LAPORAN MILESTONE 2
IF2224 TEORI BAHASA FORMAL DAN OTOMATA
SYNTAX ANALYSIS

Kelompok 6 DFAntastic (DFC)

Mayla Yaffa Ludmilla 13523050
Anella Utari Gunadi 13523078
Muhammad Edo Raduputu Aprima 13523096
Athian Nugraha Muarajuang 13523106

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2025

DAFTAR ISI

DAFTAR ISI 1
1. DESKRIPSI TUGAS 2
2. LANDASAN TEORI 6
2.1. Syntax Analyzer 6
2.2. Parse Tree 6
2.3. Recursive Descent 7
3. PERANCANGAN & IMPLEMENTASI 8
3.1. Perancangan 8
3.1.1. Aturan Grammar 8
3.2. Implementasi 12
a. node.py 12
b. Fungsi dasar kelas parser 12
c. Fungsi parser grammar utama 14
d. Fungsi header program 15
e. Fungsi bagian deklarasi 15
f. Fungsi tipe data (type processing) 18
g. Fungsi Subprogram (procedure dan function) 21
h. Fungsi block program 23
i. Fungsi parse compound statement 24
j. Fungsi parse pemanggilan subprogram (procedure dan function) 27
k. Fungsi parse expression serta operator 29
4. Pengujian 35
4.1. Pengujian 0 35
4.2. Pengujian 1 35
4.3. Pengujian 2 37
4.4. Pengujian 3 41
4.5. Pengujian 4 47
4.6. Pengujian 5 47
5. KESIMPULAN DAN SARAN 52
5.1. Kesimpulan 52
5.2. Saran 52
LAMPIRAN 53
REFERENSI 57

1. DESKRIPSI TUGAS

Pada milestone ini, kami diminta untuk membuat tahapan kedua dari compiler, yaitu syntax analysis. Parser berfungsi melakukan analisis sintaksis (syntax analysis) dengan menggunakan Recursive Descent untuk mengenali struktur gramatikal dalam deretan token.

Tahapan ini bertujuan untuk menganalisis struktur sintaksis dari list of tokens yang dihasilkan lexer dan membangun Parse Tree yang merepresentasikan hierarki struktur program Pascal-S sesuai dengan grammar bahasa. Parse Tree ini akan menjadi masukan bagi tahap semantic analysis dalam proses kompilasi.

Masukan	List Token (misalnya VAR(X), ASSIGN(=), OPERATOR(+))
Keluaran	Parse Tree
Algoritma	Recursive Descent

Program harus menggunakan Recursive Descent dengan grammar yang di-hardcode dalam program dan keseluruhan grammar dijelaskan kembali dalam laporan yang dibuat. Input awal dari program tetap merupakan source code PASCAL-S sehingga sebelum dimasukkan ke parser, kode dimasukkan terlebih dahulu ke lexer.

Input Command (contoh yang digunakan menggunakan Python)
Format: python [Compiler] [Kode Pascal]

```
python compiler.py program.pas
```

Isi program.pas

```
program Hello;

variabel
  a, b: integer;

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

program.pas ketika diubah menjadi token

```
KEYWORD(program)
IDENTIFIER(Hello)
SEMICOLON(;)
KEYWORD(variabel)
IDENTIFIER(a)
COMMA(,)
```

```
IDENTIFIER(b)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(mulai)
IDENTIFIER(a)
ASSIGN_OPERATOR(:=)
NUMBER(5)
SEMICOLON(;)
IDENTIFIER(b)
ASSIGN_OPERATOR(:=)
IDENTIFIER(a)
ARITHMETIC_OPERATOR(+)
NUMBER(10)
SEMICOLON(;)
KEYWORD(writeln)
LPARENTHESIS(()
STRING_LITERAL('Result = ')
COMMA(,)
IDENTIFIER(b)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(selesai)
DOT(.)
```

Proses (grammar belum tentu benar)

```
program → program-header declaration-part compound-statement DOT
program-header → KEYWORD(program) IDENTIFIER SEMICOLON
declaration-part → ...
compound-statement → ...
... grammar lainnya

Berdasarkan grammar “program”, non-terminal yang pertama ditemukan adalah
“program-header”. Setelah itu di dalam “program-header” dilakukan pengecekan
token input.

KEYWORD(program)
IDENTIFIER(Hello)
SEMICOLON(;)

Karena token input sesuai, berarti “program-header” valid. Sehingga output
sementara akan seperti ini

<program>
 ├── <program-header>
 │      ├── KEYWORD(program)
 │      ├── IDENTIFIER(Hello)
 │      └── SEMICOLON(;)
... sisa output

Kemudian sisa pengecekan dilakukan untuk “declaration-part”,
“compound-statement”, dan DOT.

Kemudian juga disini dilakukan error checking di level parser. Contohnya
```

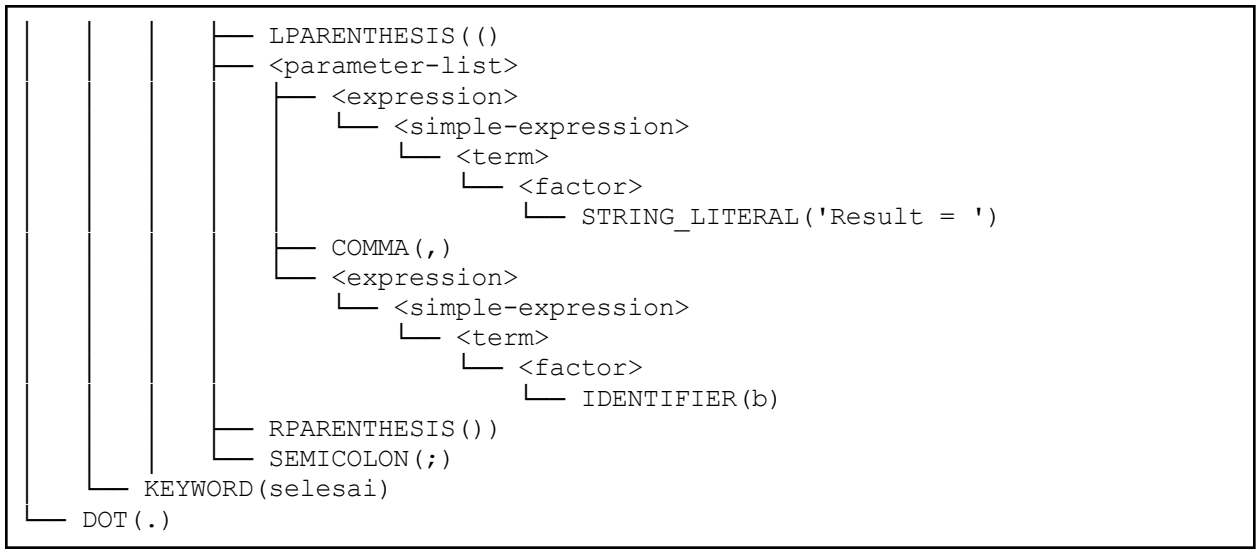
adalah misal untuk “program-header”, token yang dibaca adalah seperti berikut.

```
KEYWORD(program)
IDENTIFIER(Hello)
DOT(.)
```

Maka parser akan mengeluarkan error. Untuk pesan error dibebaskan yang penting informatif. Contohnya adalah “Syntax error: unexpected token DOT(.), expected SEMICOLON(;)”.

Keluaran (output)

```
<program>
├── <program-header>
│   ├── KEYWORD(program)
│   ├── IDENTIFIER(Hello)
│   └── SEMICOLON(;)
├── <declaration-part>
│   └── <var-declaration>
│       ├── KEYWORD(variabel)
│       ├── <identifier-list>
│       │   ├── IDENTIFIER(a)
│       │   ├── COMMA(,)
│       │   └── IDENTIFIER(b)
│       ├── COLON(:)
│       ├── <type>
│       │   └── KEYWORD(integer)
│       └── SEMICOLON(;)
├── <compound-statement>
│   ├── KEYWORD(mulai)
│   ├── <statement-list>
│   │   ├── <assignment-statement>
│   │   │   ├── IDENTIFIER(a)
│   │   │   ├── ASSIGN_OPERATOR(:=)
│   │   │   └── <expression>
│   │   │       └── <simple-expression>
│   │   │           └── <term>
│   │   │               └── <factor>
│   │   │                   └── NUMBER(5)
│   │   ├── SEMICOLON(;)
│   │   ├── <assignment-statement>
│   │   │   ├── IDENTIFIER(b)
│   │   │   ├── ASSIGN_OPERATOR(:=)
│   │   │   └── <expression>
│   │   │       ├── <simple-expression>
│   │   │       │   ├── <term>
│   │   │       │   │   └── <factor>
│   │   │       │   │       └── IDENTIFIER(a)
│   │   │       │   ├── ARITHMETIC_OPERATOR(+)
│   │   │       │   └── <term>
│   │   │       │       └── <factor>
│   │   │       │           └── NUMBER(10)
│   │   ├── SEMICOLON(;)
│   │   ├── <procedure-call>
│   │   │   ├── KEYWORD(writeln)
```


LPARENTHESIS(()
<parameter-list>
<expression>
<simple-expression>
<term>
<factor>
STRING_LITERAL('Result = ')
COMMA(,)
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(b)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(selesai)
DOT(.)

2. LANDASAN TEORI

2.1. Syntax Analyzer

Syntax Analyzer adalah tahap setelah Lexical Analysis dalam proses kompilasi, yang bertugas memeriksa apakah token-token yang dihasilkan oleh lexer tersusun sesuai dengan aturan grammar bahasa pemrograman. Sederetan token yang tidak mengikuti aturan sintaks akan dilaporkan sebagai kesalahan sintaks (syntax error). Secara logika deretan token yang bersesuaian dengan sintaks tertentu akan dinyatakan sebagai pohon parsing (parse tree).

Tahap-tahap dalam syntax analyzer mencakup:

1. Parsing, dimana syntax analyzer nganalisis token berdasarkan aturan grammar dan membangun Parse Tree.
2. Error handling, dimana syntax analyzer menemukan dan melaporkan error sesuai lokasinya.
3. Symbol table creation, dimana syntax analyzer menyimpan informasi simbol yang telah dibaca.

2.2. Parse Tree

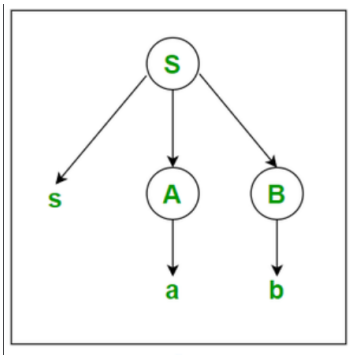
Parse tree adalah sebuah struktur pohon yang digunakan dalam compiler untuk menggambarkan bagaimana sebuah string atau kode program dibentuk berdasarkan aturan grammar tertentu. Struktur ini dibuat pada tahap parsing. Daun pohon dalam parse tree harus berupa simbol terminal, sementara node lainnya adalah simbol non-terminal. Jika pohon dibaca dari kiri ke kanan, pohon akan menghasilkan kembali input hasilnya.

Parse tree memiliki beberapa kegunaan penting dalam proses kompilasi. Struktur ini membantu melakukan analisis sintaks dengan menunjukkan secara jelas bagaimana input mengikuti aturan grammar bahasa. Selain itu, parse tree menyediakan representasi data di memori yang terorganisasi sesuai dengan struktur bahasa, sehingga memudahkan tahap-tahap lanjutan dalam compiler. Keuntungan lainnya adalah parse tree memungkinkan compiler melakukan beberapa kali pemrosesan terhadap data tanpa perlu mengulang proses parsing dari awal.

Sebagai contoh, ada grammar sebagai berikut

```
S -> sAB
A -> a
B -> b
```

Jika input string adalah “sab”, parse treenya berbentuk



Gambar 2.1 Parse tree untuk input “sab”

Sumber: [4]

2.3. Recursive Descent

Algoritma recursive descent parser adalah algoritma parser top-down yang menggunakan fungsi-fungsi rekursif, satu fungsi untuk setiap aturan grammar. Algoritma ini akan membaca input dari kiri ke kanan dan mencoba mencocokkannya dengan produksi grammar untuk membentuk parse tree. Algoritma ini mudah diimplementasikan sehingga sering digunakan untuk compiler kecil atau interpreter sederhana.

```
S()
{     Choose any S production, S ->X1X2…..Xk;
      for (i = 1 to k)
      {
          If ( Xi is a non-terminal)
          Call procedure Xi();
          else if ( Xi equals the current input, increment input)
          Else /* error has occurred, backtrack and try another
possibility */
      }
}
```

3. PERANCANGAN & IMPLEMENTASI

3.1. Perancangan

3.1.1. Aturan Grammar

Notasi:

`<non-terminal>`	Simbol non-terminal
`TERMINAL`	Simbol terminal (Token dari Lexer)
`::=`	“Didefinisikan sebagai”
`|`	Pilihan (OR)
`[ ... ]`	Opsional (0 atau 1 kali)
`{ ... }`	Repetisi (0 atau lebih kali)
`( ... )`	Pengelompokan
ε	Empty string / tidak ada

1. **Struktur Program Utama**

   Aturan-aturan ini mendefinisikan struktur keseluruhan dari sebuah program PASCAL-S.

