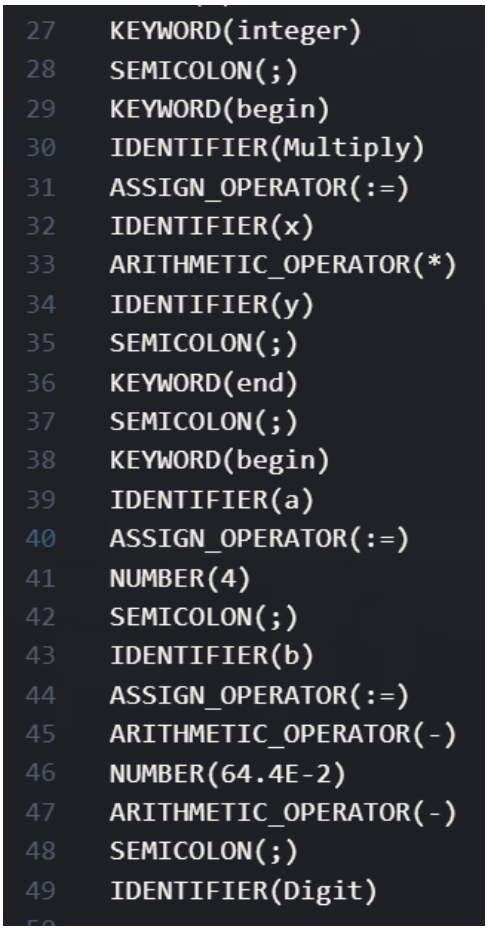
3.4. input-substring.pas

Deskripsi	Input	Output
Output tidak memiliki error, membuktikan dapat menghandle kasus double apostrophe pada sebuah string dan tidak memecahnya menjadi 2 string bagian.	 ``` program ConditionTest; var a, b: integer; begin abanku := -15; b := 20; if a < b then writeln('''a'' is less than ''b''') else writeln('a is greater or equal to b'); end. ```	 ``` KEYWORD(program) IDENTIFIER(ConditionTest) SEMICOLON(;) KEYWORD(var) IDENTIFIER(a) COMMA(,) IDENTIFIER(b) COLON(:) KEYWORD(integer) SEMICOLON(;) KEYWORD(begin) IDENTIFIER(abanku) ASSIGN_OPERATOR(:=) ARITHMETIC_OPERATOR(-) NUMBER(15) SEMICOLON(;) IDENTIFIER(b) ASSIGN_OPERATOR(:=) NUMBER(20) SEMICOLON(;) ```

		SEMICOLON(;) KEYWORD(if) IDENTIFIER(a) RELATIONAL_OPERATOR(<) IDENTIFIER(b) KEYWORD(then) IDENTIFIER(writeln) LPARENTHESIS(() STRING_LITERAL('''a'' is less than ''b''') RPARENTHESIS()) KEYWORD(else) IDENTIFIER(writeln) LPARENTHESIS(() STRING_LITERAL('a is greater or equal to b') RPARENTHESIS()) SEMICOLON(;) KEYWORD(end) DOT(.)

3.5. input-identifier.pas

Deskripsi	Input	Output
Output error karena ketika berada di state identifier (Digit) bertemu '$' sehingga berhenti memproses. Output di sini juga membuktikan sudah dapat menghandle kasus scientific notation (NUMBER (-64.4E-2) beserta error nya.	 program Complex; var a, b: integer; msg: string; function Multiply(x, y: integer): integer; begin Multiply := x * y; end; begin a := 4; b := -64.4E-2 - 0.5E+; Digit$ku := 5; msg := 'bdfdf'''; writeln(msg, Multiply(a, b)); end.	Error: invalid '0.5E+' (Invalid scientific notation) Error: Unknown symbol '$' KEYWORD(program) IDENTIFIER(Complex) SEMICOLON(;) KEYWORD(var) IDENTIFIER(a) COMMA(,) IDENTIFIER(b) COLON(:) KEYWORD(integer) SEMICOLON(;) IDENTIFIER(msg) COLON(:) KEYWORD(string) SEMICOLON(;) KEYWORD(function) IDENTIFIER(Multiply) LPARENTHESIS(() IDENTIFIER(x) COMMA(,) IDENTIFIER(y) COLON(:) KEYWORD(integer) RPARENTHESIS()) COLON(:) KEYWORD(integer)

