Laporan Milestone 1
Tugas Besar 1 IF3170 Intelegensi Artifisial
Lexical Analysis

Disusun oleh:
Maggie Zeta Rosida S - 13521117
Buege Mahara Putra - 13523037
Hanif Kalyana Aditya - 13523041
Jethro Jens Norbert Simatupang - 13523081

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
JL. GANESA 10, BANDUNG 40132
2025

BAB 1
LANDASAN TEORI

1.1 Kompilasi dan Analisis Leksikal

Kompilasi adalah proses mengubah kode sumber (source code) yang ditulis dalam bahasa pemrograman tingkat tinggi menjadi kode mesin atau kode objek yang dapat dieksekusi oleh komputer.
Proses ini terbagi menjadi beberapa tahapan utama, yaitu:

1. Analisis Leksikal (Lexical Analysis)
2. Analisis Sintaksis (Syntax Analysis)
3. Analisis Semantik (Semantic Analysis)
4. Optimasi Kode (Code Optimization)
5. Pembangkitan Kode (Code Generation)

Analisis Leksikal (Lexical Analysis) adalah tahap pertama dan fundamental dalam proses kompilasi. Tahap ini berfungsi sebagai "mata" kompilator, yaitu program membaca input, yang pada dasarnya adalah rangkaian karakter mentah, dari kiri ke kanan. Tujuannya adalah untuk mengelompokkan karakter-karakter tersebut menjadi unit-unit bermakna yang dikenal sebagai token. Analisis leksikal juga bertanggung jawab untuk membuang karakter yang tidak relevan, seperti spasi, tab, baris baru, dan komentar.

1.2 Token dan *Regular Expression*

Token adalah unit terkecil yang memiliki arti logis dalam sebuah bahasa pemrograman. Setiap token biasanya terdiri dari dua komponen utama:

1. Tipe (Type): Kategori token (misalnya, KEYWORD, IDENTIFIER, NUMBER, SEMICOLON).
2. Nilai (Value) / Leksem (Lexeme): Urutan karakter aktual dalam kode sumber yang membentuk token tersebut (misalnya, untuk tipe KEYWORD, leksemnya bisa berupa program, begin, atau end).

Dalam konteks tugas ini, token yang harus dikenali meliputi kata kunci Pascal-S, operator (aritmatika, relasional, penugasan), identifier, literal (angka, karakter, string), serta simbol tanda baca (“;”, “,”, “:”) .
Regular Expression (Regex) adalah notasi formal yang digunakan untuk mendefinisikan pola dari urutan karakter yang membentuk setiap jenis token. Setiap token dalam sebuah bahasa dapat diwakili oleh sebuah ekspresi regular. Misalnya, pola untuk *identifier* umumnya didefinisikan sebagai huruf diikuti oleh nol atau lebih huruf atau angka. Dengan regex, lexer dapat mencocokkan pola karakter yang terbaca dengan definisi token yang sesuai.

1.3 *Deterministic Finite Automata* (DFA)

Proses pengenalan token dalam analisis leksikal diimplementasikan menggunakan model DFA. DFA adalah mesin abstrak yang dapat menerima atau menolak rangkaian *string* input berdasarkan serangkaian aturan transisi yang terbatas. DFA memiliki

karakteristik deterministik, artinya untuk setiap pasangan *state* dan *input*, hanya ada satu *state* berikutnya yang mungkin.

Secara formal, DFA M didefinisikan sebagai tupel 5, $M = (Q, \Sigma, \delta, q_0, F)$, dengan:

- Q: Himpunan semua *state*
- Σ: Himpunan alfabet input
- δ: Fungsi transisi, $\delta: Q \times \Sigma \rightarrow Q$
- q_0: State awal (*initial state*)
- F: Himpunan state akhir (*final state*) atau state penerima

Dalam implementasi lexer, DFA digunakan untuk memvalidasi apakah urutan karakter yang dibaca dari kode sumber sesuai dengan pola (ekspresi regular) dari salah satu tipe token. *State* akhir (F) dalam DFA akan diasosiasikan dengan tipe token tertentu. Saat proses *scanning* karakter demi karakter, lexer mengikuti transisi DFA. Ketika urutan karakter selesai dan DFA berada di *state* akhir, token yang bersangkutan dikenali dan dikeluarkan sebagai *output*.

1.4 Bahasa Pascal-S

Pascal-S adalah subset sederhana dari bahasa pemrograman Pascal, yang sering digunakan untuk tujuan pembelajaran kompilasi dan teori bahasa formal. Bahasa ini memiliki seperangkat *keyword*, struktur kontrol, dan tipe data yang disederhanakan. Lexer yang dikembangkan harus mampu mengenali secara tepat semua *keyword*, *identifier*, *operator*, dan *literal* sesuai dengan spesifikasi token yang diberikan dalam daftar acuan.

Dalam Milestone 1 Tugas Besar TBFO, lexer bertugas untuk mengenali token-token yang valid pada bahasa Pascal-S sesuai dengan spesifikasi yang diberikan. Token-token tersebut nantinya akan menjadi input bagi tahap berikutnya, yaitu syntax analysis pada Milestone 2.

