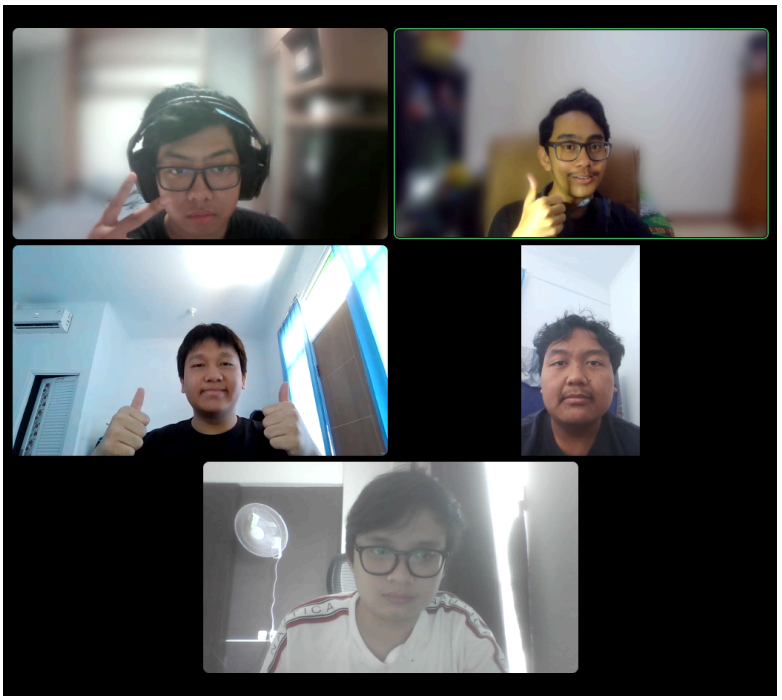
IF2224 – Teori Bahasa Formal & Otomata
Laporan Tugas Besar Milestone 1 - PASCAL-S Compiler



Dipersiapkan oleh:

K3 Kelompok2

Rhio Bimo Prakoso Sugiyanto	13523123
Muhammad Aulia Azka	13523137
Arlow Emmanuel Hergara	13523161
Fachriza Ahmad Setiyono	13523162
Filbert Engyo	13523163

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

2025

Daftar Isi

Daftar Gambar

Make sure to select headings in the sidebar to see a table of contents.

1. Landasan Teori

1.1. Deterministic Finite Otomata (DFA)

Deterministic Finite Automaton (DFA) adalah model matematika dalam ilmu komputer yang digunakan untuk mengenali pola dan bahasa formal. DFA adalah sebuah mesin status hingga yang menghitung input berupa rangkaian simbol dan menentukan penerimaan atau penolakan input tersebut berdasarkan jalur status yang unik dan telah ditentukan secara deterministik. Dengan kata lain, pada setiap keadaan dan simbol input tertentu, DFA hanya akan memiliki satu transisi menuju status berikutnya.

DFA berfungsi sebagai alat untuk memproses string dan memutuskan apakah string tersebut termasuk dalam bahasa yang dikenali oleh DFA itu. Secara formal, DFA didefinisikan sebagai kelima tupel $M = (Q, \Sigma, \delta, q0, F)$, di mana Q adalah himpunan status terbatas, Σ adalah alfabet input, δ adalah fungsi transisi yang menentukan status berikutnya berdasarkan status saat ini dan simbol input, q0 adalah status awal, dan F adalah himpunan status penerima (final). Mesin ini membaca simbol input satu per satu dari kiri ke kanan dan berpindah status sesuai fungsi transisi tanpa adanya ambiguitas.

DFA banyak digunakan dalam bidang teori otomata dan bahasa formal, termasuk dalam pembuatan lexer atau lexical analyzer di proses kompilasi. Pada lexer, DFA digunakan untuk memindai dan mengklasifikasikan urutan karakter menjadi token-token yang bermakna secara efisien dan deterministik. Dengan struktur yang simpel dan deterministik, DFA juga sering digunakan dalam pengenalan pola dan validasi input seperti alamat email atau format data tertentu.

Keunikan DFA terletak pada sifat deterministiknya di mana setiap input memiliki jalur status tunggal, sehingga menjamin tidak adanya ketidakpastian selama komputasi. Selain itu, DFA mampu mengenali bahasa reguler dan sangat efektif dalam aplikasi praktis karena algoritma transisinya yang sederhana. DFA juga dapat dikonversi dari model automata non deterministik (NFA) menggunakan metode konstruksi powerset, memastikan kesetaraan bahasa yang dikenali.

1.2. Token

Token adalah unit terkecil dari makna yang dihasilkan selama proses analisis leksikal dalam sebuah compiler atau interpreter. Menurut sumber dari GeeksforGeeks, token merupakan himpunan karakter yang dianggap sebagai satu kesatuan dalam struktur bahasa pemrograman. Dalam proses lexer, setiap token mewakili kategori tertentu seperti kata kunci, identifier, operator, literal, atau simbol lain yang memiliki arti khusus dalam bahasa sumber. Token ini merupakan output utama dari tahap leksikal yang kemudian digunakan dalam proses parsing untuk membangun struktur sintaksis program.

Token berfungsi dalam menyederhanakan proses pemrosesan kode sumber dengan mengelompokkan karakter-karakter menjadi unsur bahasa yang bermakna. Token memudahkan compiler dalam mengenali pola dan struktur program secara efisien, mengurangi kompleksitas dalam analisis sintaksis. Selain itu, token membantu dalam identifikasi kesalahan leksikal seperti uniknya penempatan karakter yang

tidak sesuai aturan bahasa, misalnya, identifier yang diawali angka atau kata kunci yang salah penulisan. Dengan demikian, token bertindak sebagai representasi formal dari bagian-bagian kode yang bermakna, yang berdampingan dengan fitur-layar lainnya dalam sistem kompilasi.

Dalam proses tokenisasi, lexer memecah input karakter menjadi token berdasarkan pola yang telah ditentukan dengan menggunakan ekspresi reguler atau pola tertentu. Token ini kemudian disimpan sebagai struktur data yang menyimpan tipe dan nilai literalnya, seperti yang dijelaskan dalam contoh implementasi lexer menggunakan objek token. Proses ini mendukung otomatisasi pengenalan elemen-elemen bahasa formal dan meningkatkan efisiensi analisis kode. Jadi, token adalah inti dari proses leksikal, mewakili hasil akhir dari scanner yang berfungsi sebagai dasar komunikasi antara lexer dan parser dalam rangka membangun struktur program secara otomatis dan terstruktur.

1.3. Lexeme

Lexeme adalah urutan karakter dalam kode sumber yang mencocokkan pola tertentu yang sudah ditetapkan sebelumnya dan membentuk token yang valid dalam proses lexical analysis (lexing). Sebagai contoh, dalam ekspresi seperti x + 5, bagian x dan 5 adalah lexeme yang sesuai dengan token identifier dan literal angka. Lexeme adalah unit dasar dalam proses scan kode sumber sebelum pengkategorian lebih lanjut menjadi token.

Lexeme berfungsi sebagai kumpulan karakter yang dikenali oleh lexer sesuai dengan pola aturan bahasa pemrograman yang diterapkan. Lexer pertama-tama memindai kode sumber dan menggabungkan karakter menjadi lexeme berdasarkan pola (pattern) yang didefinisikan, kemudian lexeme tersebut diklasifikasikan menjadi token. Dengan kata lain, lexeme adalah "bahan mentah" yang dicocokkan dan diidentifikasi sebelum diberi label token.

Lexeme adalah unit proses pertama yang dihasilkan oleh lexer, yang kemudian dikategorikan menjadi token sesuai jenisnya seperti keyword, identifier, operator, dan literal. Pola atau pattern merupakan aturan yang mendefinisikan bagaimana lexeme dikenali dan dikelompokkan ke dalam token tertentu. Setiap token didefinisikan dengan satu atau beberapa pattern yang menerima lexeme tertentu sebagai input. Dengan demikian, lexeme menjadi jembatan antara input kode mentah dan token yang terstruktur dalam compiler.

1.4. Lexical Analyzer (Lexer)

Lexer atau lexical analyzer adalah tahap awal dalam proses kompilasi yang bertugas mengubah kode sumber (source code) dari format karakter mentah menjadi unit-unit sintaksis kecil yang dikenal sebagai token. Token ini merepresentasikan elemen bahasa pemrograman yang paling dasar, seperti kata kunci, identifier, operator, dan literal. Dalam kata lain, lexer memecah aliran karakter menjadi segmen-segmen bermakna yang akan diproses lebih lanjut oleh parser.

Lexer memiliki beberapa fungsi utama dalam kompilasi. Pertama, lexer membaca kode sumber dan mengelompokkan karakter menjadi token sesuai dengan aturan

tata bahasa. Kedua, lexer menghapus noise leksikal seperti spasi putih dan komentar yang tidak diperlukan pada proses berikutnya. Ketiga, lexer juga berperan dalam pendeteksian kesalahan leksikal, misalnya error akibat identifier yang tidak memenuhi aturan bahasa pemrograman.

Sebagai tahap pertama dalam kompilasi, lexer berperan sebagai penghubung antara kode sumber dan parser. Output dari lexer adalah urutan token yang lebih mudah dianalisis secara sintaksis oleh parser. Dengan menyederhanakan kode melalui tokenisasi, lexer memudahkan deteksi pola dan struktur bahasa pada tahap parsing. Selain itu, lexer menjaga ketelitian dengan memastikan bahwa input kode sumber sesuai dengan aturan bahasa sebelum analisis lebih lanjut dimulai.

