IF2224 - Teori Bahasa Formal dan Automata

Laporan Milestone 3 Tugas Besar 1

Semantic 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

LANDASAN TEORI

1. Analisis Semantik

Analisis semantik merupakan tahap pada kompilasi setelah analisis sintaksis. Sebuah parser memastikan bahwa kode memenuhi aturan produksi, tetapi masih mungkin menerima program yang tidak benar secara logika. Maka, tujuan analisis semantik adalah memastikan program memiliki makna yang benar. Tahap ini melakukan pemeriksaan terhadap konsistensi tipe data, deklarasi variabel dan fungsi, lingkup program, serta *control flow.* Analisis *control flow* merupakan proses verifikasi alur eksekusi program agar sejalan dengan peraturan bahasa pemrograman.

2. Attributed Grammar

Pada analisis semantik, digunakan Attributed Grammar, yakni sebuah *Context-Free Grammar* yang juga dilengkapi atribut dan aturan semantik.

3. Symbol Table

Symbol Table menyimpan informasi mengenai *identifier* yang dideklarasikan program, seperti nama, tipe data, hingga lingkup. Pada kompilator ini, *symbol table* diimplementasikan menggunakan struktur data *stack.* Informasi yang ditampung pada *symbol table* digunakan juga untuk mengkalkulasi *offset* memori. Pada implementasi *semantic analyzer* PASCAL-S, terdapat tiga jenis tabel:

1. Tabel *tab*: tabel yang menyimpan informasi umum tentang *identifier,* seperti *const,* variable, prosedur, fungsi, dan tipe dasar. Objek pada *tab* diisi terlebih dahulu oleh *keyword* yang dikhususkan dan telah terdefinisi, diikuti objek-objek identifier yang dideklarasikan oleh kode program. Atribut pada tabel *tab* meliputi:
    a. name: nama identifier dari objek
    b. obj: tipe objek identifier dari objek, seperti variabel, tipe, prosedur
    c. typ: tipe dasar dari identifier, seperti boolean, char, real
    d. ref: link ke sebuah array atau blok untuk tipe array atau rekaman
    e. nrm: menentukan pass-by-value atau pass-by-reference
    f. lev: Tingkat *lexical level* deklarasi objek
    g. adr: Nilai konstanta atau offset variabel terhadap stack frame, alamat akan diisi saat tahap berikutnya
2. Block tab: Menyimpan informasi mengenai blok prosedur. Tabel ini dapat menampung deklarasi prosedur atau fungsi, dan juga deklarasi tipe rekaman. Atribut pada tabel *btab* meliputi:
    a. blocks: indeks entri block
    b. last: pointer ke identifier terakhir pada prosedur atau rekaman block
    c. lpar: pointer ke parameter terakhir pada block tersebut
    d. psze: total ukuran parameter block
    e. vsze: total ukuran variabel lokal block
3. Array tab: Digunakan untuk menyimpan informasi struktur data array. Tabel ini dapat menampung deklarasi tipe array dengan tipe apapun. Atribut pada tabel *atab* meliputi:
    a. arrays: indeks entri array

b. xtyp: tipe indeks array (Integer)
c. etyp: tipe elemen array, dapat berupa rekaman atau array
d. eref: tipe elemen array jika merupakan tipe yang *user-defined*
e. low: batas bawah indeks array
f. high: batas atas indeks array
g. elsz: ukuran satu elemen array
h. size: total ukuran array

4. Abstract Syntax Tree

Abstract Syntax Tree merupakan representasi pohon sederhana dari *parse tree*. Pembuatan AST meliputi penyederhanaan *parse tree* yang sebelumnya merepresentasikan keseluruhan token yang ada pada kode hingga menjadi representasi alur program dari kode yang tersedia. Selanjutnya, pada tahap analisis semantik, AST ditelusuri menggunakan konsep *Visitor*. Setiap *node* pada AST akan mengunjungi *child node* secara rekursif (*Depth First*). Kemudian fungsi visit akan mengakses dan memperbarui informasi *symbol table*. Kemudian akan ditambahkan anotasi *node* tersebut agar dihasilkan *Decorated AST*.

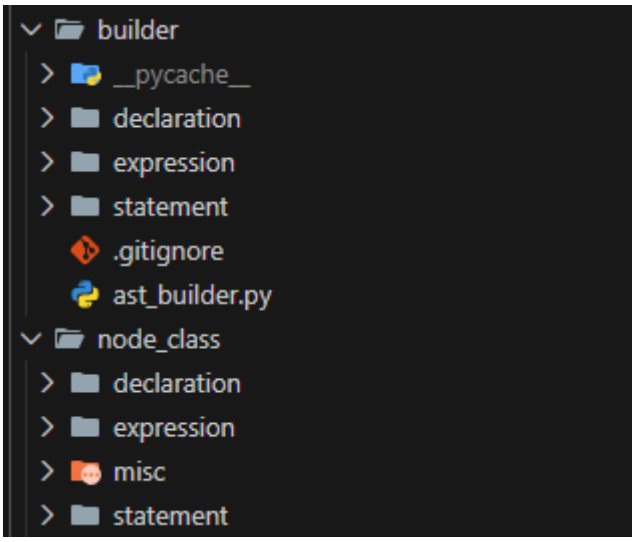
5. Pengecekan Tipe dan Lingkup

Mekanisme pengecekan dilakukan dengan mengunjungi setiap node AST. Saat kompilator memasuki blok kode baru, *scope* baru akan di *push* ke *stack symbol table*. Setiap deklarasi variabel dan fungsi akan dimasukkan ke tabel. Maka, setiap ditemukan penggunaan *identifier*, kompilator akan melakukan pencarian pada *symbol table* untuk memastikan *identifier* telah dideklarasikan dan dalam lingkup yang valid. Hal ini memastikan alur logika dari program itu memiliki makna yang jelas.

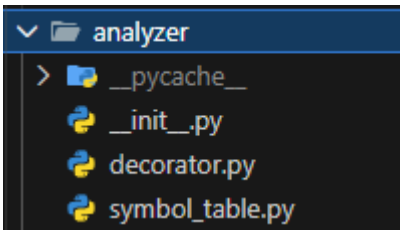
PERANCANGAN DAN IMPLEMENTASI

1. Perancangan

Tahap ini direpresentasikan dalam tiga proses: pembentukan AST (*Abstract Syntax Tree*), pengisian *symbol table*, dan pendekorasian AST. Pembentukan AST dilakukan dengan melakukan traversal terhadap *parse tree* oleh sebuah fungsi *builder* eksternal yang melakukan pemrosesan khusus sesuai dengan tipe node yang dilewati. Untuk setiap tipe node yang dilewati pada pohon, *builder* menempelkan pada pohon dengan metode yang sesuai dengan fungsi yang terhubung dengan tipe node yang ada. Setiap fungsi dan setiap tipe kelas node didefinisikan secara *hard-coded* agar perilaku *builder* terdefinisi sesuai dengan spesifikasi. Definisi perilaku *builder* per tipe node ditampung pada direktori 'builder', dan definisi tiap kelas node ditampung pada direktori 'node_class'.



Setelah pembangunan AST, terdapat sebuah kelas bernama ASTDecorator yang berperan sebagai *visitor* terhadap setiap node AST, melakukan traversal untuk mengisi *symbol table* dan menghubungkan setiap pemanggilan dan deklarasi objek pada kode dengan *symbol table* yang sedang dibangun tersebut. Dalam pembangunan ini, semua variabel dan objek pada kode ditambahkan ke *symbol table* dan juga di-*lookup* untuk memverifikasi konsistensi penggunaan objek tersebut pada kode. Konsistensi dijaga dengan melihat tipe, kedalaman level, dan deklarasi objek sebelumnya.



2. Implementation

a. Node Class

ast_node.py

```python
class ASTNode:
    def __init__(self, children=None):
        if children is None:
            children = []
        self.children = children

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

    def to_string(self, prefix="", is_last=True):
        display_name = str(self)

        if prefix == "":
            result = display_name + "\n"
        else:
            connector = "└── " if is_last else "├── "
            result = prefix + connector + display_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):
        if hasattr(self, 'name'):
            return self.name
        else:
            return self.__class__.__name__
```

Penjelasan

ASTNode merupakan representasi *node* pada *Abstract Syntax Tree*. Kelas ini merupakan *parent* dari berbagai jenis *node* lainnya yang mengimplementasikan fungsi __str__ nya masing - masing. Kelas ini memiliki atribut *child* yang merupakan sebuah array yang berisikan seluruh anak pada *node* ini. Terdapat fungsi

to_string yang digunakan untuk mencetak pohon dalam bentuk yang mudah dipahami.

block_node.py

```python
from .ast_node import ASTNode

class BlockNode(ASTNode):
    def __init__(self):
        super().__init__()
        self.name = "Block"
```

Penjelasan

Merupakan representasi blok program di AST yang berisi deklarasi serta kumpulan *statement* (*compound statement*). Kelas ini memiliki atribut nama dan mewarisi kelas ASTNode.

declarations_node.py

```python
from .ast_node import ASTNode

class DeclarationsNode(ASTNode):
    def __init__(self):
        super().__init__()
        self.name = "Declarations"

    def __str__(self):
        return "Declarations"
```

Penjelasan

Merepresentasikan deklarasi secara umum. Kelas ini memiliki atribut name mewarisi kelas ASTNode.

program_node.py

```python
from .ast_node import ASTNode

class ProgramNode(ASTNode):
    def __init__(self, name):
        super().__init__()
        self.name = name

    def __str__(self):
        return f"ProgramNode(name: {self.name})"
```

Penjelasan

Merepresentasikan node Program pada AST dan merupakan akar dari AST. Kelas ini memiliki atribut name dan mewarisi kelas ASTNode.

const_decl_node.py

```python
from ..misc.ast_node import ASTNode

class ConstDeclNode(ASTNode):
    def __init__(self, name: str, value):
        super().__init__()
        self.name = name
        self.value = value  # Expression node

    def __str__(self):
        return f"ConstDecl Node(name={self.name}, value={self.value})"

    def evaluate(self):
        print(f"Evaluating ConstDecl Node: {self.name}")
        return None
```

Penjelasan

Merupakan representasi node deklarasi konstan pada AST. Kelas ini menyimpan nama serta value dari konstan yang dideklarasikan.

function_decl_node.py

```python
from ..misc.ast_node import ASTNode

class FunctionDeclNode(ASTNode):
    def __init__(self, name: str, parameters=None, return_type: str = None, declarations=None, body=None):
        super().__init__()
        self.name = name
        self.parameters = parameters if parameters else []  # List of parameter nodes
        self.return_type = return_type  # Return type (integer, real, etc.)
        self.declarations = declarations if declarations else []  # Local declarations
        self.body = body  # Statement node (compound statement)

    def __str__(self):
        return f"FunctionDecl Node(name={self.name}, params={len(self.parameters)}, return_type={self.return_type})"

    def evaluate(self):
        print(f"Evaluating FunctionDecl Node: {self.name}")
        return None
```

Penjelasan

Merupakan representasi node deklarasi fungsi pada kode. Kelas ini menyimpan nama, parameter, serta body dari fungsi tersebut (bertipe *compound statement*).

parameter_node.py

```python
from ..misc.ast_node import ASTNode

class ParameterNode(ASTNode):
    def __init__(self, name: str, param_type: str, is_var: bool = False):
        super().__init__()
        self.name = name
        self.param_type = param_type  # Parameter type (integer, real, etc.)
        self.is_var = is_var  # True if parameter is passed by reference (var)

    def __str__(self):
        var_str = "var " if self.is_var else ""
        return f"Parameter Node({var_str}{self.name}: {self.param_type})"

    def evaluate(self):
        print(f"Evaluating Parameter Node: {self.name}")
        return None
```

Penjelasan

Merupakan representasi node parameter pada AST. Kelas ini merujuk pada sebuah parameter yang dideklarasikan pada sebuah fungsi/ prosedur. Pada kelas ini disimpan nama dan tipe parameter.

procedure_decl_node.py

```python
from ..misc.ast_node import ASTNode

class ProcedureDeclNode(ASTNode):
    def __init__(self, name: str, parameters=None, declarations=None, body=None):
        super().__init__()
        self.name = name
        self.parameters = parameters if parameters else []  # List of parameter nodes
        self.declarations = declarations if declarations else []  # Local declarations
        self.body = body  # Statement node (compound statement)

    def __str__(self):
        return f"ProcedureDecl Node(name={self.name}, params={len(self.parameters)}, decls={len(self.declarations)})"

    def evaluate(self):
        print(f"Evaluating ProcedureDecl Node: {self.name}")
        return None
```

Penjelasan

Merupakan representasi node deklarasi prosedur pada kode. Kelas ini menyimpan nama, parameter, serta body dari fungsi tersebut (bertipe *compound statement*).

type_decl_node.py

```python
from ..misc.ast_node import ASTNode

class TypeDeclNode(ASTNode):
    def __init__(self, name: str, type_spec):
        super().__init__()
        self.name = name
        self.type_spec = type_spec  # Type specification (simple_type, range_type, etc.)

    def __str__(self):
        return f"TypeDecl Node(name={self.name}, type_spec={self.type_spec})"

    def evaluate(self):
        print(f"Evaluating TypeDecl Node: {self.name}")
        return None
```

Penjelasan

Merepresentasikan sebuah deklarasi tipe bentukan di AST, hasil deklarasi tipe pada bagian deklarasi program. Kelas ini menyimpan nama serta type_spec (simple_type seperti integer, real, boolean) hingga array dan range type.

var_decl_node.py

```python
from ..misc.ast_node import ASTNode

class VarDeclNode(ASTNode):
    def __init__(self, name: str, var_type: str):
        super().__init__()
        self.name = name
        self.var_type = var_type

    def __str__(self):
        return f"VarDecl Node(name={self.name}, type={self.var_type})"

    def  evaluate(self):
        print("Evaluating VarDecl Node")
        return None
```

Penjelasan

Merepresentasikan sebuah deklarasi variabel pada program. Kelas ini menyimpan nama serta tipe dari variabel yang dideklarasikan.

array_access_node.py

```python
from .factor_node import FactorNode

class ArrayAccessNode(FactorNode):
    """
    Represents array element access.
    Example: score[i], matrix[row][col]
    """
    def __init__(self, array_name, index):
        super().__init__()
        self.array_name = array_name  # String: name of the array
        self.index = index  # ExpressionNode: the index expression

        self.add_child(index)

    def __str__(self):
        return f"ArrayAccessNode({self.array_name}[{self.index}])"
```

Penjelasan

Merepresentasikan sebuah node yang mengakses sebuah array dengan ‘[ ]’. Kelas ini menyimpan nama array serta index dalam bentuk sebuah ExpressionNode. Kelas ini memiliki sebuah anak yang merupakan sebuah node yang merepresentasikan index.

expression_node.py

```python
from ..misc.ast_node import ASTNode

class ExpressionNode(ASTNode):
    """
    Represents a binary expression with a relational operator.
    Example: a = b, x < 5, y >= 10
    """
    def __init__(self, left_operand, operator=None, right_operand=None):
        super().__init__()
        self.left_operand = left_operand  # SimpleExpressionNode
        self.operator = operator  # String: '=', '<>', '<', '<=', '>', '>='
        self.right_operand = right_operand  # SimpleExpressionNode (optional)

        self.add_child(left_operand)
        if operator and right_operand:
            self.add_child(right_operand)

    def is_comparison(self):
        """Returns True if this is a comparison expression, False if just a simple expression."""
        return self.operator is not None

    def __str__(self):
        if self.is_comparison():
            return f"ExpressionNode({self.left_operand} {self.operator} {self.right_operand})"
        return f"ExpressionNode({self.left_operand})"
```

Penjelasan

Merepresentasikan sebuah ekspresi biner pada AST. Maka, kelas ini menyimpan nilai sebelah kiri dari operan, operan, serta nilai sebelah kanan dari operan. Anak dari node ini merupakan SimpleExpressionNode yang merepresentasikan sebelah kiri dan kanan dari operan.

