

Laporan Milestone II TBFO

Syntax Analysis

ParsingIsAllYouNeed



Muhammad Raihan Nazhim Oktana

13523021

13523021@std.stei.itb.ac.id

Shanice Feodora Tjahjono

13523097

13523097@std.stei.itb.ac.id

Faqih Muhammad Syuhada

13523057

13523057@std.stei.itb.ac.id

Muhammad Fathur Rizky

13523105

13523105@std.stei.itb.ac.id

**Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung
2025**

Daftar Isi

BAB I: Deskripsi Tugas	2
BAB II: Landasan Teori	3
2.1 Analisis Sintaksis (Parsing)	3
2.2 Algoritma Recursive Descent	3
2.3 Parse Tree	3
2.4 Keterhubungan antara Parser dan Lexer	4
BAB III: Perancangan & Implementasi	5
3.1 Perancangan	5
3.1.1 Arsitektur Sistem	5
3.1.2 Representasi Data dan Struktur AST	5
3.2 Implementasi	6
3.2.1 Penyesuaian Lexer dengan Post-processing	6
3.2.2 Base AST Class	7
3.2.2 Program	8
3.2.3 Expression	9
3.2.4 Declaration	10
3.2.5 Statement	12
3.2.6 Parser	13
3.2.7 Tree Formatter	16
BAB IV: Pengujian	19
4.1 Pengujian Test Cases	19
BAB V: Kesimpulan dan Saran	76
5.1 Kesimpulan	76
5.2 Saran	77
Lampiran	77
Daftar Pustaka	77

BAB I: Deskripsi Tugas

Pada era modern, bahasa pemrograman memegang peran fundamental dalam pengembangan perangkat lunak serta sistem komputasi. Salah satu komponen penting dalam ekosistem pemrograman adalah compiler, yaitu perangkat lunak yang menerjemahkan kode sumber (*source code*) ke dalam representasi yang dapat dipahami dan dieksekusi oleh mesin. Proses kompilasi terdiri atas beberapa tahap berurutan yang masing-masing memiliki fungsi spesifik dalam mentransformasikan program. Setelah tahap analisis leksikal (*lexical analysis*) berhasil dilaksanakan pada Milestone I, Tugas Besar Mata Kuliah Teori Bahasa Formal dan Otomata (IF2224-24) berlanjut ke Milestone II, yaitu tahap *syntax analysis*.

Bahasa Pascal-S merupakan subset dari bahasa pemrograman Pascal yang telah disederhanakan dengan tujuan memfasilitasi pembelajaran konsep *compiler construction* secara bertahap dan terstruktur. Dalam tugas besar ini, mahasiswa bertindak sebagai pengembang Pascal-S Compiler dan diharapkan dapat mengimplementasikan komponen-komponen inti *compiler front-end*.

Pada Milestone II, mahasiswa berfokus pada pembangunan parser yang merupakan bagian penting dari tahap *syntax analysis*. Parser bertugas memeriksa rangkaian token yang dihasilkan oleh lexer, untuk memastikan bahwa token-token tersebut tersusun sesuai dengan aturan tata bahasa formal (grammar) Pascal-S. Lebih jauh, parser dibangun menggunakan pendekatan *Recursive Descent Parsing*, yaitu metode *top-down parsing* yang mengimplementasikan setiap aturan produksi grammar sebagai prosedur rekursif.

BAB II: Landasan Teori

2.1 Analisis Sintaksis (Parsing)

Analisis sintaksis merupakan tahap kedua dalam compiler front-end yang bertugas memverifikasi apakah urutan token hasil analisis leksikal membentuk struktur sintaks yang valid berdasarkan aturan tata bahasa atau *context-free grammar* (CFG) bahasa yang dikompilasi [1].

Parser menyusun token-token menjadi struktur hierarkis yang menggambarkan hubungan sintaktis antar komponen, sekaligus mendeteksi dan melaporkan kesalahan sintaks (syntax error) yang muncul selama proses parsing [3]. Dua kategori utama teknik parsing adalah sebagai berikut.

- Top-Down Parsing, yaitu proses parsing yang dimulai dari simbol awal (start symbol) dan melakukan derivasi hingga mencapai simbol terminal.
- Bottom-Up Parsing, yaitu proses reduksi token terminal menjadi non-terminal hingga mencapai simbol awal.

Hasil akhir tahap ini adalah struktur sintaks formal yang menjadi dasar bagi analisis semantik, optimasi, dan code generation.

2.2 Algoritma Recursive Descent

Recursive Descent Parsing merupakan salah satu teknik *top-down parsing* yang mengimplementasikan setiap nonterminal dalam *grammar* sebagai prosedur rekursif yang saling memanggil sesuai aturan produksi (*production rules*) [4].

Keunggulan pendekatan ini adalah implementasinya yang sederhana, intuitif, dan mudah dipetakan langsung dari grammar sehingga banyak digunakan dalam pembuatan kompiler pendidikan maupun *domain-specific languages* (DSLs) [5]. Namun, *Recursive Descent* hanya bekerja secara efektif pada grammar yang tidak ambigu dan bebas dari left recursion [1].

Secara umum, proses kerja *Recursive Descent* meliputi sebagai berikut.

- pembacaan token secara sekuensial dari lexer,
- pencocokan token dengan simbol yang diharapkan (matching),
- pemanggilan prosedur rekursif untuk setiap non-terminal, dan
- pelaporan kesalahan sintaks jika token tidak sesuai.

Teknik ini memberikan fleksibilitas dalam penanganan error recovery sehingga pesan kesalahan dapat dibuat lebih informatif bagi pengguna.

2.3 Parse Tree

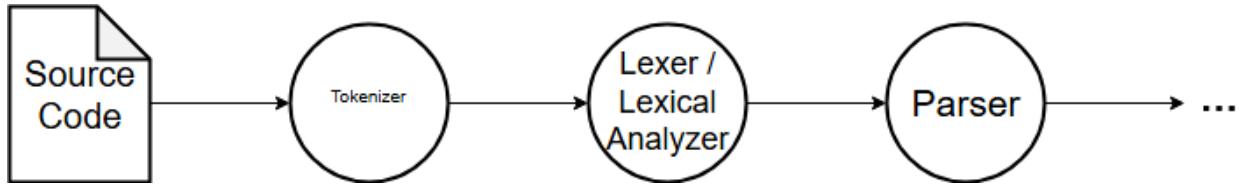
Parse Tree, atau *concrete syntax tree*, merupakan representasi hierarkis dari proses derivasi grammar terhadap suatu program. *Parse tree* menunjukkan bagaimana simbol awal grammar diturunkan hingga membentuk terminal melalui penerapan aturan produksi [6].

Struktur parse tree terdiri atas:

- Akar pohon (root): simbol awal grammar.
- Simpul internal: simbol non-terminal.
- Simpul daun: simbol terminal (token).

Parse Tree berbeda dengan *Abstract Syntax Tree* (AST). AST merupakan representasi sintaks yang lebih ringkas karena menghilangkan elemen sintaktis yang tidak relevan terhadap struktur semantik, seperti tanda kurung atau delimiter [2]. *Parse Tree* mempertahankan seluruh detail produksi grammar, sedangkan AST lebih digunakan pada tahap analisis semantik dan optimasi.

2.4 Keterhubungan antara Parser dan Lexer



Gambar 1. Tampilan tingkat tinggi alur kerja *frontend* kompilator
 (Sumber: https://createlang.rs/01_calculator/grammar_lexer_parser.html)

Analisis leksikal (*lexical analysis*) dan analisis sintaksis (*syntax analysis*) merupakan dua tahapan fundamental dalam compiler front-end yang bekerja secara berurutan dan saling melengkapi dalam membentuk pemahaman struktural terhadap program sumber. Kedua tahap ini dirancang dengan prinsip modularitas, di mana masing-masing memiliki tanggung jawab yang terpisah secara jelas, namun menghasilkan keluaran yang menjadi masukan bagi tahap berikutnya dalam proses kompilasi [1], [3].

Pada tahap awal, *lexer* bertugas melakukan pemindai karakter pada *source code* dan mengelompokkannya menjadi token-token yang bermakna. Token-token ini selanjutnya menjadi satuan dasar yang dianalisis oleh parser. Dengan demikian, analisis leksikal dapat dipandang sebagai proses abstraksi pertama yang menerjemahkan representasi tekstual menjadi struktur simbolis yang lebih terorganisasi. Token yang dihasilkan lexer harus akurat dan sesuai dengan *lexical specification* agar dapat diproses oleh parser tanpa menimbulkan ambiguitas atau kesalahan leksikal.

Parser kemudian memanfaatkan token-token tersebut untuk memverifikasi kesesuaian struktur sintaks program dengan tata bahasa formal yang didefinisikan oleh context-free grammar. Proses ini mengasumsikan bahwa masukan telah terbebas dari kesalahan leksikal sehingga parser dapat berfokus pada pengujian hubungan sintaktis antar token dan membangun representasi hierarkis berupa *parse tree* atau *abstract syntax tree*. Apabila ditemukan token yang tidak sesuai dengan aturan sintaks, parser wajib memberikan pelaporan kesalahan (*syntax error reporting*) yang informatif dan, jika memungkinkan, melakukan error recovery tanpa menghentikan proses parsing secara keseluruhan [1], [3].

Hubungan antara kedua tahap tersebut bersifat interdependen dan *pipeline-oriented*: keberhasilan *parsing* sangat bergantung pada ketepatan tokenisasi oleh *lexer*. Kesalahan pada tahap leksikal berpotensi menghasilkan *cascade errors* pada tahap sintaksis. Oleh karena itu,

pemisahan dan pengorganisasian fungsi antara lexer dan parser tidak hanya meningkatkan keterbacaan dan modularitas desain kompiler, tetapi juga memastikan reliabilitas dan efektivitas keseluruhan proses kompilasi [2], [3].

BAB III: Perancangan & Implementasi

3.1 Perancangan

Tahap perancangan bertujuan untuk mendefinisikan arsitektur sistem, struktur modul, serta hubungan antar komponen dalam proses kompilasi Pascal-S. Proses ini dilakukan secara sistematis dengan mengacu pada prinsip-prinsip compiler construction, di mana sistem dibagi menjadi beberapa tahap yang saling terintegrasi: lexical analysis, syntactic analysis, dan *abstract syntax representation*.

Secara umum, sistem Pascal-S Compiler pada milestone ini dirancang untuk mengonversi teks program sumber ke dalam bentuk Abstract Syntax Tree (AST) yang merepresentasikan struktur sintaks secara hierarkis. Proses ini dimulai dengan lexer (lexical analyzer) yang bertugas memecah kode sumber menjadi token-token yang bermakna, kemudian diteruskan ke parser yang membangun struktur AST berdasarkan grammar bahasa Pascal-S. AST ini menjadi representasi internal dari program yang dapat digunakan untuk analisis lanjutan seperti semantic analysis, intermediate code generation, atau optimization pada tahap berikutnya.

3.1.1 Arsitektur Sistem

Arsitektur sistem dirancang dengan pendekatan modular dan berlapis (layered architecture) agar setiap komponen dapat dikembangkan dan diuji secara terpisah. Struktur utama terdiri dari sebagai berikut.

- Lexer: Modul pertama yang melakukan analisis leksikal. Ia membaca input berupa source code dan menghasilkan daftar token (list[Token]).
- Parser dan AST Builder: Modul yang menerima daftar token dari lexer, memprosesnya sesuai grammar rules, dan membangun struktur Abstract Syntax Tree (AST).
- Visitor Interface: Sebuah pola desain yang memungkinkan operasi traversal dan transformasi AST tanpa perlu memodifikasi struktur node.
- Serializer (to_dict): Modul bantu yang mengubah AST menjadi representasi dictionary agar dapat disimpan, divisualisasikan, atau diuji secara otomatis.

Desain modular ini memungkinkan pengujian dan pembaruan komponen secara independen. Misalnya, modifikasi pada Lexer untuk mendukung bahasa Indonesia tidak mengharuskan perubahan pada modul AST karena antarmuka antar modul tetap konsisten (token → AST).

3.1.2 Representasi Data dan Struktur AST

Setiap elemen bahasa Pascal-S direpresentasikan sebagai node dalam Abstract Syntax Tree (AST) yang diturunkan dari kelas abstrak ASTNode. Pendekatan berbasis kelas ini memberikan fleksibilitas tinggi melalui prinsip abstraksi dan polimorfisme sehingga tipe node baru dapat ditambahkan tanpa memengaruhi keseluruhan sistem. Struktur AST diorganisasikan secara

hierarkis, mencerminkan grammar bahasa Pascal-S dan memisahkan fungsi semantik tiap komponen.

Pada tingkat atas terdapat Program-Level Nodes seperti Program dan Block yang merepresentasikan satuan program lengkap. Declaration Nodes seperti VarDeclaration, ConstDeclaration, dan TypeDeclaration mendeskripsikan entitas deklaratif, sementara Expression Nodes seperti BinaryOp, UnaryOp, dan FunctionCall menggambarkan operasi aritmetika dan logika. Terakhir, Statement Nodes seperti IfStatement, WhileStatement, dan AssignmentStatement merepresentasikan kontrol aliran dan aksi eksekusi program. Dengan desain ini, AST tidak hanya mencerminkan context-free grammar Pascal-S tetapi juga berperan sebagai representasi semantik yang siap digunakan pada tahap analisis dan generasi kode selanjutnya.

3.2 Implementasi

3.2.1 Penyesuaian Lexer dengan Post-processing

Modifikasi Lexer diperlukan karena terdapat perubahan spesifikasi yang awalnya berbahasa Inggris, sekarang menggunakan bahasa Indonesia.

```
from .hyphenated_keywords import merge_hyphenated_keywords

class Lexer:
    ...

    def tokenize(self, source_code: str) -> list[Token]:
        ...
        # Post-process to merge hyphenated keywords
        tokens = merge_hyphenated_keywords(tokens)
        return tokens
```

Diperlukan modifikasi pada tahap pascaproses (post-processing) lexer agar dapat menangani kata kunci majemuk (hyphenated keywords) yang kini ditulis dalam bahasa Indonesia, seperti turun-ke, selain-itu yang sebelumnya dalam bahasa Inggris mungkin hanya terdiri dari satu kata (downto, else).

```
from syntax import Token, TokenType

HYPHENATED_KEYWORDS = {
    "turun-ke",
    "selain-itu",
}

def merge_hyphenated_keywords(tokens: list[Token]) -> list[Token]:
    result = []
    i = 0

    while i < len(tokens):
        if i + 2 < len(tokens):
            token1 = tokens[i]
            token2 = tokens[i + 1]
```

```

        token3 = tokens[i + 2]

        if ((token1.type == TokenType.IDENTIFIER or token1.type == TokenType.KEYWORD) and
            token2.type == TokenType.ARITHMETIC_OPERATOR and
            token2.value == "-" and
            (token3.type == TokenType.IDENTIFIER or token3.type == TokenType.KEYWORD)):

            potential_keyword = f'{token1.value}-{token3.value}'

            if potential_keyword.lower() in HYPHENATED_KEYWORDS:
                merged_token = Token(
                    type=TokenType.KEYWORD,
                    value=potential_keyword,
                    start=token1.start,
                    end=token3.end
                )
                result.append(merged_token)
                i += 3
                continue

            result.append(tokens[i])
            i += 1

    return result

```

Secara ilmiah, modifikasi ini dibutuhkan karena perubahan spesifikasi bahasa menyebabkan pergeseran satuan leksikal (lexical unit)—dari satu token menjadi gabungan dua token yang terpisah oleh tanda hubung (-). Tanpa mekanisme penggabungan (melalui fungsi merge_hyphenated_keywords), lexer akan menganggap selain-ituf sebagai dua token terpisah (selain, -, dan itu), padahal secara sintaksis ketiganya membentuk satu konstruksi terminal tunggal dalam grammar bahasa.

3.2.2 Base AST Class

Sebelum memasuki tahap perancangan node spesifik untuk setiap komponen bahasa, diperlukan landasan abstraksi yang seragam untuk seluruh struktur sintaks. Bagian ini berfungsi sebagai penghubung (bridge) antara hasil parsing dan proses analisis semantik berikutnya melalui pembentukan Abstract Syntax Tree (AST). Kelas dasar ASTNode menyediakan antarmuka umum yang akan diwarisi oleh seluruh node dalam AST sehingga setiap elemen sintaks dapat diproses secara homogen tanpa kehilangan kekhususan perilaku masing-masing. Dengan adanya lapisan dasar ini, sistem menjadi modular dan mudah diperluas, memungkinkan penerapan pola desain visitor untuk menjembatani analisis lintas tahap, seperti evaluasi semantik dan generasi kode menengah.

```

from abc import ABC, abstractmethod
from typing import Any

class ASTNode(ABC):

```

```

@abstractmethod
def accept(self, visitor: Any) -> Any:
    pass

@abstractmethod
def to_dict(self) -> dict:
    pass

```

Secara konseptual, AST merupakan struktur pohon yang memuat representasi hierarkis dari konstruksi sintaks suatu bahasa pemrograman setelah fase *parsing*. Metode `accept(visitor)` menerapkan pola desain Visitor Pattern, yaitu prinsip yang memungkinkan pemisahan antara struktur data dan logika pemrosesan. Ini memperkuat *extensibility* sistem—misalnya, dapat menambahkan *semantic analyzer* atau *code generator* tanpa memodifikasi kelas node-nya. Metode `to_dict()` berperan dalam serialisasi semantik: memungkinkan transformasi AST ke format representatif seperti JSON untuk keperluan visualisasi, debugging, atau komunikasi lintas modul.

3.2.2 Program

Setelah mendefinisikan kerangka dasar AST, langkah berikutnya adalah membentuk representasi tingkat tertinggi dari sebuah program. Bagian ini berfungsi sebagai puncak hierarki sintaks abstrak, di mana seluruh komponen seperti deklarasi, ekspresi, dan pernyataan disusun secara terstruktur di dalam blok utama. Kelas Program dan Block menjadi penghubung antara struktur sintaks yang terpisah menjadi satu kesatuan program utuh, menjadikan bagian ini sebagai jembatan antara syntactic composition dan execution semantics. Pendekatan ini memungkinkan analisis konteks dan pembentukan tabel simbol dilakukan secara deterministik di seluruh lingkup (scope) program.

```

@dataclass
class Program(ASTNode):

    name: str
    block: ASTNode  # Block

    def accept(self, visitor: Any) -> Any:
        return visitor.visit_program(self)

    def to_dict(self) -> dict:
        return {
            'type': 'Program',
            'name': self.name,
            'block': self.block.to_dict()
        }

@dataclass
class Block(ASTNode):

    declarations: list[ASTNode]
    compound_statement: CompoundStatement

```

```

def accept(self, visitor: Any) -> Any:
    return visitor.visit_block(self)

def to_dict(self) -> dict:
    return {
        'type': 'Block',
        'declarations': [decl.to_dict() for decl in self.declarations],
        'compound_statement': self.compound_statement.to_dict()
    }

```

File ini membentuk lapisan tertinggi dari AST, yakni representasi seluruh program Pascal-S. Kelas Program menjadi akar dari AST, menggabungkan nama program dan blok utama (Block). Block sendiri mengorganisasi deklarasi dan *compound statement*, merepresentasikan *lexical scope* dan *execution scope* secara eksplisit. Desain hierarkis ini mencerminkan teori nested block structure yang diperkenalkan dalam bahasa Pascal dan Algol, di mana deklarasi bersifat lokal terhadap blok tempat ia didefinisikan. Dengan adanya pemisahan antara deklarasi dan tubuh program, sistem dapat melakukan analisis simbol (symbol table construction) secara deterministik dan mendukung *recursive descent parsing* dengan lebih efisien.

3.2.3 Expression

Setelah struktur program terbentuk, fokus berpindah ke representasi ekspresi yang menjadi fondasi dari evaluasi dan manipulasi nilai dalam bahasa pemrograman. Modul ekspresi ini menjembatani antara syntactic analysis dan semantic evaluation karena setiap ekspresi menggambarkan operasi yang dapat dihitung. Dengan membangun hierarki node seperti BinaryOp, UnaryOp, dan literal, sistem dapat mengekspresikan hubungan matematis maupun logis secara eksplisit. Bagian ini menjadi krusial untuk memastikan proses kompilasi dapat menurunkan makna formal (semantic meaning) dari setiap konstruksi sintaks konkret.

```

@dataclass
class Expression(ASTNode):
    pass

@dataclass
class BinaryOp(Expression):
    left: Expression
    operator: str
    right: Expression

@dataclass
class UnaryOp(Expression):
    operator: str
    operand: Expression

@dataclass
class Variable(Expression):
    name: str
    index: Expression = None # For array indexing
    field: str = None # For record field access

@dataclass

```

```

class Number(Expression):
    value: Any # int or float

@dataclass
class String(Expression):
    value: str

@dataclass
class Char(Expression):
    value: str

@dataclass
class Boolean(Expression):
    value: bool

@dataclass
class FunctionCall(Expression):
    name: str
    arguments: list[Expression]

```

Kelas seperti BinaryOp dan UnaryOp merepresentasikan struktur pohon operator, di mana setiap operasi aritmetika atau logika dipecah menjadi node yang memiliki operand kiri dan kanan (atau tunggal pada kasus unary). Pendekatan ini mengikuti teori *abstract syntax representation*, di mana sintaks konkret seperti $a + b * c$ diubah menjadi bentuk terstruktur yang memungkinkan *precedence* dan *associativity* ditentukan secara hierarkis.

Node literal seperti Number, String, Char, dan Boolean mendefinisikan terminal symbol dalam grammar yang secara formal tidak dapat dipecah lagi dalam struktur sintaks. Sementara Variable dan FunctionCall merepresentasikan simbol yang memerlukan resolusi semantik pada tahap berikutnya (misalnya pencarian alamat memori atau pemanggilan subrutin). Dengan demikian, modul ini berperan penting dalam menghubungkan sintaks dan semantik, serta memungkinkan implementasi *evaluator*, *optimizer*, atau *intermediate code generator* yang bersifat sistematis.