```
<program> ::= <program-header> <block> DOT

<program-header> ::= KEYWORD(program) IDENTIFIER SEMICOLON

<block> ::= <declaration-part> <compount-statement>
```

2. **Bagian Deklarasi**

   Aturan-aturan untuk mendeklarasikan konstanta, tipe, variabel, dan subprogram.

```
<declaration-part> ::=
     { <const-declaration> }
     { <type-declaration> }
     { <var-declaration> }
     { <subprogram-declaration> }
```

```
<const-declaration> ::=
     KEYWORD(konstanta) (IDENTIFIER RELATIONAL_OPERATOR(=)
     <expression> SEMICOLON)+

<type-declaration> ::=
     KEYWORD(tipe) (IDENTIFIER RELATIONAL_OPERATOR(=) <type>
     SEMICOLON)+

<var-declaration> ::=
     KEYWORD(variabel) (<identifier-list> COLON <type>
     SEMICOLON)+

<identifier-list> ::= IDENTIFIER { COMMA IDENTIFIER }
```

3. Definisi Tipe

Aturan untuk mendefinisikan tipe data, baik sederhana maupun larik (*array*).

```
<type> ::= <simple-type> | <array-type>

<simple-type> ::=
     KEYWORD(integer)
     | KEYWORD(real)
     | KEYWORD(boolean)
     | KEYWORD(char)

<array-type> ::=
     KEYWORD(larik) LBRACKET <range> RBRACKET KEYWORD(dari)
     <type>

<range> ::= <expression> RANGE_OPERATOR <expression>
```

4. Deklarasi Subprogram (Prosedur & Fungsi)

Aturan untuk mendefinisikan prosedur dan fungsi.

```
<subprogram-declaration> ::=
     <procedure-declaration> | <function-declaration>

<procedure-declaration> ::=
     KEYWORD(prosedur) IDENTIFIER [ <formal-parameter-list>
     ] SEMICOLON <block> SEMICOLON

<function-declaration> ::=
```

```
     KEYWORD(fungsi) IDENTIFIER [ <formal-parameter-list> ]
     COLON <type> SEMICOLON <block> SEMICOLON

<formal-parameter-list> ::=
     LPARENTHESIS <parameter-group> { SEMICOLON
     <parameter-group> } RPARENTHESIS

<parameter-group> ::= <identifier-list> COLON <type>
```

5. Pernyataan (Statements)

Aturan-aturan yang mendefinisikan berbagai jenis pernyataan yang dapat dieksekusi.

```
<compound-statement> ::=
     KEYWORD(mulai) <statement-list> KEYWORD(selesai)

<statement-list> ::= <statement> { SEMICOLON <statement> }

<statement> ::=
     <assignment-statement>
     | <procedure-call>
     | <if-statement>
     | <while-statement>
     | <for-statement>
     | <compound-statement>
     | <empty-statement>

<empty-statement> ::= ε

<assignment-statement> ::=
     IDENTIFIER ASSIGN_OPERATOR <expression>

<procedure-call> ::=
     IDENTIFIER [ LPARENTHESIS <parameter-list> RPARENTHESIS
     ]

<parameter-list> ::= <expression> { COMMA <expression> }

<if-statement> ::=
     KEYWORD(jika) <expression> KEYWORD(maka) <statement> [
     KEYWORD(selain-itu) <statement> ]

<while-statement> ::=
     KEYWORD(selama) <expression> KEYWORD(lakukan)
     <statement>

<for-statement> ::=
```

```
     KEYWORD(untuk) IDENTIFIER ASSIGN_OPERATOR <expression>
     <direction> <expression> KEYWORD(lakukan) <statement>

<direction> ::= KEYWORD(ke) | KEYWORD(turun-ke)
```

6. Ekspresi (Expressions)

Aturan-aturan yang mendefinisikan ekspresi, operator, dan operand, dengan memperhatikan *operator precedence*.

```
<expression> ::=
     <simple-expression> [ <relational-operator>
     <simple-expression> ]

<simple-expression> ::=
     [ <sign> ] <term> { <additive-operator> <term> }

<term> ::=
     <factor> { <multiplicative-operator> <factor> }

<factor> ::=
     IDENTIFIER
     | NUMBER
     | CHAR_LITERAL
     | STRING_LITERAL
     | <function-call>
     | LPARENTHESIS <expression> RPARENTHESIS
     | LOGICAL_OPERATOR(tidak) <factor>

<function-call> ::=
     IDENTIFIER LPARENTHESIS [ <parameter-list> ]
     RPARENTHESIS

<sign> ::=
     ARITHMETIC_OPERATOR(+) | ARITHMETIC_OPERATOR(-)

<relational-operator> ::=
     RELATIONAL_OPERATOR(=)
     | RELATIONAL_OPERATOR(<>)
     | RELATIONAL_OPERATOR(<)
     | RELATIONAL_OPERATOR(<=)
     | RELATIONAL_OPERATOR(>)
     | RELATIONAL_OPERATOR(>=)

<additive-operator> ::=
     ARITHMETIC_OPERATOR(+)
     | ARITHMETIC_OPERATOR(-)
     | LOGICAL_OPERATOR(atau)
```

```
<multiplicative-operator> ::=
      ARITHMETIC_OPERATOR(*)
      | ARITHMETIC_OPERATOR(/)
      | ARITHMETIC_OPERATOR(bagi)
      | ARITHMETIC_OPERATOR(mod)
      | LOGICAL_OPERATOR(dan)
```

3.2. Implementasi

a. node.py

Kelas node adalah representasi dasar untuk membangun parse tree. Setiap objek node memiliki label yang menyimpan nama node, token yang menyimpan objek token terkait dengan node, dan children yang menyimpan daftar anak yang dimiliki node saat ini.

```
from src.common.pascal_token import Token

class Node:
    def __init__(self, label, token=None):
        self.label: str = label
        self.token: Token | None = token
        self.children: list[Node] = []

    def add_children(self, node):
        if node:
            self.children.append(node)
        return node

    def print_tree(self, prefix: str = "", is_last: bool = True):
        connector = "└── " if is_last else "├── "
        if self.token:
            print(prefix + connector + f"{self.label}({self.token.value})")
        else:
            print(prefix + connector + f"{self.label}")

        child_prefix = prefix + ("    " if is_last else "│   ")
        for i, child in enumerate(self.children):
            is_last_child = (i == len(self.children) - 1)
            child.print_tree(child_prefix, is_last_child)
```

b. Fungsi dasar kelas parser

```
class Parser:
    def __init__(self, tokens: list[Token], raise_on_error: bool = False):
        self.tokens = tokens
        self.current_index = 0
        self.errors = []
        self.raise_on_error = raise_on_error

    def peek(self) -> Token | None:
        # lihat token saat ini tanpa mengonsumsi
        if self.current_index < len(self.tokens):
            return self.tokens[self.current_index]
        return None

    def consume_token(self) -> Token | None:
        # ngambil token saat ini dan berpindah ke token berikutnya
        token = self.peek()
        if token:
            self.current_index += 1
        return token

    def match_token(self, expected_type: str, expected_value: str | None = None) ->
Token | None:
        # mastiin token saat ini sesuai dengan yg diharapkan.
        # kalau ngga, muncul error syntax
        token = self.peek()
        if token is None:
            self.error(f"{expected_type}({expected_value})" if expected_value else
expected_type, None)
            return None

        if token.token_type == expected_type and (expected_value is None or
token.value.lower() == expected_value.lower()):
            return self.consume_token()
        else:
            self.error(
                f"{expected_type}({expected_value})" if expected_value else
expected_type,
                token
            )
            return None

    def error(self, expected: str, actual_token: Token | None):
        actual_desc = self._fmt_token(actual_token)
        line, col = (actual_token.line, actual_token.column) if actual_token else
```

```
(None, None)
        msg = f"Syntax error: expected {expected}, but got {actual_desc}"
        logging.error(msg)
        self.errors.append(msg)
        if self.raise_on_error:
            raise TokenUnexpectedError(expected, actual_desc, line, col)

    def _fmt_token(self, tok: Token | None) -> str:
        if tok is None:
            return "EOF"
        return f"{tok.token_type}({tok.value}) @ {tok.line}:{tok.column}
```

c. Fungsi parser grammar utama

Fungsi ini memulai proses parsing berdasarkan aturan grammar utama. Fungsi akan membuat node root untuk parse tree, lalu memanggil parser untuk setiap bagian grammar dan menampilkan hasil parse tree.

```
    def parse_program(self):
        # Aturan grammar:
        # <program> ::= <program-header> <declaration-part> <compound-statement> DOT

        print("\nSTART PARSING...")

        program_node = Node("<program>")

        program_header = self.parse_program_header()
        if program_header:
            program_node.add_children(program_header)

        decl_part = self.parse_declaration_part()
        if decl_part:
            program_node.add_children(decl_part)

        compound_stmt = self.parse_compound_statement()
        if compound_stmt:
            program_node.add_children(compound_stmt)

        dot_token = self.match_token("DOT", ".")
        if dot_token:
            program_node.add_children(Node("DOT", dot_token))

        program_node.print_tree()
```

```
        print("\nFINISH PARSING...")
        return program_node
```

d. Fungsi header program

Fungsi ini mencakup proses parsing bagian awal program yang menentukan struktur dasar sebelum memasuki deklarasi dan block utama. Fungsi utama yang dipakai adalah parse_program_header(), yang bertugas membaca keyword program, nama program, dan penutup titik koma. Fungsi ini menjadi pintu masuk ke seluruh proses parsing setelah lexer.

```
def parse_program_header(self):
        """
        <program-header> ::= 'program' <identifier> ';'
        """
        node = Node("<program-header>")
        node.add_children(Node("KEYWORD", self.match_token("KEYWORD", "program")))
        node.add_children(Node("IDENTIFIER", self.match_token("IDENTIFIER")))
        node.add_children(Node("SEMICOLON", self.match_token("SEMICOLON", ";")))
        return node
```

e. Fungsi bagian deklarasi

Fungsi ini mencakup seluruh proses parsing deklarasi yang dapat muncul di bagian awal program maupun di dalam subprogram. Parser menggunakan fungsi parse_declaration_part() sebagai pengendali utama, lalu mengarahkan parsing ke kategori deklarasi yang sesuai. Proses ini meliputi deklarasi konstanta, tipe, dan variabel. Fungsi ini menggunakan *lookahead* untuk mendeteksi token awal, kemudian memanggil parser spesifik untuk tiap jenis deklarasi.

```
def parse_declaration_part(self):
        """<declaration-part> ::=
            { <const-declaration> }
            { <type-declaration> }
            { <var-declaration> }
            { <subprogram-declaration> }
        """
        node = Node("<declaration-part>")

        while True:
            tok = self.peek()
            if not tok or tok.token_type != "KEYWORD":
```

```
                break

            kw = tok.value.lower()

            if kw == "konstanta":
                const_node = self.parse_const_declaration()
                if const_node:
                    node.add_children(const_node)
                continue

            if kw == "tipe":
                type_decl = self.parse_type_declaration()
                if type_decl:
                    node.add_children(type_decl)
                continue

            if kw == "variabel":
                var_decl = self.parse_var_declaration()
                if var_decl:
                    node.add_children(var_decl)
                continue

            if kw in ("prosedur", "fungsi"):
                subprog = self.parse_subprogram_declaration()
                if subprog:
                    node.add_children(subprog)
                continue

            break

        return node if node.children else None

    # ====== CONST DECLARATION ======
    def parse_const_declaration(self):
        """<const-declaration> ::= 'konstanta' ( IDENTIFIER '=' <expression> ';'
)+"""
        node = Node("<const-declaration>")
        node.add_children(Node("KEYWORD", self.match_token("KEYWORD", "konstanta")))

        # Minimal satu definisi konstanta
        while True:
            ident = self.match_token("IDENTIFIER")
            if not ident:
                break
            node.add_children(Node("IDENTIFIER", ident))
```

```
            eq = self.match_token("RELATIONAL_OPERATOR", "=")
            if not eq:
                break
            node.add_children(Node("RELATIONAL_OPERATOR", eq))

            value_expr = self.parse_expression()
            if value_expr:
                node.add_children(value_expr)
            else:
                # fallback sederhana: kalau parse_expression belum diisi,
                # setidaknya konsumsi literal / identifier.
                lit = self.peek()
                if lit and lit.token_type in ("NUMBER", "CHAR_LITERAL",
"STRING_LITERAL", "IDENTIFIER"):
                    node.add_children(Node(lit.token_type, self.consume_token()))
                else:
                    break

            semi = self.match_token("SEMICOLON", ";")
            if not semi:
                break
            node.add_children(Node("SEMICOLON", semi))