factor_node.py

```python
from ..misc.ast_node import ASTNode

class FactorNode(ASTNode):
    """
    Base class for all factor nodes (primary expressions).
    Factors are the atomic units of expressions.
    """
    def __init__(self):
        super().__init__()

    def __str__(self):
        return "FactorNode(base)"
```

Penjelasan

Merupakan kelas dasar untuk seluruh node factor.

function_call_node.py

```python
from .factor_node import FactorNode

class FunctionCallNode(FactorNode):
    """
    Represents a function call in an expression.
    Example: calculateSum(a, b), max(x, y, z)
    """
    def __init__(self, function_name, arguments=None):
        super().__init__()
        self.function_name = function_name  # String: name of the function
        self.arguments = arguments if arguments is not None else []  # List of ExpressionNode

        # Add arguments as children
        for arg in self.arguments:
            self.add_child(arg)

    def add_argument(self, argument):
        """Add an argument to the function call."""
        self.arguments.append(argument)
        self.add_child(argument)

    def __str__(self):
        args_str = ", ".join(str(arg) for arg in self.arguments)
        return f"FunctionCallNode({self.function_name}({args_str}))"
```

Penjelasan

Merupakan representasi pemanggilan sebuah fungsi pada ekspresi. Pada kelas ini, disimpan nama (string) serta larik yang berisikan argumen pemanggilan fungsi. Argumen yang disimpan merupakan objek dari kelas ExpressionNode. Node ini memiliki beberapa anak yang masing - masing merujuk pada argumen pemanggilan fungsi.

identifier_node.py

```python
from .factor_node import FactorNode

class IdentifierNode(FactorNode):
    """
    Represents a variable reference.
    Example: x, myVar, counter
    """
    def __init__(self, name):
        super().__init__()
        self.name = name  # String: the identifier name

    def __str__(self):
        return f"IdentifierNode({self.name})"
```

Penjelasan

Merepresentasikan referensi sebuah variabel/ *identifier*. Kelas ini menyimpan nama dari variabel/ *identifier* yang disimpan.

literal_node.py

```python
from .factor_node import FactorNode

class LiteralNode(FactorNode):
    """
    Represents a literal value (number, character, or string).
    Example: 42, 'a', "hello"
    """
    def __init__(self, value, literal_type):
        super().__init__()
        self.value = value  # The actual value
        self.literal_type = literal_type  # 'NUMBER', 'CHAR_LITERAL', 'STRING_LITERAL'

    def __str__(self):
        return f"LiteralNode({self.literal_type}: {self.value})"


class NumberLiteral(LiteralNode):
    """Represents a numeric literal."""
    def __init__(self, value):
        super().__init__(value, 'NUMBER')

    def __str__(self):
        return f"NumberLiteral({self.value})"


class CharLiteral(LiteralNode):
    """Represents a character literal."""
    def __init__(self, value):
        super().__init__(value, 'CHAR_LITERAL')

    def __str__(self):
        return f"CharLiteral({self.value})"


class StringLiteral(LiteralNode):
    """Represents a string literal."""
    def __init__(self, value):
        super().__init__(value, 'STRING_LITERAL')

    def __str__(self):
        return f"StringLiteral({self.value})"
```

Penjelasan

Merepresentasikan sebuah literal pada AST. Literal dapat bertipe angka, karakter, atau string. Pada node ini disimpan nilai serta tipe dari literal. Terdapat 3 kelas turunan dari LiteralNode yang masing - masing merepresentasikan ketiga jenis tipe literal yang ada.

parenthesized_expression_node.py

```python
from .factor_node import FactorNode

class ParenthesizedExpressionNode(FactorNode):
    """
    Represents a parenthesized expression for grouping.
    Example: (a + b), ((x * y) + z)
    """
    def __init__(self, expression):
        super().__init__()
        self.expression = expression  # ExpressionNode: the inner expression

        self.add_child(expression)

    def __str__(self):
        return f"ParenthesizedExpressionNode(({self.expression}))"
```

Penjelasan

Merepresentasikan sebuah ekspresi yang diawali serta diakhiri sebuah kurung. Node ini memiliki sebuah anak yang merupakan ekspresi yang ada di dalam kurung tersebut. Anak ini merupakan objek dari kelas ExpressionNode.

simple_expression_node.py

```python
from ..misc.ast_node import ASTNode

class SimpleExpressionNode(ASTNode):
    """
    Represents an additive expression or logical OR expression.
    Can be unary (with sign) or a chain of terms with additive operators.
    Example: +a, a + b - c, x atau y
    """
    def __init__(self, sign=None):
        super().__init__()
        self.sign = sign  # Optional '+' or '-' for unary expressions
        self.terms = []  # List of TermNode
        self.operators = []  # List of operators: '+', '-', 'atau'

    def add_term(self, term, operator=None):
        """Add a term to the expression. Operator is the one that comes BEFORE this term."""
        if len(self.terms) > 0 and operator is None:
            raise ValueError("Operator required for additional terms")

        self.terms.append(term)
        self.add_child(term)

        if operator:
            self.operators.append(operator)

    def is_unary(self):
        """Returns True if this is a unary expression with a sign."""
        return self.sign is not None and len(self.terms) == 1

    def __str__(self):
        if self.is_unary():
            return f"SimpleExpressionNode({self.sign}{self.terms[0]})"

        result = str(self.terms[0]) if self.terms else ""
        for i, op in enumerate(self.operators):
            if i + 1 < len(self.terms):
                result += f" {op} {self.terms[i + 1]}"
        return f"SimpleExpressionNode({result})"
```

Penjelasan

Merepresentasikan ekspresi penjumlahan atau logika OR. Ekspresi ini dapat berbentuk uner ataupun penjumlahan berantai. Kelas ini menyimpan larik yang berisikan term yang merupakan objek dari sebuah TermNode.

term_node.py

```python
from ..misc.ast_node import ASTNode

class TermNode(ASTNode):
    """
    Represents a multiplicative expression or logical AND expression.
    A chain of factors with multiplicative operators.
    Example: a * b, x / y mod z, flag dan flag2
    """
    def __init__(self):
        super().__init__()
        self.factors = []  # List of FactorNode
        self.operators = []  # List of operators: '*', '/', 'bagi', 'mod', 'dan'

    def add_factor(self, factor, operator=None):
        """Add a factor to the term. Operator is the one that comes BEFORE this factor."""
        if len(self.factors) > 0 and operator is None:
            raise ValueError("Operator required for additional factors")

        self.factors.append(factor)
        self.add_child(factor)

        if operator:
            self.operators.append(operator)

    def is_single_factor(self):
        """Returns True if this term contains only one factor."""
        return len(self.factors) == 1

    def __str__(self):
        if not self.factors:
            return "TermNode(empty)"

        result = str(self.factors[0])
        for i, op in enumerate(self.operators):
            if i + 1 < len(self.factors):
                result += f" {op} {self.factors[i + 1]}"
        return f"TermNode({result})"
```

Penjelasan

Merepresentasikan satu “term” dalam ekspresi, sebuah rangkaian factor yang dihubungkan operator perkalian/pembagian/modulo atau logika AND (dan). Atribut:
factors: list FactorNode. Masing-masing factor bisa berupa literal, identifier, unary expression, array access, function call, dll.
operators: list string operator yang berada di antara factors: '*', '/', 'bagi' (div), 'mod', 'dan'.
children : setiap factor ditambahkan via add_child(factor).

unary_expression_node.py

```python
from .factor_node import FactorNode

class UnaryExpressionNode(FactorNode):
    """
    Represents a unary expression (logical NOT).
    Example: tidak flag, tidak (a = b)
    """
    def __init__(self, operator, operand):
        super().__init__()
        self.operator = operator  # String: 'tidak' (not)
        self.operand = operand  # FactorNode: the operand

        self.add_child(operand)

    def __str__(self):
        return f"UnaryExpressionNode({self.operator} {self.operand})"
```

Penjelasan

Merepresentasikan ekspresi uner (NOT). Kelas ini menyimpan operator (‘tidak’) serta sebuah operand dalam bentuk FactorNode. Kelas ini memiliki anak sebuah operand.

assignment_sstatement_node.py

```python
from ..misc.ast_node import ASTNode

class AssignmentStatementNode(ASTNode):
    def __init__(self, variable_name: str, value):
        super().__init__()
        self.variable_name = variable_name
        self.value = value

    def __str__(self):
        return f"AssignmentStatement Node(variable_name={self.variable_name}, value={self.value})"

    def evaluate(self):
        print(f"Evaluating AssignmentStatement Node: {self.variable_name} = {self.value}")
        return None
```

Penjelasan

Merepresentasikan assignment pada AST. Kelas ini menyimpan nama serta nilai dari variabel tersebut.

compound_statement_node.py

```python
from ..misc.ast_node import ASTNode

class CompoundStatementNode(ASTNode):
    def __init__(self):
        super().__init__()

    def evaluate(self):
        print("Evaluating CompoundStatement Node")
        return None
```

Penjelasan

Merepresentasikan kumpulan dari statement yang ada, di antara *begin* dan *end*.

for_statement.py

```python
from ..misc.ast_node import ASTNode

class ForStatementNode(ASTNode):
    def __init__(self, variable: str, start: int, end: int, body: list):
        super().__init__()
        self.name = "for_statement"
        self.variable = variable
        self.start = start
        self.end = end
        self.body = body

        if body:
            self.add_child(body)

    def __str__(self):
        return f"ForStatement Node(variable={self.variable}, start={self.start}, end={self.end})"

    def evaluate(self):
        print("Evaluating ForStatement Node")
        return None
```

Penjelasan

Merepresentasikan for loop pada AST. Kelas ini menyimpan variabel *increment*, nilai awal dan akhir variabel inkremen, dan body (isi dari loop, berbentuk node statement). Node ini akan memiliki body sebagai anak.

if_statement.py

```python
from ..misc.ast_node import ASTNode

class IfStatementNode(ASTNode):
    def __init__(self, compare, ifbody, elsebody=None):
        super().__init__()
        self.name = "if_statement"
        self.compare = compare
        self.ifbody = ifbody
        self.elsebody = elsebody

    def __str__(self):
        return f"If Statement Node(compare={self.compare}, ifbody={self.ifbody}, elsebody={self.elsebody})"

    def evaluate(self):
        print("Evaluating If Statement Node")
        return None
```

Penjelasan

Merepresentasikan jika-maka. Kelas ini menyimpan ekspresi komparasi, serta isi dari jika-maka, yang berbentuk statement/ compound-statement.

procedure_function_call_node.py

```python
from ..misc.ast_node import ASTNode

class procedureFunctionCallNode(ASTNode):
    def __init__(self, name: str, arguments: list):
        super().__init__()
        self.name = name
        self.arguments = arguments

    def __str__(self):
        args_str = ', '.join(str(arg) for arg in self.arguments)
        return f"ProcedureFunctionCall Node(name={self.name}, arguments=[{args_str}])"

    def evaluate(self):
        print("Evaluating ProcedureFunctionCall Node")
        return None
```

Penjelasan

Merepresentasikan pemanggilan fungsi atau prosedur. Kelas ini menyimpan nama serta argumen pemanggilan.

statement_list_node.py

```python
from ..misc.ast_node import ASTNode

class StatementListNode(ASTNode):
    def __init__(self):
        super().__init__()
```

Penjelasan

Merepresentasikan list parameter yang ada dalam sebuah fungsi/prosedur.

statement_node.py

```python
from ..misc.ast_node import ASTNode

class StatementNode(ASTNode):
    def __init__(self):
        super().__init__()
        self.statement_type = "statement"

    def evaluate(self):
        print("Evaluating Statement Node")
        return None
```

Penjelasan

Merupakan kelas generik untuk seluruh tipe statement.

while_statement_node.py

```python
from ..misc.ast_node import ASTNode

class WhileStatementNode(ASTNode):
    def __init__(self, condition, body):
        super().__init__()
        self.name = "while_statement"
        self.condition = condition
        self.body = body

        if body:
            self.add_child(body)

    def __str__(self):
        return f"WhileStatement Node(condition={self.condition})"

    def evaluate(self):
        print("Evaluating WhileStatement Node")
        return None
```

Penjelasan

Merupakan representasi *while-loop*. Kelas ini menyimpan ekspresi kondisi dari pengulangan dan statement dari isi pengulangan. Objek dari kelas ini akan memiliki sebuah anak yang merupakan sebuah node statement.

b. Builder

ast_builder.py

```python
from ..node_class.misc.program_node import ProgramNode
from ..node_class.misc.declarations_node import DeclarationsNode
from ..node_class.misc.block_node import BlockNode
from .statement.statement_builder import build_statements
from .declaration.declaration_builder import build_declaration

def build_ast(parse_tree):
    if parse_tree.name != "<program>":
        raise ValueError(f"Expected <program>, got {parse_tree.name}")

    program_header_node = parse_tree.children[0]
    program_name = program_header_node.children[1].name.split('(')[1][:-1]
```

```
    program_node = ProgramNode(program_name)

    declarations_node = DeclarationsNode()

    block_node = BlockNode()

    for child in parse_tree.children:
        if child.name == "<declaration-part>":
            declaration_nodes = build_declaration(child)
            for declaration_node in declaration_nodes:
                declarations_node.add_child(declaration_node)
        elif child.name == "<compound-statement>":
            statement_list = child.children[1]
            statements = build_statements(statement_list)
            for statement in statements:
                block_node.add_child(statement)

    program_node.add_child(declarations_node)
    program_node.add_child(block_node)

    return program_node
```

Penjelasan

Kelas builder utama yang dimulai dari pembangunan program Node hingga ke Node pada tree yang lebih dalam

const_builder.py

```
import sys
import os
sys.path.append(os.path.join(os.path.dirname(__file__), '..',
'..'))

from node_class.declaration.const_decl_node import
```

```
ConstDeclNode
from .helpers import extract_simple_value


def build_const_declaration(parse_node):
    i = 0
    children = parse_node.children

    if children[i].name.startswith("KEYWORD"):
        i += 1

    const_name = "unknown"
    if i < len(children) and
children[i].name.startswith("IDENTIFIER"):
        const_name =
children[i].name.split("(")[1].rstrip(")")
        i += 1

    if i < len(children) and
children[i].name.startswith("RELATIONAL_OPERATOR"):
        i += 1

    expression_value = None
    if i < len(children) and children[i].name ==
"<expression>":
        expression_value = extract_simple_value(children[i])
        i += 1

    return ConstDeclNode(const_name, expression_value)
```

Penjelasan

Kelas builder yang memproses node <const_declaration> untuk membangun AST, yaitu deklarasi konstanta pada program.

declaration_builder.py

```
from .const_builder import build_const_declaration
from .type_builder import build_type_declaration
```

```
from .var_builder import build_var_declaration
from .procedure_builder import build_procedure_declaration
from .function_builder import build_function_declaration


def build_declaration(parse_node):
    all_declarations = []

    for child in parse_node.children:
        if child.name == "<const_declaration>":
            const_node = build_const_declaration(child)
            all_declarations.append(const_node)

        elif child.name == "<type_declaration>":
            type_node = build_type_declaration(child)
            all_declarations.append(type_node)

        elif child.name == "<var_declaration>":
            var_nodes = build_var_declaration(child)
            all_declarations.extend(var_nodes)

        elif child.name == "<subprogram-declaration>":
            for subchild in child.children:
                if subchild.name ==
"<procedure-declaration>":
                    proc_node =
build_procedure_declaration(subchild)
                    all_declarations.append(proc_node)
                elif subchild.name ==
"<function-declaration>":
                    func_node =
build_function_declaration(subchild)
                    all_declarations.append(func_node)

    return all_declarations
```

Penjelasan

Kelas builder yang memproses isi keseluruhan node <declaration> untuk membangun AST, yaitu kumpulan deklarasi pada program.

