

IF2224 - Teori Bahasa Formal dan Automata

Laporan Milestone 2 Tugas Besar 1

Syntax Analysis



Kelompok 10 - BetterCallPascal

Julius Arthur	13523030
Samuel Gerrard H. Girsang	13523064
Nadhif Al Rozin	13523076
Lutfi Hakim Yusra	13523084

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

2025

DAFTAR ISI

DAFTAR ISI.....	2
LANDASAN TEORI.....	3
1. Parser.....	3
2. Hubungan dengan Lexer.....	3
3. Recursive Descent.....	3
4. Parse Tree.....	4
Source:	
https://www.geeksforgeeks.org/compiler-design/parse-tree-and-syntax-tree/	4
PERANCANGAN DAN IMPLEMENTASI.....	5
1. Perancangan.....	5
2. Implementation.....	6
node.py.....	7
parser.py.....	8
statement_parser.py.....	10
declaration_parser.py.....	18
expression_parser.py.....	27
PENGUJIAN.....	34
1. test1.pas.....	34
2. test2.pas.....	53
3. test3.pas.....	58
4. test4.pas.....	64
5. test5.pas.....	68
KESIMPULAN DAN SARAN.....	75
LAMPIRAN.....	76
REFERENSI.....	77

LANDASAN TEORI

1. Parser

Dalam proses kompilasi, *parser* merupakan fase kedua, yang berfungsi untuk memeriksa aliran *token* yang dihasilkan oleh *lexer* dan memastikan bahwa urutan *token* tersebut sesuai dengan aturan *syntax* dari *source code*, atau pada kasus ini, bahasa PASCAL-S. Jika *lexer* bertugas memeriksa apakah kata-kata yang muncul valid dan logis menurut aturan bahasa, *parser* memeriksa apakah urutan kata-kata tersebut valid dan logis menurut aturan bahasa. Output dari fase kompilasi *parser* merupakan sebuah *parse tree*, yang akan digunakan untuk fase kompilasi berikutnya.

2. Hubungan dengan Lexer

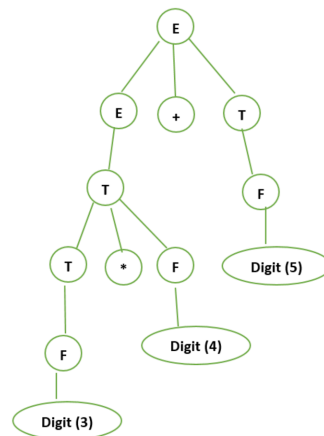
Dari sebuah *source code*, *lexer* mengubah keseluruhan tulisan jadi sebuah aliran *token-token* yang merepresentasikan keseluruhan teks. *Parser* kemudian melakukan traversal satu per satu *token* untuk memeriksa strukturnya. Tipe dan isi dari sebuah *token* dimanfaatkan oleh *parser* untuk membangun *parse tree*.

3. Recursive Descent

Dalam proses pembuatan *parse tree* yang merepresentasikan kode, *parser* melakukan algoritma yang umum, yaitu *recursive descent*. *Recursive descent* adalah metode pembangunan *parse tree* dari *source code* yang dimulai dari akar, dan simbol non-terminal dari bahasa direpresentasikan dalam bentuk sebuah fungsi yang akan dipanggil berulang kali hingga mencapai leaf, yang merupakan simbol terminal. Berikut cara kerja dari *recursive descent*:

1. Non-terminal dalam grammar bahasa, seperti `<program>`, diimplementasikan sebagai fungsi yang terpisah.
2. Setiap fungsi untuk setiap simbol non-terminal berfungsi mengenali struktur token-token yang sedang dilintasi *parser*.
3. Dalam sebuah fungsi, *parser* akan menentukan aturan produksi yang harus diikuti berdasarkan token-token yang dilewati.
4. Jika aturan produksi mengandung non-terminal lain, fungsi tersebut akan memanggil fungsi lain yang sesuai dengan non-terminal tersebut. Proses ini akan berulang terus hingga memproses keseluruhan dari aliran *token*.

4. Parse Tree



Source: <https://www.geeksforgeeks.org/compiler-design/parse-tree-and-syntax-tree/>

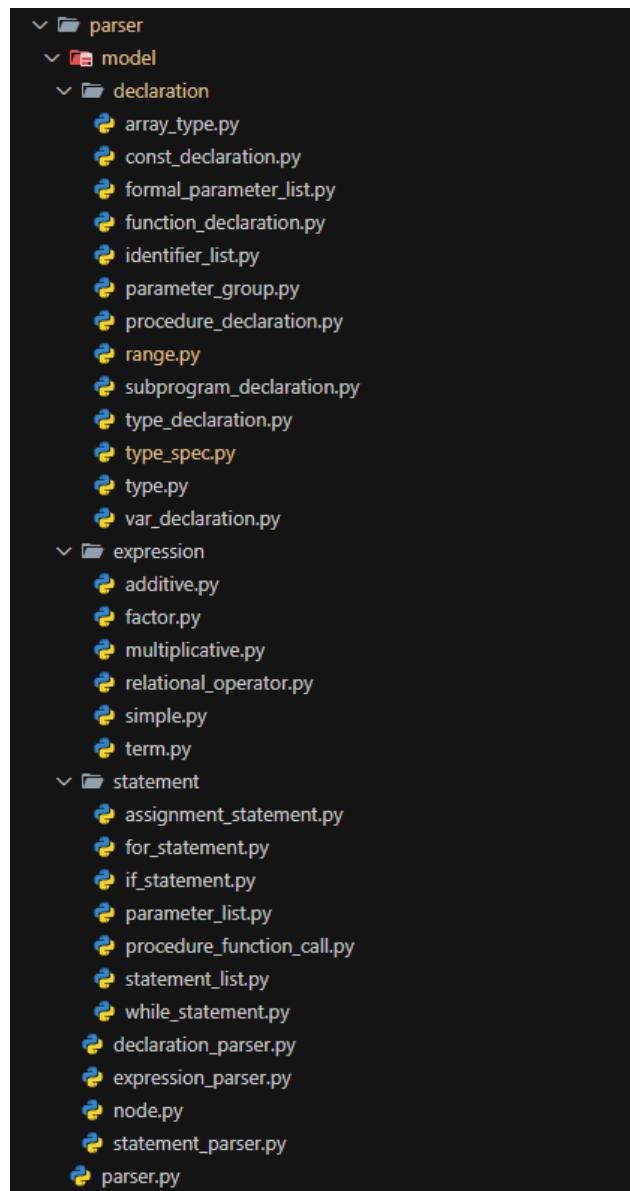
Parse Tree adalah representasi struktur yang menunjukkan bagaimana *parser* telah berhasil menerapkan aturan-aturan *grammar* untuk memvalidasi aliran token. Pada sebuah *parse tree*, *root node* merepresentasikan simbol awal dari *grammar*, *internal node* merepresentasikan simbol non-terminal, dan *leaf node* merepresentasikan token yang telah dibaca dari *source code*. Ini merupakan output dari fase kompilasi *parser*.

PERANCANGAN DAN IMPLEMENTASI

1. Perancangan

Program dibangun mengikuti aturan produksi yang tersedia di spek. Kelas *Parser* utama menahan kumpulan *token* yang diproduksi oleh *lexer*, dan dapat melakukan perlintasan dimulai dari *token* pertama. Kelas ini memiliki kelas-kelas derivasi yang bersifat lebih spesifik dan berguna untuk menjaga modularitas program. Kelas-kelas tersebut adalah *DeclarationParser*, yang berguna untuk memproses deklarasi, *ExpressionParser*, yang berguna untuk memproses ekspresi, dan *StatementParser*, yang berguna untuk memproses *statement*, atau bagian eksekusi program. *Parser* utama memiliki ketiga kelas derivasi tersebut, mengikuti pola desain *strategy*. Masing-masing kelas derivasi tersebut mewakili simbol non-terminal yang terkait, dan memiliki sebuah fungsi untuk setiap aturan produksinya.

Struktur program meliputi [main.py](#) dan dua folder yang terpisah untuk *lexer* dan *parser*. Folder untuk *lexer* mengandung implementasi automata dan pembaca *rule* dari *milestone* sebelumnya. Folder untuk *parser* mengandung [parser.py](#), mewakili kelas *Parser*. Terdapat juga *ParseNode* yang berperan mewakili simpul-simpul pada *parse tree*. Kelas-kelas derivasi dari *Parser* memiliki file sendiri, dalam bentuk *declaration_parser.py*, *statement_parser.py*, *expression_parser.py*. Setiap kelas derivasi dari *Parser* memiliki folder yang mengandung file-file berisikan fungsi yang merepresentasikan sebuah aturan produksi. Dalam penjelasan implementasi di bagian berikutnya, modularitas dari program digabung agar laporan ini mudah dibaca.



Alur dimulai oleh *Parser* yang memulai dari simbol awal pada *node* <program>, yang kemudian dibagi menjadi <program-header>, <declaration-part>, <compound-statement>. *Parser* mengolah langsung <program-header>, melakukan pengecekan awal terhadap aliran token, melihat inisialisasi program. *DeclarationParser* melanjutkan *parsing* dengan memeriksa tahap deklarasi tipe dan variabel dari program. Setelah itu, *StatementParser* memeriksa tahap eksekusi dari program, yang dibatasi oleh BEGIN dan END setelah deklarasi dilakukan. *ExpressionParser* bekerja di sela-sela keduanya untuk memproses semua ekspresi yang muncul. Setiap pemeriksaan aliran token oleh *parser* dilakukan dengan variabel *current_token* dan fungsi *advance()* yang melanjutkan aliran *token*.

2. Implementation

node.py

```
class ParseNode:
    def __init__(self, name):
        self.name = name
        self.children = []

    def add_child(self, child):
        if child:
            self.children.append(child)

    def to_string(self, prefix="", is_last=True):
        if prefix == "":
            result = self.name + "\n"
        else:
            connector = "└─ " if is_last else "├─ "
            result = prefix + connector + self.name + "\n"

        if prefix == "":
            new_prefix = "    "
        else:
            new_prefix = prefix + ("    " if is_last else "│ ")

        for i, child in enumerate(self.children):
            is_last_child = (i == len(self.children) - 1)
            result += child.to_string(new_prefix,
                                     is_last_child)

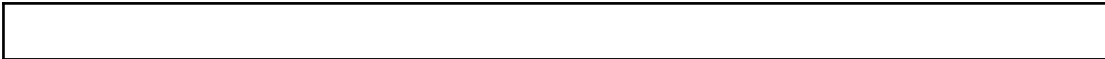
        return result

    def save_to_file(self, file_path):
        with open(file_path, 'w', encoding='utf-8') as file:
            file.write(self.to_string())

    def __str__(self):
        return self.to_string()
```

Penjelasan

ParseNode merupakan kelas yang berfungsi mewakili *parse tree* yang dihasilkan *parser*. ParseNode memiliki *children* yang terdiri dari ParseNode juga, dan memiliki metode untuk melakukan *save* hasil *parse tree*.



parser.py

```
from .model.declaration_parser import DeclarationParser
from .model.expression_parser import ExpressionParser
from .model.statement_parser import StatementParser
from .model.node import ParseNode

class Parser():
    def __init__(self, tokens):
        self.tokens = tokens
        self.position = 0

        self.declaration_parser = DeclarationParser(self)
        self.expression_parser = ExpressionParser(self)
        self.statement_parser = StatementParser(self)

    def current_token(self):
        if self.position < len(self.tokens):
            return self.tokens[self.position]
        return None

    def advance(self):
        self.position += 1

    def peek(self):
        peek_position = self.position + 1
        if peek_position < len(self.tokens):
            return self.tokens[peek_position]
        return None

    def check_token(self, expected_type,
expected_value=None):
        token = self.current_token()
        if token is None:
            raise SyntaxError("Unexpected end of input")

        token_type, token_value = token
        if token_type != expected_type:
            raise SyntaxError(f"Expected token type
```

```

{expected_type} (value: {expected_value}), got
{token_type}({token_value}) at position {self.position}")
    if expected_value and token_value != expected_value:
        raise SyntaxError(f"Expected token value
{expected_value}, got {token_value}")

    self.advance()
    return token

def parse_program_header(self):
    program_header_node = ParseNode("<program_header>")

    self.check_token("KEYWORD", "program")
    identifier = self.check_token("IDENTIFIER")
    self.check_token("SEMICOLON")

program_header_node.add_child(ParseNode(f"KEYWORD(program)"))

program_header_node.add_child(ParseNode(f"IDENTIFIER({identifier[1]})"))

program_header_node.add_child(ParseNode("SEMICOLON(;)"))

    return program_header_node

def parse_program(self):
    program_node = ParseNode("<program>")

    program_header_node = self.parse_program_header()
    program_node.add_child(program_header_node)

    declaration_part_node =
self.declaration_parser.parse_declarations()
    program_node.add_child(declaration_part_node)

    compound_statement_node =
self.statement_parser.parse_statement()
    program_node.add_child(compound_statement_node)

    self.check_token("DOT")
    program_node.add_child(ParseNode("DOT(.)"))

```

```
return program_node
```

Penjelasan

Parser merupakan kelas utama untuk melakukan *parsing*. Fungsi-fungsi utama yang digunakan untuk traversal aliran *token* dapat dilihat pada `current_token()`, `check_token()` dan `advance()`. Parser juga memiliki kelas-kelas derivasi yang berfungsi melakukan *parsing* untuk yang lebih spesifik. Parser melakukan inisialisasi pemeriksaan dengan `<program>` dan juga pemeriksaan awal pada header `<program-header>`. Setelah itu, dilanjutkan oleh kelas-kelas derivasinya.

statement_parser.py

```
from .node import ParseNode
from .statement.while_statement import parse_while_statement
from .statement.for_statement import parse_for_statement
from .statement.assignment_statement import
parse_assignment_statement
from .statement.if_statement import parse_if_statement
from .statement.statement_list import parse_statement_list
from .statement.procedure_function_call import
parse_procedure_function_call
from .statement.parameter_list import parse_parameter_list

class StatementParser():
    def __init__(self, parent):
        self.parent = parent

    def parse_statement(self):
        """Parse compound statement (mulai ... selesai)"""
        compound_statement_node =
ParseNode("<compound-statement>")

        self.parent.check_token("KEYWORD", "mulai")

        compound_statement_node.add_child(ParseNode("KEYWORD(mulai)")
)

        statement_list_node = self.parse_statement_list()

        compound_statement_node.add_child(statement_list_node)
```

```

        self.parent.check_token("KEYWORD", "selesai")

compound_statement_node.add_child(ParseNode("KEYWORD(selesai)"))

    return compound_statement_node

def parse_single_statement(self):
    """Parse any single statement (assignment, if, while,
    for, call, or compound)"""
    token = self.parent.current_token()
    if not token:
        raise SyntaxError("Expected statement, got end of
input")

    token_type, token_value = token

    # Compound statement
    if token_type == "KEYWORD" and token_value ==
"mulai":
        return self.parse_statement()

    # If statement
    if token_type == "KEYWORD" and token_value == "jika":
        return self.parse_if_statement()

    # While statement
    if token_type == "KEYWORD" and token_value ==
"selama":
        return self.parse_while_statement()

    # For statement
    if token_type == "KEYWORD" and token_value ==
"untuk":
        return self.parse_for_statement()

    # Identifier (assignment or call)
    if token_type == "IDENTIFIER":
        next_token = self.parent.peek()
        if next_token and next_token[0] == "LPAREN":
            return self.parse_function_call()

```

```

        return self.parse_assignment_statement()

    raise SyntaxError(f"Unexpected token in statement:
{token}")

    def parse_statement_list(self):
        statement_list_node = ParseNode("<statement-list>")

        first_statement =
parser.statement_parser.parse_single_statement()
        statement_list_node.add_child(first_statement)

        while parser.current_token() and
parser.current_token()[0] == "SEMICOLON":
            parser.check_token("SEMICOLON")

statement_list_node.add_child(ParseNode("SEMICOLON(;)))

        # Check if next token is 'selesai' (end of
compound statement)
        token = parser.current_token()
        if token and token[0] == "KEYWORD" and token[1]
== "selesai":
            break

        next_statement =
parser.statement_parser.parse_single_statement()
        statement_list_node.add_child(next_statement)

    return statement_list_node

    def parse_assignment_statement(self):
        assignment_node = ParseNode("<assignment-statement>")

        identifier = parser.check_token("IDENTIFIER")

assignment_node.add_child(ParseNode(f"IDENTIFIER({identifier[
1]})"))

        assignment_operator =
parser.check_token("ASSIGN_OPERATOR", ":=")

```

```

assignment_node.add_child(ParseNode(f"ASSIGN_OPERATOR({assignment_operator[1]})"))

        expression_node =
parser.expression_parser.parse_expression()
        assignment_node.add_child(expression_node)

    return assignment_node

def parse_if_statement(self):
    if_statement_node = ParseNode("<if-statement>")

    parser.check_token("KEYWORD", "jika")

    if_statement_node.add_child(ParseNode("KEYWORD(jika)"))

        expression_node =
parser.expression_parser.parse_expression()
        if_statement_node.add_child(expression_node)

    parser.check_token("KEYWORD", "maka")

    if_statement_node.add_child(ParseNode("KEYWORD(maka)"))

        statement_node =
parser.statement_parser.parse_single_statement()
        if_statement_node.add_child(statement_node)

    token = parser.current_token()
    if token and token[0] == "KEYWORD" and token[1] ==
"selain_itu":
        parser.check_token("KEYWORD", "selain_itu")

    if_statement_node.add_child(ParseNode("KEYWORD(selain_itu)"))

        else_statement_node =
parser.statement_parser.parse_single_statement()
        if_statement_node.add_child(else_statement_node)

    return if_statement_node

def parse_while_statement(self):

```

```

        while_node = ParseNode("<while_statement>")

        token = parser.current_token()
        if token and token[0] == "KEYWORD" and token[1] ==
"selama":
            selama_node = ParseNode(f"KEYWORD({token[1]})")
            while_node.add_child(selama_node)
            parser.advance()
        else:
            raise SyntaxError(f"Expected 'selama' keyword,
got {token}")

        expression_node =
parser.expression_parser.parse_expression()
        while_node.add_child(expression_node)

        token = parser.current_token()
        if token and token[0] == "KEYWORD" and token[1] ==
"lakukan":
            lakukan_node = ParseNode(f"KEYWORD({token[1]})")
            while_node.add_child(lakukan_node)
            parser.advance()
        else:
            raise SyntaxError(f"Expected 'lakukan' keyword,
got {token}")

        statement_node =
parser.statement_parser.parse_single_statement()
        while_node.add_child(statement_node)

        return while_node

    def parse_for_statement(self):
        for_node = ParseNode("<for_statement>")

        token = parser.current_token()
        if token and token[0] == "KEYWORD" and token[1] ==
"untuk":
            untuk_node = ParseNode(f"KEYWORD({token[1]})")
            for_node.add_child(untuk_node)
            parser.advance()
        else:

```

```

        raise SyntaxError(f"Expected 'untuk' keyword, got
{token}")

    token = parser.current_token()
    if token and token[0] == "IDENTIFIER":
        identifier_node =
ParseNode(f"IDENTIFIER({token[1]})")
        for_node.add_child(identifier_node)
        parser.advance()
    else:
        raise SyntaxError(f"Expected identifier in for
statement, got {token}")

    token = parser.current_token()
    if token and token[0] == "ASSIGN_OPERATOR" and
token[1] == ":=":
        assign_node =
ParseNode(f"ASSIGN_OPERATOR({token[1]})")
        for_node.add_child(assign_node)
        parser.advance()
    else:
        raise SyntaxError(f"Expected ':= ' in for
statement, got {token}")

    start_expr =
parser.expression_parser.parse_expression()
    for_node.add_child(start_expr)

    token = parser.current_token()
    if token and token[0] == "KEYWORD" and token[1] in
["ke", "turun-ke"]:
        direction_node =
ParseNode(f"KEYWORD({token[1]})")
        for_node.add_child(direction_node)
        parser.advance()
    else:
        raise SyntaxError(f"Expected 'ke' or 'turun-ke'
in for statement, got {token}")

    end_expr =
parser.expression_parser.parse_expression()
    for_node.add_child(end_expr)

```

```

        token = parser.current_token()
        if token and token[0] == "KEYWORD" and token[1] ==
"lakukan":
            lakukan_node = ParseNode(f"KEYWORD({token[1]})")
            for_node.add_child(lakukan_node)
            parser.advance()
        else:
            raise SyntaxError(f"Expected 'lakukan' keyword in
for statement, got {token}")

        statement_node =
parser.statement_parser.parse_single_statement()
        for_node.add_child(statement_node)

    return for_node

def parse_function_call(self):
    call_node = ParseNode('<procedure/function-call>')

    # Parse identifier (procedure/function name)
    token = parser.current_token()
    if token and token[0] == 'IDENTIFIER':
        identifier_node =
ParseNode(f'IDENTIFIER({token[1]})')
        call_node.add_child(identifier_node)
        parser.advance()
    else:
        raise SyntaxError(f"Expected identifier in
procedure/function call, got {token}")

    # Parse LPAREN
    token = parser.current_token()
    if token and token[0] == 'LPAREN':
        lparen_node = ParseNode('LPAREN()')
        call_node.add_child(lparen_node)
        parser.advance()
    else:
        raise SyntaxError(f"Expected '(' in
procedure/function call, got {token}")

    # Parse parameter list (actual parameters)

```

```

        parameter_list_node =
parser.statement_parser.parse_parameter_list()
        call_node.add_child(parameter_list_node)

        # Parse RPAREN
        token = parser.current_token()
        if token and token[0] == 'RPAREN':
            rparen_node = ParseNode('RPAREN()')
            call_node.add_child(rparen_node)
            parser.advance()
        else:
            raise SyntaxError(f"Expected ')' in
procedure/function call, got {token}")

        return call_node

def parse_parameter_list(self):
    param_list_node = ParseNode('<parameter_list>')

    expression_node =
parser.expression_parser.parse_expression()
    param_list_node.add_child(expression_node)

    token = parser.current_token()

    while token[0] == 'COMMA':
        param_list_node.add_child(ParseNode('COMMA(),'))

        parser.advance()
        token = parser.current_token()

        expression_node =
parser.expression_parser.parse_expression()
        param_list_node.add_child(expression_node)

        parser.advance()
        token = parser.current_token()

    return param_list_node

```

Penjelasan

StatementParser merupakan kelas yang berfungsi untuk melakukan *parsing*

terhadap <statement>, yang merupakan bagian eksekusi program yang terjadi setelah deklarasi, dan dibatasi oleh BEGIN dan END. Setiap fungsi yang terdapat di kelas ini menjelaskan sebuah aturan produksi yang terdapat pada spek. Fungsi utama yang digunakan oleh StatementParser untuk memulai *parsing* terhadap keseluruhan tahap eksekusi merupakan fungsi `parse_statement()`

declaration_parser.py

```
from .node import ParseNode
from .declaration.formal_parameter_list import
parse_formal_parameter_list
from .declaration.parameter_group import parse_parameter_group
from .declaration.function_declaration import
parse_function_declaration
from .declaration.procedure_declaration import
parse_procedure_declaration
from .declaration.const_declaration import parse_const_declaration
from .declaration.type_declaration import parse_type_declaration
from .declaration.var_declaration import parse_var_declaration
from .declaration.identifier_list import parse_identifier_list
from .declaration.type import parse_type
from .declaration.array_type import parse_array_type
from .declaration.range import parse_range
from .declaration.subprogram_declaration import
parse_subprogram_declaration

class DeclarationParser():
    def __init__(self, parent):
        self.parent = parent

    def parse_declarations(self):
        declaration_part_node = ParseNode("<declaration-part>")
        while True:
            current_token = self.parent.current_token()
            if current_token is None:
                break

            if current_token[0] == "KEYWORD":
                if current_token[1] == "konstanta":
                    const_declaration_node =
self.parse_const_declaration()

declaration_part_node.add_child(const_declaration_node)
                elif current_token[1] == "tipe":
                    type_declaration_node =
self.parse_type_declaration()
```

```

declaration_part_node.add_child(type_declaration_node)
        elif current_token[1] == "variabel":
            var_declaration_node =
self.parse_var_declaration()

declaration_part_node.add_child(var_declaration_node)
        elif current_token[1] in ["funksi", "prosedur"]:
            subprogram_declaration_node =
self.parse_subprogram_declaration()

declaration_part_node.add_child(subprogram_declaration_node)
        else:
            break
    else:
        break

    return declaration_part_node

def parse_const_declaration(self):
    const_decl_node = ParseNode("<const_declaration>")

    token = parser.current_token()
    if token and token[0] == "KEYWORD" and token[1] ==
"konstanta":
        const_keyword_node = ParseNode(f"KEYWORD({token[1]})")
        const_decl_node.add_child(const_keyword_node)
        parser.advance()

        while True:
            token = parser.current_token()
            if not token or token[0] != "IDENTIFIER":
                break

            identifier_node =
ParseNode(f"IDENTIFIER({token[1]})")
            const_decl_node.add_child(identifier_node)
            parser.advance()

            token = parser.current_token()
            if token and token[0] == "RELATIONAL_OPERATOR" and
token[1] == "=":
                equal_node =
ParseNode(f"RELATIONAL_OPERATOR({token[1]})")
                const_decl_node.add_child(equal_node)
                parser.advance()

                expression_node =

```

```

parser.expression_parser.parse_expression()
    const_decl_node.add_child(expression_node)

    token = parser.current_token()
    if token and token[0] == "SEMICOLON":
        semicolon_node =
ParseNode(f"SEMICOLON({token[1]})")
        const_decl_node.add_child(semicolon_node)
        parser.advance()
    else:
        raise SyntaxError(f"Expected ';' after
constant declaration, got {token}")
    else:
        raise SyntaxError(f"Expected '=' in constant
declaration, got {token}")

    return const_decl_node
else:
    raise SyntaxError(f"Expected 'konstanta' keyword, got
{token}")

def parse_type_declaration(self):
    type_decl_node = ParseNode("<type_declaration>")

    token = parser.current_token()
    if token and token[0] == "KEYWORD" and token[1] == "tipe":
        type_keyword_node = ParseNode(f"KEYWORD({token[1]})")
        type_decl_node.add_child(type_keyword_node)
        parser.advance()

    # Loop untuk parse multiple type declarations
    while True:
        token = parser.current_token()
        if not token or token[0] != "IDENTIFIER":
            break

        identifier_node = ParseNode(f"IDENTIFIER({token[1]})")
        type_decl_node.add_child(identifier_node)
        parser.advance()

        # Check for '='
        token = parser.current_token()
        if token and token[0] == "RELATIONAL_OPERATOR" and
token[1] == "=":
            equal_node =
ParseNode(f"RELATIONAL_OPERATOR({token[1]})")
            type_decl_node.add_child(equal_node)
            parser.advance()

```

```

# Parse type specification (definisi tipe
sebenarnya)

type_spec_node = parse_type_spec(parser)
type_decl_node.add_child(type_spec_node)

# Check for semicolon
token = parser.current_token()
if token and token[0] == "SEMICOLON":
    semicolon_node =
ParseNode(f"SEMICOLON({token[1]})")
    type_decl_node.add_child(semicolon_node)
    parser.advance()
else:
    raise SyntaxError(f"Expected ';' after type
declaration, got {token}")
else:
    raise SyntaxError(f"Expected '=' in type
declaration, got {token}")

    return type_decl_node
else:
    raise SyntaxError(f"Expected 'tipe' keyword, got {token}")

def parse_var_declaration(self):
    var_decl_node = ParseNode("<var_declaration>")

    token = parser.current_token()
    if token and token[0] == "KEYWORD" and token[1] ==
"variabel":
        var_keyword_node = ParseNode(f"KEYWORD({token[1]})")
        var_decl_node.add_child(var_keyword_node)
        parser.advance()

# Loop untuk parse multiple variable declarations
while True:
    token = parser.current_token()
    if not token or token[0] != "IDENTIFIER":
        break

    identifier_list_node =
parser.declaration_parser.parse_identifier_list()
    var_decl_node.add_child(identifier_list_node)

    token = parser.current_token()
    if token and token[0] == "COLON":
        colon_node = ParseNode(f"COLON({token[1]})")
        var_decl_node.add_child(colon_node)

```

```

        parser.advance()

        type_node =
parser.declaration_parser.parse_type()
        var_decl_node.add_child(type_node)

        token = parser.current_token()
        if token and token[0] == "SEMICOLON":
            semicolon_node =
ParseNode(f"SEMICOLON({token[1]})")
            var_decl_node.add_child(semicolon_node)
            parser.advance()
        else:
            raise SyntaxError(f"Expected ';' after
variable declaration, got {token}")
        else:
            raise SyntaxError(f"Expected ':' in variable
declaration, got {token}")

    return var_decl_node
else:
    raise SyntaxError(f"Expected 'variabel' keyword, got
{token}")

def parse_identifier_list(self):
    identifier_list_node = ParseNode("<identifier_list>")

    token = parser.current_token()
    if token and token[0] == "IDENTIFIER":
        identifier_node = ParseNode(f"IDENTIFIER({token[1]})")
        identifier_list_node.add_child(identifier_node)
        parser.advance()
    else:
        raise SyntaxError(f"Expected identifier, got {token}")

    while parser.current_token():
        token = parser.current_token()

        if token and token[0] == "COMMA":
            comma_node = ParseNode(f"COMMA({token[1]})")
            identifier_list_node.add_child(comma_node)
            parser.advance()

        token = parser.current_token()
        if token and token[0] == "IDENTIFIER":
            identifier_node =
ParseNode(f"IDENTIFIER({token[1]})")
            identifier_list_node.add_child(identifier_node)

```

```

        parser.advance()
    else:
        raise SyntaxError(f"Expected identifier after
',', got {token}")
    else:
        break

    return identifier_list_node

def parse_type(self):
    type_node = ParseNode('<type>')

    token = parser.current_token()

    if token[0] == 'KEYWORD' and token[1] in ['integer', 'real',
'boolean', 'char']:
        child_node = ParseNode(f'{{token[0]}}({token[1]})')

        type_node.add_child(child_node)
        parser.advance()
        return type_node

def parse_array_type(self):
    array_type_node = ParseNode('<array_type>')

    token = parser.current_token()

    if token[0] == 'KEYWORD' and token[1] == 'larik':
        array_type_node.add_child(ParseNode('KEYWORD(larik)'))
        parser.advance()
        token = parser.current_token()
    if token[0] == 'LBRACKET' and token[1] == '[':
        array_type_node.add_child(ParseNode('LBRACKET([)')')
        parser.advance()
        token = parser.current_token()

array_type_node.add_child(parser.declaration_parser.parse_range())
token = parser.current_token()

if token[0] == 'RBRACKET' and token[1] == ']':
    array_type_node.add_child(ParseNode('RBRACKET(])')')
    parser.advance()
    token = parser.current_token()
if token[0] == 'KEYWORD' and token[1] == 'dari':
    array_type_node.add_child(ParseNode('KEYWORD(dari)')')
    parser.advance()
    token = parser.current_token()

```

```

array_type_node.add_child(parser.declaration_parser.parse_type())

    return array_type_node

def parse_range(self):
    range_node = ParseNode('<range>')

    expression_node = parser.expression_parser.parse_expression()

    range_node.add_child(expression_node)

    token = parser.current_token()

    if token[0] == 'RANGE_OPERATOR' and token[1] == '..':
        range_node.add_child(ParseNode('RANGE_OPERATOR(..)'))
        parser.advance()

    expression_node = parser.expression_parser.parse_expression()

    range_node.add_child(expression_node)

    return range_node

def parse_subprogram_declaration(self):
    subprogram_declaration_node =
ParseNode("<subprogram-declaration>")

    token = parser.current_token()

    if token[0] == 'KEYWORD' and token[1] == 'fungsi':
        function_declaration_node =
parser.declaration_parser.parse_function_declaration()

subprogram_declaration_node.add_child(function_declaration_node)
        return subprogram_declaration_node

        elif token[0] == 'KEYWORD' and token[1] == 'prosedur':
            procedure_declaration_node =
parser.declaration_parser.parse_procedure_declaration()

subprogram_declaration_node.add_child(procedure_declaration_node)
            return subprogram_declaration_node

        else:
            raise SyntaxError("Expected 'fungsi' or 'prosedur'
keyword")

```

```

def parse_function_declaration(self):
    function_declaration_node =
ParseNode("<function-declaration>")

    parser.check_token("KEYWORD", "funksi")

function_declaration_node.add_child(ParseNode("KEYWORD(funksi)"))

    identifier = parser.check_token("IDENTIFIER")

function_declaration_node.add_child(ParseNode(f"IDENTIFIER({identifier[1]})"))

    current_token = parser.current_token()
    if current_token and current_token[0] == "LPAREN":
        formal_parameter_node =
parser.declaration_parser.parse_formal_parameter_list()

function_declaration_node.add_child(formal_parameter_node)

    parser.check_token("COLON")
function_declaration_node.add_child(ParseNode("COLON(:)"))

    type_node = parser.declaration_parser.parse_type()
function_declaration_node.add_child(type_node)

    parser.check_token("SEMICOLON")

function_declaration_node.add_child(ParseNode("SEMICOLON(;)"))

    declaration_node =
parser.declaration_parser.parse_declarations()
function_declaration_node.add_child(declaration_node)

    statement_node = parser.statement_parser.parse_statement()
function_declaration_node.add_child(statement_node)

    parser.check_token("SEMICOLON")

function_declaration_node.add_child(ParseNode("SEMICOLON(;)"))

    return function_declaration_node

def parse_procedure_declaration(self):
    procedure_declaration_node =
ParseNode("<procedure-declaration>")

```

```

        parser.check_token("KEYWORD", "prosedur")

    procedure_declaration_node.add_child(ParseNode("KEYWORD(prosedur)"))

    identifier = parser.check_token("IDENTIFIER")

    procedure_declaration_node.add_child(ParseNode(f"IDENTIFIER({identifier[1]})"))

    current_token = parser.current_token()
    if current_token and current_token[0] == "LPAREN":
        formal_parameter_node =
    parser.declaration_parser.parse_formal_parameter_list()

    procedure_declaration_node.add_child(formal_parameter_node)

    parser.check_token("SEMICOLON")

    procedure_declaration_node.add_child(ParseNode("SEMICOLON(;)"))

    declaration_node =
    parser.declaration_parser.parse_declarations()
    procedure_declaration_node.add_child(declaration_node)

    statement_node = parser.statement_parser.parse_statement()
    procedure_declaration_node.add_child(statement_node)

    parser.check_token("SEMICOLON")

    procedure_declaration_node.add_child(ParseNode("SEMICOLON(;)"))

    return procedure_declaration_node

def parse_formal_parameter_list(self):
    formal_parameter_list_node =
    ParseNode("<formal-parameter-list>")

    parser.check_token("LPAREN")
    formal_parameter_list_node.add_child(ParseNode("LPAREN(())"))

    parameter_group_node =
    parser.declaration_parser.parse_parameter_group()
    formal_parameter_list_node.add_child(parameter_group_node)

    while parser.current_token() and parser.current_token()[0] ==
    "SEMICOLON":
        parser.check_token("SEMICOLON")

```

```

formal_parameter_list_node.add_child(ParseNode("SEMICOLON(;)"))

        parameter_group_node =
parser.declaration_parser.parse_parameter_group()

formal_parameter_list_node.add_child(parameter_group_node)

        parser.check_token("RPAREN")
        formal_parameter_list_node.add_child(ParseNode("RPAREN(())"))

    return formal_parameter_list_node

def parse_parameter_group(self):
    parameter_group_node = ParseNode("<parameter-group>")

    identifier_list_node =
parser.declaration_parser.parse_identifier_list()
    parameter_group_node.add_child(identifier_list_node)

    parser.check_token("COLON", ":")
    parameter_group_node.add_child(ParseNode("COLON(:)"))

    type_node = parser.declaration_parser.parse_type()
    parameter_group_node.add_child(type_node)

    return parameter_group_node

```

Penjelasan

DeclarationParser merupakan kelas yang berfungsi untuk melakukan *parsing* terhadap <declaration-part>, yang merupakan bagian deklarasi program yang terjadi sebelum eksekusi, untuk melakukan inisialisasi variabel dan tipe. Setiap fungsi yang terdapat di kelas ini menjelaskan sebuah aturan produksi yang terdapat pada spek. Fungsi utama yang digunakan oleh DeclarationParser untuk memulai *parsing* terhadap keseluruhan tahap deklarasi merupakan fungsi `parse_declarations()`.

