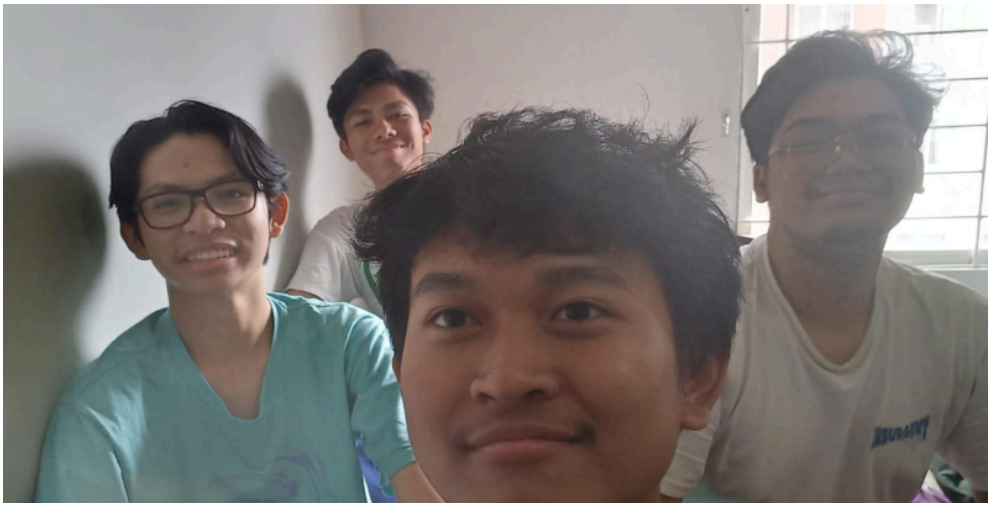
Milestone 3 IF2224 Teori Bahasa Formal dan Otomata
Semantic Analysis Untuk Compiler Bahasa Pascal-S



Kelompok CGK

Andi Farhan Hidayat	13523128
Andri Nurdianto	13523145
Rafael Marchel D. W.	13523146
Muhammad Kinan Arkansyaddad	13523152

Program Studi Teknik informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung

Daftar Isi

Landasan Teori

Setelah dilakukan *syntax analysis*, tahapan yang dibutuhkan selanjutnya adalah *semantic analysis*. *Semantic analysis* menunjukan makna dari ekspresi yang diidentifikasi oleh *syntax analysis*. *Syntax analysis* memastikan bahwa ekspresi yang dihasilkan masuk akal. Tidak hanya susunannya yang benar, tetapi juga susunan tersebut memiliki makna.

Semantic analysis melakukan anotasi pada parse tree dengan fokus untuk memvalidasi dan memperkaya struktur yang dihasilkan oleh parser. *Semantic analysis* memeriksa tipe data, referensi simbol, dan memvalidasi *flow control* (if-else)

Attributed Grammar

Attributed grammar adalah sebuah context free grammar yang diperluas dengan atribut dan aturan semantik. Attributed grammar menggunakan aturan tata bahasa yang memandu penguraian dan evaluasi semantik. Grammar ini menggunakan nilai atribut yang diproses melalui aturan semantik. Beberapa aturan yang disimpan diantaranya, jenis variabel (int, float, char) dan nilai ekspresi (1 + 2 = 3).

Symbol Table

Symbol table adalah struktur data dalam compiler yang menyimpan informasi tentang semua identifier seperti variabel dan fungsi, termasuk juga nama, tipe, dan scope. Tabel ini dibuat ketika setelah lexer membuat token baru, parser menambah informasi atribut dan scope, dan semantik analyzer melakukan pengecekan tipe. Symbol table memastikan penggunaan identifier yang benar dalam proses kompilasi.

Visitor/Visit Function

Visitor/visit function adalah mekanisme yang digunakan dalam compiler. Mekanisme ini berfungsi untuk memisahkan struktur data dari operasi yang dilakukan. Setiap jenis node memiliki fungsi visit_<node> yang bertugas menjalankan operasi tertentu tanpa harus mengubah definisi node. Dalam prosesnya, fungsi visit memanggil visit pada child node, mengakses atau memperbarui symbol table, menghitung atribut atau tipe semantik, dan menambahkan anotasi pada node seperti tipe.

Perancangan

Struktur Program

Struktur program terdiri dari file-file yang digunakan untuk lexical analysis dan parser dengan penambahan komponen semantic analysis dan Abstract Syntax Tree (AST).

```
CGK-TUBES-IF2224
├── Cargo.lock
├── Cargo.toml
├── dfa_rules.json
├── doc
├── LICENSE
├── README.md
├── src
│   ├── ast.rs      # AST Node definitions & display
│   ├── dfa.rs          # DFA untuk lexical analysis
│   ├── errors.rs       # Semantic error handling
│   ├── lexer.rs        # Lexical analyzer
│   ├── main.rs         # Entry point program
│   ├── node.rs               # Parse tree node
│   ├── parser.rs             # Syntax analyzer
│   ├── semantic_analyzer.rs  # Semantic analyzer
│   ├── symbol_table.rs       # Symbol & block tables
│   ├── token.rs              # Token definitions
│   └── types.rs              # Type system
└── test
```

Fungsi/Kelas Utama

Program ini dibuat menggunakan paradigma pemrograman berorientasi objek (struct dan impl di Rust) dengan pendekatan modular. Berikut adalah komponen-komponen yang membentuk semantic analyzer:

Nama	Tipe	Deskripsi	Justifikasi
SemanticAnalyzer	struct	Struct utama yang melakukan analisis semantik dengan menyimpan *reference* ke *symbol table* dan daftar *error*. Mengimplementasikan *pattern Visitor* untuk *traversal Parse Tree*.	Memisahkan logika *semantic analysis* dari parsing memungkinkan modularitas, parser hanya fokus pada struktur sintaksis, sementara semantic analyzer menangani makna dan *type checking*.

SymbolTable	struct	Mengelola tab (identifier table), btab (block table), dan atab (array table). Menyimpan informasi tentang semua identifier, scope, dan tipe data.	Tabel terpisah sesuai spesifikasi Pascal-S memudahkan tracking scope hierarki dan type information dengan efisien menggunakan index-based references.
TabEntry	struct	Entry dalam identifier table yang menyimpan nama, object kind (variable/procedure/etc), tipe data, level, address, dan link untuk chaining.	Backward chaining dengan link memungkinkan efficient scope lookup, setiap identifier menunjuk ke identifier sebelumnya dengan tipe yang sama.
BTabEntry	struct	Entry dalam block table yang menyimpan informasi tentang last identifier, parameter size, dan variable size.	Memisahkan block information memudahkan scope management dan code generation di fase berikutnya.
AstNode	enum	Representasi node dalam Abstract Syntax Tree dengan variant untuk semua konstruksi bahasa (Program, VarDecl, Assign, BinOp, dll). Setiap node menyimpan type information.	Enum di Rust menjamin type safety dan exhaustive pattern matching. Decorated AST dengan type annotations memudahkan code generation dan optimasi.
DataType	enum	Mendefinisikan semua tipe data yang didukung: Integer, Real, Boolean, String, Char, Array, Record, Void, Unknown, dan UserDefined.	Type system yang eksplisit memungkinkan comprehensive type checking dan error detection sebelum runtime.
ObjectKind	enum	Klasifikasi identifier: Variable, Constant, Type, Procedure, Function, atau Program.	Membedakan jenis identifier memungkinkan validation rules yang berbeda (misalnya, constant tidak bisa di-assign).
SemanticError	struct	Error handling khusus untuk kesalahan	Error reporting yang detail dengan context

		semantik dengan kategori: undeclared identifier, type mismatch, redeclaration, dll.	membantu programmer memahami dan memperbaiki kesalahan dengan cepat.
visit_program()	fn	Metode visitor utama yang memproses root program, membuat block untuk scope global dan lokal, serta mendekorasikan AST.	Entry point untuk semantic analysis yang mengatur flow dari deklarasi ke body program dengan proper scope management.
visit_var_declaration()	fn	Memproses deklarasi variabel dengan memasukkan setiap variabel ke symbol table, melakukan redeclaration checking, dan membuat individual VarDecl nodes.	Memisahkan setiap variabel dalam AST (bukan array) memberikan granularity yang lebih baik untuk analysis dan code generation.
visit_expression()	fn	Melakukan type inference dan type checking pada ekspresi. Mengembalikan AstNode dengan type annotation yang dihitung dari operand.	Type inference otomatis mengurangi verbosity sambil tetap memastikan type safety melalui static checking.
lookup()	fn	Mencari identifier dalam symbol table dengan multi-level scope search: current block → parent blocks → predefined procedures.	Implementasi scope resolution sesuai aturan visibility Pascal.
insert()	fn	Memasukkan identifier baru ke symbol table dengan backward chaining, menghubungkan ke identifier sebelumnya dengan tipe yang sama.	Backward chaining membuat traversal scope lebih efisien dan memudahkan implementasi last pointer di btab.
enter_block() / exit_block()	fn	Mengatur stack-based scope management dengan Display array untuk tracking active	Display array memberikan O(1) access ke block information di setiap level tanpa perlu traversal

		blocks di setiap nesting level.	hierarki.
can_assign() / get_arithmetic_result_type()	fn	Implementasi type compatibility rules dan type coercion logic untuk assignment dan operasi aritmatika.	Aturan tipe yang eksplisit mencegah implicit conversions yang berbahaya sambil mengizinkan safe coercions (integer → real).

Alur Kerja Program

Alur kerja program mengikuti pipeline modular dengan tiga fase utama: Lexical → Syntactic → Semantic Analysis. Berikut adalah tahapan eksekusi Milestone 3:

1. **Inisialisasi dan Lexical**
   Program main.rs membaca file .pas, menjalankan Lexer (Milestone 1), dan menghasilkan vektor token (Vec<Token>). Token ini kemudian diproses oleh Parser (Milestone 2) untuk menghasilkan Parse Tree.

2. **Setup Symbol Table**
   Objek SymbolTable diinisialisasi dengan:
   - Reserved words (indices 0-28): Semua keyword Pascal-S
   - Predefined procedures (indices 29-32): writeln, write, readln, read
   - Display array: Diinisialisasi dengan block 0 (global scope)

   User-defined identifiers dimulai dari index 33 ke atas.

3. **Semantic Analysis (Tree Traversal)**
   SemanticAnalyzer dibuat dengan reference ke symbol table. Fungsi analyze() dipanggil dengan Parse Tree sebagai input:
   a. Program Processing (visit_program)
      - Memproses program header dan memasukkan program name ke symbol table
      - Membuat block 0 untuk global scope
      - Memproses deklarasi (variables, constants, types, procedures)
      - Membuat block 1 untuk main compound statement (enter_block())
      - Memproses body program di dalam block 1

- Keluar dari block 1 (exit_block())
- Mengembalikan decorated AST dengan Program node

b. Declaration Processing Untuk setiap deklarasi variabel (visit_var_declaration):
   - Parse tipe data dari Parse Tree
   - Pada setiap identifier dalam list, *check redeclaration* dengan lookup_current_scope() dan insert() ke tabel simbol jika belum ada. Kemudian dilakukan *backward chaining* ke identifier sebelumnya dengan *object type* yang sama, *update* btab[current_block].last ke identifier terbaru, serta buat node AST VarDecl dengan *type annotation*.
   - Update variable size di current block

c. Statement Processing Untuk assignment (visit_assignment_statement):
   - Lookup target variable di symbol table
   - Visit value expression (rekursif)
   - Type checking dengan verifikasi compatibility melalui DataType::can_assign()
   - Jika mismatch, tambahkan SemanticError::TypeMismatch
   - Buat Assign node dengan type annotation

   Untuk expression (visit_expression):
   - Rekursif process operands (left dan right)
   - Type inference: Tentukan result type dari operator dan operand types
   - Validasi type compatibility untuk operator
   - Propagate type information ke parent node

d. Procedure Call Processing
   - Lookup procedure name (predefined ada di indices 29-32)
   - Process parameter list (jika ada)
   - Buat ProcCall node dengan predefined marker jika index di antara 29 hingga 32

4. Scope Management

Stack-based scope tracking menggunakan Display array:

- enter_block(): Push new block index ke display, create new btab entry
- exit_block(): Pop dari display
- lookup(): Search dari current level → level 0, lalu check predefined

5. Error Collection & Reporting

Semantic analyzer menggunakan continue-on-error strategy:

- Setiap error ditambahkan ke Vec<SemanticError>
- Analysis tetap berlanjut untuk menemukan multiple errors
- Di akhir, jika ada error: return Err(errors), jika tidak: return Ok(decorated_ast)

6. Output Generation

main.rs mencetak hasil ke terminal dan file:

- Contoh jika berhasil:

```
---SEMANTIC ANALYSIS---
Symbol Table (tab):
idx  name         obj       type      ref   nrm  lev  adr   link
-------------------------------------------------------------------------------
0    dan          type      -         -     1    0    0     -
1    larik        type      -         -     1    0    1     -
2    mulai        type      -         -     1    0    2     -
3    kasus        type      -         -     1    0    3     -
4    konstanta    type      -         -     1    0    4     -
5    bagi         type      -         -     1    0    5     -
6    turun_ke     type      -         -     1    0    6     -
7    lakukan      type      -         -     1    0    7     -
8    selain_itu   type      -         -     1    0    8     -
9    selesai      type      -         -     1    0    9     -
10   untuk        type      -         -     1    0    10    -
11   fungsi       type      -         -     1    0    11    -
12   jika         type      -         -     1    0    12    -
13   mod          type      -         -     1    0    13    -
14   tidak        type      -         -     1    0    14    -
15   dari         type      -         -     1    0    15    -
16   atau         type      -         -     1    0    16    -
17   prosedur     type      -         -     1    0    17    -
18   program      type      -         -     1    0    18    -
19   rekaman      type      -         -     1    0    19    -
20   ulangi       type      -         -     1    0    20    -
21   string       type      4         -     1    0    21    -
22   maka         type      -         -     1    0    22    -
23   ke           type      -         -     1    0    23    -
```

```
24  tipe        type      -       -   1   0   24   -
25  sampai      type      -       -   1   0   25   -
26  variabel    type      -       -   1   0   26   -
27  selama      type      -       -   1   0   27   -
28  padat       type      -       -   1   0   28   -
29  writeln     procedure 0       -   1   0   29   -
30  write       procedure 0       -   1   0   30   -
31  readln      procedure 0       -   1   0   31   -
32  read        procedure 0       -   1   0   32   -
33  Hello       program   0       -   1   0   0    -
34  a           variable  1       -   1   0   0    -
35  b           variable  1       -   1   0   0    34

Block Table (btab):
idx  last   lpar   psze   vsze
------------------------------
0   35    0     0     2
1   0     0     0     0


---DECORATED AST---
Program(name: 'Hello')
  Declarations
    VarDecl('a') → tab_index:34, type:integer, lev:0
    VarDecl('b') → tab_index:35, type:integer, lev:0
  Block
    Block → block_index:1, lev:1
      Assign('a' := 5) → type:integer
        Var(name: 'a', type: integer, tab_index: 34, level: 0)
        Literal(value: 5, type: integer)
      Assign('b' := a+10) → type:integer
        Var(name: 'b', type: integer, tab_index: 35, level: 0)
        BinOp(op: '+', type: integer)
          Left:
            Var(name: 'a', type: integer, tab_index: 34, level: 0)
          Right:
            Literal(value: 10, type: integer)
      writeln(...) → predefined, tab_index:29

--------------
```