3.2.4 Declaration

Sebelum suatu ekspresi atau pernyataan dapat dievaluasi, semua entitas seperti variabel, konstanta, fungsi, dan tipe data harus dideklarasikan terlebih dahulu. Modul deklarasi ini menjadi jembatan antara fase sintaksis dan konstruksi semantik lingkungan (environment construction). Dengan mendefinisikan node seperti VarDeclaration, ConstDeclaration, dan TypeDeclaration, sistem dapat secara sistematis membangun symbol table dan memetakan tipe data setiap entitas yang terlibat. Struktur ini juga mempersiapkan landasan bagi type checking dan scope resolution di tahap analisis semantik, memastikan bahwa seluruh referensi dalam program memiliki definisi yang sah.

```

@dataclass
class VarDeclaration(ASTNode):
    identifiers: list[str]
    type_spec: TypeSpec

```

```

@dataclass
class ConstDeclaration(ASTNode):
    identifier: str
    value: Any # Could be number, string, or boolean

@dataclass
class TypeDeclaration(ASTNode):
    identifier: str
    type_spec: TypeSpec

@dataclass
class ProcedureDeclaration(ASTNode):
    name: str
    parameters: list[Parameter]
    block: ASTNode # Block

@dataclass
class FunctionDeclaration(ASTNode):
    name: str
    parameters: list[Parameter]
    return_type: TypeSpec
    block: ASTNode # Block

@dataclass
class Parameter(ASTNode):
    identifiers: list[str]
    type_spec: TypeSpec

@dataclass
class TypeSpec(ASTNode):
    pass

@dataclass
class SimpleType(TypeSpec):
    name: str

@dataclass
class ArrayType(TypeSpec):
    index_type: Any # Could be SimpleType or range (start..end)
    element_type: TypeSpec

@dataclass
class RecordType(TypeSpec):
    fields: list[VarDeclaration]

```

Secara ilmiah, desain ini meniru context-free grammar yang menjadi dasar sintaks formal Pascal. Setiap deklarasi direpresentasikan sebagai kelas berbeda untuk menjaga *semantic granularity*—misalnya, VarDeclaration berbeda dengan ConstDeclaration karena memiliki perilaku semantik berbeda (nilai konstan bersifat *immutable*, sementara variabel tidak).

Konsep TypeSpec sebagai kelas abstrak memungkinkan pembentukan hierarki tipe seperti SimpleType, ArrayType, dan RecordType. Pendekatan ini sejalan dengan prinsip polimorfisme struktural dalam teori tipe (*type theory*), di mana semua tipe dapat diperlakukan seragam melalui abstraksi yang sama namun memiliki struktur representasi berbeda. Desain ini juga

memungkinkan penerapan *type inference* atau *type checking* dengan lebih sistematis pada tahap *semantic analysis*.

Selain itu, pemisahan ProcedureDeclaration dan FunctionDeclaration mencerminkan distingsi konseptual antara fungsi yang mengembalikan nilai dan prosedur yang tidak yang merupakan konsep dasar dalam teori pemrograman imperatif. Parameter direpresentasikan sebagai daftar agar dapat mengakomodasi definisi seperti procedure f(a, b: integer);, mencerminkan arity dan type binding antar variabel formal.

3.2.5 Statement

Setelah entitas dan ekspresi dapat diidentifikasi secara semantik, tahap berikutnya adalah mendeskripsikan alur eksekusi program melalui pernyataan (statements). Modul ini menjadi penghubung antara declarative representation dan operational semantics, karena di sinilah setiap aksi program diwujudkan. Dengan membangun node seperti AssignmentStatement, IfStatement, WhileStatement, dan ForStatement, sistem dapat merepresentasikan kontrol aliran dan perubahan keadaan (state transition) secara formal. Struktur ini menjadi dasar bagi pembentukan *control flow graph* (CFG) dan analisis statis yang akan digunakan dalam optimisasi serta interpretasi program.

```
@dataclass
class Statement(ASTNode):
    pass

@dataclass
class CompoundStatement(Statement):
    statements: list[Statement]

@dataclass
class AssignmentStatement(Statement):
    variable: Variable
    expression: Expression

@dataclass
class IfStatement(Statement):
    condition: Expression
    then_statement: Statement
    else_statement: Optional[Statement] = None

@dataclass
class WhileStatement(Statement):
    condition: Expression
    body: Statement

@dataclass
class ForStatement(Statement):
    variable: str
    start_expr: Expression
    end_expr: Expression
    direction: str # 'to' or 'downto'
    body: Statement
```

```

@dataclass
class RepeatStatement(Statement):
    statements: list[Statement]
    condition: Expression

@dataclass
class CaseStatement(Statement):
    expression: Expression
    cases: list[tuple[list[Any], Statement]] # List of (constants,
statement) pairs

@dataclass
class ProcedureCall(Statement):
    name: str
    arguments: list['Expression']

@dataclass
class EmptyStatement(Statement):

```

Kelas CompoundStatement menggabungkan beberapa pernyataan ke dalam blok begin...end, mencerminkan prinsip blok atomik dalam teori kontrol alur (control flow theory). AssignmentStatement mewakili operasi mutasi terhadap memori yang merupakan inti paradigma imperatif.

Struktur seperti IfStatement, WhileStatement, ForStatement, dan RepeatStatement menggambarkan konsep kontrol kondisional dan iteratif yang secara formal dikaitkan dengan flow graph dan state transition system. Dengan membangun node khusus untuk tiap jenis pernyataan, sistem dapat dengan mudah dianalisis secara semantik atau ditransformasikan menjadi control flow graph (CFG).

CaseStatement mencerminkan model multi-branch selection, sedangkan ProcedureCall mewakili pemanggilan subrutin tanpa nilai kembalian. Kelas EmptyStatement digunakan untuk menjaga kompleksitas sintaks tetap terdefinisi, bahkan ketika tidak ada aksi eksplisit, mendukung formalitas grammar seperti optional rule (ϵ -production) dalam context-free grammar.

3.2.6 Parser

```

class Parser:
    def __init__(self):
        self.tokens = []
        self.current_index = 0
        self.current_token = None

    def error(self, message: str) -> None:
        raise SyntaxError(message, self.current_token)

    def peek(self, offset: int = 0) -> Optional[Token]:
        index = self.current_index + offset
        if 0 <= index < len(self.tokens):
            return self.tokens[index]
        return None

```

```

def advance(self) -> Token:
    if self.current_token is None:
        raise UnexpectedEOFError("more tokens")

    old_token = self.current_token
    self.current_index += 1
    if self.current_index < len(self.tokens):
        self.current_token = self.tokens[self.current_index]
    else:
        self.current_token = None
    return old_token

def expect(self, token_type: TokenType, value: str = None) -> Token:
    if self.current_token is None:
        expected_desc = f"{token_type.name}"
        if value:
            expected_desc += f" '{value}'"
        raise UnexpectedEOFError(expected_desc)

    if self.current_token.type != token_type:
        expected_desc = f"{token_type.name}"
        if value:
            expected_desc += f" '{value}'"
        raise UnexpectedTokenError(expected_desc, self.current_token)

    if value is not None and self.current_token.value.lower() != value.lower():
        raise UnexpectedTokenError(
            f"{token_type.name} '{value}'", self.current_token)

    return self.advance()

def match(self, token_type: TokenType, value: str = None) -> bool:
    if self.current_token is None:
        return False

    if self.current_token.type != token_type:
        return False

    if value is not None and self.current_token.value.lower() != value.lower():
        return False

    return True

def parse(self) -> Program:
    program_node = self.parse_program()
    if self.current_token is not None:
        self.error("Expected end of file")
    return program_node

```

Kelas Parser pada cuplikan kode tersebut merepresentasikan sebuah parser bergaya recursive descent yang bekerja di atas deretan token hasil proses leksikal. Secara umum, kelas ini bertanggung jawab untuk mengelola posisi pembacaan token, melakukan navigasi ke depan (advance), memeriksa kesesuaian tipe dan nilai token terhadap harapan grammar (expect dan

match), serta memicu kesalahan sintaks ketika ditemukan token yang tidak sesuai. Objek ini juga menyimpan token saat ini melalui atribut `current_token` dan indeks posisinya di dalam daftar tokens melalui `current_index`, sehingga parser dapat berjalan secara deterministik dari awal hingga akhir input.

Konstruktor `__init__` menginisialisasi tiga atribut utama: `tokens` sebagai daftar token yang akan di-parse, `current_index` sebagai pointer numerik ke posisi token saat ini, dan `current_token` sebagai referensi langsung ke token yang sedang dianalisis. Pada tahap ini, daftar token masih kosong, sehingga pada praktiknya biasanya akan ada langkah tambahan (di luar cuplikan) untuk mengisi `self.tokens` dan mengatur `self.current_token` ke token pertama sebelum proses parsing dimulai. Desain ini memisahkan struktur data token dengan mekanisme navigasi, sehingga memudahkan pengelolaan aliran input.

Metode `error` merupakan abstraksi untuk penanganan kesalahan sintaks. Ketika dipanggil, metode ini melempar `SyntaxError` dengan pesan tertentu dan mengaitkannya dengan `current_token`. Dengan demikian, informasi lokasi kesalahan dapat dilacak melalui token yang aktif pada saat error terjadi. Pendekatan ini konsisten dengan praktik umum dalam implementasi compiler, di mana setiap kesalahan sintaks dikaitkan dengan posisi spesifik pada berkas sumber sehingga memudahkan pelaporan kepada pengguna.

Metode `peek` digunakan untuk melakukan lookahead ke depan tanpa mengubah posisi parser saat ini. Fungsi ini menerima parameter `offset` yang menyatakan seberapa jauh token di depan yang ingin dilihat relatif terhadap `current_index`. Jika indeks yang dihitung masih berada dalam rentang daftar token, fungsi akan mengembalikan token pada posisi tersebut; jika tidak, fungsi mengembalikan `None`. Fasilitas lookahead ini penting dalam desain parser recursive descent untuk mengambil keputusan berdasarkan satu atau beberapa token berikutnya tanpa mengonsumsi token tersebut.

Metode `advance` bertanggung jawab memajukan posisi pembacaan satu token ke depan. Sebelum memajukan, fungsi memeriksa apakah `current_token` bernilai `None`. Jika `current_token` sudah `None`, berarti parser telah mencapai akhir input namun masih diminta untuk membaca token tambahan, sehingga dilempar sebuah `UnexpectedEOFError` sebagai sinyal bahwa parser mengharapkan "more tokens". Jika masih ada token, fungsi menyimpan token lama dalam `old_token`, menaikkan `current_index`, dan memperbarui `current_token` menjadi token berikutnya, atau `None` jika indeks sudah melampaui panjang daftar. Fungsi kemudian mengembalikan token lama tersebut. Pola ini mendukung model konsumsi token secara sekvensial yang lazim pada parser top-down.

Metode `expect` mengimplementasikan mekanisme pemeriksaan token yang lebih ketat. Metode ini menerima parameter `token_type` dan opsional `value`. Pertama, jika `current_token` bernilai `None`, berarti parser sudah berada pada akhir input sedangkan grammar masih mengharapkan simbol tertentu, sehingga dilempar `UnexpectedEOFError` dengan deskripsi token yang diharapkan. Jika `current_token` ada, fungsi memeriksa apakah tipe token sesuai dengan `token_type`. Jika tidak, dilempar `UnexpectedTokenError` dengan informasi mengenai token yang diharapkan dan token aktual. Selanjutnya, jika argumen `value` diberikan, fungsi juga memeriksa kesesuaian nilai lexeme token secara case-insensitive (menggunakan `lower()` pada kedua sisi).

Bila semua syarat terpenuhi, token saat ini dikonsumsi melalui pemanggilan advance dan token lama dikembalikan. Dengan demikian, expect tidak hanya mengonsumsi token, tetapi juga menegakkan aturan grammar dan memastikannya dipatuhi secara ketat.

Metode match menyediakan versi pemeriksaan yang lebih lunak dibanding expect. Sama seperti expect, metode ini memeriksa apakah current_token ada, lalu memeriksa kesesuaian tipe token dan, jika diberikan, nilai token secara case-insensitive. Namun, match tidak memajukan posisi parser dan tidak melempar pengecualian jika token tidak sesuai; ia hanya mengembalikan nilai boolean True jika token cocok dan False jika tidak. Pola ini khas untuk fungsi predikat dalam parser, yang memungkinkan cabang logika “jika token berikutnya adalah X, lakukan Y” tanpa sekaligus mengonsumsi token sebelum keputusan dibuat.

Terakhir, metode parse berfungsi sebagai titik masuk utama untuk proses parsing keseluruhan program. Metode ini memanggil parse_program() yang diasumsikan membangun dan mengembalikan sebuah objek Program sebagai representasi abstrak dari program input, misalnya dalam bentuk Abstract Syntax Tree (AST). Setelah parse_program() selesai, fungsi masih melakukan pemeriksaan tambahan untuk menjamin bahwa seluruh token telah diproses dengan memeriksa apakah current_token bernilai None. Jika masih terdapat token tersisa (artinya current_token tidak None), maka fungsi memanggil self.error("Expected end of file"), menandakan bahwa ada simbol-simbol berlebih yang tidak terinterpretasi oleh grammar. Pendekatan ini menegakkan sifat complete consumption dari input, yang merupakan sifat penting untuk menjamin bahwa string yang diterima parser benar-benar merupakan anggota bahasa yang didefinisikan oleh grammar, tanpa sisa token yang tidak bermakna.

3.2.7 Tree Formatter

Untuk menampilkan struktur parse tree secara informatif dan mudah dibaca, diperlukan sebuah modul yang mampu memvisualisasikan Abstract Syntax Tree (AST) dalam bentuk hierarki yang jelas. Tree formatter pada proyek ini bertanggung jawab untuk mengubah node AST menjadi representasi pohon dengan karakter box-drawing sehingga relasi antar node tampak eksplisit. Bagian ini menjelaskan implementasi komponen tersebut dalam dua bagian besar: kode utama yang menangani proses pemformatan rekursif, serta sejumlah utilitas pembantu yang membangun wrapper nodes untuk berbagai konstruk grammar. Cuplikan yang disajikan berikut menggambarkan mekanisme inti bagaimana formatter bekerja dalam menyusun struktur AST menjadi representasi linear yang tetap mempertahankan bentuk pohnnya.

```
class TreeFormatter:

    def __init__(self, tokens: list[Token]):
        self.tokens = tokens
        self.output = []

    def format(self, node: ASTNode) -> str:
        self.output = []
        self._format_node(node, "", True, is_root=True)
        return "\n".join(self.output)
```

```

    def _format_node(self, node: Any, prefix: str, is_last: bool, is_root: bool = False):
        if node is None:
            return

        connector = "" if is_root else ("└─ " if is_last else "├─ ")

        if hasattr(node, '_node_type'):
            node_name = node._node_type
            self.output.append(f"{prefix}{connector}<{node_name}>")
            child_prefix = "" if is_root else prefix + ("    " if is_last
else " ")
            children = node._children if hasattr(node, '_children') else []
            for i, child in enumerate(children):
                self._format_node(child, child_prefix, i == len(children) - 1)

        elif isinstance(node, ASTNode):
            if isinstance(node, ParenthesizedExpression):
                children = self._get_children(node)
                for i, child in enumerate(children):
                    self._format_node(child, prefix, i == len(children) - 1)
                return

            node_name = self._get_node_name(node)
            self.output.append(f"{prefix}{connector}<{node_name}>")
            child_prefix = "" if is_root else prefix + ("    " if is_last
else " ")
            children = self._get_children(node)
            for i, child in enumerate(children):
                self._format_node(child, child_prefix, i == len(children) - 1)

        elif isinstance(node, str):
            self.output.append(f"{prefix}{connector}{node}")

        elif isinstance(node, (int, float, bool)):
            self.output.append(f"{prefix}{connector}{node}")

```

Cuplikan pertama mendeskripsikan inti dari kelas TreeFormatter, yang menjadi pusat logika formatisasi pohon. Konstruktor menerima daftar token asli sehingga terminal node seperti identifier dan literal dapat ditampilkan kembali secara konsisten pada struktur akhir. Metode format bertindak sebagai entry point untuk memulai proses pemformatan, menginisialisasi daftar keluaran, dan memanggil fungsi rekursif _format_node. Fungsi _format_node mengatur bagaimana setiap node ditampilkan, menentukan konektor visual seperti └─ atau ┌─, serta membangun indentasi menggunakan prefix yang diwariskan dari pemanggilan rekursif sebelumnya.

Implementasi ini mengakomodasi tiga kategori node sekaligus: wrapper nodes yang memiliki atribut _node_type, ASTNode sebenarnya seperti Program dan IfStatement, dan terminal seperti string atau literal numerik. Setiap jenis node ditangani secara berbeda untuk memastikan bahwa struktur pohon tetap akurat dan sesuai dengan konstruksi grammar.

Dengan pendekatan rekursif ini, module dapat berjalan secara generik terhadap seluruh jenis node selama node tersebut mengikuti antarmuka AST yang telah ditentukan.

Meskipun struktur utama pada cuplikan pertama berfokus pada bagaimana node diformat, proses tersebut membutuhkan berbagai fungsi utilitas untuk menentukan nama non-terminal, mengekstrak anak node, atau membuat wrapper nodes yang merepresentasikan kategori grammar tertentu seperti expression, identifier-list, atau declaration-part. Cuplikan berikut memperlihatkan kumpulan metode pendukung yang berfungsi mengurai node AST menjadi elemen-elemen yang siap diformat oleh `_format_node`. Komponen-komponen ini merupakan tulang punggung agar formatter memahami struktur AST secara semantik, bukan sekadar secara struktural.

```
def _get_node_name(self, node: ASTNode) -> str:
    mapping = {
        'Program': 'program',
        'Block': 'block',
        'VarDeclaration': 'var-declaration',
        'ConstDeclaration': 'const-declaration',
        ...
    }
    return mapping.get(type(node).__name__, type(node).__name__.lower())

def _get_children(self, node: ASTNode) -> list:
    children = []
    if isinstance(node, Program):
        children.append(self._create_program_header(node.name))
        if node.block.declarations:
            children.append(self._create_declarator_part(node.block.declarations))
            children.append(node.block.compound_statement)
            children.append(self._format_token('DOT', '.'))

    elif isinstance(node, VarDeclaration):
        children.append(self._create_identifier_list(node.identifiers))
        children.append(self._format_token('COLON', ':'))
        if isinstance(node.type_spec, SimpleType):
            children.append(node.type_spec)
        else:
            children.append(self._create_type_wrapper(node.type_spec))
        children.append(self._format_token('SEMICOLON', ';'))

    elif isinstance(node, BinaryOp):
        children.append(node.left)
        children.append(self._format_operator(node.operator))
        children.append(node.right)

    ...
    return children
```

Cuplikan kedua menunjukkan fungsi-fungsi yang bertugas membangun struktur internal untuk setiap jenis node AST. Fungsi `_get_node_name` memberikan pemetaan yang konsisten antara kelas Python dan nama non-terminal yang digunakan dalam pohon final, sehingga node seperti

VarDeclaration dapat ditampilkan sebagai <var-declaration> sesuai dengan aturan grammar. Dengan demikian, formatter tidak hanya menampilkan kelas Python, tetapi juga representasi konseptualnya dalam konteks bahasa yang dikompilasi.

Fungsi `_get_children` adalah komponen sentral yang mengekstrak semua anak dari node AST. Untuk setiap jenis node, terdapat pola pemrosesan yang berbeda: node Program menghasilkan header program, bagian deklarasi, dan statement utama, sedangkan node seperti VarDeclaration membentuk urutan <identifier-list>, simbol :, tipe variabel, dan simbol ;. Begitu pula untuk node ekspresi seperti BinaryOp, anak yang dihasilkan mengikuti struktur kiri-operator-kanan sebagaimana direpresentasikan pada grammar asli. Dengan memindahkan logika ini ke fungsi utilitas, formatter dapat mengolah seluruh elemen AST tanpa bergantung pada implementasi internal tiap node.

Pendekatan ini menghasilkan modularitas tinggi, memudahkan pemeliharaan, dan mendukung perluasan grammar tanpa harus memodifikasi inti metode `_format_node`. Node wrapper seperti identifier-list, expression, dan declaration-part juga dikonstruksi melalui helper class kecil yang mencantumkan `_node_type` dan daftar `_children`, memberikan fleksibilitas dalam mengelompokkan elemen-elemen grammar yang secara struktural berada pada level yang sama.

BAB IV: Pengujian

Pengujian dilakukan untuk memastikan bahwa sistem yang dibangun—khususnya komponen Lexer dan Abstract Syntax Tree (AST)—telah berfungsi sesuai dengan spesifikasi bahasa Pascal-S yang telah dimodifikasi ke dalam versi berbahasa Indonesia. Secara konseptual, tahap pengujian ini bertujuan untuk memverifikasi keakuratan proses analisis leksikal dalam mengenali dan mengklasifikasikan token, serta memastikan struktur AST yang dihasilkan merepresentasikan hubungan sintaktis dan semantik program secara benar. Selain itu, pengujian ini juga dimaksudkan untuk menilai ketahanan sistem terhadap kesalahan sintaks, kemampuan penanganan kata kunci majemuk (hyphenated keywords).

