Milestone 1 IF2224 Teori Bahasa Formal dan Otomata
Lexical Analyzer Untuk Compiler Bahasa Pascal-S

Kelompok CGK

Andi Farhan Hidayat	13523128
Andri Nurdianto	13523145
Rafael Marchel D. W.	13523146
Muhammad Kinan Arkansyaddad	13523152

Program Studi Teknik informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

Daftar Isi

Landasan Teori

Bahasa pemrograman adalah bahasa yang digunakan dalam proses komputasi. Bahasa pemrograman memiliki sistem notasi, sintaksis, semantik, dan instruksi yang membuat manusia dapat memberikan perintah terhadap komputer untuk tujuan tertentu. Segala hal yang ditulis dengan bahasa pemrograman disebut sebagai source code. Source code diproses melalui proses kompilasi atau interpreter. Lexical analysis merupakan tahap pertama dari proses kompilasi atau interpreter. Lexical analysis yang digunakan bergantung pada bahasa pemrograman yang digunakan, dalam persoalan ini lexical analysis digunakan untuk bahasa Pascal-S.

Bahasa Pascal-S adalah sebuah bahasa subset dari Pascal. Bahasa ini adalah hasil *filtering* bahasa Pascal yang dibuat pada 1971. Secara singkat, bahasa ini adalah versi *mini* dari bahasa Pascal original.

Lexical analysis adalah tahap pertama dari sebuah proses kompilasi atau interpreter. Dalam tahap ini, isi dari source code diproses menjadi sebuah token-token. Ada aturan untuk membentuk suatu token. Aturan-aturan ini didefinisikan oleh aturan bahasa (*grammar*), melalui sebuah pola. Kata kunci seperti string, angka, operator, dan simbol tanda baca memiliki sebuah aturan supaya kata kunci tersebut bisa diidentifikasi dan diubah menjadi sebuah token.

Token dideskripsikan menggunakan regular expressions, sedangkan lexical analyzer menggunakan *Deterministic Finite Automata* (DFA) untuk mengidentifikasi sebuah token. Setiap *final state* dari DFA berguna untuk mengidentifikasi tipe token sehingga lexical analyzer dapat mengklasifikasi sebuah input. Proses pembuatan DFA dari *regular expressions* dapat diotomatisasi agar token lebih efisien untuk diidentifikasi.

Finite automata adalah *state machine* yang dapat menerima simbol sebagai input dan mengubah statenya sesuai input yang masuk. *Deterministic Finite Automata* (DFA) adalah salah satu jenis finite automata yang digunakan dalam lexical analysis.

DFA adalah salah satu model matematis yang termasuk *finite automata*. DFA digunakan untuk mengenali bahasa-bahasa formal yang dapat dihasilkan oleh bahasa atau tata bahasa tertentu dalam komputasi. Secara formal, DFA didefinisikan sebagai 5 tuple (Q, Σ, δ, q0, F) dengan rincian sebagai berikut:

Q : finite set dari states
Σ : finite alfabet
δ : fungsi transisi dari Q × Σ
$q_0 \in Q$: state awal
$F \in Q$: set dari final state

DFA membaca simbol-simbol dari alfabet Σ satu per satu, mulai dari keadaan awal q0 lalu bergerak dari satu state ke state lain berdasarkan fungsi transisi δ.

Setelah DFA selesai membaca seluruh masukan, DFA akan berada pada salah satu keadaan. Jika *state* tersebut adalah *final state* (F), DFA mengenali masukan sebagai bahasa formal tertentu. Jika tidak, DFA tidak mengenali masukan tersebut.

