Tugas Besar IF2224 Teori Bahasa Formal & Otomata

Lexical Analysis (Lexer)

Oleh:

Kelompok JJK - JanganJanganKecompile

Muhammad Raihaan Perdana	13523124
Danendra Shafi Athallah	13523136
Nathanael Rachmat	13523142
M. Abizzar Gamadrian	13523155

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 Kompiler dan Tahapannya

Bahasa pemrograman merupakan jembatan komunikasi antara manusia dengan komputer. Agar instruksi yang ditulis dalam bahasa tingkat tinggi seperti Pascal-S dapat dimengerti dan dieksekusi oleh mesin, diperlukan sebuah program penerjemah yang disebut **kompiler** (*compiler*). Kompiler secara sistematis menerjemahkan kode sumber (*source code*) menjadi kode mesin (*machine code*) atau format tingkat rendah lainnya tanpa mengubah logika fungsionalitasnya. Proses ini tidak hanya memungkinkan eksekusi program, tetapi juga berperan penting dalam optimasi dan deteksi kesalahan dasar, sehingga menghasilkan aplikasi yang efisien dan andal.

Proses kompilasi secara umum terbagi menjadi beberapa tahapan utama yang berurutan, di mana keluaran dari satu tahap menjadi masukan bagi tahap berikutnya. Tahapan-tahapan tersebut adalah:

1. **Analisis Leksikal (*Lexical Analysis*):** Tahap awal yang memecah kode sumber menjadi unit-unit leksikal terkecil yang disebut *token*.
2. **Analisis Sintaksis (*Syntax Analysis*):** Memeriksa struktur gramatikal dari urutan *token* untuk memastikan sesuai dengan aturan sintaks bahasa pemrograman, biasanya dalam bentuk *parse tree*.
3. **Analisis Semantik (*Semantic Analysis*):** Memeriksa makna dan konsistensi dari program, seperti memeriksa kesesuaian tipe data dalam sebuah operasi.
4. **Generasi Kode Perantara (*Intermediate Code Generation*):** Menghasilkan representasi program dalam bentuk abstrak yang mudah dioptimasi dan diterjemahkan ke berbagai arsitektur mesin.
5. **Interpretasi (*Interpreter*):** Mengeksekusi kode, baik kode perantara maupun kode mesin, untuk menghasilkan *output* akhir.

Milestone pertama dalam tugas besar ini berfokus secara eksklusif pada implementasi tahap pertama, yaitu Analisis Leksikal.

1.2 Analisis Leksikal

Analisis leksikal adalah fondasi dari proses kompilasi. Pada tahap ini, rangkaian karakter mentah dari kode sumber dibaca dan dikelompokkan menjadi satuan-satuan bermakna yang disebut *token*. Komponen yang bertanggung jawab untuk melakukan tugas ini dikenal sebagai **penganalisis leksikal**, atau lebih umum disebut ***lexer*** atau ***scanner***.

Tujuan utama dari analisis leksikal adalah menyederhanakan tugas bagi tahap selanjutnya, yaitu analisis sintaksis (parsing). Daripada memproses karakter satu per satu, *parser* akan menerima aliran *token* yang lebih terstruktur, sehingga lebih mudah untuk mengenali struktur program. Selain itu, *lexer* juga bertugas untuk membuang elemen-elemen yang tidak relevan dengan logika program, seperti spasi, *tab*, dan baris baru.

1.3 Token

Token adalah unit leksikal terkecil dalam sebuah bahasa pemrograman yang memiliki makna. Setiap *token* yang dihasilkan oleh *lexer* umumnya terdiri dari dua komponen utama:

1. **Tipe Token (*Token Type*):** Kategori dari unit leksikal tersebut. Contohnya, KEYWORD, IDENTIFIER, NUMBER, atau ARITHMETIC_OPERATOR.
2. **Nilai Token (*Token Value*):** Rangkaian karakter aktual (disebut juga *lexeme*) yang cocok dengan pola tipe *token*. Contohnya, untuk *lexeme* "program", tipe *token*-nya adalah KEYWORD. Untuk *lexeme* "x", tipe *token*-nya adalah IDENTIFIER.

Sebagai ilustrasi, jika *lexer* memindai baris kode a := 5;, maka ia akan menghasilkan urutan *token* sebagai berikut: IDENTIFIER(a), ASSIGN_OPERATOR(:=), NUMBER(5), dan SEMICOLON(;)

1.4 Deterministic Finite Automaton (DFA)

Untuk mengenali pola-pola karakter dan mengelompokkannya menjadi *token*, *lexer* menggunakan model komputasi yang disebut **Deterministic Finite Automaton (DFA)**. Dalam teori bahasa formal, himpunan *token* dalam sebuah bahasa pemrograman dapat direpresentasikan sebagai **bahasa reguler**, yang secara matematis dapat dikenali oleh *finite automata*.

Secara formal, sebuah DFA didefinisikan sebagai 5-tuple $(Q, \Sigma, \delta, q_0, F)$, di mana:

- **Q** adalah himpunan hingga dari *state* (keadaan).

- **Σ** adalah himpunan hingga dari simbol input (alfabet).
- **δ** adalah fungsi transisi ($Q \times \Sigma \rightarrow Q$).
- $\mathbf{q_0}$ adalah *state* awal.
- **F** adalah himpunan *state* akhir (penerima).

Dalam konteks *lexer*, DFA bekerja dengan membaca kode sumber karakter demi karakter. Dimulai dari *state* awal, *lexer* akan berpindah dari satu *state* ke *state* lain berdasarkan karakter yang dibaca dan fungsi transisi yang telah didefinisikan. Ketika *lexer* mencapai *state* akhir, itu menandakan bahwa serangkaian karakter yang telah dibaca berhasil dikenali sebagai sebuah *token* yang valid. Implementasi *lexer* berbasis DFA adalah contoh nyata bagaimana model teoretis dari automata digunakan untuk memecahkan masalah praktis dalam konstruksi kompiler.