- Contoh jika gagal:

```
---SEMANTIC ERRORS---
Semantic error: Undeclared identifier 'a'
```

```
Semantic error: Undeclared identifier 'a'
Semantic error: Invalid operation '+' for types unknown
and integer
Semantic error: Type mismatch: expected integer, found
unknown
------------------
```

7. **Type Checking Flow**
   Datatype compatibility mengikuti hierarchy:
   - Assignment: Integer dapat di-assign ke Real (implicit coercion)
   - Arithmetic: Integer + Integer → Integer, Real + Integer → Real
   - Comparison: Hanya tipe yang sama bisa dibandingkan
   - Boolean: Hanya Boolean untuk kondisi if/while

   Setiap operasi memiliki result type inference:

```
Expression: a + 10
  └─ visit_expression()
     ├─ visit_simple_expression()
     │  ├─ visit_term('a') → type:Integer
     │  ├─ operator: '+'
     │  └─ visit_term(10) → type:Integer
     └─ get_arithmetic_result_type(Integer, Integer) → Integer
```

Justifikasi Pilihan Implementasi

1. *Backward chaining* pada *link* agar memudahkan *tracking last identifier* dan *efficient scope search*
2. *Separate block* untuk *main statement* karena sesuai dengan spesifikasi, *main compound statement execute* di *block* terpisah (block 1)
3. *Predefined procedures* di *reserved words* untuk menghindari duplikasi, selalu tersedia, dan index konsisten
4. Individual VarDecl *nodes* agar *granularity* lebih baik untuk *code generation* dan *debugging*
5. *Decorated* AST dengan *type annotations* sehingga setiap node dapat membawa type information untuk memudahkan fase *code generation*
6. *Continue-on-Error* agar dapat *collect* semua *errors* sekaligus untuk *better dev experience*
7. *Display array* untuk *scope* agar dapat O(1) *access* ke *block information* tanpa *tree traversal*

Implementasi

ast.rs

```rust
use crate::types::DataType;
use std::fmt;

/// AST Node - decorated abstract syntax tree
#[derive(Debug, Clone)]
pub enum AstNode {
    // Program
    Program {
        name: String,
        declarations: Vec<AstNode>,
        body: Box<AstNode>,
        tab_index: Option<usize>,
    },

    // Declarations
    VarDecl {
        names: Vec<String>,
        data_type: DataType,
        tab_indices: Vec<usize>,
        level: usize,
    },

    ConstDecl {
        name: String,
        value: Box<AstNode>,
        data_type: DataType,
        tab_index: usize,
    },

    TypeDecl {
        name: String,
        type_def: DataType,
        tab_index: usize,
    },
```

```
    ProcDecl {
        name: String,
        params: Vec<AstNode>,
        declarations: Vec<AstNode>,
        body: Box<AstNode>,
        tab_index: usize,
        block_index: usize,
    },

    FuncDecl {
        name: String,
        params: Vec<AstNode>,
        return_type: DataType,
        declarations: Vec<AstNode>,
        body: Box<AstNode>,
        tab_index: usize,
        block_index: usize,
    },

    ParamDecl {
        names: Vec<String>,
        data_type: DataType,
        is_var: bool,
        tab_indices: Vec<usize>,
    },

    // Statements
    Block {
        statements: Vec<AstNode>,
        block_index: usize,
        level: usize,
    },

    Assign {
        target: Box<AstNode>,
        value: Box<AstNode>,
        data_type: DataType,
    },
```

```
    If {
        condition: Box<AstNode>,
        then_stmt: Box<AstNode>,
        else_stmt: Option<Box<AstNode>>,
    },

    While {
        condition: Box<AstNode>,
        body: Box<AstNode>,
    },

    For {
        var_name: String,
        start: Box<AstNode>,
        end: Box<AstNode>,
        is_downto: bool,
        body: Box<AstNode>,
        tab_index: usize,
    },

    ProcCall {
        name: String,
        args: Vec<AstNode>,
        tab_index: usize,
    },

    // Expressions
    BinOp {
        op: String,
        left: Box<AstNode>,
        right: Box<AstNode>,
        data_type: DataType,
    },

    UnaryOp {
        op: String,
        operand: Box<AstNode>,
```

```
        data_type: DataType,
    },

    Var {
        name: String,
        data_type: DataType,
        tab_index: usize,
        level: usize,
    },

    Literal {
        value: LiteralValue,
        data_type: DataType,
    },

    // Empty statement
    Empty,
}

#[derive(Debug, Clone)]
pub enum LiteralValue {
    Integer(i64),
    Real(f64),
    Boolean(bool),
    Char(char),
    String(String),
}

impl fmt::Display for LiteralValue {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            LiteralValue::Integer(v) => write!(f, "{}", v),
            LiteralValue::Real(v) => write!(f, "{}", v),
            LiteralValue::Boolean(v) => write!(f, "{}", v),
            LiteralValue::Char(v) => write!(f, "'{}'", v),
            LiteralValue::String(v) => write!(f, "\"{}\"", v),
        }
    }
```

```
}

impl fmt::Display for AstNode {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        self.fmt_recursive(f, 0)
    }
}

impl AstNode {
    fn fmt_recursive(&self, f: &mut fmt::Formatter<'_>, indent: usize)
-> fmt::Result {
        let ind = "  ".repeat(indent);

        match self {
            AstNode::Program { name, declarations, body, tab_index }
=> {
                writeln!(f, "{}Program(name: '{}')", ind, name)?;
                if !declarations.is_empty() {
                    writeln!(f, "{}  Declarations", ind)?;
                    for decl in declarations {
                        decl.fmt_recursive(f, indent + 2)?;
                    }
                }
                writeln!(f, "{}  Block", ind)?;
                body.fmt_recursive(f, indent + 2)?;
            }

            AstNode::VarDecl { names, data_type, tab_indices, level }
=> {
                // Display single variable per line
                if let (Some(name), Some(tab_idx)) = (names.first(),
tab_indices.first()) {
                    writeln!(f, "{}VarDecl('{}') → tab_index:{},
type:{}, lev:{}",
                             ind, name, tab_idx, data_type, level)?;
                }
            }
```

```
            AstNode::ConstDecl { name, value, data_type, tab_index }
=> {
                writeln!(f, "{}ConstDecl(name: '{}', type: {},
tab_index: {})",
                         ind, name, data_type, tab_index)?;
                writeln!(f, "{}  Value:", ind)?;
                value.fmt_recursive(f, indent + 2)?;
            }

            AstNode::TypeDecl { name, type_def, tab_index } => {
                writeln!(f, "{}TypeDecl(name: '{}', type: {},
tab_index: {})",
                         ind, name, type_def, tab_index)?;
            }

            AstNode::ProcDecl { name, params, declarations, body,
tab_index, block_index } => {
                writeln!(f, "{}ProcDecl(name: '{}', tab_index: {},
block_index: {})",
                         ind, name, tab_index, block_index)?;
                if !params.is_empty() {
                    writeln!(f, "{}  Parameters:", ind)?;
                    for param in params {
                        param.fmt_recursive(f, indent + 2)?;
                    }
                }
                if !declarations.is_empty() {
                    writeln!(f, "{}  Declarations:", ind)?;
                    for decl in declarations {
                        decl.fmt_recursive(f, indent + 2)?;
                    }
                }
                writeln!(f, "{}  Body:", ind)?;
                body.fmt_recursive(f, indent + 2)?;
            }

            AstNode::FuncDecl { name, params, return_type,
declarations, body, tab_index, block_index } => {
```

```
                writeln!(f, "{}FuncDecl(name: '{}', return_type: {},
tab_index: {}, block_index: {})",
                         ind, name, return_type, tab_index,
block_index)?;
                if !params.is_empty() {
                    writeln!(f, "{}  Parameters:", ind)?;
                    for param in params {
                        param.fmt_recursive(f, indent + 2)?;
                    }
                }
                if !declarations.is_empty() {
                    writeln!(f, "{}  Declarations:", ind)?;
                    for decl in declarations {
                        decl.fmt_recursive(f, indent + 2)?;
                    }
                }
                writeln!(f, "{}  Body:", ind)?;
                body.fmt_recursive(f, indent + 2)?;
            }

            AstNode::ParamDecl { names, data_type, is_var, tab_indices
} => {
                writeln!(f, "{}ParamDecl(names: {:?}, type: {}, var:
{}, indices: {:?})",
                         ind, names, data_type, is_var, tab_indices)?;
            }

            AstNode::Block { statements, block_index, level } => {
                writeln!(f, "{}Block → block_index:{}, lev:{}", ind,
block_index, level)?;
                for stmt in statements {
                    stmt.fmt_recursive(f, indent + 1)?;
                }
            }

            AstNode::Assign { target, value, data_type } => {
                // Extract target and value for inline display
                let target_str = match target.as_ref() {
```

```
                    AstNode::Var { name, .. } => name.clone(),
                    _ => "?".to_string(),
                };
                let value_str = match value.as_ref() {
                    AstNode::Literal { value:
LiteralValue::Integer(v), .. } => format!("{}", v),
                    AstNode::BinOp { op, left, right, .. } => {
                        let left_str = match left.as_ref() {
                            AstNode::Var { name, .. } => name.clone(),
                            AstNode::Literal { value:
LiteralValue::Integer(v), .. } => format!("{}", v),
                            _ => "?".to_string(),
                        };
                        let right_str = match right.as_ref() {
                            AstNode::Var { name, .. } => name.clone(),
                            AstNode::Literal { value:
LiteralValue::Integer(v), .. } => format!("{}", v),
                            _ => "?".to_string(),
                        };
                        format!("{}{}{}", left_str, op, right_str)
                    },
                    _ => "...".to_string(),
                };
                writeln!(f, "{}Assign('{}' := {}) → type:{}", ind,
target_str, value_str, data_type)?;
                // Show children directly without labels
                target.fmt_recursive(f, indent + 1)?;
                value.fmt_recursive(f, indent + 1)?;
            }

            AstNode::If { condition, then_stmt, else_stmt } => {
                writeln!(f, "{}If", ind)?;
                writeln!(f, "{}  Condition:", ind)?;
                condition.fmt_recursive(f, indent + 2)?;
                writeln!(f, "{}  Then:", ind)?;
                then_stmt.fmt_recursive(f, indent + 2)?;
                if let Some(else_part) = else_stmt {
                    writeln!(f, "{}  Else:", ind)?;
```

```
                    else_part.fmt_recursive(f, indent + 2)?;
                }
            }

            AstNode::While { condition, body } => {
                writeln!(f, "{}While", ind)?;
                writeln!(f, "{}  Condition:", ind)?;
                condition.fmt_recursive(f, indent + 2)?;
                writeln!(f, "{}  Body:", ind)?;
                body.fmt_recursive(f, indent + 2)?;
            }

            AstNode::For { var_name, start, end, is_downto, body,
tab_index } => {
                writeln!(f, "{}For(var: '{}', downto: {}, tab_index:
{})",
                         ind, var_name, is_downto, tab_index)?;
                writeln!(f, "{}  Start:", ind)?;
                start.fmt_recursive(f, indent + 2)?;
                writeln!(f, "{}  End:", ind)?;
                end.fmt_recursive(f, indent + 2)?;
                writeln!(f, "{}  Body:", ind)?;
                body.fmt_recursive(f, indent + 2)?;
            }

            AstNode::ProcCall { name, args, tab_index } => {
                // Predefined procedures are at indices 29-32
                let predefined_marker = if *tab_index >= 29 &&
*tab_index <= 32 {
                    " → predefined"
                } else {
                    ""
                };
                writeln!(f, "{}{}(...){}, tab_index:{}", ind, name,
predefined_marker, tab_index)?;
            }

            AstNode::BinOp { op, left, right, data_type } => {
```

```
                writeln!(f, "{}BinOp(op: '{}', type: {})", ind, op,
data_type)?;
                writeln!(f, "{}  Left:", ind)?;
                left.fmt_recursive(f, indent + 2)?;
                writeln!(f, "{}  Right:", ind)?;
                right.fmt_recursive(f, indent + 2)?;
            }

            AstNode::UnaryOp { op, operand, data_type } => {
                writeln!(f, "{}UnaryOp(op: '{}', type: {})", ind, op,
data_type)?;
                writeln!(f, "{}  Operand:", ind)?;
                operand.fmt_recursive(f, indent + 2)?;
            }

            AstNode::Var { name, data_type, tab_index, level } => {
                writeln!(f, "{}Var(name: '{}', type: {}, tab_index:
{}, level: {})",
                         ind, name, data_type, tab_index, level)?;
            }

            AstNode::Literal { value, data_type } => {
                writeln!(f, "{}Literal(value: {}, type: {})", ind,
value, data_type)?;
            }

            AstNode::Empty => {
                writeln!(f, "{}Empty", ind)?;
            }
        }

        Ok(())
    }
}
```