expression_parser.py

```

from .node import ParseNode
from .expression.simple import parse_simple_expression
from .expression.term import parse_term
from .expression.factor import parse_factor
from .expression.relational_operator import

```

```

parse_relational_operator
from .expression.additive import parse_additive_expression
from .expression.multiplicative import
parse_multiplicative_expression

class ExpressionParser():
    def __init__(self, parent):
        self.parent = parent

    def parse_expression(self):
        expression_node = ParseNode("<expression>")
        simple_expression_node = self.parse_simple_expression()
        expression_node.add_child(simple_expression_node)

        token = self.parent.current_token()

        if token is not None:
            token_type, token_value = token
            if token_type == "RELATIONAL_OPERATOR" and token_value
in ["=", "<>", "<", "<=", ">", ">="]:
                relational_operator_node =
self.parse_relational_operator()

            expression_node.add_child(relational_operator_node)
            second_simple_expression_node =
self.parse_simple_expression()

            expression_node.add_child(second_simple_expression_node)

        return expression_node

    def parse_simple_expression(self):
        simple_expr_node = ParseNode("<simple_expression>")

        token = parser.current_token()
        if token and token[0] == "ARITHMETIC_OPERATOR" and
token[1] in ["+", "-"]:
            additive_operator_node =
parser.expression_parser.additive_operator()
            simple_expr_node.add_child(additive_operator_node)

        term_node = parser.expression_parser.parse_term()
        simple_expr_node.add_child(term_node)

        while parser.current_token():

```

```

        token = parser.current_token()

        if token[0] == "ARITHMETIC_OPERATOR" and token[1] in
["+ ", "- "]:
            second_additive_node =
parser.expression_parser.additive_operator()
            simple_expr_node.add_child(second_additive_node)

            term_node = parser.expression_parser.parse_term()
            simple_expr_node.add_child(term_node)
        elif token[0] == "LOGICAL_OPERATOR" and token[1] ==
"atau":
            second_additive_node =
parser.expression_parser.additive_operator()
            simple_expr_node.add_child(second_additive_node)

            term_node = parser.expression_parser.parse_term()
            simple_expr_node.add_child(term_node)
        else:
            break

    return simple_expr_node

def parse_term(self):
    term_node = ParseNode("<term>")

    factor_node = parser.expression_parser.parse_factor()
    term_node.add_child(factor_node)

    while parser.current_token():
        token = parser.current_token()

        if token[0] == "ARITHMETIC_OPERATOR" and token[1] in
["*", "/", "bagi", "mod"]:
            multiplicative_op_node =
parser.expression_parser.multiplicative_operator()
            term_node.add_child(multiplicative_op_node)

            factor_node =
parser.expression_parser.parse_factor()
            term_node.add_child(factor_node)
        elif token[0] == "LOGICAL_OPERATOR" and token[1] ==
"dan":
            multiplicative_op_node =
parser.expression_parser.multiplicative_operator()
            term_node.add_child(multiplicative_op_node)

```

```

        factor_node =
parser.expression_parser.parse_factor()
        term_node.add_child(factor_node)
    else:
        break

    return term_node

def parse_factor(self):
    factor_node = ParseNode("<factor>")

    token = parser.current_token()
    if token is None:
        raise SyntaxError("Unexpected end of input while parsing
factor")

    token_type, token_value = token

    if token_type == "IDENTIFIER":
        parser.advance()

factor_node.add_child(ParseNode(f"IDENTIFIER({token_value})"))

    # Check for function call, array index, or record field
access
    while True:
        next_token = parser.current_token()
        if not next_token:
            break

        if next_token[0] == "LPAREN":
            # Function call
            parser.advance()
            factor_node.add_child(ParseNode("LPAREN(()))")
            param_list =
parser.statement_parser.parse_parameter_list()
            factor_node.add_child(param_list)
            parser.check_token("RPAREN")
            factor_node.add_child(ParseNode("RPAREN(()))")
        elif next_token[0] == "LBRACKET":
            # Array indexing
            parser.advance()
            factor_node.add_child(ParseNode("LBRACKET([])"))
            index_expr =
parser.expression_parser.parse_expression()

```

```

        factor_node.add_child(index_expr)
        parser.check_token("RBRACKET")
        factor_node.add_child(ParseNode("RBRACKET(]"))))
    else:
        break

    return factor_node

    if token_type in ["NUMBER", "CHAR_LITERAL",
"STRING_LITERAL"]:
        parser.check_token(token_type)

factor_node.add_child(ParseNode(f"{token_type}({token_value})"))
    return factor_node

    if token_type == "LPAREN":
        parser.check_token("LPAREN", "(")
        factor_node.add_child(ParseNode("LPAREN(()))")
        expression_node =
parser.expression_parser.parse_expression()
        factor_node.add_child(expression_node)
        parser.check_token("RPAREN", ")")
        factor_node.add_child(ParseNode("RPAREN(()))")
        return factor_node

    if token_type == "LOGICAL_OPERATOR" and token_value ==
"tidak":
        parser.check_token("LOGICAL_OPERATOR", "tidak")

factor_node.add_child(ParseNode("LOGICAL_OPERATOR(tidak)"))
        inner_factor_node = parse_factor(parser)
        factor_node.add_child(inner_factor_node)
        return factor_node

def parse_relational_operator(self):
    relational_operator_node =
ParseNode("<relational_operator>")

    token = parser.current_token()

    if token is None:
        raise SyntaxError("Unexpected end of input while
parsing relational operator")

    token_type, token_value = token

```

```

        if token_type == "RELATIONAL_OPERATOR" and token_value in
["=", "<>", "<", "<=", ">", ">="]:
            parser.check_token("RELATIONAL_OPERATOR", token_value)

        relational_operator_node.add_child(ParseNode(f"RELATIONAL_OPERATOR
({token_value})"))
        return relational_operator_node

    def additive_operator(self):
        token = parent.current_token()

        if token[0] == 'ARITHMETIC_OPERATOR' and token[1] in ['+',
'-']:
            parent.advance()
            return ParseNode(f'ARITHMETIC_OPERATOR({token[1]})')

        if token[0] == 'LOGICAL_OPERATOR' and token[1] in
['atau']:
            parent.advance()
            return ParseNode(f'LOGICAL_OPERATOR({token[1]})')

    def multiplicative_operator(self):
        token = parent.current_token()

        if token[0] == 'ARITHMETIC_OPERATOR' and token[1] in ['*',
 '/', 'bagi', 'mod']:
            parent.advance()
            return
        ParseNode(f'MULTIPLICATIVE_OPERATOR({token[1]})')

        if token[0] == 'LOGICAL_OPERATOR' and token[1] in ['dan']:
            parent.advance()
            return
        ParseNode(f'MULTIPLICATIVE_OPERATOR({token[1]})')

```

Penjelasan

ExpressionParser merupakan kelas yang berfungsi untuk melakukan *parsing* terhadap <expression>, yang merupakan setiap ekspresi yang muncul di dalam program, meliputi interaktivitas antara dua variabel atau term. Setiap fungsi yang terdapat di kelas ini menjelaskan sebuah aturan produksi yang terdapat pada spek. Fungsi utama yang digunakan oleh ExpressionParser untuk memulai *parsing* terhadap sebuah ekspresi merupakan fungsi `parse_expression()`.

PENGUJIAN

1. test1.pas

Source Code

```
program BigTest;

konstanta
    pi = 314;
    hello = 'hi';

tipe
    age = integer;
    ScoreList = larik[1..30] dari char;
    test_range = 2..30;
    TRecord = rekaman x : integer; y : real; selesai;

variabel
    a, b: integer;
    c: char;
    flag: boolean;
    idx: integer;

fungsi AddThree(x, y: integer; z: real): real;
variabel
    temp: real;
mulai
    temp := x + y + z;
    AddThree := temp;
selesai;

prosedur Outer(u: integer);

    variabel
        localA: integer;

    prosedur Inner(v: integer);
mulai
    localA := u + v;
    write(localA);
selesai;
```

```

mulai
    localA := u * 2;
    Inner(localA);
selesai;

mulai
    a := 5;
    b := 10;
    c := 'Z';
    flag := tidak (a > 3);

    jika a < b maka
        write(a)
    selain_itu
        write(b);

    idx := 0;
    selama (idx < 3) dan (idx > 0) lakukan
        mulai
            idx := idx + 1;
            write(idx);
        selesai;

    untuk idx := 1 ke 3 lakukan
        write(idx);

    Outer(7);

    a := (b * 3 + 1) mod 4 - (2 * (3 + a));

    flag := (a < b) dan tidak (b = 10) atau (a = 5);

selesai.

```

Tokens

KEYWORD(program)
 IDENTIFIER(bigtest)
 SEMICOLON(;)
 KEYWORD(konstanta)
 IDENTIFIER(pi)

```
RELATIONAL_OPERATOR(=)
NUMBER(314)
SEMICOLON(;)
IDENTIFIER(hello)
RELATIONAL_OPERATOR(=)
STRING_LITERAL('hi')
SEMICOLON(;)
KEYWORD(tipe)
IDENTIFIER(age)
RELATIONAL_OPERATOR(=)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(scorelist)
RELATIONAL_OPERATOR(=)
KEYWORD(larik)
LBRACKET([)
NUMBER(1)
RANGE_OPERATOR(..)
NUMBER(30)
RBRACKET(])
KEYWORD(dari)
KEYWORD(char)
SEMICOLON(;)
IDENTIFIER(test_range)
RELATIONAL_OPERATOR(=)
NUMBER(2)
RANGE_OPERATOR(..)
NUMBER(30)
SEMICOLON(;)
IDENTIFIER(trecord)
RELATIONAL_OPERATOR(=)
KEYWORD(rekaman)
IDENTIFIER(x)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(y)
COLON(:)
KEYWORD(real)
SEMICOLON(;)
KEYWORD(selesai)
SEMICOLON(;)
KEYWORD(variabel)
IDENTIFIER(a)
COMMA(,)
IDENTIFIER(b)
COLON(:)
KEYWORD(integer)
SEMICOLON(;
```

IDENTIFIER(c)
COLON(:)
KEYWORD(char)
SEMICOLON(;)
IDENTIFIER(flag)
COLON(:)
KEYWORD(boolean)
SEMICOLON(;)
IDENTIFIER(idx)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(fungsi)
IDENTIFIER(addthree)
LPAREN()
IDENTIFIER(x)
COMMA(,
IDENTIFIER(y)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(z)
COLON(:)
KEYWORD(real)
RPAREN())
COLON(:)
KEYWORD(real)
SEMICOLON(;)
KEYWORD(variabel)
IDENTIFIER(temp)
COLON(:)
KEYWORD(real)
SEMICOLON(;)
KEYWORD(mulai)
IDENTIFIER(temp)
ASSIGN_OPERATOR(:=)
IDENTIFIER(x)
ARITHMETIC_OPERATOR(+)
IDENTIFIER(y)
ARITHMETIC_OPERATOR(+)
IDENTIFIER(z)
SEMICOLON(;)
IDENTIFIER(addthree)
ASSIGN_OPERATOR(:=)
IDENTIFIER(temp)
SEMICOLON(;)
KEYWORD(selesai)
SEMICOLON(;)
KEYWORD(prosedur)

IDENTIFIER(outer)
LPAREN()
IDENTIFIER(u)
COLON(:)
KEYWORD(integer)
RPAREN()
SEMICOLON(;)
KEYWORD(variabel)
IDENTIFIER(locala)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(prosedur)
IDENTIFIER(inner)
LPAREN()
IDENTIFIER(v)
COLON(:)
KEYWORD(integer)
RPAREN()
SEMICOLON(;)
KEYWORD(mulai)
IDENTIFIER(locala)
ASSIGN_OPERATOR(:=)
IDENTIFIER(u)
ARITHMETIC_OPERATOR(+)
IDENTIFIER(v)
SEMICOLON(;)
IDENTIFIER(write)
LPAREN()
IDENTIFIER(locala)
RPAREN()
SEMICOLON(;)
KEYWORD(selesai)
SEMICOLON(;)
KEYWORD(mulai)
IDENTIFIER(locala)
ASSIGN_OPERATOR(:=)
IDENTIFIER(u)
ARITHMETIC_OPERATOR(*)
NUMBER(2)
SEMICOLON(;)
IDENTIFIER(inner)
LPAREN()
IDENTIFIER(locala)
RPAREN()
SEMICOLON(;)
KEYWORD(selesai)
SEMICOLON(;)
KEYWORD(mulai)

IDENTIFIER(a)
ASSIGN_OPERATOR(:=)
NUMBER(5)
SEMICOLON(;)
IDENTIFIER(b)
ASSIGN_OPERATOR(:=)
NUMBER(10)
SEMICOLON(;)
IDENTIFIER(c)
ASSIGN_OPERATOR(:=)
CHAR_LITERAL('z')
SEMICOLON(;)
IDENTIFIER(flag)
ASSIGN_OPERATOR(:=)
LOGICAL_OPERATOR(tidak)
LPAREN()
IDENTIFIER(a)
RELATIONAL_OPERATOR(>)
NUMBER(3)
RPAREN())
SEMICOLON(;)
KEYWORD(jika)
IDENTIFIER(a)
RELATIONAL_OPERATOR(<)
IDENTIFIER(b)
KEYWORD(maka)
IDENTIFIER(write)
LPAREN()
IDENTIFIER(a)
RPAREN())
KEYWORD(selain_itu)
IDENTIFIER(write)
LPAREN()
IDENTIFIER(b)
RPAREN())
SEMICOLON(;)
IDENTIFIER(idx)
ASSIGN_OPERATOR(:=)
NUMBER(0)
SEMICOLON(;)
KEYWORD(selama)
LPAREN()
IDENTIFIER(idx)
RELATIONAL_OPERATOR(<)
NUMBER(3)
RPAREN())
LOGICAL_OPERATOR(dan)
LPAREN()
IDENTIFIER(idx)

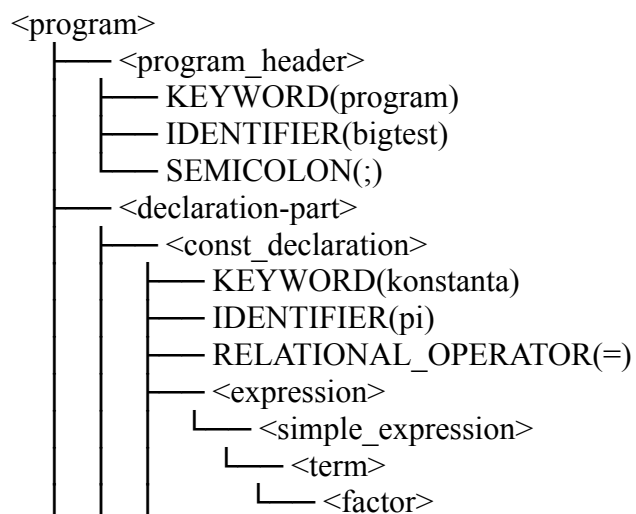
RELATIONAL_OPERATOR(>)
NUMBER(0)
RPAREN()
KEYWORD(lakukan)
KEYWORD(mulai)
IDENTIFIER(idx)
ASSIGN_OPERATOR(:=)
IDENTIFIER(idx)
ARITHMETIC_OPERATOR(+)
NUMBER(1)
SEMICOLON(;)
IDENTIFIER(write)
LPAREN()
IDENTIFIER(idx)
RPAREN()
SEMICOLON(;)
KEYWORD(selesai)
SEMICOLON(;)
KEYWORD(untuk)
IDENTIFIER(idx)
ASSIGN_OPERATOR(:=)
NUMBER(1)
KEYWORD(ke)
NUMBER(3)
KEYWORD(lakukan)
IDENTIFIER(write)
LPAREN()
IDENTIFIER(idx)
RPAREN()
SEMICOLON(;)
IDENTIFIER(outer)
LPAREN()
NUMBER(7)
RPAREN()
SEMICOLON(;)
IDENTIFIER(a)
ASSIGN_OPERATOR(:=)
LPAREN()
IDENTIFIER(b)
ARITHMETIC_OPERATOR(*)
NUMBER(3)
ARITHMETIC_OPERATOR(+)
NUMBER(1)
RPAREN()
ARITHMETIC_OPERATOR(mod)
NUMBER(4)
ARITHMETIC_OPERATOR(-)
LPAREN()
NUMBER(2)

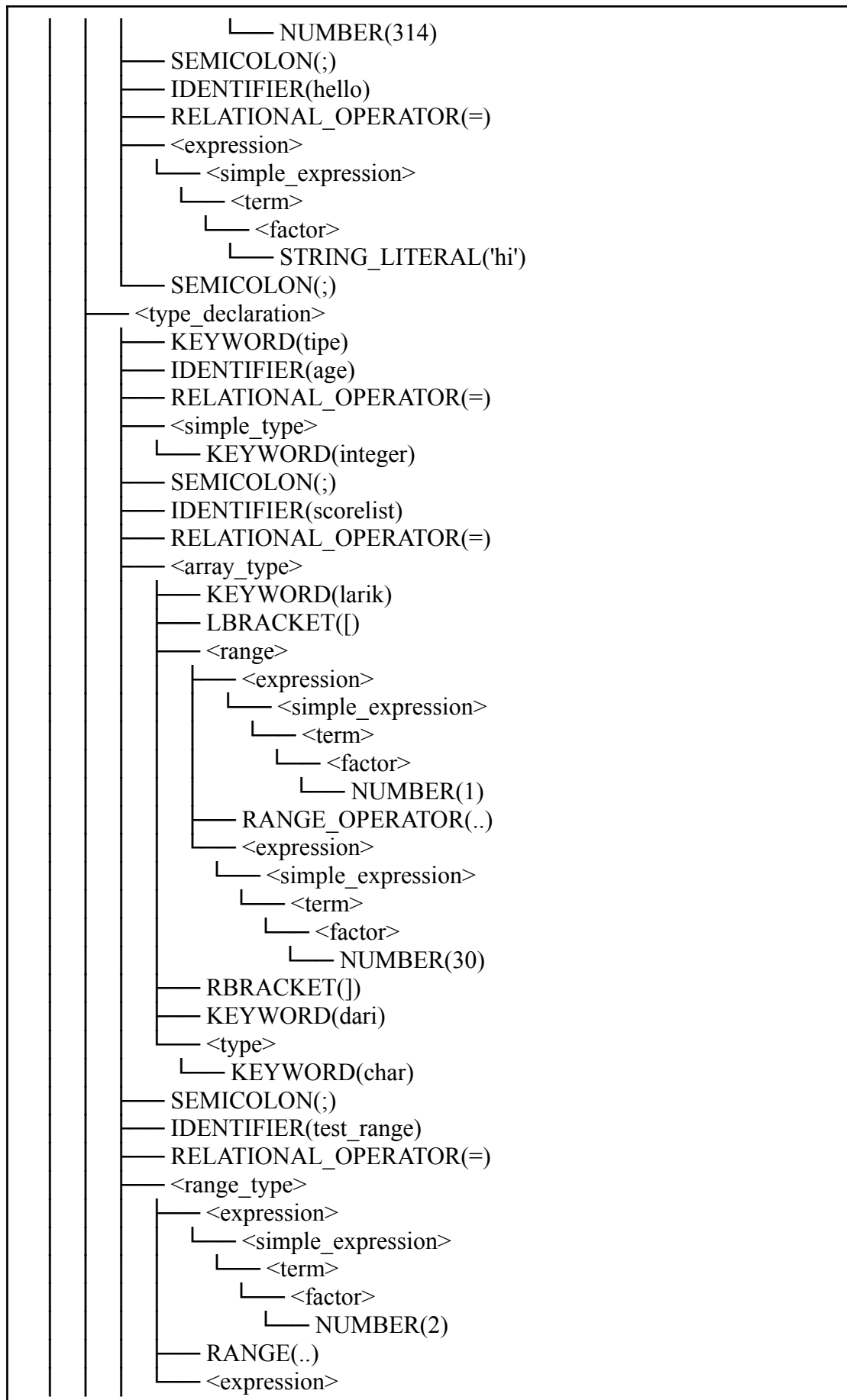
```

ARITHMETIC_OPERATOR(*)
LPAREN()
NUMBER(3)
ARITHMETIC_OPERATOR(+)
IDENTIFIER(a)
RPAREN()
RPAREN()
SEMICOLON(;)
IDENTIFIER(flag)
ASSIGN_OPERATOR(:=)
LPAREN()
IDENTIFIER(a)
RELATIONAL_OPERATOR(<)
IDENTIFIER(b)
RPAREN()
LOGICAL_OPERATOR(dan)
LOGICAL_OPERATOR(tidak)
LPAREN()
IDENTIFIER(b)
RELATIONAL_OPERATOR(=)
NUMBER(10)
RPAREN()
LOGICAL_OPERATOR(atau)
LPAREN()
IDENTIFIER(a)
RELATIONAL_OPERATOR(=)
NUMBER(5)
RPAREN()
SEMICOLON(;)
KEYWORD(selesai)
DOT(.)

