IF2224 – Teori Bahasa Formal & Otomata
Laporan Tugas Besar Milestone 2 - 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

1. Landasan Teori

1.1. Syntax Analysis atau *Parser*

Syntax Analysis atau yang biasa disebut Parser, adalah fase kedua dari proses cara kerja *compiler*. Pada proses *scanner* atau *Lexer*, keluaran atau *output* dari proses itu adalah deretan token. Pada proses Parser, parser akan memeriksa apakah token-token tersebut tersusun dalam urutan yang benar sesuai dengan aturan tata bahasa (grammar) dari bahasa sumber. Tujuan utamanya adalah memastikan bentuk program benar, bukan maknanya. Dengan kata lain, parser mengecek "apakah susunan kata - katanya sudah sesuai kaidah", lalu menghasilkan representasi struktur hierarkis yang akan dipakai tahap berikutnya, seperti analisis semantik dan pembuatan intermediate code.

Berbeda dengan proses scanner atau *Lexer* yang keluarannya atau *output*-nya berupa deretan token, proses Parser akan menghasilkan keluaran atau output berupa *Parse Tree*. *Parse Tree* adalah representasi grafis berbentuk pohon dari bagaimana sebuah string atau statement dalam kode sumber diturunkan dari simbol awal grammar. Terdapat dua pendekatan dalam membuat atau *generate Parse Tree* yaitu pendekatan *Top Down Parser* dan *Bottom Up Parser*.

Pada pendekatan *Top Down*, proses Parser memulai membangun pohon dari akar (simbol awal) lalu "membayangkan" bentuk program yang sah menurut aturan, menurunkannya langkah demi langkah hingga cocok dengan token yang datang. Secara konseptual, pendekatan *Top Down* membangun derivasi kiri. Terdapat dua macam pendekatan Top Down yaitu dengan *backtracking* dan *tanpa backtracking*. Jika dengan *backtracking*, algoritma yang biasa digunakan adalah algoritma *brute force*. Jika tanpa *backtracking*, algoritma yang biasa digunakan adalah algoritma *Recursive Descent* dan *Predictive Parser* (LL(1), LL(K)). Kelebihan pendekatan ini adalah transparans. Struktur kode hampir satu banding satu dengan grammar, mudah disisipkan laporan kesalahan di posisi yang tepat, dan mudah dirawat ketika grammar berubah. Kekurangannya, kekuatan pengenalannya lebih terbatas. Jika grammar mengandung ambiguitas atau pola yang secara alami lebih cocok dipecahkan dari bawah (misalnya banyak produksi yang saling tumpang-tindih), pendekatan *Top Down* bisa memerlukan transformasi grammar yang signifikan atau terjerumus ke *backtracking* yang tidak efisien.

Pada pendekatan *Bottom Up*, proses Parser membangun pohon dari daun ke atas menuju akar. Proses ini mencoba "mereduksi" input string kembali menjadi simbol awal. Parser mencari "*handle*", yakni substring input yang persis cocok dengan sisi kanan sebuah produksi pada konteks yang tepat, lalu melakukan reduksi menjadi non-*terminal* pada sisi kirinya. Pada pendekatan ini biasanya menggunakan algoritma LR Parser (LR(0), SLR(1), LALR(1), dan CLR(1)). Keunggulan *Bottom Up* adalah daya ungkapnya luas. Resolusi ambiguitas operator dapat ditangani sistematis lewat deklarasi prioritas dan asosiativitas pada generator parser. Kekurangannya, kode hasil generator cenderung kurang intuitif bila dibandingkan recursive descent, dan ketika terjadi error sintaks, jejaknya bisa lebih sulit dibaca karena keputusan parser tergantung keadaan automata dan tabel parse, bukan aliran fungsi yang eksplisit.

1.2. Context-Free Grammar (CFG)

Context-Free Grammar (CFG) adalah suatu model formal yang mendeskripsikan struktur frasa suatu bahasa pemrograman. Pada dasarnya, *Context-Free Grammar* (CFG) bisa dibayangkan sebagai aturan tata bahasa yang menentukan cara membentuk semua string (kalimat) yang valid dalam sebuah bahasa. *Context-Free Grammar* (CFG) jauh lebih kuat daripada Regular Expression atau Finite Automata karena mampu menangani struktur yang bersifat rekursif atau bersarang. Sebuah CFG terdiri dari *Variables* (V), *Terminals* (T), *Productions* (P), dan *Start Symbol* (S).

Variables (V) adalah himpunan terbatas dari simbol non-*terminal*. Simbol non-*terminal* ini mewakili kategori sintaksis. *Terminals* (T) adalah himpunan terbatas dari simbol *terminal*. Simbol terminal adalah simbol yang yang identik dengan token hasil leksikal (seperti IDENTIFIER, NUMBER, atau token tanda baca). *Productions* (P) adalah himpunan terbatas dari aturan produksi. *Productions* (P) menjelaskan bagaimana sebuah nonterminal dapat diperluas menjadi rangkaian simbol lain. Ini adalah inti dari sebuah CFG. *Symbol* (S) adalah sebuah simbol khusus yang bertugas menjadi titik awal dari semua proses pembentukan string.

1.3. Parse Tree

Seperti pada penjelasan terkait Parser, *Parse Tree* adalah sebuah representasi grafis yang menunjukkan bagaimana sebuah string dapat diturunkan dari simbol awal (start symbol) sebuah CFG. Ini adalah cara visual untuk melacak proses derivasi dan melihat struktur sintaksis dari sebuah string. *Parse Tree* memiliki berbagai keuntungan seperti menunjukan struktur sintaksis. *Parse Tree* secara eksplisit menunjukkan bagaimana komponen - komponen sebuah string dikelompokkan bersama. Lalu Alternatif Derivasi. *Parse Tree* menyajikan informasi yang sama dengan derivasi tetapi dalam format yang lebih intuitif dan mudah dipahami. Lalu bagus dalam mendeteksi ambiguitas. Jika sebuah string dapat menghasilkan lebih dari satu Parse Tree yang valid, maka grammar tersebut bersifat ambigu.

Struktur *Parse Tree* terdiri dari akar, simpul internal, dan daun. Akar atau yang biasa disebut *root* adalah simpul paling atas dari pohon yang dilabeli dengan simbol S. Simpol internal adalah setiap simpul yang bukan daun. Simpul tersebut harus dilabeli dengan simbol V (*Variables*). Daun adalah setiap simpul yang dilabeli simbol T (*terminals*), sebuah *variables*, atau epsilon dan tidak memiliki anak.

1.4. Recursive Descent

Seperti pada penjelasan terkait Parser, *Recursive Descent* adalah salah satu algoritma yang digunakan untuk membuat *Parse Tree* yang menggunakan pendekatan *Top Down*. Ini adalah algoritma yang paling umum dalam membuat *Parse Tree*. Algoritma ini cukup efisien dan tidak memerlukan *backtracking*. Untuk setiap simbol non-*terminal* dalam gambar perlu dibuat prosedur atau fungsinya. Masing - masing fungsi bertugas mencocokan pola sesuai produksi yang bersesuaian. Karena struktur kodenya paralel dengan struktur grammar, pendekatan ini sangat mudah dibaca dan dirawat. Agar berjalan mulus tanpa *backtracking*, *Recursive Descent* biasanya dipakai dalam bentuk prediktif satu

token. Ini berarti grammar harus disiapkan sehingga keputusan alternatif produksi dapat dibuat hanya dengan melihat token berikutnya. Kelebihan utama *Recursive Descent* adalah transparansi dan modularitas. Karena setiap fungsi mencerminkan satu non-*terminal*, perubahan grammar mudah diproyeksikan ke perubahan kode, dan pengecekan atau logging pada titik-titik tertentu dapat disisipkan tanpa mengganggu keseluruhan alur. Di sisi lain, kekurangannya adalah keterbatasan pada kelas grammar yang dapat ditangani tanpa *backtracking.*

Cara kerja dari *Recursive Descent* yang pertama adalah, Proses parsing dimulai dengan memanggil prosedur untuk simbol awal. Lalu yang kedua, setiap prosedur bertanggung jawab untuk "mengenali" bagian dari input string yang cocok dengan non-*terminal* tersebut. Lalu yang ketiga, ketika sebuah prosedur bertemu dengan non-*terminal* lain di aturan produksinya, ia akan memanggil prosedur yang bersesuaian. Lalu yang keempat, ketika bertemu simbol *terminal*, ia akan mencocokkannya dengan token input saat ini dan maju ke token berikutnya.

1.5. Keterkaitan dengan Lexer

Keterkaitan antara lexer (atau scanner) dan parser adalah hubungan fungsional yang sekuensial dan sangat erat dalam analisis kode sumber, yang secara kolektif dikenal sebagai frontend dari kompilator atau interpreter. Lexer bertanggung jawab sebagai tahap pertama, yakni analisis leksikal. Tugas utamanya adalah membaca urutan karakter mentah dari kode sumber, kemudian mengelompokkan karakter-karakter tersebut menjadi satuan-satuan bermakna yang disebut token (seperti pengenal, kata kunci, operator, atau literal), sambil membuang spasi dan komentar yang tidak relevan. Token ini adalah unit dasar tata bahasa sebuah bahasa pemrograman, dan secara efektif, lexer bertindak sebagai pemasok data yang membersihkan dan mengorganisasi input mentah agar siap diproses pada tahap selanjutnya.

Setelah lexer menghasilkan aliran token, giliran parser untuk mengambil alih prosesnya melalui analisis sintaksis. Parser menerima aliran token ini sebagai inputnya, dan kemudian memeriksa apakah urutan token tersebut mematuhi aturan tata bahasa (grammar) yang telah ditetapkan. Jika struktur kode valid, parser akan menyusun representasi hierarkis, biasanya berupa Abstract Syntax Tree (AST), yang menggambarkan struktur logis dari program. Dengan demikian, parser sepenuhnya bergantung pada output token yang bersih dan terstruktur dari lexer untuk menjalankan fungsinya, karena ia tidak dirancang untuk memproses karakter mentah. Keduanya bekerja dalam sebuah simbiosis, di mana lexer menyediakan kosakata (token) dan parser memverifikasi kalimat (struktur sintaksis) program.

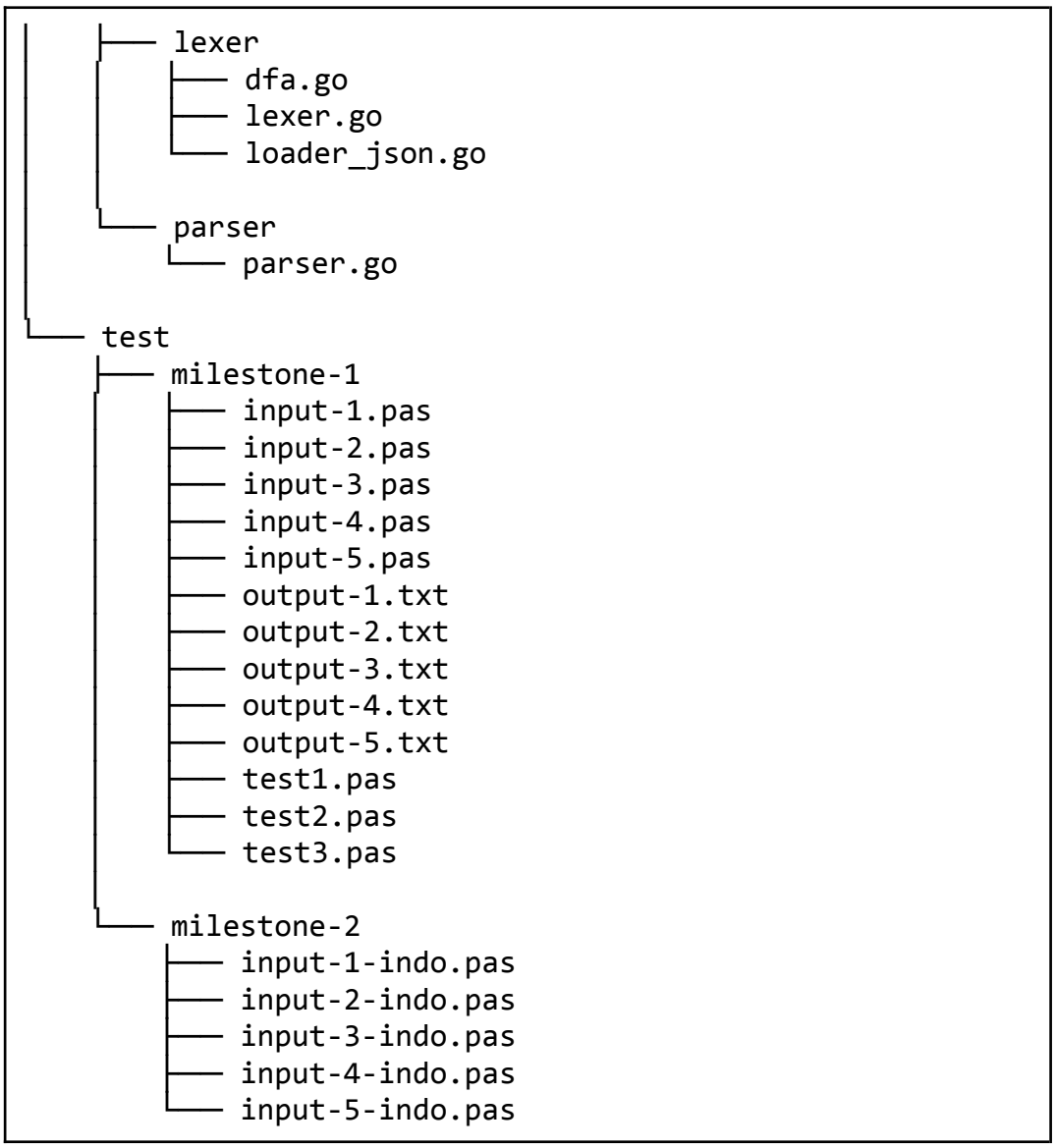
2. Perancangan & Implementasi

2.1. Program

Untuk pengimplementasian *syntax analyzer (parser)*, 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

```
NIG-TUBES-IF2224
├── .gitignore
├── README.md
│
├── bin
│   ├── pslex.exe
│   └── pspar
│
├── config
│   ├── tokenizer.json
│   └── tokenizer_indo.json
│
├── doc
│   ├── .gitignore
│   ├── Laporan-1-NIG.pdf
│   ├── Laporan-2-NIG.pdf
│   └── lexer-dfa-full.pdf
│
├── psc
│   ├── go.mod
│   │
│   ├── cmd
│   │   ├── psc
│   │   │   └── main.go
│   │   │
│   │   ├── pslex
│   │   │   └── main.go
│   │   │
│   │   └── pspar
│   │       └── main.go
│   │
│   ├── common
│   │   └── io.go
│   │
│   ├── datatype
│   │   ├── parsetree.go
│   │   └── token.go
│   │
```