		 KEYWORD(integer) SEMICOLON(;) KEYWORD(begin) IDENTIFIER(Multiply) ASSIGN_OPERATOR(:=) IDENTIFIER(x) ARITHMETIC_OPERATOR(*) IDENTIFIER(y) SEMICOLON(;) KEYWORD(end) SEMICOLON(;) KEYWORD(begin) IDENTIFIER(a) ASSIGN_OPERATOR(:=) NUMBER(4) SEMICOLON(;) IDENTIFIER(b) ASSIGN_OPERATOR(:=) ARITHMETIC_OPERATOR(-) NUMBER(64.4E-2) ARITHMETIC_OPERATOR(-) SEMICOLON(;) IDENTIFIER(Digit)

BAB IV
KESIMPULAN & SARAN

4.1. Kesimpulan

Dari proses perancangan dan implementasi *lexer* berbasis Deterministic Finite Automaton untuk bahasa Pascal-S, dapat disimpulkan bahwa proyek ini berhasil merealisasikan konsep teori automata ke dalam praktik nyata pada tahap awal proses kompilasi. Konsep *regular language* dan *finite automata* yang dipelajari secara teoretis dapat diterapkan langsung untuk mengenali pola token dalam bahasa pemrograman, di mana setiap token Pascal-S direpresentasikan sebagai ekspresi regular yang dimodelkan melalui transisi DFA. Dengan sifat deterministiknya, setiap pasangan *(state, input)* hanya memiliki satu arah transisi, sehingga proses pengenalan token berjalan secara efisien dengan kompleksitas linear O(n) tanpa perlu *backtracking*.

Implementasi tabel transisi δ (*transition table*) juga membuat sistem mudah diperluas atau dimodifikasi tanpa mengubah logika utama. Lexer yang dihasilkan mampu mengenali seluruh komponen sintaks dasar Pascal-S seperti *identifier*, *number*, operator majemuk (:=, <=, <>, ..), *delimiter*, serta literal karakter dan string berantai. Selain itu, mekanisme penanganan kesalahan seperti *unclosed string* dan karakter ilegal telah berjalan dengan baik, menghasilkan pesan galat yang informatif disertai posisi baris dan kolom. Berdasarkan hasil pengujian terhadap berkas uji seperti input-halo.pas, input-arrays-index.pas, input-unclosed-string.pas, dan input-unclosed-comment.pas, lexer terbukti mampu menghasilkan keluaran token yang sesuai dengan spesifikasi leksikal Pascal-S dan bebas kesalahan pada input valid. Secara keseluruhan, implementasi ini berhasil memenuhi tujuan Milestone 1, yaitu membangun *lexical analyzer* deterministik yang stabil, efisien, dan selaras dengan prinsip teori bahasa formal dan automata.

4.2. Saran

Sebagai saran, pengembangan selanjutnya dapat difokuskan pada peningkatan modularitas dan efisiensi agar lexer lebih mudah diintegrasikan dengan tahap-tahap

kompilasi berikutnya seperti *parser* dan *semantic analyzer*. Struktur tabel transisi DFA dapat dioptimalkan menggunakan *hash map* atau struktur data dua dimensi agar pencarian transisi antar state berlangsung lebih cepat, terutama untuk kode sumber berukuran besar. Selain itu, disarankan untuk menambahkan *unit testing* otomatis bagi setiap kategori token serta melakukan *stress testing* terhadap kasus ekstrem seperti komentar bersarang dan literal panjang untuk memastikan stabilitas serta ketahanan sistem. Dokumentasi kode dan penulisan komentar yang konsisten juga perlu diperhatikan agar proses pengembangan selanjutnya menjadi lebih mudah dipahami oleh seluruh anggota tim. Terakhir, pengembangan dapat diarahkan pada perluasan cakupan lexer agar mendukung fitur tambahan bahasa Pascal lainnya, seperti tipe data *real*, *boolean*, dan *record*, dengan menambahkan aturan regular expression serta transisi baru tanpa harus mengubah arsitektur utama yang sudah ada.