function_builder.py

```
import sys
import os
sys.path.append(os.path.join(os.path.dirname(__file__), '..',
'..'))

from node_class.declaration.function_decl_node import
FunctionDeclNode
from .helpers import extract_parameters, extract_type


def build_function_declaration(parse_node):
    from .declaration_builder import build_declaration

    i = 0
    children = parse_node.children

    if children[i].name.startswith("KEYWORD"):
        i += 1

    func_name = "unknown"
    if children[i].name.startswith("IDENTIFIER"):
        func_name =
children[i].name.split("(")[1].rstrip(")")
        i += 1

    parameters = []
    if i < len(children) and children[i].name ==
"<formal-parameter-list>":
        parameters = extract_parameters(children[i])
        i += 1

    if i < len(children) and
children[i].name.startswith("COLON"):
        i += 1

    return_type = "unknown"
    if i < len(children) and children[i].name == "<type>":
        return_type = extract_type(children[i])
```

```
        i += 1

    if i < len(children) and
children[i].name.startswith("SEMICOLON"):
        i += 1

    declarations = []
    if i < len(children) and children[i].name ==
"<declaration-part>":
        declarations = build_declaration(children[i])
        i += 1

    body = None
    if i < len(children) and children[i].name ==
"<compound-statement>":
        body = children[i]
        i += 1

    return FunctionDeclNode(func_name, parameters,
return_type, declarations, body)
```

Penjelasan

penjelasan

helpers.py

```
import sys
import os
sys.path.append(os.path.join(os.path.dirname(__file__), '..',
'..'))

from node_class.declaration.parameter_node import
ParameterNode
```

```python
def extract_identifier_list(parse_node):
    """Extract list of identifiers from <identifier_list>"""
    identifiers = []
    for child in parse_node.children:
        if child.name.startswith("IDENTIFIER"):
            identifier = child.name.split("(")[1].rstrip(")")
            identifiers.append(identifier)
    return identifiers


def extract_type(parse_node):
    """Extract type from <type> node"""
    for child in parse_node.children:
        if child.name.startswith("KEYWORD"):
            return child.name.split("(")[1].rstrip(")")
        elif child.name.startswith("IDENTIFIER"):
            # User-defined type
            return child.name.split("(")[1].rstrip(")")
        elif child.name.startswith("<"):
            # Nested type spec (array, record, etc.)
            return extract_type_spec(child)
    return "unknown"


def extract_type_spec(parse_node):
    """Extract type specification from <simple_type>,
<range_type>, etc."""
    if parse_node.name == "<simple_type>":
        for child in parse_node.children:
            if child.name.startswith("KEYWORD"):
                return child.name.split("(")[1].rstrip(")")

    elif parse_node.name == "<custom_type>":
        # User-defined type name
        for child in parse_node.children:
            if child.name.startswith("IDENTIFIER"):
                return child.name.split("(")[1].rstrip(")")
        return "unknown"

    elif parse_node.name == "<range_type>":
        return "range"
```

```
    elif parse_node.name == "<array_type>":
        # Extract array bounds and element type
        # Structure: larik[low..high] dari element_type
        array_info = {"type": "array"}

        for child in parse_node.children:
            if child.name == "<range>":
                # Extract low and high bounds
                bounds = extract_range_bounds(child)
                array_info["low"] = bounds["low"]
                array_info["high"] = bounds["high"]
            elif child.name == "<type>":
                # Extract element type
                array_info["element_type"] =
extract_type(child)

        return array_info

    elif parse_node.name == "<record_type>":
        # Extract record fields
        fields = extract_record_fields(parse_node)
        return {"type": "record", "fields": fields}

    return "unknown"


def extract_range_bounds(range_node):
    """Extract low and high bounds from <range> node"""
    bounds = {"low": 0, "high": 0}
    numbers = []

    def extract_number_from_node(node):
        """Recursively extract numbers from node"""
        if node.name.startswith("NUMBER"):
            num_str = node.name.split("(")[1].rstrip(")")
            return int(num_str)
        # Numbers might be wrapped in expression nodes
        for child in node.children:
            num = extract_number_from_node(child)
            if num is not None:
                return num
        return None
```

```
    for child in range_node.children:
        num = extract_number_from_node(child)
        if num is not None:
            numbers.append(num)

    if len(numbers) >= 2:
        bounds["low"] = numbers[0]
        bounds["high"] = numbers[1]

    return bounds


def extract_simple_value(parse_node):
    """Extract simple constant value dari <expression>
node"""
    # Untuk konstanta sederhana, cari NUMBER atau
STRING_LITERAL
    if parse_node.name == "<expression>":
        for child in parse_node.children:
            if child.name == "<simple_expression>":
                return extract_simple_value(child)

    elif parse_node.name == "<simple_expression>":
        for child in parse_node.children:
            if child.name == "<term>":
                return extract_simple_value(child)

    elif parse_node.name == "<term>":
        for child in parse_node.children:
            if child.name == "<factor>":
                return extract_simple_value(child)

    elif parse_node.name == "<factor>":
        for child in parse_node.children:
            if child.name.startswith("NUMBER"):
                return child.name.split("(")[1].rstrip(")")
            elif child.name.startswith("STRING_LITERAL"):
                return child.name.split("(")[1].rstrip(")")
            elif child.name.startswith("CHAR_LITERAL"):
                return child.name.split("(")[1].rstrip(")")
```

```
    return None


def extract_parameters(parse_node):
    parameters = []

    for child in parse_node.children:
        if child.name == "<parameter-group>":
            identifiers = []
            param_type = "unknown"
            is_var = False

            for subchild in child.children:
                if subchild.name == "<identifier_list>":
                    identifiers =
extract_identifier_list(subchild)
                elif subchild.name == "<type>":
                    param_type = extract_type(subchild)
                elif subchild.name.startswith("KEYWORD") and
"var" in subchild.name.lower():
                    is_var = True

            for identifier in identifiers:
                param_node = ParameterNode(identifier,
param_type, is_var)
                parameters.append(param_node)

    return parameters


def extract_record_fields(parse_node):
    """Extract field declarations from <record_type> node"""
    fields = []
    i = 0
    children = parse_node.children

    # Skip 'rekaman' keyword
    if i < len(children) and
children[i].name.startswith("KEYWORD"):
        i += 1

    # Extract fields until 'selesai'
```

```
    while i < len(children):
        if children[i].name.startswith("KEYWORD") and
"selesai" in children[i].name:
            break

        # Field name
        if children[i].name.startswith("IDENTIFIER"):
            field_name =
children[i].name.split("(")[1].rstrip(")")
            i += 1

            # Skip colon
            if i < len(children) and
children[i].name.startswith("COLON"):
                i += 1

            # Field type
            field_type = "unknown"
            if i < len(children) and
children[i].name.startswith("<"):
                field_type = extract_type_spec(children[i])
                i += 1

            fields.append({"name": field_name, "type":
field_type})

            # Skip semicolon
            if i < len(children) and
children[i].name.startswith("SEMICOLON"):
                i += 1
        else:
            i += 1

    return fields
```

Penjelasan

Pemrosesan khusus untuk melakukan ekstraksi value secara langsung beberapa node agar AST lebih bersih.

procedure_builder.py

```
import sys
import os
sys.path.append(os.path.join(os.path.dirname(__file__), '..',
'..'))

from node_class.declaration.procedure_decl_node import
ProcedureDeclNode
from .helpers import extract_parameters


def build_procedure_declaration(parse_node):
    from .declaration_builder import build_declaration

    i = 0
    children = parse_node.children

    if children[i].name.startswith("KEYWORD"):
        i += 1

    proc_name = "unknown"
    if children[i].name.startswith("IDENTIFIER"):
        proc_name =
children[i].name.split("(")[1].rstrip(")")
        i += 1

    parameters = []
    if i < len(children) and children[i].name ==
"<formal-parameter-list>":
        parameters = extract_parameters(children[i])
        i += 1

    if i < len(children) and
children[i].name.startswith("SEMICOLON"):
        i += 1

    declarations = []
    if i < len(children) and children[i].name ==
"<declaration-part>":
        declarations = build_declaration(children[i])
```

```
        i += 1

    body = None
    if i < len(children) and children[i].name ==
"<compound-statement>":
        body = children[i]
        i += 1

    return ProcedureDeclNode(proc_name, parameters,
declarations, body)
```

Penjelasan

Kelas builder yang memproses isi keseluruhan node <procedure_declaration> untuk membangun AST, yaitu deklarasi fungsi pada program.

type_builder.py

```
import sys
import os
sys.path.append(os.path.join(os.path.dirname(__file__), '..',
'..'))

from node_class.declaration.type_decl_node import
TypeDeclNode
from .helpers import extract_type_spec


def build_type_declaration(parse_node):
    i = 0
    children = parse_node.children

    if children[i].name.startswith("KEYWORD"):
        i += 1
```

```
    type_name = "unknown"
    if i < len(children) and
children[i].name.startswith("IDENTIFIER"):
        type_name =
children[i].name.split("(")[1].rstrip(")")
        i += 1

    if i < len(children) and
children[i].name.startswith("RELATIONAL_OPERATOR"):
        i += 1

    type_spec = "unknown"
    if i < len(children) and
children[i].name.startswith("<"):
        type_spec = extract_type_spec(children[i])
        i += 1

    return TypeDeclNode(type_name, type_spec)
```

Penjelasan

Kelas builder yang memproses isi keseluruhan node <type_declaration> untuk membangun AST, yaitu deklarasi tipe pada program.

var_builder.py

```
import sys
import os
sys.path.append(os.path.join(os.path.dirname(__file__), '..',
'..'))

from node_class.declaration.var_decl_node import VarDeclNode
from .helpers import extract_identifier_list, extract_type


def build_var_declaration(parse_node):
    var_nodes = []
    i = 0
    children = parse_node.children
```

```
    if children[i].name.startswith("KEYWORD"):
        i += 1

    identifiers = []
    if i < len(children) and children[i].name ==
"<identifier_list>":
        identifiers = extract_identifier_list(children[i])
        i += 1

    if i < len(children) and
children[i].name.startswith("COLON"):
        i += 1

    var_type = "unknown"
    if i < len(children) and children[i].name == "<type>":
        var_type = extract_type(children[i])
        i += 1

    for identifier in identifiers:
        var_node = VarDeclNode(identifier, var_type)
        var_nodes.append(var_node)

    return var_nodes
```

Penjelasan

Kelas builder yang memproses isi keseluruhan node <identifier_list> untuk membangun AST, yaitu deklarasi fungsi pada program.

expression_builder.py

```
from ...node_class.expression import (
    ExpressionNode,
    SimpleExpressionNode,
    TermNode,
    FactorNode,
    LiteralNode,
    NumberLiteral,
```

```
    CharLiteral,
    StringLiteral,
    IdentifierNode,
    FunctionCallNode,
    ArrayAccessNode,
    ParenthesizedExpressionNode,
    UnaryExpressionNode,
)
from ...node_class.expression.field_access_node import
FieldAccessNode


def build_expression(parse_node):
    """
    Convert <expression> parse node to ExpressionNode AST.
    Structure: <expression> → <simple_expression>
[<relational_operator> <simple_expression>]
    """
    if parse_node.name != "<expression>":
        raise ValueError(f"Expected <expression>, got
{parse_node.name}")

    left_operand = None
    operator = None
    right_operand = None

    for child in parse_node.children:
        if child.name == "<simple_expression>":
            if left_operand is None:
                left_operand = build_simple_expression(child)
            else:
                right_operand =
build_simple_expression(child)
        elif child.name == "<relational_operator>":
            operator = extract_relational_operator(child)

    return ExpressionNode(left_operand, operator,
right_operand)


def build_simple_expression(parse_node):
    """
```

```
    Convert <simple_expression> parse node to
SimpleExpressionNode AST.
    Structure: <simple_expression> → [+|-] <term>
{(+|-|atau) <term>}
    """
    if parse_node.name != "<simple_expression>":
        raise ValueError(f"Expected <simple_expression>, got
{parse_node.name}")

    simple_expr = SimpleExpressionNode()
    current_operator = None

    for child in parse_node.children:
        if child.name == "<term>":
            term = build_term(child)
            simple_expr.add_term(term, current_operator)
            current_operator = None
        elif child.name.startswith("ARITHMETIC_OPERATOR"):
            op_value = extract_token_value(child.name)
            if len(simple_expr.terms) == 0:
                # This is a unary sign
                simple_expr.sign = op_value
            else:
                # This is a binary operator
                current_operator = op_value
        elif child.name.startswith("LOGICAL_OPERATOR"):
            op_value = extract_token_value(child.name)
            if op_value == "atau":
                current_operator = op_value

    return simple_expr


def build_term(parse_node):
    """
    Convert <term> parse node to TermNode AST.
    Structure: <term> → <factor> {(*|/|bagi|mod|dan)
<factor>}
    """
    if parse_node.name != "<term>":
        raise ValueError(f"Expected <term>, got
{parse_node.name}")
```

```
    term = TermNode()
    current_operator = None

    for child in parse_node.children:
        if child.name == "<factor>":
            factor = build_factor(child)
            term.add_factor(factor, current_operator)
            current_operator = None
        elif child.name.startswith("ARITHMETIC_OPERATOR") or
child.name.startswith("MULTIPLICATIVE_OPERATOR"):
            current_operator =
extract_token_value(child.name)
        elif child.name.startswith("LOGICAL_OPERATOR"):
            op_value = extract_token_value(child.name)
            if op_value == "dan":
                current_operator = op_value

    return term


def build_factor(parse_node):
    """
    Convert <factor> parse node to appropriate FactorNode
subclass.
    Factor can be:
    - Literal (NUMBER, CHAR_LITERAL, STRING_LITERAL)
    - Identifier (variable)
    - Function call (IDENTIFIER followed by LPAREN)
    - Array access (IDENTIFIER followed by LBRACKET)
    - Parenthesized expression (LPAREN <expression> RPAREN)
    - Unary expression (tidak <factor>)
    """
    if parse_node.name != "<factor>":
        raise ValueError(f"Expected <factor>, got
{parse_node.name}")

    # Check for literal values
    for child in parse_node.children:
        if child.name.startswith("NUMBER"):
            value = extract_token_value(child.name)
            return NumberLiteral(value)
```

```
        elif child.name.startswith("CHAR_LITERAL"):
            value = extract_token_value(child.name)
            return CharLiteral(value)
        elif child.name.startswith("STRING_LITERAL"):
            value = extract_token_value(child.name)
            return StringLiteral(value)

    # Check for unary NOT operator
    if len(parse_node.children) >= 2:
        if
parse_node.children[0].name.startswith("LOGICAL_OPERATOR"):
            op =
extract_token_value(parse_node.children[0].name)
            if op == "tidak":
                operand =
build_factor(parse_node.children[1])
                return UnaryExpressionNode(op, operand)

    # Check for parenthesized expression
    if len(parse_node.children) >= 3:
        if parse_node.children[0].name.startswith("LPAREN"):
            # Find the expression node
            for child in parse_node.children:
                if child.name == "<expression>":
                    inner_expr = build_expression(child)
                    return
ParenthesizedExpressionNode(inner_expr)

    # Check for identifier (variable, function call, array
access, or field access)
    if parse_node.children and
parse_node.children[0].name.startswith("IDENTIFIER"):
        identifier =
extract_token_value(parse_node.children[0].name)
        current_node = IdentifierNode(identifier)
        i = 1

        # Process chained accessors (function call, array
index, field access)
        while i < len(parse_node.children):
            child = parse_node.children[i]
```

```
            if child.name.startswith("LPAREN"):
                # Function call
                arguments = []
                for j in range(i, len(parse_node.children)):
                    if parse_node.children[j].name ==
"<parameter_list>":
                        arguments =
build_parameter_list(parse_node.children[j])
                        break
                current_node = FunctionCallNode(identifier if
isinstance(current_node, IdentifierNode) else
str(current_node), arguments)
                i += 1
                # Skip to RPAREN
                while i < len(parse_node.children) and not
parse_node.children[i].name.startswith("RPAREN"):
                    i += 1
                i += 1

            elif child.name.startswith("LBRACKET"):
                # Array access
                i += 1
                if i < len(parse_node.children) and
parse_node.children[i].name == "<expression>":
                    index =
build_expression(parse_node.children[i])
                    current_node = ArrayAccessNode(
                        identifier if
isinstance(current_node, IdentifierNode) else current_node,
                        index
                    )
                    i += 1
                # Skip RBRACKET
                if i < len(parse_node.children) and
parse_node.children[i].name.startswith("RBRACKET"):
                    i += 1

            elif child.name.startswith("DOT"):
                # Field access
                i += 1
                if i < len(parse_node.children) and
parse_node.children[i].name.startswith("IDENTIFIER"):
```

```
                    field_name =
extract_token_value(parse_node.children[i].name)
                    current_node =
FieldAccessNode(current_node, field_name)
                    i += 1
            else:
                i += 1

        return current_node

    raise ValueError(f"Unable to parse factor:
{parse_node.name}")


def build_parameter_list(parse_node):
    """
    Convert <parameter_list> parse node to list of
ExpressionNode.
    """
    if parse_node.name != "<parameter_list>":
        raise ValueError(f"Expected <parameter_list>, got
{parse_node.name}")

    arguments = []
    for child in parse_node.children:
        if child.name == "<expression>":
            arguments.append(build_expression(child))

    return arguments


def extract_relational_operator(parse_node):
    """Extract the relational operator from
<relational_operator> node."""
    if parse_node.name != "<relational_operator>":
        raise ValueError(f"Expected <relational_operator>,
got {parse_node.name}")

    for child in parse_node.children:
        if child.name.startswith("RELATIONAL_OPERATOR"):
            return extract_token_value(child.name)
```

```
    return None


def extract_token_value(token_name):
    """
    Extract the value from a token name.
    Example: "NUMBER(42)" → "42", "IDENTIFIER(x)" → "x"
    """
    if "(" in token_name and ")" in token_name:
        return token_name.split("(")[1].rstrip(")")
    return token_name
```