Perancangan

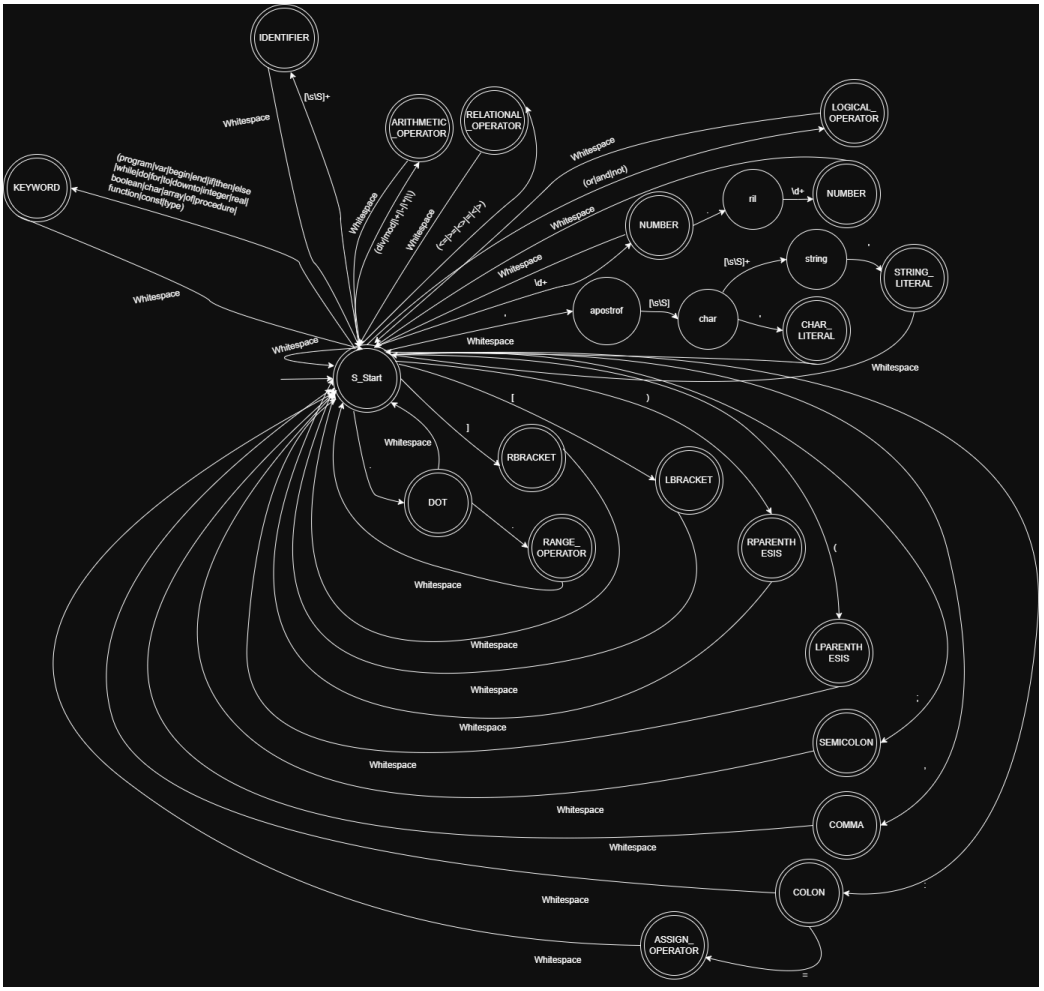
Perancangan dimulai dengan mengidentifikasi tipe token terlebih dahulu. Hal itu digunakan untuk mengetahui tuple yang dibutuhkan dalam perancangan DFA. Berikut adalah seluruh daftar token yang digunakan dalam bahasa Pascal-S.

Tabel 2.1 Daftar Token Pascal-S

No	Tipe (type)	Nilai (value)	Keterangan
1	**KEYWORD**	`program, var, begin, end, if, then, else, while, do, for, to, downto, integer, real, boolean, char, array, of, procedure, function, const, type, true, false`	Kata kunci yang sudah didefinisikan oleh bahasa Pascal-S dan memiliki fungsi khusus dalam struktur program.
2	**IDENTIFIER**	`x, y, z, sum, avg, count`	Nama yang didefinisikan oleh pengguna, misalnya nama variabel, prosedur, atau fungsi.
3	**ARITHMETIC_OPERATOR**	`+, -, *, /, div, mod`	-
4	**RELATIONAL_OPERATOR**	`=, <>, <, <=, >, >=`	-
5	**LOGICAL_OPERATOR**	`and, or, not`	-
6	**ASSIGN_OPERATOR**	`:=`	Operator penugasan yang digunakan untuk memberi nilai ke variabel
7	**NUMBER**	`22, 3, 2018`	Bilangan berupa integer atau ril
8	**CHAR_LITERAL**	`'a', 'b', 'c'`	-

9	**STRING_LITERAL**	`'tbfo', 'seru sekali'`	-
10	**SEMICOLON**	`;`	-
11	**COMMA**	`,`	-
12	**COLON**	`:`	-
13	**DOT**	`.`	-
14	**LPARENTHESIS**	`(`	-
15	**RPARENTHESIS**	`)`	-
16	**LBRACKET**	`[`	-
17	**RBRACKET**	`]`	-
18	**RANGE_OPERATOR**	`..`	-