            # cek apakah masih ada IDENTIFIER lagi (definisi konstanta berikutnya)
            nxt = self.peek()
            if not (nxt and nxt.token_type == "IDENTIFIER"):
                break

        return node

    # ====== TYPE DECLARATION ======
    def parse_type_declaration(self):
        """<type-declaration> ::= 'tipe' ( IDENTIFIER '=' <type> ';' )+"""
        node = Node("<type-declaration>")
        node.add_children(Node("KEYWORD", self.match_token("KEYWORD", "tipe")))

        while True:
            ident = self.match_token("IDENTIFIER")
            if not ident:
                break
            node.add_children(Node("IDENTIFIER", ident))

            eq = self.match_token("RELATIONAL_OPERATOR", "=")
            if not eq:
```

```
                break
            node.add_children(Node("RELATIONAL_OPERATOR", eq))

            type_node = self.parse_type()
            if type_node:
                node.add_children(type_node)

            semi = self.match_token("SEMICOLON", ";")
            if not semi:
                break
            node.add_children(Node("SEMICOLON", semi))

            nxt = self.peek()
            if not (nxt and nxt.token_type == "IDENTIFIER"):
                break

        return node
def parse_var_declaration(self):
        """<var-declaration> ::= 'variabel' ( <identifier-list> ':' <type> ';' )+"""
        node = Node("<var-declaration>")
        node.add_children(Node("KEYWORD", self.match_token("KEYWORD", "variabel")))

        while True:
            ident_list = self.parse_identifier_list()
            if not ident_list:
                break
            node.add_children(ident_list)

            node.add_children(Node("COLON", self.match_token("COLON", ":")))

            type_node = self.parse_type()
            if type_node:
                node.add_children(type_node)

            semi = self.match_token("SEMICOLON", ";")
            if not semi:
                break
            node.add_children(Node("SEMICOLON", semi))

            nxt = self.peek()
            # kalau setelah ';' masih IDENTIFIER, berarti masih dalam blok var yang
sama
            if not (nxt and nxt.token_type == "IDENTIFIER"):
                break
```

```
        return node
```

f. Fungsi tipe data (*type processing*)

Fungsi ini bertanggung jawab memproses berbagai bentuk tipe data yang dapat bertuliskan di dalam program PASCAL-S. Parser mendukung tipe dasar, tipe *custom*, dan tipe komposit seperti array. Fungsi parse_type() menjadi entry point yang mendistribusikan parsing ke fungsi lain. Untuk tipe array, parser menggunakan fungsi parse_array_type(), sementara rentang array ditangani oleh parse_range().

```
def parse_type(self):
        """<type> ::= 'integer' | 'real' | 'boolean' | 'char' | <array-type> |
IDENTIFIER"""
        node = Node("<type>")
        tok = self.peek()

        if not tok:
            self.error("type", None)
            return None

        if tok.token_type == "KEYWORD":
            kw = tok.value.lower()
            if kw in ("integer", "real", "boolean", "char"):
                node.add_children(Node("KEYWORD", self.consume_token()))
                return node
            if kw == "larik":
                array_node = self.parse_array_type()
                if array_node:
                    node.add_children(array_node)
                return node

        if tok.token_type == "IDENTIFIER": # custom type
            node.add_children(Node("IDENTIFIER", self.consume_token()))
            return node

        self.error("type", tok)
        return None

    def parse_array_type(self):
        """<array-type> ::= 'larik' '[' <range> ']' 'dari' <type>"""
        node = Node("<array-type>")

        node.add_children(Node("KEYWORD", self.match_token("KEYWORD", "larik")))
```

```
        node.add_children(Node("LBRACKET", self.match_token("LBRACKET", "[")))

        range_node = self.parse_range()
        if range_node:
            node.add_children(range_node)

        node.add_children(Node("RBRACKET", self.match_token("RBRACKET", "]")))
        node.add_children(Node("KEYWORD", self.match_token("KEYWORD", "dari")))

        elem_type = self.parse_type()
        if elem_type:
            node.add_children(elem_type)

        return node

    def parse_range(self):
        """<range> ::= <expression> RANGE_OPERATOR <expression>"""
        node = Node("<range>")

        left = self.parse_expression()
        if left:
            node.add_children(left)

        node.add_children(Node("RANGE_OPERATOR", self.match_token("RANGE_OPERATOR",
"..")))

        right = self.parse_expression()
        if right:
            node.add_children(right)

        return node

def parse_array_type(self):
        """<array-type> ::= 'larik' '[' <range> ']' 'dari' <type>"""
        node = Node("<array-type>")

        node.add_children(Node("KEYWORD", self.match_token("KEYWORD", "larik")))
        node.add_children(Node("LBRACKET", self.match_token("LBRACKET", "[")))

        range_node = self.parse_range()
        if range_node:
            node.add_children(range_node)

        node.add_children(Node("RBRACKET", self.match_token("RBRACKET", "]")))
        node.add_children(Node("KEYWORD", self.match_token("KEYWORD", "dari")))
```

```
        elem_type = self.parse_type()
        if elem_type:
            node.add_children(elem_type)

        return node
```

g. Fungsi Subprogram (*procedure* dan *function*)

Fungsi ini mencakup parsing deklarasi subprogram. Fungsi parse_subprogram_declaration() menentukan apakah token awal merupakan prosedur atau fungsi, kemudian memanggil fungsi parser yang sesuai. Fungsi ini memproses nama subprogram, parameter formal, tipe pengembalian, dan block internal subprogram. Parameter formal diproses oleh parse_formal_parameter_list() dan parse_parameter_group().

```
   def parse_subprogram_declaration(self):
        """<subprogram-declaration> ::= <procedure-declaration> |
<function-declaration>"""
        tok = self.peek()
        if not tok or tok.token_type != "KEYWORD":
            return None

        if tok.value.lower() == "prosedur":
            return self.parse_procedure_declaration()
        if tok.value.lower() == "fungsi":
            return self.parse_function_declaration()
        return None

    def parse_procedure_declaration(self):
        """<procedure-declaration> ::
            'prosedur' IDENTIFIER [ <formal-parameter-list> ] ';' <block> ';'
        """
        node = Node("<procedure-declaration>")

        node.add_children(Node("KEYWORD", self.match_token("KEYWORD", "prosedur")))
        node.add_children(Node("IDENTIFIER", self.match_token("IDENTIFIER")))

        # [ formal-parameter-list ]
        tok = self.peek()
        if tok and tok.token_type == "LPARENTHESIS":
            fp = self.parse_formal_parameter_list()
            if fp:
```

```
                node.add_children(fp)

        node.add_children(Node("SEMICOLON", self.match_token("SEMICOLON", ";")))

        block = self.parse_block()
        if block:
            node.add_children(block)

        node.add_children(Node("SEMICOLON", self.match_token("SEMICOLON", ";")))

        return node

    def parse_function_declaration(self):
        """<function-declaration> ::
            'fungsi' IDENTIFIER [ <formal-parameter-list> ] ':' <type> ';' <block>
';'
        """
        node = Node("<function-declaration>")

        node.add_children(Node("KEYWORD", self.match_token("KEYWORD", "fungsi")))
        node.add_children(Node("IDENTIFIER", self.match_token("IDENTIFIER")))

        tok = self.peek()
        if tok and tok.token_type == "LPARENTHESIS":
            fp = self.parse_formal_parameter_list()
            if fp:
                node.add_children(fp)

        node.add_children(Node("COLON", self.match_token("COLON", ":")))

        ret_type = self.parse_type()
        if ret_type:
            node.add_children(ret_type)

        node.add_children(Node("SEMICOLON", self.match_token("SEMICOLON", ";")))

        block = self.parse_block()
        if block:
            node.add_children(block)

        node.add_children(Node("SEMICOLON", self.match_token("SEMICOLON", ";")))

        return node

    def parse_formal_parameter_list(self):
```

```
        """<formal-parameter-list> ::=
            '(' <parameter-group> ( ';' <parameter-group> )* ')'
        """
        node = Node("<formal-parameter-list>")

        node.add_children(Node("LPARENTHESIS", self.match_token("LPARENTHESIS",
"(")))

        param_group = self.parse_parameter_group()
        if param_group:
            node.add_children(param_group)

        while True:
            tok = self.peek()
            if not tok or tok.token_type != "SEMICOLON":
                break
            semi = self.consume_token()
            node.add_children(Node("SEMICOLON", semi))

            param_group = self.parse_parameter_group()
            if not param_group:
                break
            node.add_children(param_group)

        node.add_children(Node("RPARENTHESIS", self.match_token("RPARENTHESIS",
")")))
        return node

    def parse_parameter_group(self):
        """<parameter-group> ::= <identifier-list> ':' <type>"""
        node = Node("<parameter-group>")

        ident_list = self.parse_identifier_list()
        if not ident_list:
            return None
        node.add_children(ident_list)

        node.add_children(Node("COLON", self.match_token("COLON", ":")))

        type_node = self.parse_type()
        if type_node:
            node.add_children(type_node)

        return node
```

h. Fungsi block program

Block adalah satuan struktur yang terdiri dari deklarasi lokal dan sebuah *compound statement*. Fungsi parse_block() mengatur urutan parsing. Pertama memproses deklarasi menggunakan parse_declaration_part(), kemudian memporses *compound statement* menggunakan parse_compound_statement().

```
   def parse_block(self):
        """<block> ::= <declaration-part> <compound-statement>

        Dipakai oleh deklarasi procedure / function.
        """
        node = Node("<block>")
        decl = self.parse_declaration_part()
        if decl:
            node.add_children(decl)
        comp = self.parse_compound_statement()
        if comp:
            node.add_children(comp)
        return node
```

i. Fungsi parse compound statement

Fungsi ini membangun parse tree dari berbagai statement dalam bahasa. Fungsi parse_statement() menentukan jenis statement berdasarkan token awal, lalu memanggil fungsi yang memproses jenis statement tersebut.

```
def parse_statement(self):
        tok = self.peek()
        if not tok:
            self.error("statement", None)
            return None
        if tok.token_type == "KEYWORD":
            kw = tok.value.lower()
            if kw == "jika":
                return self.parse_if_statement()
            elif kw == "selama":
                return self.parse_while_statement()
            elif kw == "untuk":
                return self.parse_for_statement()
            elif kw == "mulai":
                return self.parse_compound_statement()
        if tok.token_type == "IDENTIFIER":
            # liat token kedua untuk memutuskan ini function call atau IDENTIFIER
```

```
biasa
            next_tok_index = self.current_index + 1
            if next_tok_index < len(self.tokens) and
self.tokens[next_tok_index].token_type == "ASSIGN_OPERATOR":
                return self.parse_assignment_statement()
            else:
                return self.parse_procedure_call()
        self.error("statement", tok)
        return None

    def parse_if_statement(self):
        # <if-statement> ::= 'if' <expression> 'then' <statement> [ 'else'
<statement> ]

        node = Node("<if-statement>")
        node.add_children(Node("KEYWORD", self.match_token("KEYWORD", "jika")))

        node.add_children(self.parse_expression())

        node.add_children(Node("KEYWORD", self.match_token("KEYWORD", "maka")))
        node.add_children(self.parse_statement())

        token = self.peek()
        if token and token.token_type == "KEYWORD" and token.value.lower() in
("selain_itu"):
            node.add_children(Node("KEYWORD", self.consume_token()))
            node.add_children(self.parse_statement())

        return node

    def parse_while_statement(self):
        # <while-statement> ::= 'while' <expression> 'do' <statement>

        node = Node("<while-statement>")
        node.add_children(Node("KEYWORD", self.match_token("KEYWORD", "selama")))

        node.add_children(self.parse_expression())

        node.add_children(Node("KEYWORD", self.match_token("KEYWORD", "lakukan")))
        node.add_children(self.parse_statement())

        return node

    def parse_for_statement(self):
        # <for-statement> ::= 'untuk' IDENTIFIER ':=' <expression> ('ke'|'turun_ke')
```

```
<expression> 'lakukan' <statement>

        node = Node("<for-statement>")
        node.add_children(Node("KEYWORD", self.match_token("KEYWORD", "untuk")))
        node.add_children(Node("IDENTIFIER", self.match_token("IDENTIFIER")))
        node.add_children(Node("ASSIGN_OPERATOR",
self.match_token("ASSIGN_OPERATOR", ":=")))
        node.add_children(self.parse_expression())
        dir_tok = self.peek()
        if dir_tok and dir_tok.token_type == "KEYWORD" and dir_tok.value.lower() in
("ke", "turun_ke", "turun-ke"):
            node.add_children(Node("KEYWORD", self.consume_token()))
        else:
            self.error("KEYWORD(ke|turun_ke)", dir_tok)
        node.add_children(self.parse_expression())
        node.add_children(Node("KEYWORD", self.match_token("KEYWORD", "lakukan")))
        node.add_children(self.parse_statement())

        return node

    # remove duplicate earlier variant of parse_assignment_statement (kept the
robust version below)

    def parse_compound_statement(self):
        """
        <compound-statement> ::= 'begin' <statement-list> 'end'
        """
        node = Node("<compound-statement>")

        tok_begin = self.match_token("KEYWORD", "mulai")
        if not tok_begin:
            # Tidak ada 'mulai', error fatal untuk compound statement
            self.error("KEYWORD(mulai)", self.peek())
            return None

        node.add_children(Node("KEYWORD", tok_begin))