Bab II

Perancangan & Implementasi

2.1 Perancangan DFA

Inti dari *lexer* yang kami bangun adalah sebuah **Deterministic Finite Automaton (DFA)** yang dirancang untuk mengenali semua unit leksikal (token) dalam bahasa Pascal-S. Seluruh aturan transisi DFA didefinisikan dalam sebuah file eksternal dfa_rules.json agar mudah dikelola dan dimodifikasi. Perancangan DFA ini mencakup beberapa logika kunci untuk menangani berbagai jenis token:

- **Identifier dan Keyword:** DFA dirancang untuk mengenali pola umum *identifier*, yaitu dimulai dengan huruf dan diikuti oleh kombinasi huruf atau angka. Setelah sebuah *lexeme* yang cocok dengan pola ini dikenali, program akan melakukan pengecekan lebih lanjut untuk menentukan apakah *lexeme* tersebut merupakan salah satu *keyword* yang telah didefinisikan (seperti program, var, begin) atau operator logika/aritmetika berbentuk kata (and, or, div, mod). Jika tidak, maka ia diklasifikasikan sebagai IDENTIFIER.
- **Angka (Integer dan Real):** DFA memiliki *state* untuk mengenali urutan digit sebagai NUMBER (integer). Jika setelah urutan digit ditemukan karakter titik (.) yang diikuti oleh digit lain, maka DFA akan bertransisi ke *state* yang mengenalinya sebagai NUMBER (real), seperti pada 2.5 atau 0.0.
- **Operator Multi-Karakter:** Perancangan DFA secara cermat menangani operator yang dapat terdiri dari satu atau dua karakter. Contohnya, ketika menerima input karakter :, DFA akan masuk ke *state* antara (S_COLON). Dari *state* ini, jika karakter berikutnya adalah =, ia akan bertransisi ke *state* akhir S_ASSIGN untuk membentuk token ASSIGN_OPERATOR (:=). Jika tidak, token COLON (:) akan terbentuk. Logika serupa diterapkan untuk .., </>, <=, dan >=.
- **String Literal:** Untuk mengenali *string literal*, DFA memasuki *state* khusus (S_IN_STRING) setelah menemukan karakter kutip tunggal ('). DFA akan terus mengonsumsi karakter hingga menemukan kutip tunggal penutup.
- **Komentar:** Sesuai dengan spesifikasi awal, DFA juga dirancang untuk dapat mengenali dua jenis format komentar, yaitu yang diapit oleh { dan } serta (* dan *). Meskipun revisi spesifikasi tugas membatalkan implementasi token komentar, logika untuk melewati (mengabaikan) blok komentar ini tetap dipertahankan dalam implementasi *lexer*.

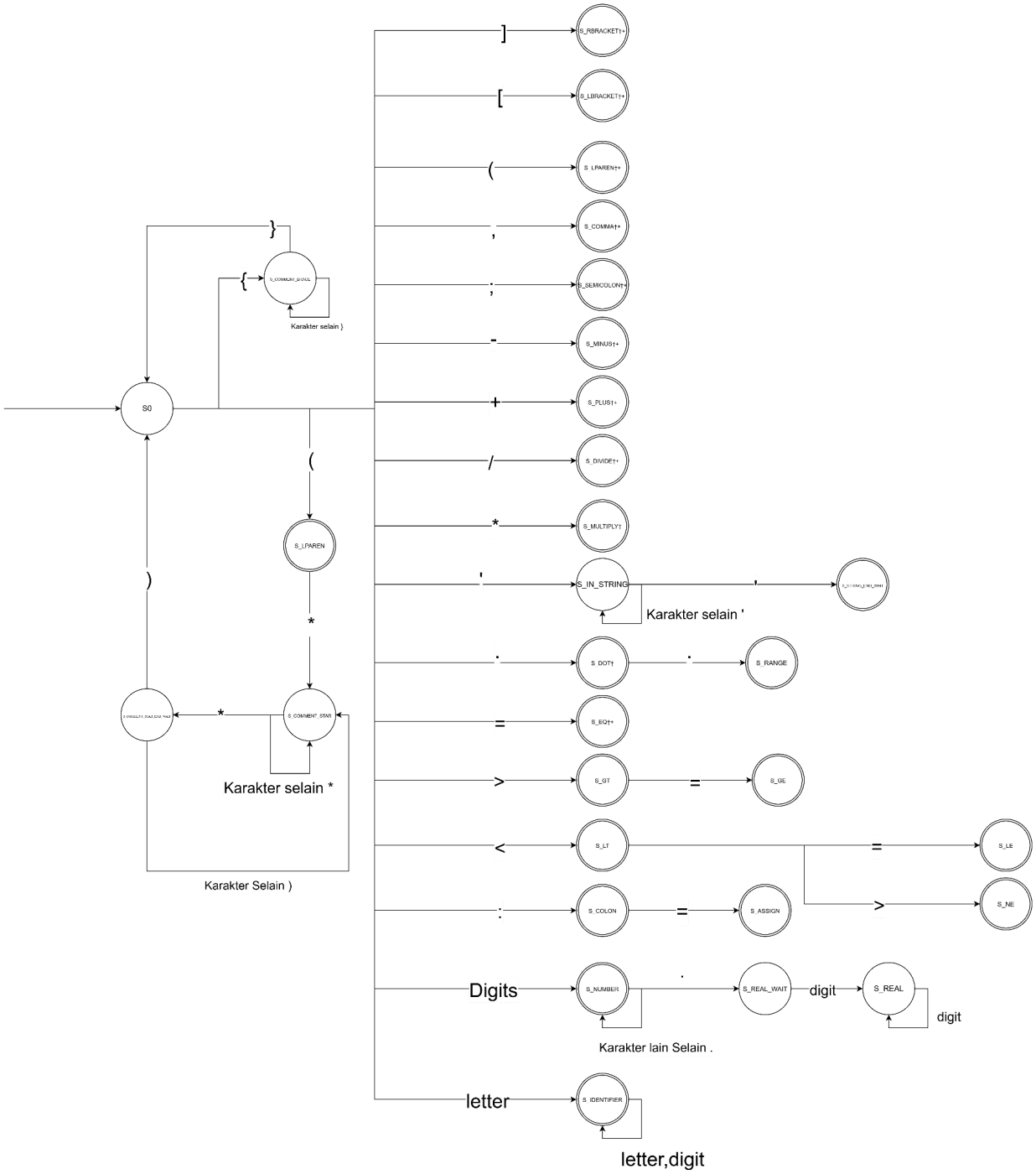


Diagram 1: Visualisasi Sederhana Alur Transisi DFA

2.2 Struktur Program

Untuk mengimplementasikan *lexer* ini, kami memilih bahasa pemrograman **Python**. Alasan utama pemilihan Python adalah karena kemampuannya yang sangat baik dalam manipulasi string dan parsing file, serta sintaksnya yang bersih dan mudah dibaca, yang mempercepat proses pengembangan. Selain itu, struktur data bawaan seperti *dictionary* sangat cocok untuk merepresentasikan tabel transisi DFA secara efisien.