4.1 Pengujian Test Cases

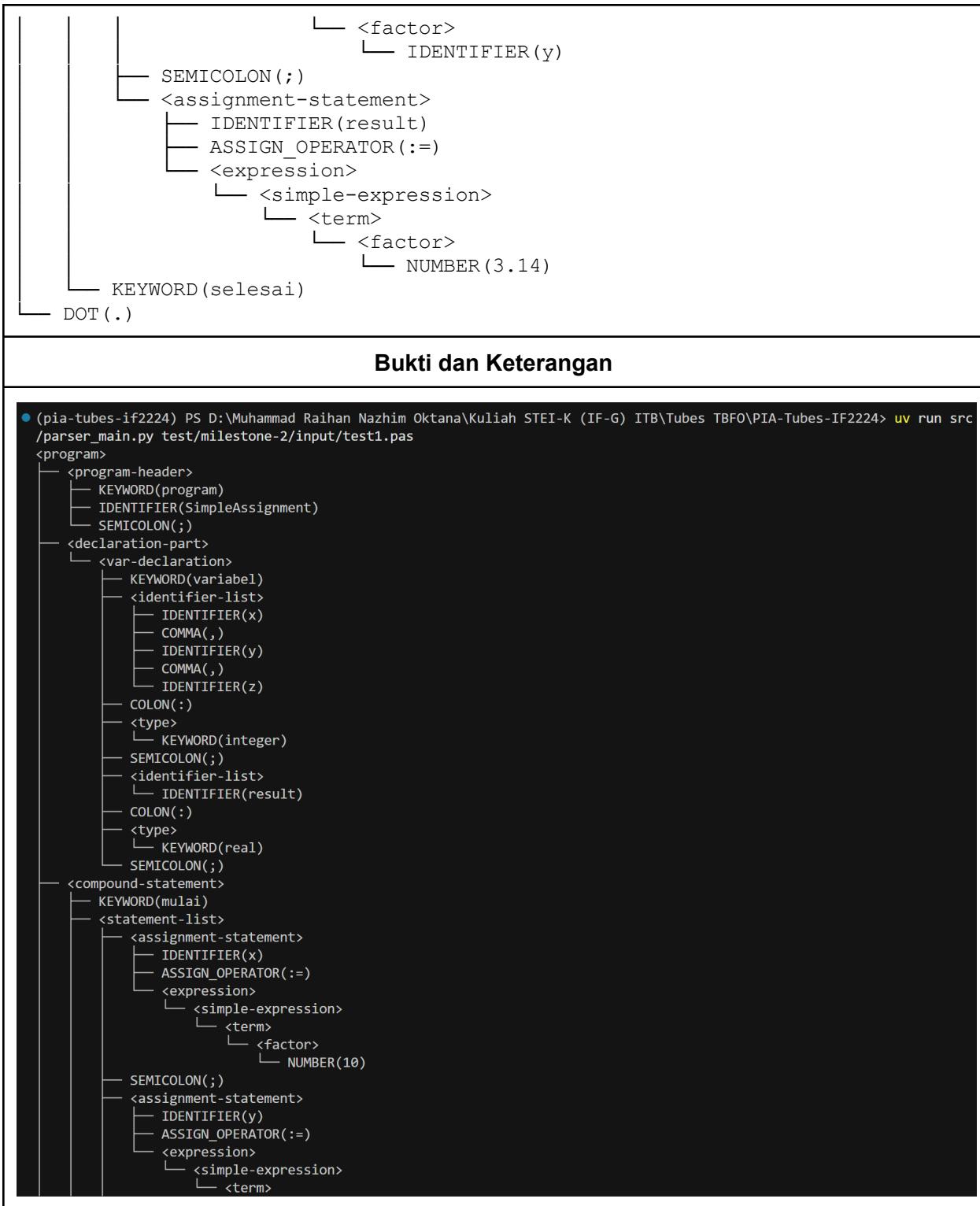
Tabel I. SimpleAssignmentTest

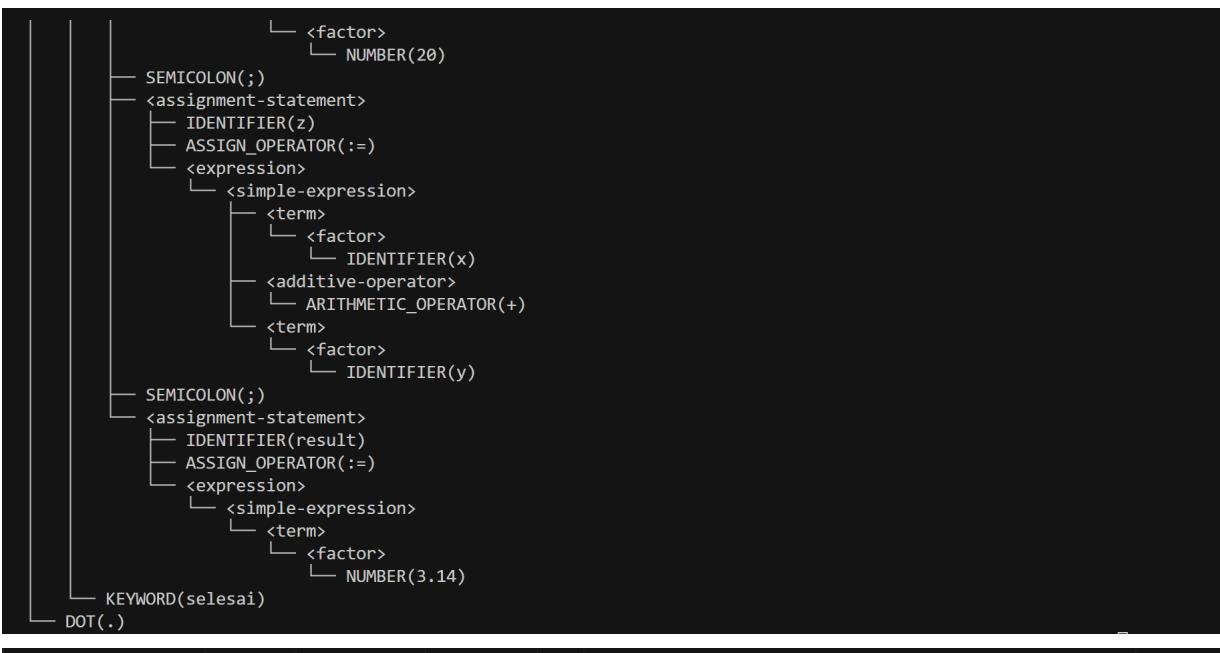
Input
program SimpleAssignment; variabel x, y, z: integer; result: real; mulai x := 10; y := 20; z := x + y; result := 3.14 selesai.
Output

```

<program>
  └── <program-header>
    ├── KEYWORD(program)
    ├── IDENTIFIER(SimpleAssignment)
    └── SEMICOLON(;)
  └── <declaration-part>
    └── <var-declaration>
      ├── KEYWORD(variabel)
      ├── <identifier-list>
        ├── IDENTIFIER(x)
        ├── COMMA(,)
        ├── IDENTIFIER(y)
        ├── COMMA(,)
        └── IDENTIFIER(z)
      ├── COLON(:)
      ├── <type>
        └── KEYWORD(integer)
      └── SEMICOLON(;)
    └── <identifier-list>
      └── IDENTIFIER(result)
    └── COLON(:)
    └── <type>
      └── KEYWORD(real)
    └── SEMICOLON(;)
  └── <compound-statement>
    ├── KEYWORD(mulai)
    └── <statement-list>
      └── <assignment-statement>
        ├── IDENTIFIER(x)
        ├── ASSIGN_OPERATOR(:=)
        └── <expression>
          └── <simple-expression>
            └── <term>
              └── <factor>
                └── NUMBER(10)
      └── SEMICOLON(;)
      └── <assignment-statement>
        ├── IDENTIFIER(y)
        ├── ASSIGN_OPERATOR(:=)
        └── <expression>
          └── <simple-expression>
            └── <term>
              └── <factor>
                └── NUMBER(20)
      └── SEMICOLON(;)
      └── <assignment-statement>
        ├── IDENTIFIER(z)
        ├── ASSIGN_OPERATOR(:=)
        └── <expression>
          └── <simple-expression>
            └── <term>
              └── <factor>
                └── IDENTIFIER(x)
            └── <additive-operator>
              └── ARITHMETIC_OPERATOR(+)
            └── <term>

```





```

● PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src/parser_main.py test/milestone-2/input/test1.pas --check --output
Parse tree saved to: test\milestone-2\output\test1.txt
[PASS] Parse tree matches expected!

```

Sukses menguji kemampuan parser dalam menangani operasi simple assignment ke variabel, hasil uji sudah benar.

Tabel II. ConditionalTest

Input
<pre> program ConditionalTest; variabel score: integer; grade: char; mulai score := 85; jika score >= 80 maka grade := 'A' selain-itu jika score >= 70 maka grade := 'B' selain-itu grade := 'C' selesai. </pre>
Output
<pre> <program> <program-header> KEYWORD (program) IDENTIFIER (ConditionalTest) SEMICOLON (;) <declaration-part> </pre>

```

    └ <var-declaration>
        ├── KEYWORD(variabel)
        ├── <identifier-list>
        │   └ IDENTIFIER(score)
        ├── COLON(:)
        ├── <type>
        │   └ KEYWORD(integer)
        ├── SEMICOLON(;)
        ├── <identifier-list>
        │   └ IDENTIFIER(grade)
        ├── COLON(:)
        ├── <type>
        │   └ KEYWORD(char)
        ├── SEMICOLON(;)

    └ <compound-statement>
        ├── KEYWORD(mulai)
        └ <statement-list>
            └ <assignment-statement>
                ├── IDENTIFIER(score)
                ├── ASSIGN_OPERATOR(:=)
                └ <expression>
                    └ <simple-expression>
                        └ <term>
                            └ <factor>
                                └ NUMBER(85)

            └ SEMICOLON(;)

    └ <if-statement>
        ├── KEYWORD(jika)
        └ <expression>
            └ <simple-expression>
                └ <term>
                    └ <factor>
                        └ IDENTIFIER(score)

        └ <relational-operator>
            └ RELATIONAL_OPERATOR(>=)

        └ <simple-expression>
            └ <term>
                └ <factor>
                    └ NUMBER(80)

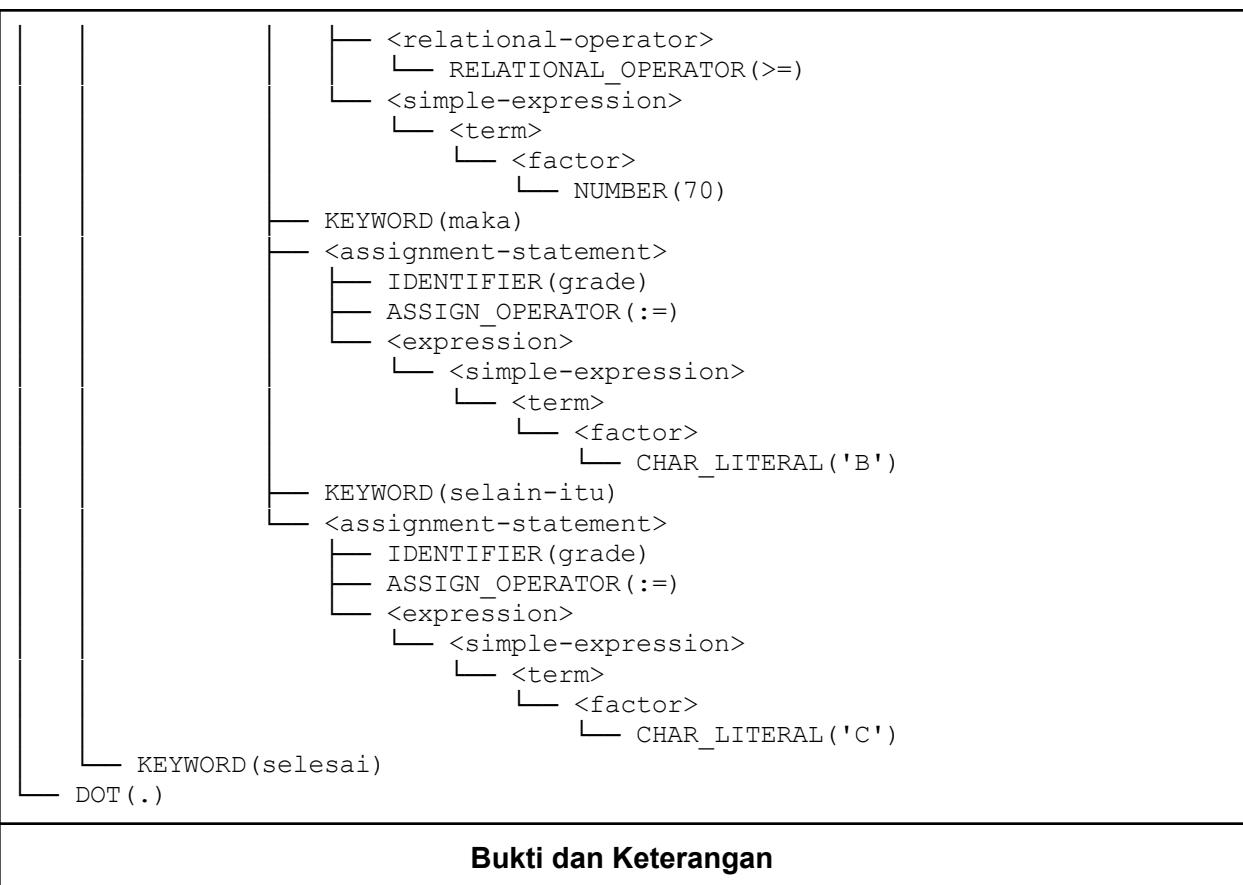
        └ KEYWORD(maka)

    └ <assignment-statement>
        ├── IDENTIFIER(grade)
        ├── ASSIGN_OPERATOR(:=)
        └ <expression>
            └ <simple-expression>
                └ <term>
                    └ <factor>
                        └ CHAR_LITERAL('A')

    └ KEYWORD(selain-itu)

    └ <if-statement>
        ├── KEYWORD(jika)
        └ <expression>
            └ <simple-expression>
                └ <term>
                    └ <factor>
                        └ IDENTIFIER(score)

```



● (pia-tubes-if2224) PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src /parser_main.py test/milestone-2/input/test2.pas

```
<program>
  <program-header>
    KEYWORD(program)
    IDENTIFIER(ConditionalTest)
    SEMICOLON(;)
  <declaration-part>
    <var-declaration>
      KEYWORD(variabel)
      <identifier-list>
        IDENTIFIER(score)
      COLON(:)
      <type>
        KEYWORD(integer)
      SEMICOLON(;)
      <identifier-list>
        IDENTIFIER(grade)
      COLON(:)
      <type>
        KEYWORD(char)
      SEMICOLON(;)
    <compound-statement>
      KEYWORD(mulai)
      <statement-list>
        <assignment-statement>
          IDENTIFIER(score)
          ASSIGN_OPERATOR(:=)
          <expression>
            <simple-expression>
              <term>
                <factor>
                  NUMBER(85)
            SEMICOLON(;)
          <if-statement>
            KEYWORD(jika)
            <expression>
              <simple-expression>
                <term>
                  <factor>
                    IDENTIFIER(score)
                <relational-operator>
                  RELATIONAL_OPERATOR(>=)
                <simple-expression>
```



Sukses menguji kemampuan parser dalam menangani kondisional if-else, hasil uji sudah benar.

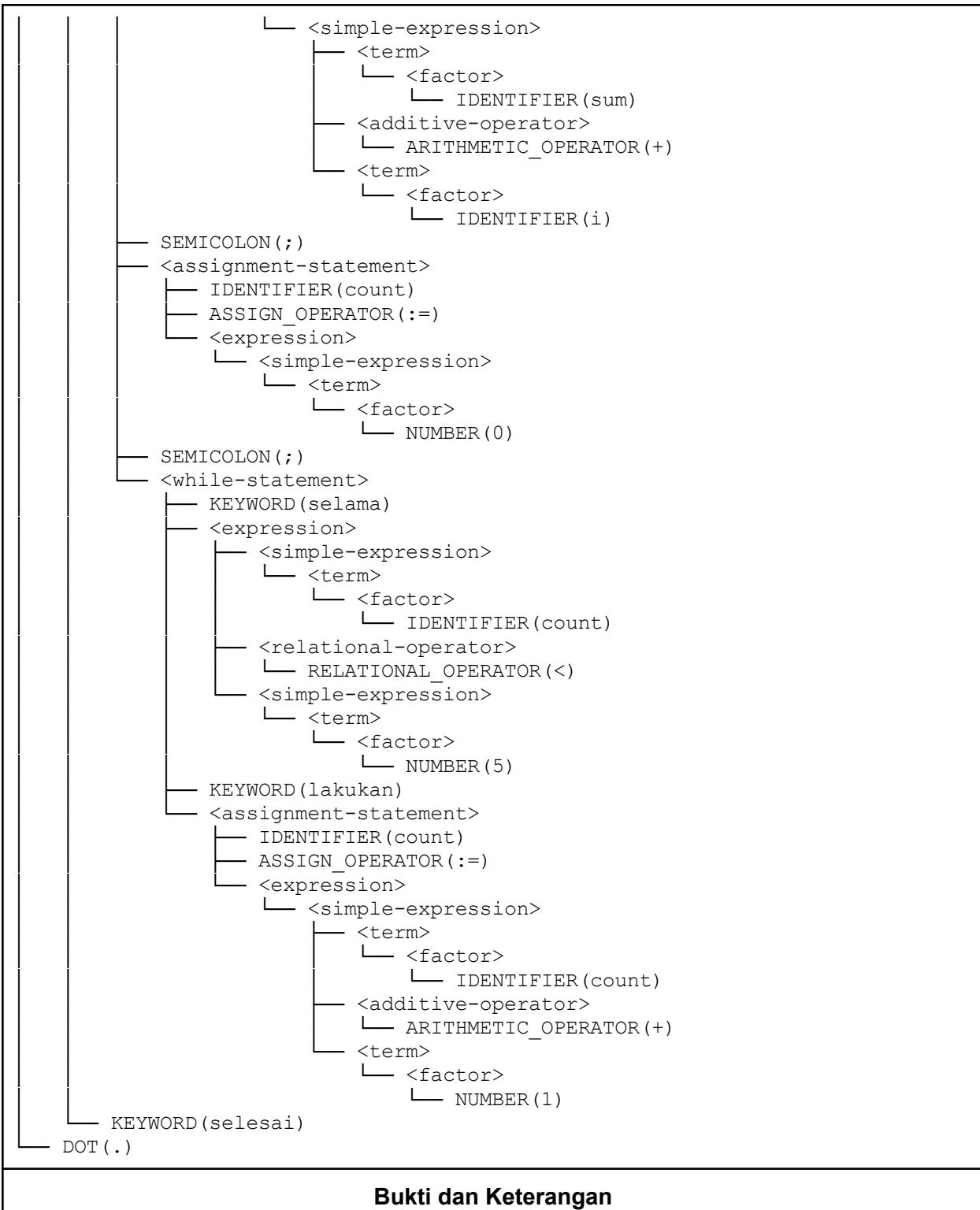
Tabel III. LoopTest

Input
<pre> program LoopTest; variabel i, sum, count: integer; mulai sum := 0; untuk i := 1 ke 10 lakukan sum := sum + i; count := 0; </pre>

```
selama count < 5 lakukan
    count := count + 1
selesai.
```

Output

```
<program>
└── <program-header>
    ├── KEYWORD(program)
    ├── IDENTIFIER(LoopTest)
    └── SEMICOLON(;)
└── <declaration-part>
    └── <var-declaration>
        ├── KEYWORD(variabel)
        ├── <identifier-list>
        │   ├── IDENTIFIER(i)
        │   ├── COMMA(,)
        │   ├── IDENTIFIER(sum)
        │   ├── COMMA(,)
        │   └── IDENTIFIER(count)
        ├── COLON(:)
        └── <type>
            └── KEYWORD(integer)
            └── SEMICOLON(;)
└── <compound-statement>
    ├── KEYWORD(mulai)
    └── <statement-list>
        ├── <assignment-statement>
        │   ├── IDENTIFIER(sum)
        │   ├── ASSIGN_OPERATOR(:=)
        │   └── <expression>
        │       └── <simple-expression>
        │           └── <term>
        │               └── <factor>
        │                   └── NUMBER(0)
        └── SEMICOLON(;)
        └── <for-statement>
            ├── KEYWORD(untuk)
            ├── IDENTIFIER(i)
            ├── ASSIGN_OPERATOR(:=)
            └── <expression>
                └── <simple-expression>
                    └── <term>
                        └── <factor>
                            └── NUMBER(1)
            └── KEYWORD(ke)
            └── <expression>
                └── <simple-expression>
                    └── <term>
                        └── <factor>
                            └── NUMBER(10)
        └── KEYWORD(lakukan)
        └── <assignment-statement>
            ├── IDENTIFIER(sum)
            ├── ASSIGN_OPERATOR(:=)
            └── <expression>
```



```
● (pia-tubes-if2224) PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src /parser_main.py test/milestone-2/input/test3.pas
<program>
  └── <program-header>
    ├── KEYWORD(program)
    ├── IDENTIFIER(LoopTest)
    └── SEMICOLON(;)
  └── <declaration-part>
    └── <var-declaration>
      ├── KEYWORD(variabel)
      ├── <identifier-list>
      │   ├── IDENTIFIER(i)
      │   ├── COMMA,,
      │   ├── IDENTIFIER(sum)
      │   ├── COMMA,,
      │   └── IDENTIFIER(count)
      ├── COLON(:)
      ├── <type>
      │   └── KEYWORD(integer)
      └── SEMICOLON(;)
  └── <compound-statement>
    ├── KEYWORD(mulai)
    └── <statement-list>
      ├── <assignment-statement>
      │   ├── IDENTIFIER(sum)
      │   ├── ASSIGN_OPERATOR(:=)
      │   └── <expression>
      │       └── <simple-expression>
      │           └── <term>
      │               └── <factor>
      │                   └── NUMBER(0)
      └── SEMICOLON(;)
      └── <for-statement>
        ├── KEYWORD(untuk)
        ├── IDENTIFIER(i)
        ├── ASSIGN_OPERATOR(:=)
        └── <expression>
            └── <simple-expression>
                └── <term>
                    └── <factor>
                        └── NUMBER(1)
      └── KEYWORD(ke)
      └── <expression>
          └── <simple-expression>
```

```

    └── <term>
        └── <factor>
            └── NUMBER(10)
    └── KEYWORD(lakukan)
    └── <assignment-statement>
        ├── IDENTIFIER(sum)
        ├── ASSIGN_OPERATOR(:=)
        └── <expression>
            └── <simple-expression>
                ├── <term>
                │   └── <factor>
                │       └── IDENTIFIER(sum)
                ├── <additive-operator>
                │   └── ARITHMETIC_OPERATOR(+)
                └── <term>
                    └── <factor>
                        └── IDENTIFIER(i)
    └── SEMICOLON(;)
    └── <assignment-statement>
        ├── IDENTIFIER(count)
        ├── ASSIGN_OPERATOR(:=)
        └── <expression>
            └── <simple-expression>
                └── <term>
                    └── <factor>
                        └── NUMBER(0)
    └── SEMICOLON(;)
    └── <while-statement>
        ├── KEYWORD(selama)
        └── <expression>
            └── <simple-expression>
                ├── <term>
                │   └── <factor>
                │       └── IDENTIFIER(count)
                ├── <relational-operator>
                │   └── RELATIONAL_OPERATOR(<)
                └── <simple-expression>
                    └── <term>
                        └── <factor>
                            └── NUMBER(5)
    └── KEYWORD(lakukan)
    └── <assignment-statement>

```

```

    └── IDENTIFIER(count)
    └── ASSIGN_OPERATOR(:=)
    └── <expression>
        └── <simple-expression>
            ├── <term>
            │   └── <factor>
            │       └── IDENTIFIER(count)
            ├── <additive-operator>
            │   └── ARITHMETIC_OPERATOR(+)
            └── <term>
                └── <factor>
                    └── NUMBER(1)
    └── KEYWORD(selesai)
    └── DOT(..)