semantic_analyzer.rs

```rust
use crate::ast::{AstNode, LiteralValue};
use crate::node::{NodeType, ParseNode};
use crate::semantic_error::{SemanticError, SemanticErrorKind};
use crate::symbol_table::{ATabEntry, SymbolTable, TabEntry};
use crate::token::TokenType;
use crate::types::{DataType, ObjectKind};

/// Semantic analyzer that transforms parse tree to decorated AST
pub struct SemanticAnalyzer {
    pub symbol_table: SymbolTable,
    pub errors: Vec<SemanticError>,
    current_proc: Option<String>,
}

impl SemanticAnalyzer {
    pub fn new() -> Self {
        SemanticAnalyzer {
            symbol_table: SymbolTable::new(),
            errors: Vec::new(),
            current_proc: None,
        }
    }

    /// Main entry point for semantic analysis
    pub fn analyze(&mut self, parse_tree: &ParseNode) ->
Result<AstNode, Vec<SemanticError>> {
        let ast = self.visit_program(parse_tree);

        if self.errors.is_empty() {
            Ok(ast)
        } else {
            Err(self.errors.clone())
        }
    }

    /// Visit program node
    fn visit_program(&mut self, node: &ParseNode) -> AstNode {
```

```
        // program -> program-header declaration-part
compound-statement DOT
        if let NodeType::Program = node.node_type {
            let program_name =
self.get_program_name(&node.children[0]);

            // Insert program into symbol table
            let tab_index = self.symbol_table.insert(TabEntry {
                name: program_name.clone(),
                link: None,
                obj: ObjectKind::Program,
                data_type: DataType::Void,
                ref_index: None,
                normal: true,
                level: 0,
                address: 0,
            });

            // Process declarations
            let declarations =
self.visit_declaration_part(&node.children[1]);

            // Enter new block for main compound statement (btab[1])
            let main_block_index = self.symbol_table.enter_block();

            // Process main compound statement
            let body =
self.visit_compound_statement(&node.children[2]);

            // Exit main block
            self.symbol_table.exit_block();

            return AstNode::Program {
                name: program_name,
                declarations,
                body: Box::new(body),
                tab_index: Some(tab_index),
            };
```

```
        }

        AstNode::Empty
    }

    /// Get program name from program header
    fn get_program_name(&self, node: &ParseNode) -> String {
        // program-header -> KEYWORD(program) IDENTIFIER SEMICOLON
        if let NodeType::Terminal(token) = &node.children[1].node_type
{
            return token.value.clone();
        }
        "Unknown".to_string()
    }

    /// Visit declaration part
    fn visit_declaration_part(&mut self, node: &ParseNode) ->
Vec<AstNode> {
        let mut declarations = Vec::new();

        for child in &node.children {
            match &child.node_type {
                NodeType::ConstDeclaration => {

declarations.extend(self.visit_const_declaration(child));
                }
                NodeType::TypeDeclaration => {

declarations.extend(self.visit_type_declaration(child));
                }
                NodeType::VarDeclaration => {

declarations.extend(self.visit_var_declaration(child));
                }
                NodeType::SubprogramDeclaration => {
                    if let Some(decl) =
self.visit_subprogram_declaration(child) {
                        declarations.push(decl);
```

```
                    }
                }
                _ => {}
            }
        }

        declarations
    }

    /// Visit variable declaration
    fn visit_var_declaration(&mut self, node: &ParseNode) ->
Vec<AstNode> {
        let mut declarations = Vec::new();
        let mut i = 1; // Skip "variabel" keyword

        while i < node.children.len() {
            // Get identifier list
            let id_list = self.get_identifier_list(&node.children[i]);
            i += 1; // Skip identifier list

            // Skip colon
            i += 1;

            // Get type
            let data_type = self.get_type(&node.children[i]);
            i += 1;

            // Skip semicolon
            i += 1;

            // Insert variables into symbol table and create separate
VarDecl for each
            let level = self.symbol_table.current_level();

            for name in &id_list {
                // Check for redeclaration
                if let Some(_) =
self.symbol_table.lookup_current_scope(name) {
```

```
                    self.errors.push(SemanticError::redeclared(
                        name.clone(),
                        None,
                    ));
                    continue;
                }

                let tab_index = self.symbol_table.insert(TabEntry {
                    name: name.clone(),
                    link: None,
                    obj: ObjectKind::Variable,
                    data_type: data_type.clone(),
                    ref_index: None,
                    normal: true,
                    level,
                    address: 0,  // TODO: change
                });

                // Create individual VarDecl for each variable
                declarations.push(AstNode::VarDecl {
                    names: vec![name.clone()],
                    data_type: data_type.clone(),
                    tab_indices: vec![tab_index],
                    level,
                });

                // Update variable size
                self.symbol_table.add_var_size(1);
            }
        }

        declarations
    }

    /// Visit constant declaration
    fn visit_const_declaration(&mut self, node: &ParseNode) ->
Vec<AstNode> {
        let mut declarations = Vec::new();
```

```
        let mut i = 1; // Skip "konstanta" keyword

        while i < node.children.len() {
            // Get identifier
            let name = if let NodeType::Terminal(token) =
&node.children[i].node_type {
                token.value.clone()
            } else {
                i += 1;
                continue;
            };
            i += 1;

            // Skip '='
            i += 1;

            // Get value expression
            let value_expr = self.visit_expression(&node.children[i]);
            let data_type = self.get_expr_type(&value_expr);
            i += 1;

            // Skip semicolon
            i += 1;

            // Check for redeclaration
            if let Some(_) =
self.symbol_table.lookup_current_scope(&name) {

self.errors.push(SemanticError::redeclared(name.clone(), None));
                continue;
            }

            let tab_index = self.symbol_table.insert(TabEntry {
                name: name.clone(),
                link: None,
                obj: ObjectKind::Constant,
                data_type: data_type.clone(),
                ref_index: None,
```

```
                normal: true,
                level: self.symbol_table.current_level(),
                address: 0,
            });

            declarations.push(AstNode::ConstDecl {
                name,
                value: Box::new(value_expr),
                data_type,
                tab_index,
            });
        }

        declarations
    }

    /// Visit type declaration
    fn visit_type_declaration(&mut self, node: &ParseNode) ->
Vec<AstNode> {
        let mut declarations = Vec::new();
        let mut i = 1; // Skip "tipe" keyword

        while i < node.children.len() {
            // Get identifier
            let name = if let NodeType::Terminal(token) =
&node.children[i].node_type {
                token.value.clone()
            } else {
                i += 1;
                continue;
            };
            i += 1;

            // Skip '='
            i += 1;

            // Get type definition
            let type_def = self.get_type(&node.children[i]);
```

```
            i += 1;

            // Skip semicolon
            i += 1;

            // Check for redeclaration
            if let Some(_) =
self.symbol_table.lookup_current_scope(&name) {

self.errors.push(SemanticError::redeclared(name.clone(), None));
                continue;
            }

            let tab_index = self.symbol_table.insert(TabEntry {
                name: name.clone(),
                link: None,
                obj: ObjectKind::Type,
                data_type: type_def.clone(),
                ref_index: None,
                normal: true,
                level: self.symbol_table.current_level(),
                address: 0,
            });

            declarations.push(AstNode::TypeDecl {
                name,
                type_def,
                tab_index,
            });
        }

        declarations
    }

    /// Visit subprogram declaration
    fn visit_subprogram_declaration(&mut self, node: &ParseNode) ->
Option<AstNode> {
        if node.children.is_empty() {
```

```
            return None;
        }

        let child = &node.children[0];
        match &child.node_type {
            NodeType::ProcedureDeclaration =>
Some(self.visit_procedure_declaration(child)),
            NodeType::FunctionDeclaration =>
Some(self.visit_function_declaration(child)),
            _ => None,
        }
    }

    /// Visit procedure declaration
    fn visit_procedure_declaration(&mut self, node: &ParseNode) ->
AstNode {
        // prosedur IDENTIFIER (params)? SEMICOLON declarations
compound-statement SEMICOLON
        let mut idx = 1; // Skip "prosedur" keyword

        let name = if let NodeType::Terminal(token) =
&node.children[idx].node_type {
            token.value.clone()
        } else {
            "unknown".to_string()
        };
        idx += 1;

        // Check for redeclaration
        if let Some(_) = self.symbol_table.lookup_current_scope(&name)
{
            self.errors.push(SemanticError::redeclared(name.clone(),
None));
        }

        // Enter new block
        let block_index = self.symbol_table.enter_block();
```

```
        // Check if parameters exist and save the node index
        let param_node_idx = if idx < node.children.len() &&
matches!(node.children[idx].node_type, NodeType::FormalParameterList)
{
            let temp_idx = idx;
            idx += 1;
            Some(temp_idx)
        } else {
            None
        };

        // Skip semicolon
        idx += 1;

        // Insert procedure into symbol table (at parent level)
        self.symbol_table.exit_block();
        let tab_index = self.symbol_table.insert(TabEntry {
            name: name.clone(),
            link: None,
            obj: ObjectKind::Procedure,
            data_type: DataType::Void,
            ref_index: Some(block_index),
            normal: true,
            level: self.symbol_table.current_level(),
            address: 0,
        });

        // Re-enter block for procedure body
        self.symbol_table.enter_block();

        // process parameters
        let params = if let Some(param_idx) = param_node_idx {

self.visit_formal_parameter_list(&node.children[param_idx])
        } else {
            Vec::new()
        };
```

```
        // Process declarations
        let declarations =
self.visit_declaration_part(&node.children[idx]);
        idx += 1;

        // Process body
        let body = self.visit_compound_statement(&node.children[idx]);

        // Exit block
        self.symbol_table.exit_block();

        AstNode::ProcDecl {
            name,
            params,
            declarations,
            body: Box::new(body),
            tab_index,
            block_index,
        }
    }

    /// Visit function declaration
    fn visit_function_declaration(&mut self, node: &ParseNode) ->
AstNode {
        // fungsi IDENTIFIER (params)? COLON type SEMICOLON
declarations compound-statement SEMICOLON
        let mut idx = 1; // Skip "fungsi" keyword

        let name = if let NodeType::Terminal(token) =
&node.children[idx].node_type {
            token.value.clone()
        } else {
            "unknown".to_string()
        };
        idx += 1;

        // Check for redeclaration
        if let Some(_) = self.symbol_table.lookup_current_scope(&name)
```

```
{
            self.errors.push(SemanticError::redeclared(name.clone(),
None));
        }

        // Enter new block for function
        let block_index = self.symbol_table.enter_block();

        // Check if parameters exist and save the node index
        let param_node_idx = if idx < node.children.len() &&
matches!(node.children[idx].node_type, NodeType::FormalParameterList)
{
            let temp_idx = idx;
            idx += 1;
            Some(temp_idx)
        } else {
            None
        };

        // Skip colon
        idx += 1;

        // Get return type
        let return_type = self.get_type(&node.children[idx]);
        idx += 1;

        // Skip semicolon
        idx += 1;

        // Insert function into symbol table (at parent level)
        self.symbol_table.exit_block();
        let tab_index = self.symbol_table.insert(TabEntry {
            name: name.clone(),
            link: None,
            obj: ObjectKind::Function,
            data_type: return_type.clone(),
            ref_index: Some(block_index),
            normal: true,
```

```
            level: self.symbol_table.current_level(),
            address: 0,
        });
        self.symbol_table.enter_block();

        // process parameters
        let params = if let Some(param_idx) = param_node_idx {

self.visit_formal_parameter_list(&node.children[param_idx])
        } else {
            Vec::new()
        };

        self.current_proc = Some(name.clone());

        // Process declarations
        let declarations =
self.visit_declaration_part(&node.children[idx]);
        idx += 1;

        // Process body
        let body = self.visit_compound_statement(&node.children[idx]);

        self.current_proc = None;

        // Exit block
        self.symbol_table.exit_block();

        AstNode::FuncDecl {
            name,
            params,
            return_type,
            declarations,
            body: Box::new(body),
            tab_index,
            block_index,
        }
    }
```

```rust
    /// Visit formal parameter list
    fn visit_formal_parameter_list(&mut self, node: &ParseNode) ->
Vec<AstNode> {
        let mut params = Vec::new();
        let mut i = 1; // Skip '('

        while i < node.children.len() - 1 { // Skip ')'
            // Skip optional 'variabel' keyword
            if let NodeType::Terminal(token) =
&node.children[i].node_type {
                if token.value == "variabel" {
                    i += 1; // Skip the 'variabel' keyword
                }
            }

            // Get identifier list
            let id_list = self.get_identifier_list(&node.children[i]);
            i += 1;

            // Skip colon
            i += 1;

            // Get type
            let data_type = self.get_type(&node.children[i]);
            i += 1;

            // Insert parameters into symbol table
            let mut tab_indices = Vec::new();
            for name in &id_list {
                let tab_index = self.symbol_table.insert(TabEntry {
                    name: name.clone(),
                    link: None,
                    obj: ObjectKind::Parameter,
                    data_type: data_type.clone(),
                    ref_index: None,
                    normal: true,
                    level: self.symbol_table.current_level(),
```

```
                    address: 0,
                });
                tab_indices.push(tab_index);
            }

            params.push(AstNode::ParamDecl {
                names: id_list,
                data_type,
                is_var: false,
                tab_indices,
            });

            // Skip semicolon if present
            if i < node.children.len() - 1
                && matches!(node.children[i].node_type,
NodeType::Terminal(_))
            {
                i += 1;
            }
        }

        params
    }

    /// Visit compound statement
    fn visit_compound_statement(&mut self, node: &ParseNode) ->
AstNode {
        // mulai statement-list selesai
        if node.children.len() >= 2 {
            let statements =
self.visit_statement_list(&node.children[1]);
            let block_index = self.symbol_table.current_block();
            let level = self.symbol_table.current_level();
            return AstNode::Block {
                statements,
                block_index,
                level,
            };
```

```
        }
        AstNode::Empty
    }

    /// Visit statement list
    fn visit_statement_list(&mut self, node: &ParseNode) ->
Vec<AstNode> {
        let mut statements = Vec::new();

        for child in &node.children {
            if let NodeType::Terminal(_) = child.node_type {
                continue; // Skip semicolons
            }

            let stmt = self.visit_statement(child);
            if !matches!(stmt, AstNode::Empty) {
                statements.push(stmt);
            }
        }

        statements
    }

    /// Visit statement
    fn visit_statement(&mut self, node: &ParseNode) -> AstNode {
        match &node.node_type {
            NodeType::AssignmentStatement =>
self.visit_assignment_statement(node),
            NodeType::IfStatement => self.visit_if_statement(node),
            NodeType::WhileStatement =>
self.visit_while_statement(node),
            NodeType::ForStatement => self.visit_for_statement(node),
            NodeType::ProcedureOrFunctionCall =>
self.visit_procedure_call(node),
            NodeType::CompoundStatement =>
self.visit_compound_statement(node),
            _ => AstNode::Empty,
        }
```

```
    }

    /// Visit assignment statement
    fn visit_assignment_statement(&mut self, node: &ParseNode) ->
AstNode {
        // IDENTIFIER := expression
        let var_name = if let NodeType::Terminal(token) =
&node.children[0].node_type {
            token.value.clone()
        } else {
            return AstNode::Empty;
        };

        // Lookup variable
        let tab_index = match self.symbol_table.lookup(&var_name) {
            Some(idx) => idx,
            None => {
                self.errors
                    .push(SemanticError::undeclared(var_name.clone(),
None));
                return AstNode::Empty;
            }
        };

        let var_type =
self.symbol_table.tab[tab_index].data_type.clone();
        let var_level = self.symbol_table.tab[tab_index].level;

        let target = AstNode::Var {
            name: var_name.clone(),
            data_type: var_type.clone(),
            tab_index,
            level: var_level,
        };

        // Visit value expression
        let value = self.visit_expression(&node.children[2]);
        let value_type = self.get_expr_type(&value);
```

```rust
        // Type check
        if !DataType::can_assign(&var_type, &value_type) {
            self.errors.push(SemanticError::type_mismatch(
                format!("{}", var_type),
                format!("{}", value_type),
                None,
            ));
        }

        AstNode::Assign {
            target: Box::new(target),
            value: Box::new(value),
            data_type: var_type,
        }
    }

    /// Visit if statement
    fn visit_if_statement(&mut self, node: &ParseNode) -> AstNode {
        // jika expression maka statement (selain_itu statement)?
        let condition = self.visit_expression(&node.children[1]);
        let cond_type = self.get_expr_type(&condition);

        // Check condition is boolean
        if cond_type != DataType::Boolean {
            self.errors.push(SemanticError::new(
                SemanticErrorKind::ConditionNotBoolean,
                None,
            ));
        }

        let then_stmt = self.visit_statement(&node.children[3]);

        let else_stmt = if node.children.len() > 5 {
            Some(Box::new(self.visit_statement(&node.children[5])))
        } else {
            None
        };
```

```
        AstNode::If {
            condition: Box::new(condition),
            then_stmt: Box::new(then_stmt),
            else_stmt,
        }
    }

    /// Visit while statement
    fn visit_while_statement(&mut self, node: &ParseNode) -> AstNode {
        // selama expression lakukan statement
        let condition = self.visit_expression(&node.children[1]);
        let cond_type = self.get_expr_type(&condition);

        // Check condition is boolean
        if cond_type != DataType::Boolean {
            self.errors.push(SemanticError::new(
                SemanticErrorKind::ConditionNotBoolean,
                None,
            ));
        }

        let body = self.visit_statement(&node.children[3]);

        AstNode::While {
            condition: Box::new(condition),
            body: Box::new(body),
        }
    }

    /// Visit for statement
    fn visit_for_statement(&mut self, node: &ParseNode) -> AstNode {
        // untuk IDENTIFIER := expression (ke|turun_ke) expression
lakukan statement
        let var_name = if let NodeType::Terminal(token) =
&node.children[1].node_type {
            token.value.clone()
        } else {
```

```
            return AstNode::Empty;
        };

        // Lookup variable
        let tab_index = match self.symbol_table.lookup(&var_name) {
            Some(idx) => idx,
            None => {
                self.errors
                    .push(SemanticError::undeclared(var_name.clone(),
None));
                return AstNode::Empty;
            }
        };

        let var_type =
self.symbol_table.tab[tab_index].data_type.clone();

        // Check variable is integer
        if var_type != DataType::Integer {
            self.errors.push(SemanticError::new(
                SemanticErrorKind::InvalidLoopVariable,
                None,
            ));
        }

        let start = self.visit_expression(&node.children[3]);

        let is_downto = if let NodeType::Terminal(token) =
&node.children[4].node_type {
            token.value == "turun_ke"
        } else {
            false
        };

        let end = self.visit_expression(&node.children[5]);
        let body = self.visit_statement(&node.children[7]);

        AstNode::For {
```

```
            var_name,
            start: Box::new(start),
            end: Box::new(end),
            is_downto,
            body: Box::new(body),
            tab_index,
        }
    }

    /// Visit procedure call
    fn visit_procedure_call(&mut self, node: &ParseNode) -> AstNode {
        // IDENTIFIER (parameter-list)?
        let name = if let NodeType::Terminal(token) =
&node.children[0].node_type {
            token.value.clone()
        } else {
            return AstNode::Empty;
        };

        // Lookup procedure/function (predefined procedures in
reserved words)
        let tab_index = match self.symbol_table.lookup(&name) {
            Some(idx) => idx,
            None => {

self.errors.push(SemanticError::undeclared(name.clone(), None));
                return AstNode::Empty;
            }
        };

        // Get arguments if present
        let args = if node.children.len() > 2 {
            // Has parameters
            let param_list_idx = node
                .children
                .iter()
                .position(|c| matches!(c.node_type,
NodeType::ParameterList));
```

```
            if let Some(idx) = param_list_idx {
                self.visit_parameter_list(&node.children[idx])
            } else {
                Vec::new()
            }
        } else {
            Vec::new()
        };

        AstNode::ProcCall {
            name,
            args,
            tab_index,
        }
    }

    /// Visit parameter list
    fn visit_parameter_list(&mut self, node: &ParseNode) ->
Vec<AstNode> {
        let mut args = Vec::new();

        for child in &node.children {
            if let NodeType::Terminal(_) = child.node_type {
                continue; // Skip commas
            }

            if let NodeType::Expression = child.node_type {
                args.push(self.visit_expression(child));
            }
        }

        args
    }

    /// Visit expression
    fn visit_expression(&mut self, node: &ParseNode) -> AstNode {
        // expression -> simple-expression (relational-op
simple-expression)?
```

```
        if node.children.len() == 1 {
            return self.visit_simple_expression(&node.children[0]);
        } else if node.children.len() == 3 {
            let left =
self.visit_simple_expression(&node.children[0]);
            let op = if let NodeType::Terminal(token) =
&node.children[1].node_type {
                token.value.clone()
            } else {
                "=".to_string()
            };
            let right =
self.visit_simple_expression(&node.children[2]);

            let left_type = self.get_expr_type(&left);
            let right_type = self.get_expr_type(&right);

            let result_type = match
DataType::get_relational_result_type(&left_type, &right_type) {
                Ok(t) => t,
                Err(_) => {
                    self.errors.push(SemanticError::invalid_operation(
                        op.clone(),
                        format!("{} and {}", left_type, right_type),
                        None,
                    ));
                    DataType::Unknown
                }
            };

            return AstNode::BinOp {
                op,
                left: Box::new(left),
                right: Box::new(right),
                data_type: result_type,
            };
        }
```

```
        AstNode::Empty
    }

    /// Visit simple expression
    fn visit_simple_expression(&mut self, node: &ParseNode) -> AstNode
{
        // simple-expression -> (sign)? term (additive-op term)*
        let mut i = 0;
        let mut result: Option<AstNode> = None;

        // Check for unary sign
        if let NodeType::Terminal(token) = &node.children[i].node_type
{
            if token.value == "+" || token.value == "-" {
                i += 1;
                let operand = self.visit_term(&node.children[i]);
                i += 1;

                if token.value == "-" {
                    let op_type = self.get_expr_type(&operand);
                    result = Some(AstNode::UnaryOp {
                        op: "-".to_string(),
                        operand: Box::new(operand),
                        data_type: op_type,
                    });
                } else {
                    result = Some(operand);
                }
            }
        }

        // If no unary sign, start with first term
        if result.is_none() {
            result = Some(self.visit_term(&node.children[i]));
            i += 1;
        }

        // Process remaining terms with operators
```

```
        while i < node.children.len() {
            if let NodeType::Terminal(token) =
&node.children[i].node_type {
                let op = token.value.clone();
                i += 1;

                if i < node.children.len() {
                    let right = self.visit_term(&node.children[i]);
                    i += 1;

                    let left = result.unwrap();
                    let left_type = self.get_expr_type(&left);
                    let right_type = self.get_expr_type(&right);

                    let result_type = if op == "atau" {
                        match
DataType::get_logical_result_type(&left_type, &right_type) {
                            Ok(t) => t,
                            Err(_) => {

self.errors.push(SemanticError::invalid_operation(
                                    op.clone(),
                                    format!("{} and {}", left_type,
right_type),
                                    None,
                                ));
                                DataType::Unknown
                            }
                        }
                    } else {
                        match
DataType::get_arithmetic_result_type(&left_type, &right_type) {
                            Ok(t) => t,
                            Err(_) => {

self.errors.push(SemanticError::invalid_operation(
                                    op.clone(),
                                    format!("{} and {}", left_type,
```

```
right_type),
                                    None,
                                ));
                                DataType::Unknown
                            }
                        }
                    };

                    result = Some(AstNode::BinOp {
                        op,
                        left: Box::new(left),
                        right: Box::new(right),
                        data_type: result_type,
                    });
                }
            } else {
                i += 1;
            }
        }

        result.unwrap_or(AstNode::Empty)
    }

    /// Visit term
    fn visit_term(&mut self, node: &ParseNode) -> AstNode {
        // term -> factor (multiplicative-op factor)*
        let mut result = self.visit_factor(&node.children[0]);
        let mut i = 1;

        while i < node.children.len() {
            if let NodeType::Terminal(token) =
&node.children[i].node_type {
                let op = token.value.clone();
                i += 1;

                if i < node.children.len() {
                    let right = self.visit_factor(&node.children[i]);
                    i += 1;
```

```
                    let left = result;
                    let left_type = self.get_expr_type(&left);
                    let right_type = self.get_expr_type(&right);

                    let result_type = if op == "dan" {
                        match
DataType::get_logical_result_type(&left_type, &right_type) {
                            Ok(t) => t,
                            Err(_) => {

self.errors.push(SemanticError::invalid_operation(
                                    op.clone(),
                                    format!("{} and {}", left_type,
right_type),
                                    None,
                                ));
                                DataType::Unknown
                            }
                        }
                    } else {
                        match
DataType::get_arithmetic_result_type(&left_type, &right_type) {
                            Ok(t) => t,
                            Err(_) => {

self.errors.push(SemanticError::invalid_operation(
                                    op.clone(),
                                    format!("{} and {}", left_type,
right_type),
                                    None,
                                ));
                                DataType::Unknown
                            }
                        }
                    };

                    result = AstNode::BinOp {
```

```
                    op,
                    left: Box::new(left),
                    right: Box::new(right),
                    data_type: result_type,
                };
            }
        } else {
            i += 1;
        }
    }

    result
}

/// Visit factor
fn visit_factor(&mut self, node: &ParseNode) -> AstNode {
    if node.children.is_empty() {
        return AstNode::Empty;
    }

    let child = &node.children[0];

    match &child.node_type {
        NodeType::Terminal(token) => match token.token_type {
            TokenType::Number => {
                // Determine if integer or real
                if token.value.contains('.') {
                    if let Ok(val) = token.value.parse::<f64>() {
                        return AstNode::Literal {
                            value: LiteralValue::Real(val),
                            data_type: DataType::Real,
                        };
                    }
                } else {
                    if let Ok(val) = token.value.parse::<i64>() {
                        return AstNode::Literal {
                            value: LiteralValue::Integer(val),
                            data_type: DataType::Integer,
```

```
                            };
                        }
                    }
                    AstNode::Empty
                }
                TokenType::CharLiteral => AstNode::Literal {
                    value:
LiteralValue::Char(token.value.chars().nth(0).unwrap_or(' ')),
                    data_type: DataType::Char,
                },
                TokenType::StringLiteral => AstNode::Literal {
                    value: LiteralValue::String(token.value.clone()),
                    data_type: DataType::String,
                },
                TokenType::Identifier => {
                    let name = token.value.clone();

                    // Lookup identifier
                    match self.symbol_table.lookup(&name) {
                        Some(idx) => {
                            let entry = &self.symbol_table.tab[idx];
                            AstNode::Var {
                                name: name.clone(),
                                data_type: entry.data_type.clone(),
                                tab_index: idx,
                                level: entry.level,
                            }
                        }
                        None => {

self.errors.push(SemanticError::undeclared(name.clone(), None));
                            AstNode::Literal {
                                value: LiteralValue::Integer(0),
                                data_type: DataType::Unknown,
                            }
                        }
                    }
                }
```

```
                TokenType::Keyword => {
                    // Handle true/false
                    if token.value == "true" {
                        return AstNode::Literal {
                            value: LiteralValue::Boolean(true),
                            data_type: DataType::Boolean,
                        };
                    } else if token.value == "false" {
                        return AstNode::Literal {
                            value: LiteralValue::Boolean(false),
                            data_type: DataType::Boolean,
                        };
                    }
                    AstNode::Empty
                }
                TokenType::LogicalOperator if token.value == "tidak"
=> {
                    // Unary not
                    let operand =
self.visit_factor(&node.children[1]);
                    let op_type = self.get_expr_type(&operand);

                    if op_type != DataType::Boolean {

self.errors.push(SemanticError::invalid_operation(
                            "tidak".to_string(),
                            format!("{}", op_type),
                            None,
                        ));
                    }

                    AstNode::UnaryOp {
                        op: "tidak".to_string(),
                        operand: Box::new(operand),
                        data_type: DataType::Boolean,
                    }
                }
                TokenType::LParenthesis => {
```

```
                    // Parenthesized expression
                    self.visit_expression(&node.children[1])
                }
                _ => AstNode::Empty,
            },
            NodeType::ProcedureOrFunctionCall => {
                // Function call
                self.visit_procedure_call(child)
            }
            _ => AstNode::Empty,
        }
    }

    /// Get type from parse tree type node
    fn get_type(&mut self, node: &ParseNode) -> DataType {
        if node.children.is_empty() {
            return DataType::Unknown;
        }

        let child = &node.children[0];

        match &child.node_type {
            NodeType::Terminal(token) => match token.value.as_str() {
                "integer" => DataType::Integer,
                "real" => DataType::Real,
                "boolean" => DataType::Boolean,
                "char" => DataType::Char,
                _ => {
                    // User-defined type or identifier
                    if let Some(idx) =
self.symbol_table.lookup(&token.value) {
                        self.symbol_table.tab[idx].data_type.clone()
                    } else {
                        DataType::UserDefined(token.value.clone())
                    }
                }
            },
            NodeType::ArrayType => {
```

```
                // larik[range] dari type
                let range_node = &child.children[2];
                let (low, high) = self.get_range(range_node);

                let elem_type = self.get_type(&child.children[5]);

                let elem_size = 1; // Simplified
                let total_size = ((high - low + 1) as usize) *
elem_size;

                let atab_index =
self.symbol_table.insert_array(ATabEntry {
                    index_type: DataType::Integer,
                    element_type: elem_type.clone(),
                    element_ref: None,
                    low_bound: low,
                    high_bound: high,
                    element_size: elem_size,
                    total_size,
                });

                DataType::Array(atab_index)
            }
            _ => DataType::Unknown,
        }
    }

    /// Get range bounds
    fn get_range(&mut self, node: &ParseNode) -> (i32, i32) {
        // expression .. expression
        let low_expr = self.visit_expression(&node.children[0]);
        let high_expr = self.visit_expression(&node.children[2]);

        let low = self.get_literal_int(&low_expr).unwrap_or(0);
        let high = self.get_literal_int(&high_expr).unwrap_or(0);

        if low > high {
            self.errors.push(SemanticError::new(
```

```
                SemanticErrorKind::InvalidArrayBounds,
                None,
            ));
        }

        (low, high)
    }

    /// Get integer value from literal node or unary expression
    fn get_literal_int(&self, node: &AstNode) -> Option<i32> {
        match node {
            AstNode::Literal { value, .. } => {
                if let LiteralValue::Integer(v) = value {
                    return Some(*v as i32);
                }
                None
            }
            AstNode::UnaryOp { op, operand, .. } => {
                // Handle unary + and -
                if let Some(val) = self.get_literal_int(operand) {
                    match op.as_str() {
                        "-" => Some(-val),
                        "+" => Some(val),
                        _ => None,
                    }
                } else {
                    None
                }
            }
            AstNode::Var { tab_index, .. } => {
                // Handle constant references
                let entry = &self.symbol_table.tab[*tab_index];
                if entry.obj == ObjectKind::Constant {
                    // TODO: Store constant values in symbol table
                    None
                } else {
                    None
                }
```

```
            }
            _ => None,
        }
    }

    /// Get identifier list from parse tree
    fn get_identifier_list(&self, node: &ParseNode) -> Vec<String> {
        let mut ids = Vec::new();

        for child in &node.children {
            if let NodeType::Terminal(token) = &child.node_type {
                if token.token_type == TokenType::Identifier {
                    ids.push(token.value.clone());
                }
            }
        }

        ids
    }

    /// Get type of an expression AST node
    fn get_expr_type(&self, node: &AstNode) -> DataType {
        match node {
            AstNode::Literal { data_type, .. } => data_type.clone(),
            AstNode::Var { data_type, .. } => data_type.clone(),
            AstNode::BinOp { data_type, .. } => data_type.clone(),
            AstNode::UnaryOp { data_type, .. } => data_type.clone(),
            AstNode::ProcCall { tab_index, .. } => {
                self.symbol_table.tab[*tab_index].data_type.clone()
            }
            _ => DataType::Unknown,
        }
    }
}
```