Struktur proyek kami sangat modular untuk memisahkan tanggung jawab (*separation of concerns*), yang terdiri dari tiga file utama:

1. **compiler.py:** Bertindak sebagai *driver* atau titik masuk program. Skrip ini bertanggung jawab untuk memproses argumen *command-line*, membaca file sumber .pas, dan mengorkestrasi panggilan fungsi dari modul *lexer*.
2. **lexer.py:** Merupakan mesin utama dari *lexer*. Modul ini berisi semua logika inti, termasuk implementasi mesin DFA, fungsi untuk memuat aturan DFA dari file JSON, dan proses *tokenisasi* itu sendiri.
3. **dfa_rules.json:** File konfigurasi yang mendefinisikan "otak" dari *lexer*. File ini berisi definisi *start state*, *final states*, daftar *keywords*, dan yang terpenting, tabel transisi DFA.

Alur kerja program secara garis besar adalah sebagai berikut:

1. Pengguna menjalankan compiler.py dengan menyediakan path ke file kode Pascal-S sebagai argumen.
2. Program memuat aturan DFA dari dfa_rules.json ke dalam memori.
3. Isi dari file kode .pas dibaca sebagai sebuah string tunggal.
4. Fungsi tokenize dari lexer.py dipanggil untuk memproses string kode tersebut karakter per karakter menggunakan mesin DFA yang telah dimuat.
5. Mesin DFA berpindah *state* sesuai dengan karakter input. Ketika sebuah token selesai dikenali, token tersebut ditambahkan ke dalam sebuah daftar.
6. Setelah seluruh kode diproses, daftar *token* yang dihasilkan akan ditulis ke dalam sebuah file output (misalnya, output-1.txt) di direktori yang sama dengan file input.

2.3 Implementasi Detail

Detail implementasi dari mesin *lexer* kami berpusat pada fungsi tokenize di dalam lexer.py. Fungsi ini mengimplementasikan loop yang membaca kode sumber karakter demi karakter. Untuk menyederhanakan logika transisi, sebuah fungsi pembantu classify_char(c) digunakan untuk mengkategorikan setiap karakter (misalnya sebagai letter, digit, atau simbol lainnya) sebelum dicocokkan dengan tabel transisi DFA.

Snippet dari lexer.py untuk Fungsi Klasifikasi Karakter

```
def classify_char(c):
    if c.isalpha():
       return "letter"
    elif c.isdigit():
       return "digit"
    elif c.isspace():
       return "space"
    else:
       return c
```

Potongan Kode 1: Fungsi pembantu untuk mengkategorikan karakter input.

Di dalam loop utama tokenize, logika untuk transisi *state* diimplementasikan dengan mencari *state* berikutnya dari struktur data transisi yang dimuat dari dfa_rules.json. Baris berikut adalah inti dari mesin DFA kami:

Snippet dari lexer.py untuk Logika Transisi DFA

```
next_state = transitions.get(state, {}).get(char_type) or transitions.get(state, {}).get(c)
```

Potongan Kode 2: Pencarian state berikutnya berdasarkan state saat ini dan tipe karakter.

Ketika tidak ada transisi valid berikutnya (next_state adalah None), program mengasumsikan akhir dari *lexeme* saat ini. Jika *state* saat ini termasuk dalam final_states, sebuah token akan dibentuk. Pada titik inilah pengecekan khusus untuk membedakan IDENTIFIER dari KEYWORD dilakukan.

Snippet dari lexer.py untuk Pembedaan Identifier dan Keyword

```
if token_type == "IDENTIFIER":
   if word.lower() in dfa["keywords"]:
      token_type = "KEYWORD"
```

```
      word = word.lower()
……..(kode lainnya untuk pengecekan logical operator)
```

Potongan Kode 3: Logika pasca-pemrosesan untuk klasifikasi ulang token identifier.

Pendekatan ini memisahkan pengenalan pola (tugas DFA) dari klasifikasi semantik (*keyword* vs *identifier*), menghasilkan desain yang bersih dan efisien.

Bab III

Pengujian

3.1 Kasus Pengujian Unik

3.1.1 Uji Fungsionalitas Dasar

test_basic.pas

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

Pengujian ini bertujuan untuk memastikan lexer dapat mengenali token-token fundamental dalam struktur program Pascal-S.

3.1.2 Uji Tipe Data Real & Operator Kompleks

test_real_operators.pas

```
program ComputeSeries;

var
  i, n: integer;
  x, sum, term: real;

begin
  n := 10;
  x := 2.5;
  sum := 0.0;
  term := 1.0;

  i := 1;
  while i <= n do
```

```
  begin
    term := term * (x / i);
    sum := sum + term;

    if i mod 2 = 0 then
      sum := sum - (term / (i + 1))
    else
      sum := sum + (term / (i + 1));

    i := i + 1;
  end;

  if sum > 50.0 then
    sum := sum / 2.0
  else
    sum := sum * 2.0;

end.
```

Memvalidasi kemampuan lexer dalam mengenali bilangan real (desimal) sebagai token NUMBER dan membedakan operator aritmetika berbentuk kata (mod) dari identifier.

3.1.3 Uji Operator Relasional & Logika

test_relational_logic.pas

```
program TestOperators;
var
  a, b: integer;
  c, d: boolean;
begin
  a := 10;
  b := 20;
  c := (a < b) and (b > a);
  d := (a <> b) or not c;
  if (a <= b) and (b >= a) then
    a := 0;
```

```
end.
```

Menguji ketangguhan DFA dalam mengenali semua variasi operator relasional (terutama yang multi-karakter seperti <>, <=, >=) dan operator logika (and, or, not).

3.1.4 Uji Kasus Khusus String Literal

test_string_edgecase.pas

```
program TestStrings;
begin
  writeln('This is Pascal-S.');
   writeln('It''s a language.'); (* This line tests the double
quote *)
  writeln(''); (* Tests an empty string *)
end.
```

Memverifikasi bahwa lexer dapat menangani kasus-kasus khusus pada STRING_LITERAL, terutama string yang mengandung karakter kutip ganda ('') dan string kosong.

3.1.5 Uji Kasus Range Operator

test_range_operator.pas

```
program TestArrayRange;
type
  MyArray = array[1 .. 100] of integer;
var
  Numbers: MyArray;
begin
  Numbers[1] := 10;
end.
```

Memvalidasi kemampuan lexer untuk mengenali token RANGE_OPERATOR (..) yang biasa digunakan dalam definisi array di Pascal.