```

```

● PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src/parser_main.py test/milestone-2/input/test3.pas --check --output
Parse tree saved to: test\milestone-2\output\test3.txt
[PASS] Parse tree matches expected!

```

Sukses menguji kemampuan parser dalam menangani kondisi looping for loop dan while loop, hasil uji sudah benar.

Tabel IV. ProcedureFunctionTest

Input

```

program ProcFuncTest;
variabel
    x, y, result: integer;

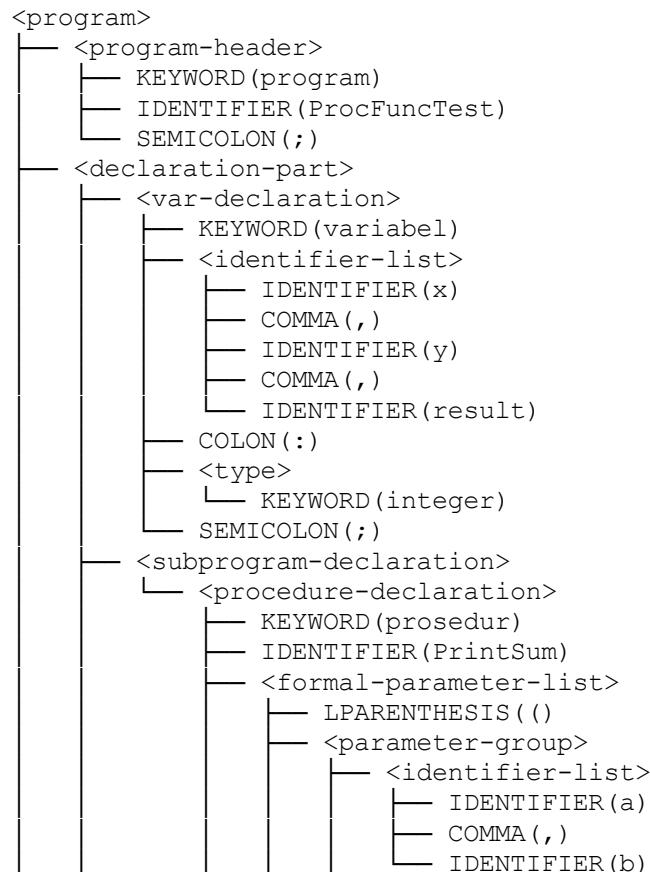
prosedur PrintSum(a, b: integer);
variabel
    total: integer;
mulai
    total := a + b;
    writeln('Sum is: ', total)
selesai;

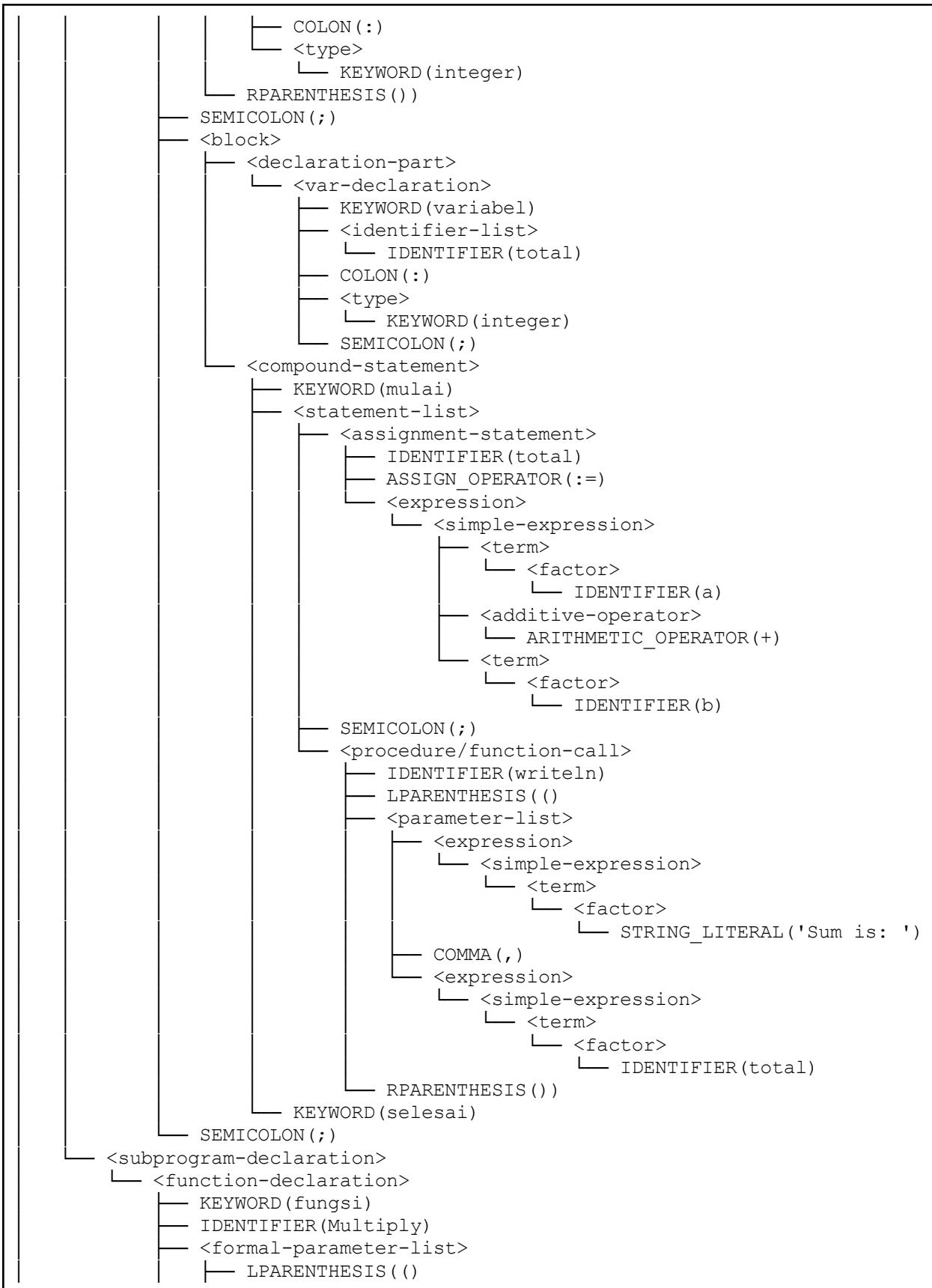
fungsi Multiply(m, n: integer): integer;
mulai
    Multiply := m * n
selesai;

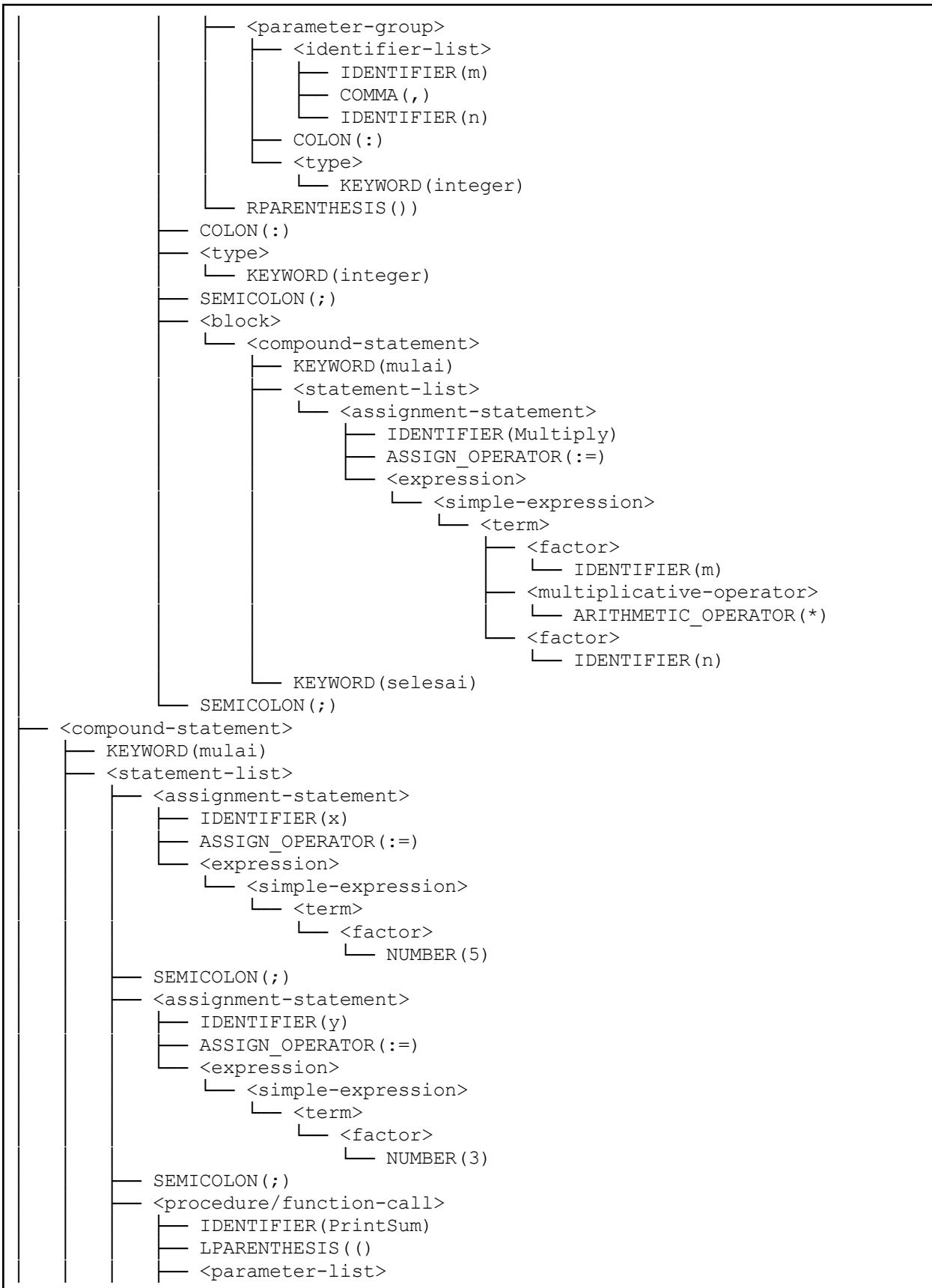
mulai
    x := 5;
    y := 3;
    PrintSum(x, y);
    result := Multiply(x, y);
    writeln('Product is: ', result)
selesai.

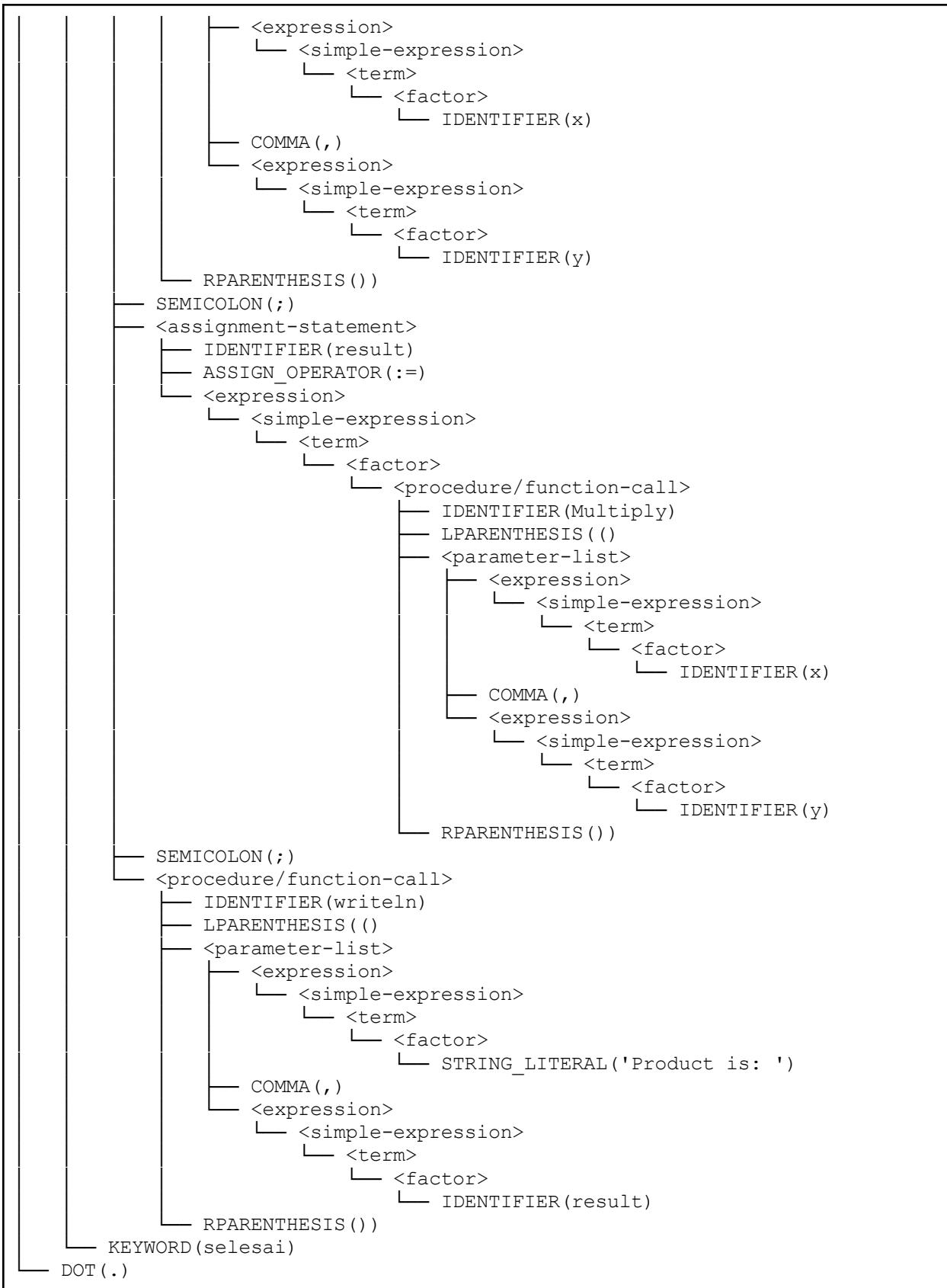
```

Output



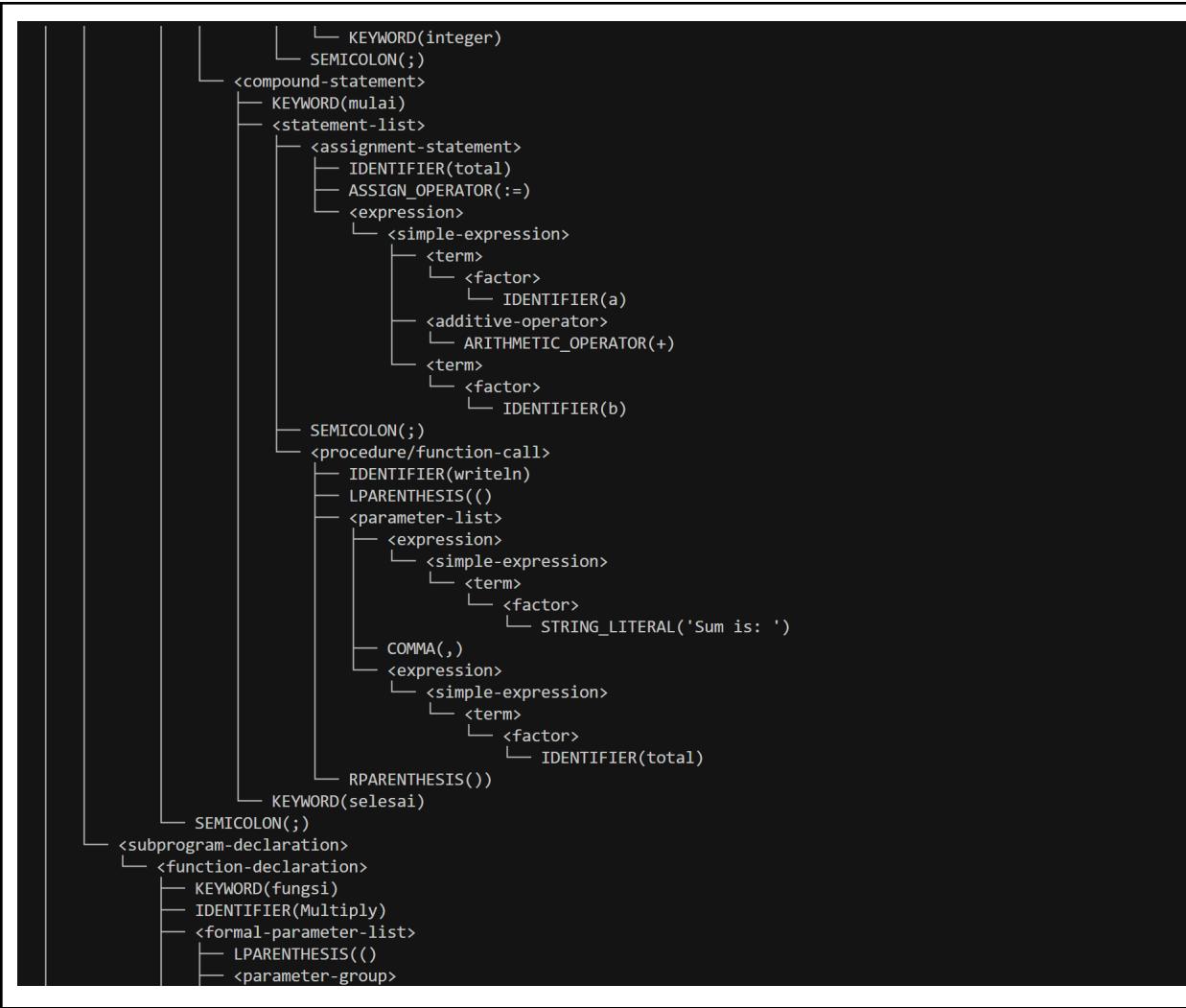






Bukti dan Keterangan

```
● (pia-tubes-if2224) PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-6) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src  
/parser_main.py test/milestone-2/input/test4.pas  
<program>  
  └── <program-header>  
    ├── KEYWORD(program)  
    ├── IDENTIFIER(ProcFuncTest)  
    └── SEMICOLON();  
  └── <declaration-part>  
    └── <var-declaration>  
      ├── KEYWORD(variabel)  
      ├── <identifier-list>  
      │   ├── IDENTIFIER(x)  
      │   ├── COMMA(),  
      │   ├── IDENTIFIER(y)  
      │   ├── COMMA(),  
      │   └── IDENTIFIER(result)  
      ├── COLON(:)  
      └── <type>  
        └── KEYWORD(integer)  
    └── SEMICOLON();  
  └── <subprogram-declaration>  
    └── <procedure-declaration>  
      ├── KEYWORD(prosedur)  
      ├── IDENTIFIER(PrintSum)  
      └── <formal-parameter-list>  
        ├── LPARENTHESIS()  
        └── <parameter-group>  
          └── <identifier-list>  
            ├── IDENTIFIER(a)  
            ├── COMMA(),  
            └── IDENTIFIER(b)  
        ├── COLON(:)  
        └── <type>  
          └── KEYWORD(integer)  
      └── RPARENTHESIS()  
    └── SEMICOLON();  
  └── <block>  
    └── <declaration-part>  
      └── <var-declaration>  
        ├── KEYWORD(variabel)  
        ├── <identifier-list>  
        │   └── IDENTIFIER(total)  
        ├── COLON(:)  
        └── <type>
```



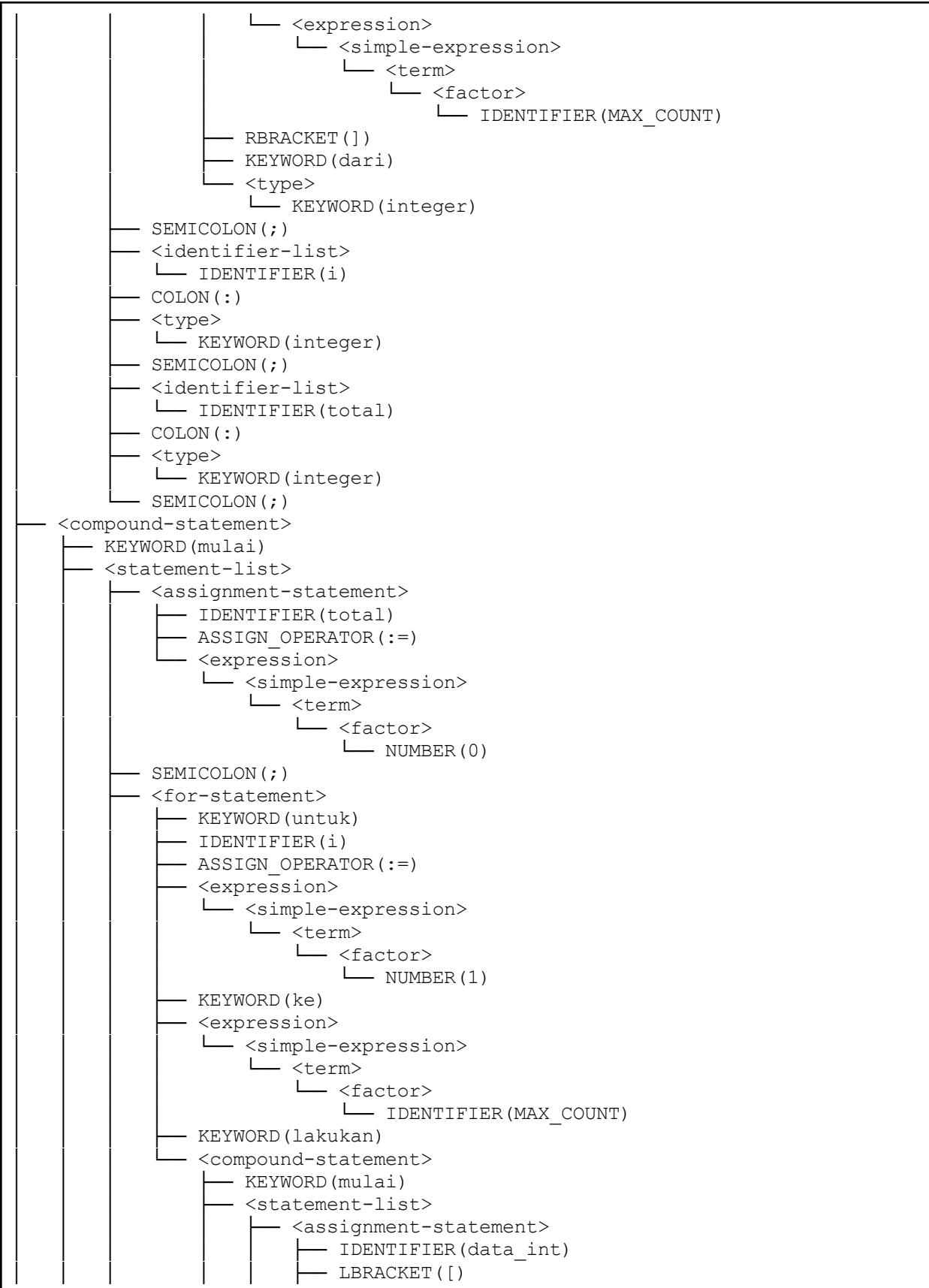




Sukses menguji kemampuan parser dalam menangani kondisi subprogram dalam bentuk prosedur ataupun fungsi, hasil uji sudah benar.