        # Cek apakah bloknya kosong (langsung 'selesai')
        if self.peek() and self.peek().value.lower() == "selesai":
            pass
        else:
            # Parse <statement> pertama
            statement_node = self.parse_statement()
            if not statement_node: # Gagal parse statement pertama, ini error
                return node
```

```
            node.add_children(statement_node)

            # Loop untuk { SEMICOLON <statement> }
            while self.peek() and self.peek().token_type == "SEMICOLON":
                semicolon_node = Node("SEMICOLON", self.consume_token())

                # Handle trailing semicolon (valid): '...; selesai'
                if self.peek() and self.peek().value.lower() == "selesai":
                    node.add_children(semicolon_node)
                    break

                node.add_children(semicolon_node) # Tambahkan semicolon

                # Wajib ada statement setelah semicolon (jika bukan 'selesai')
                statement_node = self.parse_statement()
                if not statement_node:
                    return node
                node.add_children(statement_node)

        end_tok = self.match_token("KEYWORD", "selesai")
        if not end_tok:
            self.error("KEYWORD(selesai)", self.peek())
        else:
            node.add_children(Node("KEYWORD", end_tok))

        return node

    def parse_assignment_statement(self):
        # <assignment-statement> ::= IDENTIFIER ASSIGN_OPERATOR <expression>
        node = Node("<assignment-statement>")

        # IDENTIFIER
        ident = self.match_token("IDENTIFIER")
        if not ident: return None
        node.add_children(Node("IDENTIFIER", ident))

        # ASSIGN_OPERATOR
        op = self.match_token("ASSIGN_OPERATOR", ":=")
        if not op: return None
        node.add_children(Node("ASSIGN_OPERATOR", op))

        expr_node = self.parse_expression()
        if not expr_node:
            self.error("expression", self.peek())
            return None
```

```
        node.add_children(expr_node)

        return node
```

j. Fungsi parse pemanggilan subprogram (*procedure* dan *function*)

Fungsi ini bertanggung jawab untuk mem-*parsing* pemanggilan subprogram (prosedur atau fungsi) yang terjadi di dalam badan kode. Fungsi utama di sini adalah parse_procedure_function_call. Fungsi tersebut dirancang untuk mengenali sintaks pemanggilan yang dimulai dengan IDENTIFIER.

```
   def parse_procedure_function_call(self):
        # <procedure/function-call> ::= IDENTIFIER LPARENTHESIS [ <parameter-list> ]
RPARENTHESIS
        node = Node("<procedure-function-call>")

        tok = self.peek()
        if not tok:
            return None
        if tok.token_type == "IDENTIFIER":
            node.add_children(Node("IDENTIFIER", self.consume_token()))
        else:
            return None

        lparen = self.match_token("LPARENTHESIS", "(")
        if not lparen:
            return None
        node.add_children(Node("LPARENTHESIS", lparen))

        param_list_node = self.parse_parameter_list()
        if param_list_node:
            node.add_children(param_list_node)

        rparen = self.match_token("RPARENTHESIS", ")")
        if not rparen:
            return None
        node.add_children(Node("RPARENTHESIS", rparen))

        return node


    def parse_parameter_list(self):
        # <parameter-list> ::= <expression> { COMMA <expression> }
        node = Node("<parameter-list>")
```

```
        # <expression> pertama
        expr_node = self.parse_expression()
        if not expr_node:
            # List parameter boleh kosong (misal: writeln())
            return None

        node.add_children(expr_node)

        # { COMMA <expression> }
        while True:
            tok = self.peek()
            if not tok or tok.token_type != "COMMA":
                break
            comma_node = Node("COMMA", self.consume_token())
            node.add_children(comma_node)

            expr_node = self.parse_expression()
            if not expr_node:
                self.error("expression", self.peek())
                return None # Error, koma harus diikuti ekspresi
            node.add_children(expr_node)

        return node
```

k. Fungsi parse *expression* serta *operator*

Bagian ini merupakan inti dari parser dalam menangani ekspresi aritmatika dan logika. fungsi ini memberlakukan aturan prioritas operator melalui hierarki pemanggilan fungsi.

```
   def parse_expression(self):
        # <expression> ::= <simple-expression> [ <relational-operator>
<simple-expression> ]
        node = Node("<expression>")

        left_simple_expr = self.parse_simple_expression()
        if not left_simple_expr:
            self.error("simple-expression", self.peek())
            return None
        node.add_children(left_simple_expr)

        # [ <relational-operator> <simple-expression> ]
        op_node = self.parse_relational_operator()
```

```
        if op_node:
            # kalo ada operator relasional, parse <simple-expression> kedua
            node.add_children(op_node)

            right_simple_expr = self.parse_simple_expression()
            if not right_simple_expr:
                self.error("simple-expression", self.peek())
                return None
            node.add_children(right_simple_expr)

        return node

    def parse_simple_expression(self):
        # <simple-expression> ::= [ <sign> ] <term> { <additive-operator> <term> }
        node = Node("<simple-expression>")

        # opsional [ <sign> ] (+ atau -)
        tok = self.peek()
        if tok and tok.token_type == "ARITHMETIC_OPERATOR" and tok.value in ('+',
'-'):
            sign_node = Node("SIGN", self.consume_token())
            node.add_children(sign_node)

        term_node = self.parse_term()
        if not term_node:
            self.error("term", self.peek())
            return None
        node.add_children(term_node)

        # loop untuk { <additive-operator> <term> }
        while True:
            op_node = self.parse_additive_operator()

            if not op_node:
                break

            node.add_children(op_node)

            # kalo ada operator, wajib ada <term> berikutnya
            term_node = self.parse_term()
            if not term_node:
                self.error("term", self.peek())
                return None
            node.add_children(term_node)
```

```
        return node

    def parse_term(self):
        """
        <term> ::= <factor> ( <multiplicative-operator> <factor> )*
        """
        node = Node("<term>")

        # first factor
        first_factor = self.parse_factor()
        if not first_factor:
            _tok = self.peek()
            self.error("factor", _tok)
            return None
        node.add_children(first_factor)

        # (* op factor)
        while True:
            op_node = self.parse_multiplicative_operator()
            if not op_node:
                break
            node.add_children(op_node)

            rhs = self.parse_factor()
            if not rhs:
                _tok2 = self.peek()
                self.error("factor", _tok2)
                return None
            node.add_children(rhs)

        return node

    def parse_factor(self):
        """
        <factor> ::= IDENTIFIER
                   | <function-call>
                   | NUMBER
                   | CHAR_LITERAL
                   | STRING_LITERAL
                   | LPARENTHESIS <expression> RPARENTHESIS
                   | LOGICAL_OPERATOR(tidak) <factor>
        Catatan: Pemanggilan fungsi/prosedur (IDENTIFIER (...)) akan ditangani di
rule lain
        atau dapat diperluas kemudian; di sini fokus pada bentuk-bentuk dasar sesuai
permintaan.
```

```
        """
        node = Node("<factor>")
        tok = self.peek()

        if tok is None:
            self.error("factor", None)
            return None

        # unary logical NOT: 'tidak'
        if tok.token_type == "LOGICAL_OPERATOR" and tok.value.lower() == "tidak":
            not_tok = self.consume_token()
            node.add_children(Node("LOGICAL_OPERATOR", not_tok))
            sub = self.parse_factor()
            if not sub:
                _tok3 = self.peek()
                self.error("factor", _tok3)
                return None
            node.add_children(sub)
            return node

        # parenthesized expression
        if tok.token_type == "LPARENTHESIS" and tok.value == "(":
            lpar = self.consume_token()
            node.add_children(Node("LPARENTHESIS", lpar))

            expr = self.parse_expression()
            if not expr:
                _tok4 = self.peek()
                self.error("expression", _tok4)
                return None
            node.add_children(expr)

            rpar = self.match_token("RPARENTHESIS", ")")
            if not rpar:
                return None
            node.add_children(Node("RPARENTHESIS", rpar))
            return node

        # literals and identifier
        if tok.token_type in ("NUMBER", "CHAR_LITERAL", "STRING_LITERAL"):
            node.add_children(Node(tok.token_type, self.consume_token()))
            return node

        if tok.token_type == "IDENTIFIER":
            # liat token kedua untuk memutuskan ini function call atau IDENTIFIER
```

```
biasa
            next_tok_index = self.current_index + 1
            if next_tok_index < len(self.tokens) and
self.tokens[next_tok_index].token_type == "LPARENTHESIS":
                return self.parse_function_call()
            else:
                node.add_children(Node("IDENTIFIER", self.consume_token()))
                return node

        # if no form matched
        self.error("factor", tok)
        return None

    def parse_relational_operator(self):
        """
        <relational-operator> ::= '=' | '<>' | '<' | '<=' | '>' | '>='
        """
        tok = self.peek()
        if tok and tok.token_type == "RELATIONAL_OPERATOR" and tok.value in ("=",
"<>", "<", "<=", ">", ">="):
            node = Node("<relational-operator>")
            node.add_children(Node("RELATIONAL_OPERATOR", self.consume_token()))
            return node
        return None

    def parse_additive_operator(self):
        """
        <additive-operator> ::= '+' | '-' | 'atau'
        '+' | '-' bertipe ARITHMETIC_OPERATOR, 'atau' bertipe LOGICAL_OPERATOR
        """
        tok = self.peek()
        if tok is None:
            return None

        if tok.token_type == "ARITHMETIC_OPERATOR" and tok.value in ("+", "-"):
            node = Node("<additive-operator>")
            node.add_children(Node("ARITHMETIC_OPERATOR", self.consume_token()))
            return node
        if tok.token_type == "LOGICAL_OPERATOR" and tok.value.lower() == "atau":
            node = Node("<additive-operator>")
            node.add_children(Node("LOGICAL_OPERATOR", self.consume_token()))
            return node
        return None

    def parse_multiplicative_operator(self):
```

```
        """
        <multiplicative-operator> ::= '*' | '/' | 'bagi' | 'mod' | 'dan'
        '*' '/' 'bagi' 'mod' bertipe ARITHMETIC_OPERATOR, 'dan' bertipe
LOGICAL_OPERATOR
        """
        tok = self.peek()
        if tok is None:
            return None

        if tok.token_type == "ARITHMETIC_OPERATOR" and tok.value in ("*", "/",
"bagi", "mod"):
            node = Node("<multiplicative-operator>")
            node.add_children(Node("ARITHMETIC_OPERATOR", self.consume_token()))
            return node
        if tok.token_type == "LOGICAL_OPERATOR" and tok.value.lower() == "dan":
            node = Node("<multiplicative-operator>")
            node.add_children(Node("LOGICAL_OPERATOR", self.consume_token()))
            return node
        return None
```

4. Pengujian

4.1. Pengujian 0

Tujuan pengujian: Untuk memvalidasi bahwa *parser* dapat mengenali struktur program PASCAL-S yang paling minimal. Pengujian ini memastikan integrasi dasar antara fungsi parse_program, parse_program_header, dan parse_compound_statement (kosong) berjalan dengan sukses tanpa *crash*.

File: 0-smoke-test.pas

Input

```
program tes;
mulai
selesai.
```

Output

```
KEYWORD(program)
IDENTIFIER(tes)
SEMICOLON(;)
KEYWORD(mulai)
KEYWORD(selesai)
DOT(.)

START PARSING...

└── <program>
    ├── <program-header>
    │   ├── KEYWORD(program)
    │   ├── IDENTIFIER(tes)
    │   └── SEMICOLON(;)
    ├── <compound-statement>
    │   ├── KEYWORD(mulai)
    │   └── KEYWORD(selesai)
    └── DOT(.)

FINISH PARSING...
```

4.2. Pengujian 1

Tujuan pengujian: Untuk memvalidasi kemampuan *parser* dalam mem-parsing <declaration-part> yang berisi variabel sederhana. Pengujian mengujia <compound-statement> yang berisi satu <assigment-statement>, serta memastikan penanganan *statement list* dan semicolon dengan benar.

File: 1-basic.pas

Input

```
program Basic;
variabel
   x: integer;
mulai
   x := 10;
selesai.
```

Output

```
KEYWORD(program)
IDENTIFIER(Basic)
SEMICOLON(;)
KEYWORD(variabel)
IDENTIFIER(x)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(mulai)
IDENTIFIER(x)
ASSIGN_OPERATOR(:=)
NUMBER(10)
SEMICOLON(;)
KEYWORD(selesai)
DOT(.)

START PARSING...

└── <program>
    ├── <program-header>
    │   ├── KEYWORD(program)
    │   ├── IDENTIFIER(Basic)
    │   └── SEMICOLON(;)
    ├── <declaration-part>
    │   └── <var-declaration>
    │       ├── KEYWORD(variabel)
    │       ├── <identifier-list>
    │       │   └── IDENTIFIER(x)
    │       ├── COLON(:)
    │       ├── <type>
    │       │   └── KEYWORD(integer)
    │       └── SEMICOLON(;)
    ├── <compound-statement>
    │   ├── KEYWORD(mulai)
    │   ├── <assignment-statement>
    │   │   ├── IDENTIFIER(x)
    │   │   ├── ASSIGN_OPERATOR(:=)
    │   │   └── <expression>
    │   │       └── <simple-expression>
    │   │           └── <term>
    │   │               └── <factor>
    │   │                   └── NUMBER(10)
    │   ├── SEMICOLON(;)
    │   └── KEYWORD(selesai)
```

```
    └── DOT(.)