Berdasarkan penjelasan token sebelumnya, token adalah unit terkecil yang dihasilkan lexer, dan merupakan representasi abstrak dari bagian kode sumber yang bermakna. Penghasilan token dari aliran karakter dilakukan dengan memanfaatkan Deterministic Finite Automaton (DFA), dengan bergerak melewati rangkaian status berdasarkan karakter input, dan saat mencapai status akhir, token yang sesuai dikenali. Dengan cara ini, lexer mampu melakukan pemindaian kode secara efisien dan akurat menggunakan DFA sebagai dasar pengenalannya.

2. Perancangan & Implementasi

2.1. Diagram Transisi DFA

Pembahasan diagram transisi DFA yang digunakan pada *lexer* dibagi menjadi beberapa bagian berdasarkan token yang dibahas. Diagram yang ditunjukkan juga dipotong menjadi bagian-bagian kecil dan tidak menunjukkan keseluruhan diagram sekaligus. Hal ini dilakukan agar diagram dapat lebih mudah dipahami oleh pembaca. Diagram transisi DFA yang utuh dapat diperoleh dengan menggabungkan semua potongan diagram yang ditunjukkan dengan mencocokkan setiap *state* dan menggabung semua transisi. Diagram yang utuh juga dapat dilihat dalam format PDF pada tautan ini. Diagram transisi dibuat menggunakan *mermaid js* dengan layout *elk*, kode sumber untuk diagram dapat dilihat pada tautan ini.

Diagram transisi DFA pada laporan ini memiliki format yang berbeda dengan diagram transisi DFA pada umumnya. *State* pada diagram di laporan ini memiliki bentuk persegi panjang. Pada diagram ini, *state* biasa memiliki warna *background* putih sementara *state final* ditandakan dengan warna *background* hijau. Setiap transisi pada diagram dapat mencakup lebih dari satu karakter sehingga setiap transisi dituliskan sebagai serangkaian karakter atau *range* karakter yang dipisahkan dengan spasi. Karakter yang bukan alfanumerik dituliskan dengan dengan format [[x]] dengan x melambangkan karakter tersebut. *Range* karakter dituliskan dalam format x-y dengan x sebagai karakter pertama dan y sebagai karakter terakhir dan kedua karakter diurutkan berdasarkan nilai ASCII atau UTF-8 yang melambangkan karakter tersebut. *State* awal pada diagram transisi DFA ini dilambangkan oleh *state* 0.

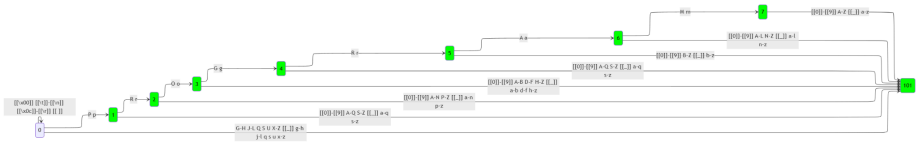
2.1.1. Token: Keyword

Setiap *state* yang berhubungan dengan *keyword* kecuali dengan *state* awal merupakan *final state*. Final state terakhir setiap *keyword* menghasilkan token *keyword* sementara *final state* pada *state* lain menghasilkan token identifier karena *keyword* yang tidak selesai menjadi identifier. Semua *state* selain dari *state* awal juga memiliki transisi ke *final state* identifier yaitu *state* 101. Transisi dilakukan pada semua input alfanumerik serta garis bawah yang tidak digunakan pada transisi lain (transisi ke *state* 101 menggunakan input sisa yang tidak digunakan transisi lain). Berikut adalah daftar *final state* masing-masing *keyword* diikuti dengan daftar potongan diagram DFA masing-masing *keyword*.

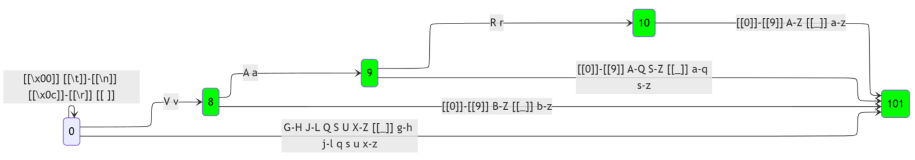
Keyword	Final State
Program	7
Var	10
Begin	15
End	18
If	20

Then	24
Else	27
While	32
Do	34
For	37
To	38
Downto	42
Integer	48
Real	52
Boolean	58
Char	62
Array	67
Of	69
Procedure	75
Function	82
Const	86
Type	89