Setelah mengidentifikasi tipe token, diagram DFA digunakan. DFA ini digunakan untuk mengidentifikasi token berdasarkan simbol tanda baca dan alfanumerik. Rancangan DFA dibuat dengan gambar diagram sebagai berikut:



Gambar 1. Rancangan Diagram DFA

Untuk membuat sebuah lexical analyzer dalam kasus ini, sebuah bahasa pemrograman dipakai sebagai lexical analyzer bahasa Pascal-S. Dalam hal ini, bahasa pemrograman yang dipakai adalah Rust.

Rust dipakai karena aman, konkuren, dan praktis. Selain itu, Rust biasa digunakan dalam system programming seperti bahasa C. *No garbage collection* yang membuat performa lebih tinggi, keamanan memori yang terjamin, dan efisiensi lebih tinggi merupakan alasan lain bahasa ini dipakai.

Struktur Program

Struktur program terdiri dari folder src yang berisi *source code file* dengan ekstensi .rs sebagai program lexical analyzer bahasa Pascal-S. Cargo.toml sebagai file konfigurasi untuk menjalankan kode dengan ekstensi .pas. File dfa_rules.json berisi rancangan representasi DFA. File test.pas dan folder test sebagai source code bahasa Pascal-S yang diujikan.

Struktur Program

```
CGK-TUBES-IF2224
```

```
├── Cargo.lock
├── Cargo.toml
├── dfa_rules.json
├── LICENSE
├── README.md
├── src
│   ├── dfa.rs
│   ├── lexer.rs
│   ├── main.rs
│   └── token.rs
├── test
│   └── milestone1
│       ├── input-1.pas
│       ├── input-2.pas
│       ├── input-3.pas
│       ├── input-4.pas
│       └── input-5.pas
└── test.pas
```

Fungsi/Kelas Utama

fungsi main

```
use std::env;
use std::fs::File;
use std::io::{BufWriter, Write};

use crate::{dfa::Dfa, lexer::Lexer};

mod token;
mod dfa;
mod lexer;

fn main() {
    let args: Vec<String> = env::args().collect();
    if args.len() < 3 {
        eprintln!("Usage: {} <path_to_pascal_file>
<pathtooutput>", args[0]);
        return;
    }

    let filepath = &args[1];
    let pathtooutput = &args[2];

    let dfa = match Dfa::from_file("dfa_rules.json") {
        Ok(d) => d,
        Err(e) => {
            eprintln!("Error loading dfa_rules.json: {}", e);
            return;
        }
    };
```

```
    let source_code = match std::fs::read_to_string(filepath) {
        Ok(s) => s,
        Err(e) => {
            eprintln!("Error reading file {}: {}", filepath, e);
            return;
        }
    };

    let mut lexer = Lexer::new(source_code, dfa);
    let mut tokens = Vec::new();

    while let Some(token) = lexer.get_next_token() {
        tokens.push(token);
    }

    println!("---TOKENS---");
    for token in &tokens {
        println!("{}", token);
    }
    println!("------------");

    let file = match File::create(pathtooutput) {
        Ok(f) => f,
        Err(e) => {
            eprintln!("Error output file {}: {}", pathtooutput,
e);
            return;
        }
    };
    let mut writer = BufWriter::new(file);

    writeln!(writer, "---TOKENS---").unwrap();
    for token in &tokens {
        writeln!(writer, "{}", token).unwrap();
    }
    writeln!(writer, "------------").unwrap();

    writer.flush().unwrap();
}
```

fungsi Dfa

```
impl Dfa {
    pub fn from_file(path: &str) -> Result<Self, Box<dyn
std::error::Error>> {
        let file_content = std::fs::read_to_string(path)?;
        let dfa: Dfa = serde_json::from_str(&file_content)?;
        Ok(dfa)
    }
}
```