FINISH PARSING...
```

4.3. Pengujian 2

Tujuan pengujian: Untuk memvalidasi kemampuan *parser* dalam menangani ekspresi aritmatika dan logika. Pengujian ini secara spesifik menguji implementasi operator precedence dan *parsing* ekspresi di dalam tanda kurung. Pengujian ini juga memvalidasi *parsing* struktur kontrol jika-maka-selain-itu.

File: 2-expression.pas

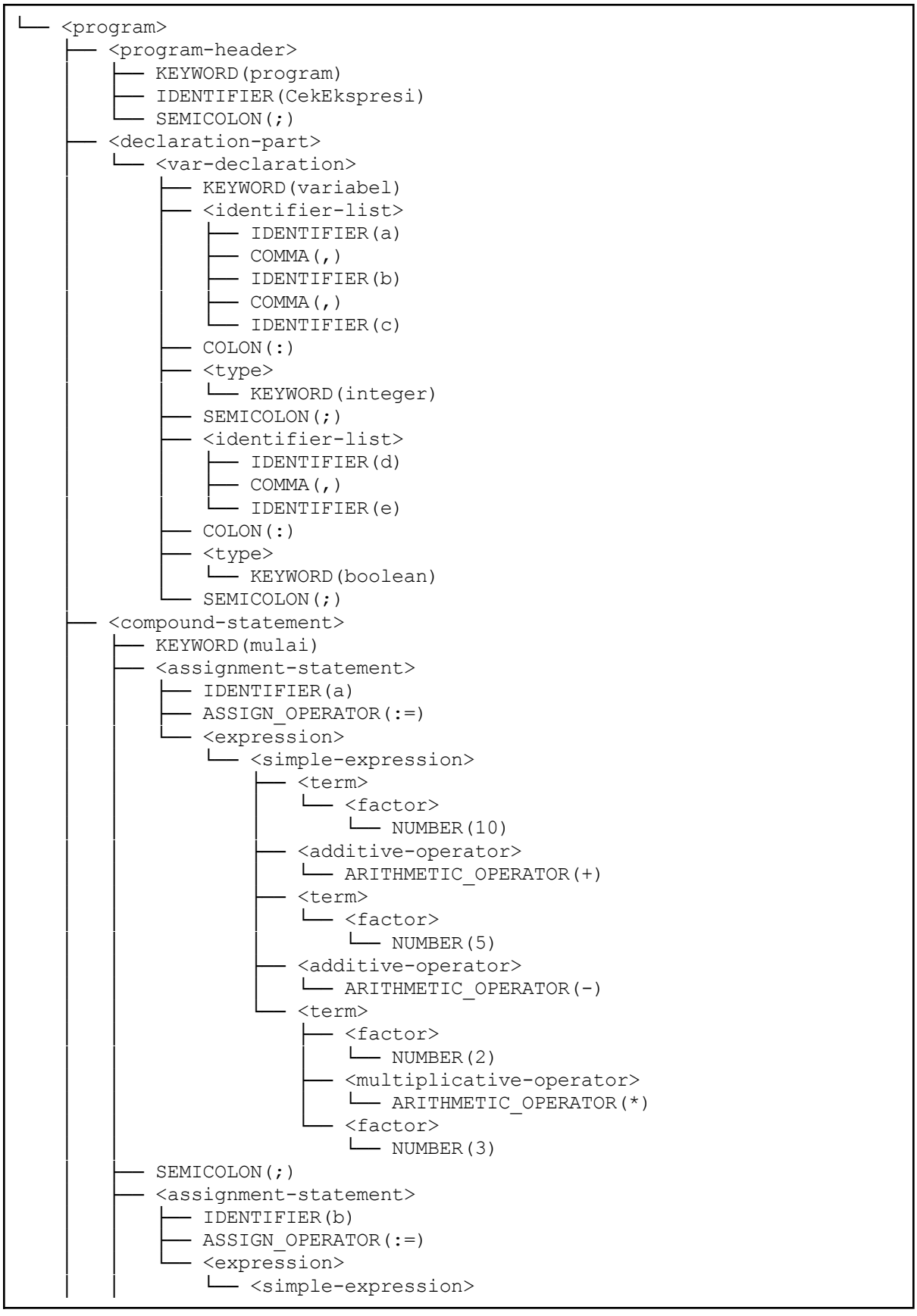
Input

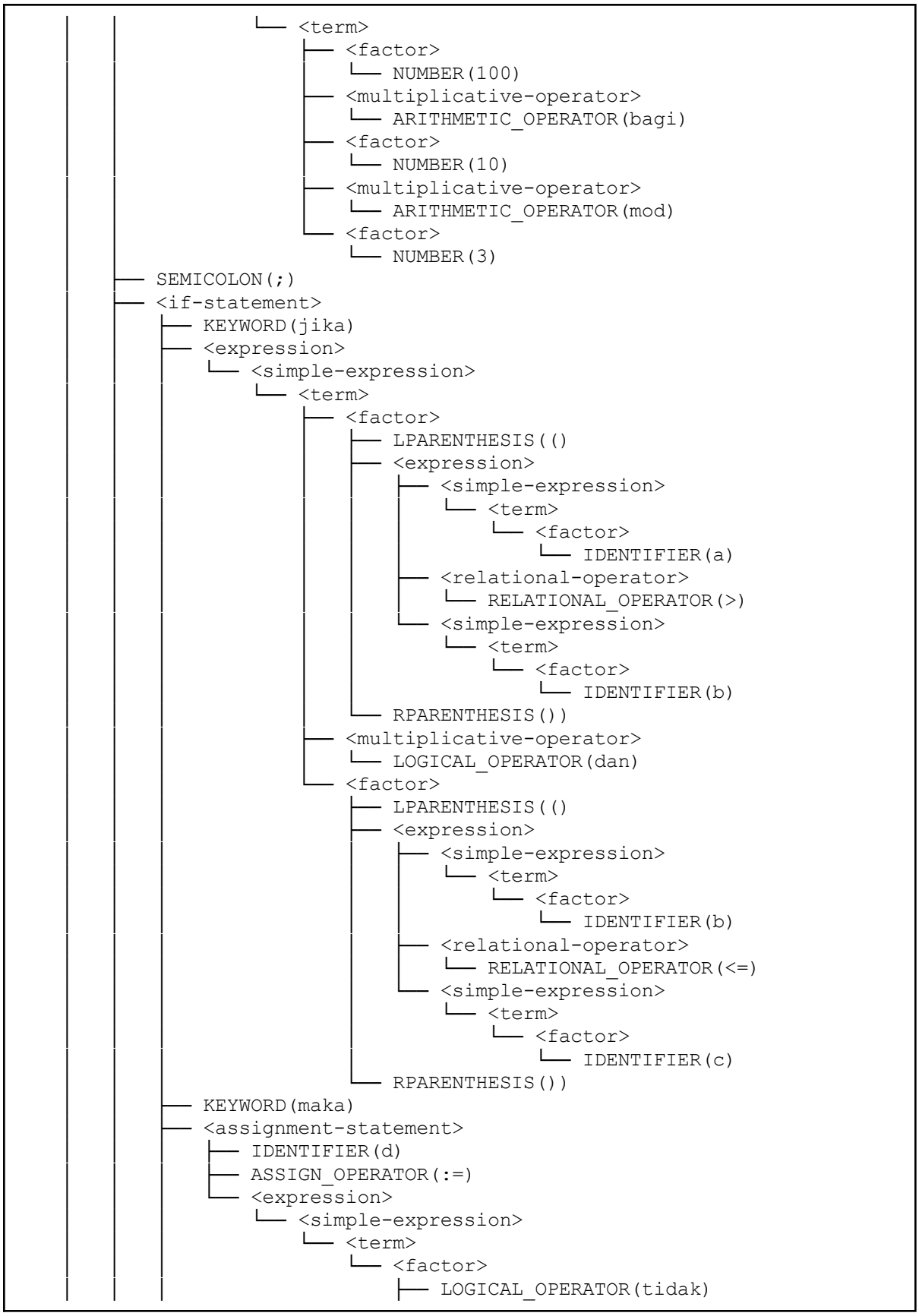
```
program CekEkspresi;
variabel
   a, b, c: integer;
   d, e: boolean;
mulai
   { Tes Aritmatika }
   a := 10 + 5 - 2 * 3;
   b := 100 bagi 10 mod 3;

   { Tes Relasional & Logika }
   jika (a > b) dan (b <= c) maka
      d := tidak e
   selain_itu
      e := (a <> c) atau (c = 10);
selesai.
```

Output

```
KEYWORD(program)
IDENTIFIER(CekEkspresi)
SEMICOLON(;)
KEYWORD(variabel)
IDENTIFIER(a)
COMMA(,)
IDENTIFIER(b)
COMMA(,)
IDENTIFIER(c)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(d)
COMMA(,)
IDENTIFIER(e)
COLON(:)
KEYWORD(boolean)
SEMICOLON(;)
```

```
KEYWORD(mulai)
IDENTIFIER(a)
ASSIGN_OPERATOR(:=)
NUMBER(10)
ARITHMETIC_OPERATOR(+)
NUMBER(5)
ARITHMETIC_OPERATOR(-)
NUMBER(2)
ARITHMETIC_OPERATOR(*)
NUMBER(3)
SEMICOLON(;)
IDENTIFIER(b)
ASSIGN_OPERATOR(:=)
NUMBER(100)
ARITHMETIC_OPERATOR(bagi)
NUMBER(10)
ARITHMETIC_OPERATOR(mod)
NUMBER(3)
SEMICOLON(;)
KEYWORD(jika)
LPARENTHESIS(()
IDENTIFIER(a)
RELATIONAL_OPERATOR(>)
IDENTIFIER(b)
RPARENTHESIS())
LOGICAL_OPERATOR(dan)
LPARENTHESIS(()
IDENTIFIER(b)
RELATIONAL_OPERATOR(<=)
IDENTIFIER(c)
RPARENTHESIS())
KEYWORD(maka)
IDENTIFIER(d)
ASSIGN_OPERATOR(:=)
LOGICAL_OPERATOR(tidak)
IDENTIFIER(e)
KEYWORD(selain_itu)
IDENTIFIER(e)
ASSIGN_OPERATOR(:=)
LPARENTHESIS(()
IDENTIFIER(a)
RELATIONAL_OPERATOR(<>)
IDENTIFIER(c)
RPARENTHESIS())
LOGICAL_OPERATOR(atau)
LPARENTHESIS(()
IDENTIFIER(c)
RELATIONAL_OPERATOR(=)
NUMBER(10)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(selesai)
DOT(.)

START PARSING...
```


└── <program>
├── <program-header>
│ ├── KEYWORD(program)
│ ├── IDENTIFIER(CekEkspresi)
│ └── 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(d)
│ │ ├── COMMA(,)
│ │ └── IDENTIFIER(e)
│ ├── COLON(:)
│ ├── <type>
│ │ └── KEYWORD(boolean)
│ └── SEMICOLON(;)
├── <compound-statement>
│ ├── KEYWORD(mulai)
│ ├── <assignment-statement>
│ │ ├── IDENTIFIER(a)
│ │ ├── ASSIGN_OPERATOR(:=)
│ │ └── <expression>
│ │ └── <simple-expression>
│ │ ├── <term>
│ │ │ └── <factor>
│ │ │ └── NUMBER(10)
│ │ ├── <additive-operator>
│ │ │ └── ARITHMETIC_OPERATOR(+)
│ │ ├── <term>
│ │ │ └── <factor>
│ │ │ └── NUMBER(5)
│ │ ├── <additive-operator>
│ │ │ └── ARITHMETIC_OPERATOR(-)
│ │ └── <term>
│ │ ├── <factor>
│ │ │ └── NUMBER(2)
│ │ ├── <multiplicative-operator>
│ │ │ └── ARITHMETIC_OPERATOR(*)
│ │ └── <factor>
│ │ └── NUMBER(3)
│ ├── SEMICOLON(;)
│ ├── <assignment-statement>
│ │ ├── IDENTIFIER(b)
│ │ ├── ASSIGN_OPERATOR(:=)
│ │ └── <expression>
│ │ └── <simple-expression>


└── <term>
├── <factor>
└── NUMBER(100)
├── <multiplicative-operator>
└── ARITHMETIC_OPERATOR(bagi)
├── <factor>
└── NUMBER(10)
├── <multiplicative-operator>
└── ARITHMETIC_OPERATOR(mod)
└── <factor>
└── NUMBER(3)
├── SEMICOLON(;)
├── <if-statement>
├── KEYWORD(jika)
├── <expression>
└── <simple-expression>
└── <term>
├── <factor>
├── LPARENTHESIS(()
├── <expression>
├── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(a)
├── <relational-operator>
└── RELATIONAL_OPERATOR(>)
└── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(b)
└── RPARENTHESIS())
├── <multiplicative-operator>
└── LOGICAL_OPERATOR(dan)
└── <factor>
├── LPARENTHESIS(()
├── <expression>
├── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(b)
├── <relational-operator>
└── RELATIONAL_OPERATOR(<=)
└── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(c)
└── RPARENTHESIS())
├── KEYWORD(maka)
├── <assignment-statement>
├── IDENTIFIER(d)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
├── LOGICAL_OPERATOR(tidak)