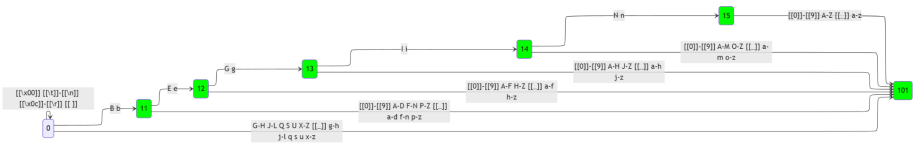
Keyword Program



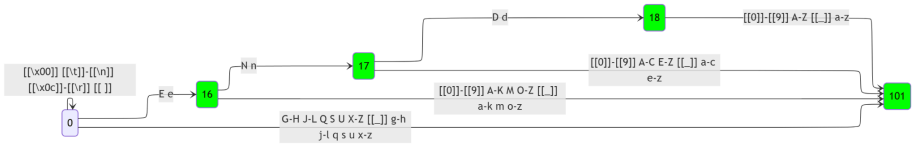
Keyword Var



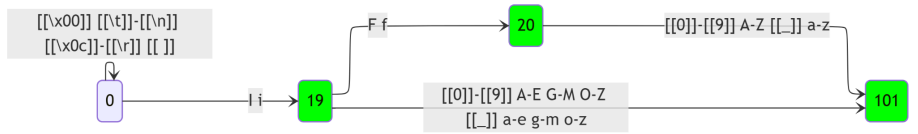
Keyword Begin



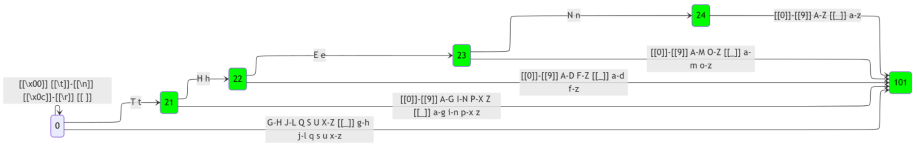
Keyword End



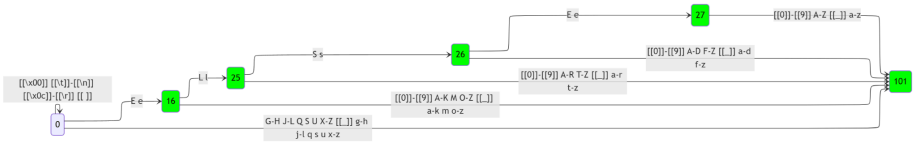
Keyword If



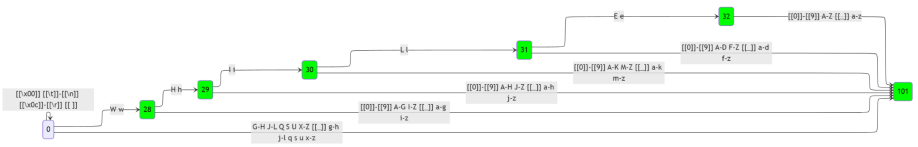
Keyword Then



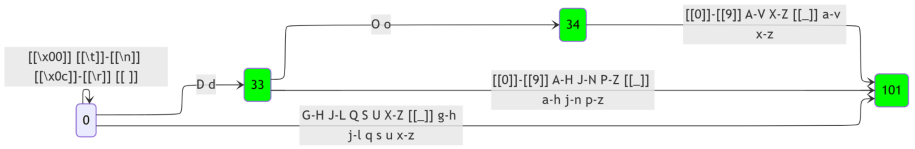
Keyword Else



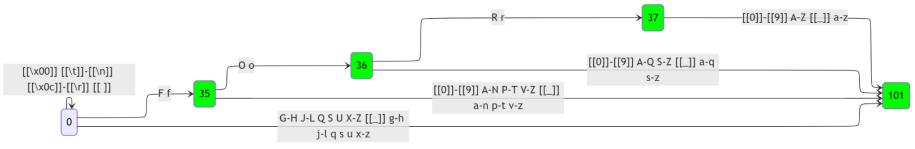
Keyword While



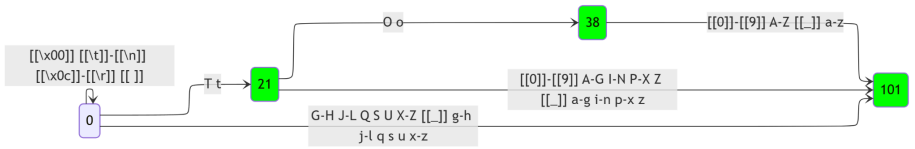
Keyword Do



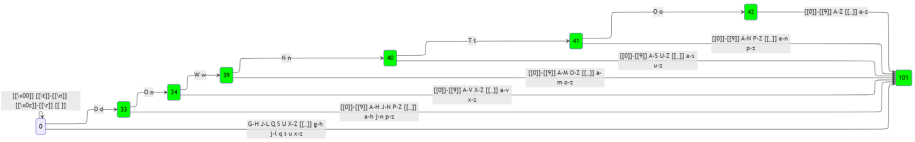
Keyword For



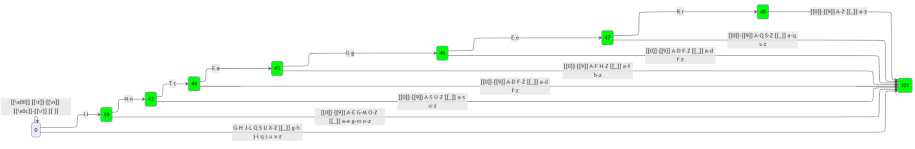
Keyword To



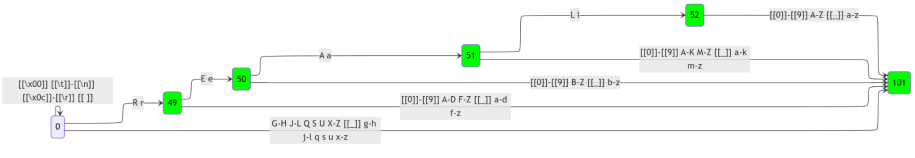
Keyword Downto



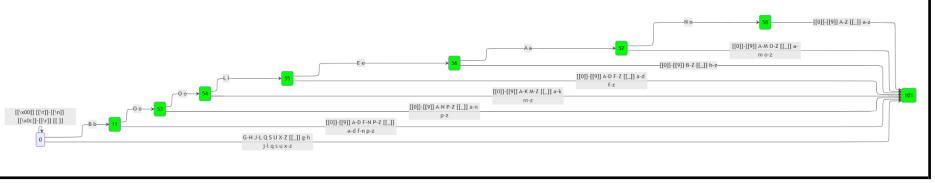
Keyword Integer



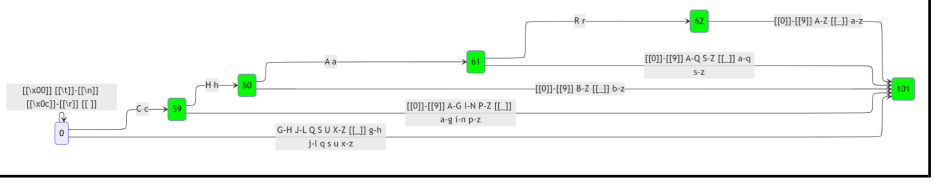
Keyword Real



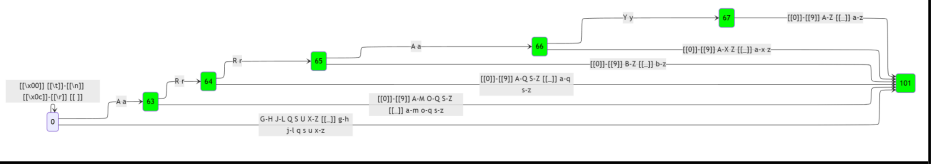
Keyword Boolean



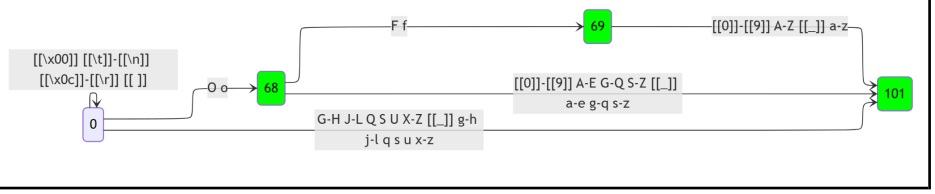
Keyword Char



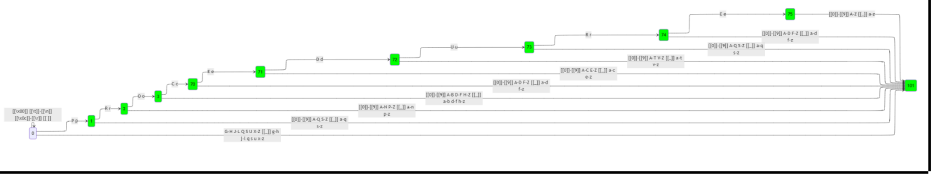
Keyword Array



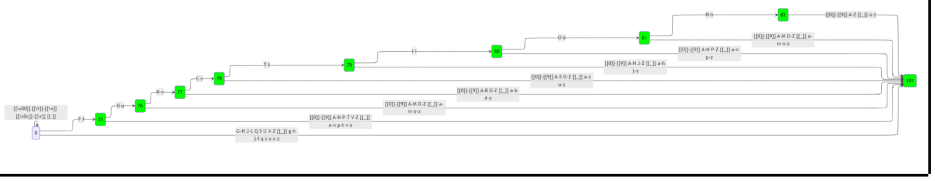
Keyword Of



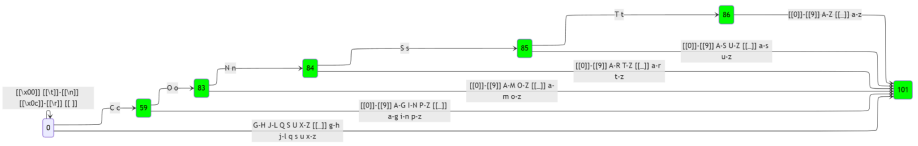
Keyword Procedure



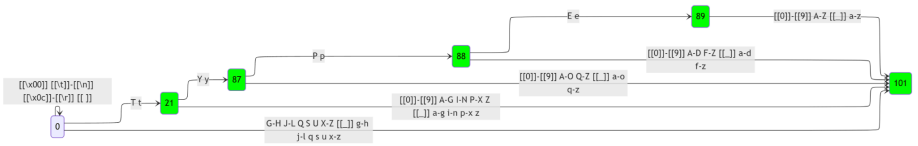
Keyword Function



Keyword Const



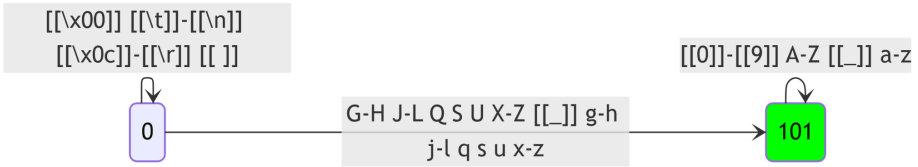
Keyword Type



2.1.2. *Token: Identifier*

Selain dari *final state* pada *state-state* berhubungan dengan *keyword*, *identifier* juga memiliki *final state* sendiri pada *state* 101. Semua *state* pada berhubungan dengan *keyword* beserta dengan *state* awal dapat melakukan transisi ke *state* 101. Transisi ke *state* 101 dilakukan jika input berikutnya adalah karakter yang tidak memiliki transisi lain dan merupakan alfanumerik atau garis bawah. *State* 101 juga bisa melakukan transisi ke dirinya sendiri dengan input alfanumerik atau garis bawah.

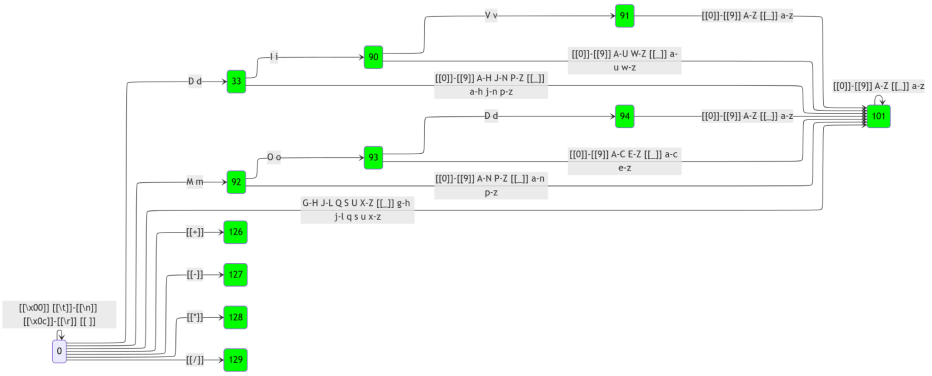
Token Indentifier



2.1.3. *Token: Arithmetic Operator*

Kebanyakan *arithmetic operator* hanya melibatkan satu transisi langsung ke *final state* masing-masing operator. Namun terdapat pengecualian pada dua operator yaitu mod dan div yang memerlukan 3 transisi untuk mencapai *final state*. Operator mod dan div juga memiliki *final state* pada setiap *state* yang bukan *final state* untuk *arithmetic operator* yang menghasilkan *identifier*. Setiap *state* yang berhubungan dengan operator mod dan div juga memiliki transisi ke *state* 101 yang merupakan *final state* dari *identifier* sama seperti transisi bersangkut *token keyword*. Berikut adalah daftar *final state* yang menghasilkan *arithmetic operator* untuk setiap operator.

Operator	Final State
div	91
mod	94
+	126
-	127
*	128
/	129

Token Arithmetic Operator


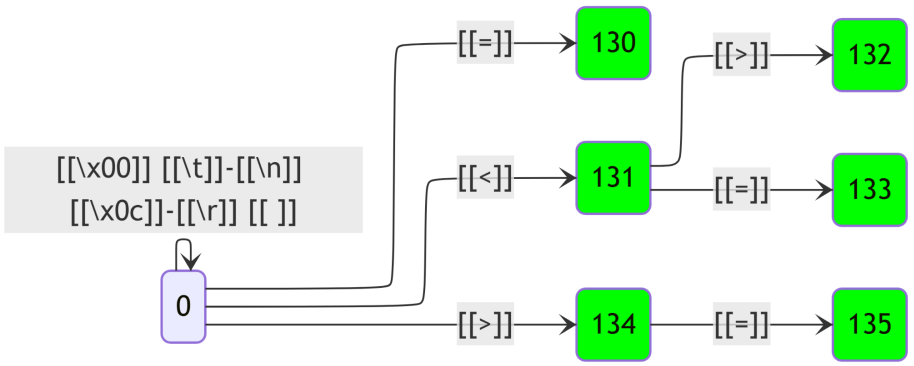
2.1.4. Token: Relational Operator

Diagram transisi untuk *relational operator* melibatkan transisi langsung ke *final state* sebagian besar operator-operator. Operator-operator lainnya memerlukan satu transisi tambahan setelah mencapai *final state* tersebut untuk mencapai *final state*-nya masing-masing. Berikut adalah daftar tiap operator dan *final state*-nya.