implementation Display

```
impl fmt::Display for Token {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        // We convert the enum variant to a string for the output
        let type_str = match self.token_type {
            TokenType::Keyword => "KEYWORD",
            TokenType::Identifier => "IDENTIFIER",
            TokenType::ArithmeticOperator =>
"ARITHMETIC_OPERATOR",
            TokenType::RelationalOperator =>
"RELATIONAL_OPERATOR",
            TokenType::LogicalOperator => "LOGICAL_OPERATOR",
            TokenType::AssignOperator => "ASSIGN_OPERATOR",
            TokenType::Number => "NUMBER",
            TokenType::CharLiteral => "CHAR_LITERAL",
            TokenType::StringLiteral => "STRING_LITERAL",
            TokenType::Semicolon => "SEMICOLON",
            TokenType::Comma => "COMMA",
            TokenType::Colon => "COLON",
            TokenType::Dot => "DOT",
            TokenType::LParenthesis => "LPARENTHESIS",
            TokenType::RParenthesis => "RPARENTHESIS",
            TokenType::LBracket => "LBRACKET",
            TokenType::RBracket => "RBRACKET",
            TokenType::RangeOperator => "RANGE_OPERATOR",
        };
        write!(f, "{}({})", type_str, self.value)
    }
}
```

fungsi new

```
pub fn new(source: String, dfa: Dfa) -> Self {
    Lexer { source: source.chars().collect(), dfa, position: 0}
}
```

fungsi get_next_token

```
pub fn get_next_token(&mut self) -> Option<Token> {
        while self.position < self.source.len() &&
self.source[self.position].is_whitespace() {
            self.position += 1;
        }

        if self.position >= self.source.len() {
            return None;
        }

        let mut current_state = self.dfa.start_state.clone();
```

```
        let start_pos = self.position;
        let mut last_final_state: Option<(String, usize)> = None;

        while self.position < self.source.len() {
            let current_char = self.source[self.position];

            if let Some(next_state) =
self.get_next_state(&current_state, current_char) {
                current_state = next_state;
                self.position += 1;

                if
self.dfa.final_states.contains_key(&current_state) {
                    last_final_state =
Some((current_state.clone(), self.position));
                }

                if current_state == "S_Start" {
                    let token_value: String =
self.source[start_pos..self.position].iter().collect();
                    if token_value.starts_with('{') ||
token_value.starts_with("(*") {
                        return self.get_next_token();
                    }
                }
            } else {
                break;
            }
        }

        if let Some((final_state, end_pos)) = last_final_state {
            let value: String =
self.source[start_pos..end_pos].iter().collect();
            self.position = end_pos;

            if let Some(token_type_str) =
self.dfa.final_states.get(&final_state) {
                let mut token = self.create_token(token_type_str,
value);

                if token.token_type == TokenType::Identifier {
                    self.check_identifier(&mut token);
                }

                if token.token_type == TokenType::StringLiteral {
                    let content =
&token.value[1..&token.value.len() - 1];
                    if content.chars().count() == 1 {
                        token.token_type =
TokenType::CharLiteral;
                    }
                }

                return Some(token);
            }
```

```
        }

        if self.position < self.source.len() {
            eprintln!("Error: Invalid token starting with '{}' at
position {}", self.source[start_pos], start_pos);
            self.position = self.source.len();
        }

        None
    }
```

fungsi get_next_state

```
fn get_next_state(&self, current_state: &str, ch: char) ->
Option<String> {
        if let Some(transitions) =
self.dfa.transitions.get(current_state) {
            if let Some(next_state) =
transitions.get(&ch.to_string()) {
                return Some(next_state.clone());
            }

            for (key, next_state) in transitions {
                if key.contains('-') && key.len() == 3 {
                    let mut parts = key.chars();
                    let start = parts.next()?;
                    parts.next();
                    let end = parts.next()?;
                    if ch >= start && ch <= end {
                        return Some(next_state.clone());
                    }
                } else if key.contains(ch) && !key.contains('-')
{
                    return Some(next_state.clone());
                }
            }

            if let Some(next_state) = transitions.get("any") {
                return Some(next_state.clone());
            }
        }

        None
    }
```

fungsi create_token

```
fn create_token(&self, token_type_str: &str, value: String) ->
Token {
        let token_type = match token_type_str {
```

```
            "IDENTIFIER" => TokenType::Identifier,
            "NUMBER" => TokenType::Number,
            "STRING_LITERAL" => TokenType::StringLiteral,
            "ASSIGN_OPERATOR" => TokenType::AssignOperator,
            "RELATIONAL_OPERATOR" =>
TokenType::RelationalOperator,
            "ARITHMETIC_OPERATOR" =>
TokenType::ArithmeticOperator,
            "COLON" => TokenType::Colon,
            "DOT" => TokenType::Dot,
            "RANGE_OPERATOR" => TokenType::RangeOperator,
            "SEMICOLON" => TokenType::Semicolon,
            "COMMA" => TokenType::Comma,
            "LPARENTHESIS" => TokenType::LParenthesis,
            "RPARENTHESIS" => TokenType::RParenthesis,
            "LBRACKET" => TokenType::LBracket,
            "RBRACKET" => TokenType::RBracket,
            _ => panic!("Unknown token type: {}",
token_type_str),
        };
        Token { token_type, value }
    }
```