2.1.1. parsetree.go

```go
package datatype

import (
    "fmt"
    "strings"
)

type NodeType int

const (
    PROGRAM_NODE NodeType = iota
```

```
    PROGRAM_HEADER_NODE
    DECLARATION_PART_NODE
    CONST_DECLARATION_PART_NODE
    CONST_DECLARATION_NODE
    TYPE_DECLARATION_PART_NODE
    TYPE_DECLARATION_NODE
    VAR_DECLARATION_PART_NODE
    VAR_DECLARATION_NODE
    IDENTIFIER_LIST_NODE
    TYPE_NODE
    ARRAY_TYPE_NODE
    RANGE_NODE
    SUBPROGRAM_DECLARATION_NODE
    PROCEDURE_DECLARATION_NODE
    FUNCTION_DECLARATION_NODE
    FORMAL_PARAMETER_LIST_NODE
    COMPOUND_STATEMENT_NODE
    STATEMENT_LIST_NODE
    ASSIGNMENT_STATEMENT_NODE
    IF_STATEMENT_NODE
    WHILE_STATEMENT_NODE
    FOR_STATEMENT_NODE
    SUBPROGRAM_CALL_NODE
    PARAMETER_LIST_NODE
    EXPRESSION_NODE
    SIMPLE_EXPRESSION_NODE
    TERM_NODE
    FACTOR_NODE
    RELATIONAL_OPERATOR_NODE
    ADDITIVE_OPERATOR_NODE
    MULTIPLICATIVE_OPERATOR_NODE
    ARRAY_ACCESS_NODE
    TOKEN_NODE
)

type ParseTree struct {
    RootType   NodeType
    TokenValue *Token
    Children   []ParseTree
}
```

```go
func (t NodeType) String() string {
    names := [...]string{
        "<program>",
        "<program-header>",
        "<declaration-part>",
        "<const-declaration-part>",
        "<const-declaration>",
        "<type-declaration-part>",
        "<type-declaration>",
        "<var-declaration-part>",
        "<var-declaration>",
        "<identifier-list>",
        "<type>",
        "<array-type>",
        "<range>",
        "<subprogram-declaration>",
        "<procedure-declaration>",
        "<function-declaration>",
        "<formal-parameter-list>",
        "<compound-statement>",
        "<statement-list>",
        "<assignment-statement>",
        "<if-statement>",
        "<while-statement>",
        "<for-statement>",
        "<procedure/function-call>",
        "<parameter-list>",
        "<expression>",
        "<simple-expression>",
        "<term>",
        "<factor>",
        "<relational-operator>",
        "<additive-operator>",
        "<multiplicative-operator>",
        "<array-access>",
    }

    if int(t) < 0 || int(t) >= len(names) {
        return "UNKNOWN"
```

```
    }

    return names[t]
}

func (t ParseTree) String() string {
    var sb strings.Builder
    t.writeString(&sb, "", true, true)
    return sb.String()
}

func (t ParseTree) writeString(sb *strings.Builder, prefix string, isLast
bool, isRoot bool) {
    if prefix != "" {
        if isLast {
            sb.WriteString(prefix + "└─")
        } else {
            sb.WriteString(prefix + "├─")
        }
    }

    if t.RootType != TOKEN_NODE {
        sb.WriteString(t.RootType.String())
    }
    if t.TokenValue != nil {
        sb.WriteString(t.TokenValue.Type.String())
        sb.WriteString(fmt.Sprintf(" (%s)", t.TokenValue.Lexeme))
    }
    sb.WriteString("\n")

    childPrefix := prefix
    if isRoot || prefix != "" {
        if isLast {
            childPrefix += "  "
        } else {
            childPrefix += "│ "
        }
    }

    for i, child := range t.Children {
```

```
        child.writeString(sb, childPrefix, i == len(t.Children)-1, false)
    }
}
```

Program parsetree.go berfungsi untuk menulis bentuk pohon dari hasil parsing suatu kode program pascal sesuai dengan token dan node yang dihandle oleh fungsi-fungsi dalam parser.go.