Operator	Final State
=	130
<	131
<>	132
<=	133
>	134

>=	135

Token Relational Operator

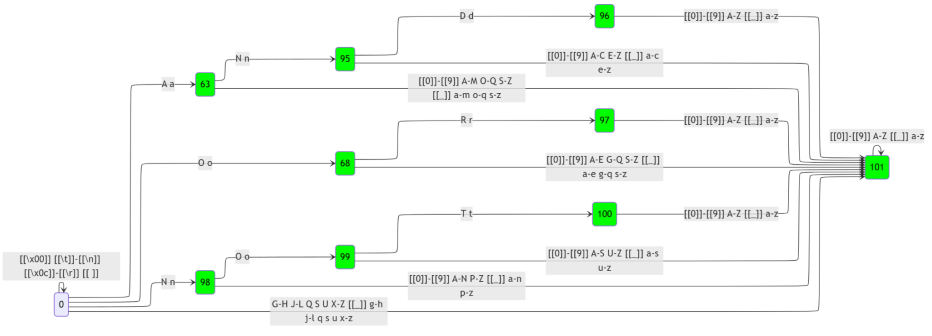


2.1.5. Token: Logical Operator

DFA menangani *logical operator* mirip dengan *keyword* karena semua *logical operator* tersusun dari huruf-huruf alfanumerik. Setiap operator memiliki *final state* yang menghasilkan *logical operator* dan *state* lain merupakan *final state* yang menghasilkan *identifier*. Setiap *state* juga memiliki transisi ke *state* 101 yang merupakan *final state* untuk *identifier* dengan aturan yang sama dengan *keyword*. Berikut adalah daftar operator dan *final state-nya* masing-masing.

Operator	Final State
and	96
or	97
not	100

Token Logical Operator

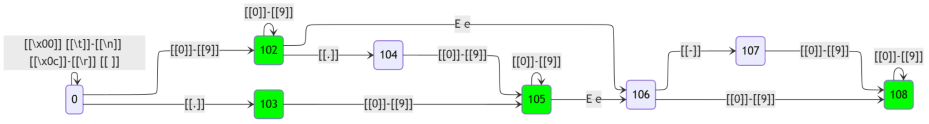


2.1.6. *Token: Number*

DFA dapat membaca token *number* dalam bentuk integer, real, dan juga *scientific notation*. Titik digunakan untuk memisahkan antara bagian integer dan bagian desimal. Huruf e digunakan untuk menyatakan perkalian dengan eksponen berbasis 10 jika menggunakan *scientific notation*. Angka real dapat dimulai langsung dengan titik yang secara implisit menyatakan bahwa bagian integernya merupakan 0. Token hanya menerima angka tanpa *sign* yang berarti angka negatif tidak diterima tetapi Eksponen tetap dapat negatif jika menggunakan *scientific notation*. Agar dapat mendukung integer, real, dan juga *scientific notation* DFA memiliki *final state* untuk masing-masing jenis yang menghasilkan token *number*. Selain itu, bagian DFA untuk *number* juga memiliki *final state* yang mengeluarkan *dot* karena titik dapat menjadi karakter awal dari suatu *number*. Berikut adalah daftar *final state* yang ada dalam bagian DFA ini.

Token	Final State
NUMBER(integer)	102
NUMBER(real)	105
NUMBER(real)	108
DOT	103

Token Number



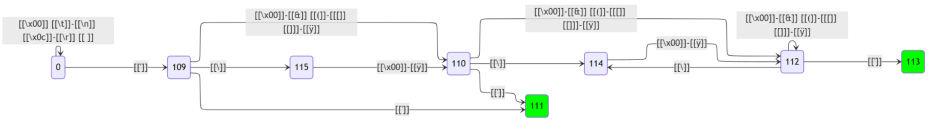
2.1.7. *Token: Char Literal & String Literal*

Baik *char literal* maupun *string literal* dimulai dengan karakter petik tunggal. Petik tunggal tersebut dapat diikuti dengan petik tunggal lagi, karakter “\”, atau karakter lainnya. Apabila diikuti langsung dengan petik tunggal, maka *state* akan mencapai *final state* yang mengeluarkan *char literal*. Sementara itu, apabila diikuti dengan “\”, maka akan masuk ke *state escape* yang dapat diikuti oleh karakter apapun untuk masuk ke *state* yang sama dengan apabila petik tunggal diikuti dengan karakter selain petik tunggal lain atau “\”. Pada *state* yang melambangkan petik tunggal diikuti oleh karakter selain petik tunggal atau “\”, terdapat transisi ke *final state* yang mengeluarkan *char literal* apabila karakter berikutnya

merupakan petik tunggal. *Final state* untuk *char literal* terletak pada *state* 111.

Selain dari transisi ke *final state char literal*, *state* yang melambangkan konten *char literal* dapat juga bertransisi ke *state string content* atau *state string escape*. Transisi ke *string escape* dilakukan dengan input "\" sementara transisi ke *string content* dilakukan dengan input karakter selain petik tunggal dan "\". *State string escape* dapat melakukan transisi ke *string content* dengan input karakter apapun. *State string content* dapat melakukan transisi ke dirinya sendiri dengan semua karakter selain petik tunggal atau "\". *String content* juga dapat melakukan transisi ke *final state* yang mengeluarkan *string literal* dengan input petik tunggal. *Final state* untuk *string literal* terletak pada *state* 113.

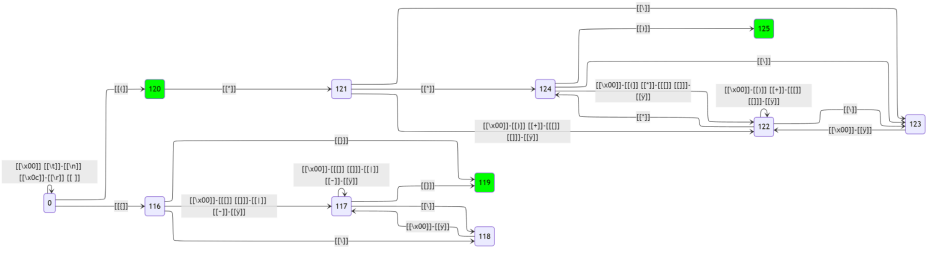
Token Char Literal & String Literal



2.1.8. Token: Comment

Bagian DFA yang memproses token *comment* bekerja mirip dengan *token string literal* dengan karakter pertama setelah penanda mulai komen langsung masuk ke konten atau *escape* tanpa logika penanganan karakter pertama. DFA juga diduplikasi untuk menangani dua blok komen ("{}" dan "(**)"). Untuk blok komen menggunakan kurung dan bintang, terdapat logika tambahan untuk bertransisi balik dari bintang penutup ke konten komen apabila bintang penutup tidak diikuti dengan kurung tutup. *Final state* yang menghasilkan token *comment* terletak pada *state* 119 untuk komen dengan kurung kurawal dan pada *state* 125 untuk komen dengan kurung dan bintang. Terdapat juga *final state* yang menghasilkan token *lparenthesis* pada *state* 120.

Token Comment

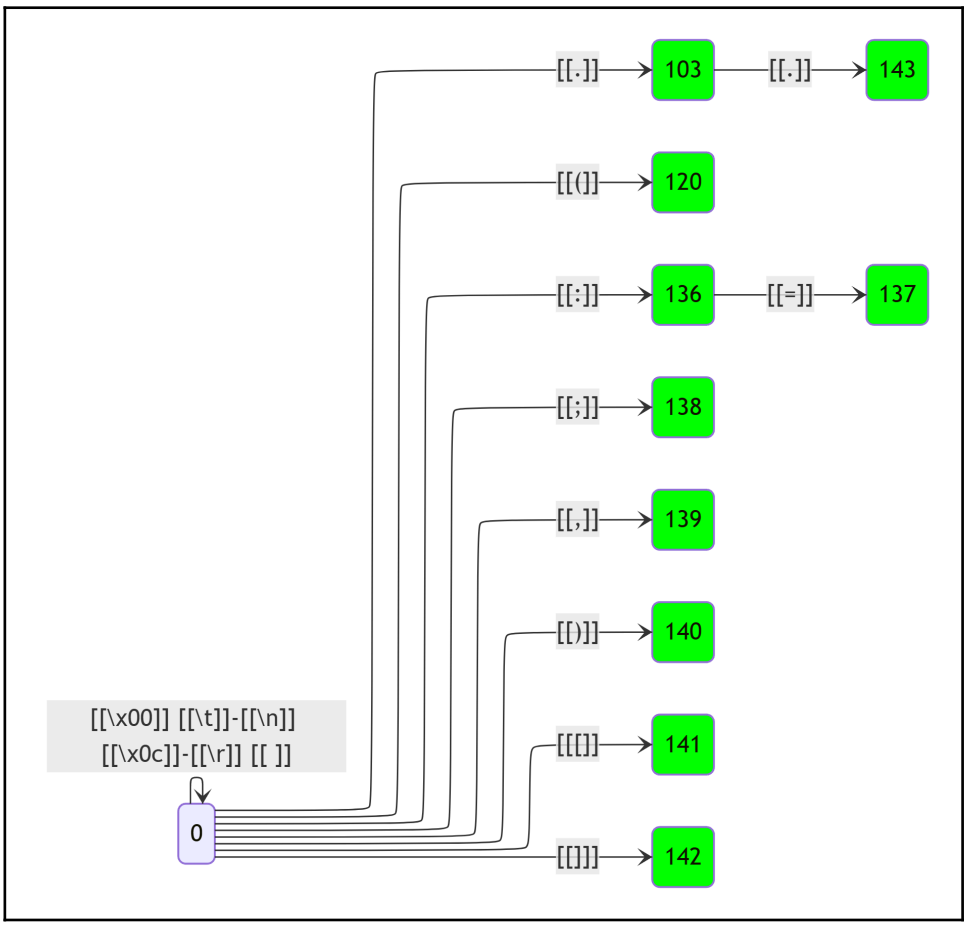


2.1.9. Token: Other

Bagian DFA ini berisi *state-state* dan *transisi-transisi* yang berhubungan dengan token-token tersisa, yaitu *assign operator*, *semicolon*, *comma*, *colon*, *dot*, *lparenthesis*, *rparenthesis*, *lbracket*, *rbracket*, dan *range operator*. Kebanyakan dari token-token ini hanya memerlukan satu transisi dan hanya dua yang memerlukan dua transisi dari *state* lain. Berikut adalah daftar *token* dan *state final* yang mengeluarkannya.