fungsi check identifier

```
fn check_identifier(&self, token: &mut Token) {
        if self.dfa.keywords.contains(&token.value) {
            token.token_type = TokenType::Keyword;
        } else if
self.dfa.word_logical_operators.contains(&token.value) {
            token.token_type = TokenType::LogicalOperator;
        } else if
self.dfa.word_arithmetic_operators.contains(&token.value) {
            token.token_type = TokenType::ArithmeticOperator;
        }
    }
```

Alur Kerja Program

File main.rs akan memanggil source code bahasa Pascal-S menggunakan variabel source_code dan membaca DFA dari file dfa_rules.json lalu dibuat objek lexer yang dipanggil dari file lexer.rs. Pada file lexer.rs, lexer akan membaca kode dari kiri ke kanan lalu akan menggunakan fungsi get_next_state dan dfa.transitions untuk berpindah antar state. Ketika menemukan final state, akan dibuat token sementara yang sesuai serta jika tidak token selesai dan dikembalikan. Setelah itu, lexer akan kembali ke state awal untuk membuat token selanjutnya sampai seluruh kode dalam source code selesai terbaca.

Implementasi

input

```
program TestAllTokens;

{ This is a block comment
  Good luck UTS. }

var
  my_integer, another_var : integer;
  a_real_number          : real;
  is_done                : boolean;
  my_char                : char;

const
  PI = 3.14159;

begin
  (* Tes assignments and expressions *)
  my_integer := 100;
  another_var := my_integer + 20 div 5;
  a_real_number := my_integer / 3.0;

  (* Tes Relational and logical operators *)
  if (my_integer > 50) and (another_var <> 104) then
  begin
    is_done := true;
  end
  else
  begin
    is_done := false;
  end;

  (* Character and String Literals *)
  my_char := 'A';
  writeln('This is a test string literal.');

  (* Testing multi-character operators *)
  if another_var <= 105 then
    writeln('Less than or equal');

  (* Range operator for arrays or subranges *)
  { array declaration is just for tokenizing '..' }
  type
    Numbers = array[1..10] of integer;

end. (* End of the test program. *)
```

output

```
$ cargo run test.pas
---TOKENS---
KEYWORD(program)
IDENTIFIER(TestAllTokens)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(my_integer)
COMMA(,)
IDENTIFIER(another_var)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(a_real_number)
COLON(:)
KEYWORD(real)
SEMICOLON(;)
IDENTIFIER(is_done)
COLON(:)
KEYWORD(boolean)
SEMICOLON(;)
IDENTIFIER(my_char)
COLON(:)
KEYWORD(char)
SEMICOLON(;)
KEYWORD(const)
IDENTIFIER(PI)
RELATIONAL_OPERATOR(=)
NUMBER(3.14159)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(my_integer)
ASSIGN_OPERATOR(:=)
NUMBER(100)
SEMICOLON(;)
IDENTIFIER(another_var)
ASSIGN_OPERATOR(:=)
IDENTIFIER(my_integer)
ARITHMETIC_OPERATOR(+)
```

```
$ cargo run test.pas
NUMBER(20)
ARITHMETIC_OPERATOR(div)
NUMBER(5)
SEMICOLON(;)
IDENTIFIER(a_real_number)
ASSIGN_OPERATOR(:=)
IDENTIFIER(my_integer)
ARITHMETIC_OPERATOR(/)
NUMBER(3.0)
SEMICOLON(;)
KEYWORD(if)
LPARENTHESIS(()
IDENTIFIER(my_integer)
RELATIONAL_OPERATOR(>)
NUMBER(50)
RPARENTHESIS())
LOGICAL_OPERATOR(and)
LPARENTHESIS(()
IDENTIFIER(another_var)
RELATIONAL_OPERATOR(<>)
NUMBER(104)
RPARENTHESIS())
KEYWORD(then)
KEYWORD(begin)
IDENTIFIER(is_done)
ASSIGN_OPERATOR(:=)
IDENTIFIER(true)
SEMICOLON(;)
KEYWORD(end)
KEYWORD(else)
KEYWORD(begin)
IDENTIFIER(is_done)
ASSIGN_OPERATOR(:=)
IDENTIFIER(false)
SEMICOLON(;)
KEYWORD(end)
SEMICOLON(;)
```

```
$ cargo run test.pas
KEYWORD(end)
SEMICOLON(;)
IDENTIFIER(my_char)
ASSIGN_OPERATOR(:=)
CHAR_LITERAL('A')
SEMICOLON(;)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('This is a test string literal.')
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(if)
IDENTIFIER(another_var)
RELATIONAL_OPERATOR(<=)
NUMBER(105)
KEYWORD(then)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Less than or equal')
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(type)
IDENTIFIER(Numbers)
RELATIONAL_OPERATOR(=)
KEYWORD(array)
LBRACKET([)
NUMBER(1)
RANGE_OPERATOR(..)
NUMBER(10)
RBRACKET(])
KEYWORD(of)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(end)
DOT(.)
------------
```