BAB 2
PERANCANGAN & IMPLEMENTASI

2.1 Perancangan Sistem

Pada Milestone 1 ini, dibuat sebuah diagram *Deterministic Finite Automata* yang dirancang untuk merepresentasikan seluruh proses *lexical analysis* dalam bahasa Pascal-S. Diagram yang dibuat ini dimulai dari *start state* S0 dan bercabang menuju beberapa lintasan utama yang masing-masing menangani satu tipe/kategori token tertentu. Setiap lintasan tersebut memiliki *accepting state* atau *final state* yang menunjukkan akhir dari pengenalan sebuah token. Setelah mencapai *final state*, lexer akan menghasilkan token dan kembali ke S0 untuk memproses token berikutnya.

2.1.1 Diagram DFA

DFA yang dirancang mencakup seluruh tipe token yang digunakan dalam bahasa Pascal-S antara lain :

- *Identifier* dan *Keyword*
- Bilangan (integer dan real)
- String literal
- Operator relasional dan *assignment* (`=`, `<>`, `<`, `<=`, `>`, `>=`, `:=`)
- Operator aritmatika (`+`, `-`, `*`, `/`)
- Tanda baca (`;`, `,`, `[`, `]`, `(`,`)`,`{`,`}`)
- Komentar (`(* . . . *)` dan `{ . . . }`)
- Spasi dan karakter tak dikenal

Setiap tipe token tersebut melewati satu atau beberapa lintasan dalam DFA dari *start state* hingga *accepting state* atau *final state* yang berbeda-beda. Dalam perancangan DFA ini, beberapa karakter dikelompokkan agar memudahkan dalam penulisan atau pembuatan DFA tersebut. Misalnya, `[WS]` merujuk pada input berupa *whitespace* seperti ‘ ’, `\t`, `\n`, `\r`. Selain itu, input lain seperti `[OTHER]` merepresentasikan karakter lain yang tidak termasuk kelompok manapun misalnya `@` dan `?`. Sementara, khusus untuk `[OTHER_NOT_LETTER_DIGIT]` pada ‘lintasan’ *identifier* ditujukan untuk sebuah masukan yang tidak termasuk baik dalam kelompok `[LETTER]` seperti `A-Z`, `a-z` maupun `[DIGIT]` seperti `0-9`.

2.1.2 Lintasan/Transisi Utama

2.1.2.1 *Identifier* dan *Keyword*

Dari *start state* S0, jika karakter pertama termasuk dalam kelas `[LETTER]`, automata berpindah ke state S_ID. *State* ini terus mempertahankan dirinya (S_ID → S_ID) selama karakter berikutnya masih berupa `[LETTER]` atau `[DIGIT]` yang membentuk rangkaian

huruf dan angka. Setelah ditemukan karakter yang bukan huruf atau angka (`[OTHER_NOT_LETTER_DIGIT]`), automata berhenti dan menerima token *identifier*. Token ini kemudian diklasifikasikan lebih lanjut menjadi *keyword*, *operator logika*, atau *identifier* biasa.

2.1.2.2 Bilangan (Integer dan Real)

Jika dari S0 dibaca karakter pertama berupa `[DIGIT]`, DFA masuk ke `S_NUM_INT`. State ini menerima deretan digit dan akan tetap di sana selama karakter berikutnya juga digit. Apabila ditemukan tanda titik (`.`), automata berpindah ke `S_NUM_DOT_LOOK` untuk menentukan apakah itu bagian dari angka real atau operator *range* (`..`).

- Jika setelah titik ditemukan digit lagi, DFA berpindah ke `S_NUM_REAL` dan terus membaca digit sebagai bagian bilangan real.
- Jika setelah titik ternyata ada titik lagi (..), DFA akan menutup token integer dan karakter kedua titik dikembalikan untuk ditangani oleh lintasan operator *range* (*rejected*).
- Jika setelah titik muncul karakter lain, DFA menerima bilangan integer, bukan real.

Dengan demikian, jalur ini mampu membedakan bilangan 123, 45.67, maupun urutan 1..10 dengan benar.

2.1.2.3 Literal String dan Karakter

Ketika automata membaca tanda kutip tunggal (`'`) dari S0, proses akan berpindah ke *state* `S_QUOTE`. Di dalam state ini, semua karakter selain `'` dan newline dianggap bagian dari literal *string*.

- Apabila ditemukan `'` lagi, DFA berpindah ke `S_QUOTE_SEEN` yang berarti tanda kutip penutup telah ditemukan.
- Namun jika setelahnya langsung muncul ' lagi ('), maka dianggap sebagai *escaped quote*, dan DFA kembali ke `S_QUOTE`.

Jika setelah `S_QUOTE_SEEN` muncul karakter lain, maka literal dianggap selesai dan automata berpindah ke *final state* untuk token string. Sementara itu, jika file berakhir sebelum tanda penutup ditemukan, DFA berpindah ke *state* error (`S_UNKNOWN`), menandakan literal tidak ditutup.

2.1.2.4 Operator Relasional dan *Assignment*

Untuk mengenali operator seperti `=`, `<>`, `<`, `<=`, `>`, `>=`, dan `:=` automata memiliki beberapa lintasan khusus:

a. Membaca `:` dari S0 akan menuju `S_COLON`. Jika setelahnya `=` terbaca, maka token `:=` (*assign operator*) diterima. Jika tidak, hanya token *colon* (`:`) yang dihasilkan.

b. Membaca `<` menuju `S_LT`. Jika setelahnya `=` atau `>` ditemukan, DFA menerima token `<=` atau `<>` (operator relasional dua karakter). Jika tidak, `<` tunggal diterima.

c. Membaca `>` menuju `S_GT`, dengan logika serupa (`>=` atau `>`).

d. Membaca `=` langsung menghasilkan token relasional `=`.