2.1.2. parser.go

```
package parser

import (
    "fmt"
    "strings"

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

type Parser struct {
    buffer []dt.Token
    pos    int
}

type ParseError struct {
    buffer   []dt.Token
    Line     int
    Col      int
    Tips     string
    Got      *dt.Token
    Expected []dt.TokenType
}

func reconstruct_source(tokens []dt.Token, line int) string {
    var sb strings.Builder
    currentLine := max(1, line-1)
    currenCol := 1
    foundLine := false
    for _, t := range tokens {
        if t.Line < max(1, line-1) {
            continue
```

```
        }
        if t.Line > line {
            break
        }
        if !foundLine && t.Line == currentLine {
            sb.WriteString(fmt.Sprintf("\n%d: ", t.Line))
            foundLine = true
        } else if t.Line > currentLine {
            sb.WriteString(fmt.Sprintf("\n%d: ", t.Line))
            currenCol = 1
            currentLine = t.Line
        }
        sb.WriteString(strings.Repeat(" ", max(t.Col-currenCol, 0)))
        sb.WriteString(t.Lexeme)
        currenCol += max(t.Col-currenCol, 0) + len(t.Lexeme)

    }
    return sb.String()
}

func (e *ParseError) Error() string {
    expected := ""
    if len(e.Expected) > 0 {
        for i, t := range e.Expected {
            if i > 0 {
                expected += ", "
            }
            expected += t.String()
        }
        expected = fmt.Sprintf("(expected: %s)", expected)
    }

    if e.Tips != "" {
        return fmt.Sprintf(`
Line %d, Col %d
%s
%s
SyntaxError: Unexpected token '%s' %s
%s
`,
```

```
            e.Line, e.Col, reconstruct_source(e.buffer, e.Line),
strings.Repeat(" ", e.Col-1+3)+"^", e.Got.Lexeme, expected, e.Tips)
    } else {
        return fmt.Sprintf(`
Line %d, Col %d
%s
%s
SyntaxError: Unexpected token '%s' %s
`,
            e.Line, e.Col, reconstruct_source(e.buffer, e.Line),
strings.Repeat(" ", e.Col-1+3)+"^", e.Got.Lexeme, expected)
    }
}

func (p *Parser) createParseError(expectedType dt.TokenType, tips string)
error {
    curr := p.buffer[p.pos]
    return &ParseError{
        buffer:   p.buffer,
        Line:     curr.Line,
        Col:      curr.Col,
        Tips:     tips,
        Expected: []dt.TokenType{expectedType},
        Got:      &curr,
    }
}

func (p *Parser) createParseErrorMany(expectedType []dt.TokenType, tips
string) error {
    curr := p.buffer[p.pos]
    return &ParseError{
        buffer:   p.buffer,
        Line:     curr.Line,
        Col:      curr.Col,
        Tips:     tips,
        Expected: expectedType,
        Got:      &curr,
    }
}
```

```go
func New(tokens []dt.Token) *Parser {
    return &Parser{
        buffer: tokens,
        pos:    0,
    }
}

func (p *Parser) peek() *dt.Token {
    return &p.buffer[p.pos]
}

func (p *Parser) consume(expectedType dt.TokenType) *dt.Token {
    curr := p.peek()
    if curr.Type == expectedType {
        p.pos++
        return curr
    }
    return nil
}

func (p *Parser) consumeMany(expectedType []dt.TokenType) *dt.Token {
    curr := p.peek()
    for _, tokenType := range expectedType {
        if curr.Type == tokenType {
            p.pos++
            return curr
        }
    }
    return nil
}

func (p *Parser) consumeExact(expectedType dt.TokenType, expectedLexeme
string) *dt.Token {
    curr := p.peek()
    if curr.Type == expectedType && curr.Lexeme == expectedLexeme {
        p.pos++
        return curr
    }

    return nil
```

```
}

func (p *Parser) match(expectedType dt.TokenType) bool {
    curr := p.peek()
    return curr.Type == expectedType
}

func (p *Parser) matchExact(expectedType dt.TokenType, expectedLexeme
string) bool {
    curr := p.peek()
    return curr.Type == expectedType && curr.Lexeme == expectedLexeme
}

func (p *Parser) Parse() (*dt.ParseTree, error) {
    tree, err := p.parseProgram()
    return tree, err
}

func (p *Parser) parseProgram() (*dt.ParseTree, error) {
    headerTree, err := p.parseProgramHeader()

    if err != nil {
        return nil, err
    }

    declarationTree, err := p.parseDeclarationPart()

    if err != nil {
        return nil, err
    }

    compoundTree, err := p.parseCompoundStatement()
    if err != nil {
        return nil, err
    }

    dotToken := p.consume(dt.DOT)
    if dotToken == nil {
        return nil, p.createParseError(dt.DOT, "program must end with a dot
(.)")
```

```
    }

    if p.pos < len(p.buffer) {
        curr := p.peek()
        return nil, &ParseError{
            buffer: p.buffer,
            Line:   curr.Line,
            Col:    curr.Col,
            Tips:   "unexpected token after program end (.)",
            Got:    curr,
        }
    }

    programTree := dt.ParseTree{
        RootType:   dt.PROGRAM_NODE,
        TokenValue: nil,
        Children: []dt.ParseTree{
            *headerTree,
            *declarationTree,
            *compoundTree,
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: dotToken,
                Children:   make([]dt.ParseTree, 0),
            },
        },
    }

    return &programTree, nil
}

func (p *Parser) parseProgramHeader() (*dt.ParseTree, error) {

    if p.consumeExact(dt.KEYWORD, "program") == nil {
        return nil, p.createParseError(dt.KEYWORD, "all programs must start
with program keyword")
    }

    if p.consume(dt.IDENTIFIER) == nil {
        return nil, p.createParseError(dt.IDENTIFIER, "program name must
```

```
only use alphanumerical characters and underscores")
    }

    if p.consume(dt.SEMICOLON) == nil {
        return nil, p.createParseError(dt.SEMICOLON, "program name must be a
single word and strictly end with ;")
    }

    headerTree := dt.ParseTree{
        RootType:   dt.PROGRAM_HEADER_NODE,
        TokenValue: nil,
        Children: []dt.ParseTree{{
            RootType:   dt.TOKEN_NODE,
            TokenValue: &p.buffer[0],
            Children:   make([]dt.ParseTree, 0),
        }, {
            RootType:   dt.TOKEN_NODE,
            TokenValue: &p.buffer[1],
            Children:   make([]dt.ParseTree, 0),
        }, {
            RootType:   dt.TOKEN_NODE,
            TokenValue: &p.buffer[2],
            Children:   make([]dt.ParseTree, 0),
        }},
    }

    return &headerTree, nil
}

func (p *Parser) parseDeclarationPart() (*dt.ParseTree, error) {
    var err error

    declarationTree := dt.ParseTree{
        RootType:   dt.DECLARATION_PART_NODE,
        TokenValue: nil,
        Children:   make([]dt.ParseTree, 0),
    }

    for err == nil {
        constDeclaration, newErr := p.parseConstDeclarationPart()
```

```
        if constDeclaration == nil && newErr == nil {
            break
        }
        err = newErr
        if err == nil {
            declarationTree.Children = append(declarationTree.Children,
*constDeclaration)
        }
    }

    for err == nil {
        typeDeclaration, newErr := p.parseTypeDeclarationPart()
        if typeDeclaration == nil && newErr == nil {
            break
        }
        err = newErr
        if err == nil {
            declarationTree.Children = append(declarationTree.Children,
*typeDeclaration)
        }
    }

    for err == nil {
        varDeclaration, newErr := p.parseVarDeclarationPart()
        if varDeclaration == nil && newErr == nil {
            break
        }
        err = newErr
        if err == nil {
            declarationTree.Children = append(declarationTree.Children,
*varDeclaration)
        }
    }

    for err == nil {
        subprogramDeclaration, newErr := p.parseSubprogramDeclaration()

        if subprogramDeclaration == nil && newErr == nil {
            break
        }
```

```
        err = newErr
        if err == nil {
            declarationTree.Children = append(declarationTree.Children,
*subprogramDeclaration)
        }
    }

    return &declarationTree, err
}

func (p *Parser) parseConstDeclarationPart() (*dt.ParseTree, error) {
    expectedConst := p.consumeExact(dt.KEYWORD, "konstanta")
    if expectedConst == nil {
        return nil, nil
    }

    constDeclaration := dt.ParseTree{
        RootType:   dt.CONST_DECLARATION_PART_NODE,
        TokenValue: nil,
        Children: []dt.ParseTree{
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedConst,
                Children:   nil,
            },
        },
    }

    if !p.match(dt.IDENTIFIER) {
        return nil, p.createParseError(dt.IDENTIFIER, "expected at least one
const definition")
    }

    for p.match(dt.IDENTIFIER) {
        constDefintion, err := p.parseConstDeclaration()

        if err != nil {
            return nil, err
        }
```

```
        constDeclaration.Children = append(constDeclaration.Children,
            *constDefintion,
        )
    }
    return &constDeclaration, nil
}

func (p *Parser) parseConstDeclaration() (*dt.ParseTree, error) {

    identifier := p.consume(dt.IDENTIFIER)

    if identifier == nil {
        return nil, p.createParseError(dt.IDENTIFIER, "expected identifier
on const definition")
    }

    expectedEqual := p.consumeExact(dt.RELATIONAL_OPERATOR, "=")
    if expectedEqual == nil {
        return nil, p.createParseError(dt.RELATIONAL_OPERATOR, "expected =
after const identifier")
    }

    value := p.consumeMany([]dt.TokenType{dt.CHAR_LITERAL,
dt.STRING_LITERAL, dt.NUMBER})
    if value == nil {
        return nil, p.createParseErrorMany([]dt.TokenType{dt.CHAR_LITERAL,
dt.STRING_LITERAL, dt.NUMBER}, "expected literal value for const
definition")
    }

    expectedSemicolon := p.consume(dt.SEMICOLON)
    if expectedSemicolon == nil {
        return nil, p.createParseError(dt.SEMICOLON, "const definition must
end with semicolon")
    }

    constDefintion := dt.ParseTree{
        RootType:   dt.CONST_DECLARATION_NODE,
        TokenValue: nil,
```

```
        Children: []dt.ParseTree{
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: identifier,
                Children:   nil,
            },
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedEqual,
                Children:   nil,
            },
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: value,
                Children:   nil,
            },
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedSemicolon,
                Children:   nil,
            },
        },
    }

    return &constDefintion, nil
}

func (p *Parser) parseTypeDeclarationPart() (*dt.ParseTree, error) {
    expectedType := p.consumeExact(dt.KEYWORD, "tipe")
    if expectedType == nil {
        return nil, nil
    }

    typeDeclarationPart := dt.ParseTree{
        RootType:   dt.TYPE_DECLARATION_PART_NODE,
        TokenValue: nil,
        Children: []dt.ParseTree{
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedType,
```

```
                Children:   nil,
            },
        },
    }

    if !p.match(dt.IDENTIFIER) {
        return nil, p.createParseError(dt.IDENTIFIER, "expected at least one
type declaration")
    }

    for p.match(dt.IDENTIFIER) {
        typeDeclaration, err := p.parseTypeDeclaration()

        if err != nil {
            return nil, err
        }

        typeDeclarationPart.Children = append(typeDeclarationPart.Children,
            *typeDeclaration,
        )
    }
    return &typeDeclarationPart, nil
}

func (p *Parser) parseTypeDeclaration() (*dt.ParseTree, error) {

    identifier := p.consume(dt.IDENTIFIER)

    if identifier == nil {
        return nil, p.createParseError(dt.IDENTIFIER, "expected identifier
on type declaration")
    }

    expectedEqual := p.consumeExact(dt.RELATIONAL_OPERATOR, "=")
    if expectedEqual == nil {
        return nil, p.createParseError(dt.RELATIONAL_OPERATOR, "expected =
after type identifier")
    }

    parsedType, err := p.parseType()
```

```
    if err != nil {
        return nil, err
    }

    expectedSemicolon := p.consume(dt.SEMICOLON)
    if expectedSemicolon == nil {
        return nil, p.createParseError(dt.SEMICOLON, "type declaration must
end with semicolon")
    }

    typeDeclaration := dt.ParseTree{
        RootType:   dt.TYPE_DECLARATION_NODE,
        TokenValue: nil,
        Children: []dt.ParseTree{
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: identifier,
                Children:   nil,
            },
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedEqual,
                Children:   nil,
            },
            *parsedType,
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedSemicolon,
                Children:   nil,
            },
        },
    }

    return &typeDeclaration, nil
}

func (p *Parser) parseVarDeclarationPart() (*dt.ParseTree, error) {
    expectedVar := p.consumeExact(dt.KEYWORD, "variabel")
    if expectedVar == nil {
        return nil, nil
```

```
    }

    varDeclaration := dt.ParseTree{
        RootType:   dt.VAR_DECLARATION_PART_NODE,
        TokenValue: nil,
        Children: []dt.ParseTree{
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedVar,
                Children:   nil,
            },
        },
    }

    if !p.match(dt.IDENTIFIER) {
        return nil, p.createParseError(dt.IDENTIFIER, "expected at least one
variable declaration")
    }

    for p.match(dt.IDENTIFIER) {
        variableDeclaration, err := p.parseVarDeclaration()

        if err != nil {
            return nil, err
        }

        varDeclaration.Children = append(varDeclaration.Children,
            *variableDeclaration,
        )
    }
    return &varDeclaration, nil
}

func (p *Parser) parseVarDeclaration() (*dt.ParseTree, error) {

    identifier, err := p.parseIdentifierList()

    if err != nil {
        return nil, err
    }
```

```
    expectedColon := p.consume(dt.COLON)
    if expectedColon == nil {
        return nil, p.createParseError(dt.COLON, "")
    }

    parsedType, err := p.parseType()

    if err != nil {
        return nil, err
    }

    expectedSemicolon := p.consume(dt.SEMICOLON)
    if expectedSemicolon == nil {
        return nil, p.createParseError(dt.SEMICOLON, "variable declaration
must end with semicolon")
    }

    variableDeclaration := dt.ParseTree{
        RootType:   dt.VAR_DECLARATION_NODE,
        TokenValue: nil,
        Children: []dt.ParseTree{
            *identifier,
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedColon,
                Children:   nil,
            },
            *parsedType,
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedSemicolon,
                Children:   nil,
            },
        },
    }

    return &variableDeclaration, nil
}
```

```go
func (p *Parser) parseIdentifierList() (*dt.ParseTree, error) {

    expectedIdentifier := p.consume(dt.IDENTIFIER)
    if expectedIdentifier == nil {
        return nil, p.createParseError(dt.IDENTIFIER, "")
    }

    identifierListTree := dt.ParseTree{
        RootType:   dt.IDENTIFIER_LIST_NODE,
        TokenValue: nil,
        Children: []dt.ParseTree{{
            RootType:   dt.TOKEN_NODE,
            TokenValue: expectedIdentifier,
            Children:   make([]dt.ParseTree, 0),
        }},
    }

    for {
        expectedComma := p.consume(dt.COMMA)
        if expectedComma == nil {
            break
        }
        expectedIdentifier := p.consume(dt.IDENTIFIER)
        if expectedIdentifier == nil {
            return nil, p.createParseError(dt.IDENTIFIER, "expected
identifier after comma")
        }
        identifierListTree.Children = append(identifierListTree.Children,
            dt.ParseTree{
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedComma,
                Children:   make([]dt.ParseTree, 0),
            },
            dt.ParseTree{
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedIdentifier,
                Children:   make([]dt.ParseTree, 0),
            },
        )
    }
```

```
    return &identifierListTree, nil
}

func (p *Parser) parseType() (*dt.ParseTree, error) {
    typeTree := dt.ParseTree{
        RootType:   dt.TYPE_NODE,
        TokenValue: nil,
        Children:   make([]dt.ParseTree, 1),
    }

    if p.match(dt.IDENTIFIER) {
        typeTree.Children[0] = dt.ParseTree{
            RootType:   dt.TOKEN_NODE,
            TokenValue: p.consume(dt.IDENTIFIER),
            Children:   make([]dt.ParseTree, 0),
        }
    } else {
        if !p.match(dt.KEYWORD) {
            return nil, p.createParseError(dt.KEYWORD, "expected type")
        }

        switch p.peek().Lexeme {
        case "integer":
            fallthrough
        case "real":
            fallthrough
        case "boolean":
            fallthrough
        case "char":
            typeTree.Children[0] = dt.ParseTree{
                RootType:   dt.TOKEN_NODE,
                TokenValue: p.consume(dt.KEYWORD),
                Children:   make([]dt.ParseTree, 0),
            }
        case "larik":
            fallthrough
        case "array":
            arrayTypeTree, err := p.parseArrayType()
```

```
            if err != nil {
                return nil, err
            }

            typeTree.Children[0] = *arrayTypeTree
        }
    }

    return &typeTree, nil
}

func (p *Parser) parseArrayType() (*dt.ParseTree, error) {

    expectedLarik := p.consumeExact(dt.KEYWORD, "larik")
    if expectedLarik == nil {
        return nil, p.createParseError(dt.KEYWORD, "expected larik keyword")
    }

    expectedLB := p.consume(dt.LBRACKET)
    if expectedLB == nil {
        return nil, p.createParseError(dt.LBRACKET, "expected [ after
larik")
    }

    rangeTree, err := p.parseRange()

    if err != nil {
        return nil, err
    }

    expectedRB := p.consume(dt.RBRACKET)
    if expectedRB == nil {
        return nil, p.createParseError(dt.RBRACKET, "expected ] after larik
range")
    }

    expectedDari := p.consumeExact(dt.KEYWORD, "dari")
    if expectedDari == nil {
        return nil, p.createParseError(dt.KEYWORD, "expected 'dari' after
]")
```

```
    }

    typeTree, err := p.parseType()

    if err != nil {
        return nil, err
    }

    arrayTypeTree := dt.ParseTree{
        RootType:   dt.ARRAY_TYPE_NODE,
        TokenValue: nil,
        Children: []dt.ParseTree{{
            RootType:   dt.TOKEN_NODE,
            TokenValue: expectedLarik,
            Children:   make([]dt.ParseTree, 0),
        }, {
            RootType:   dt.TOKEN_NODE,
            TokenValue: expectedLB,
            Children:   make([]dt.ParseTree, 0),
        },
            *rangeTree,
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedRB,
                Children:   make([]dt.ParseTree, 0),
            }, {
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedDari,
                Children:   make([]dt.ParseTree, 0),
            },
            *typeTree,
        },
    }

    return &arrayTypeTree, nil
}

func (p *Parser) parseRange() (*dt.ParseTree, error) {
    expression1, err := p.parseExpression()
    if err != nil {
```

```
        return nil, err
    }

    rangeOperator := p.consume(dt.RANGE_OPERATOR)
    if rangeOperator == nil {
        return nil, p.createParseError(dt.RANGE_OPERATOR, "expected '..' for
range")
    }

    expression2, err := p.parseExpression()
    if err != nil {
        return nil, err
    }

    rangeTree := dt.ParseTree{
        RootType:   dt.RANGE_NODE,
        TokenValue: nil,
        Children: []dt.ParseTree{
            *expression1,
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: rangeOperator,
                Children:   make([]dt.ParseTree, 0),
            },
            *expression2,
        },
    }

    return &rangeTree, nil
}

func (p *Parser) parseStatement() (*dt.ParseTree, error) {
    if p.matchExact(dt.KEYWORD, "mulai") {
        return p.parseCompoundStatement()
    }
    if p.matchExact(dt.KEYWORD, "jika") {
        return p.parseIfStatement()
    }
    if p.matchExact(dt.KEYWORD, "selama") {
        return p.parseWhileStatement()
```

```
    }
    if p.matchExact(dt.KEYWORD, "untuk") {
        return p.parseForStatement()
    }
    if p.match(dt.IDENTIFIER) {
        if p.pos+1 < len(p.buffer) && (p.buffer[p.pos+1].Type ==
dt.ASSIGN_OPERATOR || p.buffer[p.pos+1].Type == dt.LBRACKET) {
            return p.parseAssignmentStatement()
        } else {
            return p.parseSubprogramCall()
        }
    }
    return nil, p.createParseErrorMany(
        []dt.TokenType{dt.KEYWORD, dt.IDENTIFIER},
        "expected a statement (jika, selama, untuk, mulai, or identifier)",
    )
}

func (p *Parser) parseSubprogramDeclaration() (*dt.ParseTree, error) {
    procTree, err := p.parseProcedureDeclaration()
    if err != nil {
        return nil, err
    }
    if procTree != nil {
        return &dt.ParseTree{
            RootType: dt.SUBPROGRAM_DECLARATION_NODE,
            Children: []dt.ParseTree{*procTree},
        }, nil
    }

    funcTree, err := p.parseFunctionDeclaration()
    if err != nil {
        return nil, err
    }
    if funcTree != nil {
        return &dt.ParseTree{
            RootType: dt.SUBPROGRAM_DECLARATION_NODE,
            Children: []dt.ParseTree{*funcTree},
        }, nil
    }
```

```
    return nil, nil
}

func (p *Parser) parseProcedureDeclaration() (*dt.ParseTree, error) {
    prosedurToken := p.consumeExact(dt.KEYWORD, "prosedur")
    if prosedurToken == nil {
        return nil, nil
    }

    identifier := p.consume(dt.IDENTIFIER)
    if identifier == nil {
        return nil, p.createParseError(dt.IDENTIFIER, "expected procedure
name")
    }

    procTree := dt.ParseTree{
        RootType: dt.PROCEDURE_DECLARATION_NODE,
        Children: []dt.ParseTree{
            {RootType: dt.TOKEN_NODE, TokenValue: prosedurToken},
            {RootType: dt.TOKEN_NODE, TokenValue: identifier},
        },
    }

    if p.match(dt.LPARENTHESIS) {
        paramList, err := p.parseFormalParameterList()
        if err != nil {
            return nil, err
        }
        procTree.Children = append(procTree.Children, *paramList)
    }

    semicolon1 := p.consume(dt.SEMICOLON)
    if semicolon1 == nil {
        return nil, p.createParseError(dt.SEMICOLON, "expected ';' after
procedure header")
    }
    procTree.Children = append(procTree.Children, dt.ParseTree{RootType:
dt.TOKEN_NODE, TokenValue: semicolon1})
```

```
    declarations, err := p.parseDeclarationPart()
    if err != nil {
        return nil, err
    }
    procTree.Children = append(procTree.Children, *declarations)

    compoundStmt, err := p.parseCompoundStatement()
    if err != nil {
        return nil, err
    }
    procTree.Children = append(procTree.Children, *compoundStmt)

    semicolon2 := p.consume(dt.SEMICOLON)
    if semicolon2 == nil {
        return nil, p.createParseError(dt.SEMICOLON, "expected ';' after
procedure block")
    }
    procTree.Children = append(procTree.Children, dt.ParseTree{RootType:
dt.TOKEN_NODE, TokenValue: semicolon2})

    return &procTree, nil
}

func (p *Parser) parseFunctionDeclaration() (*dt.ParseTree, error) {
    fungsiToken := p.consumeExact(dt.KEYWORD, "fungsi")
    if fungsiToken == nil {
        return nil, nil
    }

    identifier := p.consume(dt.IDENTIFIER)
    if identifier == nil {
        return nil, p.createParseError(dt.IDENTIFIER, "expected function
name")
    }

    funcTree := dt.ParseTree{
        RootType: dt.FUNCTION_DECLARATION_NODE,
        Children: []dt.ParseTree{
            {RootType: dt.TOKEN_NODE, TokenValue: fungsiToken},
            {RootType: dt.TOKEN_NODE, TokenValue: identifier},
```

```
        },
    }

    if p.match(dt.LPARENTHESIS) {
        paramList, err := p.parseFormalParameterList()
        if err != nil {
            return nil, err
        }
        funcTree.Children = append(funcTree.Children, *paramList)
    }

    colonToken := p.consume(dt.COLON)
    if colonToken == nil {
        return nil, p.createParseError(dt.COLON, "expected ':' for function
return type")
    }

    returnType, err := p.parseType()
    if err != nil {
        return nil, err
    }

    semicolon1 := p.consume(dt.SEMICOLON)
    if semicolon1 == nil {
        return nil, p.createParseError(dt.SEMICOLON, "expected ';' after