Tabel V. ArrayConstantTest

Input
<pre>program ArrayStrictTest; konstanta MAX_COUNT = 10; variabel data_int: larik[1..MAX_COUNT] dari integer; i: integer; total: integer; mulai total := 0; untuk i := 1 ke MAX_COUNT lakukan mulai data_int[i] := i * 2; total := total + data_int[i]; selesai; writeln('Sum: ', total); selesai.</pre>
Output
<pre><program> <program-header> KEYWORD(program) IDENTIFIER(ArrayStrictTest) SEMICOLON(;) <declaration-part> <const-declaration> KEYWORD(konstanta) IDENTIFIER(MAX_COUNT) ASSIGN_OPERATOR(=) NUMBER(10) SEMICOLON(;) <var-declaration> KEYWORD(variabel) <identifier-list> IDENTIFIER(data_int) COLON(:) <type> <array-type> KEYWORD(larik) LBRACKET([]) <range> <expression> <simple-expression> <term> <factor> NUMBER(1) RANGE_OPERATOR(..)</pre>



```

    └── <expression>
        └── <simple-expression>
            └── <term>
                └── <factor>
                    └── IDENTIFIER(i)
    └── RBRACKET []
    └── ASSIGN_OPERATOR(:=)
    └── <expression>
        └── <simple-expression>
            └── <term>
                └── <factor>
                    └── IDENTIFIER(i)
                └── <multiplicative-operator>
                    └── ARITHMETIC_OPERATOR(*)
                └── <factor>
                    └── NUMBER(2)
    └── SEMICOLON(;)
    └── <assignment-statement>
        └── IDENTIFIER(total)
        └── ASSIGN_OPERATOR(:=)
        └── <expression>
            └── <simple-expression>
                └── <term>
                    └── <factor>
                        └── IDENTIFIER(total)
                └── <additive-operator>
                    └── ARITHMETIC_OPERATOR(+)
                └── <term>
                    └── <factor>
                        └── IDENTIFIER(data_int)
                        └── LBRACKET []
                        └── <expression>
                            └── <simple-expression>
                                └── <term>
                                    └── <factor>
                                        └── IDENTIFIER(i)
    └── RBRACKET []
    └── KEYWORD(selesai)
    └── SEMICOLON(;)
    └── <procedure/function-call>
        └── IDENTIFIER(writeln)
        └── LPARENTHESIS()
        └── <parameter-list>
            └── <expression>
                └── <simple-expression>
                    └── <term>
                        └── <factor>
                            └── STRING_LITERAL('Sum: ')
            └── COMMA(,)
            └── <expression>
                └── <simple-expression>
                    └── <term>
                        └── <factor>
                            └── IDENTIFIER(total)
    └── RPARENTHESIS()
    └── KEYWORD(selesai)

```

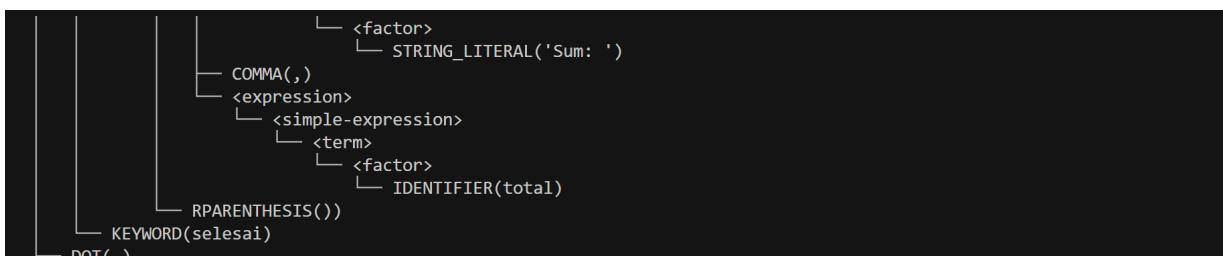
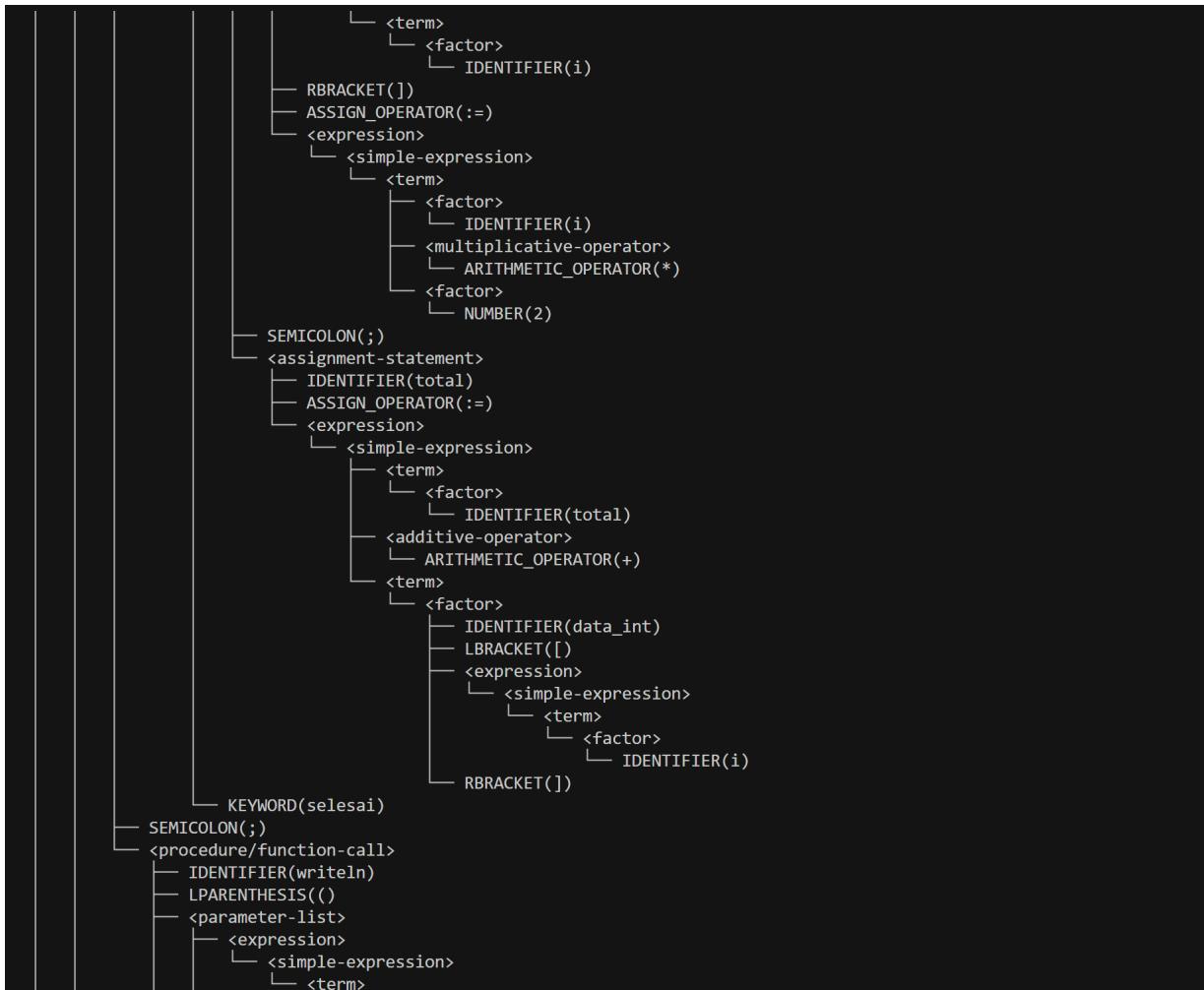
└─ DOT(.)

Bukti dan Keterangan

```
● (pia-tubes-if2224) PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src
/parser_main.py test/milestone-2/input/test5.pas
<program>
  <program-header>
    └─ KEYWORD(program)
    └─ IDENTIFIER(ArrayStrictTest)
    └─ SEMICOLON(;)
  <declaration-part>
    <const-declaration>
      └─ KEYWORD(konstanta)
      └─ IDENTIFIER(MAX_COUNT)
      └─ ASSIGN_OPERATOR(=)
      └─ NUMBER(10)
      └─ SEMICOLON(;)
    <var-declaration>
      └─ KEYWORD(variabel)
      <identifier-list>
        └─ IDENTIFIER(data_int)
      └─ COLON(:)
      <type>
        └─ <array-type>
          └─ KEYWORD(larik)
          └─ LBRACKET([)
          <range>
            <expression>
              └─ <simple-expression>
                └─ <term>
                  └─ <factor>
                    └─ NUMBER(1)
            └─ RANGE_OPERATOR(..)
            <expression>
              └─ <simple-expression>
                └─ <term>
                  └─ <factor>
                    └─ IDENTIFIER(MAX_COUNT)
          └─ RBRACKET()]
          └─ KEYWORD(dari)
          <type>
            └─ KEYWORD(integer)
      └─ SEMICOLON(;)
      <identifier-list>
        └─ IDENTIFIER(i)
      └─ COLON(:)
      <type>
```

```
    └── KEYWORD(integer)
    └── SEMICOLON(;)
    └── <identifier-list>
        └── IDENTIFIER(total)
    └── COLON(:)
    └── <type>
        └── KEYWORD(integer)
    └── SEMICOLON(;)

<compound-statement>
    └── KEYWORD(mulai)
    └── <statement-list>
        └── <assignment-statement>
            └── IDENTIFIER(total)
            └── ASSIGN_OPERATOR(:=)
            └── <expression>
                └── <simple-expression>
                    └── <term>
                        └── <factor>
                            └── NUMBER(0)
        └── SEMICOLON(;)
        └── <for-statement>
            └── KEYWORD(until)
            └── IDENTIFIER(i)
            └── ASSIGN_OPERATOR(:=)
            └── <expression>
                └── <simple-expression>
                    └── <term>
                        └── <factor>
                            └── NUMBER(1)
        └── KEYWORD(ke)
        └── <expression>
            └── <simple-expression>
                └── <term>
                    └── <factor>
                        └── IDENTIFIER(MAX_COUNT)
        └── KEYWORD(lakukan)
        └── <compound-statement>
            └── KEYWORD(mulai)
            └── <statement-list>
                └── <assignment-statement>
                    └── IDENTIFIER(data_int)
                    └── LBRACKET([])
                └── <expression>
                    └── <simple-expression>
```



```
● PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src/parser_main.py test/milestone-2/input/test5.pas --check --output
Parse tree saved to: test\milestone-2\output\test5.txt
[PASS] Parse tree matches expected!
```

Sukses menguji kemampuan parser dalam menangani tipe data array constant, hasil uji sudah benar.

Tabel VI. ComplexExpressionTest

Input
program ComplexExpressionTest; variabel a, b, c: integer;

```

x, y: real;
flag, result: boolean;
mulai
    a := 10;
    b := 5;
    c := 3;

    x := (a + b) * c / 2.0;
    y := a bagi b + c mod 2;

    flag := (a > b) dan (b < c);
    result := tidak flag atau (a <> c);

    jika (a >= 10) dan ((b <= 5) atau (c = 3)) maka
        writeln('Condition is true')
    selain-itu
        writeln('Condition is false')
selesai.

```

Output

```

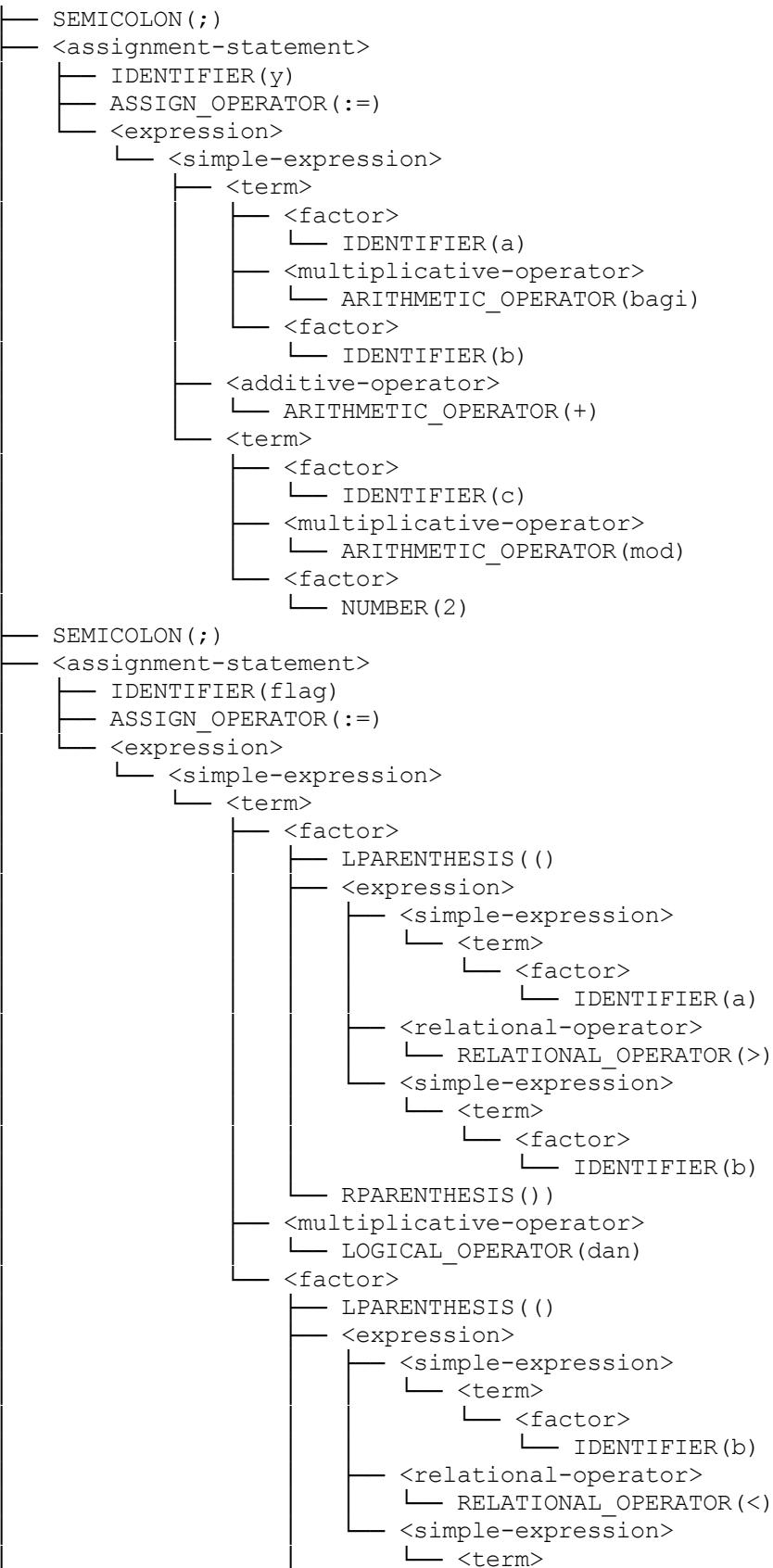
<program>
└── <program-header>
    ├── KEYWORD(program)
    ├── IDENTIFIER(ComplexExpressionTest)
    └── SEMICOLON(;)
└── <declaration-part>
    └── <var-declaration>
        ├── KEYWORD(variabel)
        └── <identifier-list>
            ├── IDENTIFIER(a)
            ├── COMMA(,)
            ├── IDENTIFIER(b)
            ├── COMMA(,)
            └── IDENTIFIER(c)
        ├── COLON(:)
        └── <type>
            └── KEYWORD(integer)
        └── SEMICOLON(;)
        └── <identifier-list>
            ├── IDENTIFIER(x)
            ├── COMMA(,)
            └── IDENTIFIER(y)
        ├── COLON(:)
        └── <type>
            └── KEYWORD(real)
        └── SEMICOLON(;)
        └── <identifier-list>
            ├── IDENTIFIER(flag)
            ├── COMMA(,)
            └── IDENTIFIER(result)
        ├── COLON(:)
        └── <type>
            └── KEYWORD(boolean)
        └── SEMICOLON(;)
└── <compound-statement>

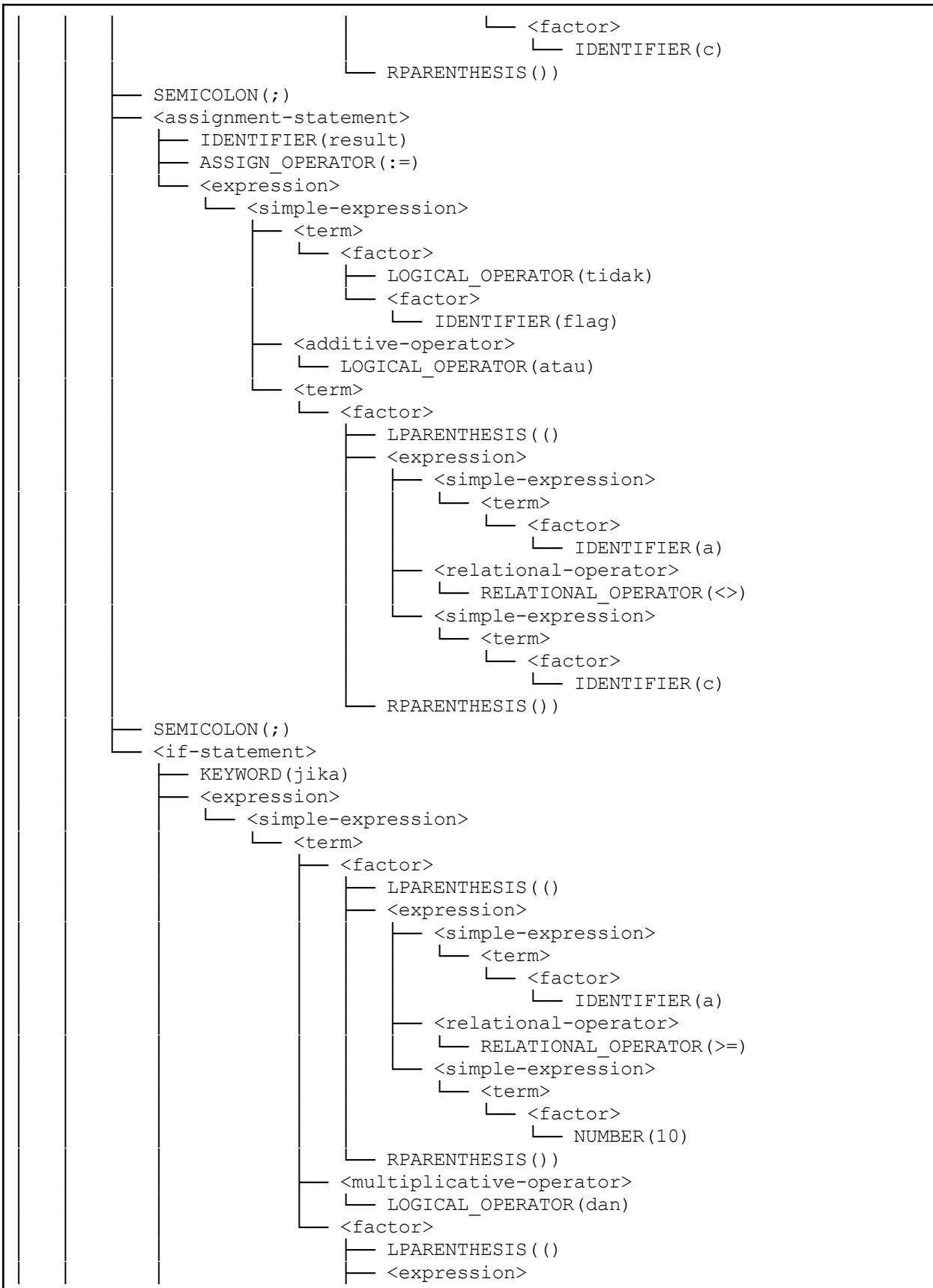
```