3.1.6 Uji Kasus Error Handling

test_error_handling.pas

```
program InvalidInput;
var
  a: integer;
begin
  a := 10 @ 5;
  # This is an invalid comment style
end.
```

Memvalidasi kemampuan lexer untuk mengenali token yang tidak valid dengan harapan lexer memberhentikan proses compile.

3.2 Hasil dan Pembahasan Pengujian

3.2.1 Hasil Uji Fungsionalitas Dasar

test > milestone-1 > result_basic.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(.)
```

Pembahasan: Hasil pengujian ini menunjukkan bahwa fungsionalitas inti dari lexer telah berhasil diimplementasikan. Lexer mampu memecah kode sumber menjadi unit-unit leksikal fundamental sesuai dengan spesifikasi bahasa Pascal-S. Keberhasilan ini divalidasi melalui beberapa poin kunci. Pertama, DFA secara akurat membedakan antara KEYWORD (misalnya, program, var, begin) dan

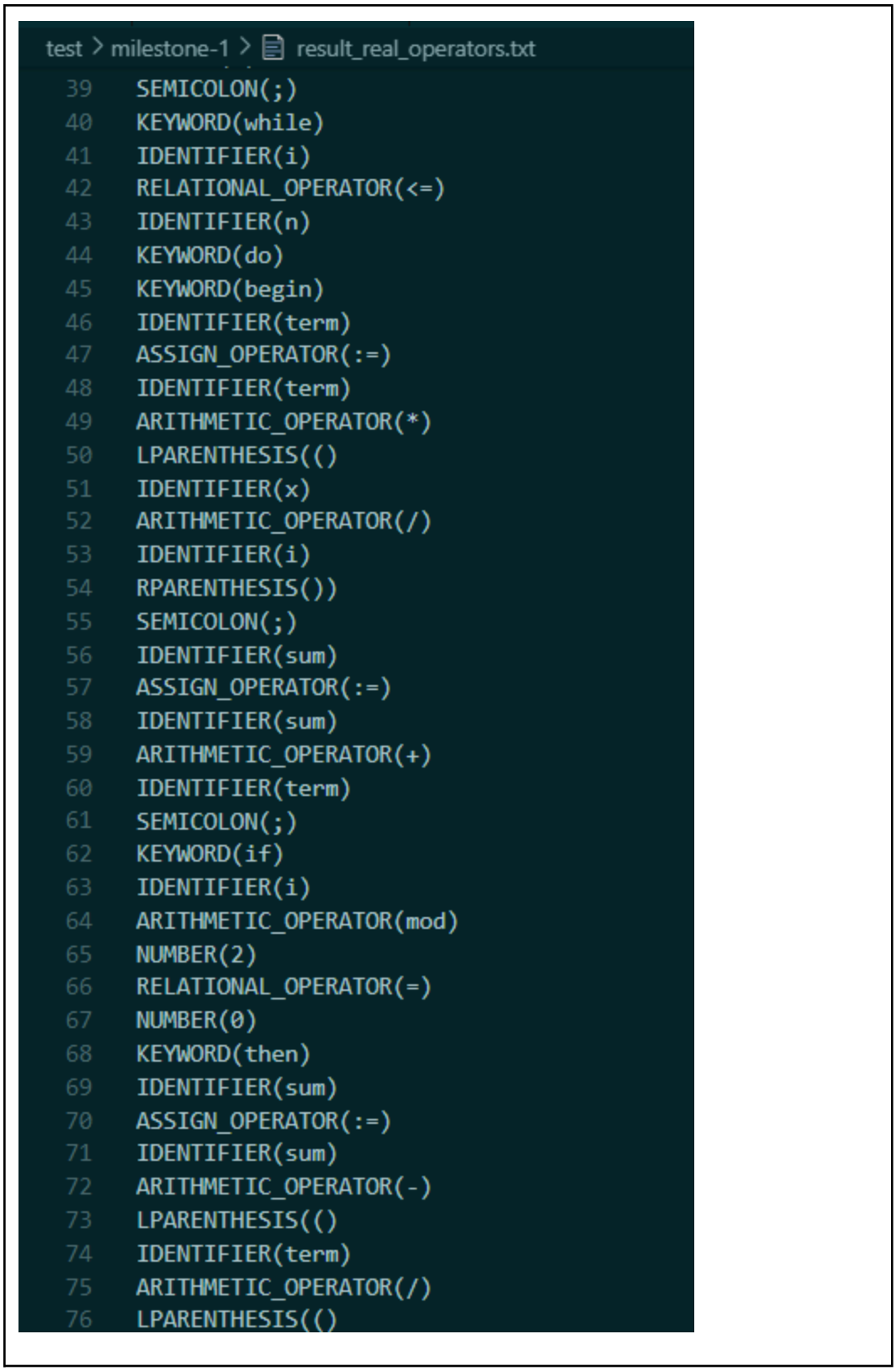
IDENTIFIER (misalnya, Hello, a, writeln). Ini membuktikan bahwa setelah mengenali pola identifier umum, program melakukan pengecekan terhadap daftar kata kunci yang telah didefinisikan.

Kedua, lexer berhasil mengenali operator multi-karakter seperti := sebagai satu token tunggal (ASSIGN_OPERATOR), yang menandakan transisi state pada DFA untuk operator sejenis ini sudah berjalan dengan benar. Selain itu, berbagai tipe token lain seperti NUMBER, STRING_LITERAL, dan semua simbol pungtuasi (misalnya SEMICOLON, LPARENTHESIS, DOT) juga berhasil diklasifikasikan. Terakhir, tidak adanya token untuk spasi atau baris baru pada output mengonfirmasi bahwa lexer secara efektif mengabaikan whitespace, yang merupakan salah satu tujuan utama dari analisis leksikal. Secara keseluruhan, pengujian dasar ini memberikan keyakinan bahwa fondasi DFA yang dirancang sudah solid dan siap untuk menangani kasus-kasus yang lebih kompleks

3.2.2 Hasil Uji Tipe Data Real & Operator Kompleks

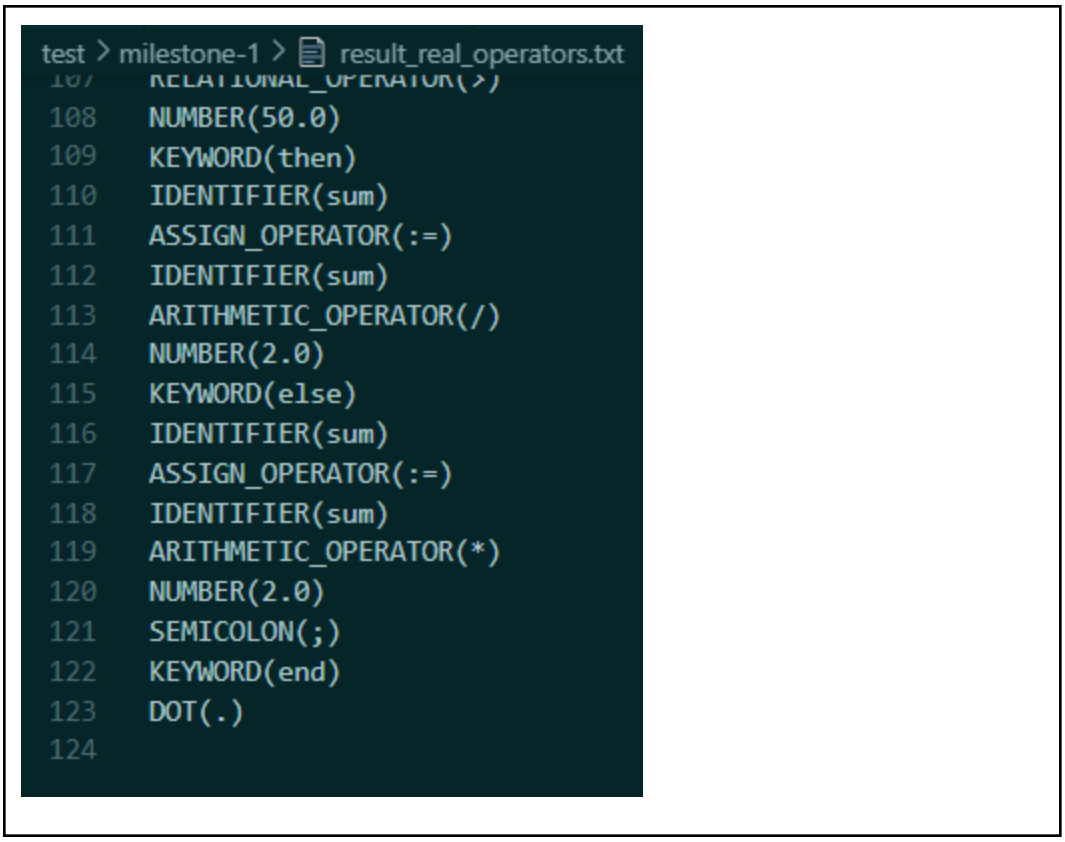
test > milestone-1 > result_real_operators.txt