```
    │   │   │                   └── <factor>
    │   │   │                       └── IDENTIFIER(e)
    │   │   ├── KEYWORD(selain_itu)
    │   │   └── <assignment-statement>
    │   │       ├── IDENTIFIER(e)
    │   │       ├── ASSIGN_OPERATOR(:=)
    │   │       └── <expression>
    │   │           └── <simple-expression>
    │   │               ├── <term>
    │   │               │   └── <factor>
    │   │               │       ├── LPARENTHESIS(()
    │   │               │       ├── <expression>
    │   │               │       │   ├── <simple-expression>
    │   │               │       │   │   └── <term>
    │   │               │       │   │       └── <factor>
    │   │               │       │   │           └── IDENTIFIER(a)
    │   │               │       │   ├── <relational-operator>
    │   │               │       │   │   └── RELATIONAL_OPERATOR(<>)
    │   │               │       │   └── <simple-expression>
    │   │               │       │       └── <term>
    │   │               │       │           └── <factor>
    │   │               │       │               └── IDENTIFIER(c)
    │   │               │       └── RPARENTHESIS())
    │   │               ├── <additive-operator>
    │   │               │   └── LOGICAL_OPERATOR(atau)
    │   │               └── <term>
    │   │                   └── <factor>
    │   │                       ├── LPARENTHESIS(()
    │   │                       ├── <expression>
    │   │                       │   ├── <simple-expression>
    │   │                       │   │   └── <term>
    │   │                       │   │       └── <factor>
    │   │                       │   │           └── IDENTIFIER(c)
    │   │                       │   ├── <relational-operator>
    │   │                       │   │   └── RELATIONAL_OPERATOR(=)
    │   │                       │   └── <simple-expression>
    │   │                       │       └── <term>
    │   │                       │           └── <factor>
    │   │                       │               └── NUMBER(10)
    │   │                       └── RPARENTHESIS())
    │   ├── SEMICOLON(;)
    │   └── KEYWORD(selesai)
    └── DOT(.)

FINISH PARSING...
```

4.4. Pengujian 3

Tujuan pengujian: Untuk memvalidasi kemampuan *parser* dalam mem-parsing keseluruhan <declaration-part> secara berurutan, mencakup konstanta, tipe (termasuk larik), variabel, prosedur, dan fungsi. Pengujian ini juga memvalidasi penggunaan tipe data *custom* (misalnya MyArray) dan pemanggilan prosedur serta fungsi dari dalam blok program utama.

File: 3-all-declaration.pas

Input

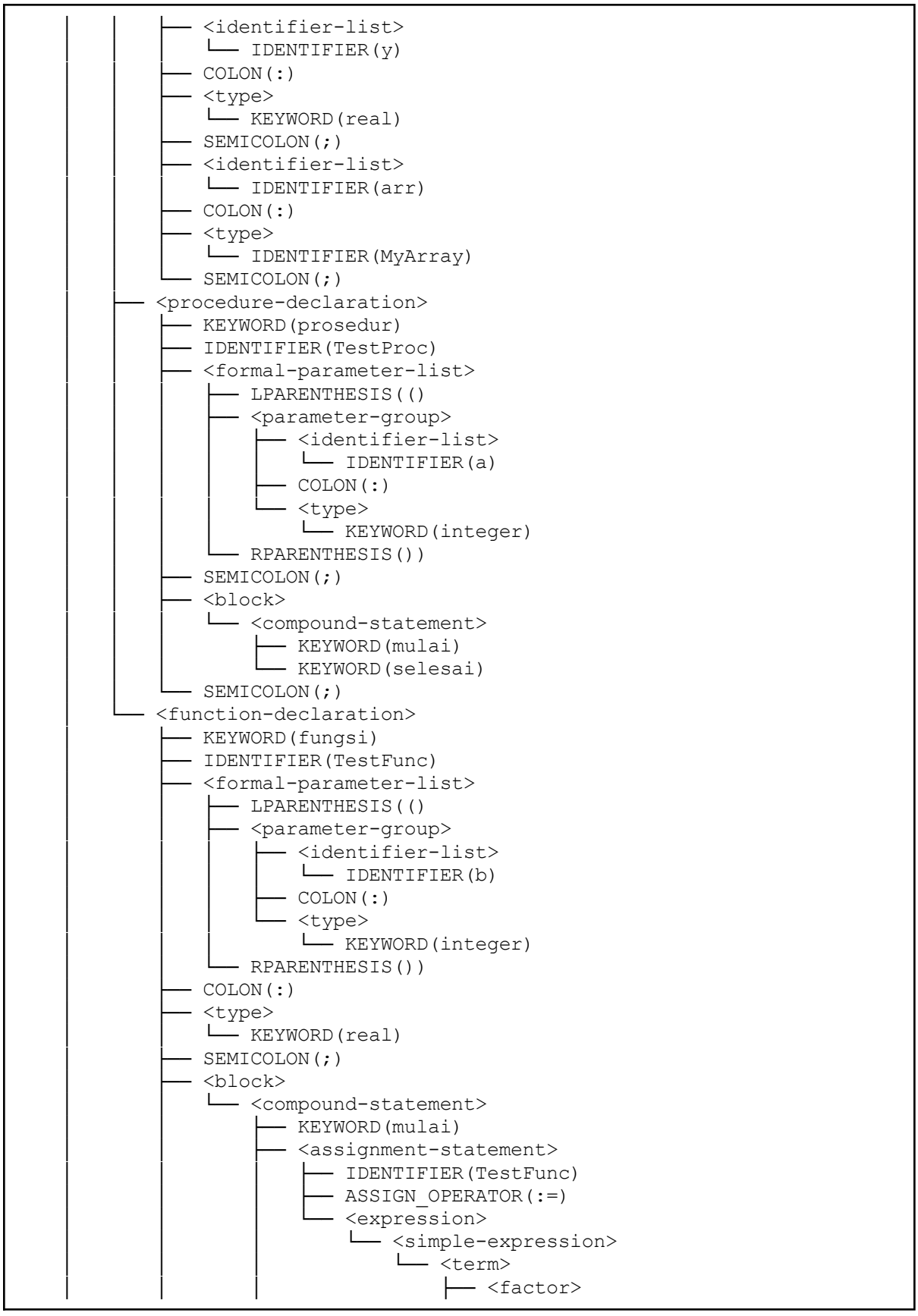
```
program AllDeclarations;
konstanta
   PI = 3.14;
tipe
   MyArray = larik [1..10] dari real;
variabel
   x: integer;
   y: real;
   arr: MyArray;

prosedur TestProc(a: integer);
mulai
   { prosedur ini tidak melakukan apa-apa }
selesai;

fungsi TestFunc(b: integer): real;
mulai
   TestFunc := b * PI;
selesai;

mulai
   { Blok utama }
   x := 10;
   y := TestFunc(x);
   TestProc(x);
selesai.
```

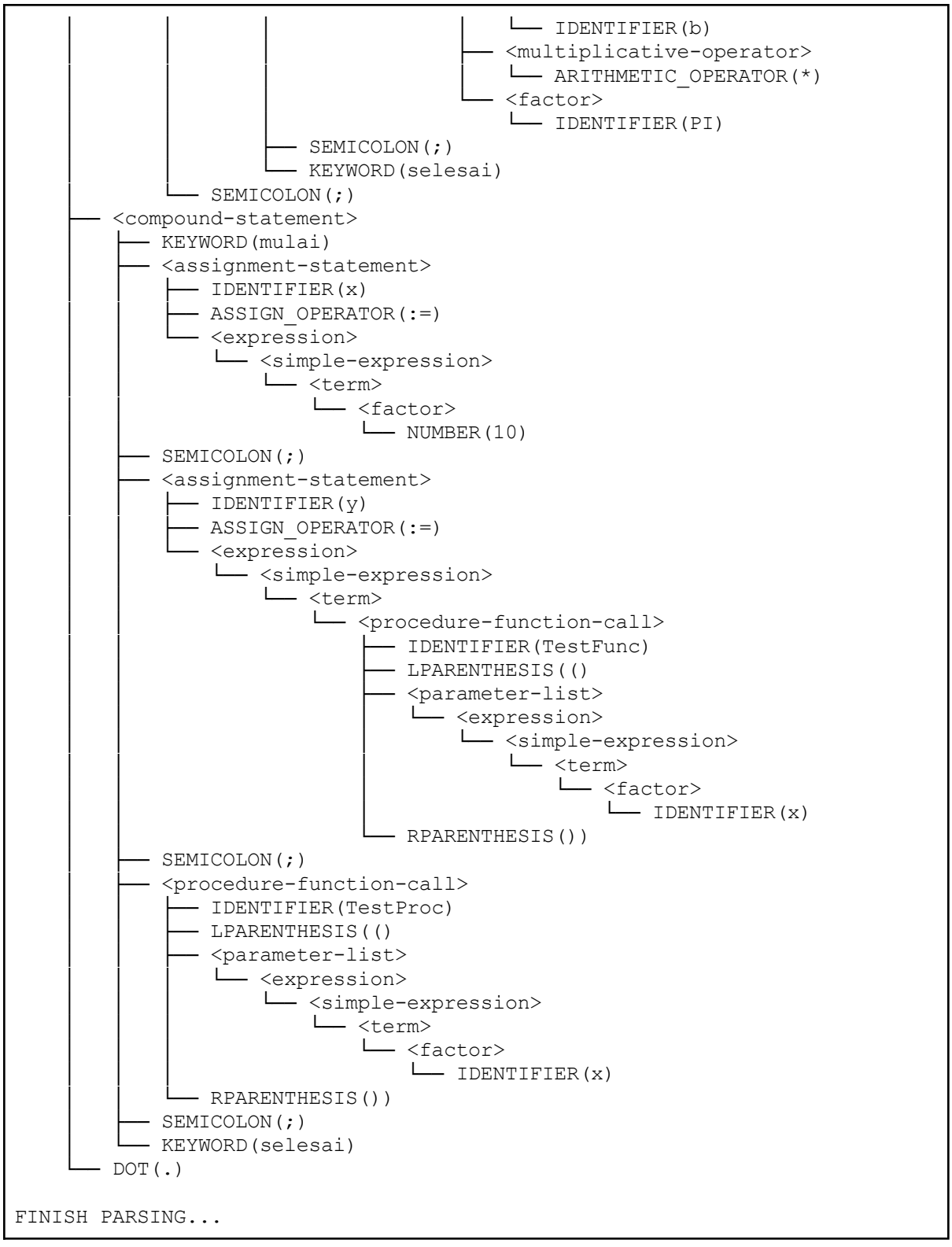
Output

```
KEYWORD(program)
IDENTIFIER(AllDeclarations)
SEMICOLON(;)
KEYWORD(konstanta)
IDENTIFIER(PI)
RELATIONAL_OPERATOR(=)
NUMBER(3.14)
SEMICOLON(;)
KEYWORD(tipe)
IDENTIFIER(MyArray)
RELATIONAL_OPERATOR(=)
KEYWORD(larik)
LBRACKET([)
NUMBER(1)
RANGE_OPERATOR(..)
NUMBER(10)
RBRACKET(])
KEYWORD(dari)
KEYWORD(real)
SEMICOLON(;)
KEYWORD(variabel)
IDENTIFIER(x)
```

```
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(y)
COLON(:)
KEYWORD(real)
SEMICOLON(;)
IDENTIFIER(arr)
COLON(:)
IDENTIFIER(MyArray)
SEMICOLON(;)
KEYWORD(prosedur)
IDENTIFIER(TestProc)
LPARENTHESIS(()
IDENTIFIER(a)
COLON(:)
KEYWORD(integer)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(mulai)
KEYWORD(selesai)
SEMICOLON(;)
KEYWORD(fungsi)
IDENTIFIER(TestFunc)
LPARENTHESIS(()
IDENTIFIER(b)
COLON(:)
KEYWORD(integer)
RPARENTHESIS())
COLON(:)
KEYWORD(real)
SEMICOLON(;)
KEYWORD(mulai)
IDENTIFIER(TestFunc)
ASSIGN_OPERATOR(:=)
IDENTIFIER(b)
ARITHMETIC_OPERATOR(*)
IDENTIFIER(PI)
SEMICOLON(;)
KEYWORD(selesai)
SEMICOLON(;)
KEYWORD(mulai)
IDENTIFIER(x)
ASSIGN_OPERATOR(:=)
NUMBER(10)
SEMICOLON(;)
IDENTIFIER(y)
ASSIGN_OPERATOR(:=)
IDENTIFIER(TestFunc)
LPARENTHESIS(()
IDENTIFIER(x)
RPARENTHESIS())
SEMICOLON(;)
IDENTIFIER(TestProc)
LPARENTHESIS(()
IDENTIFIER(x)
```

```
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(selesai)
DOT(.)