BAB V

LAMPIRAN

Link Release Repository Github:

https://github.com/KennethhPoenadi/ARA-Tubes-IF2224.git

Link workspace Diagram

https://magjac.com/graphviz-visual-editor/?dot=digraph%20finite_state_machine%20%7B%0D%0A%20fontname%3D%22Helvetica%2CArial%2Csans-serif%22%0D%0A%20node%20%5Bfontname%3D%22Helvetica%2CArial%2Csans-serif%22%5D%0D%0A%20edge%20%5Bfontname%3D%22Helvetica%2CArial%2Csans-serif%22%5D%0D%0A%20rankdir%3DLR%3B%0D%0A%0D%0A%20node%20%5Bshape%20%3D%20doublecircle%5D%3B%20START%20IDENTIFIER%20COLON%20SEMICOLON%20NUMBER%20DOT%20COMMA%20LT%20LEQ%20NEQ%20GT%20EQ%20ARITHMETIC_OP%20LPAREN%20RPAREN%20LBRACK%20RBRACK%20GEQ%20ASSIGNMENT_OP%20NUMBER_FLOAT%20S_NUMBER%20CHAR_LITERAL%20STRING_LITERAL%20RANGE_OPERATOR%3B%0D%0A%20node%20%5Bshape%20%3D%20circle%5D%3B%0D%0A%0D%0A%20START%20-%3E%20IDENTIFIER%20%5Blabel%3D%22%7BA..Z%2Ca..z%2C_%7D%22%5D%3B%0D%0A%20START%20-%3E%20COLON%20%5Blabel%3D%22%3A%22%5D%3B%0D%0A%20START%20-%3E%20SEMICOLON%20%5Blabel%3D%22%3B%22%5D%3B%0D%0A%20START%20-%3E%20NUMBER%20%5Blabel%3D%22%7B0..9%7D%22%5D%3B%0D%0A%20START%20-%3E%20DOT%20%5Blabel%3D%22.%22%5D%3B%0D%0A%20START%20-%3E%20COMMA%20%5Blabel%3D%22%2C%22%5D%3B%0D%0A%20START%20-%3E%20APOSTROPHE%20%5Blabel%3D%22%27%22%5D%3B%0D%0A%20START%20-%3E%20LT%20%5Blabel%3D%22%3C%22%5D%3B%0D%0A%20START%20-%3E%20GT%20%5Blabel%3D%22%3E%22%5D%3B%0D%0A%20START%20-%3E%20EQ%20%5Blabel%3D%22%3D%22%5D%3B%0D%0A%20START%20-%3E%20ARITHMETIC_OP%20%5Blabel%3D%22%7B%2B%2C%2A%2C%2F%2C-%7D%22%5D%3B%0D%0A%20START%20-%3E%20LPAREN%20%5Blabel%3D%22%28%22%5D%3B%0D%0A%20START%20-%3E%20RPAREN%20%5Blabel%3D%22%29%22%5D%3B%0D%0A%20START%20-%3E%20LBRACK%20%5Blabel%3D%22%5B%22%5D%3B%0D%0A%20START%20-%3E%20RBRACK%20%5Blabel%3D%22%5D%22%5D%3B%0D%0A%20%0D%0A%20LT%20-%3E%20LEQ%20%5Blabel%3D%22%3D%22%5D%3B%0D%0A%20LT%20-%3E%20NEQ%20%5Blabel%3D%22%3E%22%5D%3B%0D%0A%20GT%20-%3E%20GEQ%20%5Blabel%3D%22%3D%22%5D%3B%0D%0A%20COLON%20-%3E%20ASSIGNMENT_OP%20%5Blabel%3D%22%3D%22%5D%3B%0D%0A%0D%0A%20IDENTIFIER%20-%3E%20IDENTIFIER%20%5Blabel%3D%22%7BA..Z%2Ca..z%2C0..9%2C_%7D%22%5D%3B%0D%0A%20NUMBER%20-%3E%20NUMBER%20%5Blabel%3D%22%7B0..9%7D%22%5D%3B%0D