```

Tree

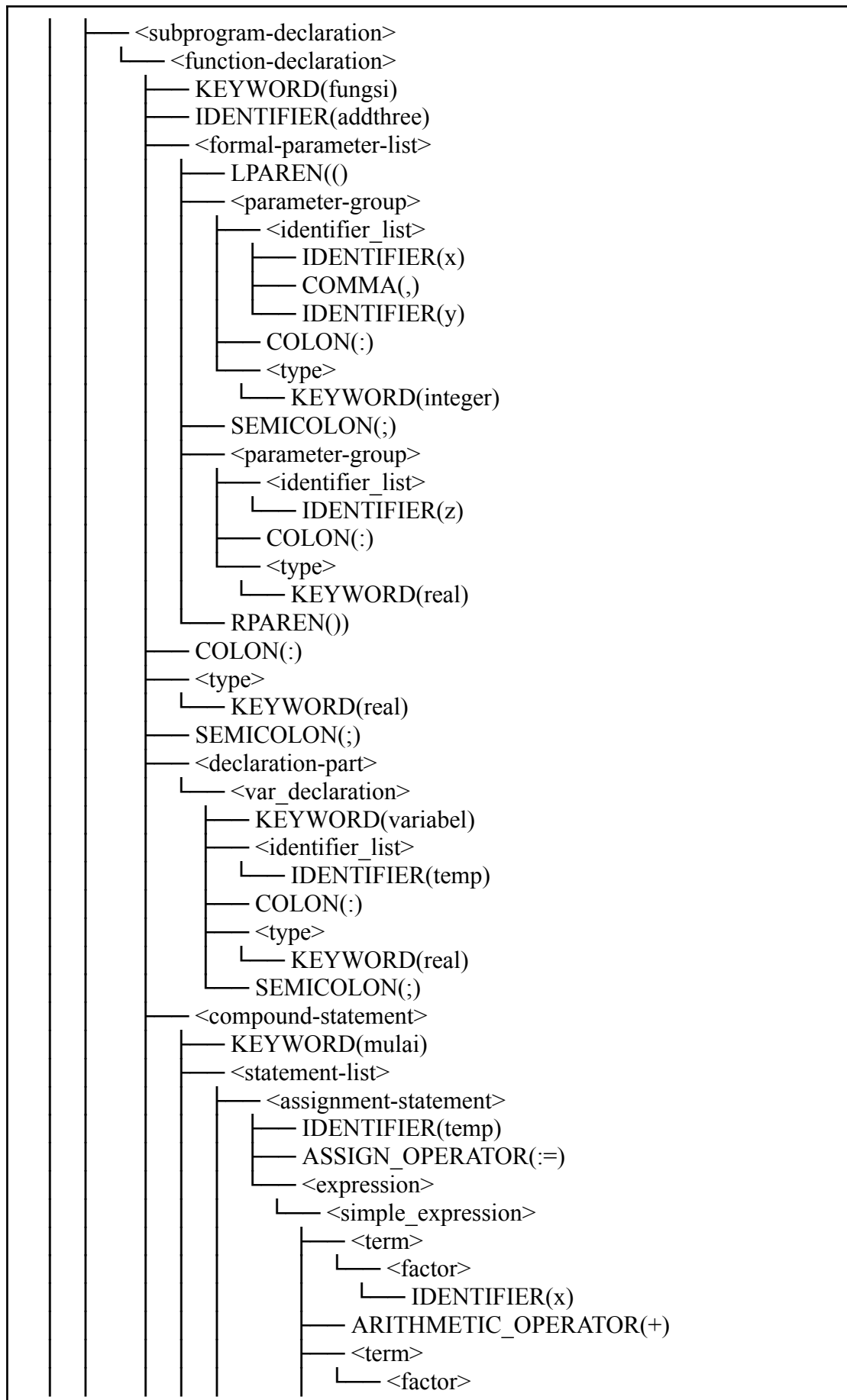


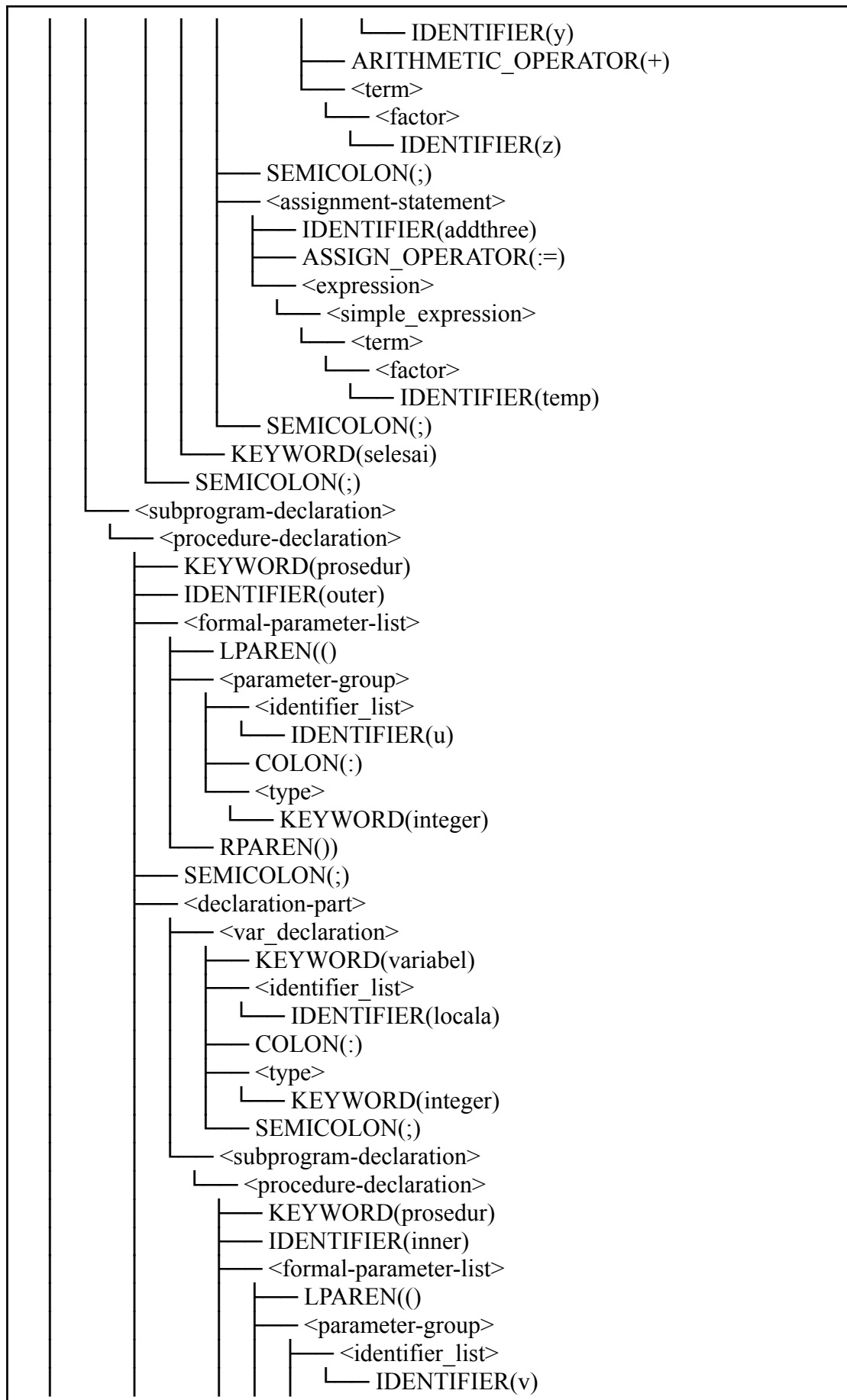


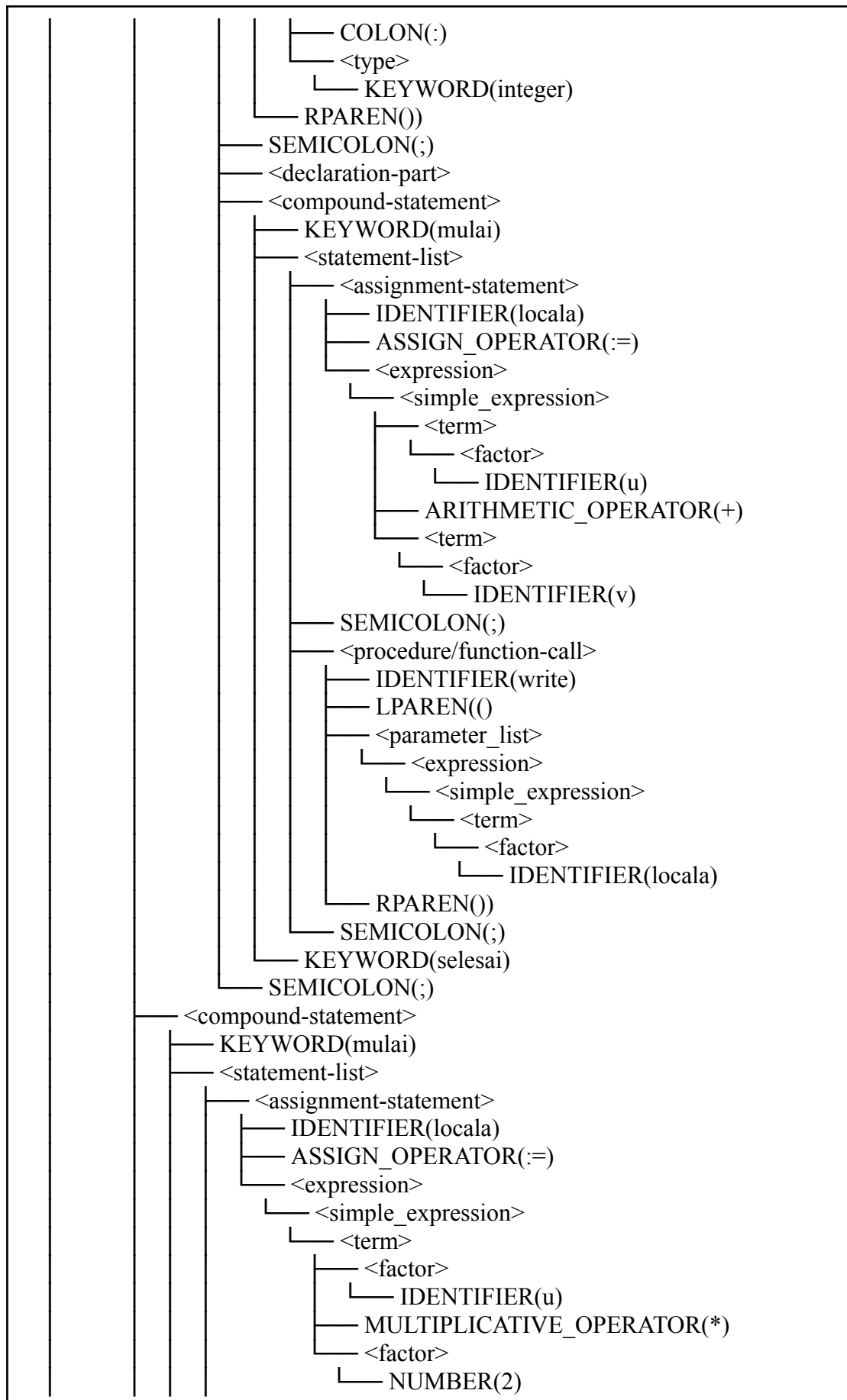
```

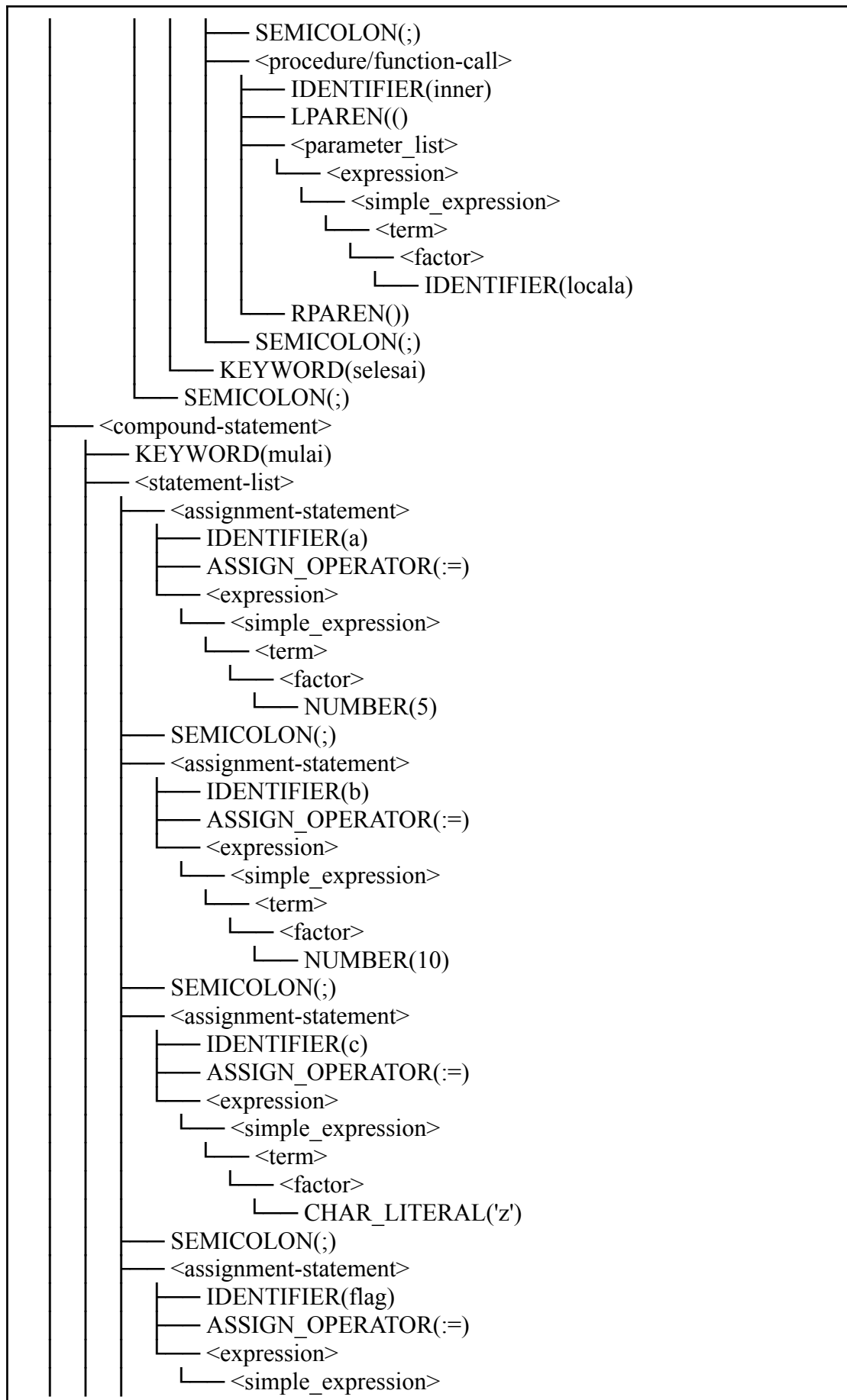
└─ <simple_expression>
  └─ <term>
    └─ <factor>
      └─ NUMBER(30)
SEMICOLON(;)
IDENTIFIER(trecord)
RELATIONAL_OPERATOR(=)
<record_type>
├─ KEYWORD(rekaman)
├─ IDENTIFIER(x)
├─ COLON(:)
├─ <simple_type>
│   └─ KEYWORD(integer)
├─ SEMICOLON(;)
├─ IDENTIFIER(y)
├─ COLON(:)
├─ <simple_type>
│   └─ KEYWORD(real)
├─ SEMICOLON(;)
└─ KEYWORD(selesai)
SEMICOLON(;)
<var_declaration>
├─ KEYWORD(variabel)
├─ <identifier_list>
│   ├── IDENTIFIER(a)
│   ├── COMMA(,)
│   └─ IDENTIFIER(b)
├─ COLON(:)
├─ <type>
│   └─ KEYWORD(integer)
├─ SEMICOLON(;)
├─ <identifier_list>
│   └─ IDENTIFIER(c)
├─ COLON(:)
├─ <type>
│   └─ KEYWORD(char)
├─ SEMICOLON(;)
├─ <identifier_list>
│   └─ IDENTIFIER(flag)
├─ COLON(:)
├─ <type>
│   └─ KEYWORD(boolean)
├─ SEMICOLON(;)
├─ <identifier_list>
│   └─ IDENTIFIER(idx)
├─ COLON(:)
├─ <type>
│   └─ KEYWORD(integer)
└─ SEMICOLON(;)

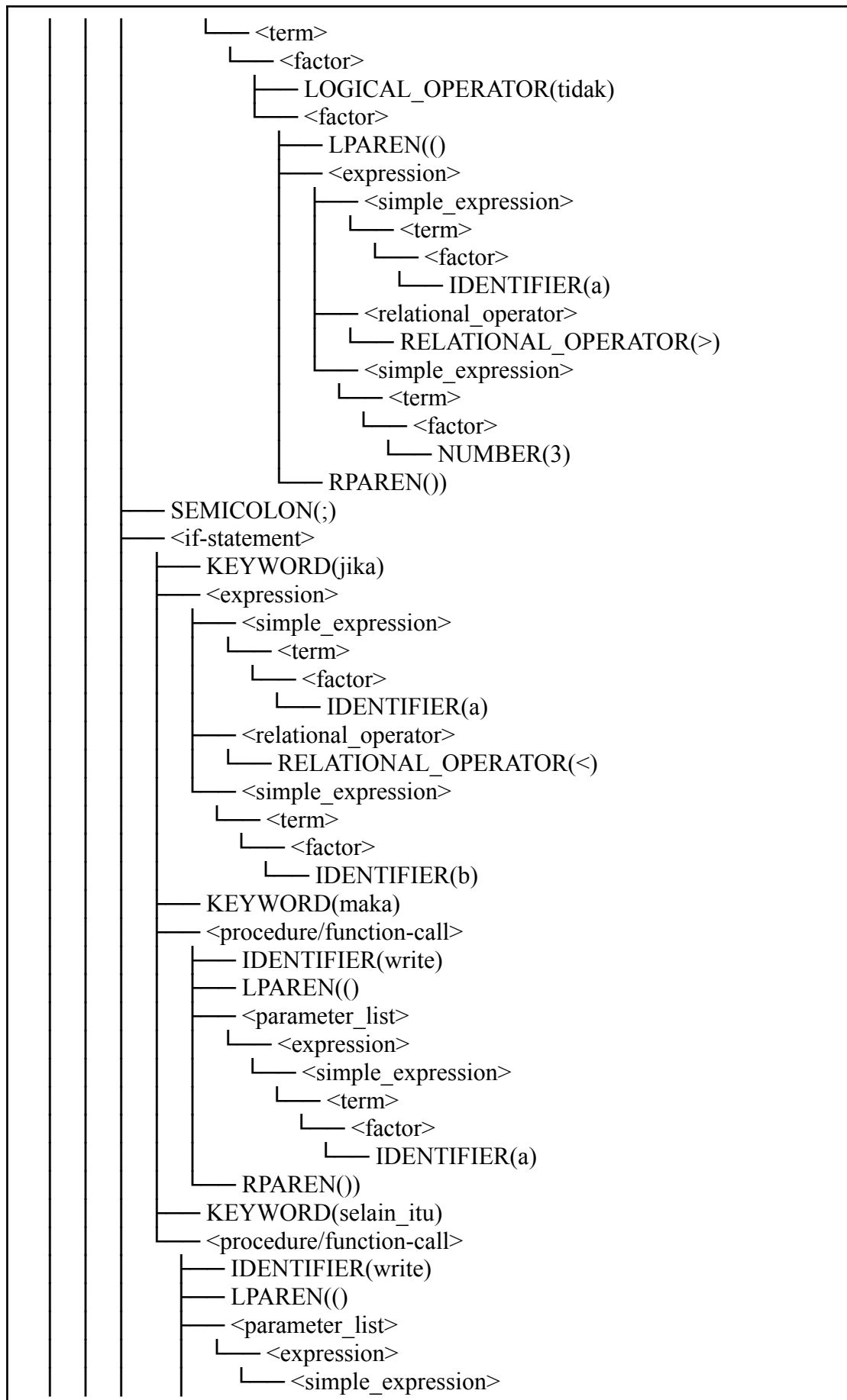
```