START PARSING...

└── <program>
    ├── <program-header>
    │   ├── KEYWORD(program)
    │   ├── IDENTIFIER(AllDeclarations)
    │   └── SEMICOLON(;)
    ├── <declaration-part>
    │   ├── <const-declaration>
    │   │   ├── KEYWORD(konstanta)
    │   │   ├── IDENTIFIER(PI)
    │   │   ├── RELATIONAL_OPERATOR(=)
    │   │   ├── <expression>
    │   │   │   └── <simple-expression>
    │   │   │       └── <term>
    │   │   │           └── <factor>
    │   │   │               └── NUMBER(3.14)
    │   │   └── SEMICOLON(;)
    │   ├── <type-declaration>
    │   │   ├── KEYWORD(tipe)
    │   │   ├── IDENTIFIER(MyArray)
    │   │   ├── RELATIONAL_OPERATOR(=)
    │   │   ├── <type>
    │   │   │   └── <array-type>
    │   │   │       ├── KEYWORD(larik)
    │   │   │       ├── LBRACKET([)
    │   │   │       ├── <range>
    │   │   │       │   ├── <expression>
    │   │   │       │   │   └── <simple-expression>
    │   │   │       │   │       └── <term>
    │   │   │       │   │           └── <factor>
    │   │   │       │   │               └── NUMBER(1)
    │   │   │       │   ├── RANGE_OPERATOR(..)
    │   │   │       │   └── <expression>
    │   │   │       │       └── <simple-expression>
    │   │   │       │           └── <term>
    │   │   │       │               └── <factor>
    │   │   │       │                   └── NUMBER(10)
    │   │   │       ├── RBRACKET(])
    │   │   │       ├── KEYWORD(dari)
    │   │   │       └── <type>
    │   │   │           └── KEYWORD(real)
    │   │   └── SEMICOLON(;)
    │   ├── <var-declaration>
    │   │   ├── KEYWORD(variabel)
    │   │   ├── <identifier-list>
    │   │   │   └── IDENTIFIER(x)
    │   │   ├── COLON(:)
    │   │   ├── <type>
    │   │   │   └── KEYWORD(integer)
    │   │   ├── SEMICOLON(;)
```


├── <identifier-list>
└── IDENTIFIER(y)
├── COLON(:)
├── <type>
└── KEYWORD(real)
├── SEMICOLON(;)
├── <identifier-list>
└── IDENTIFIER(arr)
├── COLON(:)
├── <type>
└── IDENTIFIER(MyArray)
└── SEMICOLON(;)
├── <procedure-declaration>
├── KEYWORD(prosedur)
├── IDENTIFIER(TestProc)
├── <formal-parameter-list>
├── LPARENTHESIS(()
├── <parameter-group>
├── <identifier-list>
└── IDENTIFIER(a)
├── COLON(:)
└── <type>
└── KEYWORD(integer)
└── RPARENTHESIS())
├── SEMICOLON(;)
├── <block>
└── <compound-statement>
├── KEYWORD(mulai)
└── KEYWORD(selesai)
└── SEMICOLON(;)
└── <function-declaration>
├── KEYWORD(fungsi)
├── IDENTIFIER(TestFunc)
├── <formal-parameter-list>
├── LPARENTHESIS(()
├── <parameter-group>
├── <identifier-list>
└── IDENTIFIER(b)
├── COLON(:)
└── <type>
└── KEYWORD(integer)
└── RPARENTHESIS())
├── COLON(:)
├── <type>
└── KEYWORD(real)
├── SEMICOLON(;)
├── <block>
└── <compound-statement>
├── KEYWORD(mulai)
├── <assignment-statement>
├── IDENTIFIER(TestFunc)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
└── <term>
├── <factor>


└── IDENTIFIER(b)
├── <multiplicative-operator>
└── ARITHMETIC_OPERATOR(*)
└── <factor>
└── IDENTIFIER(PI)
├── SEMICOLON(;)
└── KEYWORD(selesai)
└── SEMICOLON(;)
├── <compound-statement>
├── KEYWORD(mulai)
├── <assignment-statement>
├── IDENTIFIER(x)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(10)
├── SEMICOLON(;)
├── <assignment-statement>
├── IDENTIFIER(y)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
└── <term>
└── <procedure-function-call>
├── IDENTIFIER(TestFunc)
├── LPARENTHESIS(()
├── <parameter-list>
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(x)
└── RPARENTHESIS())
├── SEMICOLON(;)
├── <procedure-function-call>
├── IDENTIFIER(TestProc)
├── LPARENTHESIS(()
├── <parameter-list>
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(x)
└── RPARENTHESIS())
├── SEMICOLON(;)
└── KEYWORD(selesai)
└── DOT(.)
FINISH PARSING...

4.5. Pengujian 4

Tujuan pengujian: Untuk mendemonstrasikan kemampuar *parser* dalam menangani *error handling* ketika menemukan kesalahan sintaks (urutan token yang salah). Pengujian ini secara spesifik memvalidasi deteksi token yang hilang (misalnya RPARENTHESIS yang diharapkan) dan memastikan *parser* melaporkan pesan error yang informatif tanpa *crash*.

File: 4-errors.pas

Input

```
program ErrorTest;
mulai
   { Error: ')' hilang setelah 'Hello' }
   writeln('Hello';
selesai.
```

Output

```
ERROR:root:Syntax error: expected RPARENTHESIS()), but got SEMICOLON(;) @
4:19
ERROR:root:Syntax error: expected DOT(.), but got SEMICOLON(;) @ 4:19
KEYWORD(program)
IDENTIFIER(ErrorTest)
SEMICOLON(;)
KEYWORD(mulai)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Hello')
SEMICOLON(;)
KEYWORD(selesai)
DOT(.)

START PARSING...

└── <program>
    ├── <program-header>
    │   ├── KEYWORD(program)
    │   ├── IDENTIFIER(ErrorTest)
    │   └── SEMICOLON(;)
    └── <compound-statement>
        └── KEYWORD(mulai)

FINISH PARSING...
```

4.6. Pengujian 5

Tujuan pengujian: Untuk memvalidasi kemampuan *parser* dalam mem-parsing semua jenis *statement loop*, termasuk <while-statement> (selama…lakukan) dan <for-statement> (varian ke dan turun_ke). Pengujian ini juga secara spesifik menguji kemampuan *parser* dalam

menangani struktur bersarang (nesting), seperti <for-statement> di dalam <while-statement>, yang juga berisi blok <compound-statement> di dalamnya.

File: 5-loops.pas

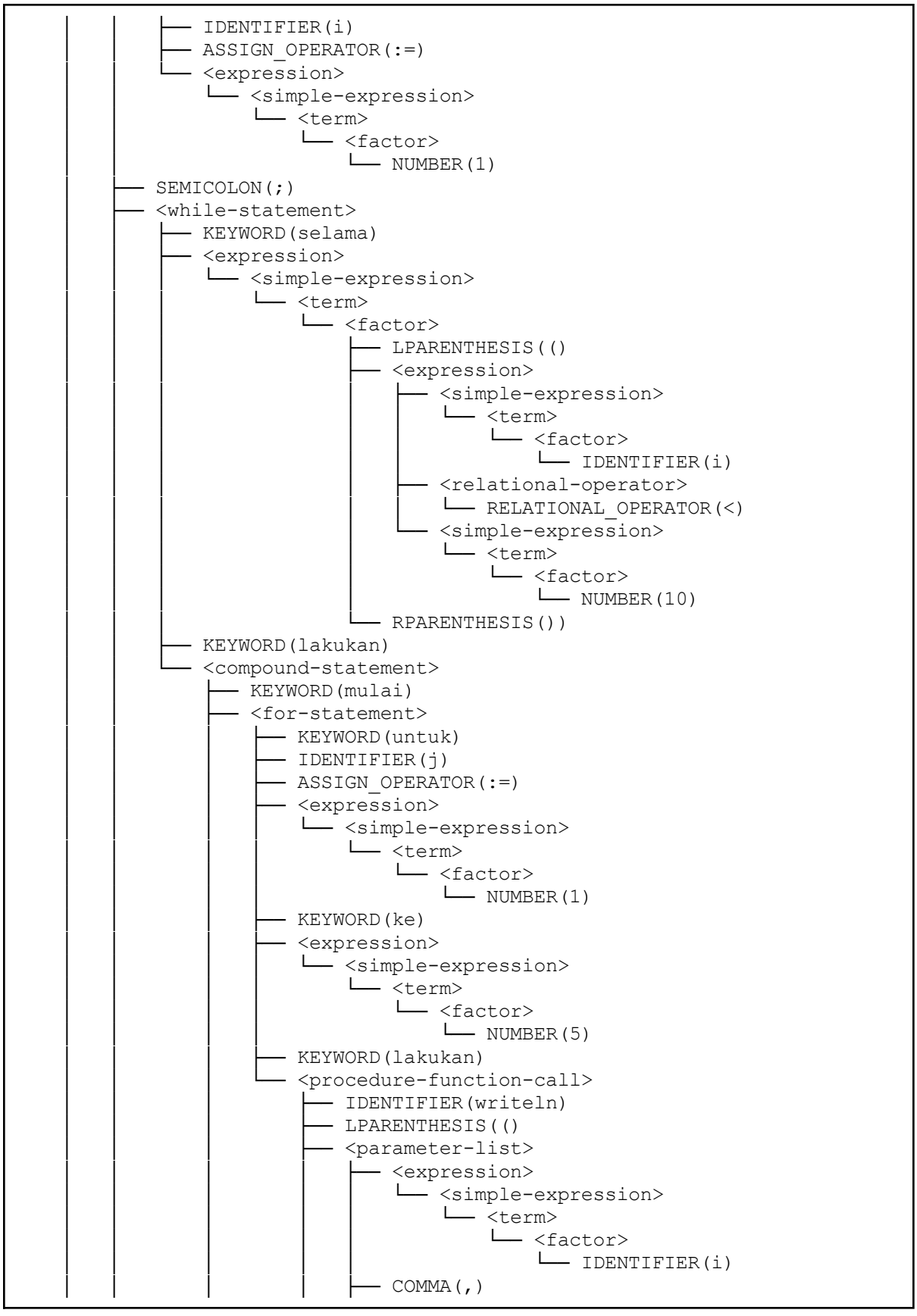
Input

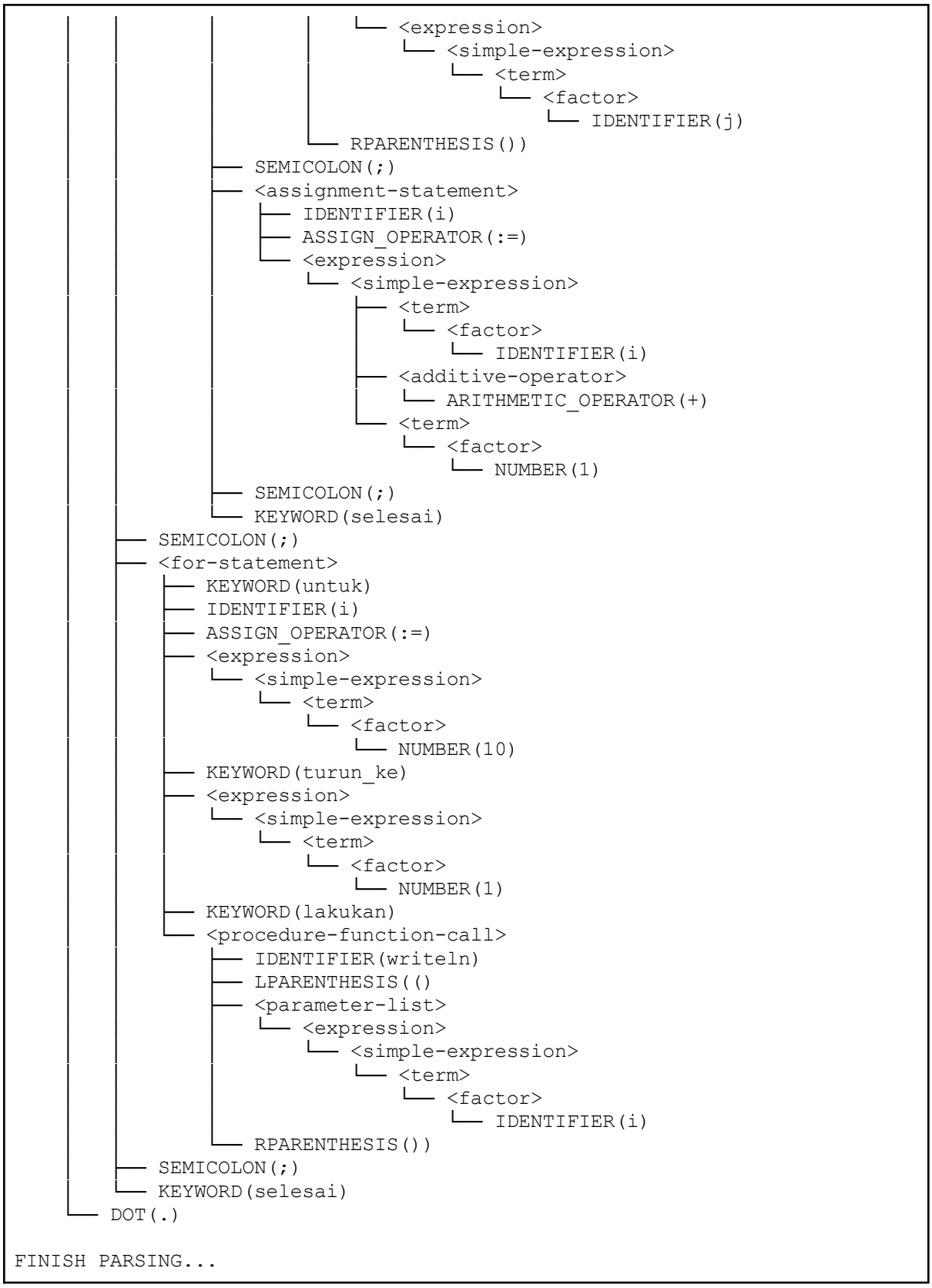
```
program TestLoops;
variabel
   i, j: integer;
mulai
   { Tes 1: <while-statement> }
   i := 1;
   selama (i < 10) lakukan
   mulai
      { Tes 2: <for-statement> (bersarang) }
      untuk j := 1 ke 5 lakukan
         writeln(i, j);

      i := i + 1;
   selesai;