Penjelasan

Kelas builder utama untuk pemrosesan expression yang muncul dalam program

helpres.py

```
def extract_identifier(parse_node):
    if parse_node.name.startswith("IDENTIFIER"):
        return parse_node.name.split("(")[1].rstrip(")")
    return "unknown"


def extract_keyword(parse_node):
    if parse_node.name.startswith("KEYWORD"):
        return parse_node.name.split("(")[1].rstrip(")")
    return "unknown"


def find_child_by_name(parse_node, name):
    for child in parse_node.children:
        if child.name == name:
            return child
    return None


def find_children_by_name(parse_node, name):
```

```
    return [child for child in parse_node.children if
child.name == name]
```

Penjelasan

Kelas builder utama untuk melakukan ekstraksi value dan nama pada pohon dari anak-anak sebuah node.

statement_builder.py

```
import sys
import os
sys.path.append(os.path.join(os.path.dirname(__file__), '..',
'..'))

from node_class.statement.assignment_statement_node import
AssignmentStatementNode
from node_class.statement.if_statement import IfStatementNode
from node_class.statement.for_statement_node import
ForStatementNode
from node_class.statement.while_statement_node import
WhileStatementNode
from node_class.statement.procedure_function_call_node import
procedureFunctionCallNode
from node_class.statement.statement_list_node import
StatementListNode
from node_class.misc.block_node import BlockNode
from .helpers import extract_identifier, extract_keyword
from ..expression.expression_builder import build_expression
from ...node_class.expression.array_access_node import
ArrayAccessNode
from ...node_class.expression.field_access_node import
FieldAccessNode
from ...node_class.expression.identifier_node import
IdentifierNode


def build_assignment_statement(parse_node):
```

```
    if parse_node.name != "<assignment-statement>":
        raise ValueError(f"Expected <assignment-statement>,
got {parse_node.name}")

    # Build left-hand side (can be identifier, array access,
or field access)
    lhs = None
    expression_node = None

    i = 0
    children = parse_node.children

    # Get base identifier
    if i < len(children) and
children[i].name.startswith("IDENTIFIER"):
        identifier = extract_identifier(children[i])
        lhs = IdentifierNode(identifier)
        i += 1

        # Process array indexing and/or field access
        while i < len(children) and not
children[i].name.startswith("ASSIGN_OPERATOR"):
            if children[i].name.startswith("LBRACKET"):
                # Array access
                i += 1
                if i < len(children) and children[i].name ==
"<expression>":
                    index = build_expression(children[i])
                    # Use current lhs (could be identifier
string or node) as base
                    if isinstance(lhs, IdentifierNode):
                        lhs = ArrayAccessNode(lhs.name,
index)
                    else:
                        lhs = ArrayAccessNode(lhs, index)
                    i += 1
                # Skip RBRACKET
                if i < len(children) and
children[i].name.startswith("RBRACKET"):
                    i += 1
            elif children[i].name.startswith("DOT"):
                # Field access
```

```
                i += 1
                if i < len(children) and
children[i].name.startswith("IDENTIFIER"):
                    field_name =
extract_identifier(children[i])
                    lhs = FieldAccessNode(lhs, field_name)
                    i += 1
            else:
                i += 1

    # Skip ASSIGN_OPERATOR and get right-hand side expression
    while i < len(children):
        if children[i].name == "<expression>":
            expression_node = build_expression(children[i])
            break
        i += 1

    # Create assignment node with LHS node
    # For backward compatibility, extract variable name from
LHS
    if isinstance(lhs, IdentifierNode):
        variable_name = lhs.name
    elif isinstance(lhs, ArrayAccessNode):
        # Extract base array name
        if isinstance(lhs.array_name, str):
            variable_name = lhs.array_name
        elif isinstance(lhs.array_name, IdentifierNode):
            variable_name = lhs.array_name.name
        else:
            variable_name = str(lhs.array_name)
    elif isinstance(lhs, FieldAccessNode):
        # Extract base record name
        if isinstance(lhs.record_expr, IdentifierNode):
            variable_name = lhs.record_expr.name
        elif isinstance(lhs.record_expr, str):
            variable_name = lhs.record_expr
        else:
            variable_name = str(lhs.record_expr)
    else:
        variable_name = "unknown"

    # Create assignment node - store both variable_name and
```

```
lhs_node
    assignment_node = AssignmentStatementNode(variable_name,
expression_node)
    assignment_node.lhs_node = lhs  # Store the full LHS
access node
    return assignment_node


def build_if_statement(parse_node):
    if parse_node.name != "<if-statement>":
        raise ValueError(f"Expected <if-statement>, got
{parse_node.name}")

    condition = None
    then_body = None
    else_body = None

    i = 0
    children = parse_node.children

    # Skip 'jika'
    if i < len(children) and
children[i].name.startswith("KEYWORD"):
        i += 1

    if i < len(children) and children[i].name ==
"<expression>":
        condition = build_expression(children[i])
        i += 1

    # Skip 'maka'
    if i < len(children) and
children[i].name.startswith("KEYWORD"):
        i += 1

    if i < len(children) and
(children[i].name.startswith("<") or
children[i].name.startswith("IDENTIFIER")):
        then_body = build_statement(children[i])
        i += 1

    if i < len(children) and
```

```
children[i].name.startswith("KEYWORD") and "selain_itu" in
children[i].name:
        i += 1
        if i < len(children):
            else_body = build_statement(children[i])

    return IfStatementNode(condition, then_body, else_body)


def build_for_statement(parse_node):
    if parse_node.name != "<for_statement>":
        raise ValueError(f"Expected <for_statement>, got
{parse_node.name}")

    variable = "unknown"
    start_expr = None
    end_expr = None
    direction = "ke"
    body = None

    i = 0
    children = parse_node.children

    # Skip 'untuk'
    if i < len(children) and
children[i].name.startswith("KEYWORD"):
        i += 1

    if i < len(children) and
children[i].name.startswith("IDENTIFIER"):
        variable = extract_identifier(children[i])
        i += 1

    # Skip ':='
    if i < len(children) and
children[i].name.startswith("ASSIGN_OPERATOR"):
        i += 1

    if i < len(children) and children[i].name ==
"<expression>":
        start_expr = build_expression(children[i])
        i += 1
```

```
    if i < len(children) and
children[i].name.startswith("KEYWORD"):
        direction = extract_keyword(children[i])
        i += 1

    if i < len(children) and children[i].name ==
"<expression>":
        end_expr = build_expression(children[i])
        i += 1

    # Skip 'lakukan'
    if i < len(children) and
children[i].name.startswith("KEYWORD"):
        i += 1

    if i < len(children):
        body = build_statement(children[i])

    return ForStatementNode(variable, start_expr, end_expr,
body)


def build_while_statement(parse_node):
    if parse_node.name != "<while_statement>":
        raise ValueError(f"Expected <while_statement>, got
{parse_node.name}")

    condition = None
    body = None

    i = 0
    children = parse_node.children

    # Skip 'selama'
    if i < len(children) and
children[i].name.startswith("KEYWORD"):
        i += 1

    if i < len(children) and children[i].name ==
"<expression>":
        condition = build_expression(children[i])
```

```
        i += 1

    # Skip 'lakukan'
    if i < len(children) and
children[i].name.startswith("KEYWORD"):
        i += 1

    if i < len(children):
        body = build_statement(children[i])

    return WhileStatementNode(condition, body)


def build_procedure_function_call(parse_node):
    if parse_node.name != "<procedure/function-call>":
        raise ValueError(f"Expected
<procedure/function-call>, got {parse_node.name}")

    function_name = "unknown"
    arguments = []

    i = 0
    children = parse_node.children

    if i < len(children) and
children[i].name.startswith("IDENTIFIER"):
        function_name = extract_identifier(children[i])
        i += 1

    # Skip LPAREN
    if i < len(children) and
children[i].name.startswith("LPAREN"):
        i += 1

    # Get parameter list
    if i < len(children) and children[i].name ==
"<parameter_list>":
        param_list_node = children[i]
        for param_child in param_list_node.children:
            if param_child.name == "<expression>":

arguments.append(build_expression(param_child))
```

```
        i += 1

    return procedureFunctionCallNode(function_name,
arguments)


def build_compound_statement(parse_node):
    if parse_node.name != "<compound-statement>":
        raise ValueError(f"Expected <compound-statement>, got
{parse_node.name}")

    block_node = BlockNode()

    for child in parse_node.children:
        if child.name == "<statement-list>":
            statements = build_statements(child)
            for stmt in statements:
                block_node.add_child(stmt)
            break

    return block_node


def build_statement(parse_node):
    node_name = parse_node.name

    builders = {
        "<assignment-statement>": build_assignment_statement,
        "<if-statement>": build_if_statement,
        "<for_statement>": build_for_statement,
        "<while_statement>": build_while_statement,
        "<compound-statement>": build_compound_statement,
        "<procedure/function-call>":
build_procedure_function_call,
    }

    if node_name in builders:
        return builders[node_name](parse_node)

    if node_name.startswith("IDENTIFIER"):
        return build_procedure_function_call(parse_node)
```

```
    raise ValueError(f"Unknown statement type: {node_name}")


def build_statements(parse_node):
    all_statements = []

    for child in parse_node.children:
        if child.name.startswith("KEYWORD") or
child.name.startswith("SEMICOLON"):
            continue

        if child.name.startswith("<") or
child.name.startswith("IDENTIFIER"):
            all_statements.append(build_statement(child))

    return all_statements
```

Penjelasan

Kelas builder utama untuk pemrosesan keseluruhan statement bagian blok kode program yang muncul dalam program