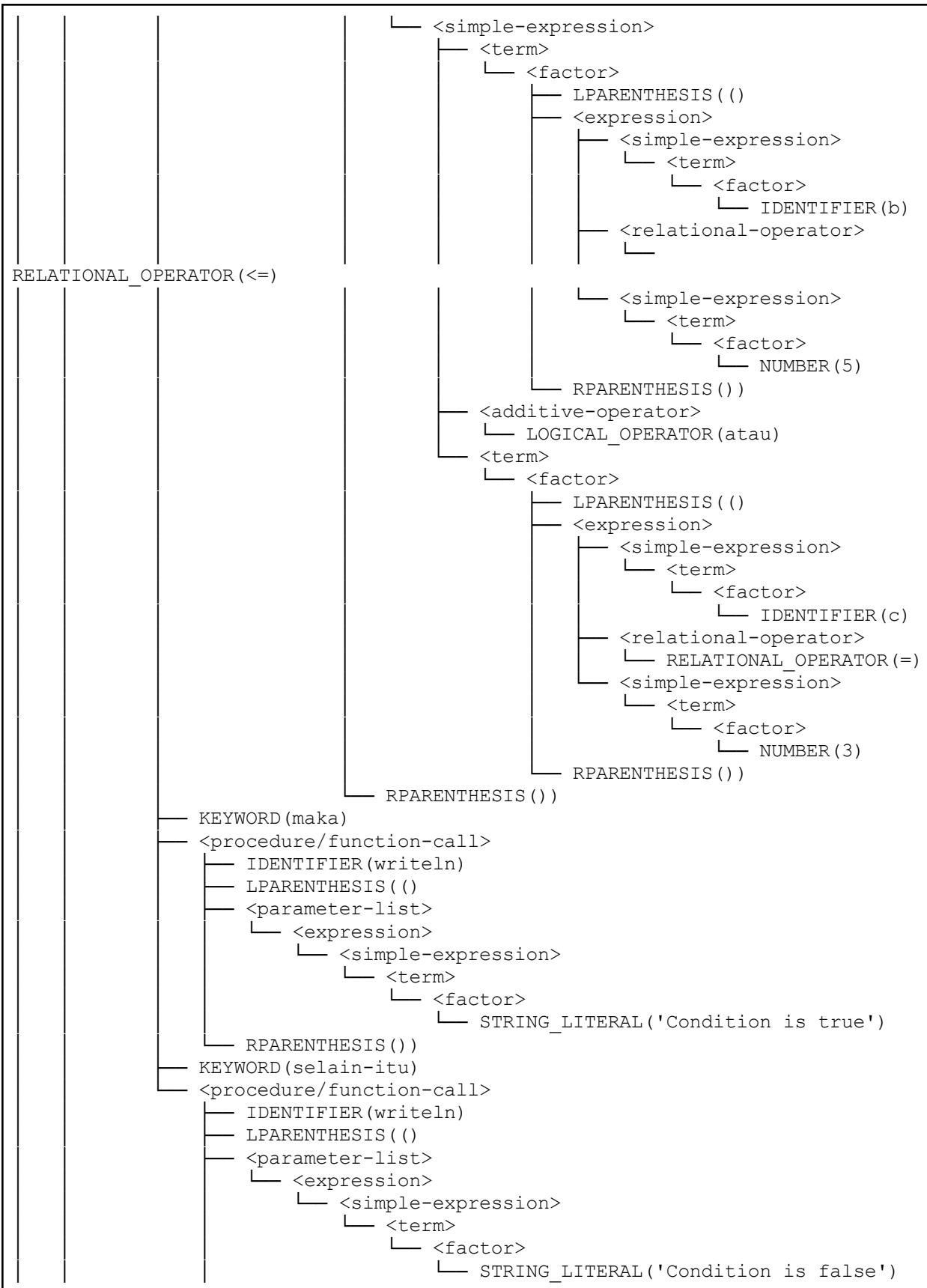
```

  └── KEYWORD (mulai)
  └── <statement-list>
      ├── <assignment-statement>
      │   ├── IDENTIFIER (a)
      │   ├── ASSIGN_OPERATOR (:=)
      │   └── <expression>
      │       └── <simple-expression>
      │           └── <term>
      │               └── <factor>
      │                   └── NUMBER (10)
      └── SEMICOLON (;)
      └── <assignment-statement>
          ├── IDENTIFIER (b)
          ├── ASSIGN_OPERATOR (:=)
          └── <expression>
              └── <simple-expression>
                  └── <term>
                      └── <factor>
                          └── NUMBER (5)
      └── SEMICOLON (;)
      └── <assignment-statement>
          ├── IDENTIFIER (c)
          ├── ASSIGN_OPERATOR (:=)
          └── <expression>
              └── <simple-expression>
                  └── <term>
                      └── <factor>
                          └── NUMBER (3)
      └── SEMICOLON (;)
      └── <assignment-statement>
          ├── IDENTIFIER (x)
          ├── ASSIGN_OPERATOR (:=)
          └── <expression>
              └── <simple-expression>
                  └── <term>
                      └── <factor>
                          └── LPARENTHESIS (())
                          └── <expression>
                              └── <simple-expression>
                                  └── <term>
                                      └── <factor>
                                          └── IDENTIFIER (a)
                                  └── <additive-operator>
                                      └── ARITHMETIC_OPERATOR (+)
                                  └── <term>
                                      └── <factor>
                                          └── IDENTIFIER (b)
                          └── RPARENTHESIS ())
              └── <multiplicative-operator>
                  └── ARITHMETIC_OPERATOR (*)
              └── <factor>
                  └── IDENTIFIER (c)
              └── <multiplicative-operator>
                  └── ARITHMETIC_OPERATOR (/)
              └── <factor>
                  └── NUMBER (2.0)

```







```
    └── RPARENTHESIS()
└── KEYWORD(selesai)
└── DOT(.)
```

Bukti dan Keterangan

```
● (pia-tubes-if2224) PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src
/parser_main.py test/milestone-2/input/test6.pas
<program>
  ├── <program-header>
  │   ├── KEYWORD(program)
  │   ├── IDENTIFIER(ComplexExpressionTest)
  │   └── SEMICOLON(;)
  ├── <declaration-part>
  │   └── <var-declaration>
  │       ├── KEYWORD(variabel)
  │       ├── <identifier-list>
  │       │   ├── IDENTIFIER(a)
  │       │   ├── COMMA(,)
  │       │   ├── IDENTIFIER(b)
  │       │   ├── COMMA(,)
  │       │   ├── IDENTIFIER(c)
  │       │   ├── COLON(:)
  │       │   ├── <type>
  │       │       └── KEYWORD(integer)
  │       │   ├── SEMICOLON(;)
  │       │   ├── <identifier-list>
  │       │       ├── IDENTIFIER(x)
  │       │       ├── COMMA(,)
  │       │       ├── IDENTIFIER(y)
  │       │       ├── COLON(:)
  │       │       ├── <type>
  │       │           └── KEYWORD(real)
  │       │       ├── SEMICOLON(;)
  │       │       ├── <identifier-list>
  │       │           ├── IDENTIFIER(flag)
  │       │           ├── COMMA(,)
  │       │           ├── IDENTIFIER(result)
  │       │           ├── COLON(:)
  │       │           ├── <type>
  │       │               └── KEYWORD(boolean)
  │       │           ├── SEMICOLON(;)
  │       └── <compound-statement>
  │           ├── KEYWORD(mulai)
  │           └── <statement-list>
  │               └── <assignment-statement>
  │                   ├── IDENTIFIER(a)
  │                   ├── ASSIGN_OPERATOR(:=)
  │                   └── <expression>
  │                       └── <simple-expression>
```

```

        └── <term>
            └── <factor>
                └── NUMBER(10)
    └── SEMICOLON(;)
    └── <assignment-statement>
        ├── IDENTIFIER(b)
        ├── ASSIGN_OPERATOR(:=)
        └── <expression>
            └── <simple-expression>
                └── <term>
                    └── <factor>
                        └── NUMBER(5)
    └── SEMICOLON(;)
    └── <assignment-statement>
        ├── IDENTIFIER(c)
        ├── ASSIGN_OPERATOR(:=)
        └── <expression>
            └── <simple-expression>
                └── <term>
                    └── <factor>
                        └── NUMBER(3)
    └── SEMICOLON(;)
    └── <assignment-statement>
        ├── IDENTIFIER(x)
        ├── ASSIGN_OPERATOR(:=)
        └── <expression>
            └── <simple-expression>
                └── <term>
                    └── <factor>
                        └── LPARENTHESIS(())
                        └── <expression>
                            └── <simple-expression>
                                └── <term>
                                    └── <factor>
                                        └── IDENTIFIER(a)
                                └── <additive-operator>
                                    └── ARITHMETIC_OPERATOR(+)
                                └── <term>
                                    └── <factor>
                                        └── IDENTIFIER(b)
                            └── RPARENTHESIS(())
                            └── <multiplicative-operator>
                                └── ARITHMETIC_OPERATOR(*)
                └── <factor>

```

```

    └── IDENTIFIER(c)
    └── <multiplicative-operator>
        └── ARITHMETIC_OPERATOR(/)
    └── <factor>
        └── NUMBER(2.0)

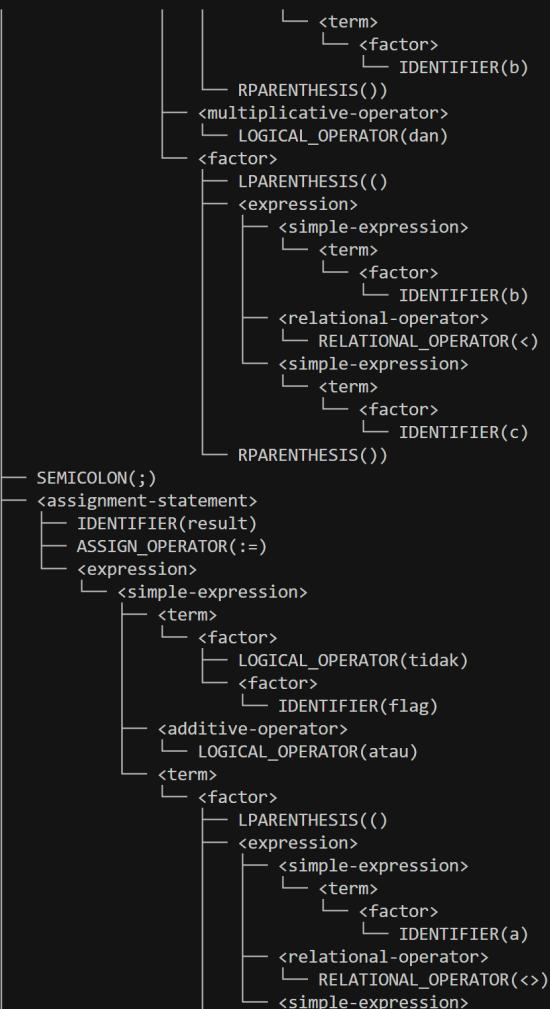
-- SEMICOLON();

-- <assignment-statement>
    └── IDENTIFIER(y)
    └── ASSIGN_OPERATOR(:=)
    └── <expression>
        └── <simple-expression>
            └── <term>
                └── <factor>
                    └── IDENTIFIER(a)
                └── <multiplicative-operator>
                    └── ARITHMETIC_OPERATOR(bagi)
                └── <factor>
                    └── IDENTIFIER(b)
            └── <additive-operator>
                └── ARITHMETIC_OPERATOR(+)
            └── <term>
                └── <factor>
                    └── IDENTIFIER(c)
                └── <multiplicative-operator>
                    └── ARITHMETIC_OPERATOR(mod)
                └── <factor>
                    └── NUMBER(2)

-- SEMICOLON();

-- <assignment-statement>
    └── IDENTIFIER(flag)
    └── ASSIGN_OPERATOR(:=)
    └── <expression>
        └── <simple-expression>
            └── <term>
                └── <factor>
                    └── LPARENTHESIS(())
                └── <expression>
                    └── <simple-expression>
                        └── <term>
                            └── <factor>
                                └── IDENTIFIER(a)
                        └── <relational-operator>
                            └── RELATIONAL_OPERATOR(>)
                    └── <simple-expression>

```







```
● PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src/parser_main.py test\milestone-2\input\test6.pas --check --output
Parse tree saved to: test\milestone-2\output\test6.txt
[PASS] Parse tree matches expected!
```

Sukses menguji kemampuan parser dalam menangani bentuk ekspresi yang complex, hasil uji sudah benar.

Tabel VII. MinimalProgramTest

Input
program MinimalTest; mulai selesai.
Output
<program> <program-header> KEYWORD (program) IDENTIFIER (MinimalTest) SEMICOLON (;

```

└── <compound-statement>
    ├── KEYWORD (mulai)
    └── KEYWORD (selesai)
    └── DOT (..)

```

Bukti dan Keterangan

```

● (pia-tubes-if2224) PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src
/parser_main.py test/milestone-2/input/test7.pas
<program>
└── <program-header>
    ├── KEYWORD(program)
    ├── IDENTIFIER(MinimalTest)
    └── SEMICOLON(;)
└── <compound-statement>
    ├── KEYWORD(mulai)
    └── KEYWORD(selesai)
    └── DOT(..)

● PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src/parser_main.py tes
t/milestone-2/input/test7.pas --check --output
Parse tree saved to: test\milestone-2\output\test7.txt
[PASS] Parse tree matches expected!

```

Sukses menguji kemampuan parser dalam menangani bentuk minimal program utama yang tidak berisi apapun, hasil uji sudah benar.

Tabel VIII. EmptySubprogramTest

Input
<pre> program EmptyDowntoTest; variabel i: integer; prosedur EmptyProc; mulai selesai; fungsi EmptyFunc: integer; mulai EmptyFunc := 0 selesai; mulai untuk i := 10 turun-ke 1 lakukan EmptyProc(); i := EmptyFunc() selesai. </pre>
Output
<pre> <program> └── <program-header> ├── KEYWORD(program) ├── IDENTIFIER(EmptyDowntoTest) └── SEMICOLON(;) └── <declaration-part> </pre>



```
    └─ <factor>
        └─ NUMBER(1)

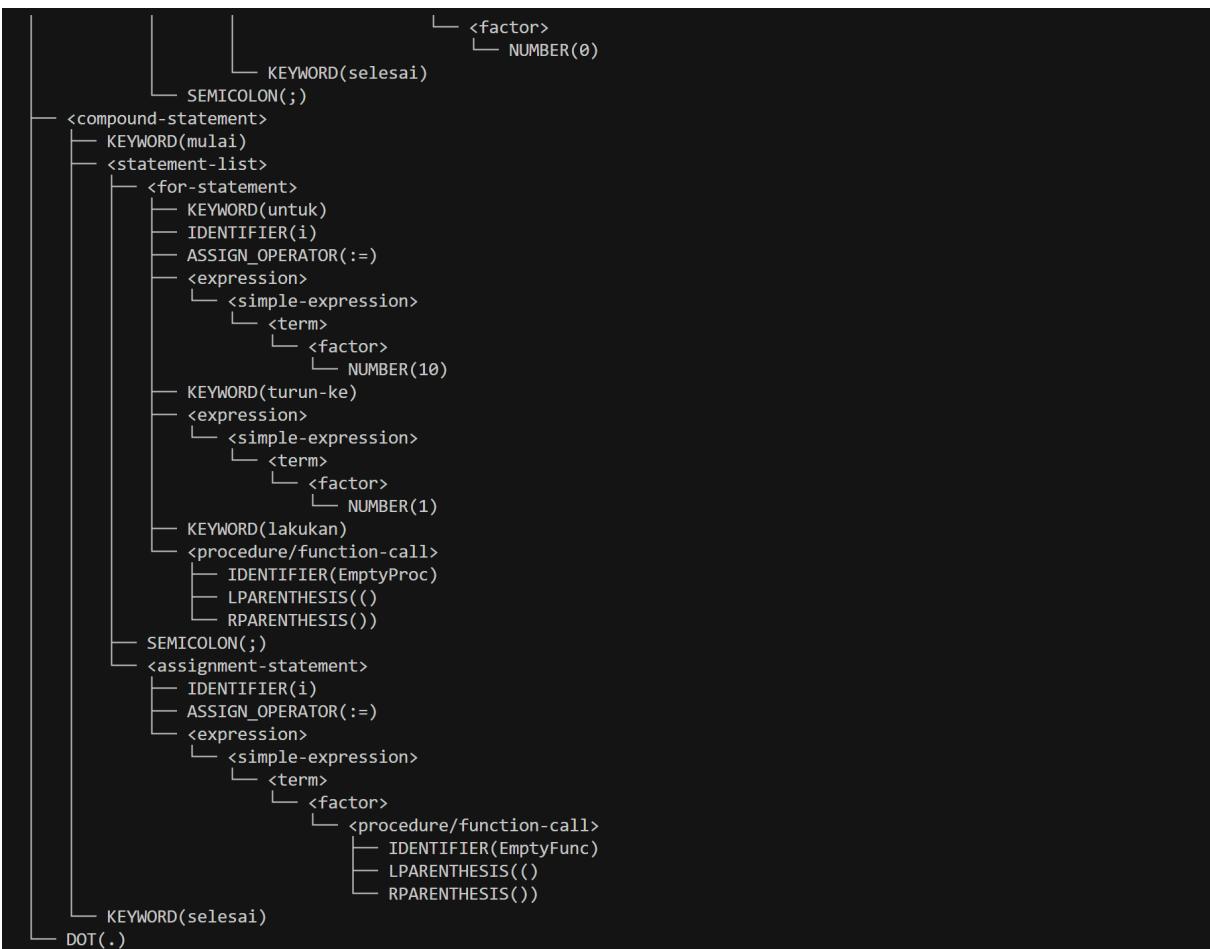
    └─ KEYWORD(lakukan)
    └─ <procedure/function-call>
        └─ IDENTIFIER(EmptyProc)
        └─ LPARENTHESIS(())
        └─ RPARENTHESIS(())

    └─ SEMICOLON(;)
    └─ <assignment-statement>
        └─ IDENTIFIER(i)
        └─ ASSIGN_OPERATOR(:=)
        └─ <expression>
            └─ <simple-expression>
                └─ <term>
                    └─ <factor>
                        └─ <procedure/function-call>
                            └─ IDENTIFIER(EmptyFunc)
                            └─ LPARENTHESIS(())
                            └─ RPARENTHESIS(())

    └─ KEYWORD(selesai)
    └─ DOT(.)
```

Bukti dan Keterangan

```
● (pia-tubes-if2224) PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-6) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src /parser_main.py test/milestone-2/input/test8.pas
<program>
  └── <program-header>
    ├── KEYWORD(program)
    ├── IDENTIFIER(EmptyDowntoTest)
    └── SEMICOLON(;)
  └── <declaration-part>
    └── <var-declaration>
      ├── KEYWORD(variabel)
      ├── <identifier-list>
      │   └── IDENTIFIER(i)
      ├── COLON(:)
      ├── <type>
      │   └── KEYWORD(integer)
      └── SEMICOLON(;)
    └── <subprogram-declaration>
      └── <procedure-declaration>
        ├── KEYWORD(prosedur)
        ├── IDENTIFIER(EmptyProc)
        └── SEMICOLON(;)
      └── <block>
        └── <compound-statement>
          ├── KEYWORD(mulai)
          └── KEYWORD(selesai)
        └── SEMICOLON(;)
    └── <subprogram-declaration>
      └── <function-declaration>
        ├── KEYWORD(fungsi)
        ├── IDENTIFIER(EmptyFunc)
        ├── COLON(:)
        ├── <type>
        │   └── KEYWORD(integer)
        └── SEMICOLON(;)
      └── <block>
        └── <compound-statement>
          ├── KEYWORD(mulai)
          ├── <statement-list>
          │   └── <assignment-statement>
          │       ├── IDENTIFIER(EmptyFunc)
          │       └── ASSIGN_OPERATOR(:=)
          └── <expression>
            └── <simple-expression>
              └── <term>
```



```
● PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src/parser_main.py test/milestone-2/input/test8.pas --check --output
Parse tree saved to: test\milestone-2\output\test8.txt
[PASS] Parse tree matches expected!
```

Sukses menguji kemampuan parser dalam menangani bentuk subprogram yang tidak berisi apapun, hasil uji sudah benar.

Tabel IX. NestedCompoundTest

Input
<pre>program NestedCompoundTest; variabel x, y: integer; done: boolean; mulai x := 0; done := false; selama tidak done lakukan mulai x := x + 1; y := x * 2;</pre>