```
KEYWORD(program)
IDENTIFIER(ComputeSeries)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(i)
COMMA(,)
IDENTIFIER(n)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(x)
COMMA(,)
IDENTIFIER(sum)
COMMA(,)
IDENTIFIER(term)
COLON(:)
KEYWORD(real)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(n)
ASSIGN_OPERATOR(:=)
NUMBER(10)
SEMICOLON(;)
IDENTIFIER(x)
ASSIGN_OPERATOR(:=)
NUMBER(2.5)
SEMICOLON(;)
IDENTIFIER(sum)
ASSIGN_OPERATOR(:=)
NUMBER(0.0)
SEMICOLON(;)
IDENTIFIER(term)
ASSIGN_OPERATOR(:=)
NUMBER(1.0)
SEMICOLON(;)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
NUMBER(1)
```


test > milestone-1 > result_real_operators.txt
39 SEMICOLON(;)
40 KEYWORD(while)
41 IDENTIFIER(i)
42 RELATIONAL_OPERATOR(<=)
43 IDENTIFIER(n)
44 KEYWORD(do)
45 KEYWORD(begin)
46 IDENTIFIER(term)
47 ASSIGN_OPERATOR(:=)
48 IDENTIFIER(term)
49 ARITHMETIC_OPERATOR(*)
50 LPARENTHESIS(()
51 IDENTIFIER(x)
52 ARITHMETIC_OPERATOR(/)
53 IDENTIFIER(i)
54 RPARENTHESIS())
55 SEMICOLON(;)
56 IDENTIFIER(sum)
57 ASSIGN_OPERATOR(:=)
58 IDENTIFIER(sum)
59 ARITHMETIC_OPERATOR(+)
60 IDENTIFIER(term)
61 SEMICOLON(;)
62 KEYWORD(if)
63 IDENTIFIER(i)
64 ARITHMETIC_OPERATOR(mod)
65 NUMBER(2)
66 RELATIONAL_OPERATOR(=)
67 NUMBER(0)
68 KEYWORD(then)
69 IDENTIFIER(sum)
70 ASSIGN_OPERATOR(:=)
71 IDENTIFIER(sum)
72 ARITHMETIC_OPERATOR(-)
73 LPARENTHESIS(()
74 IDENTIFIER(term)
75 ARITHMETIC_OPERATOR(/)
76 LPARENTHESIS(()

test > milestone-1 > result_real_operators.txt