Token	Final State
ASSIGN_OPERATOR	137
SEMICOLON	138
COMMA	139
COLON	136
DOT	103
LPARENTHESIS	120
RPARENTHESIS	140
LBRACKET	141
RBRACKET	142
RANGE_OPERATOR	143

```
Token Other
```



2.2. Program

Untuk pengimplementasian *lexical analyzer*, kami memutuskan untuk menggunakan **Golang** versi 1.24.2 Golang dipilih karena sintaksnya yang mudah dimengerti (*high-level language*), tetapi tetap memiliki kelebihan seperti efisiensi memori, performa yang cepat, dan *build ecosystem* yang stabil.

Project Structure

```
.
├── bin
│   └── pslex
├── cmd
│   └── pslex
│       └── main.go
├── doc
│   ├── Laporan-1-NIG.pdf
│   └── lexer-dfa-full.pdf
├── go.mod
├── README.md
├── src
│   ├── datatype
│   │   └── datatype.go
│   ├── dfa
│   │   ├── dfa.go
│   │   └── loader_json.go
│   ├── io
│   │   ├── input.go
│   │   └── output.go
│   ├── lexer
│   │   └── lexer.go
│   └── rules
│       └── tokenizer.json
└── test
    └── milestone-1
        ├── input-1.pas
        ├── input-2.pas
        ├── input-3.pas
        ├── input-4.pas
        ├── input-5.pas
        ├── output-1.txt
        ├── output-2.txt
        ├── output-3.txt
        ├── output-4.txt
        └── output-5.txt
```

2.2.1. datatype.go

```go
package datatype

type TokenType int

const (
    KEYWORD TokenType = iota
```

```
    IDENTIFIER
    ARITHMETIC_OPERATOR
    RELATIONAL_OPERATOR
    LOGICAL_OPERATOR
    ASSIGN_OPERATOR
    NUMBER
    CHAR_LITERAL
    STRING_LITERAL
    SEMICOLON
    COMMA
    COLON
    DOT
    LPARENTHESIS
    RPARENTHESIS
    LBRACKET
    RBRACKET
    RANGE_OPERATOR
    COMMENT
)

type Token struct {
    Type   TokenType
    Lexeme string // substring aslinya
    Line   int    // posisi awal token (untuk error/report)
    Col    int
}

func (t TokenType) String() string {
    names := [...]string{
        "KEYWORD", "IDENTIFIER", "ARITHMETIC_OPERATOR",
"RELATIONAL_OPERATOR", "LOGICAL_OPERATOR",
        "ASSIGN_OPERATOR", "NUMBER", "CHAR_LITERAL", "STRING_LITERAL",
"SEMICOLON", "COMMA", "COLON",
        "DOT", "LPARENTHESIS", "RPARENTHESIS", "LBRACKET", "RBRACKET",
"RANGE_OPERATOR",
        "COMMENT",
    }
    if int(t) < 0 || int(t) >= len(names) {
        return "UNKNOWN"
    }
```

```
    return names[t]
}
```

Program datatype.go berfungsi untuk mendefinisikan tipe dan bentuk data dari setiap token yang akan dipakai di lexer. Enum TokenType berfungsi untuk mendefinisikan kategori token yang bisa dihasilkan lexer. TokenType juga dipakai lexer saat memetakan label DFA. Struct Token berfungsi untuk menyimpan informasi - informasi penting suatu token seperti TokenType, Lexeme, dan posisi awal token. Lalu fungsi String() yang bertugas untuk mencetak TokenType menjadi string yang nantinya akan dicetak sebagai output.

2.2.2. loader_json.go

```
package dfa

import (
    "encoding/json"
    "fmt"
    "os"
)

type rawTransition struct {
    From  int    `json:"from"`
    Input string `json:"input"`
    To    int    `json:"to"`
}

type rawFinal struct {
    State  int    `json:"state"`
    Output string `json:"output"`
}

type rawSpec struct {
    States      int             `json:"states"`
    Start       int             `json:"start"`
    FinalArray  []rawFinal      `json:"final"`
    Transitions []rawTransition `json:"transitions"`
}
```

```go
func LoadJSON(path string) (*DFA, error) {
    b, err := os.ReadFile(path)
    if err != nil {
        return nil, err
    }

    var spec rawSpec
    if err := json.Unmarshal(b, &spec); err != nil {
        return nil, err
    }

    d := &DFA{
        Start:  State(spec.Start),
        Finals: make(map[State]string),
        Trans:  make(map[State]map[rune]State),
    }

    for _, f := range spec.FinalArray {
        d.Finals[State(f.State)] = f.Output
    }

    for _, t := range spec.Transitions {
        runes := []rune(t.Input)
        if len(runes) != 1 {
            return nil, fmt.Errorf("input '%s' harus 1 rune", t.Input)
        }
        if d.Trans[State(t.From)] == nil {
            d.Trans[State(t.From)] = make(map[rune]State)
        }
        d.Trans[State(t.From)][runes[0]] = State(t.To)
    }

    d.Reset()
    return d, nil
}
```

Program loader_json.go bertugas untuk membaca rule dari mesin DFA yang bertipe file json. Program ini memiliki struct rawTransition, rawFinal, dan rawSpec yang berfungsi untuk

menyimpan informasi - informasi penting terkait rule mesin DFA seperti state asal, state tujuan, karakter input, dan state finale.

2.2.3. dfa.go

```go
package dfa

type State int
type DFA struct {
    Start  State
    Finals map[State]string
    Trans  map[State]map[rune]State
    curr   State
}

func (d *DFA) Reset() {
    d.curr = d.Start
}

func (d *DFA) State() State {
    return d.curr
}

func (d *DFA) IsFinal(s State) (string, bool) {
    lab, ok := d.Finals[s]
    return lab, ok
}

func (d *DFA) Step(r rune) (State, bool) { // Buat ngecek next statenya apa
    next, ok := d.Trans[d.curr][r]
    return next, ok
}

func (d *DFA) Advance(r rune) bool { // Pindah State
    if next, ok := d.Trans[d.curr][r]; ok {
        d.curr = next
        return true
    }
    return false
}
```

Program dfa.go berfungsi sebagai struktur dan operasi dari mesin DFA yang nantinya dipakai lexer untuk jalan di atas input, rune demi rune. Struct DFA berfungsi untuk menyimpan state awal dari DFA, himpunan state final beserta label tokennya, tabel transisi deterministik, dan posisi state saat ini. Fungsi Reset() berfungsi untuk mengset curr kembali ke start. Fungsi State() berfungsi untuk mengembalikan state saat ini. Fungsi IsFinals(s State) (string, bool) berfungsi untuk mengecek apakah s adalah state final. Fungsi Step(r rune) (State, bool) berfungsi untuk melihat transisi dari curr jika membaca rune r, tanpa mengubah curr. Fungsi Advance(r rune) bool berfungsi untuk melangkah jika ada state berikutnya, jika tidak ada state berikutnya maka akan mengembalikan false.

2.2.4. input.go

```go
package io

import (
    "os"
    "unicode/utf8"
)

type RuneReader struct {
    buf      []byte
    off      int
    line     int
    col      int
    filePath string
}

func NewRuneReaderFromFile(path string) (*RuneReader, error) {
    b, err := os.ReadFile(path)

    if err != nil {
        return nil, err
    }

    return &RuneReader{buf: b, line: 1, col: 1, filePath: path}, nil
}

func (r *RuneReader) EOF() bool {
    return r.off >= len(r.buf)
```

```
}

func (r *RuneReader) Offset() int {
    return r.off
}

func (r *RuneReader) Pos() (int, int) {
    return r.line, r.col
}

func (r *RuneReader) FilePath() string {
    return r.filePath
}

func (r *RuneReader) Slice(start, end int) string {
    if start < 0 {
        start = 0
    }

    if end > len(r.buf) {
        end = len(r.buf)
    }

    if start > end {
        start, end = end, start
    }

    return string(r.buf[start:end])
}

type Snapshot struct{ off, line, col int } // Buat nyimpen posisi baca saat
ini

func (r *RuneReader) Snapshot() Snapshot {
    return Snapshot{r.off, r.line, r.col}
}

func (r *RuneReader) Restore(s Snapshot) {
    r.off, r.line, r.col = s.off, s.line, s.col
}
```

```
func (r *RuneReader) Seek(off int) {
    r.off = off
}

func (r *RuneReader) Peek() rune {
    if r.EOF() {
        return 0
    }

    ch, _ := utf8.DecodeRune(r.buf[r.off:])
    return ch
}

func (r *RuneReader) Read() (rune, bool) {
    if r.EOF() {
        return 0, false
    }

    ch, w := utf8.DecodeRune(r.buf[r.off:])
    r.off += w

    switch ch {
    case '\r':
        if !r.EOF() && r.buf[r.off] == '\n' {
            r.off++
        }

        r.line++
        r.col = 1
    case '\n':
        r.line++
        r.col = 1
    default:
        r.col++
    }

    return ch, true
}
```

Program input.go berfungsi untuk membaca kode pascal per karakter sambil melacak offset byte, baris, dan kolom. Struct RuneReader berfungsi sebagai pembaca sumber untuk membaca per-rune dari kode pascal. Fungsi Offsite() int berfungsi untuk membaca indeks byte saat ini. Fungsi Pos() (line, col) berfungsi untuk membaca posisi saat ini. Fungsi Slice(start, end) string berfungsi untuk mengambil substring langsung dari buffer berdasarkan byte offset. Ini dipakai untuk menghasilkan Lexeme tanpa membangun ulang karakter satu per satu. Fungsi Snapshot() dan Restore() berfungsi untuk menyimpan dan mengembalikan off, line, dan col. Lexer akan memakai fungsi tersebut untuk rollback ke ujung longest match. Fungsi Peek() rune berfungsi untuk mengintip rune berikutnya tanpa menggeser posisi. Fungsi Read() (rune, bool) berfungsi untuk membaca rune berikutnya dan menggerakan posisi.