2.1.2.5 Operator Aritmatika dan Punctuation

Dari S0, setelah DFA membaca salah satu dari (`+`, `-`, `*`, `/`), DFA langsung menghasilkan token operator aritmatika karena masing-masing bersifat *single-character* token. Sementara karakter seperti (`;`, `,`, `[`, `]`) diterima langsung sebagai *punctuation tokens* dengan kategori yang sesuai.

2.1.2.6 Komentar

a. *Brace comment* `{ ... }`
Ketika DFA membaca `{` dari S0, *state* akan berpindah ke `S_BRACE_COMMENT`. DFA tetap di *state* ini selama belum menemukan `}` dan baru keluar setelah simbol penutup tersebut ditemukan. Apabila input berakhir sebelum `}` muncul, *state* berpindah ke `S_UNKNOWN` yang menandai error.

b. *Parenthesis-star comment* `(* ... *)`
Ketika DFA membaca `(` dari S0 dan diikuti dengan `*`, *state* akan berpindah ke `S_PARENSTAR_COMMENT`. Di dalam *state* ini, apabila DFA membaca simbol `*` yang diikuti dengan `)` akan menyebabkan DFA keluar dari komentar.

Kedua jenis komentar tidak menghasilkan token apa pun dan hanya berfungsi untuk *skip* input.

2.1.2.7 Whitespace dan Karakter Tidak Dikenal

Dalam prosesnya, masukan berupa karakter *whitespace* seperti ‘ ’, `\t`, `\n`, `\r` yang direpresentasikan dengan `[WS]` akan membawa DFA ke *state* `S_WS`. Hal ini berarti seluruh karakter *whitespace* akan

dilewati hingga karakter bukan whitespace terbaca. Whitespace sebenarnya tidak menghasilkan token dan hanya berfungsi sebagai pemisah antar token. Sementara itu, karakter lain yang tidak masuk kategori manapun akan mengarahkan DFA ke `S_UNKNOWN` sebagai pesan kesalahan leksikal.

2.2 Implementasi Program

Struktur Folder terdiri dari:

1. src/ → berisi implementasi Python
2. rules/ → berisi spesifikasi token dan DFA
3. test → input/output hasil pengujian
4. doc/ → laporan dan diagram

2.2.1 Token

Bagian define token bertujuan untuk mendefinisikan seluruh token yang dapat dikenali Pascal-S Lexer. Perancangan dilakukan menggunakan bahasa Python 3.12.

File utama:

1. rules/token_spec.json → spesifikasi formal semua token

   File token_spec.json dirancang supaya bisa dibaca langsung oleh lexer tanpa perlu hard-coding di dalam program. Menggunakan format JSON karena mudah dibaca manusia dan mesin.

```
{
  "version": "1.0",
  "case_insensitive": true,
  "types": [...],
  "keywords": [...],
  "logical_operators": [...],
  "arithmetic_operators_symbol": [...],
  "relational_operators_symbol": [...],
  "assign_operator": [":="],
  "patterns": {...},
  "classification_rules": [...]
}
```

   - `"case_insensitive": true` -> untuk memastikan lexer mengenali token Pascal-S tanpa membedakan huruf besar/kecil.
   - `"patterns"` -> menggunakan Regular Expression untuk mendefinisikan token dinamis seperti IDENTIFIER, NUMBER, CHAR_LITERAL, dan STRING_LITERAL.
   - `"classification_rules"` -> menentukan urutan prioritas pencocokan, agar lexer mengenali token yang lebih spesifik terlebih dahulu (misalnya := sebelum :).
   - `"output_format": "TOKEN_TYPE(value)"` -> menentukan format keluaran token agar seragam.

2. src/tokens.py -> kelas dan enum untuk representasi token
   File tokens.py mendefinisikan struktur token dan fungsi.
   a. Kelas dan enum
      Enum menjamin tiap tipe token unik

```
from enum import Enum, auto
from dataclasses import dataclass

class TokenType(Enum):
    KEYWORD = auto()
    LOGICAL_OPERATOR = auto()
    IDENTIFIER = auto()
    NUMBER = auto()
    ...
```

   b. Kelas token
      Digunakan untuk menyimpan hasil identifikasi token dan posisi munculnya.

```
@dataclass
class Token:
    type: TokenType
    value: str
    line: int
    column: int
```

   c. Fungsi classify
      Fungsi modular untuk mempermudah integrasi dengan DFA parser.

```
def    classify_word(lexeme:    str)    ->
TokenType:
    low = lexeme.lower()
          if  low  in  KEYWORDS:  return
TokenType.KEYWORD
         if  low  in  LOGICAL_OPS:  return
TokenType.LOGICAL_OPERATOR
    ...
```

2.2.2 Lexer

Bagian ini bertujuan untuk melakukan proses analisis leksikal (lexical analysis) terhadap kode sumber Pascal-S. Lexer bertugas membaca setiap karakter dari file .pas, mengenali pola token berdasarkan aturan yang sudah ditentukan dalam token_spec.json, lalu menghasilkan daftar token terstruktur.

File: src/lexer.py

File lexer.py berisi fungsi utama tokenize() yang mengubah *source code* menjadi deretan token.