```
77   IDENTIFIER(i)
78   ARITHMETIC_OPERATOR(+)
79   NUMBER(1)
80   RPARENTHESIS())
81   RPARENTHESIS())
82   KEYWORD(else)
83   IDENTIFIER(sum)
84   ASSIGN_OPERATOR(:=)
85   IDENTIFIER(sum)
86   ARITHMETIC_OPERATOR(+)
87   LPARENTHESIS(()
88   IDENTIFIER(term)
89   ARITHMETIC_OPERATOR(/)
90   LPARENTHESIS(()
91   IDENTIFIER(i)
92   ARITHMETIC_OPERATOR(+)
93   NUMBER(1)
94   RPARENTHESIS())
95   RPARENTHESIS())
96   SEMICOLON(;)
97   IDENTIFIER(i)
98   ASSIGN_OPERATOR(:=)
99   IDENTIFIER(i)
100  ARITHMETIC_OPERATOR(+)
101  NUMBER(1)
102  SEMICOLON(;)
103  KEYWORD(end)
104  SEMICOLON(;)
105  KEYWORD(if)
106  IDENTIFIER(sum)
107  RELATIONAL_OPERATOR(>)
108  NUMBER(50.0)
109  KEYWORD(then)
110  IDENTIFIER(sum)
111  ASSIGN_OPERATOR(:=)
112  IDENTIFIER(sum)
113  ARITHMETIC_OPERATOR(/)
114  NUMBER(2.0)
```



Pembahasan: Pengujian ini berhasil memvalidasi dua aspek penting dari DFA yang telah dirancang. Pertama, kemampuan lexer untuk mengenali bilangan real (desimal) telah terbukti. Token seperti NUMBER(2.5), NUMBER(0.0), dan NUMBER(50.0) menunjukkan bahwa DFA dapat bertransisi dengan benar dari state pengenalan digit ke state yang menangani titik desimal, dan kemudian kembali ke state pengenalan digit untuk bagian fraksional. Ini memastikan bahwa tipe data numerik, baik integer maupun real, dapat diproses dengan akurat.

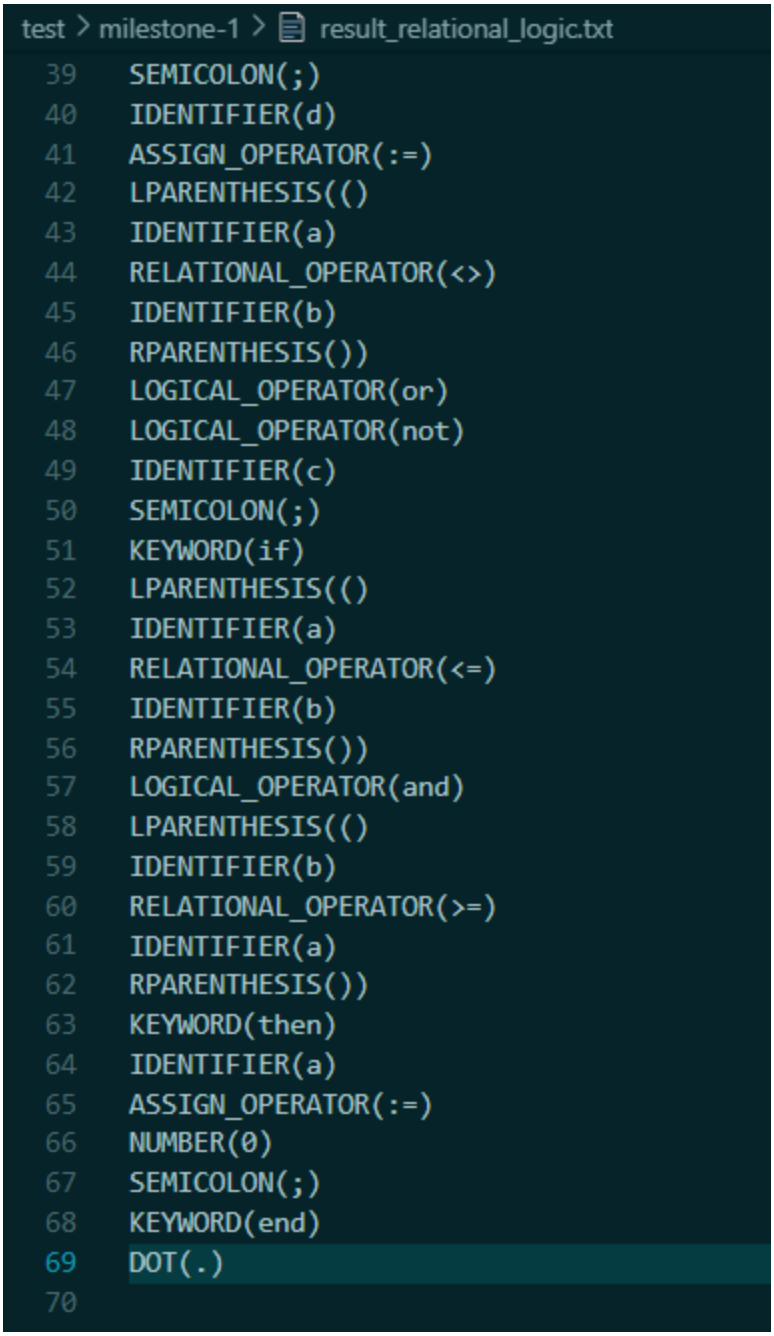
Kedua, pengujian ini menunjukkan keberhasilan dalam membedakan operator aritmetika berbentuk kata dari IDENTIFIER. Lexeme mod berhasil dikenali sebagai ARITHMETIC_OPERATOR dan bukan sebagai IDENTIFIER biasa. Hal ini membuktikan implementasi logika pasca-pemrosesan yang efektif, di mana setelah DFA mengenali sebuah lexeme yang cocok dengan pola identifier, sistem melakukan pengecekan terhadap daftar keyword dan operator khusus (seperti mod, div, and) sebelum finalisasi tipe token. Selain itu, operator relasional multi-karakter seperti <= juga dikenali sebagai satu token tunggal, yang

mengonfirmasi penanganan transisi lookahead pada DFA berjalan sesuai rancangan.

3.2.3 Hasil Uji Operator Relasional & Logika

test > milestone-1 > result_relational_logic.txt