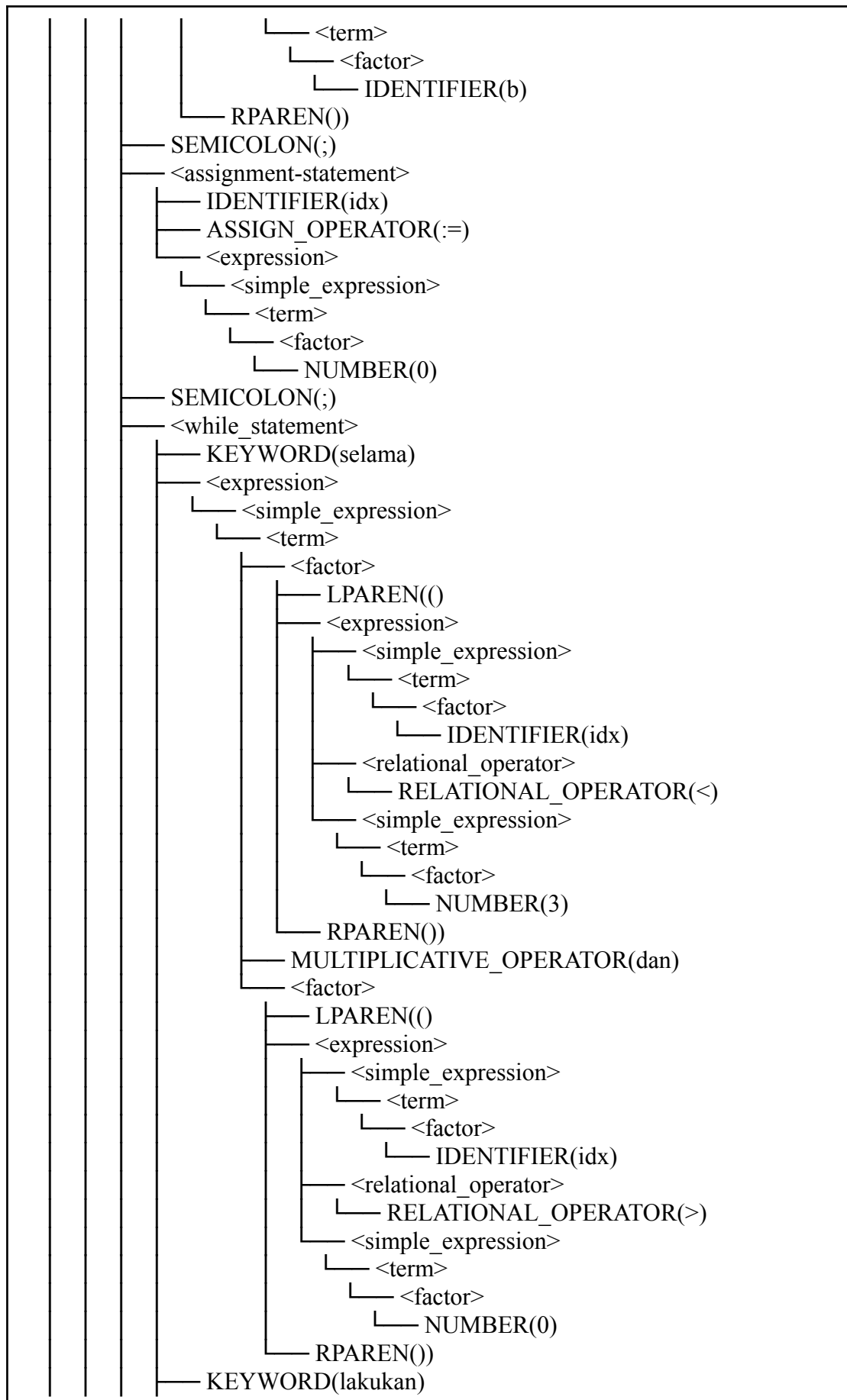


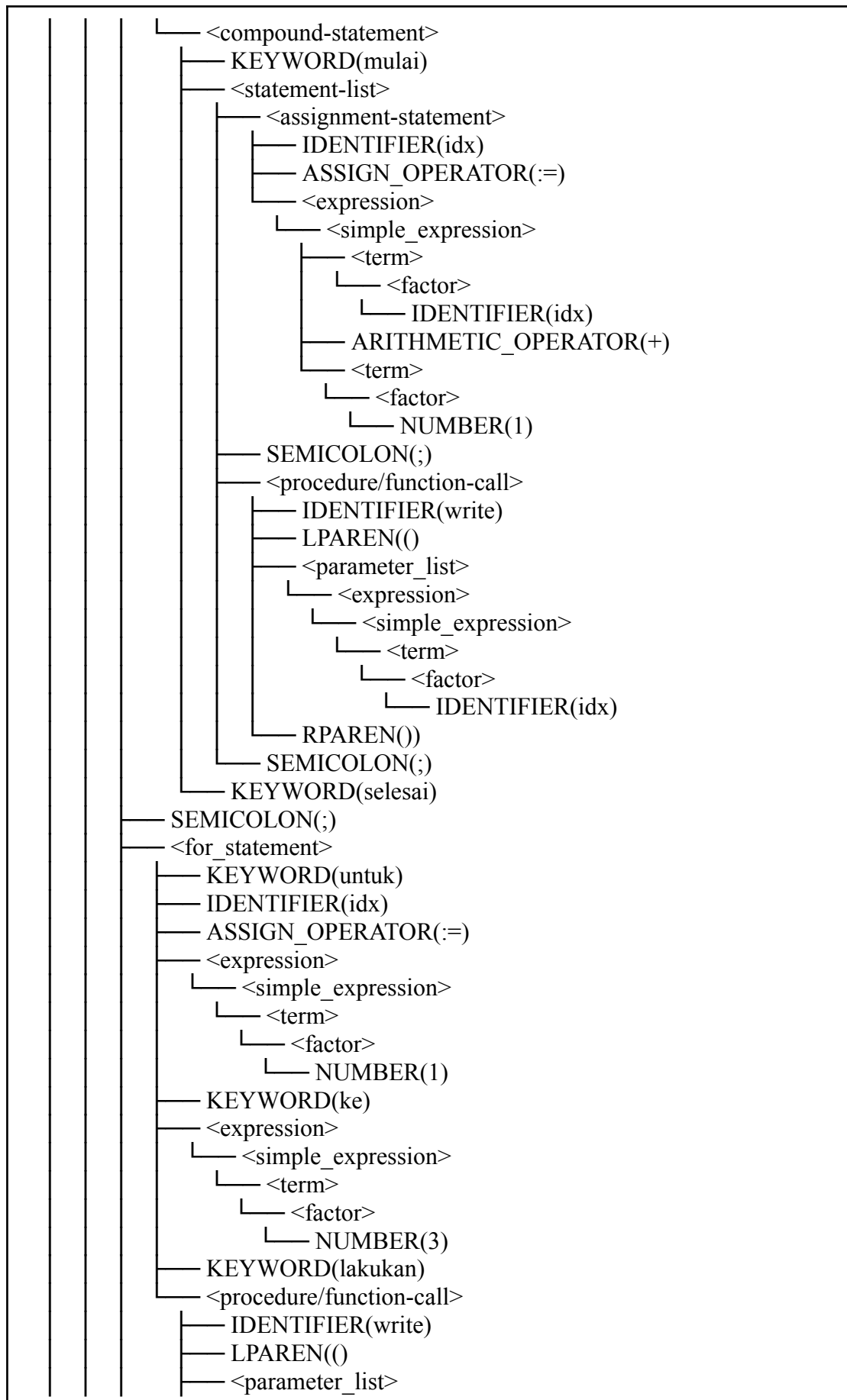


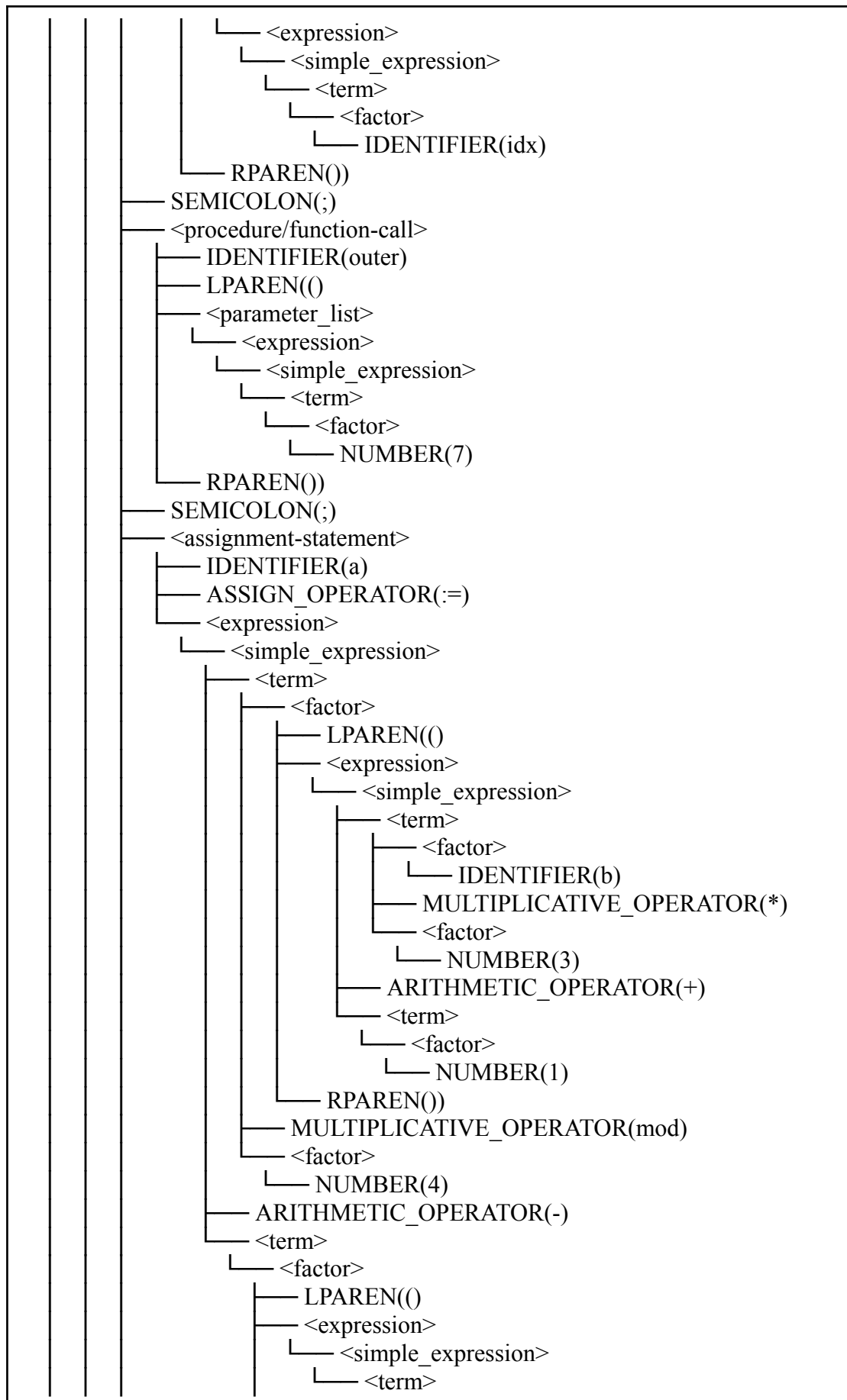


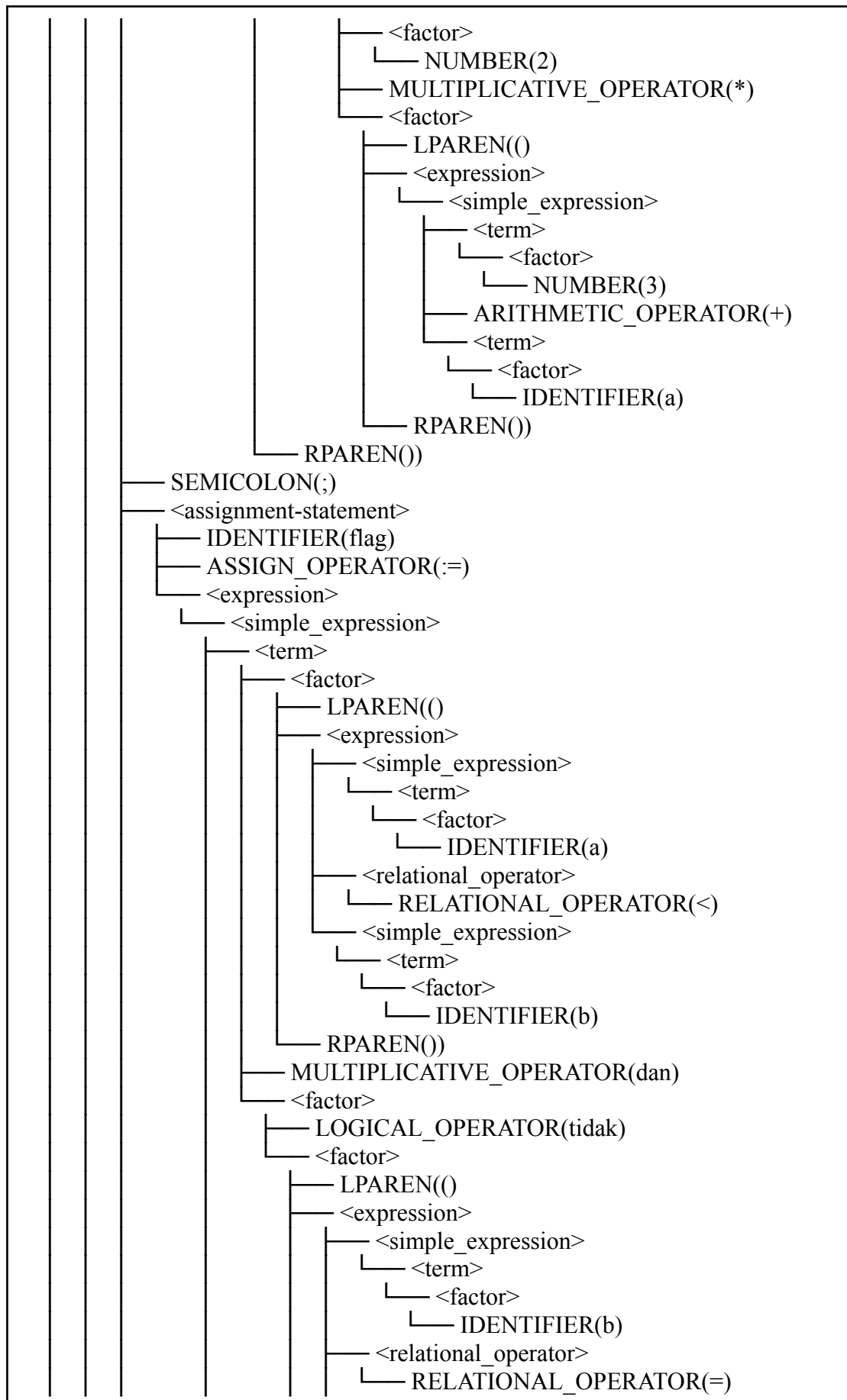













```

    flag: Boolean;

mulai
    aw := 1;
selesai.