   { Tes 3: <for-statement> dengan 'turun_ke' }
   untuk i := 10 turun_ke 1 lakukan
      writeln(i);
selesai.
```

Output

```
KEYWORD(program)
IDENTIFIER(TestLoops)
SEMICOLON(;)
KEYWORD(variabel)
IDENTIFIER(i)
COMMA(,)
IDENTIFIER(j)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(mulai)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
NUMBER(1)
SEMICOLON(;)
KEYWORD(selama)
LPARENTHESIS(()
IDENTIFIER(i)
RELATIONAL_OPERATOR(<)
NUMBER(10)
RPARENTHESIS())
KEYWORD(lakukan)
KEYWORD(mulai)
KEYWORD(untuk)
IDENTIFIER(j)
```

```
ASSIGN_OPERATOR(:=)
NUMBER(1)
KEYWORD(ke)
NUMBER(5)
KEYWORD(lakukan)
IDENTIFIER(writeln)
LPARENTHESIS(()
IDENTIFIER(i)
COMMA(,)
IDENTIFIER(j)
RPARENTHESIS())
SEMICOLON(;)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
IDENTIFIER(i)
ARITHMETIC_OPERATOR(+)
NUMBER(1)
SEMICOLON(;)
KEYWORD(selesai)
SEMICOLON(;)
KEYWORD(untuk)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
NUMBER(10)
KEYWORD(turun_ke)
NUMBER(1)
KEYWORD(lakukan)
IDENTIFIER(writeln)
LPARENTHESIS(()
IDENTIFIER(i)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(selesai)
DOT(.)

START PARSING...

└── <program>
    ├── <program-header>
    │   ├── KEYWORD(program)
    │   ├── IDENTIFIER(TestLoops)
    │   └── SEMICOLON(;)
    ├── <declaration-part>
    │   └── <var-declaration>
    │       ├── KEYWORD(variabel)
    │       ├── <identifier-list>
    │       │   ├── IDENTIFIER(i)
    │       │   ├── COMMA(,)
    │       │   └── IDENTIFIER(j)
    │       ├── COLON(:)
    │       ├── <type>
    │       │   └── KEYWORD(integer)
    │       └── SEMICOLON(;)
    ├── <compound-statement>
    │   ├── KEYWORD(mulai)
    │   ├── <assignment-statement>
```


IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
NUMBER(1)
SEMICOLON(;)
<while-statement>
KEYWORD(selama)
<expression>
<simple-expression>
<term>
<factor>
LPARENTHESIS(()
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(i)
<relational-operator>
RELATIONAL_OPERATOR(<)
<simple-expression>
<term>
<factor>
NUMBER(10)
RPARENTHESIS())
KEYWORD(lakukan)
<compound-statement>
KEYWORD(mulai)
<for-statement>
KEYWORD(untuk)
IDENTIFIER(j)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
NUMBER(1)
KEYWORD(ke)
<expression>
<simple-expression>
<term>
<factor>
NUMBER(5)
KEYWORD(lakukan)
<procedure-function-call>
IDENTIFIER(writeln)
LPARENTHESIS(()
<parameter-list>
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(i)
COMMA(,)


└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(j)
└── RPARENTHESIS())
├── SEMICOLON(;)
├── <assignment-statement>
├── IDENTIFIER(i)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
├── <term>
└── <factor>
└── IDENTIFIER(i)
├── <additive-operator>
└── ARITHMETIC_OPERATOR(+)
└── <term>
└── <factor>
└── NUMBER(1)
├── SEMICOLON(;)
└── KEYWORD(selesai)
├── SEMICOLON(;)
├── <for-statement>
├── KEYWORD(untuk)
├── IDENTIFIER(i)
├── ASSIGN_OPERATOR(:=)
├── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(10)
├── KEYWORD(turun_ke)
├── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(1)
├── KEYWORD(lakukan)
└── <procedure-function-call>
├── IDENTIFIER(writeln)
├── LPARENTHESIS(()
├── <parameter-list>
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(i)
└── RPARENTHESIS())
├── SEMICOLON(;)
└── KEYWORD(selesai)
└── DOT(.)
FINISH PARSING...

5. KESIMPULAN DAN SARAN

5.1. Kesimpulan

Implementasi *Syntax Analysis* untuk *milestone* 2 ini berhasil dengan *parser* yang menggunakan metode *Recursive Descent*. *Parser* ini secara akurat memvalidasi token dari *lexer* berdasarkan *grammar* PASCAL-S yang di-*hardcode*. Hasil pengujian membuktikan *parser* sukses membangun *Parse Tree* untuk kode yang valid dan melaporkan pesan error informatif untuk kode yang tidak valid.

5.2.Saran

Untuk pengerjaan selanjutnya, *Parse Tree* yang dihasilkan dapat dimanfaatkan sebagai masukan untuk tahap kompilasi berikutnya, yaitu *Semantic Analysis*. Untuk *parser*-nya sendiri dapat dikembangkan lebih lanjut untuk kemudahan pengembangan *Semantic Analysis*. Untuk kakak-kakak asisten lab IRK, mengenai QnA serta spesifikasi dari tugas besar, spesifikasi dapat direvisi apa yang telah dinyatakan dalam QnA, seperti *wording* yang salah serta mengenai *assignment*.

LAMPIRAN

- Tautan *repository* GitHub : https://github.com/anellautari/DFC-Tubes-IF2224

PEMBAGIAN TUGAS

NAMA	NIM	TUGAS	Persentase (%)
Mayla Yaffa Ludmilla	13523050	Implementasi statement & grammar utama parser, laporan dasar teori & implementasi.	25
Anella Utari Gunadi	13523078	Implementasi declaration & subprogram parser, laporan implementasi & deskripsi tugas	25
Muhammad Edo Raduputu Aprima	13523096	Definisi aturan grammar, implementasi assignment statement, procedure function, dan expression	25
Athian Nugraha Muarajuang	13523106	Implementasi error handling, parse term, factor, serta operators. Laporan kesimpulan dan saran	25

GRAMMAR YANG DIGUNAKAN

(Notasi EBNF)

```
<program> ::= <program-header> <block> DOT

<program-header> ::= KEYWORD(program) IDENTIFIER SEMICOLON

<block> ::= <declaration-part> <compound-statement>
```

```
<declaration-part> ::=
     { <const-declaration> }
     { <type-declaration> }
     { <var-declaration> }
     { <subprogram-declaration> }

<const-declaration> ::=
     KEYWORD(konstanta) (IDENTIFIER RELATIONAL_OPERATOR(=)
     <expression> SEMICOLON)+

<type-declaration> ::=
     KEYWORD(tipe) (IDENTIFIER RELATIONAL_OPERATOR(=) <type>
     SEMICOLON)+

<var-declaration> ::=
     KEYWORD(variabel) (<identifier-list> COLON <type> SEMICOLON)+

<identifier-list> ::= IDENTIFIER { COMMA IDENTIFIER }

<type> ::= <simple-type> | <array-type>

<simple-type> ::=
     KEYWORD(integer)
     | KEYWORD(real)
     | KEYWORD(boolean)
     | KEYWORD(char)

<array-type> ::=
     KEYWORD(larik) LBRACKET <range> RBRACKET KEYWORD(dari) <type>

<range> ::= <expression> RANGE_OPERATOR <expression>

<subprogram-declaration> ::=
     <procedure-declaration> | <function-declaration>

<procedure-declaration> ::=
     KEYWORD(prosedur) IDENTIFIER [ <formal-parameter-list> ]
     SEMICOLON <block> SEMICOLON

<function-declaration> ::=
     KEYWORD(fungsi) IDENTIFIER [ <formal-parameter-list> ] COLON
     <type> SEMICOLON <block> SEMICOLON

<formal-parameter-list> ::=
     LPARENTHESIS <parameter-group> { SEMICOLON <parameter-group> }
     RPARENTHESIS

<parameter-group> ::= <identifier-list> COLON <type>

<compound-statement> ::=
     KEYWORD(mulai) <statement-list> KEYWORD(selesai)
```

```
<statement-list> ::= <statement> { SEMICOLON <statement> }

<statement> ::=
     <assignment-statement>
     | <procedure-call>
     | <if-statement>
     | <while-statement>
     | <for-statement>
     | <compound-statement>
     | <empty-statement>

<empty-statement> ::= ε

<assignment-statement> ::=
     IDENTIFIER ASSIGN_OPERATOR <expression>

<procedure-call> ::=
     IDENTIFIER [ LPARENTHESIS <parameter-list> RPARENTHESIS ]

<parameter-list> ::= <expression> { COMMA <expression> }

<if-statement> ::=
     KEYWORD(jika) <expression> KEYWORD(maka) <statement> [
     KEYWORD(selain-itu) <statement> ]

<while-statement> ::=
     KEYWORD(selama) <expression> KEYWORD(lakukan) <statement>

<for-statement> ::=
     KEYWORD(untuk) IDENTIFIER ASSIGN_OPERATOR <expression>
     <direction> <expression> KEYWORD(lakukan) <statement>

<direction> ::= KEYWORD(ke) | KEYWORD(turun-ke)

<expression> ::=
     <simple-expression> [ <relational-operator>
     <simple-expression> ]

<simple-expression> ::=
     [ <sign> ] <term> { <additive-operator> <term> }

<term> ::=
     <factor> { <multiplicative-operator> <factor> }

<factor> ::=
     IDENTIFIER
     | NUMBER
     | CHAR_LITERAL
     | STRING_LITERAL
     | <function-call>
```

```
      | LPARENTHESIS <expression> RPARENTHESIS
      | LOGICAL_OPERATOR(tidak) <factor>

<function-call> ::=
      IDENTIFIER LPARENTHESIS [ <parameter-list> ] RPARENTHESIS

<sign> ::=
      ARITHMETIC_OPERATOR(+) | ARITHMETIC_OPERATOR(-)

<relational-operator> ::=
      RELATIONAL_OPERATOR(=)
      | RELATIONAL_OPERATOR(<>)
      | RELATIONAL_OPERATOR(<)
      | RELATIONAL_OPERATOR(<=)
      | RELATIONAL_OPERATOR(>)
      | RELATIONAL_OPERATOR(>=)

<additive-operator> ::=
      ARITHMETIC_OPERATOR(+)
      | ARITHMETIC_OPERATOR(-)
      | LOGICAL_OPERATOR(atau)

<multiplicative-operator> ::=
      ARITHMETIC_OPERATOR(*)
      | ARITHMETIC_OPERATOR(/)
      | ARITHMETIC_OPERATOR(bagi)
      | ARITHMETIC_OPERATOR(mod)
      | LOGICAL_OPERATOR(dan)
```

REFERENSI

[1] Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation. 3rd ed*, 2007. Tersedia: https://cdn-edunex.itb.ac.id/29161-Formal-Language-Theory-and-Automata/1629640939613_Introduction-to-Automata-Theory,-Languages,-and-Computation-Edition-3.pdf [Diakses: 11-Oktober-2025].

[2] Wirth, Niklaus. "PASCAL-S: A Subset and its implementation", Tersedia: http://pascal.hansotten.com/uploads/pascals/PASCAL-S%20A%20subset%20and%20its%20Implementation%20012.pdf

[3] tutorialspoint.com. *Compiler Design*. Tersedia: https://www.tutorialspoint.com/compiler_design/index.htm [Diakses 11-Oktober-2025]

[4] geeksforgeeks.com. *Introduction to Syntax Analysis in Compiler Design.* Tersedia: https://www.geeksforgeeks.org/compiler-design/introduction-to-syntax-analysis-in-compiler-design/ [Diakses 15 November 2025]

[5] geeksforgeeks.com. *Recursive Descent Parser.* Tersedia: https://www.geeksforgeeks.org/compiler-design/recursive-descent-parser/ [Diakses 16 November 2025]