```
KEYWORD(program)
IDENTIFIER(TestOperators)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(a)
COMMA(,)
IDENTIFIER(b)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(c)
COMMA(,)
IDENTIFIER(d)
COLON(:)
KEYWORD(boolean)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(a)
ASSIGN_OPERATOR(:=)
NUMBER(10)
SEMICOLON(;)
IDENTIFIER(b)
ASSIGN_OPERATOR(:=)
NUMBER(20)
SEMICOLON(;)
IDENTIFIER(c)
ASSIGN_OPERATOR(:=)
LPARENTHESIS(()
IDENTIFIER(a)
RELATIONAL_OPERATOR(<)
IDENTIFIER(b)
RPARENTHESIS())
LOGICAL_OPERATOR(and)
LPARENTHESIS(()
IDENTIFIER(b)
RELATIONAL_OPERATOR(>)
IDENTIFIER(a)
RPARENTHESIS())
```

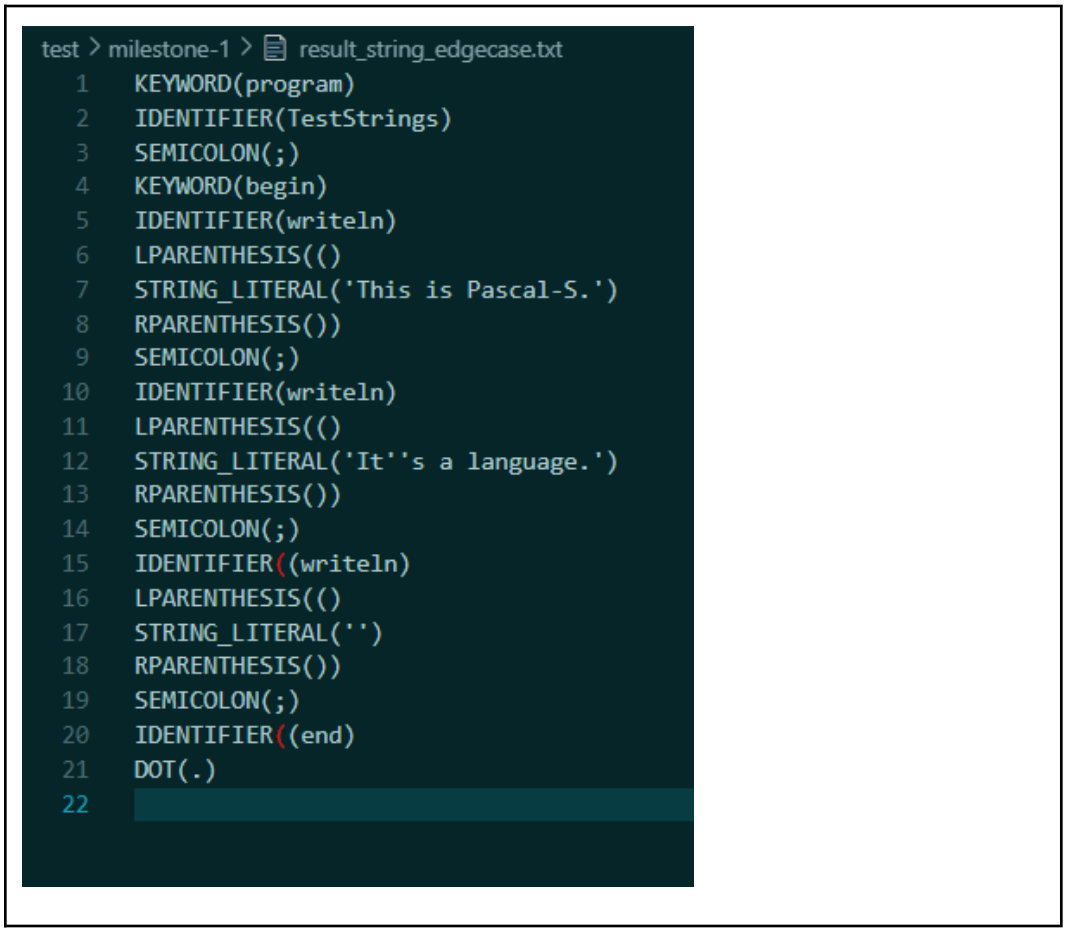


Pembahasan: Pengujian ini secara khusus menargetkan ketangguhan DFA dalam menangani operator yang memerlukan lookahead. Hasilnya sangat memuaskan, karena semua variasi RELATIONAL_OPERATOR berhasil dikenali dengan tepat. Token RELATIONAL_OPERATOR(<> , RELATIONAL_OPERATOR(<=), dan RELATIONAL_OPERATOR(>=) membuktikan bahwa DFA dirancang dengan benar untuk memasuki state antara setelah membaca karakter pertama (misalnya,

<) dan kemudian bertransisi ke state akhir yang sesuai berdasarkan karakter berikutnya (> atau =) atau menyimpulkan token sebagai operator satu karakter jika tidak ada karakter lanjutan yang cocok.

Lebih lanjut, pengujian ini sekali lagi menegaskan keandalan mekanisme klasifikasi ulang untuk operator berbentuk kata. Lexeme and, or, dan not berhasil diidentifikasi sebagai LOGICAL_OPERATOR, bukan IDENTIFIER. Ini menunjukkan bahwa logika pasca-pengenalan pola identifier bekerja secara konsisten untuk semua kata kunci khusus yang telah ditentukan, termasuk operator logika. Keberhasilan dalam pengujian ini krusial, karena pengenalan operator relasional dan logika yang akurat merupakan prasyarat fundamental untuk tahap analisis sintaksis, terutama dalam mem-parsing ekspresi boolean dan struktur kendali program.

3.2.4 Hasil Uji Kasus Khusus String Literal



Pembahasan: Pengujian ini berhasil membuktikan bahwa DFA yang dirancang sangat tangguh dalam mem-parsing STRING_LITERAL. Tiga kasus penting berhasil ditangani dengan benar. Pertama, string literal standar 'This is Pascal-S.' dikenali tanpa masalah. Kedua, dan yang paling krusial, lexer mampu menangani escape sequence untuk karakter kutip tunggal di dalam string. Token STRING_LITERAL('It''s a language.') menunjukkan bahwa DFA dapat memproses dua karakter kutip tunggal ('') secara berurutan sebagai satu karakter kutip literal di dalam string, dan tidak salah mengartikannya sebagai akhir dari string tersebut. Ketiga, lexer juga berhasil mengenali string kosong, '', sebagai token STRING_LITERAL yang valid. Ini menunjukkan bahwa DFA dapat menangani kasus di mana karakter kutip pembuka langsung diikuti oleh karakter kutip penutup. Keberhasilan dalam menangani kasus-kasus edge case ini sangat penting, karena memastikan bahwa lexer dapat memproses semua input string yang valid sesuai aturan leksikal Pascal, sehingga mencegah bug parsing pada tahap selanjutnya.

3.2.5 Hasil Uji Kasus Range Operator

test > milestone-1 > result_range_operator.txt

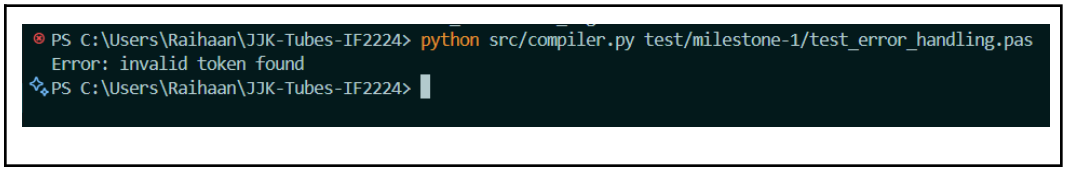
```
KEYWORD(program)
IDENTIFIER(TestArrayRange)
SEMICOLON(;)
KEYWORD(type)
IDENTIFIER(MyArray)
RELATIONAL_OPERATOR(=)
KEYWORD(array)
LBRACKET([)
NUMBER(1)
RANGE_OPERATOR(..)
NUMBER(100)
RBRACKET(])
KEYWORD(of)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(Numbers)
COLON(:)
IDENTIFIER(MyArray)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(Numbers)
LBRACKET([)
NUMBER(1)
RBRACKET(])
ASSIGN_OPERATOR(:=)
NUMBER(10)
SEMICOLON(;)
KEYWORD(end)
DOT(.)
```

Pembahasan: Hasil pengujian ini sangat signifikan karena mengonfirmasi bahwa lexer telah mengimplementasikan token RANGE_OPERATOR sesuai dengan revisi terbaru spesifikasi. DFA berhasil mengenali dua karakter titik (.) yang berurutan sebagai satu token tunggal, yaitu RANGE_OPERATOR(..), dan tidak salah menginterpretasikannya sebagai dua token DOT yang terpisah.

Keberhasilan ini menunjukkan desain DFA yang cermat, yang mampu membedakan antara lexeme . (untuk akhir program atau pada bilangan real) dan lexeme .. (untuk rentang).

Secara teknis, ini membuktikan bahwa DFA memiliki state transisi yang tepat setelah membaca karakter titik pertama. Dari state tersebut, jika karakter berikutnya adalah titik kedua, DFA akan berpindah ke state akhir yang menghasilkan token RANGE_OPERATOR. Namun, jika karakter berikutnya bukan titik, DFA akan menyimpulkan lexeme tersebut sebagai token DOT. Kemampuan untuk menangani token yang memiliki prefiks yang sama ini adalah inti dari analisis leksikal yang andal dan menjadi dasar yang krusial bagi parser untuk dapat memahami sintaksis deklarasi tipe data seperti array dan subrentang pada tahap kompilasi berikutnya.