Deklarasi Program	Hasil
`program TestAllTokens;`	KEYWORD(program) IDENTIFIER(TestAllTokens) SEMICOLON(;)
Deklarasi variabel dan tipe data	**Hasil**

`var`
`  my_integer, another_var :`
`integer;`
`  a_real_number : real;`
`  is_done : boolean;`
`  my_char : char;`
`const`
`  PI = 3.14159;`
	KEYWORD(var)
	IDENTIFIER(my_integer)
	COMMA(,)
	IDENTIFIER(another_var)
	COLON(:)
	KEYWORD(integer)
	SEMICOLON(;)
	IDENTIFIER(a_real_number)
	COLON(:)
	KEYWORD(real)
	SEMICOLON(;)
	IDENTIFIER(is_done)
	COLON(:)
	KEYWORD(boolean)
	SEMICOLON(;)
	IDENTIFIER(my_char)
	COLON(:)

	KEYWORD(char) SEMICOLON(;) KEYWORD(const) IDENTIFIER(PI) RELATIONAL_OPERATOR(=) NUMBER(3.14159)
Variable assignment dan operator	Hasil

`my_integer := 100;`
`another_var := my_integer + 20 div 5;`
`a_real_number := my_integer / 3.0;`
	IDENTIFIER(my_integer)
	ASSIGN_OPERATOR(:=)
	NUMBER(100)
	SEMICOLON(;)
	IDENTIFIER(another_var)
	ASSIGN_OPERATOR(:=)
	IDENTIFIER(my_integer)
	ARITHMETIC_OPERATOR(+)
	NUMBER(20)
	ARITHMETIC_OPERATOR(div)
	NUMBER(5)
	SEMICOLON(;)
	IDENTIFIER(a_real_number)
	ASSIGN_OPERATOR(:=)
	IDENTIFIER(my_integer)
	ARITHMETIC_OPERATOR(/)
	NUMBER(3.0)
	SEMICOLON(;)

Conditional expression	Hasil

`if (my_integer > 50) and (another_var <> 104) then`
`  begin`
`    is_done := true;`
`  end`
`  else`
`  begin`
`    is_done := false;`
`  end;`
	KEYWORD(if)
	LPARENTHESIS(()
	IDENTIFIER(my_integer)
	RELATIONAL_OPERATOR(>)
	NUMBER(50)
	RPARENTHESIS())
	LOGICAL_OPERATOR(and)
	LPARENTHESIS(()
	IDENTIFIER(another_var)
	RELATIONAL_OPERATOR(<>)
	NUMBER(104)
	RPARENTHESIS())
	KEYWORD(then)
	KEYWORD(begin)
	IDENTIFIER(is_done)