%0A%20NUMBER%20-%3E%20NUMBER_DOT%20%5Blabel%3D%22.%22%5D%3B%0D%0A%20NUMBER_DOT%20-%3E%20NUMBER_FLOAT%20%5Blabel%3D%22%7B0..9%7D%22%5D%3B%0D%0A%20NUMBER_FLOAT%20-%3E%20NUMBER_FLOAT%20%5Blabel%3D%22%7B0..9%7D%22%5D%3B%0D%0A%20DOT%20-%3E%20RANGE_OPERATOR%5Blabel%3D%22.%22%5D%0D%0A%0D%0A%20NUMBER%20-%3E%20NOTATION%20%5Blabel%3D%22%7Be%2CE%7D%22%5D%3B%0D%0A%20NUMBER_FLOAT%20-%3E%20NOTATION%20%5Blabel%3D%22%7Be%2CE%7D%22%5D%3B%0D%0A%20NOTATION%20-%3E%20NOTATION_OP%20%5Blabel%3D%22%7B%2B%2C-%7D%22%5D%3B%0D%0A%20NOTATION%20-%3E%20S_NUMBER%20%5Blabel%3D%22%7B0..9%7D%22%5D%3B%0D%0A%20NOTATION_OP%20-%3E%20S_NUMBER%20%5Blabel%3D%22%7B0..9%7D%22%5D%3B%0D%0A%20S_NUMBER%20-%3E%20S_NUMBER%20%5Blabel%3D%22%7B0..9%7D%22%5D%3B%0D%0A%0D%0A%20APOSTROPHE%20-%3E%20CHAR_BUILD%20%5Blabel%3D%22not%20%27%22%5D%3B%0D%0A%20APOSTROPHE%20-%3E%20STRING_LITERAL%5Blabel%3D%22%5C%27%22%5D%0D%0A%20CHAR_BUILD%20-%3E%20STRING_BUILD%20%5Blabel%3D%22not%20%27%22%5D%3B%0D%0A%20CHAR_BUILD%20-%3E%20CHAR_LITERAL%20%5Blabel%3D%22%27%22%5D%3B%0D%0A%20CHAR_LITERAL%20-%3E%20SUB_STRING%20%5Blabel%3D%22%27%22%5D%3B%0D%0A%20SUB_STRING%20-%3E%20STRING_LITERAL%5Blabel%3D%22%27%22%5D%3B%0D%0A%20STRING_BUILD%20-%3E%20STRING_BUILD%20%5Blabel%3D%22not%20%27%22%5D%3B%0D%0A%20STRING_BUILD%20-%3E%20STRING_LITERAL%20%5Blabel%3D%22%27%22%5D%3B%0D%0A%20STRING_LITERAL%20-%3E%20SUB_STRING%20%5Blabel%3D%22%27%22%5D%3B%0D%0A%20SUB_STRING%20-%3E%20STRING_BUILD%20%5Blabel%3D%22not%20%27%22%5D%3B%0D%0A%0D%0A%7D

Nama	NIM	Bagian Kerja
Muhammad Syarafi Akmal	13522076	Rules, Lexer, Test Case, Laporan
Kenneth Poenadi	13523040	Rules, Diagram DFA, Laporan
M Zahran Ramadhan	13523104	Rules, Lexer, Test Case, Laporan
Bob Kunanda	13523086	Rules, Diagram DFA, Laporan

BAB VI

REFERENSI

[1] NJIT, “Formal Definition of DFA,” *CS 341 Notes*, New Jersey Institute of Technology, 2025. [Online]. web.njit.edu

[2] Univ. of Pennsylvania, “Deterministic Finite Automata (DFA’s),” *CIS 2620 Notes*, Univ. of Pennsylvania, 2019. [Online]. Penn Engineering

[3] Cornell Univ., “Lecture 21: deterministic finite automata,” *CS 2800 Discrete Structures*, 2017. [Online]. Cornell Computer Science

[4] Portland State Univ., “Regular Languages—Finite Automata 5-tuple (Q, Σ, δ, q_0, F),” *TOC Notes*, 2018. [Online]. PSU | Portland State University

[5] Harvard Univ., S. Chong, “CS153 Lecture 8: Lexing—RE → NFA → DFA; scanner implementation,” *Course Slides*, 2019. [Online]. groups.seas.harvard.edu