c. Analyzer

decorator.py

```
from .symbol_table import SymbolTable

class SemanticError(Exception):
    pass

class ASTDecorator:
    def __init__(self):
        self.symbol_table = SymbolTable()
        self.errors = []
        self.current_function = None
        self.ast_output = []  # Store decorated AST output

    def decorate(self, ast_root):
```

```
        try:
            self.visit(ast_root)
        except Exception as e:
            self.errors.append(f"Decoration error: {str(e)}")

        return self.symbol_table, self.errors

    def get_decorated_ast_string(self, node=None, indent=0,
prefix="", is_last=True):
        """Generate tree-like string representation of
decorated AST"""
        if node is None:
            return ""

        result = []
        node_type = type(node).__name__

        # Build node representation with annotations
        node_repr = self._build_node_repr(node, node_type)

        # Add tree structure
        if indent == 0:
            result.append(node_repr)
        else:
            connector = "└─ " if is_last else "├─ "
            result.append(prefix + connector + node_repr)

        # Visit children
        children = self._get_node_children(node, node_type)

        for i, child in enumerate(children):
            is_last_child = (i == len(children) - 1)
            if indent == 0:
                child_prefix = " "
            else:
                child_prefix = prefix + ("   " if is_last
else "│  ")

            child_str = self.get_decorated_ast_string(child,
indent + 1, child_prefix, is_last_child)
            if child_str:
                result.append(child_str)
```

```
        return "\n".join(result)

    def _build_node_repr(self, node, node_type):
        """Build node representation with semantic
annotations"""
        parts = [node_type]

        # Add node-specific information
        if node_type == "ProgramNode":
            parts[0] = f"ProgramNode(name: '{node.name}')"
            if hasattr(node, 'symbol_idx'):
                parts.append(f" →
tab_index:{node.symbol_idx}")

        elif node_type in ["VarDeclNode", "ConstDeclNode",
"TypeDeclNode"]:
            name = node.name if hasattr(node, 'name') else
'?'
            parts[0] = f"{node_type}('{name}')"
            if hasattr(node, 'symbol_idx'):
                parts.append(f" →
tab_index:{node.symbol_idx}")
            if hasattr(node, 'type_code'):
                type_name =
self._type_code_to_name(node.type_code)
                parts.append(f", type:{type_name}")
            if hasattr(node, 'symbol_idx') and
node.symbol_idx < len(self.symbol_table.tab):
                entry =
self.symbol_table.tab[node.symbol_idx]
                parts.append(f", lev:{entry.lev}")

        elif node_type in ["ProcedureDeclNode",
"FunctionDeclNode"]:
            name = node.name if hasattr(node, 'name') else
'?'
            parts[0] = f"{node_type}('{name}')"
            if hasattr(node, 'symbol_idx'):
                parts.append(f" →
tab_index:{node.symbol_idx}")
            if hasattr(node, 'block_idx'):
```

```
                parts.append(f",
block_index:{node.block_idx}")
            if node_type == "FunctionDeclNode" and
hasattr(node, 'type_code'):
                type_name =
self._type_code_to_name(node.type_code)
                parts.append(f", return_type:{type_name}")

        elif node_type == "AssignmentStatementNode":
            var_name = node.variable_name if hasattr(node,
'variable_name') else '?'
            parts[0] = f"Assign('{var_name}' := ...)"
            if hasattr(node, 'variable_type'):
                type_name =
self._type_code_to_name(node.variable_type)
                parts.append(f" → type:{type_name}")
            if hasattr(node, 'symbol_idx'):
                parts.append(f",
tab_index:{node.symbol_idx}")

        elif node_type == "IdentifierNode":
            name = node.name if hasattr(node, 'name') else
'?'
            parts[0] = f"Identifier('{name}')"
            if hasattr(node, 'symbol_idx'):
                parts.append(f" →
tab_index:{node.symbol_idx}")
            if hasattr(node, 'expression_type'):
                type_name =
self._type_code_to_name(node.expression_type)
                parts.append(f", type:{type_name}")

        elif node_type == "NumberLiteral":
            value = node.value if hasattr(node, 'value') else
'?'
            parts[0] = f"NumberLiteral({value})"
            if hasattr(node, 'expression_type'):
                type_name =
self._type_code_to_name(node.expression_type)
                parts.append(f" → type:{type_name}")

        elif node_type in ["CharLiteral", "StringLiteral"]:
```

```
            value = node.value if hasattr(node, 'value') else
'?'
            parts[0] = f"{node_type}('{value}')"
            if hasattr(node, 'expression_type'):
                type_name =
self._type_code_to_name(node.expression_type)
                parts.append(f" → type:{type_name}")

        elif node_type == "ExpressionNode":
            if hasattr(node, 'operator') and node.operator:
                parts[0] = f"Expression(op:
'{node.operator}')"
            else:
                parts[0] = "Expression"
            if hasattr(node, 'expression_type'):
                type_name =
self._type_code_to_name(node.expression_type)
                parts.append(f" → type:{type_name}")

        elif node_type == "SimpleExpressionNode":
            if hasattr(node, 'operators') and node.operators:
                parts[0] = f"SimpleExpression(ops:
{node.operators})"
            else:
                parts[0] = "SimpleExpression"
            if hasattr(node, 'expression_type'):
                type_name =
self._type_code_to_name(node.expression_type)
                parts.append(f" → type:{type_name}")

        elif node_type == "TermNode":
            if hasattr(node, 'operators') and node.operators:
                parts[0] = f"Term(ops: {node.operators})"
            else:
                parts[0] = "Term"
            if hasattr(node, 'expression_type'):
                type_name =
self._type_code_to_name(node.expression_type)
                parts.append(f" → type:{type_name}")

        elif node_type in ["IfStatementNode",
"WhileStatementNode", "ForStatementNode"]:
```

```
            if node_type == "ForStatementNode" and
hasattr(node, 'variable'):
                parts[0] = f"ForStatement(var:
'{node.variable}')"
            else:
                parts[0] = node_type

        elif node_type in ["FunctionCallNode",
"ProcedureFunctionCallNode"]:
            name = node.function_name if hasattr(node,
'function_name') else (node.name if hasattr(node, 'name')
else '?')
            parts[0] = f"{node_type}('{name}')"
            if hasattr(node, 'symbol_idx'):
                parts.append(f" →
tab_index:{node.symbol_idx}")
            if hasattr(node, 'expression_type'):
                type_name =
self._type_code_to_name(node.expression_type)
                parts.append(f", type:{type_name}")

        elif node_type == "ArrayAccessNode":
            name = node.array_name if hasattr(node,
'array_name') else '?'
            parts[0] = f"ArrayAccess('{name}')"
            if hasattr(node, 'symbol_idx'):
                parts.append(f" →
tab_index:{node.symbol_idx}")
            if hasattr(node, 'expression_type'):
                type_name =
self._type_code_to_name(node.expression_type)
                parts.append(f", elem_type:{type_name}")

        return "".join(parts)

    def _type_code_to_name(self, type_code):
        type_map = {
            SymbolTable.TYPE_NOTYPE: "notype",
            SymbolTable.TYPE_INTEGER: "integer",
            SymbolTable.TYPE_REAL: "real",
            SymbolTable.TYPE_BOOLEAN: "boolean",
            SymbolTable.TYPE_CHAR: "char",
```

```
            SymbolTable.TYPE_ARRAY: "array",
            SymbolTable.TYPE_RECORD: "record"
        }
        return type_map.get(type_code,
f"unknown({type_code})")

    def _get_node_children(self, node, node_type):
        children = []

        # Handle special cases with specific child ordering
        if node_type == "ProgramNode":
            if hasattr(node, 'children'):
                children.extend(node.children)

        elif node_type == "DeclarationsNode":
            if hasattr(node, 'children'):
                children.extend(node.children)

        elif node_type in ["ProcedureDeclNode",
"FunctionDeclNode"]:
            if hasattr(node, 'parameters') and
node.parameters:
                for param in node.parameters:
                    children.append(param)
            if hasattr(node, 'declarations') and
node.declarations:
                for decl in node.declarations:
                    children.append(decl)
            if hasattr(node, 'children'):
                children.extend(node.children)

        elif node_type == "AssignmentStatementNode":
            if hasattr(node, 'value'):
                children.append(node.value)

        elif node_type == "ExpressionNode":
            if hasattr(node, 'left_operand'):
                children.append(node.left_operand)
            if hasattr(node, 'right_operand') and
node.right_operand:
                children.append(node.right_operand)
```

```
        elif node_type == "SimpleExpressionNode":
            if hasattr(node, 'terms'):
                children.extend(node.terms)

        elif node_type == "TermNode":
            if hasattr(node, 'factors'):
                children.extend(node.factors)

        elif node_type == "IfStatementNode":
            if hasattr(node, 'compare'):
                children.append(node.compare)
            if hasattr(node, 'ifbody'):
                children.append(node.ifbody)
            if hasattr(node, 'elsebody') and node.elsebody:
                children.append(node.elsebody)

        elif node_type == "WhileStatementNode":
            if hasattr(node, 'condition'):
                children.append(node.condition)
            if hasattr(node, 'children'):
                children.extend(node.children)

        elif node_type == "ForStatementNode":
            if hasattr(node, 'start'):
                children.append(node.start)
            if hasattr(node, 'end'):
                children.append(node.end)
            if hasattr(node, 'children'):
                children.extend(node.children)

        elif node_type == "ArrayAccessNode":
            if hasattr(node, 'index'):
                children.append(node.index)

        elif node_type in ["FunctionCallNode",
"ProcedureFunctionCallNode"]:
            if hasattr(node, 'arguments'):
                children.extend(node.arguments)

        elif hasattr(node, 'children'):
            children.extend(node.children)
```

```
        return children

    def save_decorated_ast(self, ast_root, filepath):
        """Save decorated AST to file"""
        ast_string = self.get_decorated_ast_string(ast_root)
        with open(filepath, 'w', encoding='utf-8') as f:
            f.write("Decorated AST:\n")
            f.write("=" * 80 + "\n\n")
            f.write(ast_string)
            f.write("\n")

    def visit(self, node):
        node_type = type(node).__name__
        method_name = f'visit_{node_type}'

        visitor = getattr(self, method_name,
self.visit_generic)
        return visitor(node)

    def visit_generic(self, node):
        for child in node.children:
            self.visit(child)

    def visit_ProgramNode(self, node):
        # Add program name to symbol table
        idx = self.symbol_table.enter(
            name=node.name,
            obj=SymbolTable.OBJ_PROGRAM,
            typ=SymbolTable.TYPE_NOTYPE,
            nrm=1,
            adr=0
        )
        node.symbol_idx = idx

        # Program is at global scope (level 0)
        for child in node.children:
            self.visit(child)

    def visit_DeclarationsNode(self, node):
        for child in node.children:
            self.visit(child)
```

```
    def visit_VarDeclNode(self, node):
        idx, existing =
self.symbol_table.lookup_current_scope(node.name)
        if existing is not None:
            self.errors.append(
                f"Error: Variable '{node.name}' already
declared in current scope"
            )
            return

        # Try to get primitive type first
        type_code = self.get_type_code(node.var_type)
        ref = 0

        # If not a primitive type, look up user-defined type
        if type_code == SymbolTable.TYPE_NOTYPE:
            type_idx, type_entry =
self.symbol_table.lookup(node.var_type)
            if type_entry and type_entry.obj ==
SymbolTable.OBJ_TYPE:
                type_code = type_entry.type
                ref = type_entry.ref
            else:
                self.errors.append(f"Error: Unknown type
'{node.var_type}'")
                return

        # Calculate address based on scope
        if self.symbol_table.current_level == 0:
            # Global variable: use global data size
            adr = self.symbol_table.global_data_size
            type_size =
self.symbol_table.get_type_size(type_code, ref)
            self.symbol_table.global_data_size += type_size
        else:
            # Local variable: use block vsze
            adr = self.calculate_address()

        idx = self.symbol_table.enter(
            name=node.name,
            obj=SymbolTable.OBJ_VARIABLE,
            typ=type_code,
```

```
            ref=ref,
            nrm=1,
            adr=adr
        )

        node.symbol_idx = idx
        node.type_code = type_code

        # Update block table: last pointer and VSize for
local variables
        if self.symbol_table.current_level > 0 and
self.symbol_table.current_block <
len(self.symbol_table.btab):
            # Only add to VSize if this is a local variable
(not a parameter)
            # Parameters have already been counted in PSize
            block =
self.symbol_table.btab[self.symbol_table.current_block]

            # Check if this variable comes after the last
parameter
            if block.lpar == 0 or idx > block.lpar:
                type_size =
self.symbol_table.get_type_size(type_code, ref)
                block.vsze += type_size

            # Always update the last pointer to the most
recent identifier
            block.last = idx

    def visit_ConstDeclNode(self, node):
        idx, existing =
self.symbol_table.lookup_current_scope(node.name)
        if existing is not None:
            self.errors.append(
                f"Error: Constant '{node.name}' already
declared in current scope"
            )
            return

        type_code = self.infer_constant_type(node.value)
```

```
        # Store constant value in adr field
        const_value = node.value
        if isinstance(const_value, str):
            # Try to convert string to int
            try:
                const_value = int(const_value)
            except ValueError:
                try:
                    const_value = int(float(const_value))
                except ValueError:
                    const_value = 0

        idx = self.symbol_table.enter(
            name=node.name,
            obj=SymbolTable.OBJ_CONSTANT,
            typ=type_code,
            adr=const_value
        )

        node.symbol_idx = idx
        node.type_code = type_code

    def visit_TypeDeclNode(self, node):
        idx, existing =
self.symbol_table.lookup_current_scope(node.name)
        if existing is not None:
            self.errors.append(
                f"Error: Type '{node.name}' already declared
in current scope"
            )
            return

        ref = 0
        type_code = SymbolTable.TYPE_NOTYPE

        if hasattr(node, 'type_spec'):
            type_code, ref =
self.process_type_spec(node.type_spec)

        idx = self.symbol_table.enter(
            name=node.name,
            obj=SymbolTable.OBJ_TYPE,
```

```
            typ=type_code,
            ref=ref
        )

        node.symbol_idx = idx
        node.type_code = type_code

    def visit_ProcedureDeclNode(self, node):
        idx, existing =
self.symbol_table.lookup_current_scope(node.name)
        if existing is not None:
            self.errors.append(
                f"Error: Procedure '{node.name}' already
declared in current scope"
            )
            return

        # Create new block for procedure
        block_idx = self.symbol_table.enter_scope()

        # Enter procedure name in symbol table with reference
to its block
        idx = self.symbol_table.enter(
            name=node.name,
            obj=SymbolTable.OBJ_PROCEDURE,
            typ=SymbolTable.TYPE_NOTYPE,
            ref=block_idx,
            lev=self.symbol_table.current_level - 1  #
Procedure declared in parent scope
        )

        node.symbol_idx = idx
        node.block_idx = block_idx

        if hasattr(node, 'parameters') and node.parameters:
            self.process_parameters(node.parameters,
block_idx)

        # Visit local declarations
        if hasattr(node, 'declarations') and
node.declarations:
            for decl in node.declarations:
```

```
                self.visit(decl)

        # Visit body and other children
        for child in node.children:
            self.visit(child)

        self.symbol_table.exit_scope()

    def visit_FunctionDeclNode(self, node):
        idx, existing =
self.symbol_table.lookup_current_scope(node.name)
        if existing is not None:
            self.errors.append(
                f"Error: Function '{node.name}' already
declared in current scope"
            )
            return

        # Create new block for function
        block_idx = self.symbol_table.enter_scope()
        self.current_function = node.name

        return_type = self.get_type_code(node.return_type)

        # Enter function name in symbol table with reference
to its block
        idx = self.symbol_table.enter(
            name=node.name,
            obj=SymbolTable.OBJ_FUNCTION,
            typ=return_type,
            ref=block_idx,
            lev=self.symbol_table.current_level - 1  #
Function declared in parent scope
        )

        node.symbol_idx = idx
        node.block_idx = block_idx
        node.type_code = return_type

        if hasattr(node, 'parameters') and node.parameters:
            self.process_parameters(node.parameters,
block_idx)
```

```
        # Visit local declarations
        if hasattr(node, 'declarations') and
node.declarations:
            for decl in node.declarations:
                self.visit(decl)

        # Visit body and other children
        for child in node.children:
            self.visit(child)

        self.current_function = None
        self.symbol_table.exit_scope()

    def process_parameters(self, parameters, block_idx):
        param_offset = 0
        last_param_idx = 0

        for param in parameters:
            type_code = self.get_type_code(param.param_type)
            nrm = 0 if param.is_var else 1

            idx = self.symbol_table.enter(
                name=param.name,
                obj=SymbolTable.OBJ_VARIABLE,
                typ=type_code,
                nrm=nrm,
                adr=param_offset
            )

            param.symbol_idx = idx
            param.type_code = type_code

            type_size =
self.symbol_table.get_type_size(type_code)
            param_offset += type_size
            last_param_idx = idx

        if block_idx < len(self.symbol_table.btab):
            self.symbol_table.btab[block_idx].lpar =
last_param_idx
            self.symbol_table.btab[block_idx].psze =
```

```
param_offset

    def visit_BlockNode(self, node):
        for child in node.children:
            self.visit(child)

    def visit_AssignmentStatementNode(self, node):
        idx, entry =
self.symbol_table.lookup(node.variable_name)
        if entry is None:
            self.errors.append(
                f"Error: Undefined variable
'{node.variable_name}'"
            )
            return

        if entry.obj != SymbolTable.OBJ_VARIABLE:
            self.errors.append(
                f"Error: '{node.variable_name}' is not a
variable"
            )
            return

        node.symbol_idx = idx
        node.variable_type = entry.type

        if hasattr(node, 'value'):
            expr_type = self.visit_expression(node.value)
            node.expression_type = expr_type

            if not self.types_compatible(entry.type,
expr_type):
                self.errors.append(
                    f"Error: Type mismatch in assignment to
'{node.variable_name}'"
                )

    def visit_expression(self, node):
        node_type = type(node).__name__

        if node_type == 'ExpressionNode':
            return self.visit_ExpressionNode(node)
```

```
        elif node_type == 'SimpleExpressionNode':
            return self.visit_SimpleExpressionNode(node)
        elif node_type == 'TermNode':
            return self.visit_TermNode(node)
        else:
            return self.visit_factor(node)

    def visit_ExpressionNode(self, node):
        left_type = self.visit_expression(node.left_operand)

        if node.operator is not None:
            right_type =
self.visit_expression(node.right_operand)

            if not self.types_compatible(left_type,
right_type):
                self.errors.append("Error: Type mismatch in
comparison")

            node.expression_type = SymbolTable.TYPE_BOOLEAN
            return SymbolTable.TYPE_BOOLEAN

        node.expression_type = left_type
        return left_type

    def visit_SimpleExpressionNode(self, node):
        if not node.terms:
            return SymbolTable.TYPE_NOTYPE

        result_type = self.visit_expression(node.terms[0])

        if node.operators and 'atau' in node.operators:
            result_type = SymbolTable.TYPE_BOOLEAN

        for i, term in enumerate(node.terms[1:]):
            term_type = self.visit_expression(term)

            if i < len(node.operators):
                op = node.operators[i]
                if op in ['+', '-']:
                    if result_type == SymbolTable.TYPE_REAL
or term_type == SymbolTable.TYPE_REAL:
```

```
                        result_type = SymbolTable.TYPE_REAL

        node.expression_type = result_type
        return result_type

    def visit_TermNode(self, node):
        if not node.factors:
            return SymbolTable.TYPE_NOTYPE

        result_type = self.visit_factor(node.factors[0])

        if node.operators and 'dan' in node.operators:
            result_type = SymbolTable.TYPE_BOOLEAN

        for i, factor in enumerate(node.factors[1:]):
            factor_type = self.visit_factor(factor)

            if i < len(node.operators):
                op = node.operators[i]
                if op in ['*', '/', 'bagi', 'mod']:
                    if result_type == SymbolTable.TYPE_REAL
or factor_type == SymbolTable.TYPE_REAL:
                        result_type = SymbolTable.TYPE_REAL

        node.expression_type = result_type
        return result_type

    def visit_factor(self, node):
        node_type = type(node).__name__

        if node_type == 'NumberLiteral':
            if '.' in str(node.value):
                node.expression_type = SymbolTable.TYPE_REAL
                return SymbolTable.TYPE_REAL
            node.expression_type = SymbolTable.TYPE_INTEGER
            return SymbolTable.TYPE_INTEGER

        elif node_type == 'CharLiteral':
            node.expression_type = SymbolTable.TYPE_CHAR
            return SymbolTable.TYPE_CHAR

        elif node_type == 'StringLiteral':
```

```
            node.expression_type = SymbolTable.TYPE_CHAR
            return SymbolTable.TYPE_CHAR

        elif node_type == 'IdentifierNode':
            idx, entry = self.symbol_table.lookup(node.name)
            if entry is None:
                self.errors.append(f"Error: Undefined
identifier '{node.name}'")
                return SymbolTable.TYPE_NOTYPE

            node.symbol_idx = idx
            node.expression_type = entry.type
            return entry.type

        elif node_type == 'UnaryExpressionNode':
            operand_type = self.visit_factor(node.operand)
            node.expression_type = SymbolTable.TYPE_BOOLEAN
            return SymbolTable.TYPE_BOOLEAN

        elif node_type == 'ParenthesizedExpressionNode':
            expr_type =
self.visit_expression(node.expression)
            node.expression_type = expr_type
            return expr_type

        elif node_type == 'FunctionCallNode':
            return self.visit_FunctionCallNode(node)

        elif node_type == 'ArrayAccessNode':
            return self.visit_ArrayAccessNode(node)

        return SymbolTable.TYPE_NOTYPE

    def visit_FunctionCallNode(self, node):
        idx, entry =
self.symbol_table.lookup(node.function_name)
        if entry is None:
            self.errors.append(f"Error: Undefined function
'{node.function_name}'")
            return SymbolTable.TYPE_NOTYPE

        if entry.obj != SymbolTable.OBJ_FUNCTION and
```

```
entry.obj != SymbolTable.OBJ_PROCEDURE:
            self.errors.append(f"Error:
'{node.function_name}' is not a function")
            return SymbolTable.TYPE_NOTYPE

        node.symbol_idx = idx
        node.expression_type = entry.type

        for arg in node.arguments:
            self.visit_expression(arg)

        return entry.type

    def visit_ArrayAccessNode(self, node):
        idx, entry =
self.symbol_table.lookup(node.array_name)
        if entry is None:
            self.errors.append(f"Error: Undefined array
'{node.array_name}'")
            return SymbolTable.TYPE_NOTYPE

        if entry.type != SymbolTable.TYPE_ARRAY:
            self.errors.append(f"Error: '{node.array_name}'
is not an array")
            return SymbolTable.TYPE_NOTYPE

        node.symbol_idx = idx

        self.visit_expression(node.index)

        if entry.ref < len(self.symbol_table.atab):
            elem_type =
self.symbol_table.atab[entry.ref].etyp
            node.expression_type = elem_type
            return elem_type

        return SymbolTable.TYPE_NOTYPE

    def visit_IfStatementNode(self, node):
        if hasattr(node, 'compare'):
            cond_type = self.visit_expression(node.compare)
            if cond_type != SymbolTable.TYPE_BOOLEAN:
```

```
                self.errors.append("Error: If condition must
be boolean")

        if hasattr(node, 'ifbody'):
            self.visit(node.ifbody)
        if hasattr(node, 'elsebody'):
            self.visit(node.elsebody)

    def visit_WhileStatementNode(self, node):
        if hasattr(node, 'condition'):
            cond_type = self.visit_expression(node.condition)
            if cond_type != SymbolTable.TYPE_BOOLEAN:
                self.errors.append("Error: While condition
must be boolean")

        for child in node.children:
            self.visit(child)

    def visit_ForStatementNode(self, node):
        if hasattr(node, 'variable'):
            idx, entry =
self.symbol_table.lookup(node.variable)
            if entry is None:
                self.errors.append(f"Error: Undefined loop
variable '{node.variable}'")
            elif entry.type != SymbolTable.TYPE_INTEGER:
                self.errors.append("Error: Loop variable must
be integer")
            else:
                node.symbol_idx = idx

        if hasattr(node, 'start'):
            self.visit_expression(node.start)
        if hasattr(node, 'end'):
            self.visit_expression(node.end)

        for child in node.children:
            self.visit(child)

    def visit_ProcedureFunctionCallNode(self, node):
        idx, entry = self.symbol_table.lookup(node.name)
        if entry is None:
```

```
            self.errors.append(f"Error: Undefined
procedure/function '{node.name}'")
            return

        node.symbol_idx = idx

        if hasattr(node, 'arguments'):
            for arg in node.arguments:
                self.visit_expression(arg)

    def get_type_code(self, type_name):
        type_map = {
            'integer': SymbolTable.TYPE_INTEGER,
            'real': SymbolTable.TYPE_REAL,
            'boolean': SymbolTable.TYPE_BOOLEAN,
            'char': SymbolTable.TYPE_CHAR,
        }
        return type_map.get(type_name.lower(),
SymbolTable.TYPE_NOTYPE)

    def infer_constant_type(self, value):
        return SymbolTable.TYPE_INTEGER

    def process_type_spec(self, type_spec):
        """Process type specification and return (type_code,
ref)"""
        # Handle simple type strings
        if isinstance(type_spec, str):
            if type_spec in ['integer', 'real', 'boolean',
'char']:
                return self.get_type_code(type_spec), 0
            return SymbolTable.TYPE_NOTYPE, 0

        # Handle array type dictionary
        if isinstance(type_spec, dict) and
type_spec.get("type") == "array":
            # Get element type
            elem_type_str = type_spec.get("element_type",
"integer")
            elem_type_code =
self.get_type_code(elem_type_str)
```

```
            # Get bounds
            low = type_spec.get("low", 0)
            high = type_spec.get("high", 0)

            # Index type is always integer for now
            xtyp = SymbolTable.TYPE_INTEGER

            # Create entry in atab
            atab_idx = self.symbol_table.enter_array(
                xtyp=xtyp,
                etyp=elem_type_code,
                eref=0,  # No nested arrays for now
                low=low,
                high=high
            )

            return SymbolTable.TYPE_ARRAY, atab_idx

        # Handle record type dictionary
        if isinstance(type_spec, dict) and
type_spec.get("type") == "record":
            fields = type_spec.get("fields", [])

            # Create entry in rtab
            rtab_idx = self.symbol_table.enter_record(fields)

            return SymbolTable.TYPE_RECORD, rtab_idx

        return SymbolTable.TYPE_NOTYPE, 0

    def calculate_address(self):
        if self.symbol_table.current_block <
len(self.symbol_table.btab):
            return
self.symbol_table.btab[self.symbol_table.current_block].vsze
        return 0

    def types_compatible(self, type1, type2):
        if type1 == type2:
            return True
        if {type1, type2} == {SymbolTable.TYPE_INTEGER,
SymbolTable.TYPE_REAL}:
```

```
            return True
        return False
```