Proses lexical analysis dibagi ke beberapa tahap:

a. Mengabaikan whitespace dan komentar
   Semua spasi, tab, newline, dan komentar di-skip karena tidak berpengaruh pada struktur sintaks.

```
if ch.isspace():
    if ch == "\n":
        ...
if ch == "{" and "}" in source_code[i:]:
    end_idx = source_code.find("}", i + 1)
    if end_idx == -1:
        ...
if source_code.startswith("(*", i):
    end_idx = source_code.find("*)", i + 2)
    if end_idx == -1:
        ...
```

b. Mengenali token multi-karakter

Token seperti `..`, `:=`, `<=`, `>=`, `<>` diperiksa lebih dulu agar tidak salah terbaca sebagai dua token terpisah (`:`, `=`).

```
for sym in LONGEST_FIRST:
    if source_code.startswith(sym, i):
        matched = sym
        break
```

c. Mengenali string

Token string diapit tanda `'...'` dan dapat berisi karakter apa pun kecuali newline.

```
if ch == "'":
    end_idx = i + 1
            while   end_idx   <   length   and
source_code[end_idx] != "'":
        if source_code[end_idx] == "\n":
            line += 1
            col = 1
        end_idx += 1
    ...
```

d. Mengenali keyword dan identifier

Jika awalan berupa huruf, lexer akan membaca hingga karakter bukan alfanumerik. Lalu, dicek apakah termasuk keyword, operator kata (and, or, mod, div), atau identifier biasa

```
if ch.isalpha():
    start = i
    while i < length and (source_code[i].isalnum()
or source_code[i] == "_"):
        i += 1
    lexeme = source_code[start:i]
                                token_type        =
classify_word_or_operator_word(lexeme)
     tokens.append(Token(token_type, lexeme, line,
col))
    col += len(lexeme)
    continue
```

e. Mengenali angka

Lexer mengenali angka dengan atau tanpa titik desimal.

```
if ch.isdigit():
    start = i
    has_dot = False
    while i < length and (source_code[i].isdigit()
or source_code[i] == "."):
        if source_code[i] == ".":
            if has_dot:
                break
            has_dot = True
        i += 1
    lexeme = source_code[start:i]
     tokens.append(Token(TokenType.NUMBER, lexeme,
line, col))
    col += len(lexeme)
    continue
```

f. Mengenali operator dan tanda baca tunggal

Simbol seperti ;, :, ,, +, -, (,) dikenali lewat pemetaan PUNCTUATION dan ARITH_SYMBOL di tokens.py.

```
token_type = classify_punct_or_ops(ch)
if token_type:
        tokens.append(Token(token_type, ch, line,
col))
    i += 1
    col += 1
    continue
```

g. Mendeteksi token yang tidak dikenal

Jika karakter tidak cocok dengan pola mana pun, token diberi tipe UNKNOWN dan ditampilkan peringatan

```
tokens.append(Token(TokenType.UNKNOWN, ch, line,
col))
print(f"Unknown token '{ch}' at line {line},
column {col}")
i += 1
col += 1
```

2.2.3 Reader

File: src/reader.py

File ini bertanggung jawab untuk membaca dan menampilkan hasil tokenisasi. Kelas Reader menerima list token dari lexer, lalu menampilkan hasilnya dalam format TOKEN_TYPE(value)

```
from src.tokens import Token

class Reader:
    def __init__(self, tokens: list[Token]):
        self.tokens = tokens
        self.pos = 0
```

```
    def read(self):
        for tok in self.tokens:
            print(f"{tok.type.name}({tok.value})")
```

2.2.4 Compiler

Pada milestone ini, bagian compiler hanya dapat menjalankan proses analisis leksikal.

File: src/compiler.py

```
import sys
from pathlib import Path
from src.lexer import tokenize
from src.reader import Reader

def main():
    if len(sys.argv) != 2:
                    print("Usage:  python  compiler.py
[program.pas]")
        sys.exit(1)

    pascal_file = sys.argv[1]
    if not Path(pascal_file).exists():
              print(f"Error:  file  '{pascal_file}'  not
found")
        sys.exit(1)

     with open(pascal_file, "r", encoding="utf-8") as
f:
        code = f.read()

    tokens = tokenize(code)
    reader = Reader(tokens)
    reader.read()

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

Program dimulai dengan fungsi main() yang membaca argumen file .pas, memanggil lexer, lalu menampilkan hasil tokenisasi menggunakan Reader.

BAB 3
PENGUJIAN

3.1 Kasus Uji 1

Input 1:

```
program HelloWorld;
begin
  writeln('Hello, world!');
end.
```

Output 1:

```
KEYWORD(program)
IDENTIFIER(HelloWorld)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Hello, world!')
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(end)
DOT(.)
```

Screenshot 1:

3.2 Kasus Uji 2

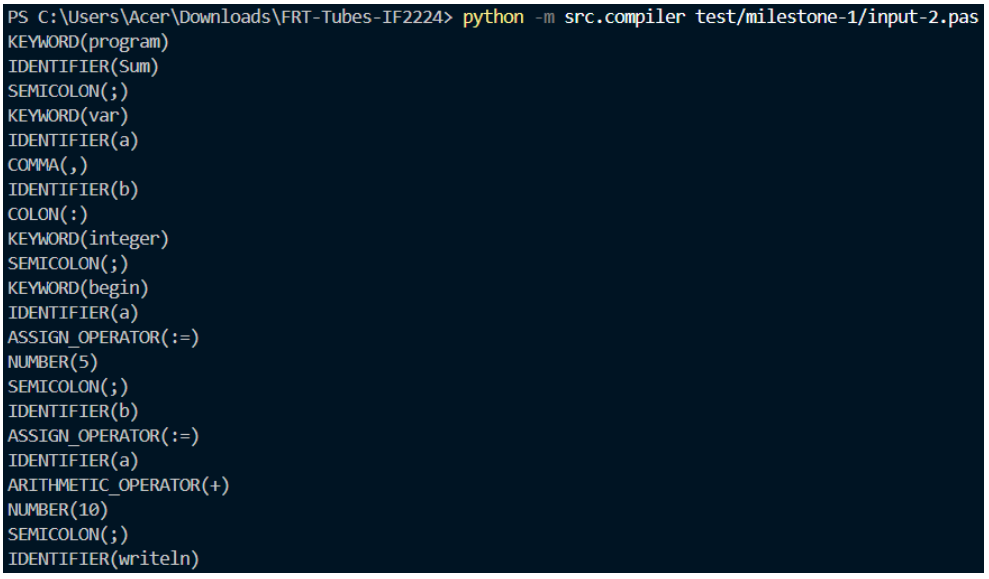
Input 2:

```
program Sum;

var
  a, b: integer;

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

Output 2:

```
KEYWORD(program)
IDENTIFIER(Sum)
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(.)
```

Screenshot 2:



```
LPARENTHESIS(()
STRING_LITERAL('Result = ')
COMMA(,)
IDENTIFIER(b)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(end)
DOT(.)
```

3.3 Kasus Uji 3

Input 3:

```
program EvenOdd;

var
  i: integer;

begin
  i := 0;
  while i < 5 do
  begin
    if i mod 2 = 0 then
      writeln(i, ' is even')
    else
      writeln(i, ' is odd');
    i := i + 1;
  end;
end.
```

Output 3:

```
KEYWORD(program)
IDENTIFIER(EvenOdd)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(i)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
NUMBER(0)
SEMICOLON(;)
KEYWORD(while)
IDENTIFIER(i)
RELATIONAL_OPERATOR(<)
NUMBER(5)
KEYWORD(do)
KEYWORD(begin)
KEYWORD(if)
IDENTIFIER(i)
ARITHMETIC_OPERATOR(mod)
```

```
NUMBER(2)
RELATIONAL_OPERATOR(=)
NUMBER(0)
KEYWORD(then)
IDENTIFIER(writeln)
LPARENTHESIS(()
IDENTIFIER(i)
COMMA(,)
STRING_LITERAL(' is even')
RPARENTHESIS())
KEYWORD(else)
IDENTIFIER(writeln)
LPARENTHESIS(()
IDENTIFIER(i)
COMMA(,)
STRING_LITERAL(' is odd')
RPARENTHESIS())
SEMICOLON(;)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
IDENTIFIER(i)
ARITHMETIC_OPERATOR(+)
NUMBER(1)
SEMICOLON(;)
KEYWORD(end)
SEMICOLON(;)
KEYWORD(end)
DOT(.)
```

Screenshot 3:

```
PS C:\Users\Acer\Downloads\FRT-Tubes-IF2224> python -m src.compiler test/milestone-1/input-3.pas
KEYWORD(program)
IDENTIFIER(EvenOdd)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(i)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
NUMBER(0)
SEMICOLON(;)
KEYWORD(while)
IDENTIFIER(i)
RELATIONAL_OPERATOR(<)
NUMBER(5)
KEYWORD(do)
KEYWORD(begin)
KEYWORD(if)
IDENTIFIER(i)
ARITHMETIC_OPERATOR(mod)
NUMBER(2)
RELATIONAL_OPERATOR(=)
NUMBER(0)
KEYWORD(then)
```

```
IDENTIFIER(writeln)
LPARENTHESIS(()
IDENTIFIER(i)
COMMA(,)
STRING_LITERAL(' is even')
RPARENTHESIS())
KEYWORD(else)
IDENTIFIER(writeln)
LPARENTHESIS(()
IDENTIFIER(i)
COMMA(,)
STRING_LITERAL(' is odd')
RPARENTHESIS())
SEMICOLON(;)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
IDENTIFIER(i)
ARITHMETIC_OPERATOR(+)
NUMBER(1)
SEMICOLON(;)
KEYWORD(end)
SEMICOLON(;)
KEYWORD(end)
DOT(.)
```

3.4 Kasus Uji 4

Input 4:

```
program ArrayAndFunction;

var
  arr: array[1..5] of integer;
  i: integer;

function SumArray(a: array of integer; n:
integer): integer;

var
  s, j: integer;

begin
  s := 0;
  for j := 0 to n - 1 do
    s := s + a[j];
  SumArray := s;
end;

begin
  for i := 1 to 5 do
    arr[i] := i * 2;
  writeln('Total sum: ', SumArray(arr, 5));
end.
```