2.2.5. lexer.go

```
package lexer

import (
    "fmt"
    "strings"

    "github.com/Azzkaaaa/NIG-Tubes-IF2224/src/datatype"
    "github.com/Azzkaaaa/NIG-Tubes-IF2224/src/dfa"
    iox "github.com/Azzkaaaa/NIG-Tubes-IF2224/src/io"
)

type Lexer struct {
    d *dfa.DFA
    r *iox.RuneReader
}

func New(d *dfa.DFA, r *iox.RuneReader) *Lexer {
    return &Lexer{d: d, r: r}
}

func isWS(r rune) bool {
    return r == ' ' || r == '\t' || r == '\n' || r == '\r' || r == '\f' || r
== 0
}

func (lx *Lexer) ScanAll() ([]datatype.Token, []error) {
```

```
    var toks []datatype.Token
    var errs []error

    for !lx.r.EOF() {
        startOff := lx.r.Offset()
        startLine, startCol := lx.r.Pos()
        startSnap := lx.r.Snapshot()

        lx.d.Reset()

        lastOkOff := -1
        lastOkLabel := ""
        lastSnap := startSnap

        for !lx.r.EOF() {
            ch := lx.r.Peek()
            if !lx.d.Advance(ch) {
                break
            }
            lx.r.Read()

            if lab, ok := lx.d.IsFinal(lx.d.State()); ok {
                lastOkOff = lx.r.Offset()
                lastOkLabel = lab
                lastSnap = lx.r.Snapshot()
            }
        }

        if lastOkOff != -1 {
            lx.r.Restore(lastSnap)
            lex := lx.r.Slice(startOff, lastOkOff)

            tt := mapLabel(lastOkLabel)
            // if tt == datatype.IDENTIFIER {
            //  if _, ok := datatype.Keywords[strings.ToLower(lex)]; ok {
            //      tt = datatype.KEYWORD
            //  }
            // }

            lex = strings.Trim(lex, " \t\r\n\f")
```

```
            switch tt {
            case datatype.STRING_LITERAL, datatype.CHAR_LITERAL,
datatype.COMMENT:
            default:
                lex = strings.ToLower(lex)
            }

            if lex != "" || tt == datatype.STRING_LITERAL || tt ==
datatype.CHAR_LITERAL || tt == datatype.COMMENT {
                toks = append(toks, datatype.Token{
                    Type:   tt,
                    Lexeme: lex,
                    Line:   startLine,
                    Col:    startCol,
                })
            }
            continue
        }

        if ch, ok := lx.r.Read(); ok {
            if !isWS(ch) {
                errs = append(errs, fmt.Errorf("unrecognized %q at %d:%d",
ch, startLine, startCol))
            }
        }
    }

    return toks, errs
}

func mapLabel(label string) datatype.TokenType {
    switch label {
    case "KEYWORD":
        return datatype.KEYWORD
    case "IDENTIFIER":
        return datatype.IDENTIFIER
    case "NUMBER":
        return datatype.NUMBER
    case "CHAR_LITERAL":
```

```
        return datatype.CHAR_LITERAL
    case "STRING_LITERAL":
        return datatype.STRING_LITERAL
    case "ARITHMETIC_OPERATOR":
        return datatype.ARITHMETIC_OPERATOR
    case "RELATIONAL_OPERATOR":
        return datatype.RELATIONAL_OPERATOR
    case "LOGICAL_OPERATOR":
        return datatype.LOGICAL_OPERATOR
    case "ASSIGN_OPERATOR":
        return datatype.ASSIGN_OPERATOR
    case "RANGE_OPERATOR":
        return datatype.RANGE_OPERATOR
    case "SEMICOLON":
        return datatype.SEMICOLON
    case "COMMA":
        return datatype.COMMA
    case "COLON":
        return datatype.COLON
    case "DOT":
        return datatype.DOT
    case "LPARENTHESIS":
        return datatype.LPARENTHESIS
    case "RPARENTHESIS":
        return datatype.RPARENTHESIS
    case "LBRACKET":
        return datatype.LBRACKET
    case "RBRACKET":
        return datatype.RBRACKET
    case "COMMENT":
        return datatype.COMMENT
    default:
        return datatype.IDENTIFIER
    }
}
```

Program lexer.go berfungsi untuk mengubah kode pascal menjadi deretan kode. Program lexer.go akan memakai mesin DFA untuk mengecek mana saja yang token. Program lexer.go akan membaca input RuneReader untuk mendapatkan potongan karakter per karakter dari

kode pascal. Struct Lexer berfungsi untuk menyatukan mesin DFA dan pembaca input. Fungsi ScanAll() adalah fungsi inti untuk memparsing kode pascal menjadi deretan token. Fungsi ini akan mengembalikan slice token dan slice error. Program akan start dari offset byte awal lalu akan menyimpan offset, line, dan col yang nantinya akan dipakai untuk rollback. Lalu program akan melakukan loop untuk parsing kode menjadi deretan token. Fungsi mapLabel(label string) berfungsi untuk memetakan label string dari DFA json.

2.2.6. output.go

```
package io

import (
    "bufio"
    "fmt"
    "os"

    "github.com/Azzkaaaa/NIG-Tubes-IF2224/src/datatype"
)

func PrintTokens(tokens []datatype.Token) {
    for _, t := range tokens {
        fmt.Printf("%s(%s)\n", t.Type.String(), t.Lexeme)
    }
}

func PrintErrors(errs []error) {
    for _, e := range errs {
        fmt.Fprintln(os.Stderr, e)
    }
}

func WriteTokensAndErrorsToFile(path string, tokens []datatype.Token, errs
[]error) error {
    f, err := os.Create(path)
    if err != nil {
        return err
    }
    defer f.Close()

    w := bufio.NewWriter(f)
    for _, t := range tokens {
```

```
        if _, err := fmt.Fprintf(w, "%s(%s)\n", t.Type.String(), t.Lexeme);
err != nil {
            return err
        }
    }
    for _, e := range errs {
        if _, err := fmt.Fprintln(w, e); err != nil {
            return err
        }
    }
    return w.Flush()
}
```

Program output.go berfungsi untuk menghasilkan output deretan token dari kode pascal. Fungsi WriteTokensToFile(path string, tokens []datatype.Token) error berfungsi untuk mencetak deretan token ke dalam file teks.

2.2.7. main.go

```
package main

import (
    "flag"
    "fmt"
    "log"
    "os"

    "github.com/Azzkaaaa/NIG-Tubes-IF2224/src/dfa"
    iox "github.com/Azzkaaaa/NIG-Tubes-IF2224/src/io"
    "github.com/Azzkaaaa/NIG-Tubes-IF2224/src/lexer"
)

func main() {
    rules := flag.String("rules", "src/rules/tokenizer.json", "path ke DFA
JSON")
    in := flag.String("input", "", "path file sumber")
    out := flag.String("out", "", "opsional: file output token")
    flag.Parse()
```

```
    if *in == "" {
        fmt.Fprintln(os.Stderr, "missing --input <file>")
        os.Exit(2)
    }

    d, err := dfa.LoadJSON(*rules)
    if err != nil {
        log.Fatal(err)
    }

    rr, err := iox.NewRuneReaderFromFile(*in)
    if err != nil {
        log.Fatal(err)
    }

    tokens, errs := lexer.New(d, rr).ScanAll()

    iox.PrintTokens(tokens)

    for _, e := range errs {
        fmt.Fprintln(os.Stderr, e)
    }

    if *out != "" {
        if err := iox.WriteTokensAndErrorsToFile(*out, tokens, errs); err !=
nil {
            log.Fatal(err)
        }
    }
}
```

Program main.go berfungsi untuk menjalankan lexer dan menghasilkan output dari lexer yang berupa deretan token dari kode pascal. Program akan membuat import dan mengsetup flags dari file DFA json sebagai rules dan file kode pascal sebagai input. Program akan mengload rules DFA dari json dan akan membaca kode pascal sebagai RuneReader. Lalu program akan memanggil fungsi ScanAll() untuk melakukan parsing kode pascal menjadi deretan token.

2.2.8. Format Rule

Berikut adalah format file rule yang ditulis dalam json.

tokenizer.json

```
{
    “states”: <int>,
    “final”: [
        {
            “state”: <int>,
            “output”: <string>
        },
        ...
    ],
    “transitions”: [
        {
            “from”: <int>,
            “input”: <char>,
            “to”: <int>
        },
        ...
    ]
}
```

2.3. Alur Kerja Program

Proses dimulai ketika file sumber kode PASCAL dibaca oleh modul `RuneReader` dari package `io`. Modul ini berfungsi untuk membaca input per karakter (`rune`) sambil melacak posisi baris dan kolom agar kesalahan dapat dilaporkan secara akurat. Setiap karakter yang dibaca kemudian dikirimkan ke mesin Deterministic Finite Automaton (DFA) yang didefinisikan dalam package dfa. Mesin DFA memanfaatkan data transisi yang dimuat dari berkas JSON melalui fungsi `LoadJSON()` pada `loader_json.go`, sehingga seluruh aturan tokenisasi (seperti state, transisi, dan state final) dapat dimodifikasi tanpa mengubah kode utama.