Penjelasan

File program decorator.py berfungsi untuk membuat Decorated AST sembari mengisi *symbol table* berdasarkan pohon AST yang telah dibangun sebelumnya. Pemrosesan ini dilakukan dengan menggunakan pendekatan handling function visit yang dikhususkan untuk setiap class node yang terdapat di AST. File ini membangun *symbol table* melalui fungsi enter, dan menempelkan indexnya pada node tersebut sebagai proses dekorasi. Validitas dari AST dicek utamanya oleh lookup pada *symbol table* untuk mengecek nama identifier yang di-declare dan dipanggil pada kode.

symbol_table.py

```
class SymbolTableEntry:
    def __init__(self, name, obj, typ, ref=0, nrm=1, lev=0,
adr=0, link=0):
        self.name = name        # Identifier name
        self.link = link        # Pointer to previous
identifier in same scope
        self.obj = obj          # Object class (constant,
variable, type, procedure, function)
        self.type = typ         # Basic type (integer,
boolean, char, real, array, record)
        self.ref = ref          # Pointer to atab/btab if
composite type
        self.nrm = nrm          # 1 = normal variable, 0 =
var parameter (by reference)
        self.lev = lev          # Lexical level (0 = global,
1+ = local)
        self.adr = adr          # Address/offset/value
depending on obj type

    def __str__(self):
        return (f"Entry(name={self.name}, obj={self.obj},
type={self.type}, "
                f"ref={self.ref}, nrm={self.nrm},
lev={self.lev}, adr={self.adr}, link={self.link})")


class ArrayTableEntry:
```

```
    def __init__(self, xtyp, etyp, eref=0, low=0, high=0,
elsz=1, size=0):
        self.xtyp = xtyp    # Index type
        self.etyp = etyp    # Element type
        self.eref = eref    # Element reference if composite
        self.low = low      # Lower bound
        self.high = high    # Upper bound
        self.elsz = elsz    # Element size
        self.size = size    # Total array size

    def __str__(self):
        return (f"Array(xtyp={self.xtyp}, etyp={self.etyp},
eref={self.eref}, "
                f"low={self.low}, high={self.high},
elsz={self.elsz}, size={self.size})")


class BlockTableEntry:
    def __init__(self, last=0, lpar=0, psze=0, vsze=0):
        self.last = last    # Last identifier in block
        self.lpar = lpar    # Last parameter
        self.psze = psze    # Parameter size
        self.vsze = vsze    # Variable size

    def __str__(self):
        return f"Block(last={self.last}, lpar={self.lpar},
psze={self.psze}, vsze={self.vsze})"


class SymbolTable:
    # Object types (obj field)
    OBJ_CONSTANT = 1
    OBJ_VARIABLE = 2
    OBJ_TYPE = 3
    OBJ_PROCEDURE = 4
    OBJ_FUNCTION = 5
    OBJ_PROGRAM = 6

    # Basic types (type field) - starting after reserved
words (index 29+)
    TYPE_NOTYPE = 0
    TYPE_INTEGER = 1
```

```
    TYPE_REAL = 2
    TYPE_BOOLEAN = 3
    TYPE_CHAR = 4
    TYPE_ARRAY = 5
    TYPE_RECORD = 6

    # Type sizes
    TYPE_SIZES = {
        TYPE_INTEGER: 4,
        TYPE_REAL: 8,
        TYPE_BOOLEAN: 1,
        TYPE_CHAR: 1,
    }

    def __init__(self):
        self.tab = []   # Main identifier table
        self.atab = []  # Array table
        self.btab = []  # Block table

        self.current_level = 0
        self.current_block = 0
        self.display = [0]  # Display register for scope
management
        self.global_data_size = 0  # Track global variable
offsets

        # Initialize with reserved words and built-in types
(indices 0-28)
        self._init_reserved_words()

    def _init_reserved_words(self):
        """Initialize reserved words (indices 0-28)"""
        reserved = [
            "program", "variabel", "var", "mulai", "selesai",
"jika", "maka",
            "selain_itu", "selama", "lakukan", "untuk", "ke",
"turun-ke",
            "integer", "real", "boolean", "char", "larik",
"dari", "prosedur",
            "fungsi", "konstanta", "tipe", "rekaman",
"sampai", "ulangi",
            "bagi", "mod", "dan", "atau", "tidak"
```

```
        ]

        for i, word in enumerate(reserved):
            # Reserved words have special obj type
            self.tab.append(SymbolTableEntry(
                name=word,
                obj=0,  # Reserved word
                typ=self.TYPE_NOTYPE,
                lev=0
            ))

        # Add predefined procedures
        predefined_procs = ['write', 'writeln', 'read',
'readln']
        for proc in predefined_procs:
            self.tab.append(SymbolTableEntry(
                name=proc,
                obj=self.OBJ_PROCEDURE,
                typ=self.TYPE_NOTYPE,
                lev=0,
                link=0
            ))

    def enter_scope(self):
        self.current_level += 1
        block_idx = len(self.btab)
        self.btab.append(BlockTableEntry())
        self.display.append(len(self.tab))
        self.current_block = block_idx
        return block_idx

    def exit_scope(self):
        if self.current_level > 0:
            self.current_level -= 1
            self.display.pop()
            if len(self.display) > 0:
                # Find previous block
                for i in range(len(self.btab) - 1, -1, -1):
                    if i < len(self.btab):
                        self.current_block = i
                        break
```

```
    def lookup(self, name, level=None):
        if level is None:
            level = self.current_level

        # Search from current level backwards
        for lev in range(level, -1, -1):
            if lev < len(self.display):
                start_idx = self.display[lev]
                for i in range(len(self.tab) - 1, start_idx -
1, -1):
                    if self.tab[i].name == name and
self.tab[i].lev == lev:
                        return i, self.tab[i]

        return None, None

    def lookup_current_scope(self, name):
        if self.current_level < len(self.display):
            start_idx = self.display[self.current_level]
            for i in range(len(self.tab) - 1, start_idx - 1,
-1):
                if self.tab[i].name == name and
self.tab[i].lev == self.current_level:
                    return i, self.tab[i]
        return None, None

    def enter(self, name, obj, typ, ref=0, nrm=1, adr=0,
lev=None):
        # Use provided level or current level
        level = lev if lev is not None else
self.current_level

        # Get link to previous identifier in this scope
        link = 0

        # Only create links for identifiers within the same
block/scope
        # Link points to previous identifier of similar kind
(not programs)
        if level < len(self.display):
            start_idx = self.display[level]
            if len(self.tab) > start_idx:
```

```
                # For Level 0: only link
variables/constants/types (not program name)
                # For Level > 0: link all identifiers in that
block
                if level == 0:
                    # At global level, only link identifiers
of the same category (exclude OBJ_PROGRAM)
                    for i in range(len(self.tab) - 1,
start_idx - 1, -1):
                        if (self.tab[i].lev == level and i >=
35 and
                            self.tab[i].obj !=
self.OBJ_PROGRAM):  # Don't link to program
                            link = i
                            break
                else:
                    # In nested scopes, link to the most
recent entry in this scope
                    for i in range(len(self.tab) - 1,
start_idx - 1, -1):
                        if self.tab[i].lev == level:
                            link = i
                            break

        entry = SymbolTableEntry(
            name=name,
            obj=obj,
            typ=typ,
            ref=ref,
            nrm=nrm,
            lev=level,
            adr=adr,
            link=link
        )

        self.tab.append(entry)
        idx = len(self.tab) - 1

        # Update block table's last pointer
        # Only update if the identifier's level is > 0 (i.e.,
it's inside a procedure/function)
        if level > 0:
```

```
            # If lev was explicitly provided and is less than
current_level,
            # we're declaring something in parent scope (like
a nested procedure/function name)
            # In that case, update the parent block's last
pointer
            if lev is not None and level <
self.current_level:
                # This identifier belongs to parent scope,
update parent block
                # The parent block is current_block - 1 (we
just entered a new scope)
                parent_block = self.current_block - 1
                if parent_block >= 0 and parent_block <
len(self.btab):
                    self.btab[parent_block].last = idx
            elif self.current_block < len(self.btab):
                # Normal case: update current block
                self.btab[self.current_block].last = idx

        return idx

    def enter_array(self, xtyp, etyp, eref, low, high):
        elsz = self.TYPE_SIZES.get(etyp, 1)
        if eref > 0:  # Composite element type
            if eref < len(self.atab):
                elsz = self.atab[eref].size

        size = (high - low + 1) * elsz

        entry = ArrayTableEntry(
            xtyp=xtyp,
            etyp=etyp,
            eref=eref,
            low=low,
            high=high,
            elsz=elsz,
            size=size
        )

        self.atab.append(entry)
        return len(self.atab) - 1
```

```
    def enter_record(self, fields):
        # Create block entry for record
        btab_idx = len(self.btab)
        self.btab.append(BlockTableEntry())

        offset = 0
        last_field_idx = 0

        for field in fields:
            field_name = field.get("name", "")
            field_type = field.get("type", "integer")

            # Get type code
            if field_type in ['integer', 'real', 'boolean',
'char']:
                type_map = {
                    'integer': self.TYPE_INTEGER,
                    'real': self.TYPE_REAL,
                    'boolean': self.TYPE_BOOLEAN,
                    'char': self.TYPE_CHAR
                }
                type_code = type_map.get(field_type,
self.TYPE_INTEGER)
            else:
                type_code = self.TYPE_INTEGER  # Default

            field_size = self.TYPE_SIZES.get(type_code, 4)

            # Enter field into tab as a special variable
            # Fields have obj=OBJ_VARIABLE, but are linked
through btab
            # Use lev=current_level+1 so they don't interfere
with normal identifier lookup
            # (fields are conceptually in a nested scope
within the record type)
            idx = len(self.tab)
            entry = SymbolTableEntry(
                name=field_name,
                obj=self.OBJ_VARIABLE,
                typ=type_code,
                ref=0,
```

```
                nrm=1,
                lev=self.current_level + 1,  # Fields at
level beyond current (not searchable via normal lookup)
                adr=offset,
                link=last_field_idx if last_field_idx > 0
else 0
            )
            self.tab.append(entry)

            last_field_idx = idx
            offset += field_size

        # Update btab entry
        self.btab[btab_idx].last = last_field_idx
        self.btab[btab_idx].lpar = 0  # No parameters
        self.btab[btab_idx].psze = 0  # No parameter size
        self.btab[btab_idx].vsze = offset  # Total record
size

        return btab_idx

    def get_type_size(self, typ, ref=0):
        """Get size of a type"""
        if typ in self.TYPE_SIZES:
            return self.TYPE_SIZES[typ]
        elif typ == self.TYPE_ARRAY and ref < len(self.atab):
            return self.atab[ref].size
        elif typ == self.TYPE_RECORD and ref <
len(self.btab):
            return self.btab[ref].vsze  # Record size stored
in vsze
        return 1

    def to_string(self):
        """Generate formatted string representation of symbol
tables"""
        result = []

        # Main symbol table (tab)
        result.append("=" * 120)
        result.append("SYMBOL TABLE (tab)")
        result.append("=" * 120)
```

```
        result.append(f"{'Index':<6} {'Name':<15} {'Obj':<6}
{'Type':<6} {'Ref':<6} {'Nrm':<6} {'Lev':<6} {'Adr':<6}
{'Link':<6}")
        result.append("-" * 120)

        for i, entry in enumerate(self.tab):
            result.append(
                f"{i:<6} {entry.name:<15} {entry.obj:<6}
{entry.type:<6} "
                f"{entry.ref:<6} {entry.nrm:<6}
{entry.lev:<6} {entry.adr:<6} {entry.link:<6}"
            )

        # Array table (atab)
        if self.atab:
            result.append("\n" + "=" * 120)
            result.append("ARRAY TABLE (atab)")
            result.append("=" * 120)
            result.append(f"{'Index':<6} {'XTyp':<10}
{'ETyp':<10} {'ERef':<6} {'Low':<6} {'High':<6} {'ElSz':<6}
{'Size':<6}")
            result.append("-" * 120)

            for i, entry in enumerate(self.atab):
                xtyp_name = self._type_to_string(entry.xtyp)
                etyp_name = self._type_to_string(entry.etyp)
                result.append(
                    f"{i:<6} {xtyp_name:<10} {etyp_name:<10}
{entry.eref:<6} "
                    f"{entry.low:<6} {entry.high:<6}
{entry.elsz:<6} {entry.size:<6}"
                )

        # Block table (btab)
        if self.btab:
            result.append("\n" + "=" * 120)
            result.append("BLOCK TABLE (btab)")
            result.append("=" * 120)
            result.append(f"{'Index':<6} {'Last':<6}
{'LPar':<6} {'PSize':<6} {'VSize':<6}")
            result.append("-" * 120)
```

```
            for i, entry in enumerate(self.btab):
                result.append(
                    f"{i:<6} {entry.last:<6} {entry.lpar:<6}
{entry.psze:<6} {entry.vsze:<6}"
                )

        return "\n".join(result)

    def _obj_to_string(self, obj):
        """Convert object type to string"""
        obj_map = {
            0: "RESV",
            self.OBJ_CONSTANT: "CONST",
            self.OBJ_VARIABLE: "VAR",
            self.OBJ_TYPE: "TYPE",
            self.OBJ_PROCEDURE: "PROC",
            self.OBJ_FUNCTION: "FUNC",
            self.OBJ_PROGRAM: "PROG",
        }
        return obj_map.get(obj, str(obj))

    def _type_to_string(self, typ):
        """Convert type to string"""
        type_map = {
            self.TYPE_NOTYPE: "NONE",
            self.TYPE_INTEGER: "INT",
            self.TYPE_REAL: "REAL",
            self.TYPE_BOOLEAN: "BOOL",
            self.TYPE_CHAR: "CHAR",
            self.TYPE_ARRAY: "ARRAY",
            self.TYPE_RECORD: "REC",
        }
        return type_map.get(typ, str(typ))

    def save_to_file(self, filepath):
        """Save symbol table to file"""
        with open(filepath, 'w', encoding='utf-8') as f:
            f.write(self.to_string())
```