3.2.6 Hasil Uji Kasus Error Handling



Pembahasan: Hasil pengujian ini menunjukkan bahwa mekanisme error handling pada lexer berfungsi dengan baik. Sesuai dengan spesifikasi, program tidak mengalami crash saat dihadapkan pada input yang tidak valid. Sebaliknya, program berhasil mengidentifikasi karakter (@) yang tidak memiliki transisi valid dalam DFA dari state awal. Saat menemui simbol yang tidak dikenal ini, lexer langsung menghentikan proses tokenisasi dan melaporkan "invalid token found".

Perilaku ini sangat penting karena memastikan integritas proses kompilasi. Dengan berhenti secara terkendali, lexer mencegah token yang salah atau tidak lengkap untuk diteruskan ke tahap parser, yang dapat menyebabkan kesalahan berantai yang lebih sulit untuk dilacak. Kemampuan untuk menangani input yang salah secara elegan menunjukkan bahwa lexer yang dibangun tidak hanya mampu mengenali pola yang benar tetapi juga cukup tangguh untuk menolak pola yang salah, yang merupakan ciri khas dari penganalisis leksikal yang andal.

Bab IV

Penutup

4.1 Kesimpulan

Berdasarkan perancangan, implementasi, dan pengujian yang telah dilakukan, dapat disimpulkan bahwa lexer untuk bahasa Pascal-S telah berhasil dibangun sesuai dengan spesifikasi yang diberikan pada Milestone 1. Lexer yang diimplementasikan dengan model Deterministic Finite Automaton (DFA) mampu melakukan analisis leksikal dengan akurat, memecah kode sumber menjadi unit-unit token yang bermakna. Keberhasilan ini divalidasi melalui serangkaian pengujian yang mencakup berbagai kasus, mulai dari fungsionalitas dasar, pengenalan tipe data integer dan real, penanganan operator multi-karakter dan berbentuk kata, hingga kasus-kasus khusus seperti string literal dan range operator. Selain itu, implementasi error handling yang efektif memastikan program dapat berhenti secara terkendali ketika menghadapi input yang tidak valid. Dengan demikian, lexer ini telah membentuk fondasi yang kokoh untuk tahap kompilasi selanjutnya, yaitu analisis sintaksis.

4.2 Saran

Meskipun lexer yang dibangun telah berfungsi sesuai target, terdapat beberapa area yang dapat dikembangkan lebih lanjut untuk meningkatkan kualitas dan fungsionalitasnya. Pertama, mekanisme pelaporan kesalahan (error handling) dapat dibuat lebih informatif. Saat ini, program hanya menampilkan pesan "invalid token found". Di masa mendatang, pesan ini dapat diperkaya dengan informasi spesifik seperti nomor baris dan kolom tempat karakter tidak valid ditemukan, sehingga sangat membantu proses debugging. Selain itu, file aturan DFA (dfa_rules.json) dapat dioptimalkan dengan menggunakan satu transisi default atau wildcard yang konsisten untuk menangani "karakter lain", yang akan menyederhanakan logika pemrosesan di dalam kode. Sebagai persiapan krusial untuk milestone berikutnya yaitu analisis sintaksis, sangat disarankan untuk mengimplementasikan sebuah kelas TokenManager yang membungkus aliran token. Kelas ini, dengan menyediakan metode esensial seperti peek() untuk melihat token berikutnya dan consume() untuk memajukannya, akan menciptakan jembatan yang bersih dan terstruktur antara lexer dan parser, sehingga mempermudah proses implementasi parser secara signifikan.

Lampiran

Link Github Repository : https://github.com/fliegenhaan/JJK-Tubes-IF2224.git

Link Diagram : https://drive.google.com/file/d/1iRkdA6PxP96ra3_NZBBQHKHXOPRHp2pd/view?usp=drive_link

Pembagian Tugas :

No	Tugas	NIM
1	Implementasi kode program (`compiler.py`, `lexer.py`)	13523142
2	Perancangan dan translasi DFA	13523124
3	Diagram DFA dan dokumentasi laporan	13523136, 13523155

Daftar Pustaka

Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung. Spesifikasi Tugas Besar IF2224 - Formal Languages and Automata Theory, Milestone 1 - Lexical Analysis. Diakses pada 17 Oktober 2025, dari https://docs.google.com/document/d/1w0GmHW5L0gKZQWbgmtJPFmOzlpSWBknNPdugucn4eII/edit?usp=sharing

GeeksforGeeks. "Introduction to Lexical Analysis." GeeksforGeeks. Diakses pada 14 Oktober 2025, dari https://www.geeksforgeeks.org/introduction-of-lexical-analysis/

GeeksforGeeks. "Deterministic Finite Automaton (DFA)." GeeksforGeeks. Diakses pada 15 Oktober 2025, dari https://www.geeksforgeeks.org/introduction-of-finite-automata/

TutorialsPoint. "Compiler Design - Lexical Analysis." TutorialsPoint. Diakses pada 15 Oktober 2025, dari https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm

Free Pascal Documentation. "Pascal Language Reference." Free Pascal Wiki. Diakses pada 15 Oktober 2025, dari https://www.freepascal.org/docs.html

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. Compilers: Principles, Techniques, and Tools (2nd Edition). Pearson Education, 2006.