function header")
    }

    funcTree.Children = append(funcTree.Children,
        dt.ParseTree{RootType: dt.TOKEN_NODE, TokenValue: colonToken},
        *returnType,
        dt.ParseTree{RootType: dt.TOKEN_NODE, TokenValue: semicolon1},
    )

    declarations, err := p.parseDeclarationPart()
    if err != nil {
        return nil, err
    }
    funcTree.Children = append(funcTree.Children, *declarations)
```

```
    compoundStmt, err := p.parseCompoundStatement()
    if err != nil {
        return nil, err
    }
    funcTree.Children = append(funcTree.Children, *compoundStmt)

    semicolon2 := p.consume(dt.SEMICOLON)
    if semicolon2 == nil {
        return nil, p.createParseError(dt.SEMICOLON, "expected ';' after
function block")
    }
    funcTree.Children = append(funcTree.Children, dt.ParseTree{RootType:
dt.TOKEN_NODE, TokenValue: semicolon2})

    return &funcTree, nil
}

func (p *Parser) parseFormalParameterList() (*dt.ParseTree, error) {
    lpToken := p.consume(dt.LPARENTHESIS)
    if lpToken == nil {
        return nil, p.createParseError(dt.LPARENTHESIS, "expected '(' to
start parameter list")
    }

    paramListTree := dt.ParseTree{
        RootType: dt.FORMAL_PARAMETER_LIST_NODE,
        Children: []dt.ParseTree{
            {RootType: dt.TOKEN_NODE, TokenValue: lpToken},
        },
    }
    if !p.match(dt.RPARENTHESIS) {
        idList, err := p.parseIdentifierList()
        if err != nil {
            return nil, err
        }

        colonToken := p.consume(dt.COLON)
        if colonToken == nil {
            return nil, p.createParseError(dt.COLON, "expected ':' after
identifier list in parameters")
```

```
        }

        typeNode, err := p.parseType()
        if err != nil {
            return nil, err
        }

        paramListTree.Children = append(paramListTree.Children,
            *idList,
            dt.ParseTree{RootType: dt.TOKEN_NODE, TokenValue: colonToken},
            *typeNode,
        )
        for p.match(dt.SEMICOLON) {
            semicolonToken := p.consume(dt.SEMICOLON)
            nextIdList, err := p.parseIdentifierList()
            if err != nil {
                return nil, p.createParseError(dt.IDENTIFIER, "expected
identifier list after ';'")
            }

            nextColonToken := p.consume(dt.COLON)
            if nextColonToken == nil {
                return nil, p.createParseError(dt.COLON, "expected ':' after
identifier list in parameters")
            }

            nextTypeNode, err := p.parseType()
            if err != nil {
                return nil, err
            }

            paramListTree.Children = append(paramListTree.Children,
                dt.ParseTree{RootType: dt.TOKEN_NODE, TokenValue:
semicolonToken},
                *nextIdList,
                dt.ParseTree{RootType: dt.TOKEN_NODE, TokenValue:
nextColonToken},
                *nextTypeNode,
            )
        }
```

```
    }

    rpToken := p.consume(dt.RPARENTHESIS)
    if rpToken == nil {
        return nil, p.createParseError(dt.RPARENTHESIS, "expected ')' to end
parameter list")
    }
    paramListTree.Children = append(paramListTree.Children,
        dt.ParseTree{RootType: dt.TOKEN_NODE, TokenValue: rpToken},
    )

    return &paramListTree, nil
}

func (p *Parser) parseCompoundStatement() (*dt.ParseTree, error) {
    mulaiToken := p.consumeExact(dt.KEYWORD, "mulai")
    if mulaiToken == nil {
        return nil, p.createParseError(dt.KEYWORD, "expected 'mulai'
keyword")
    }

    stmtList, err := p.parseStatementList()
    if err != nil {
        return nil, err
    }

    selesaiToken := p.consumeExact(dt.KEYWORD, "selesai")
    if selesaiToken == nil {
        return nil, p.createParseError(dt.KEYWORD, "expected 'selesai'
keyword to end compound statement")
    }

    compoundTree := dt.ParseTree{
        RootType: dt.COMPOUND_STATEMENT_NODE,
        Children: []dt.ParseTree{
            {RootType: dt.TOKEN_NODE, TokenValue: mulaiToken},
            *stmtList,
            {RootType: dt.TOKEN_NODE, TokenValue: selesaiToken},
        },
    }
```

```
    return &compoundTree, nil
}

func (p *Parser) parseStatementList() (*dt.ParseTree, error) {
    stmt, err := p.parseStatement()
    if err != nil {
        return nil, err
    }

    stmtListTree := dt.ParseTree{
        RootType: dt.STATEMENT_LIST_NODE,
        Children: []dt.ParseTree{*stmt},
    }

    for p.match(dt.SEMICOLON) {
        semicolon := p.consume(dt.SEMICOLON)

        if p.matchExact(dt.KEYWORD, "selesai") {
            stmtListTree.Children = append(stmtListTree.Children,
dt.ParseTree{
                RootType:   dt.TOKEN_NODE,
                TokenValue: semicolon,
            })
            break
        }

        nextStmt, err := p.parseStatement()
        if err != nil {
            return nil, err
        }

        stmtListTree.Children = append(stmtListTree.Children,
            dt.ParseTree{
                RootType:   dt.TOKEN_NODE,
                TokenValue: semicolon,
            },
            *nextStmt,
        )
    }
    return &stmtListTree, nil
```

```
}

func (p *Parser) parseAssignmentStatement() (*dt.ParseTree, error) {
    var lvalue dt.ParseTree

    p.pos++
    if p.match(dt.LBRACKET) {
        p.pos--

        plvalue, err := p.parseArrayAccess()

        if err != nil {
            return nil, err
        }

        lvalue = *plvalue
    } else {
        p.pos--

        identifier := p.consume(dt.IDENTIFIER)
        if identifier == nil {
            return nil, p.createParseError(dt.IDENTIFIER, "expected
identifier for assignment")
        }

        lvalue = dt.ParseTree{
            RootType:   dt.TOKEN_NODE,
            TokenValue: identifier,
        }
    }

    assignOp := p.consume(dt.ASSIGN_OPERATOR)
    if assignOp == nil {
        return nil, p.createParseError(dt.ASSIGN_OPERATOR, "expected ':='
after identifier")
    }

    expr, err := p.parseExpression()
    if err != nil {
        return nil, err
```

```
    }

    assignTree := dt.ParseTree{
        RootType: dt.ASSIGNMENT_STATEMENT_NODE,
        Children: []dt.ParseTree{
            lvalue,
            {RootType: dt.TOKEN_NODE, TokenValue: assignOp},
            *expr,
        },
    }
    return &assignTree, nil
}

func (p *Parser) parseIfStatement() (*dt.ParseTree, error) {
    jikaToken := p.consumeExact(dt.KEYWORD, "jika")
    if jikaToken == nil {
        return nil, p.createParseError(dt.KEYWORD, "expected 'jika'")
    }

    expr, err := p.parseExpression()
    if err != nil {
        return nil, err
    }

    makaToken := p.consumeExact(dt.KEYWORD, "maka")
    if makaToken == nil {
        return nil, p.createParseError(dt.KEYWORD, "expected 'maka' after
if-expression")
    }

    thenStmt, err := p.parseStatement()
    if err != nil {
        return nil, err
    }

    ifTree := dt.ParseTree{
        RootType: dt.IF_STATEMENT_NODE,
        Children: []dt.ParseTree{
            {RootType: dt.TOKEN_NODE, TokenValue: jikaToken},
            *expr,
```

```
            {RootType: dt.TOKEN_NODE, TokenValue: makaToken},
            *thenStmt,
        },
    }

    if p.matchExact(dt.KEYWORD, "selain_itu") {
        selainItuToken := p.consumeExact(dt.KEYWORD, "selain_itu")

        elseStmt, err := p.parseStatement()
        if err != nil {
            return nil, err
        }

        ifTree.Children = append(ifTree.Children,
            dt.ParseTree{RootType: dt.TOKEN_NODE, TokenValue:
selainItuToken},
            *elseStmt,
        )
    }

    return &ifTree, nil
}

func (p *Parser) parseWhileStatement() (*dt.ParseTree, error) {
    selamaToken := p.consumeExact(dt.KEYWORD, "selama")
    if selamaToken == nil {
        return nil, p.createParseError(dt.KEYWORD, "expected 'selama'")
    }

    expr, err := p.parseExpression()
    if err != nil {
        return nil, err
    }

    lakukanToken := p.consumeExact(dt.KEYWORD, "lakukan")
    if lakukanToken == nil {
        return nil, p.createParseError(dt.KEYWORD, "expected 'lakukan' after
while-expression")
    }
```

```
    stmt, err := p.parseStatement()
    if err != nil {
        return nil, err
    }

    whileTree := dt.ParseTree{
        RootType: dt.WHILE_STATEMENT_NODE,
        Children: []dt.ParseTree{
            {RootType: dt.TOKEN_NODE, TokenValue: selamaToken},
            *expr,
            {RootType: dt.TOKEN_NODE, TokenValue: lakukanToken},
            *stmt,
        },
    }
    return &whileTree, nil
}

func (p *Parser) parseForStatement() (*dt.ParseTree, error) {
    untukToken := p.consumeExact(dt.KEYWORD, "untuk")
    if untukToken == nil {
        return nil, p.createParseError(dt.KEYWORD, "expected 'untuk'")
    }

    identifier := p.consume(dt.IDENTIFIER)
    if identifier == nil {
        return nil, p.createParseError(dt.IDENTIFIER, "expected counter
identifier after 'untuk'")
    }

    assignOp := p.consume(dt.ASSIGN_OPERATOR)
    if assignOp == nil {
        return nil, p.createParseError(dt.ASSIGN_OPERATOR, "expected ':='
after for-identifier")
    }

    startExpr, err := p.parseExpression()
    if err != nil {
        return nil, err
    }
```

```
    // (KEYWORD(ke)|KEYWORD(turun-ke))
    var directionToken *dt.Token
    if p.matchExact(dt.KEYWORD, "ke") {
        directionToken = p.consumeExact(dt.KEYWORD, "ke")
    } else if p.matchExact(dt.KEYWORD, "turun_ke") {
        directionToken = p.consumeExact(dt.KEYWORD, "turun_ke")
    }

    if directionToken == nil {
        return nil, p.createParseError(dt.KEYWORD, "expected 'ke' or
'turun_ke' in for loop")
    }

    endExpr, err := p.parseExpression()
    if err != nil {
        return nil, err
    }

    lakukanToken := p.consumeExact(dt.KEYWORD, "lakukan")
    if lakukanToken == nil {
        return nil, p.createParseError(dt.KEYWORD, "expected 'lakukan' in
for loop")
    }

    stmt, err := p.parseStatement()
    if err != nil {
        return nil, err
    }

    forTree := dt.ParseTree{
        RootType: dt.FOR_STATEMENT_NODE,
        Children: []dt.ParseTree{
            {RootType: dt.TOKEN_NODE, TokenValue: untukToken},
            {RootType: dt.TOKEN_NODE, TokenValue: identifier},
            {RootType: dt.TOKEN_NODE, TokenValue: assignOp},
            *startExpr,
            {RootType: dt.TOKEN_NODE, TokenValue: directionToken},
            *endExpr,
            {RootType: dt.TOKEN_NODE, TokenValue: lakukanToken},
            *stmt,
```

```
        },
    }
    return &forTree, nil
}

func (p *Parser) parseSubprogramCall() (*dt.ParseTree, error) {
    identifier := p.consume(dt.IDENTIFIER)
    if identifier == nil {
        return nil, p.createParseError(dt.RANGE_OPERATOR, "expected
function/procedure identifier")
    }

    subprogramCall := dt.ParseTree{
        RootType:   dt.SUBPROGRAM_CALL_NODE,
        TokenValue: nil,
        Children: []dt.ParseTree{
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: identifier,
                Children:   make([]dt.ParseTree, 0),
            },
        },
    }

    if p.match(dt.LPARENTHESIS) {
        lp := p.consume(dt.LPARENTHESIS)
        params, err := p.parseParameterList()
        if err != nil {
            return nil, err
        }
        rp := p.consume(dt.RPARENTHESIS)

        if rp == nil {
            return nil, p.createParseError(dt.RPARENTHESIS, "expected )
after function call")
        }

        subprogramCall.Children = append(subprogramCall.Children,
            dt.ParseTree{
                RootType:   dt.TOKEN_NODE,
```

```
                TokenValue: lp,
                Children:   nil,
            },
            *params,
            dt.ParseTree{
                RootType:   dt.TOKEN_NODE,
                TokenValue: rp,
                Children:   nil,
            },
        )
    }

    return &subprogramCall, nil
}

func (p *Parser) parseParameterList() (*dt.ParseTree, error) {
    expression, err := p.parseExpression()
    if err != nil {
        return nil, err
    }

    parameterListTree := dt.ParseTree{
        RootType:   dt.PARAMETER_LIST_NODE,
        TokenValue: nil,
        Children: []dt.ParseTree{
            *expression,
        },
    }

    for {
        expectedComma := p.consume(dt.COMMA)
        if expectedComma == nil {
            break
        }
        nextExpression, err := p.parseExpression()
        if err != nil {
            return nil, err
        }
        parameterListTree.Children = append(parameterListTree.Children,
            dt.ParseTree{
```

```
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedComma,
                Children:   make([]dt.ParseTree, 0),
            },
            *nextExpression,
        )
    }

    return &parameterListTree, nil
}

func (p *Parser) parseExpression() (*dt.ParseTree, error) {
    simpleExpression, err := p.parseSimpleExpression()

    if err != nil {
        return nil, err
    }

    expressionTree := dt.ParseTree{
        RootType:   dt.EXPRESSION_NODE,
        TokenValue: nil,
        Children: []dt.ParseTree{
            *simpleExpression,
        },
    }

    if p.match(dt.RELATIONAL_OPERATOR) {
        relationalOperator, err := p.parseRelationalOperator()

        if err != nil {
            return nil, p.createParseError(dt.RELATIONAL_OPERATOR, "expected
relational opeartor after simple expression")
        }

        simpleExpression2, err := p.parseSimpleExpression()

        if err != nil {
            return nil, err
        }
```

```
        expressionTree.Children = append(expressionTree.Children,
            *relationalOperator,
            *simpleExpression2,
        )
    }

    return &expressionTree, nil
}

func (p *Parser) parseSimpleExpression() (*dt.ParseTree, error) {
    var expectedArithmeticOperator *dt.Token
    if p.matchExact(dt.ARITHMETIC_OPERATOR, "+") {
        expectedArithmeticOperator = p.consumeExact(dt.ARITHMETIC_OPERATOR,
"+")
    } else if p.matchExact(dt.ARITHMETIC_OPERATOR, "-") {
        expectedArithmeticOperator = p.consumeExact(dt.ARITHMETIC_OPERATOR,
"-")
    } else if p.match(dt.ARITHMETIC_OPERATOR) { // arithmetic operator
selain +,-
        return nil, p.createParseError(dt.ARITHMETIC_OPERATOR, "only + and -
operator are allowed before term")
    }
    simpleExpressionTree := dt.ParseTree{
        RootType:   dt.SIMPLE_EXPRESSION_NODE,
        TokenValue: nil,
        Children:   make([]dt.ParseTree, 0),
    }
    if expectedArithmeticOperator != nil {
        simpleExpressionTree.Children =
append(simpleExpressionTree.Children,
            dt.ParseTree{
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedArithmeticOperator,
                Children:   make([]dt.ParseTree, 0),
            },
        )
    }

    term, err := p.parseTerm()
    if err != nil {
```

```
        return nil, err
    }
    simpleExpressionTree.Children = append(simpleExpressionTree.Children,
        *term,
    )
    for {
        additiveOperator, err := p.parseAdditiveOperator()
        if err != nil {
            break
        }

        nextTerm, err := p.parseTerm()
        if err != nil {
            return nil, err
        }
        simpleExpressionTree.Children =
append(simpleExpressionTree.Children,
            *additiveOperator,
            *nextTerm,
        )
    }
    return &simpleExpressionTree, nil
}

func (p *Parser) parseTerm() (*dt.ParseTree, error) {
    factor, err := p.parseFactor()
    if err != nil {
        return nil, err
    }
    termTree := dt.ParseTree{
        RootType:   dt.TERM_NODE,
        TokenValue: nil,
        Children: []dt.ParseTree{
            *factor,
        },
    }
    for {
        multiplicativeOperator, err := p.parseMultiplicativeOperator()
        if err != nil {
            break
```

```
        }

        nextFactor, err := p.parseFactor()
        if err != nil {
            return nil, err
        }
        termTree.Children = append(termTree.Children,
            *multiplicativeOperator,
            *nextFactor,
        )
    }
    return &termTree, nil
}

func (p *Parser) parseFactor() (*dt.ParseTree, error) {
    factorTree := dt.ParseTree{
        RootType:   dt.FACTOR_NODE,
        TokenValue: nil,
        Children:   make([]dt.ParseTree, 0),
    }
    if p.match(dt.IDENTIFIER) {
        identifier := p.consume(dt.IDENTIFIER)

        if p.match(dt.LPARENTHESIS) {
            p.pos--
            factor, err := p.parseSubprogramCall()
            if err != nil {
                return nil,
p.createParseErrorMany([]dt.TokenType{dt.IDENTIFIER, dt.NUMBER,
dt.CHAR_LITERAL, dt.STRING_LITERAL, dt.LPARENTHESIS}, "cannot parse factor")
            }
            factorTree.Children = append(factorTree.Children,
                *factor,
            )
        } else if p.match(dt.LBRACKET) {
            p.pos--
            element, err := p.parseArrayAccess()
            if err != nil {
                return nil, err
            }
```

```
            factorTree.Children = append(factorTree.Children,
                *element,
            )
        } else {
            factorTree.Children = append(factorTree.Children,
                dt.ParseTree{
                    RootType:   dt.TOKEN_NODE,
                    TokenValue: identifier,
                    Children:   nil,
                },
            )
        }
    } else if p.match(dt.NUMBER) {
        factor := p.consume(dt.NUMBER)
        factorTree.Children = append(factorTree.Children,
            dt.ParseTree{
                RootType:   dt.TOKEN_NODE,
                TokenValue: factor,
                Children:   nil,
            },
        )
    } else if p.match(dt.CHAR_LITERAL) {
        factor := p.consume(dt.CHAR_LITERAL)
        factorTree.Children = append(factorTree.Children,
            dt.ParseTree{
                RootType:   dt.TOKEN_NODE,
                TokenValue: factor,
                Children:   nil,
            },
        )
    } else if p.match(dt.STRING_LITERAL) {
        factor := p.consume(dt.STRING_LITERAL)
        factorTree.Children = append(factorTree.Children,
            dt.ParseTree{
                RootType:   dt.TOKEN_NODE,
                TokenValue: factor,
                Children:   nil,
            },
        )
    } else if p.match(dt.LPARENTHESIS) {
```

```
        lp := p.consume(dt.LPARENTHESIS)
        expr, err := p.parseExpression()
        if err != nil {
            return nil, err
        }
        rp := p.consume(dt.RPARENTHESIS)
        if rp == nil {
            return nil, p.createParseError(dt.RPARENTHESIS, "closing ) not
found after expression")
        }
        factorTree.Children = append(factorTree.Children,
            dt.ParseTree{
                RootType:   dt.TOKEN_NODE,
                TokenValue: lp,
                Children:   nil,
            },
            *expr,
            dt.ParseTree{
                RootType:   dt.TOKEN_NODE,
                TokenValue: rp,
                Children:   nil,
            },
        )
    } else if p.matchExact(dt.LOGICAL_OPERATOR, "tidak") {
        expectedNot := p.consumeExact(dt.LOGICAL_OPERATOR, "tidak")
        factor, err := p.parseFactor()
        if err != nil {
            return nil, err
        }
        factorTree.Children = append(factorTree.Children,
            dt.ParseTree{
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedNot,
                Children:   nil,
            },
            *factor,
        )
    } else {
        return nil, p.createParseErrorMany([]dt.TokenType{dt.IDENTIFIER,
dt.NUMBER, dt.CHAR_LITERAL, dt.STRING_LITERAL, dt.LPARENTHESIS}, "cannot
```

```
parse factor")
    }

    return &factorTree, nil
}

func (p *Parser) parseRelationalOperator() (*dt.ParseTree, error) {
    expectedRelationalOperator := p.consume(dt.RELATIONAL_OPERATOR)

    if expectedRelationalOperator == nil {
        return nil, p.createParseError(dt.RELATIONAL_OPERATOR, "")
    }

    relationalOperator := dt.ParseTree{
        RootType:   dt.RELATIONAL_OPERATOR_NODE,
        TokenValue: nil,
        Children:   []dt.ParseTree{
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedRelationalOperator,
                Children:   make([]dt.ParseTree, 0),
            },
        },
    }

    return &relationalOperator, nil
}

func (p *Parser) parseAdditiveOperator() (*dt.ParseTree, error) {
    var expectedAdditionOperator *dt.Token
    if p.matchExact(dt.ARITHMETIC_OPERATOR, "+") {
        expectedAdditionOperator = p.consumeExact(dt.ARITHMETIC_OPERATOR,
"+")
    } else if p.matchExact(dt.ARITHMETIC_OPERATOR, "-") {
        expectedAdditionOperator = p.consumeExact(dt.ARITHMETIC_OPERATOR,
"-")
    } else if p.matchExact(dt.LOGICAL_OPERATOR, "atau") {
        expectedAdditionOperator = p.consumeExact(dt.LOGICAL_OPERATOR,
"atau")
    } else {
```

```
        return nil,
p.createParseErrorMany([]dt.TokenType{dt.ARITHMETIC_OPERATOR,
dt.LOGICAL_OPERATOR}, "additive operator not found")
    }

    additiveOperator := dt.ParseTree{
        RootType:   dt.ADDITIVE_OPERATOR_NODE,
        TokenValue: nil,
        Children:   []dt.ParseTree{
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedAdditionOperator,
                Children:   make([]dt.ParseTree, 0),
            },
        },
    }

    return &additiveOperator, nil
}

func (p *Parser) parseMultiplicativeOperator() (*dt.ParseTree, error) {
    var expectedMultiplicativeOperator *dt.Token
    if p.matchExact(dt.ARITHMETIC_OPERATOR, "*") {
        expectedMultiplicativeOperator =
p.consumeExact(dt.ARITHMETIC_OPERATOR, "*")
    } else if p.matchExact(dt.ARITHMETIC_OPERATOR, "/") {
        expectedMultiplicativeOperator =
p.consumeExact(dt.ARITHMETIC_OPERATOR, "/")
    } else if p.matchExact(dt.ARITHMETIC_OPERATOR, "bagi") {
        expectedMultiplicativeOperator =
p.consumeExact(dt.ARITHMETIC_OPERATOR, "bagi")
    } else if p.matchExact(dt.ARITHMETIC_OPERATOR, "mod") {
        expectedMultiplicativeOperator =
p.consumeExact(dt.ARITHMETIC_OPERATOR, "mod")
    } else if p.matchExact(dt.LOGICAL_OPERATOR, "dan") {
        expectedMultiplicativeOperator = p.consumeExact(dt.LOGICAL_OPERATOR,
"dan")
    } else {
        return nil,
p.createParseErrorMany([]dt.TokenType{dt.ARITHMETIC_OPERATOR,
```

```
dt.LOGICAL_OPERATOR}, "multiplicative operator not found")
    }

    multiplicativeOperator := dt.ParseTree{
        RootType:   dt.MULTIPLICATIVE_OPERATOR_NODE,
        TokenValue: nil,
        Children:   []dt.ParseTree{
            {
                RootType:   dt.TOKEN_NODE,
                TokenValue: expectedMultiplicativeOperator,
                Children:   make([]dt.ParseTree, 0),
            },
        },
    }

    return &multiplicativeOperator, nil
}

func (p *Parser) parseArrayAccess() (*dt.ParseTree, error) {
    arrayAccess := dt.ParseTree{
        RootType:   dt.ARRAY_ACCESS_NODE,
        TokenValue: nil,
        Children:   make([]dt.ParseTree, 0, 4),
    }

    if !p.match(dt.IDENTIFIER) {
        return nil, p.createParseError(dt.IDENTIFIER, "expected array
identifier")
    }

    arrayAccess.Children = append(arrayAccess.Children, dt.ParseTree{
        RootType:   dt.TOKEN_NODE,
        TokenValue: p.consume(dt.IDENTIFIER),
        Children:   make([]dt.ParseTree, 0),
    })

    if !p.match(dt.LBRACKET) {
        return nil, p.createParseError(dt.LBRACKET, "expected [")
    }
```

```
    arrayAccess.Children = append(arrayAccess.Children, dt.ParseTree{
        RootType:   dt.TOKEN_NODE,
        TokenValue: p.consume(dt.LBRACKET),
        Children:   make([]dt.ParseTree, 0),
    })

    expression, err := p.parseExpression()
    if err != nil {
        return nil, err
    }

    arrayAccess.Children = append(arrayAccess.Children,
        *expression,
    )

    if !p.match(dt.RBRACKET) {
        return nil, p.createParseError(dt.RBRACKET, "expected ]")
    }

    arrayAccess.Children = append(arrayAccess.Children, dt.ParseTree{
        RootType:   dt.TOKEN_NODE,
        TokenValue: p.consume(dt.RBRACKET),
        Children:   make([]dt.ParseTree, 0),
    })

    return &arrayAccess, nil
}
```