Penjelasan

File *symbol_table* mengandung kelas untuk merepresentasikan *tab, atab,* dan *btab*. Table ini menahan scope yang sedang ditelurusi *decorator* beserta mencatat

objek-objek yang telah dideklarasikan. Fungsi utama yang digunakan oleh *decorator* adalah enter() untuk mencatat sebuah objek baru pada *tab*, enter_array() untuk mencatat sebuah array baru pada *atab*, enter_record() untuk mencatat ke *btab*. Fungsi lookup() melakukan pencarian objek, dan lookup_current_scope() melakukan pencarian objek yang dibatasi pada scope tertentu. Kedua fungsi tersebut berperan sebagai validator utama.

PENGUJIAN

1. **test1.pas**

Source Code

```
program SimpleCalculation;

variabel
    x, y, z: integer;
    result: real;

mulai
    x := 10;
    y := 20;
    z := x + y;
    result := 3.14
selesai.
```

Decorated AST

```
ProgramNode(name: 'simplecalculation') → tab_index:35
  ├─ DeclarationsNode
  │   ├─ VarDeclNode('x') → tab_index:36, type:integer, lev:0
  │   ├─ VarDeclNode('y') → tab_index:37, type:integer, lev:0
  │   ├─ VarDeclNode('z') → tab_index:38, type:integer, lev:0
  │   └─ VarDeclNode('result') → tab_index:39, type:real, lev:0
  └─ BlockNode
      ├─ Assign(':= ...') → type:integer, tab_index:36
      │   └─ Expression → type:integer
      │       └─ SimpleExpression → type:integer
      │           └─ Term → type:integer
      │               └─ NumberLiteral(10) → type:integer
      ├─ Assign(':= ...') → type:integer, tab_index:37
      │   └─ Expression → type:integer
      │       └─ SimpleExpression → type:integer
      │           └─ Term → type:integer
      │               └─ NumberLiteral(20) → type:integer
      ├─ Assign(':= ...') → type:integer, tab_index:38
      │   └─ Expression → type:integer
      │       └─ SimpleExpression(ops: ['+']) → type:integer
      │           ├─ Term → type:integer
      │           │   └─ Identifier('x') → tab_index:36, type:integer
      │           └─ Term → type:integer
      │               └─ Identifier('y') → tab_index:37, type:integer
      │
```

```
    └─ Assign('result' := ...) → type:real, tab_index:39
      └─ Expression → type:real
        └─ SimpleExpression → type:real
          └─ Term → type:real
            └─ NumberLiteral(3.14) → type:real
```

Symbol Table

```
================================================================
SYMBOL TABLE (tab)
================================================================
Index  Name          Obj    Type   Ref    Nrm    Lev    Adr    Link
----------------------------------------------------------------------------------------------------
0      program       0      0      0      1      0      0      0
1      variabel      0      0      0      1      0      0      0
2      var           0      0      0      1      0      0      0
3      mulai         0      0      0      1      0      0      0
4      selesai       0      0      0      1      0      0      0
5      jika          0      0      0      1      0      0      0
6      maka          0      0      0      1      0      0      0
7      selain_itu    0      0      0      1      0      0      0
8      selama        0      0      0      1      0      0      0
9      lakukan       0      0      0      1      0      0      0
10     untuk         0      0      0      1      0      0      0
11     ke            0      0      0      1      0      0      0
12     turun-ke      0      0      0      1      0      0      0
13     integer       0      0      0      1      0      0      0
14     real          0      0      0      1      0      0      0
15     boolean       0      0      0      1      0      0      0
16     char          0      0      0      1      0      0      0
17     larik         0      0      0      1      0      0      0
18     dari          0      0      0      1      0      0      0
19     prosedur      0      0      0      1      0      0      0
20     fungsi        0      0      0      1      0      0      0
21     konstanta     0      0      0      1      0      0      0
22     tipe          0      0      0      1      0      0      0
23     rekaman       0      0      0      1      0      0      0
24     sampai        0      0      0      1      0      0      0
25     ulangi        0      0      0      1      0      0      0
26     bagi          0      0      0      1      0      0      0
27     mod           0      0      0      1      0      0      0
28     dan           0      0      0      1      0      0      0
29     atau          0      0      0      1      0      0      0
30     tidak         0      0      0      1      0      0      0
31     write         4      0      0      1      0      0      0
32     writeln       4      0      0      1      0      0      0
```

```
33   read          4   0   0   1   0   0   0
34   readln        4   0   0   1   0   0   0
35   simplecalculation 6   0   0   1   0   0   0
36   x             2   1   0   1   0   0   0
37   y             2   1   0   1   0   4   36
38   z             2   1   0   1   0   8   37
39   result        2   2   0   1   0   12  38
```

2. **test2.pas**

Source Code

```
program ArraysAndTypes;

tipe
    IntArray = larik[1..10] dari integer;
    RealArray = larik[0..5] dari real;
    CharArray = larik[1..20] dari char;

variabel
    numbers: IntArray;
    scores: RealArray;
    name: CharArray;
    count: integer;

mulai
    count := 5
selesai.
```

Decorated AST

```
ProgramNode(name: 'arraysandtypes') → tab_index:35
  ├── DeclarationsNode
  │    ├── TypeDeclNode('intarray') → tab_index:36, type:array, lev:0
  │    ├── TypeDeclNode('realarray') → tab_index:37, type:array, lev:0
  │    ├── TypeDeclNode('chararray') → tab_index:38, type:array, lev:0
  │    ├── VarDeclNode('numbers') → tab_index:39, type:array, lev:0
  │    ├── VarDeclNode('scores') → tab_index:40, type:array, lev:0
  │    ├── VarDeclNode('name') → tab_index:41, type:array, lev:0
  │    └── VarDeclNode('count') → tab_index:42, type:integer, lev:0
  └── BlockNode
       └── Assign('count' := ...) → type:integer, tab_index:42
            └── Expression → type:integer
                 └── SimpleExpression → type:integer
                      └── Term → type:integer
```

└─ NumberLiteral(5) → type:integer

Symbol Table

```
==========================================================================
SYMBOL TABLE (tab)
==========================================================================
Index  Name          Obj    Type   Ref    Nrm    Lev    Adr    Link
--------------------------------------------------------------------------------------------------------
0      program       0      0      0      1      0      0      0
1      variabel      0      0      0      1      0      0      0
2      var           0      0      0      1      0      0      0
3      mulai         0      0      0      1      0      0      0
4      selesai       0      0      0      1      0      0      0
5      jika          0      0      0      1      0      0      0
6      maka          0      0      0      1      0      0      0
7      selain_itu    0      0      0      1      0      0      0
8      selama        0      0      0      1      0      0      0
9      lakukan       0      0      0      1      0      0      0
10     untuk         0      0      0      1      0      0      0
11     ke            0      0      0      1      0      0      0
12     turun-ke      0      0      0      1      0      0      0
13     integer       0      0      0      1      0      0      0
14     real          0      0      0      1      0      0      0
15     boolean       0      0      0      1      0      0      0
16     char          0      0      0      1      0      0      0
17     larik         0      0      0      1      0      0      0
18     dari          0      0      0      1      0      0      0
19     prosedur      0      0      0      1      0      0      0
20     fungsi        0      0      0      1      0      0      0
21     konstanta     0      0      0      1      0      0      0
22     tipe          0      0      0      1      0      0      0
23     rekaman       0      0      0      1      0      0      0
24     sampai        0      0      0      1      0      0      0
25     ulangi        0      0      0      1      0      0      0
26     bagi          0      0      0      1      0      0      0
27     mod           0      0      0      1      0      0      0
28     dan           0      0      0      1      0      0      0
29     atau          0      0      0      1      0      0      0
30     tidak         0      0      0      1      0      0      0
31     write         4      0      0      1      0      0      0
32     writeln       4      0      0      1      0      0      0
33     read          4      0      0      1      0      0      0
34     readln        4      0      0      1      0      0      0
35     arraysandtypes 6     0      0      1      0      0      0
36     intarray      3      5      0      1      0      0      0
37     realarray     3      5      1      1      0      0      36
```

```
38    chararray     3    5    2    1    0    0    37
39    numbers       2    5    0    1    0    0    38
40    scores        2    5    1    1    0    40   39
41    name          2    5    2    1    0    88   40
42    count         2    1    0    1    0    108  41

============================================================
ARRAY TABLE (atab)
============================================================
Index  XTyp     ETyp     ERef   Low    High   ElSz   Size
------------------------------------------------------------------------------------------------------
0      1        1        0      1      10     4      40
1      1        2        0      0      5      8      48
2      1        4        0      1      20     1      20
```

3. test3.pas

Source Code

```
program RecordTypes;

tipe
    Point = rekaman
        x: integer;
        y: integer;
    selesai;

    Person = rekaman
        name: char;
        age: integer;
        height: real;
        active: boolean;
    selesai;

variabel
    p1, p2: Point;
    student: Person;
    teacher: Person;
    counter: integer;

mulai
    counter := 0
selesai.
```

Decorated AST

```
ProgramNode(name: 'recordtypes') → tab_index:35
  ├─ DeclarationsNode
  │   ├─ TypeDeclNode('point') → tab_index:38, type:record, lev:0
  │   ├─ TypeDeclNode('person') → tab_index:43, type:record, lev:0
  │   ├─ VarDeclNode('p1') → tab_index:44, type:record, lev:0
  │   ├─ VarDeclNode('p2') → tab_index:45, type:record, lev:0
  │   ├─ VarDeclNode('student') → tab_index:46, type:record, lev:0
  │   ├─ VarDeclNode('teacher') → tab_index:47, type:record, lev:0
  │   └─ VarDeclNode('counter') → tab_index:48, type:integer, lev:0
  └─ BlockNode
      └─ Assign('counter' := ...) → type:integer, tab_index:48
          └─ Expression → type:integer
              └─ SimpleExpression → type:integer
                  └─ Term → type:integer
                      └─ NumberLiteral(0) → type:integer
```