```

Tokens

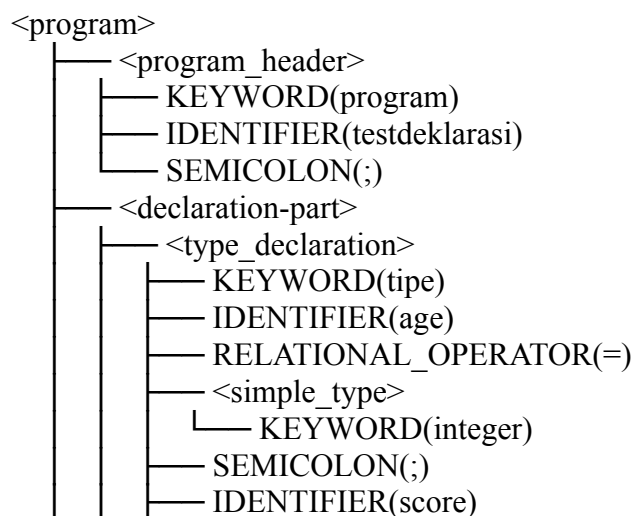
KEYWORD(program)
 IDENTIFIER(testdeklarasi)
 SEMICOLON(;)
 KEYWORD(tipe)
 IDENTIFIER(age)
 RELATIONAL_OPERATOR(=)
 KEYWORD(integer)
 SEMICOLON(;)
 IDENTIFIER(score)
 RELATIONAL_OPERATOR(=)
 KEYWORD(larik)
 LBRACKET([)
 NUMBER(1)
 RANGE_OPERATOR(..)
 NUMBER(30)
 RBRACKET(])
 KEYWORD(dari)
 KEYWORD(char)
 SEMICOLON(;)
 IDENTIFIER(test_range)
 RELATIONAL_OPERATOR(=)
 NUMBER(2)
 RANGE_OPERATOR(..)
 NUMBER(30)
 SEMICOLON(;)
 IDENTIFIER(record)
 RELATIONAL_OPERATOR(=)
 KEYWORD(rekaman)
 IDENTIFIER(x)
 COLON(:)
 KEYWORD(integer)
 SEMICOLON(;)
 IDENTIFIER(y)
 COLON(:)
 KEYWORD(real)
 SEMICOLON(;)
 KEYWORD(selesai)
 SEMICOLON(;)
 KEYWORD(konstanta)

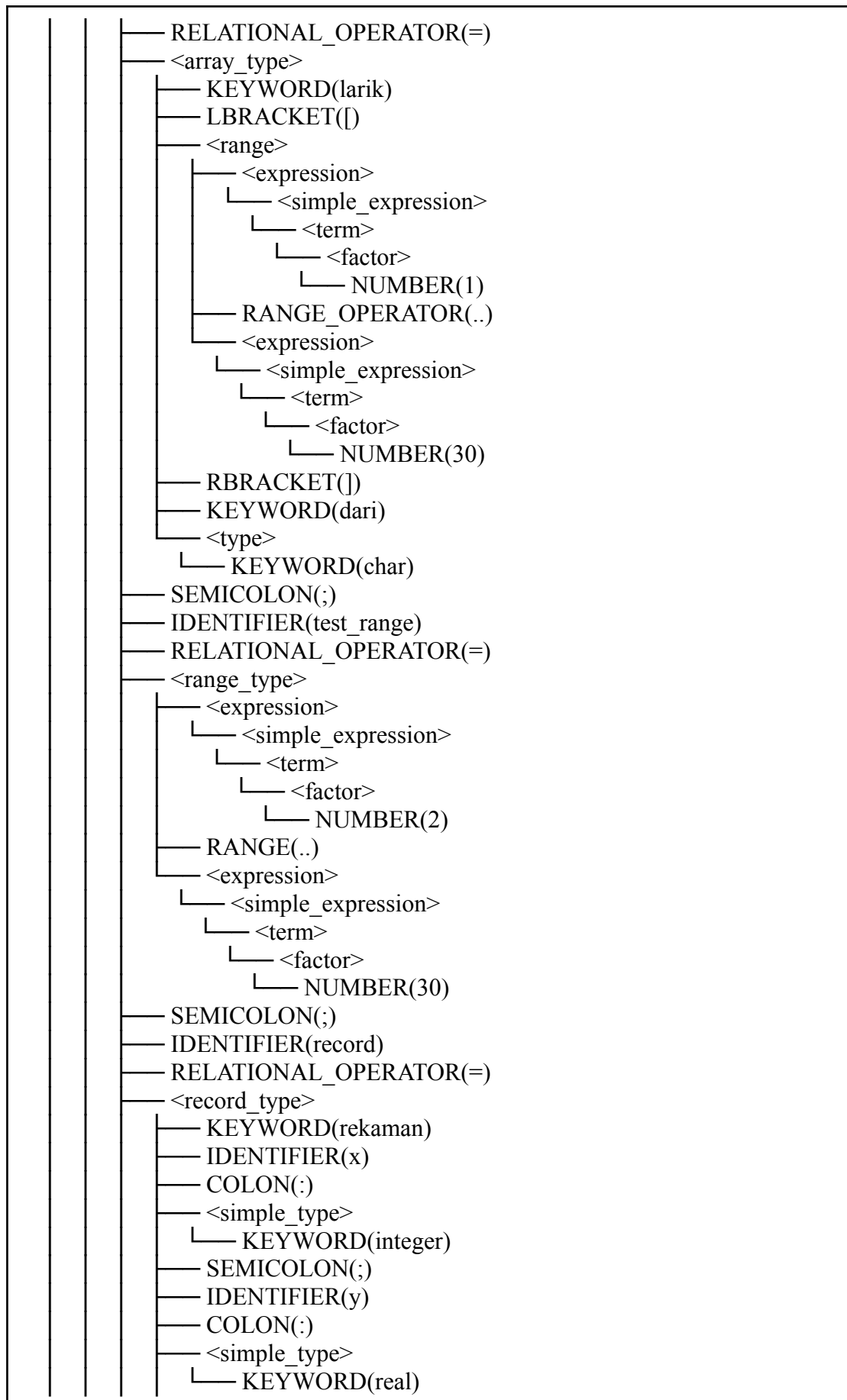
```

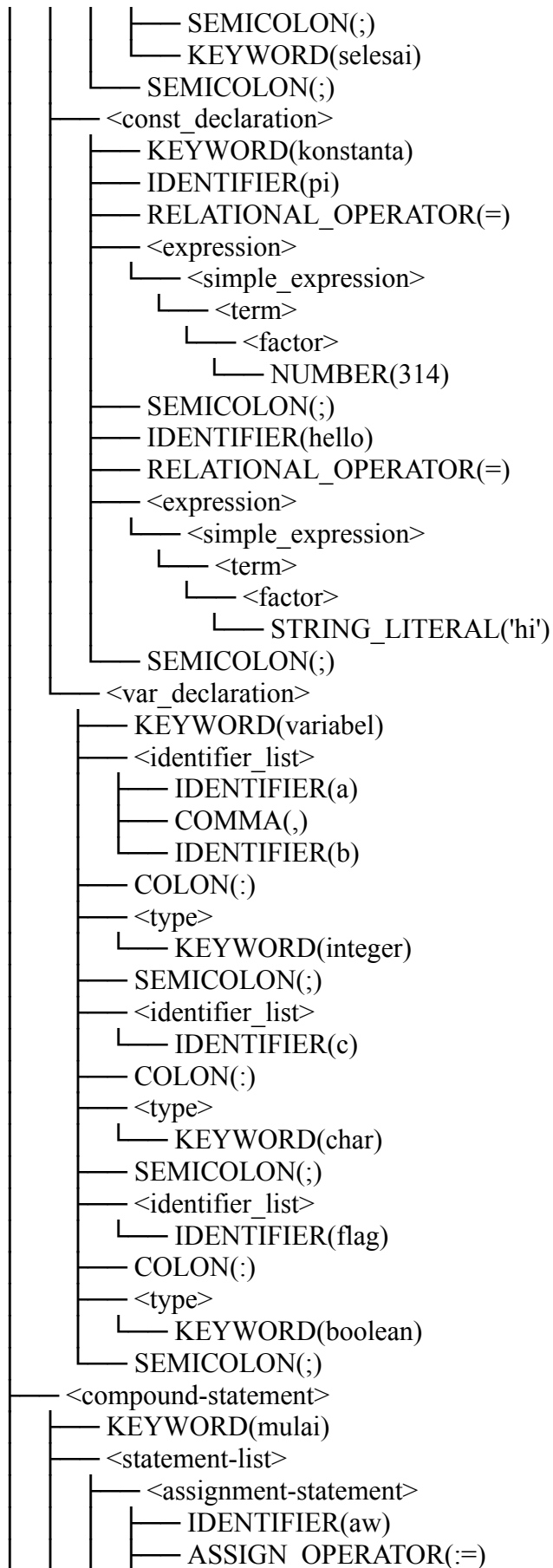
IDENTIFIER(pi)
RELATIONAL_OPERATOR(=)
NUMBER(314)
SEMICOLON(;
IDENTIFIER(hello)
RELATIONAL_OPERATOR(=)
STRING_LITERAL('hi')
SEMICOLON(;
KEYWORD(variabel)
IDENTIFIER(a)
COMMA(,
IDENTIFIER(b)
COLON(:)
KEYWORD(integer)
SEMICOLON(;
IDENTIFIER(c)
COLON(:)
KEYWORD(char)
SEMICOLON(;
IDENTIFIER(flag)
COLON(:)
KEYWORD(boolean)
SEMICOLON(;
KEYWORD(mulai)
IDENTIFIER(aw)
ASSIGN_OPERATOR(:=)
NUMBER(1)
SEMICOLON(;
KEYWORD(selesai)
DOT(.)

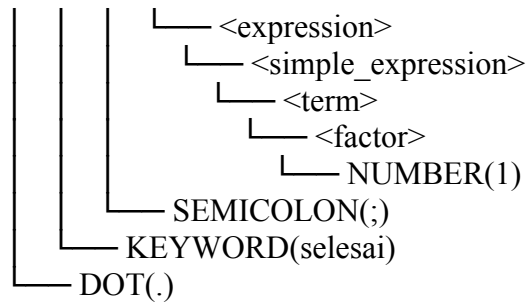
```

Tree









3. test3.pas

Source Code

```
program Loop;

variabel
  x, i: integer;
  ok: boolean;

mulai
  x := 5;
  ok := tidak (x = 3);

  jika ok maka
    write(x)
  selain_itu
    write(0);

  i := 1;
  selama i < 4 lakukan
    mulai
      write(i);
      i := i + 1;
    selesai;

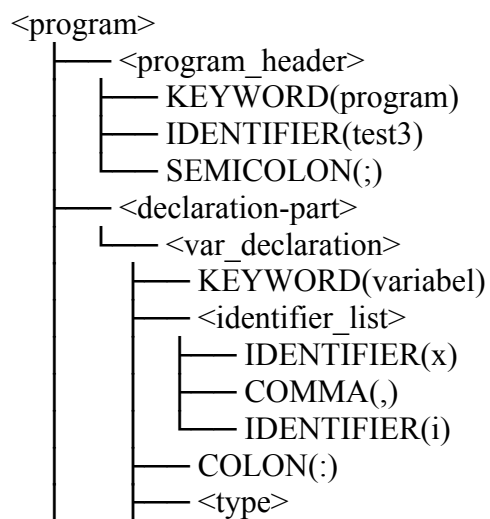
  untuk i := 1 ke 3 lakukan
    write(i);

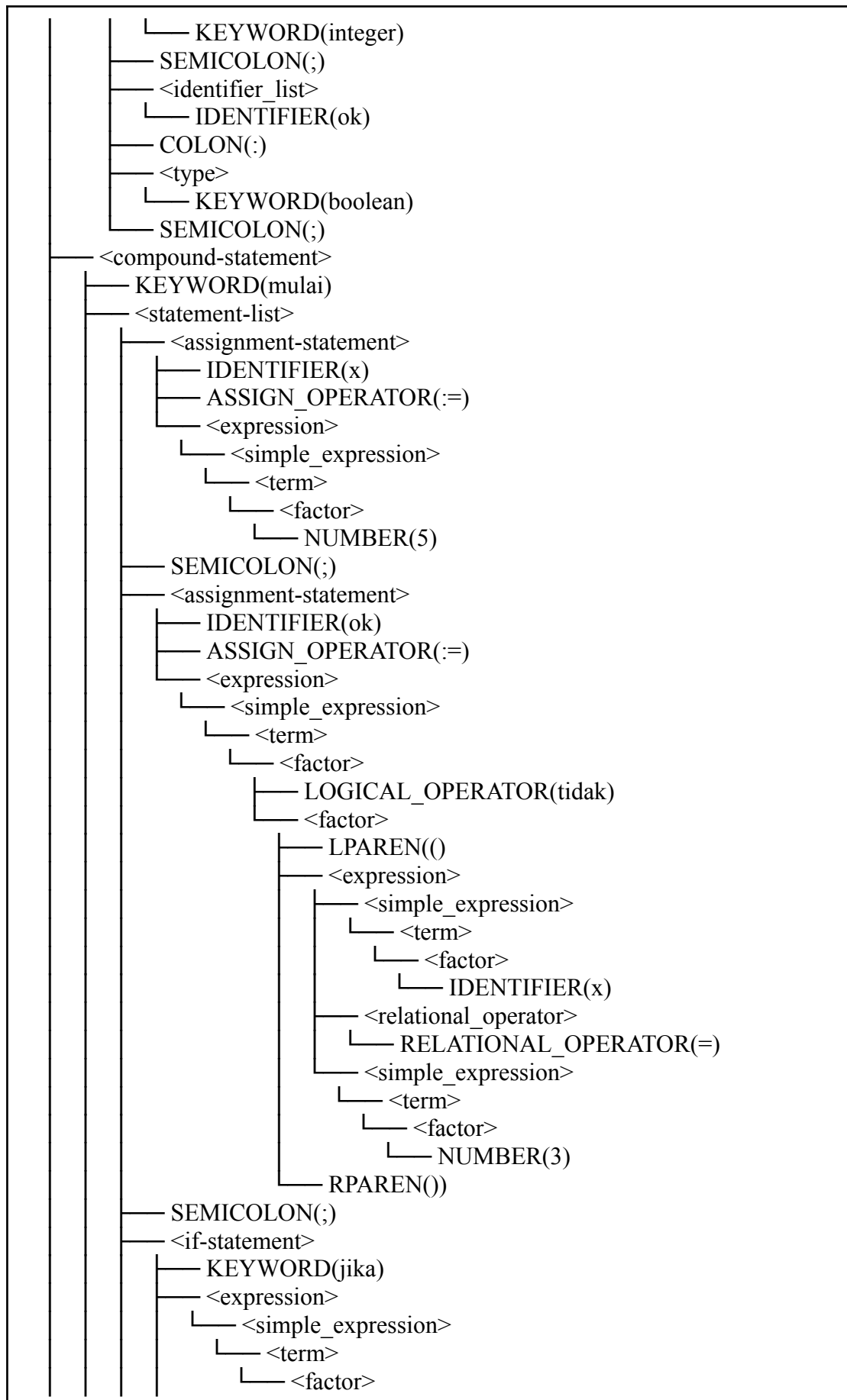
selesai.
```