File parser.go berisi logika utama yang menangani *parsing* dari token-token. File ini mendefinisikan tipe data parser yang bertanggung jawab melakukan *parsing* serta menghasilkan error dengan format yang rapih dan dapat merekonstruksi bagian program yang salah jika ditemukan dalam proses *parsing*. Parser memiliki fungsi-fungsi bantuan untuk membantu dalam matching token untuk menentukan production rule yang digunakan. Selain itu, parser juga memiliki fungsi-fungsi untuk tiap production rule yang berlaku dalam bahasa Pascal-S dimana setiap fungsi menghasilkan parsetree untuk *production rule* masing-masing. Proses parsing dilakukan secara rekursif mengikuti prinsip *recursive descent parsing* dimana parser memanggil fungsi production rule program yang kemudian memanggil production rule bagian-bagian program lainnya dan seterusnya sehingga terbentuk parsetree lengkap untuk program. Setiap fungsi production rule mampu mendefinisikan aturannya sendiri-sendiri untuk memilih urutan produksi dan pembagian token yang sesuai dengan kebutuhan bahasa Pascal-S.

2.1.3. main.go

```
package main

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

    iox "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/common"
    dt "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/datatype"
    "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/lexer"
    "github.com/Azzkaaaa/NIG-Tubes-IF2224/psc/parser"
)

func main() {
    rules := flag.String("rules", "config/tokenizer.json", "path ke DFA
JSON")
    in := flag.String("input", "", "path file sumber")
    flag.Parse()

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

    d, err := lexer.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()

    for _, e := range errs {
```

```
        fmt.Fprintln(os.Stderr, e)
    }

    if len(errs) > 0 {
        os.Exit(1)
    }

    tokens = slices.Collect(func(yield func(dt.Token) bool) {
        for _, token := range tokens {
            if token.Type != dt.COMMENT {
                if !yield(token) {
                    return
                }
            }
        }
    })

    parseTree, err := parser.New(tokens).Parse()

    if err != nil {
        fmt.Printf("%v", err)
    }

    if parseTree != nil {
        fmt.Println(parseTree.String())
    }

}
```

File main.go berisi fungsi main yang menjadi titik mulainya program. Fungsi main menjalankan lexer yang sudah dibuat pada milestone sebelumya dan meneruskan hasilnya ke parser. Fungsi ini mencetak parsetree yang dihasilkan oleh parser ke standard output jika hasil lexing dan parsing berjalan dengan benar. Jika tidak, fungsi ini mencetak error dari lexer atau error dari parser.