	ASSIGN_OPERATOR(:=) IDENTIFIER(true) SEMICOLON(;) KEYWORD(end) KEYWORD(else) KEYWORD(begin) IDENTIFIER(is_done) ASSIGN_OPERATOR(:=) IDENTIFIER(false) SEMICOLON(;) KEYWORD(end) SEMICOLON(;)
Char literal dan string literal	Hasil
`my_char := 'A';` `writeln('This is a test string literal.');`	IDENTIFIER(writeln) LPARENTHESIS(() STRING_LITERAL('This is a test string literal.') RPARENTHESIS()) SEMICOLON(;)
Multi-character operator	Hasil
`if another_var <= 105 then` `    writeln('Less than or equal');`	KEYWORD(if) IDENTIFIER(another_var) RELATIONAL_OPERATOR(<=) NUMBER(105) KEYWORD(then) IDENTIFIER(writeln) LPARENTHESIS(() STRING_LITERAL('Less than or equal') RPARENTHESIS()) SEMICOLON(;)
Range operator	Hasil
`type` `    Numbers = array[1..10] of integer;`	KEYWORD(type) IDENTIFIER(Numbers) RELATIONAL_OPERATOR(=) KEYWORD(array) LBRACKET([) NUMBER(1) RANGE_OPERATOR(..) NUMBER(10) RBRACKET(])

	KEYWORD(of) KEYWORD(integer) SEMICOLON(;)
End of program	Hasil
`end.`	KEYWORD(end) DOT(.)

Hasil perubahan rangkaian kode menjadi token oleh lexical analyzer berhasil berjalan sesuai fungsinya. Deklarasi program memiliki *keyword* program dan tanda baca ; dikenali sebagai token KEYWORD dan SEMICOLON. Deklarasi tipe data pada source code diproses menjadi IDENTIFIER sebagai nama variabel dan KEYWORD sebagai tipe data (integer, real, boolean, dll). Pemrosesan assignment berhasil mengenali operator :=, +, /, dan div sebagai ASSIGN_OPERATOR dan ARITHMETIC_OPERATOR. Ekspresi kondisional seperti if, then, else dikenali sebagai KEYWORD. Untuk string dan char lexer berhasil mengenali menjadi STRING_LITERAL dan CHAR_LITERAL. Multi-character operator seperti <= dan range operator .. berhasil dikenali sebagai RELATIONAL_OPERATOR dan RANGE_OPERATOR. Program ditutup dengan end yang dikenali sebagai KEYWORD dan . sebagai DOT.

Pengujian

input-1.pas

input

```
program HelloWorld;

begin
    writeln('Hello World!');
end.
```

output

```
$ cargo run test/milestone1/input-1.pas
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.04s
     Running `target\debug\CGK-Tubes-IF2224.exe test/milestone1/input-1.pas`
---TOKENS---
KEYWORD(program)
IDENTIFIER(HelloWorld)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Hello World!')
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(end)
DOT(.)
------------
```

input-2.pas

input

```
program JumlahAja;

var
    a, b, hasil: integer;

begin
    a := 58;
    b := 9;
    hasil := a + b;
    writeln('hasil penjumlahan tersebut adalah ', hasil);
end.
```

output

```
$ cargo run test/milestone1/input-2.pas
     Running `target\debug\CGK-Tubes-IF2224.exe test/milestone1/input-2.pas`
---TOKENS---
KEYWORD(program)
IDENTIFIER(JumlahAja)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(a)
COMMA(,)
IDENTIFIER(b)
COMMA(,)
IDENTIFIER(hasil)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(a)
ASSIGN_OPERATOR(:=)
NUMBER(58)
SEMICOLON(;)
IDENTIFIER(b)
ASSIGN_OPERATOR(:=)
NUMBER(9)
SEMICOLON(;)
IDENTIFIER(hasil)
ASSIGN_OPERATOR(:=)
IDENTIFIER(a)
ARITHMETIC_OPERATOR(+)
IDENTIFIER(b)
SEMICOLON(;)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('hasil penjumlahan tersebut adalah ')
COMMA(,)
IDENTIFIER(hasil)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(end)
DOT(.)
------------
```

input-3.pas

input

```
program CobaChar;

var
    a, b, c, d: char;

begin
    a := 'a';
    b := 'b';
    c := 'c';
    d := 'd';

    writeln(a, b, a, c, a, d);
end.
```