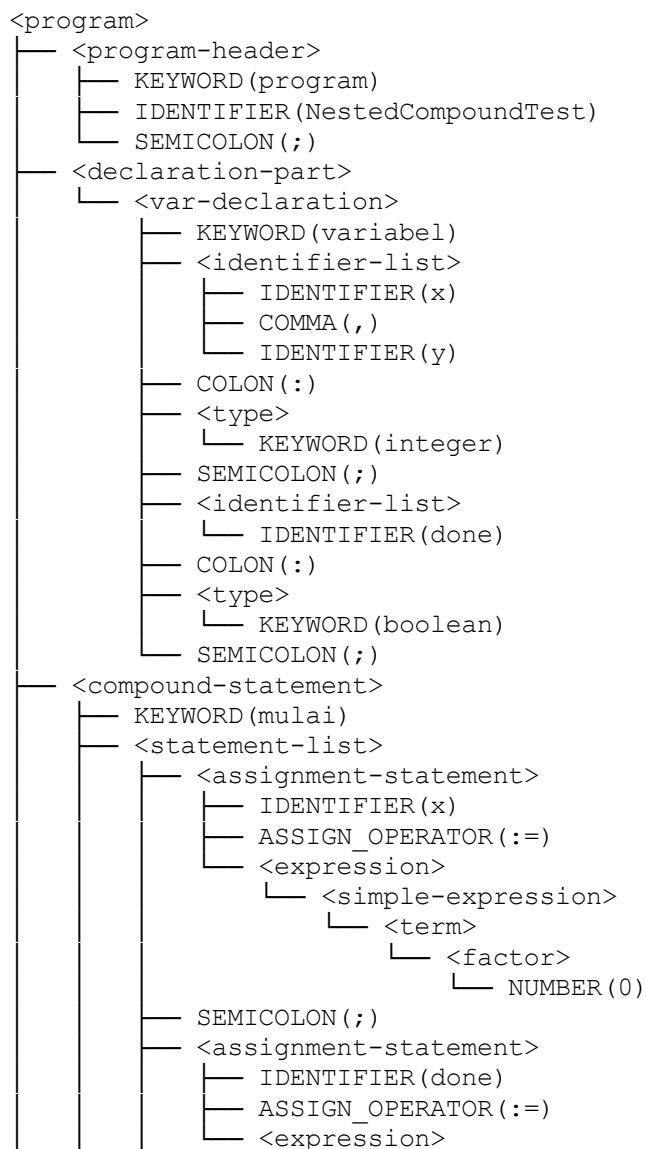
```

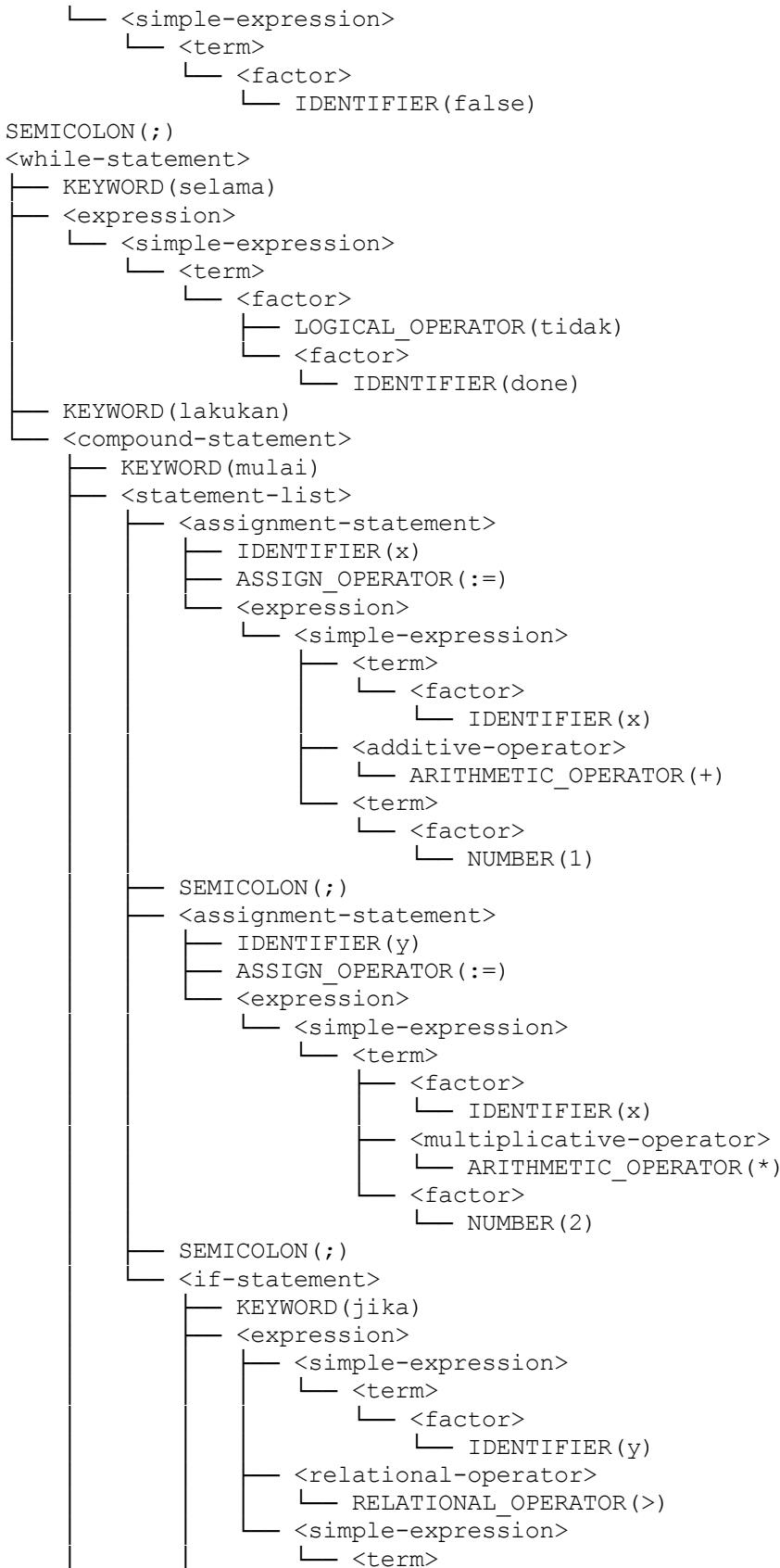
jika y > 5 maka
    mulai
        writeln(y);
        done := true
    selesai
selain-itu
    done := false;
selesai;

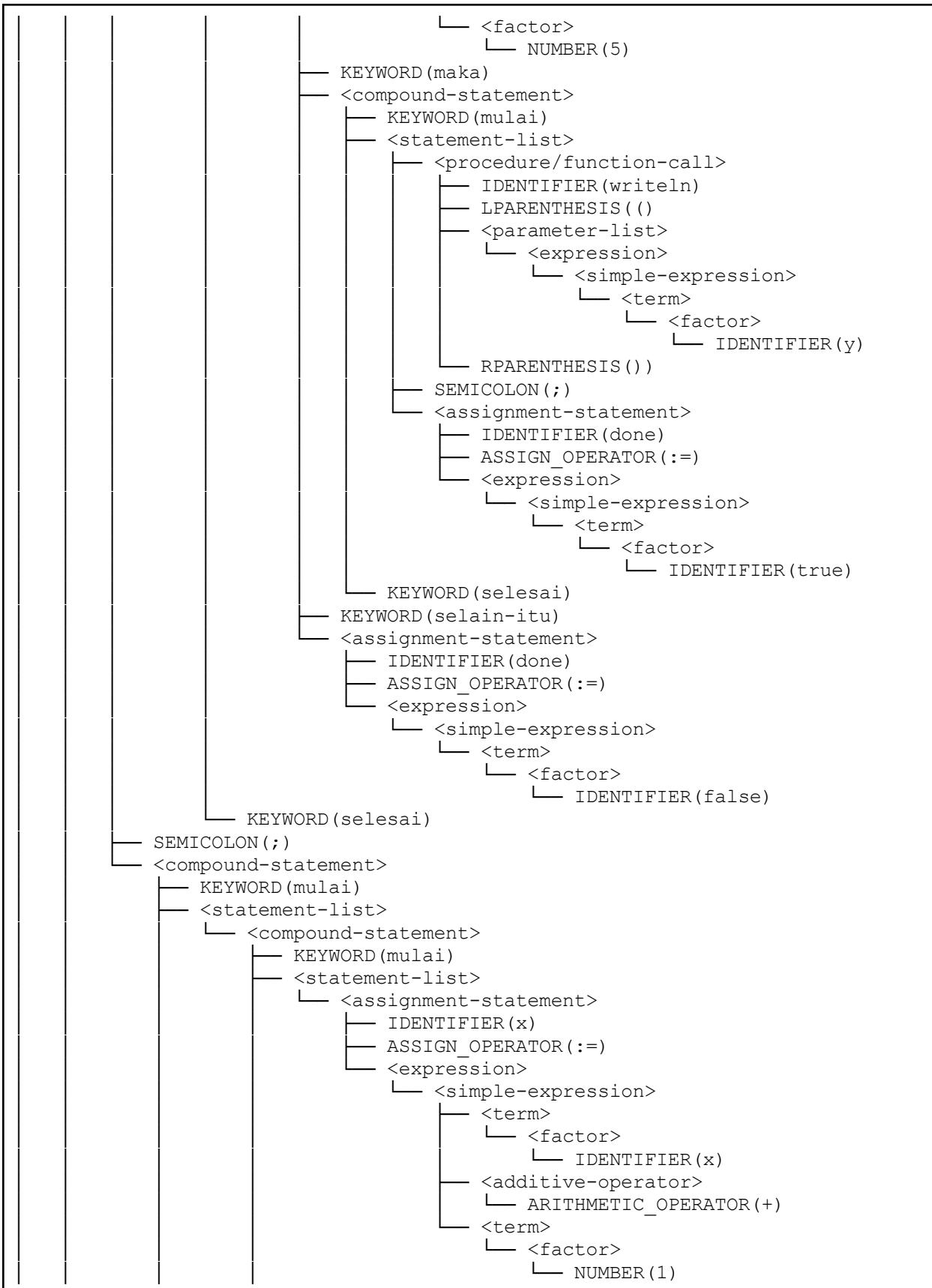
mulai
    mulai
        x := x + 1
    selesai
selesai
selesai.

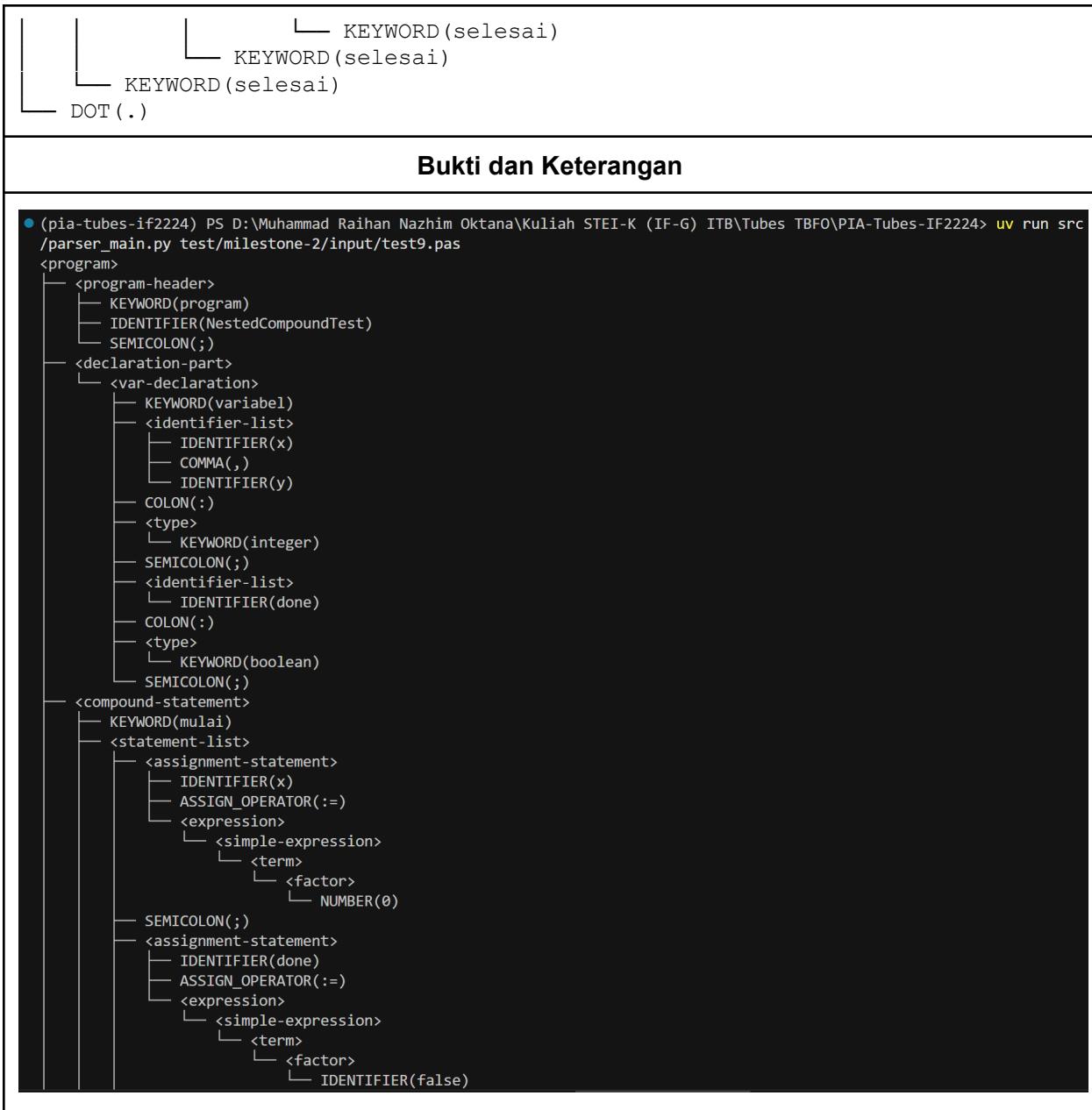
```

Output









```
SEMICOLON();  
| <while-statement>  
| | KEYWORD(selama)  
| | <expression>  
| | | <simple-expression>  
| | | | <term>  
| | | | | <factor>  
| | | | | | LOGICAL_OPERATOR(tidak)  
| | | | | <factor>  
| | | | | | IDENTIFIER(done)  
| | KEYWORD(lakukan)  
| | <compound-statement>  
| | | KEYWORD(mulai)  
| | | <statement-list>  
| | | | <assignment-statement>  
| | | | | IDENTIFIER(x)  
| | | | | ASSIGN_OPERATOR(:=)  
| | | | <expression>  
| | | | | <simple-expression>  
| | | | | | <term>  
| | | | | | | <factor>  
| | | | | | | | IDENTIFIER(x)  
| | | | | | <additive-operator>  
| | | | | | | ARITHMETIC_OPERATOR(+)  
| | | | | <term>  
| | | | | | <factor>  
| | | | | | | NUMBER(1)  
| | SEMICOLON();  
| | <assignment-statement>  
| | | IDENTIFIER(y)  
| | | ASSIGN_OPERATOR(:=)  
| | | <expression>  
| | | | <simple-expression>  
| | | | | <term>  
| | | | | | <factor>  
| | | | | | | IDENTIFIER(x)  
| | | | | | <multiplicative-operator>  
| | | | | | | ARITHMETIC_OPERATOR(*)  
| | | | | <factor>  
| | | | | | | NUMBER(2)  
| | SEMICOLON();  
| | <if-statement>  
| | | KEYWORD(jika)  
| | | <expression>
```



```
● PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src/parser_main.py tes t/milestone-2/input/test9.pas --check --output
Parse tree saved to: test\milestone-2\output\test9.txt
[PASS] Parse tree matches expected!
```

Sukses menguji kemampuan parser dalam menangani bentuk nested compound yang berarti terdapat mulai-selesai di dalam program utama / di dalam sub itu sendiri, hasil uji sudah benar.

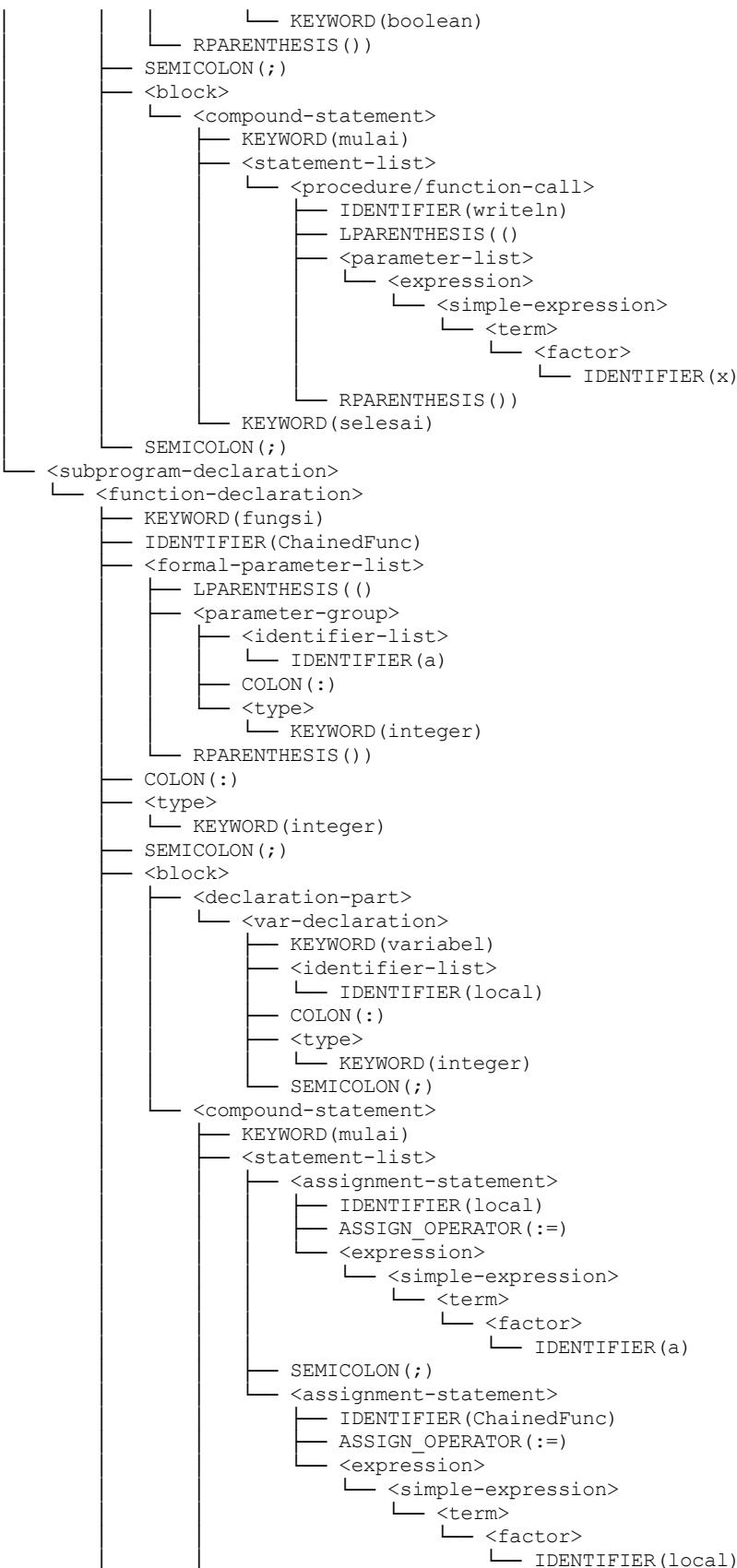
Tabel X. ComplexDeclarationsTest

Input
<pre>program ComplexDeclarationsTest; konstanta A = 1; B = 2; C = 3; tipe Range1 = 1..100; Range2 = 'a'..'z'; Vector = larik[1..5] dari integer; variabel r1: Range1; r2: Range2; v: Vector; prosedur MultiParam(x: integer; y: real; z: char; flag: boolean); mulai writeln(x) selesai; fungsi ChainedFunc(a: integer): integer; variabel local: integer; mulai local := a; ChainedFunc := local selesai; mulai r1 := 50; r2 := 'm'; v[1] := 10; MultiParam(100, 3.14, 'X', true); r1 := ChainedFunc(ChainedFunc(5)); jika tidak ((r1 = 5) atau (r1 <> 10)) dan (r2 >= 'a') maka mulai selesai selain-itu mulai selesai selesai.</pre>