2.2. Grammar yang Digunakan

No	Nama Node	Aturan Produksi
	<program>	program → program-header + program-header + declaration-part + compound-statement

	<program-header>	program-header → KEYWORD(program) + IDENTIFIER + SEMICOLON
	<declaration-part>	declaration-part → const-declaration-part* + type-declaration-part* + var-declaration-part* + subprogram-declaration-part*
	<const-declaration-part>	const-declaration-part → (KEYWORD(konstanta) + const-declaration*)?
	<const-declaration>	const-declaration → KEYWORD(identifier) + RELATIONAL_OPERATOR(=) + (CHAR_LITERAL/STRING_LITERAL /NUMBER) + SEMICOLON
	<type-declaration-part>	type-declaration-part → (KEYWORD(tipe) + type-declaration*)?
	<type-declaration>	type-declaration → IDENTIFIER + RELATIONAL_OPERATOR(=) + type + SEMICOLON
	<var-declaration-part>	var-declaration-part → (KEYWORD(variabel) + var-declaration*)?
	<var-declaration>	var-declaration → identifier-list + COLON + type + SEMICOLON
	<identifier-list>	identifier-list → IDENTIFIER + (COMMA + IDENTIFIER)*
	<type>	type → IDENTIFIER/KEYWORD(integer/real/boolean/char)/array-type
	<array-type>	array-type → KEYWORD(larik) + LBRACKET + range + RBRACKET + KEYWORD(dari) + type
	<range>	range → expression + RANGE_OPERATOR + expression

	<subprogram-declaration>	subprogram-declaration → (procedure-declaration/function-declaration)?
	<procedure-declaration>	procedure-declaration → KEYWORD(prosedur) + IDENTIFIER + formal-parameter-list + SEMICOLON + declaration-part + compound-statement + SEMICOLON
	<function-declaration>	function-declaration → KEYWORD(prosedur) + IDENTIFIER + formal-parameter-list + COLON + type + SEMICOLON + declaration-part + compound-statement + SEMICOLON
	<formal-parameter-list>	formal-parameter-list → LPARENTHESIS + identifier-list + COLON + type + (SEMICOLON + identifier-list + COLON + type)* + RPARENTHESIS
	<compound-statement>	compound-statement → KEYWORD(mulai) + statement-list + KEYWORD(selesai)
	<statement-list>	statement-list → compound-statement/if-statement/while-statement/for-statement/assignment-statement/subprogram-call + (SEMICOLON + compound-statement/if-statement/while-statement/for-statement/assignment-statement/subprogram-call)*
	<assignment-statement>	assignment-statement → array-access/IDENTIFIER + ASSIGN_OPERATOR + expression
	<if-statement>	if-statement → KEYWORD(jika) + expression + KEYWORD(maka) + statement +

		(KEYWORD(selain_itu) + statement)?
	<while-statement>	while-statement → KEYWORD(selama) + expression + KEYWORD(lakukan) + statement
	<for-statement>	for-statement → KEYWORD(untuk) + IDENTIFIER + ASSIGN_OPERATOR + expression + KEYWORD(ke/turun-ke) + expression + KEYWORD(lakukan) + statement
	<subprogram-call>	subprogram-call → IDENTIFIER + (LPARENTHESIS + parameter-list + RPARENTHESIS)
	<parameter-list>	parameter-list → expression (COMMA + expression)*
	<expression>	expression → simple-expression (relational-operator + simple-expression)?
	<simple-expression>	simple-expression → (ARITHMETIC_OPERATOR(+/-))? term (additive-operator + term)
	<term>	term → factor (multiplicative-operator + factor)*
	<factor>	factor → IDENTIFIER / NUMBER / CHAR_LITERAL / STRING_LITERAL / (LPARENTHESIS + expression + RPARENTHESIS) / (LOGICAL_OPERATOR(tidak) + factor) / subprogram-call
	<relational-operator>	relational-operator → IDENTIFIER + RELATIONAL_OPERATOR + IDENTIFIER

	<additive-operator>	additive-operator → IDENTIFIER + ARITHMETIC_OPERATOR('+' / '-')/LOGICAL_OPERATOR(atau) + IDENTIFIER
	<multiplicative-operator>	multiplicative-operator → IDENTIFIER + ARITHMETIC_OPERATOR('*' / '/' / 'bagi' / 'mod')/LOGICAL_OPERATOR(dan) + IDENTIFIER
	<array-access>	array-access → IDENTIFIER + LBRACKET + expression + RBRACKET

2.3. Alur Kerja Program

Proses analisis sintaksis dimulai setelah lexer berhasil menghasilkan daftar token. Dalam fungsi main(), program main.go pertama-tama menyaring daftar token mentah untuk membuang semua token COMMENT menggunakan slices.Collect. Daftar token yang sudah bersih ini kemudian diteruskan ke parser dengan memanggil parser.New(tokens). Panggilan ini menginisialisasi sebuah struct Parser yang bersifat stateful, yang menyimpan seluruh token di dalam buffer dan mengelola posisi token saat ini melalui sebuah indeks pos.

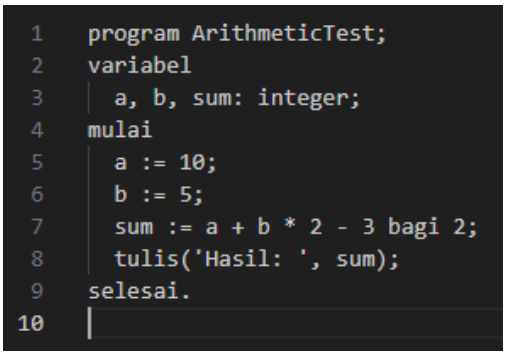
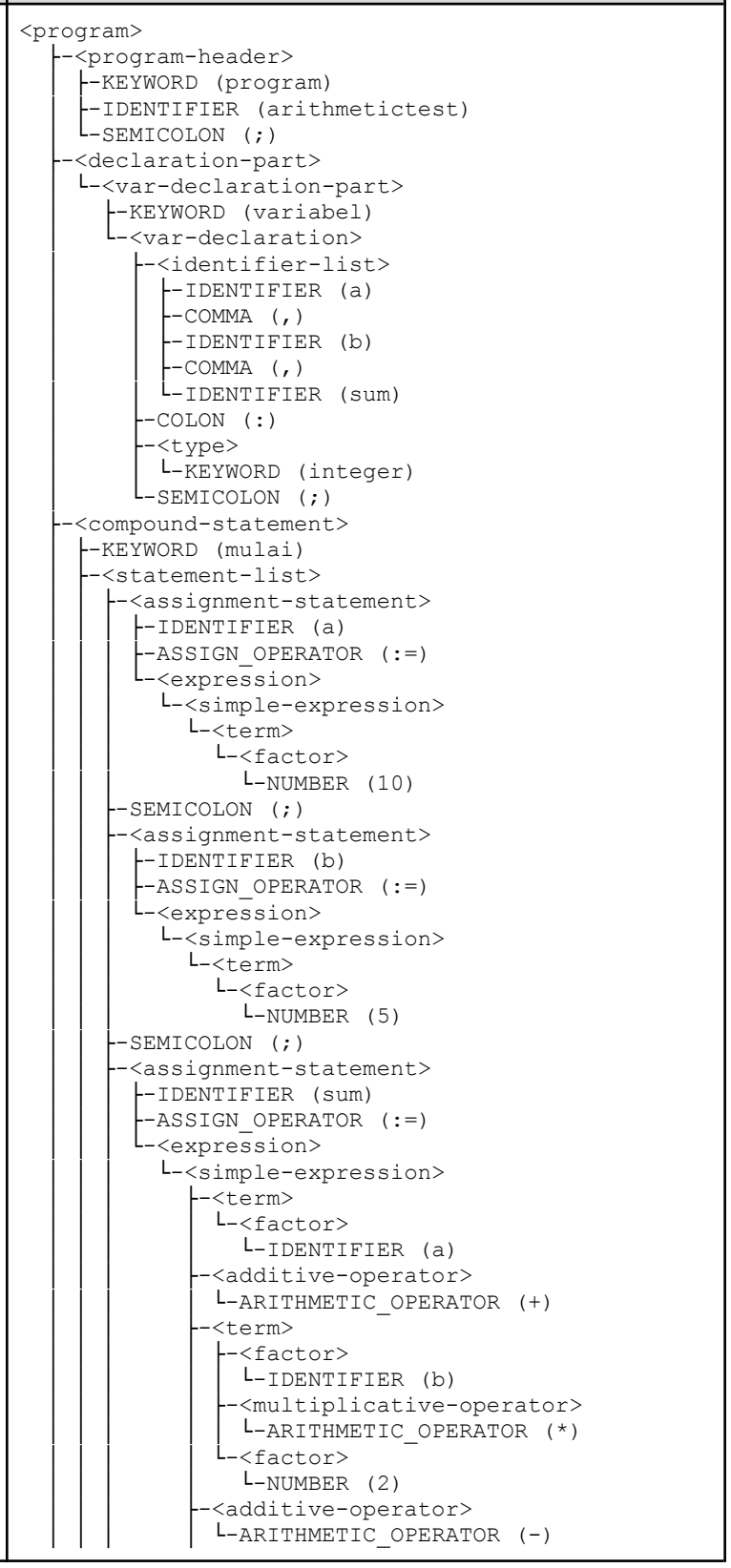
Setelah inisialisasi, main.go memanggil metode Parse(), yang kemudian memicu fungsi utama parser, yaitu parseProgram(). Parser ini diimplementasikan menggunakan strategi Recursive Descent (Penurunan Rekursif). Inti dari desain ini adalah pemetaan satu-ke-satu antara setiap non-terminal dalam tata bahasa (grammar) PASCAL-S dengan sebuah fungsi di dalam parser.go (misalnya, parseProgram(), parseDeclarationPart(), parseStatementList(), parseExpression(), dll.).

Setiap fungsi parse ini bertanggung jawab untuk memvalidasi urutan token sesuai dengan aturan produksinya. Untuk melakukannya, parser menggunakan serangkaian fungsi helper stateful seperti peek() untuk melihat token saat ini tanpa maju, match() untuk memeriksa apakah token saat ini sesuai dengan tipe yang diharapkan, dan consume() untuk memeriksa dan memajukan indeks pos jika sesuai. Fungsi-fungsi ini secara berurutan "memakan" token terminal seperti KEYWORD, SEMICOLON dan melakukan panggilan rekursif ke fungsi parse lain untuk memvalidasi non-terminal.

Selama proses validasi, parser secara dinamis membangun struktur data Parse Tree. Struktur ParseTree ini didefinisikan dalam parsetree.go bersifat rekursif, terdiri dari RootType dan sebuah slice Children. Sesuai dengan spesifikasi, token-token terminal ditambahkan ke pohon sebagai node daun dengan RootType khusus TOKEN_NODE, dan TokenValue yang menunjuk ke struktur Token aslinya.