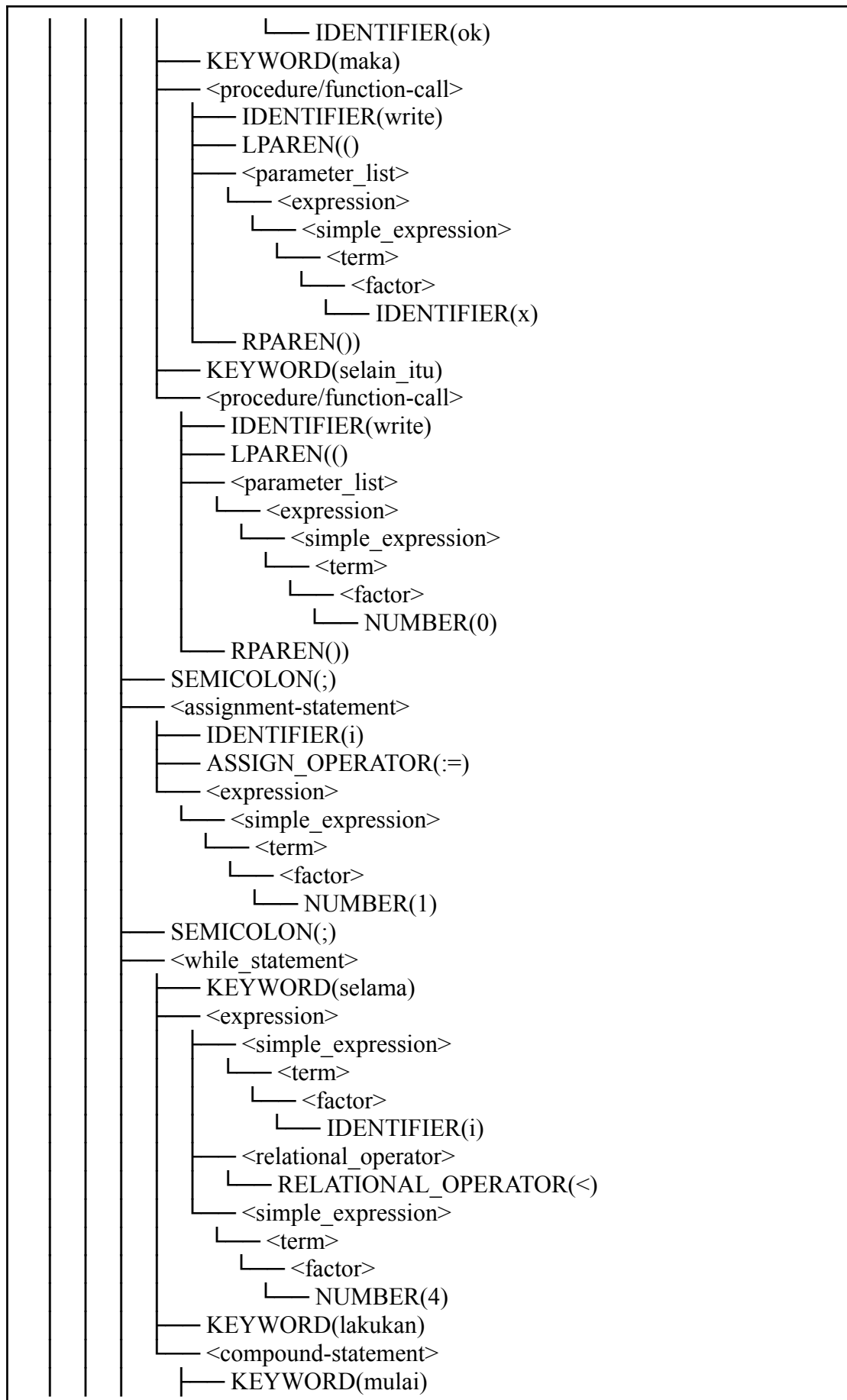
Tokens
KEYWORD(program) IDENTIFIER(test3) SEMICOLON(;) KEYWORD(variabel) IDENTIFIER(x) COMMA(, IDENTIFIER(i) COLON(:) KEYWORD(integer) SEMICOLON(;) IDENTIFIER(ok) COLON(:) KEYWORD(boolean) SEMICOLON(;) KEYWORD(mulai) IDENTIFIER(x) ASSIGN_OPERATOR(:=) NUMBER(5) SEMICOLON(;) IDENTIFIER(ok) ASSIGN_OPERATOR(:=) LOGICAL_OPERATOR(tidak) LPAREN() IDENTIFIER(x) RELATIONAL_OPERATOR(=) NUMBER(3) RPAREN()) SEMICOLON(;) KEYWORD(jika) IDENTIFIER(ok) KEYWORD(maka) IDENTIFIER(write) LPAREN() IDENTIFIER(x) RPAREN()) KEYWORD(selain_itu) IDENTIFIER(write) LPAREN() NUMBER(0) RPAREN()) SEMICOLON(;) IDENTIFIER(i) ASSIGN_OPERATOR(:=) NUMBER(1) SEMICOLON(;) KEYWORD(selama) IDENTIFIER(i)

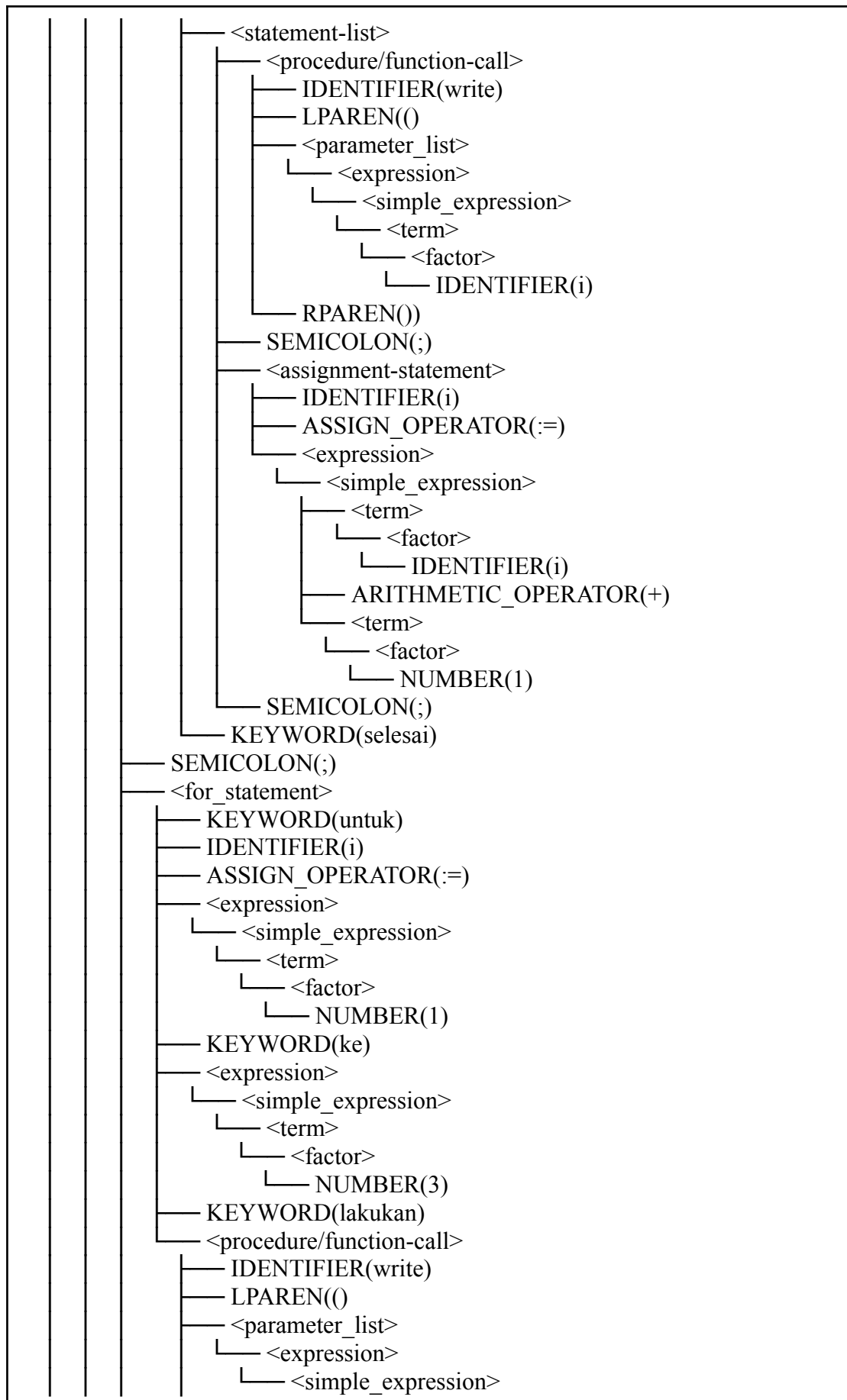
RELATIONAL_OPERATOR(<)
 NUMBER(4)
 KEYWORD(lakukan)
 KEYWORD(mulai)
 IDENTIFIER(write)
 LPAREN()
 IDENTIFIER(i)
 RPAREN()
 SEMICOLON(;)
 IDENTIFIER(i)
 ASSIGN_OPERATOR(:=)
 IDENTIFIER(i)
 ARITHMETIC_OPERATOR(+)
 NUMBER(1)
 SEMICOLON(;)
 KEYWORD(selesai)
 SEMICOLON(;)
 KEYWORD(untuk)
 IDENTIFIER(i)
 ASSIGN_OPERATOR(:=)
 NUMBER(1)
 KEYWORD(ke)
 NUMBER(3)
 KEYWORD(lakukan)
 IDENTIFIER(write)
 LPAREN()
 IDENTIFIER(i)
 RPAREN()
 SEMICOLON(;)
 KEYWORD(selesai)
 DOT(.)

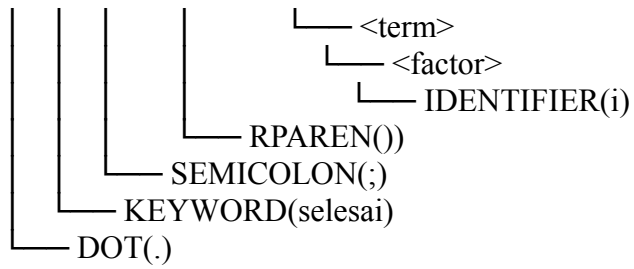
Tree











4. test4.pas

Source Code

```

program test4;

tipe
  angka = integer;
  Arr30 = larik[1..5] dari integer;
  Point = rekaman x: integer; y: integer; selesai;

variabel
  i: integer;

mulai
  untuk i := 1 ke 5 lakukan
    A := i * 2;

  write(A);
selesai.

```

Tokens

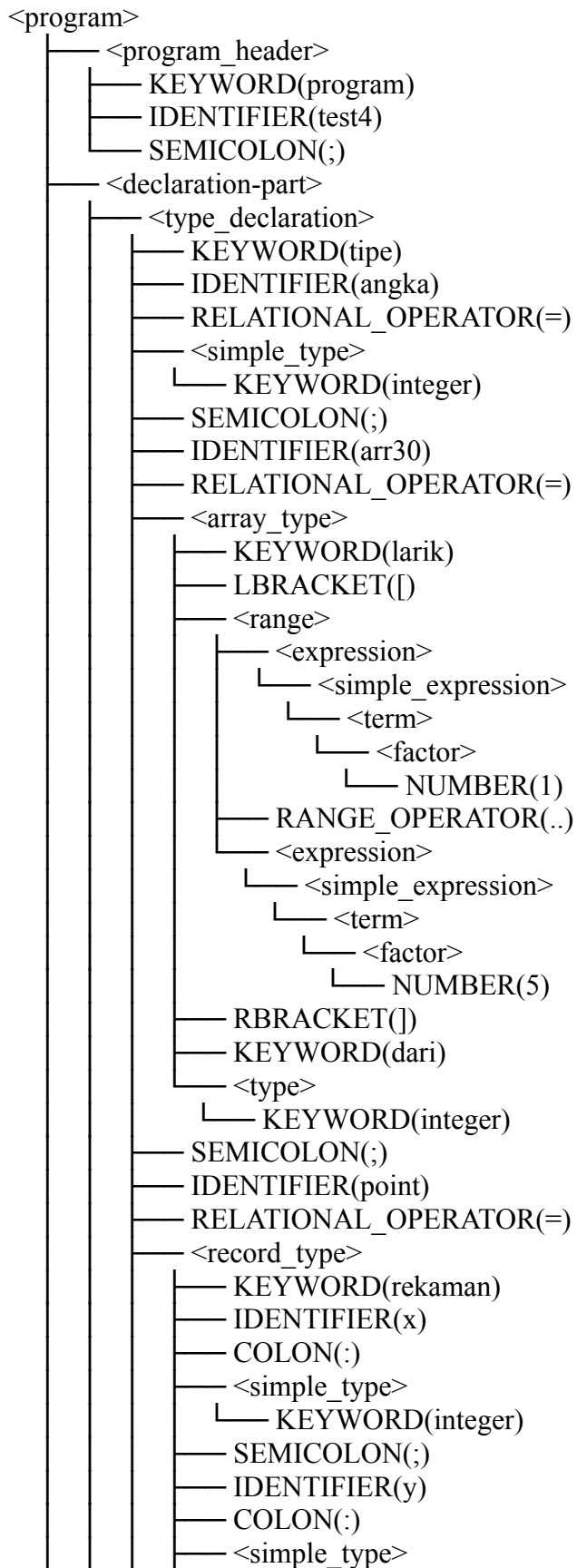
```

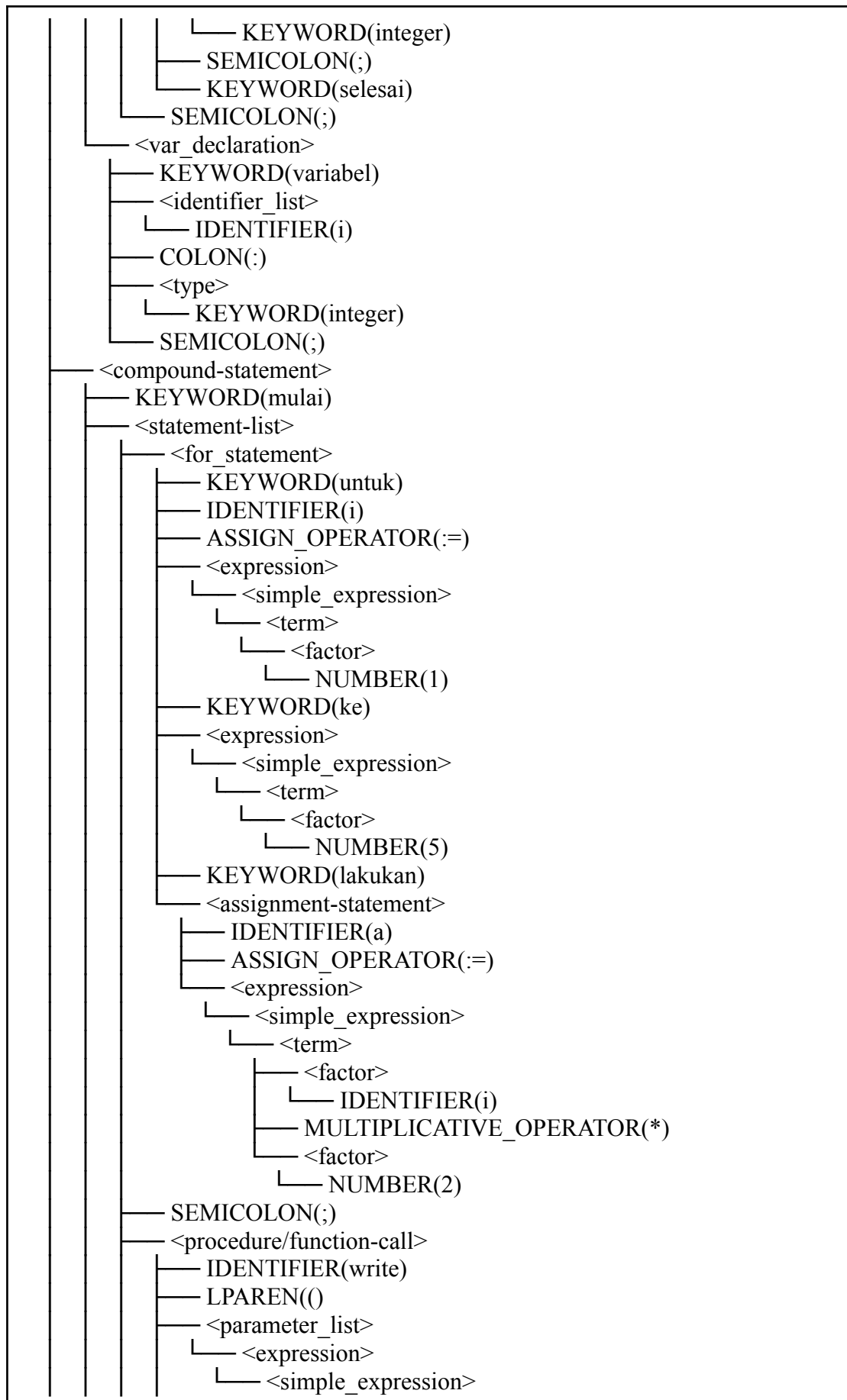
KEYWORD(program)
IDENTIFIER(test4)
SEMICOLON(; )
KEYWORD(tipe)
IDENTIFIER(angka)
RELATIONAL_OPERATOR(=)
KEYWORD(integer)
SEMICOLON(; )
IDENTIFIER(arr30)
RELATIONAL_OPERATOR(=)
KEYWORD(larik)

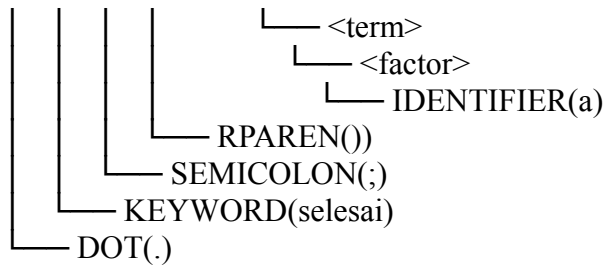
```

LBRACKET([)
NUMBER(1)
RANGE_OPERATOR(..)
NUMBER(5)
RBRACKET(])
KEYWORD(dari)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(point)
RELATIONAL_OPERATOR(=)
KEYWORD(rekaman)
IDENTIFIER(x)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(y)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(selesai)
SEMICOLON(;)
KEYWORD(variabel)
IDENTIFIER(i)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(mulai)
KEYWORD(untuk)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
NUMBER(1)
KEYWORD(ke)
NUMBER(5)
KEYWORD(lakukan)
IDENTIFIER(a)
ASSIGN_OPERATOR(:=)
IDENTIFIER(i)
ARITHMETIC_OPERATOR(*)
NUMBER(2)
SEMICOLON(;)
IDENTIFIER(write)
LPAREN()
IDENTIFIER(a)
RPAREN())
SEMICOLON(;)
KEYWORD(selesai)
DOT(.)

Tree







5. test5.pas

Source Code

```

program Test5;

funksi AddOne(x: integer): integer;
mulai
    AddOne := x + 1;
selesai;

prosedur PrintTwice(v: integer);
mulai
    write(v);
    write(v);
selesai;

variabel
    a: integer;
    ok: boolean;

mulai
    a := AddOne(7);
    ok := (a > 5) dan tidak (a = 10);

    jika ok maka
        PrintTwice(a)
    selain_itu
        write(0);

selesai.
  