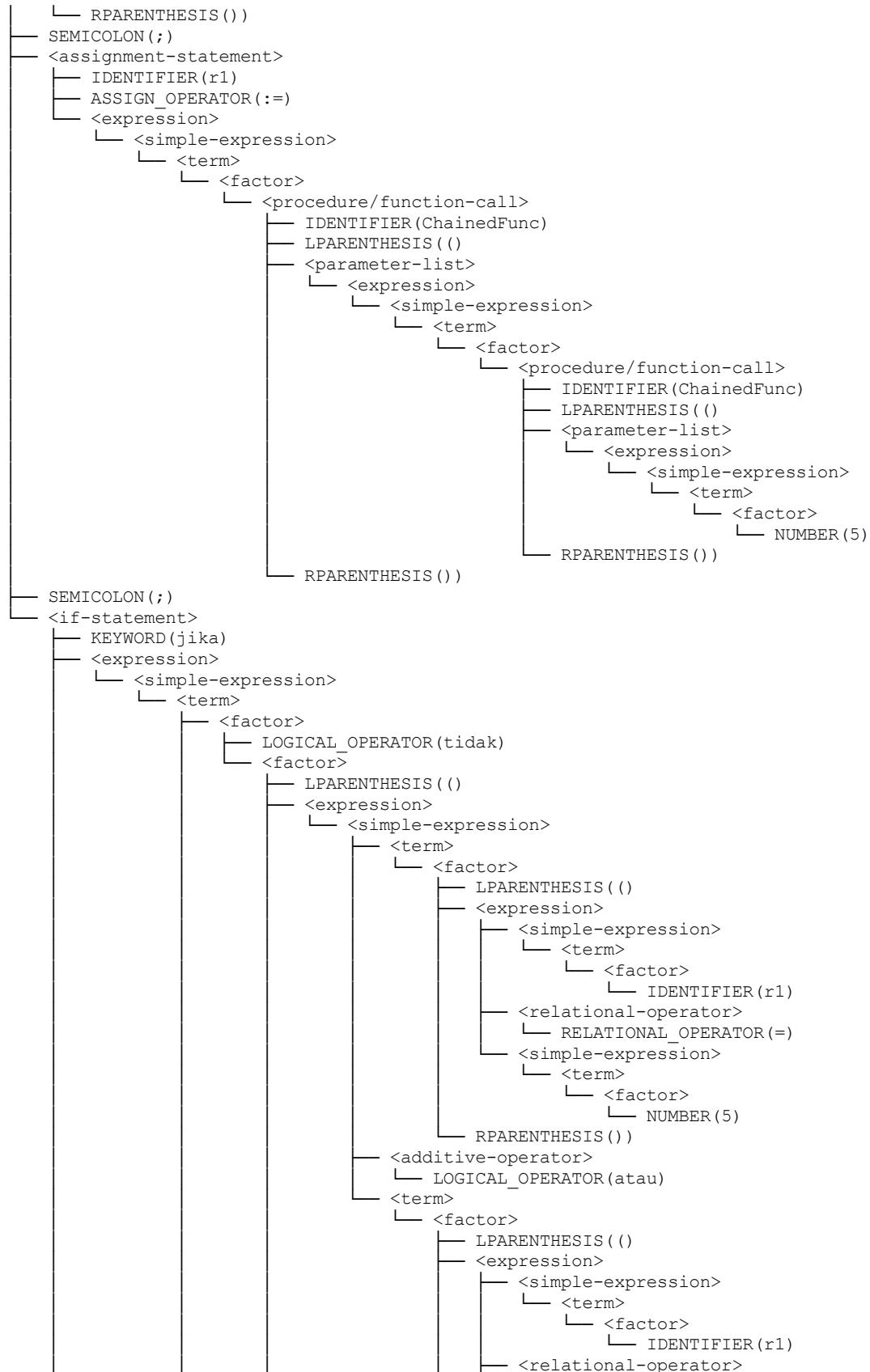
Output

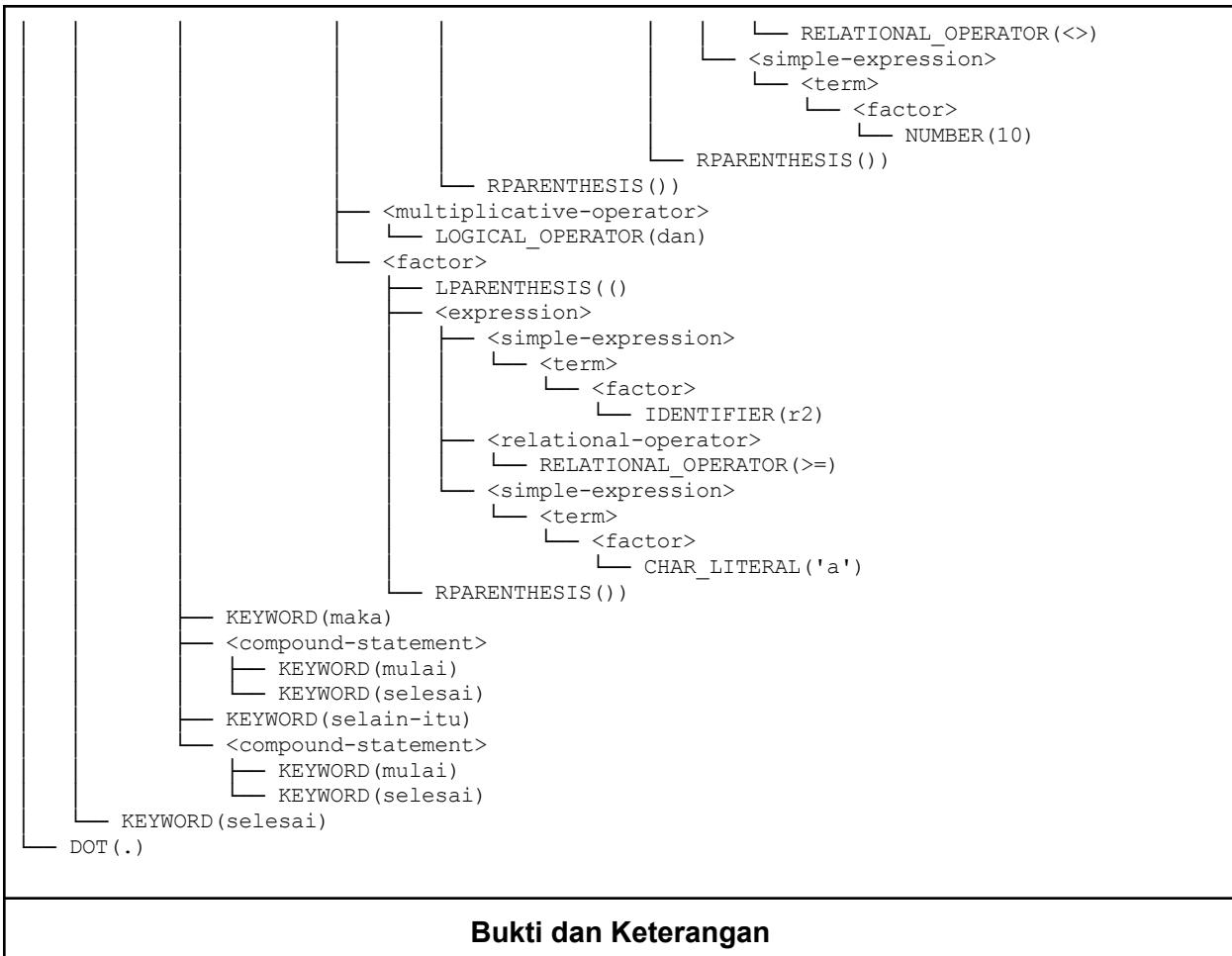
```
<program>
  <program-header>
    KEYWORD(program)
    IDENTIFIER(ComplexDeclarationsTest)
    SEMICOLON(;)
  <declaration-part>
    <const-declaration>
      KEYWORD(konstanta)
      IDENTIFIER(A)
      ASSIGN_OPERATOR(=)
      NUMBER(1)
      SEMICOLON(;)
      IDENTIFIER(B)
      ASSIGN_OPERATOR(=)
      NUMBER(2)
      SEMICOLON(;)
      IDENTIFIER(C)
      ASSIGN_OPERATOR(=)
      NUMBER(3)
      SEMICOLON(;)
    <type-declaration>
      KEYWORD(tipe)
      IDENTIFIER(Range1)
      ASSIGN_OPERATOR(=)
      <type>
        <range>
          <expression>
            <simple-expression>
              <term>
                <factor>
                  NUMBER(1)
            RANGE_OPERATOR(..)
          <expression>
            <simple-expression>
              <term>
                <factor>
                  NUMBER(100)
        SEMICOLON(;)
        IDENTIFIER(Range2)
        ASSIGN_OPERATOR(=)
        <type>
          <range>
            <expression>
              <simple-expression>
                <term>
                  <factor>
                    CHAR_LITERAL('a')
            RANGE_OPERATOR(..)
          <expression>
            <simple-expression>
              <term>
                <factor>
                  CHAR_LITERAL('z')
        SEMICOLON(;)
        IDENTIFIER(Vector)
        ASSIGN_OPERATOR(=)
        <type>
          <array-type>
            KEYWORD(larik)
            LBRACKET([)
            <range>
              <expression>
                <simple-expression>
```











- PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src/parser_main.py test/milestone-2/input/test10.pas

```
<program>
  <program-header>
    KEYWORD(program)
    IDENTIFIER(ComplexDeclarationsTest)
    SEMICOLON(;)
  <declaration-part>
    <const-declaration>
      KEYWORD(konstanta)
      IDENTIFIER(A)
      ASSIGN_OPERATOR(=)
      NUMBER(1)
      SEMICOLON(;)
      IDENTIFIER(B)
      ASSIGN_OPERATOR(=)
      NUMBER(2)
      SEMICOLON(;)
      IDENTIFIER(C)
      ASSIGN_OPERATOR(=)
      NUMBER(3)
      SEMICOLON(;)
    <type-declaration>
      KEYWORD(tipe)
      IDENTIFIER(Range1)
      ASSIGN_OPERATOR(=)
      <type>
        <range>
          <expression>
            <simple-expression>
              <term>
                <factor>
                  NUMBER(1)
            RANGE_OPERATOR(..)
            <expression>
              <simple-expression>
                <term>
                  <factor>
                    NUMBER(100)
        SEMICOLON(;)
        IDENTIFIER(Range2)
        ASSIGN_OPERATOR(=)
        <type>
          <range>
```

```

    └── <expression>
        └── <simple-expression>
            └── <term>
                └── <factor>
                    └── CHAR_LITERAL('a')
    └── RANGE_OPERATOR(..)
    └── <expression>
        └── <simple-expression>
            └── <term>
                └── <factor>
                    └── CHAR_LITERAL('z')
    └── SEMICOLON(;)
    └── IDENTIFIER(Vector)
    └── ASSIGN_OPERATOR(=)
    └── <type>
        └── <array-type>
            └── KEYWORD(larik)
            └── LBRACKET([)
            └── <range>
                └── <expression>
                    └── <simple-expression>
                        └── <term>
                            └── <factor>
                                └── NUMBER(1)
            └── RANGE_OPERATOR(..)
            └── <expression>
                └── <simple-expression>
                    └── <term>
                        └── <factor>
                            └── NUMBER(5)
            └── RBRACKET(])
            └── KEYWORD(dari)
            └── <type>
                └── KEYWORD(integer)
    └── SEMICOLON(;)
    └── <var-declaration>
        └── KEYWORD(variabel)
        └── <identifier-list>
            └── IDENTIFIER(r1)
    └── COLON(:)
    └── <type>
        └── IDENTIFIER(Range1)
    └── SEMICOLON(;)
    └── <identifier-list>

```

```
    └─ IDENTIFIER(r2)
    └─ COLON(:)
    └─ <type>
        └─ IDENTIFIER(Range2)
    └─ SEMICOLON(;)
    └─ <identifier-list>
        └─ IDENTIFIER(v)
    └─ COLON(:)
    └─ <type>
        └─ IDENTIFIER(Vector)
    └─ SEMICOLON(;)
└─ <subprogram-declaration>
    └─ <procedure-declaration>
        └─ KEYWORD(prosedur)
        └─ IDENTIFIER(MultiParam)
        └─ <formal-parameter-list>
            └─ LPARENTHESIS(())
            └─ <parameter-group>
                └─ <identifier-list>
                    └─ IDENTIFIER(x)
                └─ COLON(:)
                └─ <type>
                    └─ KEYWORD(integer)
            └─ SEMICOLON(;)
            └─ <parameter-group>
                └─ <identifier-list>
                    └─ IDENTIFIER(y)
                └─ COLON(:)
                └─ <type>
                    └─ KEYWORD(real)
            └─ SEMICOLON(;)
            └─ <parameter-group>
                └─ <identifier-list>
                    └─ IDENTIFIER(z)
                └─ COLON(:)
                └─ <type>
                    └─ KEYWORD(char)
            └─ SEMICOLON(;)
            └─ <parameter-group>
                └─ <identifier-list>
                    └─ IDENTIFIER(flag)
                └─ COLON(:)
                └─ <type>
                    └─ KEYWORD(boolean)
```

```

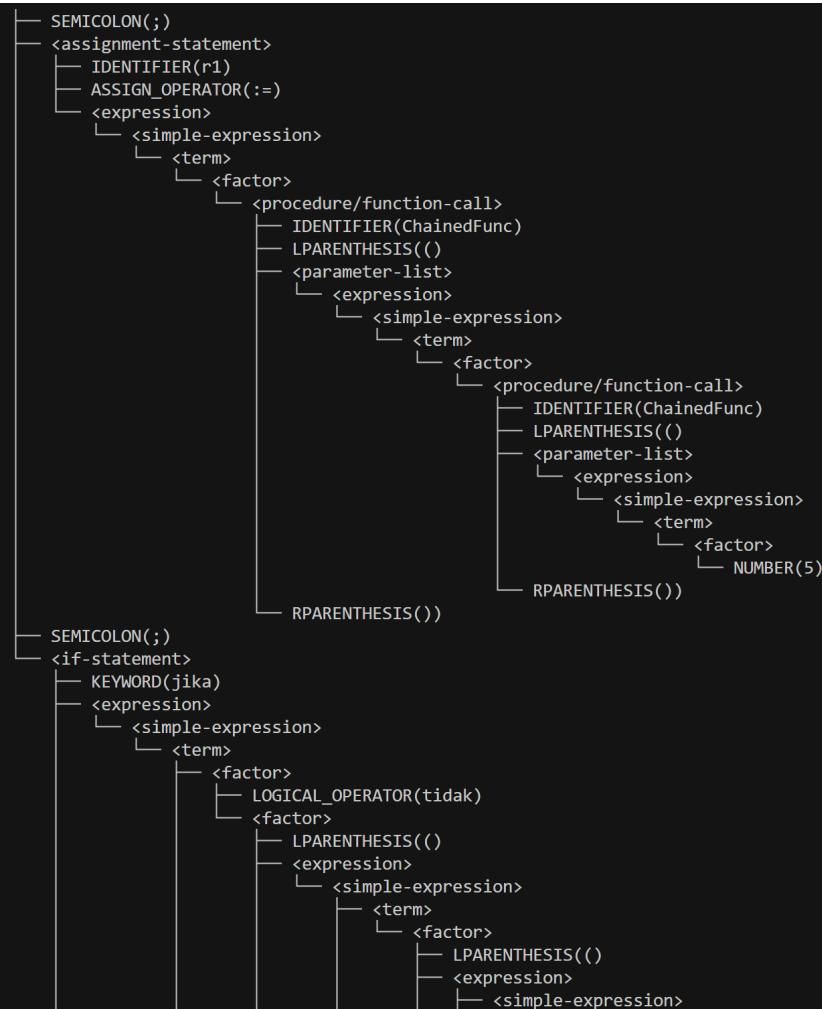
    └── RPARENTHESIS())
    └── SEMICOLON(;)
  └── <block>
    └── <compound-statement>
      ├── KEYWORD(mulai)
      └── <statement-list>
        └── <procedure/function-call>
          ├── IDENTIFIER(writeln)
          ├── LPARENTHESIS(())
          └── <parameter-list>
            └── <expression>
              └── <simple-expression>
                └── <term>
                  └── <factor>
                    └── IDENTIFIER(x)
    └── RPARENTHESIS())
    └── KEYWORD(selesai)
  └── SEMICOLON(;)
<subprogram-declaration>
└── <function-declaration>
  ├── KEYWORD(fungsi)
  ├── IDENTIFIER(ChainedFunc)
  └── <formal-parameter-list>
    ├── LPARENTHESIS(())
    └── <parameter-group>
      └── <identifier-list>
        └── IDENTIFIER(a)
    └── COLON(:)
    └── <type>
      └── KEYWORD(integer)
  └── RPARENTHESIS())
  └── COLON(:)
  └── <type>
    └── KEYWORD(integer)
  └── SEMICOLON(;)
  └── <block>
    └── <declaration-part>
      └── <var-declaration>
        ├── KEYWORD(variabel)
        └── <identifier-list>
          └── IDENTIFIER(local)
    └── COLON(:)
    └── <type>
      └── KEYWORD(integer)

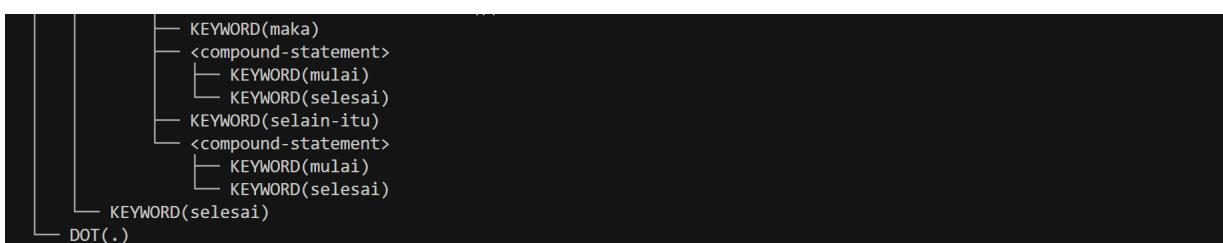
```

```
    └─ SEMICOLON(;)
   └─ <compound-statement>
      └─ KEYWORD(mulai)
   └─ <statement-list>
      └─ <assignment-statement>
         └─ IDENTIFIER(local)
         └─ ASSIGN_OPERATOR(:=)
         └─ <expression>
            └─ <simple-expression>
               └─ <term>
                  └─ <factor>
                     └─ IDENTIFIER(a)
      └─ SEMICOLON(;)
      └─ <assignment-statement>
         └─ IDENTIFIER(ChainedFunc)
         └─ ASSIGN_OPERATOR(:=)
         └─ <expression>
            └─ <simple-expression>
               └─ <term>
                  └─ <factor>
                     └─ IDENTIFIER(local)
   └─ KEYWORD(selesai)
   └─ SEMICOLON(;)

   └─ <compound-statement>
      └─ KEYWORD(mulai)
   └─ <statement-list>
      └─ <assignment-statement>
         └─ IDENTIFIER(r1)
         └─ ASSIGN_OPERATOR(:=)
         └─ <expression>
            └─ <simple-expression>
               └─ <term>
                  └─ <factor>
                     └─ NUMBER(50)
      └─ SEMICOLON(;)
      └─ <assignment-statement>
         └─ IDENTIFIER(r2)
         └─ ASSIGN_OPERATOR(:=)
         └─ <expression>
            └─ <simple-expression>
               └─ <term>
                  └─ <factor>
                     └─ CHAR_LITERAL('m')
   └─ SEMICOLON();
```

```
-- <assignment-statement>
  -- IDENTIFIER(v)
  -- LBRAKET([)
  -- <expression>
    -- <simple-expression>
      -- <term>
        -- <factor>
          -- NUMBER(1)
  -- RBRACKET()])
  -- ASSIGN_OPERATOR(:=)
  -- <expression>
    -- <simple-expression>
      -- <term>
        -- <factor>
          -- NUMBER(10)
-- SEMICOLON(;)
-- <procedure/function-call>
  -- IDENTIFIER(MultiParam)
  -- LPARENTHESIS(())
  -- <parameter-list>
    -- <expression>
      -- <simple-expression>
        -- <term>
          -- <factor>
            -- NUMBER(100)
    -- COMMA(,)
    -- <expression>
      -- <simple-expression>
        -- <term>
          -- <factor>
            -- NUMBER(3.14)
    -- COMMA(,)
    -- <expression>
      -- <simple-expression>
        -- <term>
          -- <factor>
            -- CHAR_LITERAL('X')
    -- COMMA(,)
    -- <expression>
      -- <simple-expression>
        -- <term>
          -- <factor>
            -- IDENTIFIER(true)
-- RPARENTHESIS()
```





```

● PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src/parser_main.py test/milestone-2/input/test10.pas --check --output
Parse tree saved to: test\milestone-2\output\test10.txt
[PASS] Parse tree matches expected!
  
```

Sukses menguji kemampuan parser dalam menangani bentuk complex declarations, hasil uji sudah benar.

Tabel XI. MissingSelesaiTest

Input
program MissingSelesai; mulai

<pre>x := 5; .</pre>
Output
SyntaxError at position 42: Expected KEYWORD 'selesai', got DOT '..'
Bukti dan Keterangan
<pre>(pia-tubes-if2224) PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src ❸ /parser_main.py test/milestone-2/input/test11.pas SyntaxError at position 42: Expected KEYWORD 'selesai', got DOT '..'</pre>
Sukses menguji kemampuan parser dalam menangani bentuk kesalahan syntax berupa tidak adanya keyword "selesai" di akhir program, hasil uji sudah benar.

Tabel XII. WrongAssignTest

<pre>program WrongAssign; variabel x: integer; mulai x = 5; selesai.</pre>
Output
SyntaxError at position 58: Expected LPARENTHESIS, got RELATIONAL_OPERATOR '='
Bukti dan Keterangan
<pre>❸ (pia-tubes-if2224) PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src /parsers_main.py test/milestone-2/input/test12.pas SyntaxError at position 58: Expected LPARENTHESIS, got RELATIONAL_OPERATOR '='</pre>
Sukses menguji kemampuan parser dalam menangani bentuk kesalahan syntax berupa kesalahan penulisan operator assignment pada program, hasil uji sudah benar.

Tabel XIII. MissingParenthesisTest

<pre>program MissingParen; variabel x: integer; mulai writeln('Hello'); selesai.</pre>
Output
SyntaxError at position 72: Expected RPARENTHESIS, got SEMICOLON ';'

Bukti dan Keterangan

```
① (pia-tubes-if2224) PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src  
/parser_main.py test/milestone-2/input/test13.pas  
SyntaxError at position 72: Expected RPARENTHESIS, got SEMICOLON ';'
```

Sukses menguji kemampuan parser dalam menangani bentuk kesalahan syntax berupa kesalahan ketiadaan kurung kanan / kurung tutup ")", hasil uji sudah benar.

Tabel XIV. InvalidArrayTest

Input
program InvalidArray; variabel arr: larik[1..10] integer; mulai selesai.
Output
SyntaxError at position 53: Expected KEYWORD 'dari', got KEYWORD 'integer'
Bukti dan Keterangan
<pre>① (pia-tubes-if2224) PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src /parser_main.py test/milestone-2/input/test14.pas SyntaxError at position 53: Expected KEYWORD 'dari', got KEYWORD 'integer'</pre>
Sukses menguji kemampuan parser dalam menangani bentuk kesalahan syntax berupa kesalahan ketiadaan keyword "dari" sebelum tipe data array disebutkan, hasil uji sudah benar.

Tabel XV. MissingDotTest

Input
program MissingDot; mulai x := 5; selesai
Output
SyntaxError: Unexpected end of file. Expected DOT
Bukti dan Keterangan
<pre>① (pia-tubes-if2224) PS D:\Muhammad Raihan Nazhim Oktana\Kuliah STEI-K (IF-G) ITB\Tubes TBFO\PIA-Tubes-IF2224> uv run src /parser_main.py test/milestone-2/input/test15.pas SyntaxError: Unexpected end of file. Expected DOT</pre>
Sukses menguji kemampuan parser dalam menangani bentuk kesalahan syntax berupa kesalahan ketiadaan tanda titik / dot ".i" pada akhir program, hasil uji sudah benar.

BAB V: Kesimpulan dan Saran

5.1 Kesimpulan

Tugas Besar Teori Bahasa Formal dan Otomata (IF2224-24) Milestone 2 ini berhasil mengimplementasikan tahap kedua dari proses kompilasi, yaitu pembentukan Abstract Syntax Tree (AST) beserta penyesuaian Lexer agar sesuai dengan spesifikasi bahasa Pascal-S yang telah dimodifikasi ke dalam bahasa Indonesia. Pada tahap ini, sistem tidak hanya mampu mengenali token-token hasil analisis leksikal, tetapi juga mengorganisasikannya menjadi struktur pohon sintaks abstrak yang merepresentasikan hubungan hierarkis antar elemen bahasa secara formal dan semantik.

Implementasi AST berbasis kelas ASTNode memungkinkan setiap konstruksi sintaks, seperti deklarasi, ekspresi, dan pernyataan, direpresentasikan secara modular dan extensible. Penerapan Visitor Pattern memastikan proses traversal dan analisis semantik dapat dilakukan tanpa perlu mengubah struktur internal node. Selain itu, dilakukan modifikasi post-processing pada Lexer melalui fungsi `merge_hyphenated_keywords()` untuk menangani kata kunci majemuk berbahasa Indonesia sehingga menjaga konsistensi antara hasil analisis leksikal dan sintaktis.

Dengan demikian, milestone ini menandai pencapaian penting dalam rantai pembangunan kompilator Pascal-S, di mana sistem kini mampu menghasilkan representasi sintaks abstrak yang menjadi fondasi bagi tahap selanjutnya seperti semantic analysis dan code generation. Desain yang modular dan berbasis objek juga memastikan bahwa sistem mudah diperluas serta tetap selaras dengan prinsip formal dalam teori bahasa dan otomata.

5.2 Saran

Untuk pengembangan selanjutnya, disarankan agar dilakukan integrasi AST dengan parser dan semantic analyzer guna memvalidasi konsistensi struktur dan tipe data. Selain itu, perlu dilakukan pengujian tambahan terhadap kasus kompleks seperti blok bersarang dan ekspresi majemuk untuk memastikan keandalan sistem. Penambahan fitur visualisasi AST dan pelaporan kesalahan yang lebih informatif juga direkomendasikan agar proses debugging dan analisis menjadi lebih mudah dan efisien.

Lampiran

Pranala Repotori Github: <https://github.com/fathurwithyou/PIA-Tubes-IF2224>

Pranala Workspace Diagram:

https://drive.google.com/file/d/1czvTxZ2tvb7p6O_OSoOQtYlc81k2ghwc/view

Tabel Pembagian Tugas:

NIM	Nama Lengkap	Persentase Pembagian Tugas
13523021	Muhammad Raihan Nazhim Oktana	100%
13523057	Faqih Muhammad Syuhada	100%

13523097	Shanice Feodora Tjahjono	100%
13523105	Muhammad Fathur Rizky	100%

Daftar Pustaka

- [1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Boston, MA, USA: Pearson, 2007.
- [2] A. W. Appel, *Modern Compiler Implementation in Java*. Cambridge, U.K.: Cambridge Univ. Press, 1998.
- [3] D. Grune and C. J. H. Jacobs, *Parsing Techniques: A Practical Guide*, 2nd ed. New York, NY, USA: Springer-Verlag, 2008.
- [4] T. W. Parsons, *Introduction to Compiler Construction*. New York, NY, USA: Computer Science Press, 1992.
- [5] N. Wirth, *Algorithms + Data Structures = Programs*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1976.
- [6] D. E. Knuth, “Semantics of context-free languages,” *Mathematical Systems Theory*, vol. 2, pp. 127–145, 1968.