Output 4:

```
KEYWORD(program)
IDENTIFIER(ArrayAndFunction)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(arr)
COLON(:)
KEYWORD(array)
LBRACKET([)
NUMBER(1.)
DOT(.)
NUMBER(5)
RBRACKET(])
KEYWORD(of)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(i)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(function)
IDENTIFIER(SumArray)
LPARENTHESIS(()
IDENTIFIER(a)
COLON(:)
KEYWORD(array)
KEYWORD(of)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(n)
COLON(:)
KEYWORD(integer)
RPARENTHESIS())
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(s)
COMMA(,)
IDENTIFIER(j)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(s)
ASSIGN_OPERATOR(:=)
NUMBER(0)
SEMICOLON(;)
KEYWORD(for)
IDENTIFIER(j)
ASSIGN_OPERATOR(:=)
```

```
NUMBER(0)
KEYWORD(to)
IDENTIFIER(n)
ARITHMETIC_OPERATOR(-)
NUMBER(1)
KEYWORD(do)
IDENTIFIER(s)
ASSIGN_OPERATOR(:=)
IDENTIFIER(s)
ARITHMETIC_OPERATOR(+)
IDENTIFIER(a)
LBRACKET([)
IDENTIFIER(j)
RBRACKET(])
SEMICOLON(;)
IDENTIFIER(SumArray)
ASSIGN_OPERATOR(:=)
IDENTIFIER(s)
SEMICOLON(;)
KEYWORD(end)
SEMICOLON(;)
KEYWORD(begin)
KEYWORD(for)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
NUMBER(1)
KEYWORD(to)
NUMBER(5)
KEYWORD(do)
IDENTIFIER(arr)
LBRACKET([)
IDENTIFIER(i)
RBRACKET(])
ASSIGN_OPERATOR(:=)
IDENTIFIER(i)
ARITHMETIC_OPERATOR(*)
NUMBER(2)
SEMICOLON(;)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Total sum: ')
COMMA(,)
IDENTIFIER(SumArray)
LPARENTHESIS(()
IDENTIFIER(arr)
COMMA(,)
NUMBER(5)
RPARENTHESIS())
RPARENTHESIS())
SEMICOLON(;)
```

```
KEYWORD(end)
DOT(.)
```

Screenshot 4:

```
PS C:\Users\Acer\Downloads\FRT-Tubes-IF2224> python -m src.compiler test/milestone-1/input-4.pas
KEYWORD(program)
IDENTIFIER(ArrayAndFunction)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(arr)
COLON(:)
KEYWORD(array)
LBRACKET([)
NUMBER(1.)
DOT(.)
NUMBER(5)
RBRACKET(])
KEYWORD(of)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(i)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(function)
IDENTIFIER(SumArray)
LPARENTHESIS(()
IDENTIFIER(a)
COLON(:)
KEYWORD(array)
KEYWORD(of)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(n)
COLON(:)
KEYWORD(integer)
RPARENTHESIS())
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(s)
COMMA(,)
IDENTIFIER(j)
COLON(:)
```

```
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(s)
ASSIGN_OPERATOR(:=)
NUMBER(0)
SEMICOLON(;)
KEYWORD(for)
IDENTIFIER(j)
ASSIGN_OPERATOR(:=)
NUMBER(0)
KEYWORD(to)
IDENTIFIER(n)
ARITHMETIC_OPERATOR(-)
NUMBER(1)
KEYWORD(do)
IDENTIFIER(s)
ASSIGN_OPERATOR(:=)
IDENTIFIER(s)
ARITHMETIC_OPERATOR(+)
IDENTIFIER(a)
LBRACKET([)
IDENTIFIER(j)
RBRACKET(])
```

```
SEMICOLON(;)
IDENTIFIER(SumArray)
ASSIGN_OPERATOR(:=)
IDENTIFIER(s)
SEMICOLON(;)
KEYWORD(end)
SEMICOLON(;)
KEYWORD(begin)
KEYWORD(for)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
NUMBER(1)
KEYWORD(to)
NUMBER(5)
KEYWORD(do)
IDENTIFIER(arr)
LBRACKET([)
IDENTIFIER(i)
RBRACKET(])
ASSIGN_OPERATOR(:=)
IDENTIFIER(i)
ARITHMETIC_OPERATOR(*)
NUMBER(2)
SEMICOLON(;)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Total sum: ')
COMMA(,)
IDENTIFIER(SumArray)
LPARENTHESIS(()
IDENTIFIER(arr)
COMMA(,)
NUMBER(5)
RPARENTHESIS())
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(end)
DOT(.)
```

3.5 Kasus Uji 5

Input 5:

```
program BubbleSort;

var
  arr: array[1..5] of integer;
  i, j, temp: integer;

begin
  arr[1] := 5;
  arr[2] := 2;
  arr[3] := 8;
  arr[4] := 1;
  arr[5] := 3;

  for i := 1 to 4 do
  begin
    for j := 1 to 5 - i do
    begin
```

```
      if arr[j] > arr[j + 1] then
      begin
        temp := arr[j];
        arr[j] := arr[j + 1];
        arr[j + 1] := temp;
      end;
    end;
  end;

  writeln('Sorted array:');
  for i := 1 to 5 do
    writeln(arr[i]);
end.
```

Output 5:

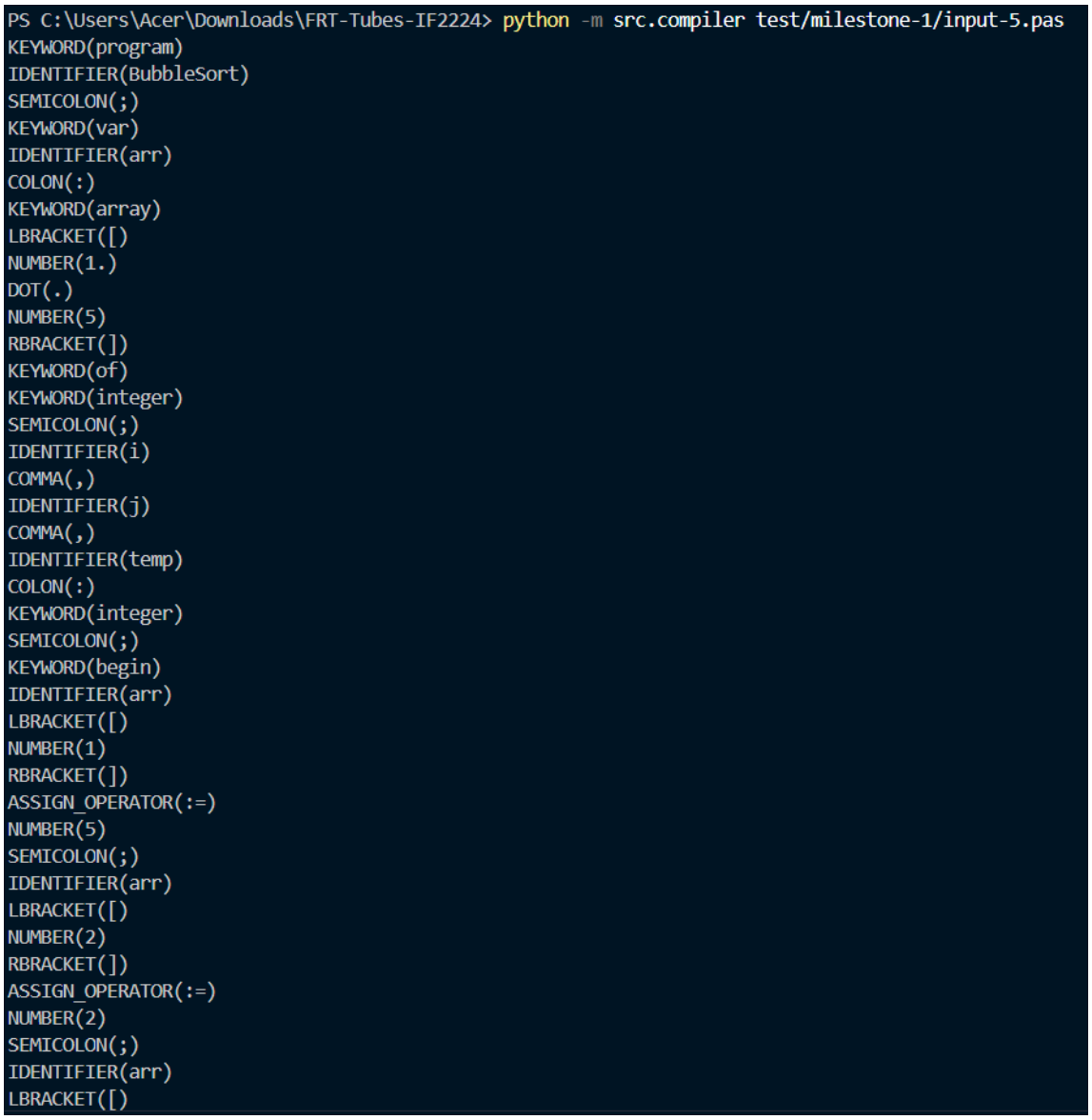
```
KEYWORD(program)
IDENTIFIER(BubbleSort)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(arr)
COLON(:)
KEYWORD(array)
LBRACKET([)
NUMBER(1.)
DOT(.)
NUMBER(5)
RBRACKET(])
KEYWORD(of)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(i)
COMMA(,)
IDENTIFIER(j)
COMMA(,)
IDENTIFIER(temp)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(arr)
LBRACKET([)
NUMBER(1)
RBRACKET(])
ASSIGN_OPERATOR(:=)
NUMBER(5)
SEMICOLON(;)
IDENTIFIER(arr)
LBRACKET([)
NUMBER(2)
```

```
RBRACKET(])
ASSIGN_OPERATOR(:=)
NUMBER(2)
SEMICOLON(;)
IDENTIFIER(arr)
LBRACKET([)
NUMBER(3)
RBRACKET(])
ASSIGN_OPERATOR(:=)
NUMBER(8)
SEMICOLON(;)
IDENTIFIER(arr)
LBRACKET([)
NUMBER(4)
RBRACKET(])
ASSIGN_OPERATOR(:=)
NUMBER(1)
SEMICOLON(;)
IDENTIFIER(arr)
LBRACKET([)
NUMBER(5)
RBRACKET(])
ASSIGN_OPERATOR(:=)
NUMBER(3)
SEMICOLON(;)
KEYWORD(for)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
NUMBER(1)
KEYWORD(to)
NUMBER(4)
KEYWORD(do)
KEYWORD(begin)
KEYWORD(for)
IDENTIFIER(j)
ASSIGN_OPERATOR(:=)
NUMBER(1)
KEYWORD(to)
NUMBER(5)
ARITHMETIC_OPERATOR(-)
IDENTIFIER(i)
KEYWORD(do)
KEYWORD(begin)
KEYWORD(if)
IDENTIFIER(arr)
LBRACKET([)
IDENTIFIER(j)
RBRACKET(])
RELATIONAL_OPERATOR(>)
IDENTIFIER(arr)
```

```
LBRACKET([)
IDENTIFIER(j)
ARITHMETIC_OPERATOR(+)
NUMBER(1)
RBRACKET(])
KEYWORD(then)
KEYWORD(begin)
IDENTIFIER(temp)
ASSIGN_OPERATOR(:=)
IDENTIFIER(arr)
LBRACKET([)
IDENTIFIER(j)
RBRACKET(])
SEMICOLON(;)
IDENTIFIER(arr)
LBRACKET([)
IDENTIFIER(j)
RBRACKET(])
ASSIGN_OPERATOR(:=)
IDENTIFIER(arr)
LBRACKET([)
IDENTIFIER(j)
ARITHMETIC_OPERATOR(+)
NUMBER(1)
RBRACKET(])
SEMICOLON(;)
IDENTIFIER(arr)
LBRACKET([)
IDENTIFIER(j)
ARITHMETIC_OPERATOR(+)
NUMBER(1)
RBRACKET(])
ASSIGN_OPERATOR(:=)
IDENTIFIER(temp)
SEMICOLON(;)
KEYWORD(end)
SEMICOLON(;)
KEYWORD(end)
SEMICOLON(;)
KEYWORD(end)
SEMICOLON(;)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Sorted array:')
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(for)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
NUMBER(1)
```

```
KEYWORD(to)
NUMBER(5)
KEYWORD(do)
IDENTIFIER(writeln)
LPARENTHESIS(()
IDENTIFIER(arr)
LBRACKET([)
IDENTIFIER(i)
RBRACKET(])
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(end)
DOT(.)
```