```

Tokens

KEYWORD(program)
IDENTIFIER(test5)
SEMICOLON(;)
KEYWORD(fungsi)
IDENTIFIER(addone)
LPAREN()
IDENTIFIER(x)
COLON(:)
KEYWORD(integer)
RPAREN())
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(mulai)
IDENTIFIER(addone)
ASSIGN_OPERATOR(:=)
IDENTIFIER(x)
ARITHMETIC_OPERATOR(+)
NUMBER(1)
SEMICOLON(;)
KEYWORD(selesai)
SEMICOLON(;)
KEYWORD(prosedur)
IDENTIFIER(printtwice)
LPAREN()
IDENTIFIER(v)
COLON(:)
KEYWORD(integer)
RPAREN())
SEMICOLON(;)
KEYWORD(mulai)
IDENTIFIER(write)
LPAREN()
IDENTIFIER(v)
RPAREN())
SEMICOLON(;)
IDENTIFIER(write)
LPAREN()
IDENTIFIER(v)
RPAREN())
SEMICOLON(;)
KEYWORD(selesai)
SEMICOLON(;)
KEYWORD(variabel)
IDENTIFIER(a)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(ok)

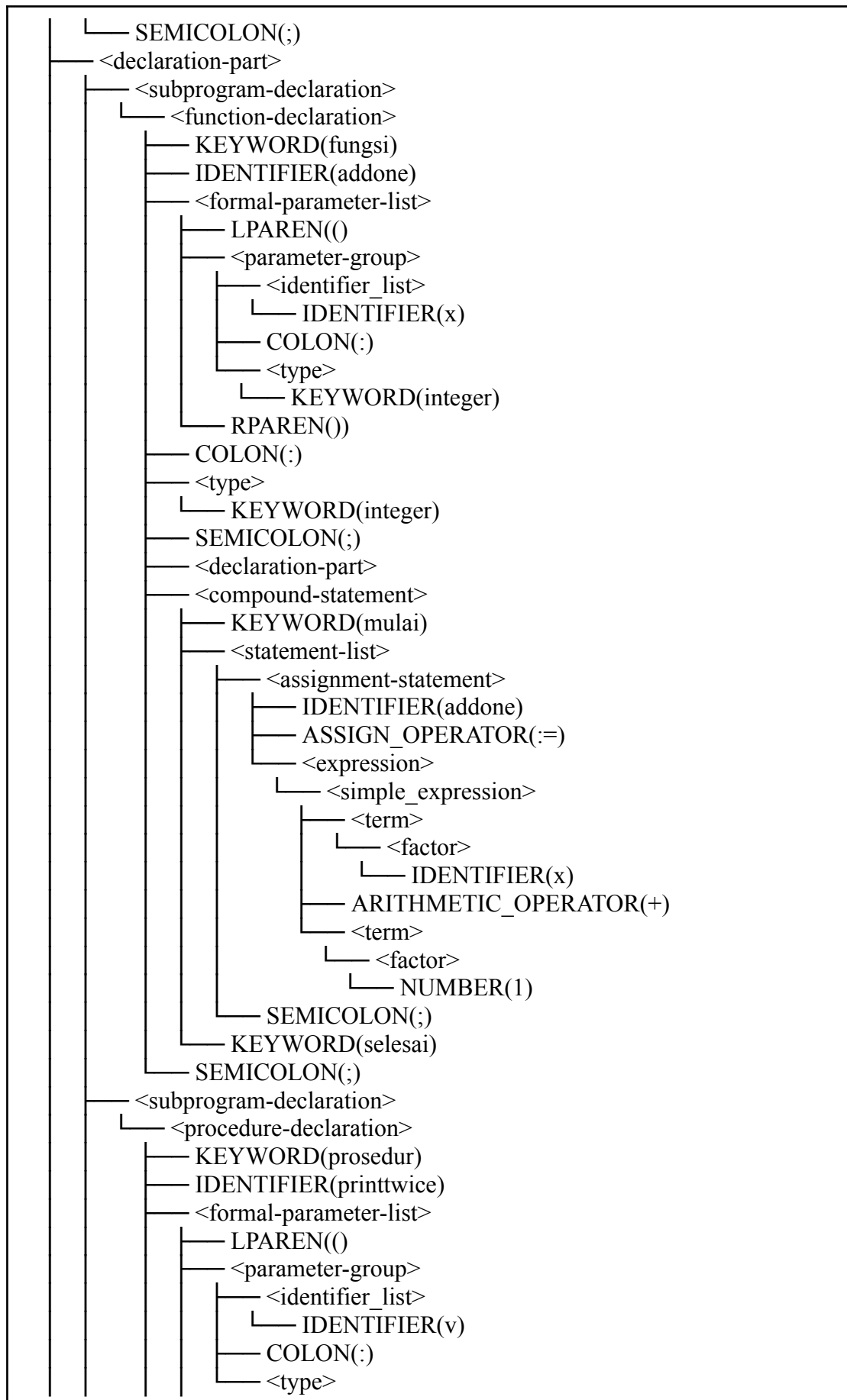
COLON(:)
 KEYWORD(boolean)
 SEMICOLON(;)
 KEYWORD(mulai)
 IDENTIFIER(a)
 ASSIGN_OPERATOR(:=)
 IDENTIFIER(addone)
 LPAREN()
 NUMBER(7)
 RPAREN())
 SEMICOLON(;)
 IDENTIFIER(ok)
 ASSIGN_OPERATOR(:=)
 LPAREN()
 IDENTIFIER(a)
 RELATIONAL_OPERATOR(>)
 NUMBER(5)
 RPAREN())
 LOGICAL_OPERATOR(dan)
 LOGICAL_OPERATOR(tidak)
 LPAREN()
 IDENTIFIER(a)
 RELATIONAL_OPERATOR(=)
 NUMBER(10)
 RPAREN())
 SEMICOLON(;)
 KEYWORD(jika)
 IDENTIFIER(ok)
 KEYWORD(maka)
 IDENTIFIER(printtwice)
 LPAREN()
 IDENTIFIER(a)
 RPAREN())
 KEYWORD(selain_itu)
 IDENTIFIER(write)
 LPAREN()
 NUMBER(0)
 RPAREN())
 SEMICOLON(;)
 KEYWORD(selesai)
 DOT(.)

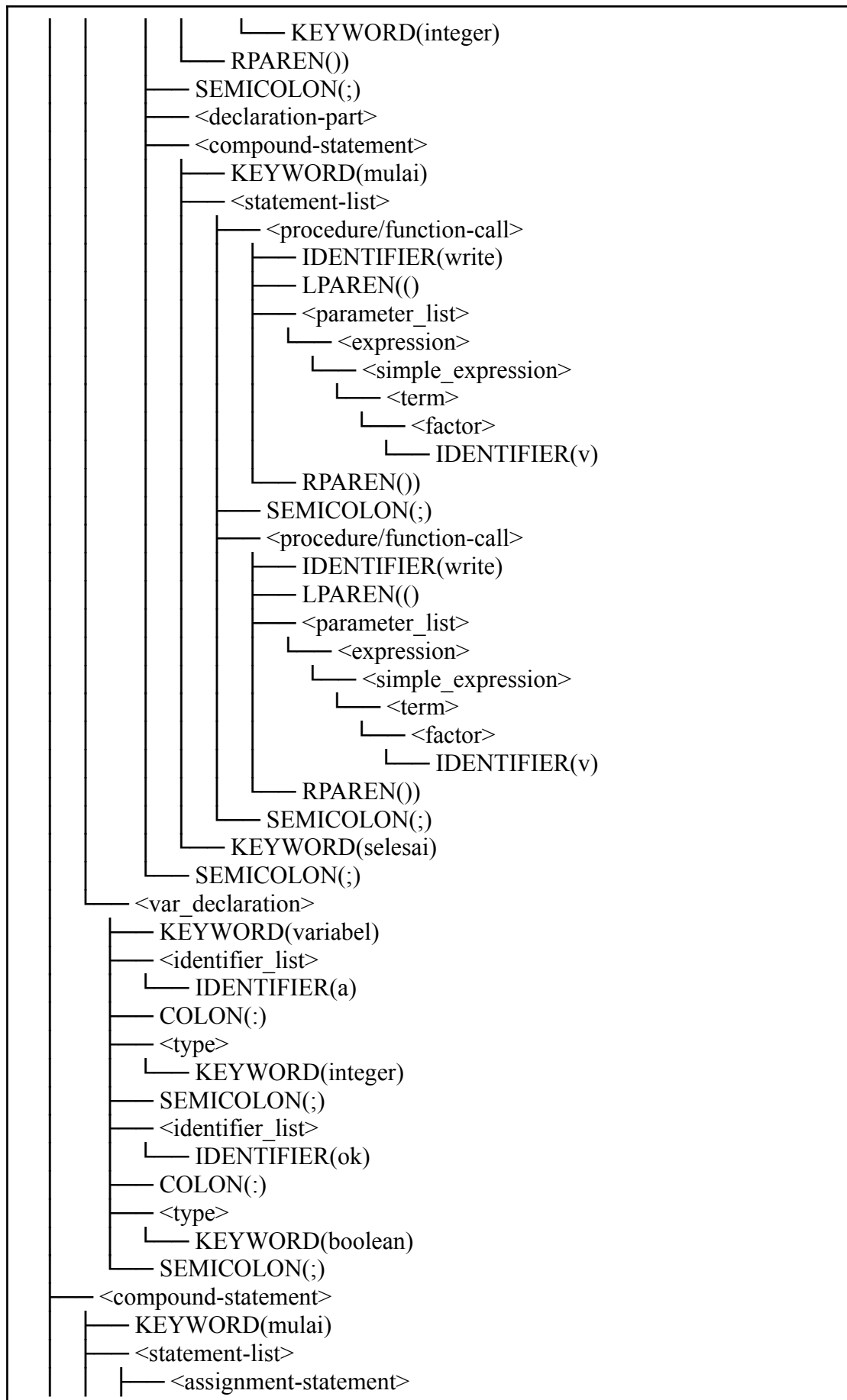
Tree

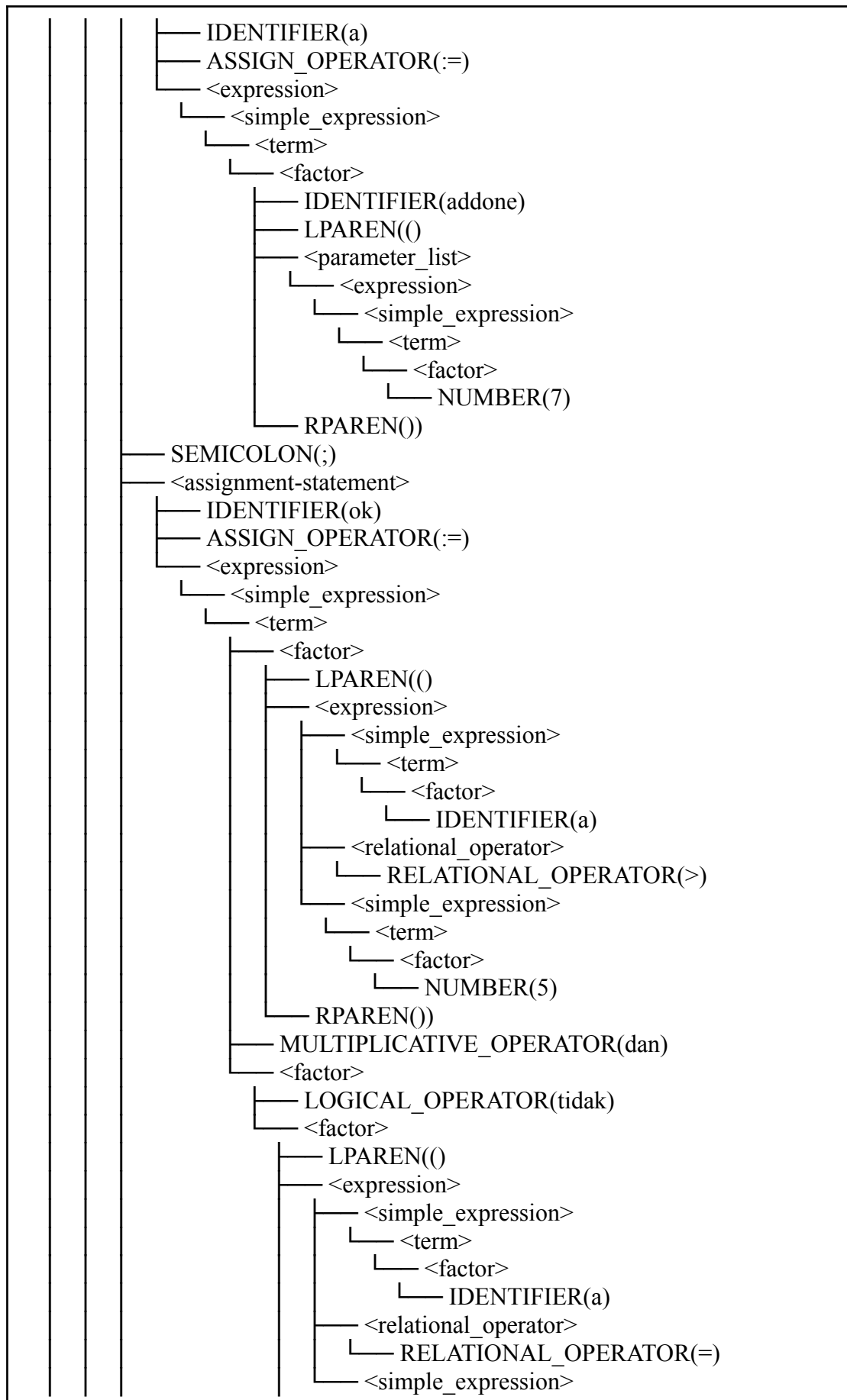
```

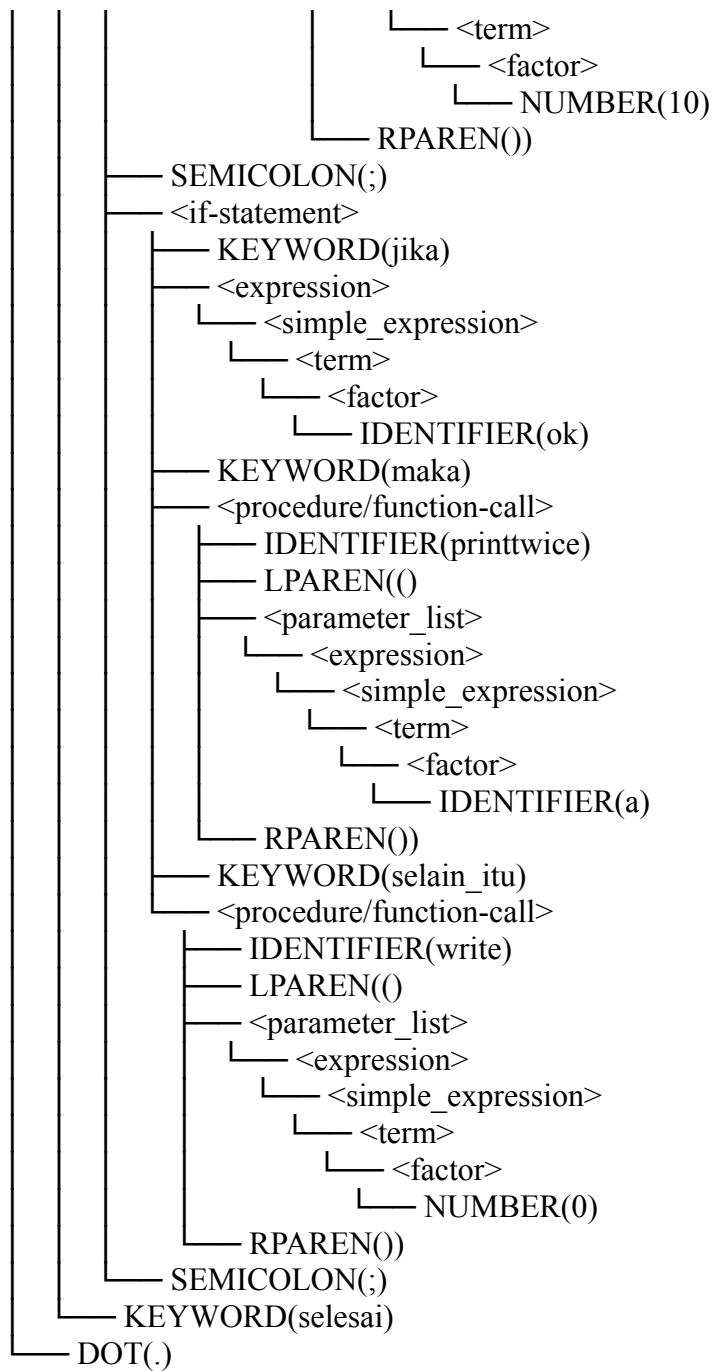
<program>
├── <program_header>
│   ├── KEYWORD(program)
│   └── IDENTIFIER(test5)

```









KESIMPULAN DAN SARAN

Parser merupakan analisis tingkat kedua (analisis *syntax*) dari proses compiling yang mengambil *token-token* dari lexer untuk kemudian diperiksa dan disusun menjadi *parse tree* untuk diproses di tahap berikutnya.

Dari pengujian yang telah dilalui, parser berbasis Recursive Descent sudah cukup kompeten dalam memproses token-token kode program berbahasa Pascal-S menjadi sebuah Parse Tree yang sesuai dengan *grammar* yang didefinisikan.

Pengerjaan Tugas Besar Milestone 2 Teori Berbahasa Formal Otomata memperluas wawasan mengenai pemanfaatan Context-Free Grammar (CFG) dan implementasinya dalam sebuah parser. Dalam segi teknis, kami menjadi paham tentang tahap analisis *syntax* dalam proses kompilasi, yang sebelumnya merupakan hal abstrak, kini menjadi lebih nyata melalui visualisasi *Parse Tree*. Dalam segi non-teknis, dari kami, sebaiknya pengerjaan dilakukan dengan pembagian tugas yang jelas dan memperbanyak pertemuan antara anggota kelompok agar lancar dalam pengerjaan. Kami juga mempelajari pentingnya manajemen waktu di tengah banyak tugas besar.

LAMPIRAN

- Link Github Release Repository:
<https://github.com/pixelatedbus/BCP-Tubes-IF2224/releases/tag/v0.2.1>
- Diagram *language syntax* terdapat di folder doc/ dalam repository dalam bentuk markdown.

Pembagian Tugas	
Julius Arthur (13523030)	Type, array-type, range, procedure/function call, parameter list, additive operator, multiplicative operator
Samuel Gerrard Hamonangan Girsang (13523064)	program_header, procedure_declaration, function declaration, formal parameter list, statement_list, assignment_statement, if_statement, factor, relational_operator
Nadhif Al-Rozin (13523076)	while_statement, for_statement, const_declaration, type_declaration, var_declaration, simple_expression, term
Lutfi Hakim Yusra (13523084)	type_spec, document

-

REFERENSI

- <https://www.bottlecaps.de/> - Language Syntax