output

```
$ cargo run test/milestone1/input-3.pas
     Running `target\debug\CGK-Tubes-IF2224.exe test/milestone1/input-3.pas`
---TOKENS---
KEYWORD(program)
IDENTIFIER(CobaChar)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(a)
COMMA(,)
IDENTIFIER(b)
COMMA(,)
IDENTIFIER(c)
COMMA(,)
IDENTIFIER(d)
COLON(:)
KEYWORD(char)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(a)
ASSIGN_OPERATOR(:=)
CHAR_LITERAL('a')
SEMICOLON(;)
IDENTIFIER(b)
ASSIGN_OPERATOR(:=)
CHAR_LITERAL('b')
SEMICOLON(;)
IDENTIFIER(c)
ASSIGN_OPERATOR(:=)
CHAR_LITERAL('c')
SEMICOLON(;)
IDENTIFIER(d)
ASSIGN_OPERATOR(:=)
CHAR_LITERAL('d')
SEMICOLON(;)
IDENTIFIER(writeln)
LPARENTHESIS(()
IDENTIFIER(a)
COMMA(,)
```

```
COMMA(,)
IDENTIFIER(b)
COMMA(,)
IDENTIFIER(a)
COMMA(,)
IDENTIFIER(c)
COMMA(,)
IDENTIFIER(a)
COMMA(,)
IDENTIFIER(d)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(end)
DOT(.)
------------
```

input-4.pas

input

```
program UTS;

var
   pekan: integer;

begin
   pekan := 8;

   if pekan = 8 then
      writeln('Semangat UTS')
   else
      writeln('Nugas moal?');
end.
```

output

```
$ cargo run test/milestone1/input-4.pas
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.03s
     Running `target\debug\CGK-Tubes-IF2224.exe test/milestone1/input-4.pas`
---TOKENS---
KEYWORD(program)
IDENTIFIER(UTS)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(pekan)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(begin)
IDENTIFIER(pekan)
ASSIGN_OPERATOR(:=)
NUMBER(8)
SEMICOLON(;)
KEYWORD(if)
IDENTIFIER(pekan)
RELATIONAL_OPERATOR(=)
NUMBER(8)
KEYWORD(then)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Semangat UTS')
RPARENTHESIS())
KEYWORD(else)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Nugas moal?')
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(end)
DOT(.)
------------
```

input-5.pas

input

```
program HitungMundur;

var
   i: integer;

begin
```

```
    for i := 3 downto 1 do
        writeln(i);
end.
```

output

```
$ cargo run test/milestone1/input-5.pas
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.03s
     Running `target\debug\CGK-Tubes-IF2224.exe test/milestone1/input-5.pas`
---TOKENS---
KEYWORD(program)
IDENTIFIER(HitungMundur)
SEMICOLON(;)
KEYWORD(var)
IDENTIFIER(i)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(begin)
KEYWORD(for)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
NUMBER(3)
KEYWORD(downto)
NUMBER(1)
KEYWORD(do)
IDENTIFIER(writeln)
LPARENTHESIS(()
IDENTIFIER(i)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(end)
DOT(.)
------------
```

Kesimpulan

Lexical analysis menggunakan lexical analyzer berhasil menjalankan fungsinya untuk mengubah source code menjadi token-token. Dengan memanfaatkan Deterministic Define Automata (DFA), lexer mampu mengenali pola simbol, character, operator, *keyword*, dan tanda baca yang ada dalam source code menjadi token yang sesuai.

Saran

- Membuat rancangan DFA yang sesuai untuk memudahkan implementasi
- Membuat struktur kode yang modular agar error lebih bisa tertangani dan kode menjadi *reusable*

Lampiran

Link Release:

Link Workspace Diagram:
https://drive.google.com/file/d/1PV4MLjwrz4b8JZN_tS4bsr-PfXeCKjHy/view?usp=sharing

Pembagian Tugas:

NIM	Tugas
13523128	Diagram, Laporan
13523145	Diagram, main, dfa_rules, input & output, Laporan
13523146	dfa_rules, dfa, lexer, token, main, Laporan
13523152	Diagram, Laporan

Referensi

"'17 Jenis Bahasa Pemrograman, Pengertian, Fungsi dan Kelebihannya" https://psti.unikama.ac.id/id/17-jenis-bahasa-pemrograman-pengertian/#:~:text=Pengertian%20bahasa%20pemrograman%20merupakan%20sistem,memproses%20data%20atau%20membuat%20aplikasi.

"Compiler Design - Lexical Analysis" https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm

"Introduction of Lexical Analysis" https://www.geeksforgeeks.org/compiler-design/introduction-of-lexical-analysis/

"Simulator Mesin Deterministic Finite Automata (DFA) Berdasarkan Diagram Transisi Menggunakan Python" https://jurnal.tau.ac.id/index.php/siskom-kb/article/download/siskom-kb.v7i1.463/339/2071#:~:text=Abstract%E2%80%94Mesin%20Deterministic%20Finite%20Automata,string%20masukan%20pada%20mesin%20DFA.