Screenshot 5:



```
NUMBER(3)
RBRACKET(])
ASSIGN_OPERATOR(:=)
NUMBER(8)
SEMICOLON(;)
IDENTIFIER(arr)
LBRACKET([)
NUMBER(4)
RBRACKET(])
ASSIGN_OPERATOR(:=)
NUMBER(1)
SEMICOLON(;)
IDENTIFIER(arr)
LBRACKET([)
NUMBER(5)
RBRACKET(])
ASSIGN_OPERATOR(:=)
NUMBER(3)
SEMICOLON(;)
KEYWORD(for)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
NUMBER(1)
KEYWORD(to)
NUMBER(4)
KEYWORD(do)
KEYWORD(begin)
KEYWORD(for)
IDENTIFIER(j)
ASSIGN_OPERATOR(:=)
NUMBER(1)
KEYWORD(to)
NUMBER(5)
ARITHMETIC_OPERATOR(-)
IDENTIFIER(i)
KEYWORD(do)
KEYWORD(begin)
KEYWORD(if)
IDENTIFIER(arr)
LBRACKET([)
```

```
IDENTIFIER(j)
RBRACKET(])
RELATIONAL_OPERATOR(>)
IDENTIFIER(arr)
LBRACKET([)
IDENTIFIER(j)
ARITHMETIC_OPERATOR(+)
NUMBER(1)
RBRACKET(])
KEYWORD(then)
KEYWORD(begin)
IDENTIFIER(temp)
ASSIGN_OPERATOR(:=)
IDENTIFIER(arr)
LBRACKET([)
IDENTIFIER(j)
RBRACKET(])
SEMICOLON(;)
IDENTIFIER(arr)
LBRACKET([)
```

```
IDENTIFIER(j)
RBRACKET(])
ASSIGN_OPERATOR(:=)
IDENTIFIER(arr)
LBRACKET([)
IDENTIFIER(j)
ARITHMETIC_OPERATOR(+)
NUMBER(1)
RBRACKET(])
SEMICOLON(;)
IDENTIFIER(arr)
LBRACKET([)
IDENTIFIER(j)
ARITHMETIC_OPERATOR(+)
NUMBER(1)
RBRACKET(])
ASSIGN_OPERATOR(:=)
IDENTIFIER(temp)
SEMICOLON(;)
KEYWORD(end)
SEMICOLON(;)
KEYWORD(end)
SEMICOLON(;)
KEYWORD(end)
SEMICOLON(;)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Sorted array:')
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(for)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
NUMBER(1)
KEYWORD(to)
NUMBER(5)
KEYWORD(do)
IDENTIFIER(writeln)
LPARENTHESIS(()
```

```
IDENTIFIER(arr)
LBRACKET([)
IDENTIFIER(i)
RBRACKET(])
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(end)
DOT(.)
```

BAB 4
KESIMPULAN DAN SARAN

4.1 Kesimpulan

- Lexical Analysis (lexer) merupakan tahap awal proses kompilasi yang mengubah rangkaian karakter menjadi token bermakna menggunakan Deterministic Finite Automata (DFA).
- Desain DFA menjadi dasar utama dalam pembuatan lexer karena menentukan bagaimana setiap karakter dibaca, dikenali, dan berpindah antar state hingga menghasilkan token yang sesuai.
- Lexer membantu tahap parsing dengan menyediakan daftar token yang terstruktur sehingga program Pascal-S dapat dianalisis dan dijalankan dengan benar.

4.2 Saran

- DFA dan aturan token harus dibuat jelas dan sesuai format agar proses pengenalan token berjalan akurat.
- Whitespace, komentar, dan error harus ditangani dengan baik agar lexer lebih akurat dan mudah diuji.

LAMPIRAN

Link Release Repository Github:
https://github.com/JethroJNS/FRT-Tubes-IF2224.git

Link *workspace* diagram:
https://drive.google.com/file/d/1Ye29iNGwghqO30mqT-_Bn0CnfE9Ehha3/view?usp=drive_link

Pembagian Tugas:

NIM	Nama	Persentase Kontribusi
13521117	Maggie Zeta Rosida S	25%
13523037	Buege Mahara Putra	25%
13523041	Hanif Kalyana Aditya	25%
13523081	Jethro Jens Norbert Simatupang	25%