Selanjutnya, modul Lexer yang terdapat pada lexer.go mengintegrasikan pembaca input (`RuneReader`) dan mesin DFA untuk menjalankan proses tokenisasi. Lexer membaca karakter demi karakter dari input, memeriksa setiap transisi DFA, dan mendeteksi apakah rangkaian karakter tertentu membentuk token yang valid. Jika DFA mencapai final state, lexer mengidentifikasi jenis token berdasarkan label keluaran (misalnya KEYWORD, IDENTIFIER, NUMBER, dan sebagainya), kemudian menyimpannya dalam struktur Token yang didefinisikan pada datatype.go. Struktur ini memuat tipe token, isi lexeme asli, serta posisi baris dan kolom awal token di dalam kode sumber.

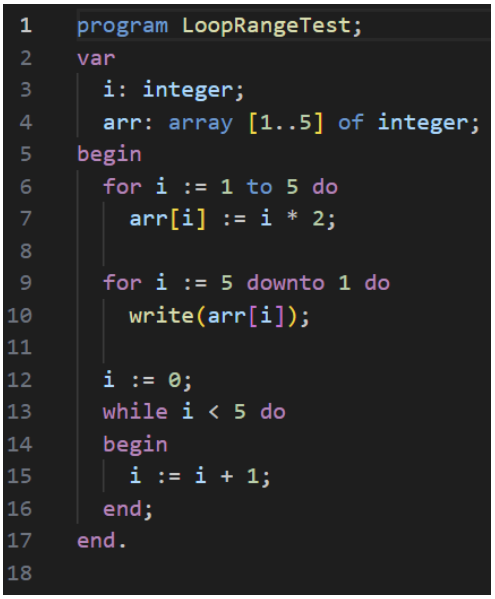
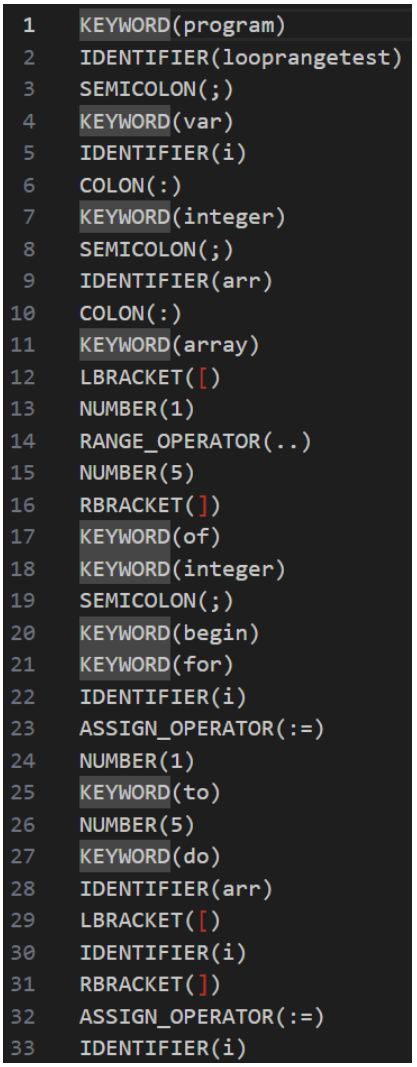
Apabila lexer menemui urutan karakter yang tidak sesuai dengan aturan DFA, maka program akan mencatatnya sebagai error leksikal, lengkap dengan informasi lokasi kesalahan. Setelah seluruh karakter selesai diproses, lexer menghasilkan daftar token beserta daftar error (jika ada) sebagai hasil akhir analisis leksikal.

Token-token ini kemudian siap digunakan pada tahap berikutnya dari proses kompilasi, yaitu analisis sintaksis. Dengan desain modular yang memisahkan fungsi pembacaan input, pengelolaan DFA, dan analisis token, alur kerja program menjadi terstruktur, efisien, dan mudah diperluas untuk pengembangan compiler secara menyeluruh.

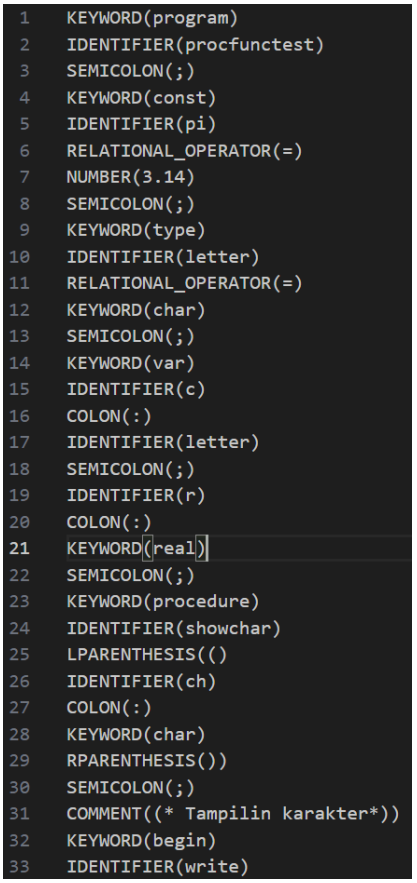
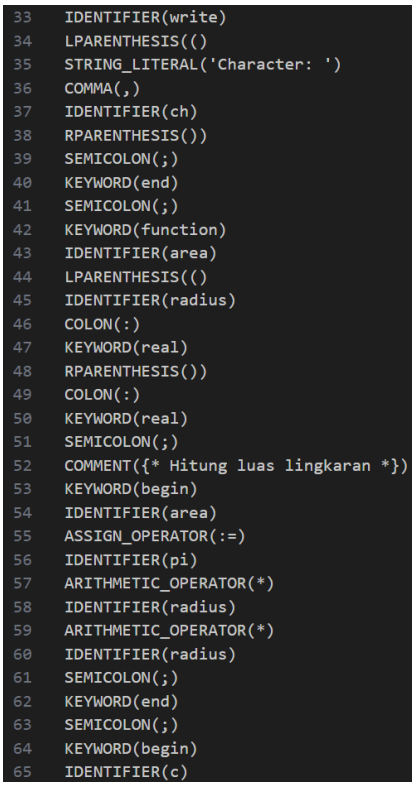
3. Pengujian

No	Input	Output	Ket.
1.	input-1.pas test\milestone-1 1 program ArithmeticTest; 1 var 2 a, b, sum: integer; 3 begin 4 a := 10; 5 b := 5; 6 sum := a + b * 2 - 3 div 2; 7 write('Result: ', sum); 8 end. 9	output-1.txt test\milestone-1 1 KEYWORD(program) 1 IDENTIFIER(arithmetictest) 2 SEMICOLON(;) 3 KEYWORD(var) 4 IDENTIFIER(a) 5 COMMA(,) 6 IDENTIFIER(b) 7 COMMA(,) 8 IDENTIFIER(sum) 9 COLON(:) 10 KEYWORD(integer) 11 SEMICOLON(;) 12 KEYWORD(begin) 13 IDENTIFIER(a) 14 ASSIGN_OPERATOR(:=) 15 NUMBER(10) 16 SEMICOLON(;) 17 IDENTIFIER(b) 18 ASSIGN_OPERATOR(:=) 19 NUMBER(5) 20 SEMICOLON(;) 21 IDENTIFIER(sum) 22 ASSIGN_OPERATOR(:=) 23 IDENTIFIER(a) 24 ARITHMETIC_OPERATOR(+) 25 IDENTIFIER(b) 26 ARITHMETIC_OPERATOR(*) 27 NUMBER(2) 28 ARITHMETIC_OPERATOR(-) 29 NUMBER(3) 30 ARITHMETIC_OPERATOR(div) 31 NUMBER(2) 32 SEMICOLON(;) 33 IDENTIFIER(write) 34 LPARENTHESIS(() 35 STRING_LITERAL('Result: ') 36 COMMA(,) 37 IDENTIFIER(sum) 38 RPARENTHESIS()) 39 SEMICOLON(;) 40 KEYWORD(end) 41 DOT(.)	Tokens: program var integer begin end := + - * div ; , . string_literal identifier number