semantic_error.rs

```
use crate::token::Token;
```

```rust
use std::fmt;

/// Semantic error types
#[derive(Debug, Clone)]
pub enum SemanticErrorKind {
    UndeclaredIdentifier(String),
    RedeclaredIdentifier(String),
    TypeMismatch { expected: String, found: String },
    InvalidOperation { op: String, types: String },
    #[allow(dead_code)]
    WrongParameterCount { expected: usize, found: usize },
    #[allow(dead_code)]
    NotCallable(String),
    #[allow(dead_code)]
    NotAssignable(String),
    InvalidArrayBounds,
    InvalidLoopVariable,
    ConditionNotBoolean,
}

/// Semantic error with location information
#[derive(Debug, Clone)]
pub struct SemanticError {
    #[allow(dead_code)]
    pub kind: SemanticErrorKind,
    pub message: String,
    pub token: Option<Token>,
}

impl SemanticError {
    pub fn new(kind: SemanticErrorKind, token: Option<Token>) -> Self
{
        let message = match &kind {
            SemanticErrorKind::UndeclaredIdentifier(name) => {
                format!("Undeclared identifier '{}'", name)
            }
            SemanticErrorKind::RedeclaredIdentifier(name) => {
                format!("Identifier '{}' is already declared in this
```

```
scope", name)
            }
            SemanticErrorKind::TypeMismatch { expected, found } => {
                format!("Type mismatch: expected {}, found {}",
expected, found)
            }
            SemanticErrorKind::InvalidOperation { op, types } => {
                format!("Invalid operation '{}' for types {}", op,
types)
            }
            SemanticErrorKind::WrongParameterCount { expected, found }
=> {
                format!(
                    "Wrong number of parameters: expected {}, found
{}",
                    expected, found
                )
            }
            SemanticErrorKind::NotCallable(name) => {
                format!("'{}' is not a procedure or function", name)
            }
            SemanticErrorKind::NotAssignable(name) => {
                format!("Cannot assign to '{}'", name)
            }
            SemanticErrorKind::InvalidArrayBounds => {
                "Invalid array bounds: lower bound must be less than
or equal to upper bound"
                    .to_string()
            }
            SemanticErrorKind::InvalidLoopVariable => {
                "Loop variable must be of integer type".to_string()
            }
            SemanticErrorKind::ConditionNotBoolean => {
                "Condition must be of boolean type".to_string()
            }
        };

        SemanticError {
```

```
            kind,
            message,
            token,
        }
    }

    pub fn undeclared(name: String, token: Option<Token>) -> Self {
        Self::new(SemanticErrorKind::UndeclaredIdentifier(name),
token)
    }

    pub fn redeclared(name: String, token: Option<Token>) -> Self {
        Self::new(SemanticErrorKind::RedeclaredIdentifier(name),
token)
    }

    pub fn type_mismatch(expected: String, found: String, token:
Option<Token>) -> Self {
        Self::new(SemanticErrorKind::TypeMismatch { expected, found },
token)
    }

    pub fn invalid_operation(op: String, types: String, token:
Option<Token>) -> Self {
        Self::new(SemanticErrorKind::InvalidOperation { op, types },
token)
    }
}

impl fmt::Display for SemanticError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        if let Some(token) = &self.token {
            write!(f, "Semantic error at {}: {}", token, self.message)
        } else {
            write!(f, "Semantic error: {}", self.message)
        }
    }
}
```

```
impl std::error::Error for SemanticError {}
```