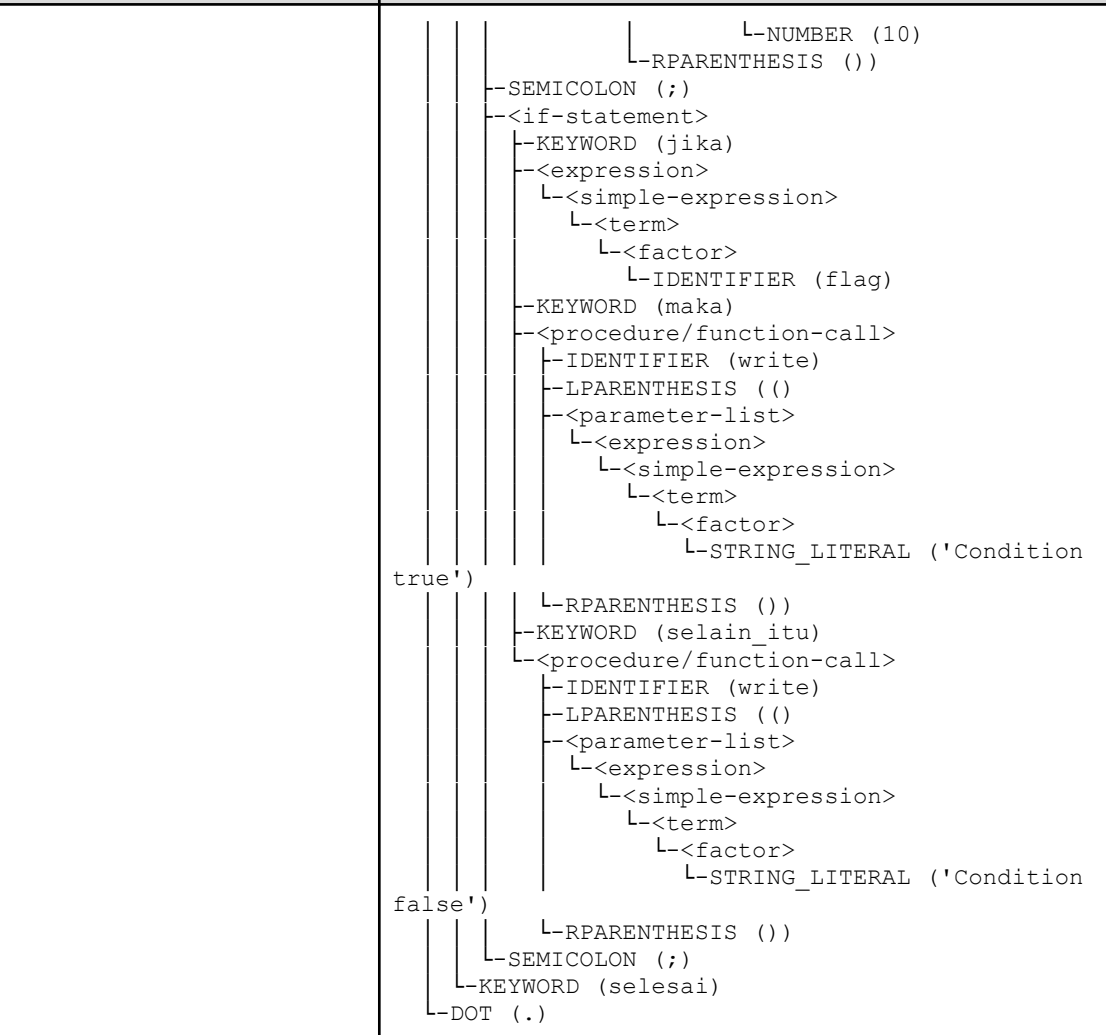
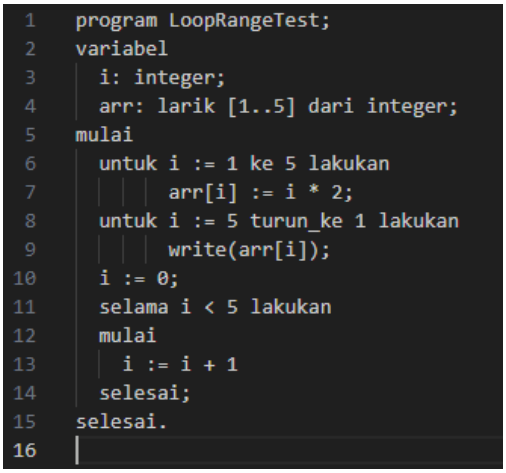
Jika pada titik mana pun sebuah fungsi parser gagal menemukan token yang diharapkan, fungsi tersebut akan berhenti dan membuat ParseError. Struct ParseError ini dari parser.go mengumpulkan informasi kontekstual penting, termasuk Line, Col, token yang Got (didapat), dan Expected (diharapkan), serta mampu merekonstruksi baris kode yang salah. main.go kemudian akan menangkap err ini dan mencetaknya ke console. Jika tidak ada error dan parseProgram() berhasil mencapai akhir buffer token setelah DOT terakhir, ia akan mengembalikan ParseTree yang utuh. main.go kemudian memanggil metode String() dari ParseTree tersebut, yang secara rekursif memformat seluruh pohon menjadi string visual (menggunakan karakter └ dan ├) untuk ditampilkan sebagai keluaran akhir.

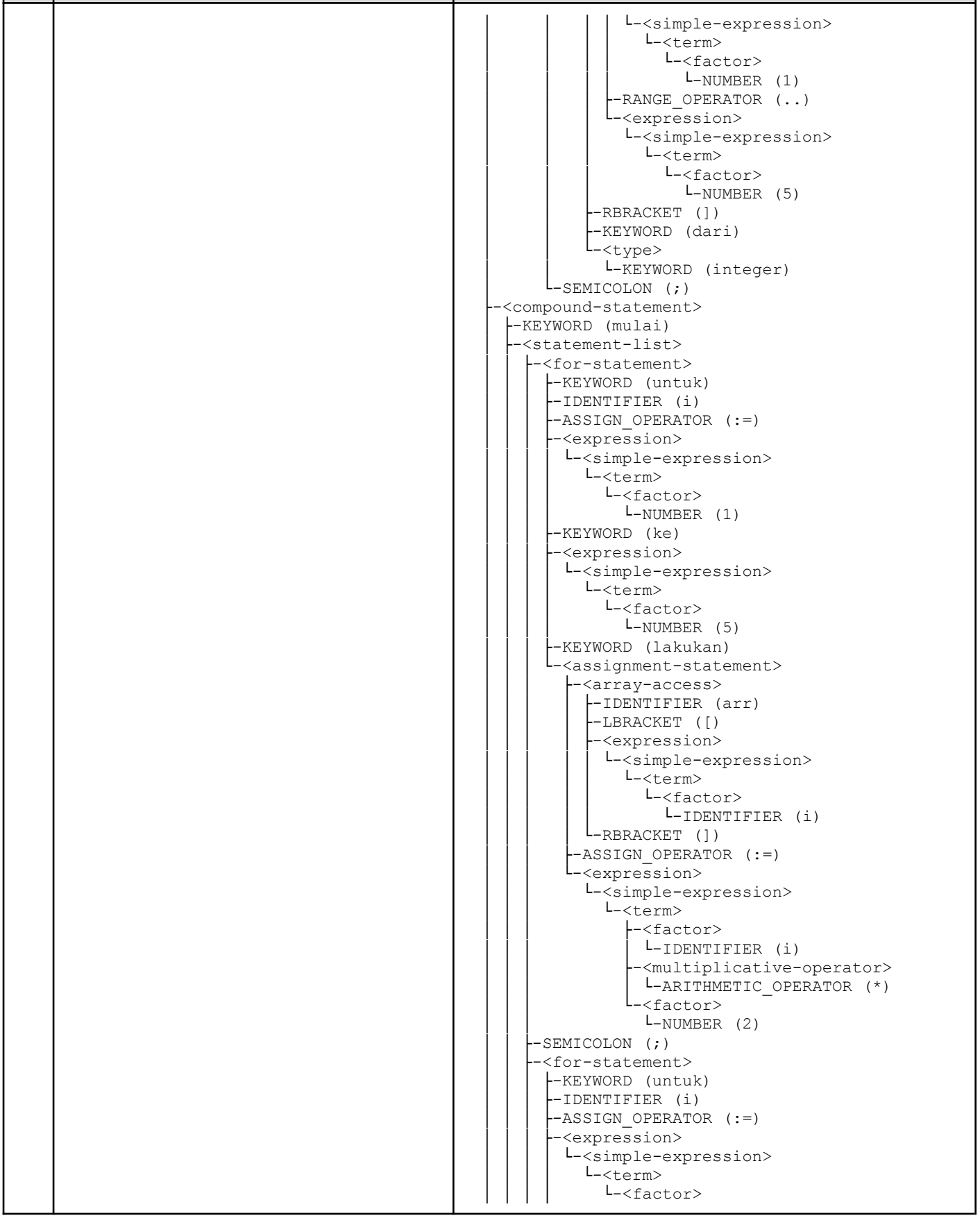
3. Pengujian

No	Input	Output
1.		

No	Input	Output
		<term> ├─<factor> │ └─NUMBER (3) ├─<multiplicative-operator> │ └─ARITHMETIC_OPERATOR (bagi) └─<factor> └─NUMBER (2) ├─SEMICOLON (;) ├─<procedure/function-call> ├─IDENTIFIER (tulis) ├─LPARENTHESIS (() ├─<parameter-list> ├─<expression> └─<simple-expression> └─<term> └─<factor> └─STRING_LITERAL ('Hasil: ') ├─COMMA (,) └─<expression> └─<simple-expression> └─<term> └─<factor> └─IDENTIFIER (sum) └─RPARENTHESIS ()) └─SEMICOLON (;) └─KEYWORD (selesai) └─DOT (.)
2.	``` program LogicalTest; variabel x, y: integer; flag: boolean; mulai x := 3; y := 7; flag := (x < y) dan tidak (x = 0) atau (y >= 10); jika flag maka write('Condition true') selain_itu write('Condition false'); selesai. ```	<program> ├─<program-header> ├─KEYWORD (program) ├─IDENTIFIER (logicaltest) └─SEMICOLON (;) ├─<declaration-part> └─<var-declaration-part> ├─KEYWORD (variabel) ├─<var-declaration> ├─<identifier-list> ├─IDENTIFIER (x) ├─COMMA (,) └─IDENTIFIER (y) ├─COLON (:) ├─<type> └─KEYWORD (integer) └─SEMICOLON (;) └─<var-declaration> ├─<identifier-list> └─IDENTIFIER (flag) ├─COLON (:) ├─<type> └─KEYWORD (boolean) └─SEMICOLON (;) ├─<compound-statement> ├─KEYWORD (mulai) ├─<statement-list> ├─<assignment-statement> ├─IDENTIFIER (x) ├─ASSIGN_OPERATOR (:=) └─<expression> └─<simple-expression> └─<term> └─<factor> └─NUMBER (3)

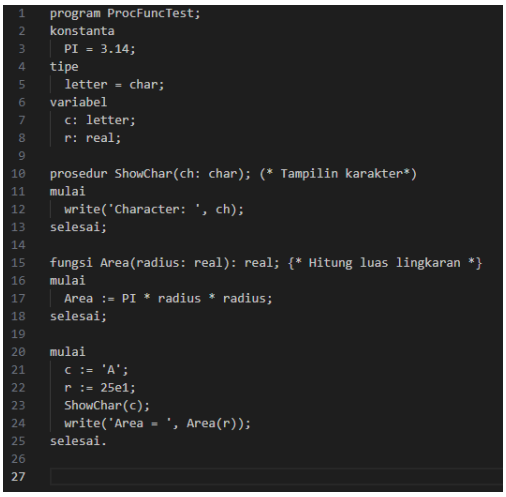
No	Input	Output
		│ │ ├─SEMICOLON (;) │ │ ├─<assignment-statement> │ │ │ ├─IDENTIFIER (y) │ │ │ ├─ASSIGN_OPERATOR (:=) │ │ │ └─<expression> │ │ │ └─<simple-expression> │ │ │ └─<term> │ │ │ └─<factor> │ │ │ └─NUMBER (7) │ │ ├─SEMICOLON (;) │ │ ├─<assignment-statement> │ │ │ ├─IDENTIFIER (flag) │ │ │ ├─ASSIGN_OPERATOR (:=) │ │ │ └─<expression> │ │ │ └─<simple-expression> │ │ │ ├─<term> │ │ │ │ ├─<factor> │ │ │ │ │ ├─LPARENTHESIS (() │ │ │ │ │ ├─<expression> │ │ │ │ │ │ ├─<simple-expression> │ │ │ │ │ │ │ └─<term> │ │ │ │ │ │ │ └─<factor> │ │ │ │ │ │ │ └─IDENTIFIER (x) │ │ │ │ │ │ ├─<relational-operator> │ │ │ │ │ │ │ └─RELATIONAL_OPERATOR (<) │ │ │ │ │ │ └─<simple-expression> │ │ │ │ │ │ └─<term> │ │ │ │ │ │ └─<factor> │ │ │ │ │ │ └─IDENTIFIER (y) │ │ │ │ │ └─RPARENTHESIS ()) │ │ │ │ ├─<multiplicative-operator> │ │ │ │ │ └─LOGICAL_OPERATOR (dan) │ │ │ │ └─<factor> │ │ │ │ ├─LOGICAL_OPERATOR (tidak) │ │ │ │ └─<factor> │ │ │ │ ├─LPARENTHESIS (() │ │ │ │ ├─<expression> │ │ │ │ │ ├─<simple-expression> │ │ │ │ │ │ └─<term> │ │ │ │ │ │ └─<factor> │ │ │ │ │ │ └─IDENTIFIER (x) │ │ │ │ │ ├─<relational-operator> │ │ │ │ │ │ └─RELATIONAL_OPERATOR (=) │ │ │ │ │ └─<simple-expression> │ │ │ │ │ └─<term> │ │ │ │ │ └─<factor> │ │ │ │ │ └─NUMBER (0) │ │ │ │ └─RPARENTHESIS ()) │ │ │ ├─<additive-operator> │ │ │ │ └─LOGICAL_OPERATOR (atau) │ │ │ └─<term> │ │ │ └─<factor> │ │ │ ├─LPARENTHESIS (() │ │ │ ├─<expression> │ │ │ │ ├─<simple-expression> │ │ │ │ │ └─<term> │ │ │ │ │ └─<factor> │ │ │ │ │ └─IDENTIFIER (y) │ │ │ │ ├─<relational-operator> │ │ │ │ │ └─RELATIONAL_OPERATOR (>=) │ │ │ │ └─<simple-expression> │ │ │ │ └─<term> │ │ │ │ └─<factor>

No	Input	Output
		
3.		