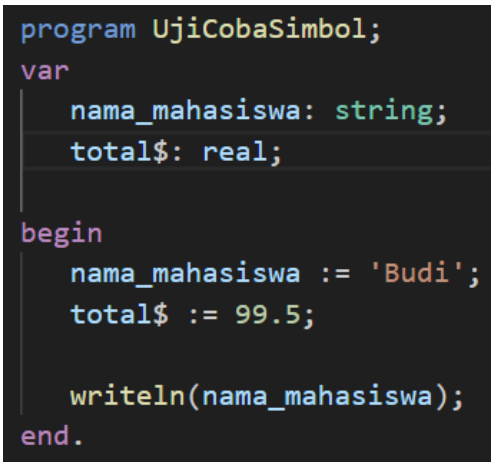
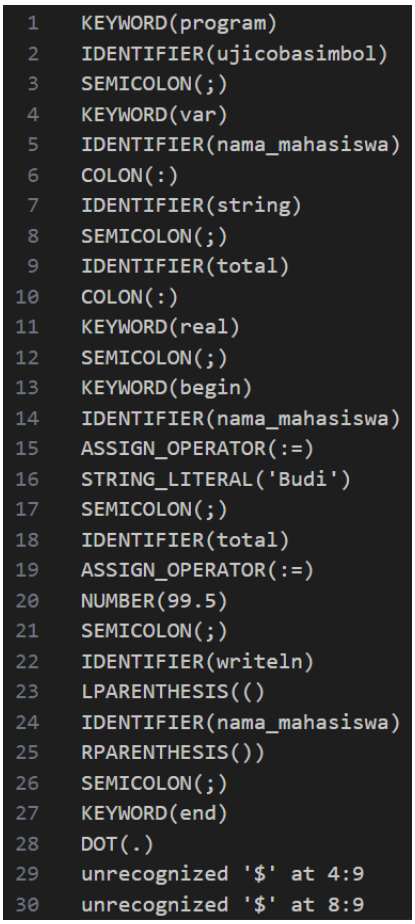
No	Input	Output	Ket.
2.	input-2.pas test\milestone-1 1 program LogicalTest; 1 var 2 x, y: integer; 3 flag: boolean; 4 begin 5 x := 3; 6 y := 7; 7 flag := (x < y) and not (x = 0) or (y ≥ 10); 8 if flag then 9 write('Condition true') 10 else 11 write('Condition false'); 12 end. 13	output-2.txt test\milestone-1 1 KEYWORD(program) 1 IDENTIFIER(logicaltest) 2 SEMICOLON(;) 3 KEYWORD(var) 4 IDENTIFIER(x) 5 COMMA(,) 6 IDENTIFIER(y) 7 COLON(:) 8 KEYWORD(integer) 9 SEMICOLON(;) 10 IDENTIFIER(flag) 11 COLON(:) 12 KEYWORD(boolean) 13 SEMICOLON(;) 14 KEYWORD(begin) 15 IDENTIFIER(x) 16 ASSIGN_OPERATOR(:=) 17 NUMBER(3) 18 SEMICOLON(;) 19 IDENTIFIER(y) 20 ASSIGN_OPERATOR(:=) 21 NUMBER(7) 22 SEMICOLON(;) 23 IDENTIFIER(flag) 24 ASSIGN_OPERATOR(:=) 25 LPARENTHESIS(() 26 IDENTIFIER(x) 27 RELATIONAL_OPERATOR(<) 28 IDENTIFIER(y) 29 RPARENTHESIS()) 30 LOGICAL_OPERATOR(and) 31 LOGICAL_OPERATOR(not) 32 LPARENTHESIS(() 33 IDENTIFIER(x) 34 RELATIONAL_OPERATOR(=) 35 NUMBER(0) 36 RPARENTHESIS()) 37 LOGICAL_OPERATOR(or) 38 LPARENTHESIS(() 39 IDENTIFIER(y) 40 RELATIONAL_OPERATOR(≥) 41 NUMBER(10) 42 RPARENTHESIS()) 43 SEMICOLON(;) 44 KEYWORD(if) 45 IDENTIFIER(flag) 46 KEYWORD(then) 47 IDENTIFIER(write) 48 LPARENTHESIS(() 49 STRING_LITERAL('Condition true') 50 RPARENTHESIS()) 51 KEYWORD(else) 52 IDENTIFIER(write) 53 LPARENTHESIS(() 54 STRING_LITERAL('Condition false') 55 RPARENTHESIS()) 56 SEMICOLON(;) 57 KEYWORD(end) 58 DOT(.)	Tokens: if then else boolean and or not < = >= ( ) := . ; . string_literal

No	Input	Output	Ket.
3.			Tokens: for to downto while do array of [ ] .. < := * ; . begin end identifier number

No	Input	Output	Ket.
		34 ARITHMETIC_OPERATOR(*) 35 NUMBER(2) 36 SEMICOLON(;) 37 KEYWORD(for) 38 IDENTIFIER(i) 39 ASSIGN_OPERATOR(:=) 40 NUMBER(5) 41 KEYWORD(downto) 42 NUMBER(1) 43 KEYWORD(do) 44 IDENTIFIER(write) 45 LPARENTHESIS(() 46 IDENTIFIER(arr) 47 LBRACKET([) 48 IDENTIFIER(i) 49 RBRACKET(]) 50 RPARENTHESIS()) 51 SEMICOLON(;) 52 IDENTIFIER(i) 53 ASSIGN_OPERATOR(:=) 54 NUMBER(0) 55 SEMICOLON(;) 56 KEYWORD(while) 57 IDENTIFIER(i) 58 RELATIONAL_OPERATOR(<) 59 NUMBER(5) 60 KEYWORD(do) 61 KEYWORD(begin) 62 IDENTIFIER(i) 63 ASSIGN_OPERATOR(:=) 64 IDENTIFIER(i) 65 ARITHMETIC_OPERATOR(+) 65 ARITHMETIC_OPERATOR(+) 66 NUMBER(1) 67 SEMICOLON(;) 68 KEYWORD(end) 69 SEMICOLON(;) 70 KEYWORD(end) 71 DOT(.)	

No	Input	Output	Ket.
4.		 	Tokens: procedure function const type char real := * . : ; ( ) string_literal char_literal identifier number comment

No	Input	Output	Ket.
		66 ASSIGN_OPERATOR(:=) 67 CHAR_LITERAL('A') 68 SEMICOLON(;) 69 IDENTIFIER(r) 70 ASSIGN_OPERATOR(:=) 71 NUMBER(25e1) 72 SEMICOLON(;) 73 IDENTIFIER(showchar) 74 LPARENTHESIS(() 75 IDENTIFIER(c) 76 RPARENTHESIS()) 77 SEMICOLON(;) 78 IDENTIFIER(write) 79 LPARENTHESIS(() 80 STRING_LITERAL('Area = ') 81 COMMA(,) 82 IDENTIFIER(area) 83 LPARENTHESIS(() 84 IDENTIFIER(r) 85 RPARENTHESIS()) 86 RPARENTHESIS()) 87 SEMICOLON(;) 88 KEYWORD(end) 89 DOT(.)	

No	Input	Output	Ket.
5.			Error by Invalid Identifier

4. Kesimpulan dan Saran

4.1. Kesimpulan

Berdasarkan eksperimen dan analisis yang dilakukan sebelumnya, dapat disimpulkan bahwa:

- Perancangan dan implementasi lexical analyzer (lexer) berbasis Deterministic Finite Automata (DFA) untuk bahasa pemrograman Pascal-S dengan bahasa Go (Golang) berhasil melakukan proses pemindaian dan pengenalan token dari kode uji secara efisien dan akurat.
- Pengujian pada berbagai file kode Pascal-S berhasil menunjukkan bahwa lexer dapat berfungsi dengan benar karena memecah kode sumber menjadi token-token yang sesuai.
- Selain mengenali token yang valid, lexer juga terbukti mampu mendeteksi dan melaporkan kesalahan leksikal seperti penggunaan karakter yang tidak valid dalam sebuah identifier.

4.2. Saran

- **Integrasi dengan Parser**
  Sebagai "Milestone 1", langkah logis berikutnya adalah melanjutkan pengembangan ke tahap *syntax analysis*. Disarankan untuk membuat sebuah *parser* yang dapat menggunakan *output* token dari lexer ini untuk membangun struktur sintaksis program, seperti *Abstract Syntax Tree* (AST).
- **Peningkatan Pelaporan Kesalahan**
  Mekanisme pelaporan kesalahan dapat ditingkatkan agar lebih informatif. Misalnya, tidak hanya melaporkan karakter yang tidak dikenali, tetapi juga memberikan saran atau konteks yang lebih jelas kepada pengguna mengenai letak dan kemungkinan penyebab kesalahan.
- **Otomatisasi Pembuatan DFA**
  Untuk pengembangan di masa depan, pertimbangkan untuk menggunakan *lexer generator* (seperti Lex atau Flex) yang dapat menghasilkan mesin DFA secara otomatis dari definisi *regular expression*. Ini dapat menyederhanakan proses pemeliharaan dan modifikasi aturan leksikal.
- **Optimasi Kinerja**
  Meskipun Golang sudah dipilih karena performanya, dapat dilakukan analisis kinerja (*profiling*) pada lexer saat memproses file kode sumber yang sangat besar untuk mengidentifikasi dan mengoptimalkan potensi *bottleneck*.

Lampiran

Pembagian Tugas

Anggota	Tugas	Persentase
13523123 Rhio Bimo Prakoso S	Fotografer, Konsumsi, Femboy Generator, Testing, dan Laporan	15
13523137 M Aulia Azka	Engine dan Testing	30
13523161 Arlow Emmanuel Hergara	Lexer (Graph and Code)	25
13523162 Fahcriza Ahmad Setiyono	Engine dan Testing	15
13523163 Filbert Engyo	Testing dan Laporan	15

Pranala Repository
https://github.com/Azzkaaaa/NIG-Tubes-IF2224

Pranala Diagram Transisi DFA
https://drive.google.com/file/d/1r1EO-N7KdHVs9TItYtjy0mLrYoRxAXwd/view?usp=sharing

Pranala Kode Sumber Diagram Transisi DFA
https://drive.google.com/file/d/1t0ML5Q50P71BOVaKT1SUcDN3aXpPx7O3/view?usp=sharing

Kode Generator DFA
https://colab.research.google.com/drive/1UjtWcLK2uFMhrfAgwIx5wcRFRHCvEUTN?usp=sharing

Daftar Pustaka

https://www.sciencedirect.com/topics/computer-science/deterministic-finite-automaton
https://www.tutorialspoint.com/automata_theory/deterministic_finite_automaton.htm
https://www.geeksforgeeks.org/compiler-design/introduction-of-lexical-analysis/
https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_tokens.htm
http://db.science.uoit.ca/teaching/csci4020u/1_lexical_analysis/1_tokens/
https://www.geeksforgeeks.org/compiler-design/token-patterns-and-lexems/
https://www.slideshare.net/slideshow/relationship-among-token-lexeme-pattern/251260602
https://www.tutorchase.com/answers/a-level/computer-science/what-is-the-role-of-a-lexer-in-a-compiler