symbol_table.rs

```
use crate::types::{DataType, ObjectKind};
use std::fmt;

// This uses Backward chaining

/// Entry in the identifier table (tab)
#[derive(Debug, Clone)]
pub struct TabEntry {
    pub name: String,
    pub link: Option<usize>,      // Pointer to previous identifier in
same scope
    pub obj: ObjectKind,           // Kind of object
    pub data_type: DataType,       // Type of the identifier
    pub ref_index: Option<usize>,  // Reference to atab/btab for
composite types
    pub normal: bool,              // true = normal variable, false =
var parameter
    pub level: usize,              // Lexical level
    pub address: usize,            // Offset/value depending on obj
type
}

/// Entry in the block table (btab)
#[derive(Debug, Clone)]
pub struct BTabEntry {
    pub last: usize,      // Last identifier in this block
    pub last_par: usize,  // Last parameter
    pub param_size: usize, // Total parameter size
    pub var_size: usize,  // Total local variable size
}

/// Entry in the array table (atab)
#[derive(Debug, Clone)]
```

```rust
pub struct ATabEntry {
    pub index_type: DataType,       // Type of array index
    pub element_type: DataType,     // Type of array elements
    pub element_ref: Option<usize>, // Reference if element is
composite
    pub low_bound: i32,             // Lower bound of array
    pub high_bound: i32,            // Upper bound of array
    pub element_size: usize,        // Size of one element
    pub total_size: usize,          // Total size of array
}

/// Symbol table with three tables: tab, btab, atab
pub struct SymbolTable {
    pub tab: Vec<TabEntry>,
    pub btab: Vec<BTabEntry>,
    pub atab: Vec<ATabEntry>,
    pub display: Vec<usize>, // Display stack for scope management
}

impl SymbolTable {
    /// Create a new symbol table initialized with reserved words and
predefined identifiers
    pub fn new() -> Self {
        let mut tab = Vec::new();

        //
============================================================
        // RESERVED WORDS (indices 0-28) - 29 entries
        //
============================================================

        // 0: AND (dan)
        tab.push(TabEntry {
            name: "dan".to_string(),
            link: None,
            obj: ObjectKind::Type,
            data_type: DataType::Unknown,
            ref_index: None,
```

```
            normal: true,
            level: 0,
            address: 0,
        });

        // 1: ARRAY (larik)
        tab.push(TabEntry {
            name: "larik".to_string(),
            link: None,
            obj: ObjectKind::Type,
            data_type: DataType::Unknown,
            ref_index: None,
            normal: true,
            level: 0,
            address: 1,
        });

        // 2: BEGIN (mulai)
        tab.push(TabEntry {
            name: "mulai".to_string(),
            link: None,
            obj: ObjectKind::Type,
            data_type: DataType::Unknown,
            ref_index: None,
            normal: true,
            level: 0,
            address: 2,
        });

        // 3: CASE (kasus)
        tab.push(TabEntry {
            name: "kasus".to_string(),
            link: None,
            obj: ObjectKind::Type,
            data_type: DataType::Unknown,
            ref_index: None,
            normal: true,
            level: 0,
```

```
        address: 3,
    });

    // 4: CONST (konstanta)
    tab.push(TabEntry {
        name: "konstanta".to_string(),
        link: None,
        obj: ObjectKind::Type,
        data_type: DataType::Unknown,
        ref_index: None,
        normal: true,
        level: 0,
        address: 4,
    });

    // 5: DIV (bagi)
    tab.push(TabEntry {
        name: "bagi".to_string(),
        link: None,
        obj: ObjectKind::Type,
        data_type: DataType::Unknown,
        ref_index: None,
        normal: true,
        level: 0,
        address: 5,
    });

    // 6: DOWNTO (turun_ke)
    tab.push(TabEntry {
        name: "turun_ke".to_string(),
        link: None,
        obj: ObjectKind::Type,
        data_type: DataType::Unknown,
        ref_index: None,
        normal: true,
        level: 0,
        address: 6,
    });
```

```
    // 7: DO (lakukan)
    tab.push(TabEntry {
        name: "lakukan".to_string(),
        link: None,
        obj: ObjectKind::Type,
        data_type: DataType::Unknown,
        ref_index: None,
        normal: true,
        level: 0,
        address: 7,
    });

    // 8: ELSE (selain_itu)
    tab.push(TabEntry {
        name: "selain_itu".to_string(),
        link: None,
        obj: ObjectKind::Type,
        data_type: DataType::Unknown,
        ref_index: None,
        normal: true,
        level: 0,
        address: 8,
    });

    // 9: END (selesai)
    tab.push(TabEntry {
        name: "selesai".to_string(),
        link: None,
        obj: ObjectKind::Type,
        data_type: DataType::Unknown,
        ref_index: None,
        normal: true,
        level: 0,
        address: 9,
    });

    // 10: FOR (untuk)
```

```
        tab.push(TabEntry {
            name: "untuk".to_string(),
            link: None,
            obj: ObjectKind::Type,
            data_type: DataType::Unknown,
            ref_index: None,
            normal: true,
            level: 0,
            address: 10,
        });

        // 11: FUNCTION (fungsi)
        tab.push(TabEntry {
            name: "fungsi".to_string(),
            link: None,
            obj: ObjectKind::Type,
            data_type: DataType::Unknown,
            ref_index: None,
            normal: true,
            level: 0,
            address: 11,
        });

        // 12: IF (jika)
        tab.push(TabEntry {
            name: "jika".to_string(),
            link: None,
            obj: ObjectKind::Type,
            data_type: DataType::Unknown,
            ref_index: None,
            normal: true,
            level: 0,
            address: 12,
        });

        // 13: MOD (mod)
        tab.push(TabEntry {
            name: "mod".to_string(),
```

```
            link: None,
            obj: ObjectKind::Type,
            data_type: DataType::Unknown,
            ref_index: None,
            normal: true,
            level: 0,
            address: 13,
        });

        // 14: NOT (tidak)
        tab.push(TabEntry {
            name: "tidak".to_string(),
            link: None,
            obj: ObjectKind::Type,
            data_type: DataType::Unknown,
            ref_index: None,
            normal: true,
            level: 0,
            address: 14,
        });

        // 15: OF (dari)
        tab.push(TabEntry {
            name: "dari".to_string(),
            link: None,
            obj: ObjectKind::Type,
            data_type: DataType::Unknown,
            ref_index: None,
            normal: true,
            level: 0,
            address: 15,
        });

        // 16: OR (atau)
        tab.push(TabEntry {
            name: "atau".to_string(),
            link: None,
            obj: ObjectKind::Type,
```

```
            data_type: DataType::Unknown,
            ref_index: None,
            normal: true,
            level: 0,
            address: 16,
        });

        // 17: PROCEDURE (prosedur)
        tab.push(TabEntry {
            name: "prosedur".to_string(),
            link: None,
            obj: ObjectKind::Type,
            data_type: DataType::Unknown,
            ref_index: None,
            normal: true,
            level: 0,
            address: 17,
        });

        // 18: PROGRAM (program)
        tab.push(TabEntry {
            name: "program".to_string(),
            link: None,
            obj: ObjectKind::Type,
            data_type: DataType::Unknown,
            ref_index: None,
            normal: true,
            level: 0,
            address: 18,
        });

        // 19: RECORD (rekaman)
        tab.push(TabEntry {
            name: "rekaman".to_string(),
            link: None,
            obj: ObjectKind::Type,
            data_type: DataType::Unknown,
            ref_index: None,
```

```
            normal: true,
            level: 0,
            address: 19,
        });

        // 20: REPEAT (ulangi)
        tab.push(TabEntry {
            name: "ulangi".to_string(),
            link: None,
            obj: ObjectKind::Type,
            data_type: DataType::Unknown,
            ref_index: None,
            normal: true,
            level: 0,
            address: 20,
        });

        // 21: STRING (string) - Note: not in dfa_rules.json keywords
        tab.push(TabEntry {
            name: "string".to_string(),
            link: None,
            obj: ObjectKind::Type,
            data_type: DataType::String,
            ref_index: None,
            normal: true,
            level: 0,
            address: 21,
        });

        // 22: THEN (maka)
        tab.push(TabEntry {
            name: "maka".to_string(),
            link: None,
            obj: ObjectKind::Type,
            data_type: DataType::Unknown,
            ref_index: None,
            normal: true,
            level: 0,
```

```
        address: 22,
    });

    // 23: TO (ke)
    tab.push(TabEntry {
        name: "ke".to_string(),
        link: None,
        obj: ObjectKind::Type,
        data_type: DataType::Unknown,
        ref_index: None,
        normal: true,
        level: 0,
        address: 23,
    });

    // 24: TYPE (tipe)
    tab.push(TabEntry {
        name: "tipe".to_string(),
        link: None,
        obj: ObjectKind::Type,
        data_type: DataType::Unknown,
        ref_index: None,
        normal: true,
        level: 0,
        address: 24,
    });

    // 25: UNTIL (sampai)
    tab.push(TabEntry {
        name: "sampai".to_string(),
        link: None,
        obj: ObjectKind::Type,
        data_type: DataType::Unknown,
        ref_index: None,
        normal: true,
        level: 0,
        address: 25,
    });
```

```
    // 26: VAR (variabel)
    tab.push(TabEntry {
        name: "variabel".to_string(),
        link: None,
        obj: ObjectKind::Type,
        data_type: DataType::Unknown,
        ref_index: None,
        normal: true,
        level: 0,
        address: 26,
    });

    // 27: WHILE (selama)
    tab.push(TabEntry {
        name: "selama".to_string(),
        link: None,
        obj: ObjectKind::Type,
        data_type: DataType::Unknown,
        ref_index: None,
        normal: true,
        level: 0,
        address: 27,
    });

    // 28: PACKED (padat)
    tab.push(TabEntry {
        name: "padat".to_string(),
        link: None,
        obj: ObjectKind::Type,
        data_type: DataType::Unknown,
        ref_index: None,
        normal: true,
        level: 0,
        address: 28,
    });

    //
```

```
============================================================
        // PREDEFINED PROCEDURES (indices 29-32)
        // These are always available and cannot be redeclared
        //
============================================================

        // 29: writeln
        tab.push(TabEntry {
            name: "writeln".to_string(),
            link: None,
            obj: ObjectKind::Procedure,
            data_type: DataType::Void,
            ref_index: None,
            normal: true,
            level: 0,
            address: 29,
        });

        // 30: write
        tab.push(TabEntry {
            name: "write".to_string(),
            link: None,
            obj: ObjectKind::Procedure,
            data_type: DataType::Void,
            ref_index: None,
            normal: true,
            level: 0,
            address: 30,
        });

        // 31: readln
        tab.push(TabEntry {
            name: "readln".to_string(),
            link: None,
            obj: ObjectKind::Procedure,
            data_type: DataType::Void,
            ref_index: None,
            normal: true,
```

```
            level: 0,
            address: 31,
        });

        // 32: read
        tab.push(TabEntry {
            name: "read".to_string(),
            link: None,
            obj: ObjectKind::Procedure,
            data_type: DataType::Void,
            ref_index: None,
            normal: true,
            level: 0,
            address: 32,
        });

        //
============================================================
        // USER IDENTIFIERS START FROM INDEX 33
        //
============================================================

        // Initialize btab with global block (index 0)
        let btab = vec![BTabEntry {
            last: 0,
            last_par: 0,
            param_size: 0,
            var_size: 0,
        }];

        let atab = Vec::new();
        let display = vec![0]; // Display[0] points to global block

        SymbolTable {
            tab,
            btab,
            atab,
            display,
```

```
        }
    }

    /// Enter a new block (for procedures, functions, or main program)
    pub fn enter_block(&mut self) -> usize {
        let block_index = self.btab.len();
        self.btab.push(BTabEntry {
            last: 0,
            last_par: 0,
            param_size: 0,
            var_size: 0,
        });
        self.display.push(block_index);
        block_index
    }

    /// Exit current block
    pub fn exit_block(&mut self) {
        if self.display.len() > 1 {
            self.display.pop();
        }
    }

    /// Get current lexical level
    pub fn current_level(&self) -> usize {
        self.display.len() - 1
    }

    /// Insert a new identifier into the symbol table
    pub fn insert(&mut self, entry: TabEntry) -> usize {
        let index = self.tab.len();
        let level = self.current_level();
        let block_index = self.display[level];

        let mut entry = entry;

        // Find previous identifier of same object type in current
block for linking
```

```
        let mut prev_same_type: Option<usize> = None;
        let mut current = self.btab[block_index].last;

        while current > 0 {
            if self.tab[current].obj == entry.obj {
                prev_same_type = Some(current);
                break;
            }
            current = self.tab[current].link.unwrap_or(0);
        }

        entry.link = prev_same_type;  // Link to previous entry of
same type (or None)

        self.tab.push(entry);

        // Update btab.last to point to the most recently inserted
identifier
        self.btab[block_index].last = index;

        index
    }

    /// Lookup an identifier in current and outer scopes
    pub fn lookup(&self, name: &str) -> Option<usize> {
        // Search from current level down to global level
        for level in (0..=self.current_level()).rev() {
            let block_index = self.display[level];
            let mut current = self.btab[block_index].last;  // Points
to last (most recent) identifier

            // Follow the backward linked list in this block
            while current > 0 {
                if self.tab[current].name == name {
                    return Some(current);
                }
                current = self.tab[current].link.unwrap_or(0);
            }
```

```
        }

        // Check reserved words and predefined procedures (indices
0-32)
        for i in 0..33 {
            if self.tab[i].name == name {
                return Some(i);
            }
        }

        // Check for dynamically inserted identifiers after index 32
        for i in 33..self.tab.len() {
            if self.tab[i].name == name && self.tab[i].level == 0 {
                return Some(i);
            }
        }

        None
    }

    /// Insert new identifier at global level after user declarations
have completed
    pub fn insert_at_global(&mut self, mut entry: TabEntry) -> usize {
        let index = self.tab.len();
        let block_index = 0;  // Always use global block

        let mut prev_same_type: Option<usize> = None;
        let mut current = self.btab[block_index].last;

        while current > 0 {
            if self.tab[current].obj == entry.obj {
                prev_same_type = Some(current);
                break;
            }
            current = self.tab[current].link.unwrap_or(0);
        }

        entry.link = prev_same_type;  // Previous entry of same type
```

```
(or None)
        entry.level = 0;  // Force global level

        self.tab.push(entry);
        index
    }

    /// Check if an identifier is a built-in procedure or constant
    /// recognized by lexer but not stored in symbol table
    pub fn is_builtin(&self, name: &str) -> bool {
        matches!(name, "writeln" | "write" | "readln" | "read" |
"true" | "false")
    }

    /// Lookup identifier only in current scope (for redeclaration
checking)
    pub fn lookup_current_scope(&self, name: &str) -> Option<usize> {
        let level = self.current_level();
        let block_index = self.display[level];
        let mut current = self.btab[block_index].last;  // Points to
most recent identifier

        while current > 0 {
            if self.tab[current].name == name {
                return Some(current);
            }
            current = self.tab[current].link.unwrap_or(0);
        }

        None
    }

    /// Add an array type to atab
    pub fn insert_array(&mut self, entry: ATabEntry) -> usize {
        let index = self.atab.len();
        self.atab.push(entry);
        index
    }
```

```
    /// Get current block index
    pub fn current_block(&self) -> usize {
        self.display[self.current_level()]
    }

    /// Update variable size for current block
    pub fn add_var_size(&mut self, size: usize) {
        let block_index = self.current_block();
        self.btab[block_index].var_size += size;
    }
}

impl fmt::Display for SymbolTable {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        writeln!(f, "Symbol Table (tab):")?;
        writeln!(f, "{:<4} {:<15} {:<12} {:<10} {:<5} {:<4} {:<4}
{:<5} {:<5}",
                 "idx", "name", "obj", "type", "ref", "nrm", "lev",
"adr", "link")?;
        writeln!(f, "{}", "-".repeat(75))?;

        for (i, entry) in self.tab.iter().enumerate() {
            writeln!(
                f,
                "{:<4} {:<15} {:<12} {:<10} {:<5} {:<4} {:<4} {:<5}
{:<5}",
                i,
                entry.name,
                format!("{}", entry.obj),
                entry.data_type.to_numeric(),  // Use numeric
representation
                entry.ref_index.map_or("-".to_string(), |r|
r.to_string()),
                if entry.normal { "1" } else { "0" },
                entry.level,
                entry.address,
                entry.link.map_or("-".to_string(), |l| l.to_string())
```

```
            )?;
        }

        writeln!(f, "\nBlock Table (btab):")?;
        writeln!(f, "{:<4} {:<6} {:<6} {:<6} {:<6}", "idx", "last",
"lpar", "psze", "vsze")?;
        writeln!(f, "{}", "-".repeat(30))?;

        for (i, entry) in self.btab.iter().enumerate() {
            writeln!(
                f,
                "{:<4} {:<6} {:<6} {:<6} {:<6}",
                i, entry.last, entry.last_par, entry.param_size,
entry.var_size
            )?;
        }

        if !self.atab.is_empty() {
            writeln!(f, "\nArray Table (atab):")?;
            writeln!(f, "{:<4} {:<10} {:<10} {:<5} {:<6} {:<6} {:<6}
{:<6}",
                     "idx", "xtyp", "etyp", "eref", "low", "high",
"elsz", "size")?;
            writeln!(f, "{}", "-".repeat(60))?;

            for (i, entry) in self.atab.iter().enumerate() {
                writeln!(
                    f,
                    "{:<4} {:<10} {:<10} {:<5} {:<6} {:<6} {:<6}
{:<6}",
                    i,
                    format!("{}", entry.index_type),
                    format!("{}", entry.element_type),
                    entry.element_ref.map_or("-".to_string(), |r|
r.to_string()),
                    entry.low_bound,
                    entry.high_bound,
                    entry.element_size,
```

```
                    entry.total_size
                )?;
            }
        }

        Ok(())
    }
}
```