Symbol Table

```
================================================================
SYMBOL TABLE (tab)
================================================================
Index  Name          Obj    Type   Ref    Nrm    Lev    Adr    Link
----------------------------------------------------------------------------------------------------
0      program       0      0      0      1      0      0      0
1      variabel      0      0      0      1      0      0      0
2      var           0      0      0      1      0      0      0
3      mulai         0      0      0      1      0      0      0
4      selesai       0      0      0      1      0      0      0
5      jika          0      0      0      1      0      0      0
6      maka          0      0      0      1      0      0      0
7      selain_itu    0      0      0      1      0      0      0
8      selama        0      0      0      1      0      0      0
9      lakukan       0      0      0      1      0      0      0
10     untuk         0      0      0      1      0      0      0
11     ke            0      0      0      1      0      0      0
12     turun-ke      0      0      0      1      0      0      0
13     integer       0      0      0      1      0      0      0
14     real          0      0      0      1      0      0      0
15     boolean       0      0      0      1      0      0      0
16     char          0      0      0      1      0      0      0
17     larik         0      0      0      1      0      0      0
18     dari          0      0      0      1      0      0      0
19     prosedur      0      0      0      1      0      0      0
20     fungsi        0      0      0      1      0      0      0
21     konstanta     0      0      0      1      0      0      0
22     tipe          0      0      0      1      0      0      0
```

```
23    rekaman       0    0    0    1    0    0    0
24    sampai        0    0    0    1    0    0    0
25    ulangi        0    0    0    1    0    0    0
26    bagi          0    0    0    1    0    0    0
27    mod           0    0    0    1    0    0    0
28    dan           0    0    0    1    0    0    0
29    atau          0    0    0    1    0    0    0
30    tidak         0    0    0    1    0    0    0
31    write         4    0    0    1    0    0    0
32    writeln       4    0    0    1    0    0    0
33    read          4    0    0    1    0    0    0
34    readln        4    0    0    1    0    0    0
35    recordtypes   6    0    0    1    0    0    0
36    x             2    1    0    1    1    0    0
37    y             2    1    0    1    1    4    36
38    point         3    6    0    1    0    0    0
39    name          2    4    0    1    1    0    0
40    age           2    1    0    1    1    1    39
41    height        2    2    0    1    1    5    40
42    active        2    3    0    1    1    13   41
43    person        3    6    1    1    0    0    38
44    p1            2    6    0    1    0    0    43
45    p2            2    6    0    1    0    8    44
46    student       2    6    1    1    0    16   45
47    teacher       2    6    1    1    0    30   46
48    counter       2    1    0    1    0    44   47

================================================================
BLOCK TABLE (btab)
================================================================
Index  Last   LPar   PSize  VSize
----------------------------------------------------------------------------------------------------
0      37     0      0      8
1      42     0      0      14
```

4. **test4.pas**

Source Code

```
program ProceduresAndFunctions;

variabel
   a, b: integer;
   sum: integer;

prosedur swap(var x: integer; var y: integer);
variabel
```

```
    temp: integer;
mulai
    temp := x;
    x := y;
    y := temp
selesai;

fungsi add(p: integer; q: integer): integer;
variabel
    result: integer;
mulai
    result := p + q
selesai;

mulai
    a := 10;
    b := 20;
    sum := 0
selesai.
```

Decorated AST

```
ProgramNode(name: 'proceduresandfunctions') → tab_index:35
  ├─ DeclarationsNode
  │   ├─ VarDeclNode('a') → tab_index:36, type:integer, lev:0
  │   ├─ VarDeclNode('b') → tab_index:37, type:integer, lev:0
  │   ├─ VarDeclNode('sum') → tab_index:38, type:integer, lev:0
  │   ├─ ProcedureDeclNode('swap') → tab_index:39, block_index:0
  │   │   ├─ ParameterNode
  │   │   ├─ ParameterNode
  │   │   └─ VarDeclNode('temp') → tab_index:42, type:integer, lev:1
  │   └─ FunctionDeclNode('add') → tab_index:43, block_index:1,
return_type:integer
  │       ├─ ParameterNode
  │       ├─ ParameterNode
  │       └─ VarDeclNode('result') → tab_index:46, type:integer, lev:1
  └─ BlockNode
      ├─ Assign('a' := ...) → type:integer, tab_index:36
      │   └─ Expression → type:integer
      │       └─ SimpleExpression → type:integer
      │           └─ Term → type:integer
      │               └─ NumberLiteral(10) → type:integer
      ├─ Assign('b' := ...) → type:integer, tab_index:37
      │   └─ Expression → type:integer
      │       └─ SimpleExpression → type:integer
      │           └─ Term → type:integer
```

```
    │           └─ NumberLiteral(20) → type:integer
    └─ Assign('sum' := ...) → type:integer, tab_index:38
      └─ Expression → type:integer
        └─ SimpleExpression → type:integer
          └─ Term → type:integer
            └─ NumberLiteral(0) → type:integer
```

Symbol Table

```
================================================================
SYMBOL TABLE (tab)
================================================================
Index  Name          Obj    Type   Ref    Nrm    Lev    Adr    Link
----------------------------------------------------------------------------------------------------
0      program       0      0      0      1      0      0      0
1      variabel      0      0      0      1      0      0      0
2      var           0      0      0      1      0      0      0
3      mulai         0      0      0      1      0      0      0
4      selesai       0      0      0      1      0      0      0
5      jika          0      0      0      1      0      0      0
6      maka          0      0      0      1      0      0      0
7      selain_itu    0      0      0      1      0      0      0
8      selama        0      0      0      1      0      0      0
9      lakukan       0      0      0      1      0      0      0
10     untuk         0      0      0      1      0      0      0
11     ke            0      0      0      1      0      0      0
12     turun-ke      0      0      0      1      0      0      0
13     integer       0      0      0      1      0      0      0
14     real          0      0      0      1      0      0      0
15     boolean       0      0      0      1      0      0      0
16     char          0      0      0      1      0      0      0
17     larik         0      0      0      1      0      0      0
18     dari          0      0      0      1      0      0      0
19     prosedur      0      0      0      1      0      0      0
20     fungsi        0      0      0      1      0      0      0
21     konstanta     0      0      0      1      0      0      0
22     tipe          0      0      0      1      0      0      0
23     rekaman       0      0      0      1      0      0      0
24     sampai        0      0      0      1      0      0      0
25     ulangi        0      0      0      1      0      0      0
26     bagi          0      0      0      1      0      0      0
27     mod           0      0      0      1      0      0      0
28     dan           0      0      0      1      0      0      0
29     atau          0      0      0      1      0      0      0
30     tidak         0      0      0      1      0      0      0
31     write         4      0      0      1      0      0      0
32     writeln       4      0      0      1      0      0      0
33     read          4      0      0      1      0      0      0
```

```
34    readln          4     0     0     1     0     0     0
35    proceduresandfunctions 6      0     0     1     0     0     0
36    a               2     1     0     1     0     0     0
37    b               2     1     0     1     0     4     36
38    sum             2     1     0     1     0     8     37
39    swap            4     0     0     1     0     0     38
40    x               2     1     0     0     1     0     0
41    y               2     1     0     0     1     4     40
42    temp            2     1     0     1     1     0     41
43    add             5     1     1     1     0     0     39
44    p               2     1     0     1     1     0     0
45    q               2     1     0     1     1     4     44
46    result          2     1     0     1     1     0     45

================================================================
BLOCK TABLE (btab)
================================================================
Index  Last   LPar   PSize  VSize
----------------------------------------------------------------------------------------------------
0     42    41    8     4
1     46    45    8     4
```

5. **test5.pas**

Source Code

```
program ComplexProgram;

konstanta
   MAX = 100;
   PI = 3.14159;

tipe
   Status = boolean;
   Age = integer;
   Score = real;

   StudentRecord = rekaman
      id: integer;
      grade: char;
      gpa: real;
   selesai;

   NumberList = larik[1..50] dari integer;
```

```
variabel
    isActive: Status;
    studentAge: Age;
    testScore: Score;
    student1: StudentRecord;
    numbers: NumberList;
    total: integer;

prosedur initialize;
mulai
    total := 0
selesai;

fungsi calculate(x: integer; y: integer): integer;
variabel
    temp: integer;
mulai
    temp := x + y
selesai;

fungsi average(count: integer; sum: real): real;
variabel
    result: real;
mulai
    result := sum
selesai;

mulai
    studentAge := 20;
    testScore := 95.5;
    total := 0
selesai.
```

Decorated AST

```
Decorated AST:
================================================================
======================

ProgramNode(name: 'complexprogram') → tab_index:35
  ├─ DeclarationsNode
  │   ├─ ConstDeclNode('max') → tab_index:36, type:integer, lev:0
  │   ├─ ConstDeclNode('pi') → tab_index:37, type:integer, lev:0
```

```
  │  ├─ TypeDeclNode('status') → tab_index:38, type:boolean, lev:0
  │  ├─ TypeDeclNode('age') → tab_index:39, type:integer, lev:0
  │  ├─ TypeDeclNode('score') → tab_index:40, type:real, lev:0
  │  ├─ TypeDeclNode('studentrecord') → tab_index:44, type:record, lev:0
  │  ├─ TypeDeclNode('numberlist') → tab_index:45, type:array, lev:0
  │  ├─ VarDeclNode('isactive') → tab_index:46, type:boolean, lev:0
  │  ├─ VarDeclNode('studentage') → tab_index:47, type:integer, lev:0
  │  ├─ VarDeclNode('testscore') → tab_index:48, type:real, lev:0
  │  ├─ VarDeclNode('student1') → tab_index:49, type:record, lev:0
  │  ├─ VarDeclNode('numbers') → tab_index:50, type:array, lev:0
  │  ├─ VarDeclNode('total') → tab_index:51, type:integer, lev:0
  │  ├─ ProcedureDeclNode('initialize') → tab_index:52, block_index:1
  │  ├─ FunctionDeclNode('calculate') → tab_index:53, block_index:2,
return_type:integer
  │  │  ├─ ParameterNode
  │  │  ├─ ParameterNode
  │  │  └─ VarDeclNode('temp') → tab_index:56, type:integer, lev:1
  │  └─ FunctionDeclNode('average') → tab_index:57, block_index:3,
return_type:real
  │     ├─ ParameterNode
  │     ├─ ParameterNode
  │     └─ VarDeclNode('result') → tab_index:60, type:real, lev:1
  └─ BlockNode
     ├─ Assign('studentage' := ...) → type:integer, tab_index:47
     │  └─ Expression → type:integer
     │     └─ SimpleExpression → type:integer
     │        └─ Term → type:integer
     │           └─ NumberLiteral(20) → type:integer
     ├─ Assign('testscore' := ...) → type:real, tab_index:48
     │  └─ Expression → type:real
     │     └─ SimpleExpression → type:real
     │        └─ Term → type:real
     │           └─ NumberLiteral(95.5) → type:real
     └─ Assign('total' := ...) → type:integer, tab_index:51
        └─ Expression → type:integer
           └─ SimpleExpression → type:integer
              └─ Term → type:integer
                 └─ NumberLiteral(0) → type:integer
```

Symbol Table

```
================================================================
SYMBOL TABLE (tab)
================================================================
Index  Name         Obj    Type   Ref    Nrm    Lev    Adr    Link
----------------------------------------------------------------------------------------------------
0      program      0      0      0      1      0      0      0
```

1	variabel	0	0	0	1	0	0	0
2	var	0	0	0	1	0	0	0
3	mulai	0	0	0	1	0	0	0
4	selesai	0	0	0	1	0	0	0
5	jika	0	0	0	1	0	0	0
6	maka	0	0	0	1	0	0	0
7	selain_itu	0	0	0	1	0	0	0
8	selama	0	0	0	1	0	0	0
9	lakukan	0	0	0	1	0	0	0
10	untuk	0	0	0	1	0	0	0
11	ke	0	0	0	1	0	0	0
12	turun-ke	0	0	0	1	0	0	0
13	integer	0	0	0	1	0	0	0
14	real	0	0	0	1	0	0	0
15	boolean	0	0	0	1	0	0	0
16	char	0	0	0	1	0	0	0
17	larik	0	0	0	1	0	0	0
18	dari	0	0	0	1	0	0	0
19	prosedur	0	0	0	1	0	0	0
20	fungsi	0	0	0	1	0	0	0
21	konstanta	0	0	0	1	0	0	0
22	tipe	0	0	0	1	0	0	0
23	rekaman	0	0	0	1	0	0	0
24	sampai	0	0	0	1	0	0	0
25	ulangi	0	0	0	1	0	0	0
26	bagi	0	0	0	1	0	0	0
27	mod	0	0	0	1	0	0	0
28	dan	0	0	0	1	0	0	0
29	atau	0	0	0	1	0	0	0
30	tidak	0	0	0	1	0	0	0
31	write	4	0	0	1	0	0	0
32	writeln	4	0	0	1	0	0	0
33	read	4	0	0	1	0	0	0
34	readln	4	0	0	1	0	0	0
35	complexprogram	6	0	0	1	0	0	0
36	max	1	1	0	1	0	100	0
37	pi	1	1	0	1	0	3	36
38	status	3	3	0	1	0	0	37
39	age	3	1	0	1	0	0	38
40	score	3	2	0	1	0	0	39
41	id	2	1	0	1	1	0	0
42	grade	2	4	0	1	1	4	41
43	gpa	2	2	0	1	1	5	42
44	studentrecord	3	6	0	1	0	0	40
45	numberlist	3	5	0	1	0	0	44
46	isactive	2	3	0	1	0	0	45
47	studentage	2	1	0	1	0	1	46
48	testscore	2	2	0	1	0	5	47
49	student1	2	6	0	1	0	13	48

```
50    numbers       2     5     0     1     0     26    49
51    total         2     1     0     1     0     226   50
52    initialize    4     0     1     1     0     0     51
53    calculate     5     1     2     1     0     0     52
54    x             2     1     0     1     1     0     0
55    y             2     1     0     1     1     4     54
56    temp          2     1     0     1     1     0     55
57    average       5     2     3     1     0     0     53
58    count         2     1     0     1     1     0     0
59    sum           2     2     0     1     1     4     58
60    result        2     2     0     1     1     0     59

================================================================
ARRAY TABLE (atab)
================================================================
Index  XTyp      ETyp      ERef   Low    High   ElSz   Size
----------------------------------------------------------------------------------------------------
0      1         1         0      1      50     4      200

================================================================
BLOCK TABLE (btab)
================================================================
Index  Last   LPar   PSize  VSize
----------------------------------------------------------------------------------------------------
0      43     0      0      13
1      0      0      0      0
2      56     55     8      4
3      60     59     12     8
```

KESIMPULAN DAN SARAN

Pada tahap ini, *Semantic Analyzer* melalui pembentukan *Abstract Syntax Tree (AST)* dan pengisian *symbol table* merupakan tahap analisis tingkat selanjutnya setelah *parser* yang memeriksa struktur deklarasi tipe, fungsi, dan variabel dalam scope-scope berbeda. Analisis ini meliputi pemeriksaan pemanggilan fungsi atau variabel bahwa hal tersebut merupakan suatu hal yang telah terdefinisi atau tidak. Logika program dilakukan melalui proses dekorasi AST dan pengisian *symbol table*.

Dari pengujian yang telah dilalui, pembentukan AST yang telah didekorasi dan juga pengisian *symbol table* sudah berjalan dengan benar.

Tantangan teknis terbesar yang dihadapi dalam milestone ini adalah tata bahasa yang didefinisikan pada tahap awal ternyata belum cukup lengkap untuk menangani kasus-kasus kompleks pada kode Pascal-S. Hal ini mengharuskan dilakukannya perombakan besar pada struktur *grammar* yang telah dibuat. Revisi signifikan sangat diperlukan, contohnya pada bagian penanganan akses elemen array dan akses field pada record yang *nested*. Ambiguitas mengenai tingkat kompleksitas program yang akan dibaca menyebabkan ketidakpastian dalam hal batasan pemrosesan bahasa. Dari itu, kami merasa terpaksa melakukan *step* tambahan untuk memastikan pemrosesan bahasa dapat mengurus kasus-kasus kompleks.

Pengerjaan Tugas Besar Milestone 3 Teori Berbahasa Formal Otomata memperluas wawasan mengenai pemanfaatan lanjut dari Context-Free Grammar (CFG) dan implementasinya dalam sebuah *Semantic Analyzer*. Dalam segi teknis, kami menjadi paham tentang tahap analisis *semantic* dalam proses kompilasi, yang sebelumnya merupakan hal abstrak, kini menjadi lebih nyata melalui pembentukan AST dan pengisian *symbol table*. 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/v0.3.1

Pembagian Tugas	
Julius Arthur (13523030)	Node Class Statement, Builder Statement, laporan
Samuel Gerrard Hamonangan Girsang (13523064)	Program Node Class, AST Builder Implementation, laporan
Nadhif Al-Rozin (13523076)	Node Class Declaration, Builder Declaration, laporan
Lutfi Hakim Yusra (13523084)	Node Class Expression, Builder Expression, Symbol Table, ASTDecorator, laporan

-

REFERENSI

- https://www.bottlecaps.de/ - Language Syntax
- http://pascal.hansotten.com/niklaus-wirth/pascal-s/ - PASCAL-S