No	Input	Output
		

No	Input	Output

```
    │   │   │   │       └─NUMBER (5)
    │   │   │   ├─KEYWORD (turun_ke)
    │   │   │   ├─<expression>
    │   │   │   │ └─<simple-expression>
    │   │   │   │   └─<term>
    │   │   │   │     └─<factor>
    │   │   │   │       └─NUMBER (1)
    │   │   │   ├─KEYWORD (lakukan)
    │   │   │   └─<procedure/function-call>
    │   │   │     ├─IDENTIFIER (write)
    │   │   │     ├─LPARENTHESIS (()
    │   │   │     ├─<parameter-list>
    │   │   │     │ └─<expression>
    │   │   │     │   └─<simple-expression>
    │   │   │     │     └─<term>
    │   │   │     │       └─<factor>
    │   │   │     │         └─<array-access>
    │   │   │     │           ├─IDENTIFIER (arr)
    │   │   │     │           ├─LBRACKET ([)
    │   │   │     │           ├─<expression>
    │   │   │     │           │ └─<simple-expression>
    │   │   │     │           │   └─<term>
    │   │   │     │           │     └─<factor>
    │   │   │     │           │       └─IDENTIFIER (i)
    │   │   │     │           └─RBRACKET (])
    │   │   │     └─RPARENTHESIS ())
    │   │   ├─SEMICOLON (;)
    │   │   ├─<assignment-statement>
    │   │   │ ├─IDENTIFIER (i)
    │   │   │ ├─ASSIGN_OPERATOR (:=)
    │   │   │ └─<expression>
    │   │   │   └─<simple-expression>
    │   │   │     └─<term>
    │   │   │       └─<factor>
    │   │   │         └─NUMBER (0)
    │   │   ├─SEMICOLON (;)
    │   │   ├─<while-statement>
    │   │   │ ├─KEYWORD (selama)
    │   │   │ ├─<expression>
    │   │   │ │ ├─<simple-expression>
    │   │   │ │ │ └─<term>
    │   │   │ │ │   └─<factor>
    │   │   │ │ │     └─IDENTIFIER (i)
    │   │   │ │ ├─<relational-operator>
    │   │   │ │ │ └─RELATIONAL_OPERATOR (<)
    │   │   │ │ └─<simple-expression>
    │   │   │ │   └─<term>
    │   │   │ │     └─<factor>
    │   │   │ │       └─NUMBER (5)
    │   │   │ ├─KEYWORD (lakukan)
    │   │   │ └─<compound-statement>
    │   │   │   ├─KEYWORD (mulai)
    │   │   │   ├─<statement-list>
    │   │   │   │ └─<assignment-statement>
    │   │   │   │   ├─IDENTIFIER (i)
    │   │   │   │   ├─ASSIGN_OPERATOR (:=)
    │   │   │   │   └─<expression>
    │   │   │   │     └─<simple-expression>
    │   │   │   │       ├─<term>
    │   │   │   │       │ └─<factor>
    │   │   │   │       │   └─IDENTIFIER (i)
    │   │   │   │       ├─<additive-operator>
    │   │   │   │       │ └─ARITHMETIC_OPERATOR (+)
```

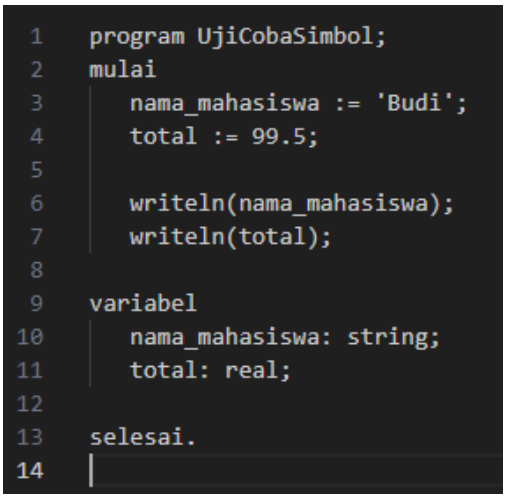
No	Input	Output
		<pre>│ │ │ │ └─<term> │ │ │ │ └─<factor> │ │ │ │ └─NUMBER (1) │ │ │ └─KEYWORD (selesai) │ │ └─SEMICOLON (;) │ └─KEYWORD (selesai) └─DOT (.)</pre>
4.	 	<pre><program> ├─<program-header> │ ├─KEYWORD (program) │ ├─IDENTIFIER (procfunctest) │ └─SEMICOLON (;) ├─<declaration-part> │ ├─<const-declaration-part> │ │ ├─KEYWORD (konstanta) │ │ └─<const-declaration> │ │ ├─IDENTIFIER (pi) │ │ ├─RELATIONAL_OPERATOR (=) │ │ ├─NUMBER (3.14) │ │ └─SEMICOLON (;) │ ├─<type-declaration-part> │ │ ├─KEYWORD (tipe) │ │ └─<type-declaration> │ │ ├─IDENTIFIER (letter) │ │ ├─RELATIONAL_OPERATOR (=) │ │ ├─<type> │ │ │ └─KEYWORD (char) │ │ └─SEMICOLON (;) │ ├─<var-declaration-part> │ │ ├─KEYWORD (variabel) │ │ ├─<var-declaration> │ │ │ ├─<identifier-list> │ │ │ │ └─IDENTIFIER (c) │ │ │ ├─COLON (:) │ │ │ ├─<type> │ │ │ │ └─IDENTIFIER (letter) │ │ │ └─SEMICOLON (;) │ │ └─<var-declaration> │ │ ├─<identifier-list> │ │ │ └─IDENTIFIER (r) │ │ ├─COLON (:) │ │ ├─<type> │ │ │ └─KEYWORD (real) │ │ └─SEMICOLON (;) │ ├─<subprogram-declaration> │ │ └─<procedure-declaration> │ │ ├─KEYWORD (prosedur) │ │ ├─IDENTIFIER (showchar) │ │ ├─<formal-parameter-list> │ │ │ ├─LPARENTHESIS (() │ │ │ ├─<identifier-list> │ │ │ │ └─IDENTIFIER (ch) │ │ │ ├─COLON (:) │ │ │ ├─<type> │ │ │ │ └─KEYWORD (char) │ │ │ └─RPARENTHESIS ()) │ │ ├─SEMICOLON (;) │ │ ├─<declaration-part> │ │ ├─<compound-statement> │ │ │ ├─KEYWORD (mulai) │ │ │ ├─<statement-list> │ │ │ │ ├─<procedure/function-call></pre>

No	Input	Output

```
    │ │   │ │ │ ├─IDENTIFIER (write)
    │ │   │ │ │ ├─LPARENTHESIS (()
    │ │   │ │ │ ├─<parameter-list>
    │ │   │ │ │ │ ├─<expression>
    │ │   │ │ │ │ │ └─<simple-expression>
    │ │   │ │ │ │ │   └─<term>
    │ │   │ │ │ │ │     └─<factor>
    │ │   │ │ │ │ │       └─STRING_LITERAL
('Character: ')
    │ │   │ │ │ │ ├─COMMA (,)
    │ │   │ │ │ │ └─<expression>
    │ │   │ │ │ │   └─<simple-expression>
    │ │   │ │ │ │     └─<term>
    │ │   │ │ │ │       └─<factor>
    │ │   │ │ │ │         └─IDENTIFIER (ch)
    │ │   │ │ │ └─RPARENTHESIS ())
    │ │   │ │ └─SEMICOLON (;)
    │ │   │ └─KEYWORD (selesai)
    │ │   └─SEMICOLON (;)
    │ └─<subprogram-declaration>
    │   └─<function-declaration>
    │     ├─KEYWORD (fungsi)
    │     ├─IDENTIFIER (area)
    │     ├─<formal-parameter-list>
    │     │ ├─LPARENTHESIS (()
    │     │ ├─<identifier-list>
    │     │ │ └─IDENTIFIER (radius)
    │     │ ├─COLON (:)
    │     │ ├─<type>
    │     │ │ └─KEYWORD (real)
    │     │ └─RPARENTHESIS ())
    │     ├─COLON (:)
    │     ├─<type>
    │     │ └─KEYWORD (real)
    │     ├─SEMICOLON (;)
    │     ├─<declaration-part>
    │     ├─<compound-statement>
    │     │ ├─KEYWORD (mulai)
    │     │ ├─<statement-list>
    │     │ │ ├─<assignment-statement>
    │     │ │ │ ├─IDENTIFIER (area)
    │     │ │ │ ├─ASSIGN_OPERATOR (:=)
    │     │ │ │ └─<expression>
    │     │ │ │   └─<simple-expression>
    │     │ │ │     └─<term>
    │     │ │ │       ├─<factor>
    │     │ │ │       │ └─IDENTIFIER (pi)
    │     │ │ │       ├─<multiplicative-operator>
    │     │ │ │       │ └─ARITHMETIC_OPERATOR (*)
    │     │ │ │       ├─<factor>
    │     │ │ │       │ └─IDENTIFIER (radius)
    │     │ │ │       ├─<multiplicative-operator>
    │     │ │ │       │ └─ARITHMETIC_OPERATOR (*)
    │     │ │ │       └─<factor>
    │     │ │ │         └─IDENTIFIER (radius)
    │     │ │ └─SEMICOLON (;)
    │     │ └─KEYWORD (selesai)
    │     └─SEMICOLON (;)
    ├─<compound-statement>
    │ ├─KEYWORD (mulai)
    │ ├─<statement-list>
    │ │ ├─<assignment-statement>
    │ │ │ ├─IDENTIFIER (c)
```

No	Input	Output

```
    │   │   ├─ASSIGN_OPERATOR (:=)
    │   │   └─<expression>
    │   │     └─<simple-expression>
    │   │       └─<term>
    │   │         └─<factor>
    │   │           └─CHAR_LITERAL ('A')
    │   ├─SEMICOLON (;)
    │   ├─<assignment-statement>
    │   │   ├─IDENTIFIER (r)
    │   │   ├─ASSIGN_OPERATOR (:=)
    │   │   └─<expression>
    │   │     └─<simple-expression>
    │   │       └─<term>
    │   │         └─<factor>
    │   │           └─NUMBER (25e1)
    │   ├─SEMICOLON (;)
    │   ├─<procedure/function-call>
    │   │   ├─IDENTIFIER (showchar)
    │   │   ├─LPARENTHESIS (()
    │   │   ├─<parameter-list>
    │   │   │ └─<expression>
    │   │   │   └─<simple-expression>
    │   │   │     └─<term>
    │   │   │       └─<factor>
    │   │   │         └─IDENTIFIER (c)
    │   │   └─RPARENTHESIS ())
    │   ├─SEMICOLON (;)
    │   ├─<procedure/function-call>
    │   │   ├─IDENTIFIER (write)
    │   │   ├─LPARENTHESIS (()
    │   │   ├─<parameter-list>
    │   │   │ ├─<expression>
    │   │   │ │ └─<simple-expression>
    │   │   │ │   └─<term>
    │   │   │ │     └─<factor>
    │   │   │ │       └─STRING_LITERAL ('Area = ')
    │   │   │ ├─COMMA (,)
    │   │   │ └─<expression>
    │   │   │   └─<simple-expression>
    │   │   │     └─<term>
    │   │   │       └─<factor>
    │   │   │         └─<procedure/function-call>
    │   │   │           ├─IDENTIFIER (area)
    │   │   │           ├─LPARENTHESIS (()
    │   │   │           ├─<parameter-list>
    │   │   │           │ └─<expression>
    │   │   │           │   └─<simple-expression>
    │   │   │           │     └─<term>
    │   │   │           │       └─<factor>
    │   │   │           │         └─IDENTIFIER (r)
    │   │   │           └─RPARENTHESIS ())
    │   │   └─RPARENTHESIS ())
    │   └─SEMICOLON (;)
    └─KEYWORD (selesai)
  └─DOT (.)
```

No	Input	Output
5.		`Line 9, Col 1` `9: variabel` `^` `SyntaxError: Unexpected token 'variabel'` `(expected: KEYWORD, IDENTIFIER)` `expected a statement (jika, selama, untuk, mulai,` `or identifier)` **Keterangan:** Ini test-case yang memang harus salah

4. Kesimpulan dan Saran

4.1. Kesimpulan

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

- Parser berbasis Recursive Descent yang dibuat telah berhasil memvalidasi struktur sintaksis program PASCAL-S sesuai grammar.
- Program mampu membangun Parse Tree secara lengkap dan menampilkan hasilnya dalam format yang mudah dibaca.
- Integrasi dengan lexer dari milestone sebelumnya berjalan baik sehingga token dapat diproses secara berurutan dan konsisten.
- Pengujian menunjukkan parser dapat menangani input valid maupun memberikan error message yang jelas ketika menemukan token yang tidak sesuai.
- Pengerjaan proyek ini meningkatkan pemahaman kelompok mengenai keterkaitan antara lexer–parser, desain grammar, dan prinsip kerja top-down parsing.

4.2. Saran

- Mengembangkan mekanisme error recovery agar parser dapat tetap melanjutkan proses setelah menemukan kesalahan.
- Memperbaiki dan memperketat grammar untuk mengurangi potensi ambiguitas atau edge case yang belum tertangani.
- Menambahkan dukungan terhadap fitur bahasa PASCAL-S yang lebih luas atau kompleks.
- Membuat visualisasi Parse Tree yang lebih interaktif atau output dalam bentuk grafis untuk memudahkan debugging.
- Menambah cakupan unit test agar parser lebih teruji terhadap variasi input dan perubahan kode di masa depan.

Lampiran

Pembagian Tugas

Anggota	Tugas	Persentase
13523123 Rhio Bimo Prakoso S	Tester, Laporan, Kode	20
13523137 M Aulia Azka	Laporan, Konsumsi	20
13523161 Arlow Emmanuel Hergara	Laporan, Debugging	20
13523162 Fachriza Ahmad Setiyono	Kode Visualisasi, Kode Grammar	20
13523163 Filbert Engyo	Laporan, Kode Grammar	20

Pranala Repository

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

Daftar Pustaka

https://www.geeksforgeeks.org/compiler-design/introduction-to-syntax-analysis-in-compiler-design/

https://www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.htm

https://youtu.be/OIKL6wFjFOo?si=LP43S3T74joksEf6

https://youtu.be/iddRD8tJi44?si=OLhl3fJQpE6nYbsq

https://youtu.be/v_wvcuJ6mGY?si=XAmFP6OuCubrUiW4

https://youtu.be/u4-rpIlV9NI?si=W1xjsGcl-jb76iO7

https://www.geeksforgeeks.org/theory-of-computation/what-is-context-free-grammar/

https://www.tutorialspoint.com/automata_theory/context_free_grammar_introduction.htm

https://www.geeksforgeeks.org/compiler-design/lr-parser/

https://www.geeksforgeeks.org/compiler-design/recursive-descent-parser/

https://www.geeksforgeeks.org/compiler-design/parse-tree-in-compiler-design/

https://www.tutorialspoint.com/automata_theory/automata_theory_parse_tree.htm

https://www.sciencedirect.com/topics/computer-science/parse-tree

~ Terimakasih ~