types.rs

```
use std::fmt;

/// Represents the data types in Pascal-S
#[derive(Debug, Clone, PartialEq)]
pub enum DataType {
    Integer,
    Real,
    Boolean,
    Char,
    String,
    Array(usize), // Index to atab
    UserDefined(String),
    Void,     // For procedures
    Unknown, // For error recovery
}

impl fmt::Display for DataType {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            DataType::Integer => write!(f, "integer"),
            DataType::Real => write!(f, "real"),
            DataType::Boolean => write!(f, "boolean"),
            DataType::Char => write!(f, "char"),
            DataType::String => write!(f, "string"),
            DataType::Array(idx) => write!(f, "array[{}]", idx),
            DataType::UserDefined(name) => write!(f, "{}", name),
```

```
            DataType::Void => write!(f, "void"),
            DataType::Unknown => write!(f, "unknown"),
        }
    }
}

/// Represents the kind of object an identifier refers to
#[derive(Debug, Clone, PartialEq)]
pub enum ObjectKind {
    Constant,
    Variable,
    Type,
    Procedure,
    Function,
    Parameter,
    Program,
}

impl fmt::Display for ObjectKind {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            ObjectKind::Constant => write!(f, "constant"),
            ObjectKind::Variable => write!(f, "variable"),
            ObjectKind::Type => write!(f, "type"),
            ObjectKind::Procedure => write!(f, "procedure"),
            ObjectKind::Function => write!(f, "function"),
            ObjectKind::Parameter => write!(f, "parameter"),
            ObjectKind::Program => write!(f, "program"),
        }
    }
}

impl DataType {
    /// Check if two types are compatible for operations
    pub fn is_compatible(&self, other: &DataType) -> bool {
        match (self, other) {
            (DataType::Integer, DataType::Integer) => true,
            (DataType::Real, DataType::Real) => true,
```

```
            (DataType::Boolean, DataType::Boolean) => true,
            (DataType::Char, DataType::Char) => true,
            (DataType::String, DataType::String) => true,
            // Integer can be promoted to Real
            (DataType::Integer, DataType::Real) | (DataType::Real,
DataType::Integer) => true,
            _ => false,
        }
    }

    /// Check if a value of type `from` can be assigned to type `to`
    pub fn can_assign(to: &DataType, from: &DataType) -> bool {
        match (to, from) {
            (DataType::Integer, DataType::Integer) => true,
            (DataType::Real, DataType::Real) => true,
            (DataType::Real, DataType::Integer) => true, // Integer
can be assigned to Real
            (DataType::Boolean, DataType::Boolean) => true,
            (DataType::Char, DataType::Char) => true,
            (DataType::String, DataType::String) => true,
            (DataType::UserDefined(a), DataType::UserDefined(b)) => a
== b,
            _ => false,
        }
    }

    /// Get the result type of a binary arithmetic operation
    pub fn get_arithmetic_result_type(left: &DataType, right:
&DataType) -> Result<DataType, String> {
        match (left, right) {
            (DataType::Integer, DataType::Integer) =>
Ok(DataType::Integer),
            (DataType::Real, DataType::Real) => Ok(DataType::Real),
            (DataType::Integer, DataType::Real) | (DataType::Real,
DataType::Integer) => {
                Ok(DataType::Real)
            }
            _ => Err(format!(
```

```
                "Arithmetic operation not supported between {} and
{}",
                left, right
            )),
        }
    }

    /// Get the result type of a relational operation (always boolean)
    pub fn get_relational_result_type(left: &DataType, right:
&DataType) -> Result<DataType, String> {
        if left.is_compatible(right) {
            Ok(DataType::Boolean)
        } else {
            Err(format!(
                "Cannot compare incompatible types {} and {}",
                left, right
            ))
        }
    }

    /// Get the result type of a logical operation (must be boolean)
    pub fn get_logical_result_type(left: &DataType, right: &DataType)
-> Result<DataType, String> {
        match (left, right) {
            (DataType::Boolean, DataType::Boolean) =>
Ok(DataType::Boolean),
            _ => Err(format!(
                "Logical operation requires boolean operands, got {}
and {}",
                left, right
            )),
        }
    }

    /// Check if this is a numeric type
    #[allow(dead_code)]
    pub fn is_numeric(&self) -> bool {
        matches!(self, DataType::Integer | DataType::Real)
```

```rust
    }

    /// Check if this is an ordinal type (can be used in for loops,
array indices)
    #[allow(dead_code)]
    pub fn is_ordinal(&self) -> bool {
        matches!(self, DataType::Integer | DataType::Char |
DataType::Boolean)
    }

    /// Convert DataType to numeric code (for Pascal-S compatibility)
    /// Following standard Pascal-S type codes:
    /// 0 = Void, 1 = Integer, 2 = Real, 3 = Boolean, 4 = String, 5 =
Char
    /// 6+ = Array/Record (ref to atab/btab)
    pub fn to_numeric(&self) -> String {
        match self {
            DataType::Void => "0".to_string(),
            DataType::Integer => "1".to_string(),
            DataType::Real => "2".to_string(),
            DataType::Boolean => "3".to_string(),
            DataType::String => "4".to_string(),
            DataType::Char => "5".to_string(),
            DataType::Array(idx) => format!("{}", idx),
            DataType::UserDefined(_) => "6".to_string(),
            DataType::Unknown => "-".to_string(),
        }
    }
}
```

Pengujian

input-1.pas

input

```
program HelloWorld;

mulai
    writeln('Hello World!');
selesai.
```

output

```
---SEMANTIC ANALYSIS---
Symbol Table (tab):
idx  name            obj          type       ref
nrm  lev  adr   link
-------------------------------------------------------
----------------------
0    dan             type         -          -     1
0    0     -
1    larik           type         -          -     1
0    1     -
2    mulai           type         -          -     1
0    2     -
3    kasus           type         -          -     1
0    3     -
4    konstanta       type         -          -     1
0    4     -
5    bagi            type         -          -     1
0    5     -
6    turun_ke        type         -          -     1
0    6     -
7    lakukan         type         -          -     1
0    7     -
8    selain_itu      type         -          -     1
0    8     -
9    selesai         type         -          -     1
0    9     -
10   untuk           type         -          -     1
0    10    -
11   fungsi          type         -          -     1
0    11    -
12   jika            type         -          -     1
0    12    -
13   mod             type         -          -     1
0    13    -
14   tidak           type         -          -     1
0    14    -
```

```
15   dari            type         -          -     1
0    15    -
16   atau            type         -          -     1
0    16    -
17   prosedur        type         -          -     1
0    17    -
18   program         type         -          -     1
0    18    -
19   rekaman         type         -          -     1
0    19    -
20   ulangi          type         -          -     1
0    20    -
21   string          type         4          -     1
0    21    -
22   maka            type         -          -     1
0    22    -
23   ke              type         -          -     1
0    23    -
24   tipe            type         -          -     1
0    24    -
25   sampai          type         -          -     1
0    25    -
26   variabel        type         -          -     1
0    26    -
27   selama          type         -          -     1
0    27    -
28   padat           type         -          -     1
0    28    -
29   writeln         procedure    0          -     1
0    29    -
30   write           procedure    0          -     1
0    30    -
31   readln          procedure    0          -     1
0    31    -
32   read            procedure    0          -     1
0    32    -
33   HelloWorld      program      0          -     1
0    0     -

Block Table (btab):
idx  last   lpar   psze   vsze
------------------------------
0    33     0      0      0
1    0      0      0      0


---DECORATED AST---
Program(name: 'HelloWorld')
  Block
    Block → block_index:1, lev:1
      writeln(...) → predefined, tab_index:29
```

```
--------------
```

input-2.pas

input

```
program JumlahAja;

variabel
    a, b, hasil: integer;

mulai
    a := 58;
    b := 9;
    hasil := a + b;
    writeln('hasil penjumlahan tersebut adalah ',
hasil);
selesai.
```

output

```
---SEMANTIC ANALYSIS---
Symbol Table (tab):
idx  name            obj          type       ref
nrm  lev  adr   link
-------------------------------------------------------
----------------------
0    dan             type         -          -     1
0    0     -
1    larik           type         -          -     1
0    1     -
2    mulai           type         -          -     1
0    2     -
3    kasus           type         -          -     1
0    3     -
4    konstanta       type         -          -     1
0    4     -
5    bagi            type         -          -     1
0    5     -
6    turun_ke        type         -          -     1
0    6     -
7    lakukan         type         -          -     1
0    7     -
8    selain_itu      type         -          -     1
0    8     -
9    selesai         type         -          -     1
0    9     -
10   untuk           type         -          -     1
```

```
0    10    -
11   fungsi          type         -          -     1
0    11    -
12   jika            type         -          -     1
0    12    -
13   mod             type         -          -     1
0    13    -
14   tidak           type         -          -     1
0    14    -
15   dari            type         -          -     1
0    15    -
16   atau            type         -          -     1
0    16    -
17   prosedur        type         -          -     1
0    17    -
18   program         type         -          -     1
0    18    -
19   rekaman         type         -          -     1
0    19    -
20   ulangi          type         -          -     1
0    20    -
21   string          type         4          -     1
0    21    -
22   maka            type         -          -     1
0    22    -
23   ke              type         -          -     1
0    23    -
24   tipe            type         -          -     1
0    24    -
25   sampai          type         -          -     1
0    25    -
26   variabel        type         -          -     1
0    26    -
27   selama          type         -          -     1
0    27    -
28   padat           type         -          -     1
0    28    -
29   writeln         procedure    0          -     1
0    29    -
30   write           procedure    0          -     1
0    30    -
31   readln          procedure    0          -     1
0    31    -
32   read            procedure    0          -     1
0    32    -
33   JumlahAja       program      0          -     1
0    0     -
34   a               variable     1          -     1
0    0     -
35   b               variable     1          -     1
0    0     34
36   hasil           variable     1          -     1
```

```
0    0     35

Block Table (btab):
idx  last   lpar   psze   vsze
------------------------------
0    36     0      0      3
1    0      0      0      0


---DECORATED AST---
Program(name: 'JumlahAja')
  Declarations
    VarDecl('a') → tab_index:34, type:integer, lev:0
    VarDecl('b') → tab_index:35, type:integer, lev:0
    VarDecl('hasil') → tab_index:36, type:integer,
lev:0
  Block
    Block → block_index:1, lev:1
      Assign('a' := 58) → type:integer
        Var(name: 'a', type: integer, tab_index: 34,
level: 0)
        Literal(value: 58, type: integer)
      Assign('b' := 9) → type:integer
        Var(name: 'b', type: integer, tab_index: 35,
level: 0)
        Literal(value: 9, type: integer)
      Assign('hasil' := a+b) → type:integer
        Var(name: 'hasil', type: integer, tab_index:
36, level: 0)
        BinOp(op: '+', type: integer)
          Left:
            Var(name: 'a', type: integer, tab_index:
34, level: 0)
          Right:
            Var(name: 'b', type: integer, tab_index:
35, level: 0)
      writeln(...) → predefined, tab_index:29

--------------
```

input-3.pas

input

```
program CobaChar;

variabel
    a, b, c, d: char;

mulai
```

```
    a := 'a';
    b := 'b';
    c := 'c';
    d := 'd';

    writeln(a, b, a, c, a, d);
selesai.
```

output

```
---SEMANTIC ANALYSIS---
Symbol Table (tab):
idx  name            obj          type       ref
nrm  lev  adr   link
-------------------------------------------------------
----------------------
0    dan             type         -          -     1
0    0     -
1    larik           type         -          -     1
0    1     -
2    mulai           type         -          -     1
0    2     -
3    kasus           type         -          -     1
0    3     -
4    konstanta       type         -          -     1
0    4     -
5    bagi            type         -          -     1
0    5     -
6    turun_ke        type         -          -     1
0    6     -
7    lakukan         type         -          -     1
0    7     -
8    selain_itu      type         -          -     1
0    8     -
9    selesai         type         -          -     1
0    9     -
10   untuk           type         -          -     1
0    10    -
11   fungsi          type         -          -     1
0    11    -
12   jika            type         -          -     1
0    12    -
13   mod             type         -          -     1
0    13    -
14   tidak           type         -          -     1
0    14    -
15   dari            type         -          -     1
0    15    -
16   atau            type         -          -     1
0    16    -
```

```
17   prosedur        type         -          -     1
0    17    -
18   program         type         -          -     1
0    18    -
19   rekaman         type         -          -     1
0    19    -
20   ulangi          type         -          -     1
0    20    -
21   string          type         4          -     1
0    21    -
22   maka            type         -          -     1
0    22    -
23   ke              type         -          -     1
0    23    -
24   tipe            type         -          -     1
0    24    -
25   sampai          type         -          -     1
0    25    -
26   variabel        type         -          -     1
0    26    -
27   selama          type         -          -     1
0    27    -
28   padat           type         -          -     1
0    28    -
29   writeln         procedure    0          -     1
0    29    -
30   write           procedure    0          -     1
0    30    -
31   readln          procedure    0          -     1
0    31    -
32   read            procedure    0          -     1
0    32    -
33   CobaChar        program      0          -     1
0    0     -
34   a               variable     5          -     1
0    0     -
35   b               variable     5          -     1
0    0     34
36   c               variable     5          -     1
0    0     35
37   d               variable     5          -     1
0    0     36

Block Table (btab):
idx  last   lpar   psze   vsze
------------------------------
0    37     0      0      4
1    0      0      0      0


---DECORATED AST---
Program(name: 'CobaChar')
```

```
  Declarations
    VarDecl('a') → tab_index:34, type:char, lev:0
    VarDecl('b') → tab_index:35, type:char, lev:0
    VarDecl('c') → tab_index:36, type:char, lev:0
    VarDecl('d') → tab_index:37, type:char, lev:0
  Block
    Block → block_index:1, lev:1
      Assign('a' := ...) → type:char
        Var(name: 'a', type: char, tab_index: 34,
level: 0)
        Literal(value: ''', type: char)
      Assign('b' := ...) → type:char
        Var(name: 'b', type: char, tab_index: 35,
level: 0)
        Literal(value: ''', type: char)
      Assign('c' := ...) → type:char
        Var(name: 'c', type: char, tab_index: 36,
level: 0)
        Literal(value: ''', type: char)
      Assign('d' := ...) → type:char
        Var(name: 'd', type: char, tab_index: 37,
level: 0)
        Literal(value: ''', type: char)
      writeln(...) → predefined, tab_index:29

--------------
```

input-4.pas

input

```
program UTS;

variabel
  pekan: integer;

mulai
  pekan := 8;

  jika pekan = 8 maka
    writeln('Semangat UTS')
  selain_itu
    writeln('Nugas moal?');
selesai.
```

output

```
---SEMANTIC ANALYSIS---
Symbol Table (tab):
idx  name            obj          type       ref
nrm  lev  adr   link
-------------------------------------------------------
----------------------
0    dan             type         -          -     1
0    0     -
1    larik           type         -          -     1
0    1     -
2    mulai           type         -          -     1
0    2     -
3    kasus           type         -          -     1
0    3     -
4    konstanta       type         -          -     1
0    4     -
5    bagi            type         -          -     1
0    5     -
6    turun_ke        type         -          -     1
0    6     -
7    lakukan         type         -          -     1
0    7     -
8    selain_itu      type         -          -     1
0    8     -
9    selesai         type         -          -     1
0    9     -
10   untuk           type         -          -     1
0    10    -
11   fungsi          type         -          -     1
0    11    -
12   jika            type         -          -     1
0    12    -
13   mod             type         -          -     1
0    13    -
14   tidak           type         -          -     1
0    14    -
15   dari            type         -          -     1
0    15    -
16   atau            type         -          -     1
0    16    -
17   prosedur        type         -          -     1
0    17    -
18   program         type         -          -     1
0    18    -
19   rekaman         type         -          -     1
0    19    -
20   ulangi          type         -          -     1
0    20    -
21   string          type         4          -     1
0    21    -
22   maka            type         -          -     1
0    22    -
```

```
23   ke              type         -          -     1
0    23    -
24   tipe            type         -          -     1
0    24    -
25   sampai          type         -          -     1
0    25    -
26   variabel        type         -          -     1
0    26    -
27   selama          type         -          -     1
0    27    -
28   padat           type         -          -     1
0    28    -
29   writeln         procedure    0          -     1
0    29    -
30   write           procedure    0          -     1
0    30    -
31   readln          procedure    0          -     1
0    31    -
32   read            procedure    0          -     1
0    32    -
33   UTS             program      0          -     1
0    0     -
34   pekan           variable     1          -     1
0    0     -

Block Table (btab):
idx  last   lpar   psze   vsze
------------------------------
0    34     0      0      1
1    0      0      0      0


---DECORATED AST---
Program(name: 'UTS')
  Declarations
    VarDecl('pekan') → tab_index:34, type:integer,
lev:0
  Block
    Block → block_index:1, lev:1
      Assign('pekan' := 8) → type:integer
        Var(name: 'pekan', type: integer, tab_index:
34, level: 0)
        Literal(value: 8, type: integer)
      If
        Condition:
          BinOp(op: '=', type: boolean)
            Left:
              Var(name: 'pekan', type: integer,
tab_index: 34, level: 0)
            Right:
              Literal(value: 8, type: integer)
        Then:
```

```
          writeln(...) → predefined, tab_index:29
        Else:
          writeln(...) → predefined, tab_index:29

--------------
```

input-5.pas

input

```
program HitungMundur;

variabel
   i: integer;

mulai
   untuk i := 3 turun-ke 1 lakukan
      writeln(i);
selesai.
```

output

```
---SEMANTIC ANALYSIS---
Symbol Table (tab):
idx  name            obj          type       ref
nrm  lev  adr   link
-----------------------------------------------------
----------------------
0    dan             type         -          -     1
0    0     -
1    larik           type         -          -     1
0    1     -
2    mulai           type         -          -     1
0    2     -
3    kasus           type         -          -     1
0    3     -
4    konstanta       type         -          -     1
0    4     -
5    bagi            type         -          -     1
0    5     -
6    turun_ke        type         -          -     1
0    6     -
7    lakukan         type         -          -     1
0    7     -
8    selain_itu      type         -          -     1
0    8     -
9    selesai         type         -          -     1
0    9     -
10   untuk           type         -          -     1
```

```
0    10    -
11   fungsi          type         -          -     1
0    11    -
12   jika            type         -          -     1
0    12    -
13   mod             type         -          -     1
0    13    -
14   tidak           type         -          -     1
0    14    -
15   dari            type         -          -     1
0    15    -
16   atau            type         -          -     1
0    16    -
17   prosedur        type         -          -     1
0    17    -
18   program         type         -          -     1
0    18    -
19   rekaman         type         -          -     1
0    19    -
20   ulangi          type         -          -     1
0    20    -
21   string          type         4          -     1
0    21    -
22   maka            type         -          -     1
0    22    -
23   ke              type         -          -     1
0    23    -
24   tipe            type         -          -     1
0    24    -
25   sampai          type         -          -     1
0    25    -
26   variabel        type         -          -     1
0    26    -
27   selama          type         -          -     1
0    27    -
28   padat           type         -          -     1
0    28    -
29   writeln         procedure    0          -     1
0    29    -
30   write           procedure    0          -     1
0    30    -
31   readln          procedure    0          -     1
0    31    -
32   read            procedure    0          -     1
0    32    -
33   HitungMundur    program      0          -     1
0    0     -
34   i               variable     1          -     1
0    0     -

Block Table (btab):
idx  last   lpar   psze   vsze
```

```
------------------------------
0     34      0       0       1
1     0       0       0       0


---DECORATED AST---
Program(name: 'HitungMundur')
  Declarations
    VarDecl('i') → tab_index:34, type:integer, lev:0
  Block
    Block → block_index:1, lev:1
      For(var: 'i', downto: true, tab_index: 34)
        Start:
          Literal(value: 3, type: integer)
        End:
          Literal(value: 1, type: integer)
        Body:
          writeln(...) → predefined, tab_index:29

--------------
```

input-6.pas

input

```
program Hello;

variabel
  b: integer;

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

output

```
---TOKENS---
KEYWORD(program)
IDENTIFIER(Hello)
SEMICOLON(;)
KEYWORD(variabel)
IDENTIFIER(b)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(mulai)
IDENTIFIER(a)
ASSIGN_OPERATOR(:=)
```

```
--------------

---SEMANTIC ERRORS---
Semantic error: Undeclared identifier 'a'
Semantic error: Undeclared identifier 'a'
Semantic error: Invalid operation '+' for types
unknown and integer
Semantic error: Type mismatch: expected integer,
found unknown
------------------
```

input-7.pas

input

```
program TestAllTokens;

{ This is a block comment
  Good luck UTS. }

variabel
  my_integer, another_var : integer;
  a_real_number          : real;
  is_done                : boolean;
  my_char                : char;

konstanta
  PI = 3.14159;

(* Range operator for arrays or subranges *)
{ array declaration is just for tokenizing '..' }
tipe
  Numbers = larik[1..10] dari integer;

mulai
  (* Tes assignments and expressions *)
  my_integer := 100;
  another_var := my_integer + 20;
  a_real_number := my_integer / 3.0;

  (* Tes Relational and logical operators *)
  jika (my_integer > 50) dan (another_var <> 104)
maka
  mulai
    is_done := true;
  selesai
  selain_itu
  mulai
    is_done := false;
  selesai;
```

```
  (* Character and String Literals *)
  my_char := 'A';
  writeln('This is a test string literal.');

  (* Testing multi-character operators *)
  jika another_var <= 105 maka
    writeln('Less than or equal');

selesai. (* End of the test program. *)
```

output

```
---SEMANTIC ANALYSIS---
Symbol Table (tab):
idx  name            obj          type       ref
nrm  lev  adr   link
-------------------------------------------------------
----------------------
0    dan             type         -          -     1
0    0     -
1    larik           type         -          -     1
0    1     -
2    mulai           type         -          -     1
0    2     -
3    kasus           type         -          -     1
0    3     -
4    konstanta       type         -          -     1
0    4     -
5    bagi            type         -          -     1
0    5     -
6    turun_ke        type         -          -     1
0    6     -
7    lakukan         type         -          -     1
0    7     -
8    selain_itu      type         -          -     1
0    8     -
9    selesai         type         -          -     1
0    9     -
10   untuk           type         -          -     1
0    10    -
11   fungsi          type         -          -     1
0    11    -
12   jika            type         -          -     1
0    12    -
13   mod             type         -          -     1
0    13    -
14   tidak           type         -          -     1
0    14    -
15   dari            type         -          -     1
0    15    -
16   atau            type         -          -     1
0    16    -
```

```
17   prosedur        type         -          -     1
0    17    -
18   program         type         -          -     1
0    18    -
19   rekaman         type         -          -     1
0    19    -
20   ulangi          type         -          -     1
0    20    -
21   string          type         4          -     1
0    21    -
22   maka            type         -          -     1
0    22    -
23   ke              type         -          -     1
0    23    -
24   tipe            type         -          -     1
0    24    -
25   sampai          type         -          -     1
0    25    -
26   variabel        type         -          -     1
0    26    -
27   selama          type         -          -     1
0    27    -
28   padat           type         -          -     1
0    28    -
29   writeln         procedure    0          -     1
0    29    -
30   write           procedure    0          -     1
0    30    -
31   readln          procedure    0          -     1
0    31    -
32   read            procedure    0          -     1
0    32    -
33   TestAllTokens   program      0          -     1
0    0     -
34   my_integer      variable     1          -     1
0    0     -
35   another_var     variable     1          -     1
0    0     34
36   a_real_number   variable     2          -     1
0    0     35
37   is_done         variable     3          -     1
0    0     36
38   my_char         variable     5          -     1
0    0     37
39   PI              constant     2          -     1
0    0     -
40   Numbers         type         0          -     1
0    0     -

Block Table (btab):
idx  last   lpar   psze   vsze
------------------------------
```

```
0    40     0      0      5
1    0      0      0      0

Array Table (atab):
idx  xtyp       etyp       eref  low    high   elsz
size
-------------------------------------------------------
-------
0    integer    integer    -     1      10     1
10


---DECORATED AST---
Program(name: 'TestAllTokens')
  Declarations
    VarDecl('my_integer') → tab_index:34,
type:integer, lev:0
    VarDecl('another_var') → tab_index:35,
type:integer, lev:0
    VarDecl('a_real_number') → tab_index:36,
type:real, lev:0
    VarDecl('is_done') → tab_index:37, type:boolean,
lev:0
    VarDecl('my_char') → tab_index:38, type:char,
lev:0
    ConstDecl(name: 'PI', type: real, tab_index: 39)
      Value:
        Literal(value: 3.14159, type: real)
    TypeDecl(name: 'Numbers', type: array[0],
tab_index: 40)
  Block
    Block → block_index:1, lev:1
      Assign('my_integer' := 100) → type:integer
        Var(name: 'my_integer', type: integer,
tab_index: 34, level: 0)
        Literal(value: 100, type: integer)
      Assign('another_var' := my_integer+20) →
type:integer
        Var(name: 'another_var', type: integer,
tab_index: 35, level: 0)
        BinOp(op: '+', type: integer)
          Left:
            Var(name: 'my_integer', type: integer,
tab_index: 34, level: 0)
          Right:
            Literal(value: 20, type: integer)
      Assign('a_real_number' := my_integer/?) →
type:real
        Var(name: 'a_real_number', type: real,
tab_index: 36, level: 0)
        BinOp(op: '/', type: real)
          Left:
```

```
            Var(name: 'my_integer', type: integer,
tab_index: 34, level: 0)
          Right:
            Literal(value: 3, type: real)
      If
        Condition:
          BinOp(op: 'dan', type: boolean)
            Left:
              BinOp(op: '>', type: boolean)
                Left:
                  Var(name: 'my_integer', type:
integer, tab_index: 34, level: 0)
                Right:
                  Literal(value: 50, type: integer)
            Right:
              BinOp(op: '<>', type: boolean)
                Left:
                  Var(name: 'another_var', type:
integer, tab_index: 35, level: 0)
                Right:
                  Literal(value: 104, type: integer)
        Then:
          Block → block_index:1, lev:1
            Assign('is_done' := ...) → type:boolean
              Var(name: 'is_done', type: boolean,
tab_index: 37, level: 0)
              Literal(value: true, type: boolean)
        Else:
          Block → block_index:1, lev:1
            Assign('is_done' := ...) → type:boolean
              Var(name: 'is_done', type: boolean,
tab_index: 37, level: 0)
              Literal(value: false, type: boolean)
      Assign('my_char' := ...) → type:char
        Var(name: 'my_char', type: char, tab_index:
38, level: 0)
        Literal(value: ''', type: char)
      writeln(...) → predefined, tab_index:29
      If
        Condition:
          BinOp(op: '<=', type: boolean)
            Left:
              Var(name: 'another_var', type: integer,
tab_index: 35, level: 0)
            Right:
              Literal(value: 105, type: integer)
        Then:
          writeln(...) → predefined, tab_index:29

--------------
```

input-8.pas

input

```
program Hello;

variabel
  a, b: integer;

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

output

```
---SEMANTIC ANALYSIS---
Symbol Table (tab):
idx  name            obj          type       ref
nrm  lev  adr   link
-------------------------------------------------------
----------------------
0    dan             type         -          -     1
0    0     -
1    larik           type         -          -     1
0    1     -
2    mulai           type         -          -     1
0    2     -
3    kasus           type         -          -     1
0    3     -
4    konstanta       type         -          -     1
0    4     -
5    bagi            type         -          -     1
0    5     -
6    turun_ke        type         -          -     1
0    6     -
7    lakukan         type         -          -     1
0    7     -
8    selain_itu      type         -          -     1
0    8     -
9    selesai         type         -          -     1
0    9     -
10   untuk           type         -          -     1
0    10    -
11   fungsi          type         -          -     1
0    11    -
12   jika            type         -          -     1
0    12    -
13   mod             type         -          -     1
0    13    -
```

```
14   tidak           type         -          -     1
0    14    -
15   dari            type         -          -     1
0    15    -
16   atau            type         -          -     1
0    16    -
17   prosedur        type         -          -     1
0    17    -
18   program         type         -          -     1
0    18    -
19   rekaman         type         -          -     1
0    19    -
20   ulangi          type         -          -     1
0    20    -
21   string          type         4          -     1
0    21    -
22   maka            type         -          -     1
0    22    -
23   ke              type         -          -     1
0    23    -
24   tipe            type         -          -     1
0    24    -
25   sampai          type         -          -     1
0    25    -
26   variabel        type         -          -     1
0    26    -
27   selama          type         -          -     1
0    27    -
28   padat           type         -          -     1
0    28    -
29   writeln         procedure    0          -     1
0    29    -
30   write           procedure    0          -     1
0    30    -
31   readln          procedure    0          -     1
0    31    -
32   read            procedure    0          -     1
0    32    -
33   Hello           program      0          -     1
0    0     -
34   a               variable     1          -     1
0    0     -
35   b               variable     1          -     1
0    0     34

Block Table (btab):
idx  last   lpar   psze   vsze
------------------------------
0    35     0      0      2
1    0      0      0      0
```

```
---DECORATED AST---
Program(name: 'Hello')
  Declarations
    VarDecl('a') → tab_index:34, type:integer, lev:0
    VarDecl('b') → tab_index:35, type:integer, lev:0
  Block
    Block → block_index:1, lev:1
      Assign('a' := 5) → type:integer
        Var(name: 'a', type: integer, tab_index: 34,
level: 0)
        Literal(value: 5, type: integer)
      Assign('b' := a+10) → type:integer
        Var(name: 'b', type: integer, tab_index: 35,
level: 0)
        BinOp(op: '+', type: integer)
          Left:
            Var(name: 'a', type: integer, tab_index:
34, level: 0)
          Right:
            Literal(value: 10, type: integer)
      writeln(...) → predefined, tab_index:29

--------------
```

input-9.pas

input

```
program NestedTest;
variabel x: integer;

prosedur Outer;

  prosedur Inner;
    mulai
      x := 10;
      writeln(x)
    selesai;

  mulai
    Inner;
    writeln('Done')
  selesai;

mulai
  x := 0;
  Outer
selesai.
selesai.
```

output

```
---SEMANTIC ANALYSIS---
Symbol Table (tab):
idx  name            obj          type       ref
nrm  lev  adr   link
-------------------------------------------------------
----------------------
0    dan             type         -          -     1
0    0     -
1    larik           type         -          -     1
0    1     -
2    mulai           type         -          -     1
0    2     -
3    kasus           type         -          -     1
0    3     -
4    konstanta       type         -          -     1
0    4     -
5    bagi            type         -          -     1
0    5     -
6    turun_ke        type         -          -     1
0    6     -
7    lakukan         type         -          -     1
0    7     -
8    selain_itu      type         -          -     1
0    8     -
9    selesai         type         -          -     1
0    9     -
10   untuk           type         -          -     1
0    10    -
11   fungsi          type         -          -     1
0    11    -
12   jika            type         -          -     1
0    12    -
13   mod             type         -          -     1
0    13    -
14   tidak           type         -          -     1
0    14    -
15   dari            type         -          -     1
0    15    -
16   atau            type         -          -     1
0    16    -
17   prosedur        type         -          -     1
0    17    -
18   program         type         -          -     1
0    18    -
19   rekaman         type         -          -     1
0    19    -
20   ulangi          type         -          -     1
0    20    -
21   string          type         4          -     1
0    21    -
22   maka            type         -          -     1
```

```
0    22    -
23   ke              type         -          -     1
0    23    -
24   tipe            type         -          -     1
0    24    -
25   sampai          type         -          -     1
0    25    -
26   variabel        type         -          -     1
0    26    -
27   selama          type         -          -     1
0    27    -
28   padat           type         -          -     1
0    28    -
29   writeln         procedure    0          -     1
0    29    -
30   write           procedure    0          -     1
0    30    -
31   readln          procedure    0          -     1
0    31    -
32   read            procedure    0          -     1
0    32    -
33   NestedTest      program      0          -     1
0    0     -
34   x               variable     1          -     1
0    0     -
35   Outer           procedure    0          1     1
0    0     -
36   Inner           procedure    0          3     1
1    0     -

Block Table (btab):
idx  last   lpar   psze   vsze
-----------------------------
0    35     0      0      1
1    0      0      0      0
2    36     0      0      0
3    0      0      0      0
4    0      0      0      0
5    0      0      0      0


---DECORATED AST---
Program(name: 'NestedTest')
  Declarations
    VarDecl('x') → tab_index:34, type:integer, lev:0
    ProcDecl(name: 'Outer', tab_index: 35,
block_index: 1)
      Declarations:
        ProcDecl(name: 'Inner', tab_index: 36,
block_index: 3)
          Body:
            Block → block_index:4, lev:2
```

```
              Assign('x' := 10) → type:integer
                Var(name: 'x', type: integer,
tab_index: 34, level: 0)
                Literal(value: 10, type: integer)
              writeln(...) → predefined, tab_index:29
      Body:
        Block → block_index:2, lev:1
          Inner(...), tab_index:36
          writeln(...) → predefined, tab_index:29
  Block
    Block → block_index:5, lev:1
      Assign('x' := 0) → type:integer
        Var(name: 'x', type: integer, tab_index: 34,
level: 0)
        Literal(value: 0, type: integer)
      Outer(...), tab_index:35

--------------
```

Kesimpulan

Semantic analysis memastikan struktur syntax yang dibuat oleh parses tidak hanya benar secara bentuk, tetapi juga memiliki makna yang valid. Dengan menggunakan attributed grammar, compiler memberi anotasi pada parse tree berupa atribut seperti tipe data dan nilai ekspresi. Symbol table digunakan untuk memverifikasi penggunaan identifier, tipe, dan scope. Proses ini dijalankan melalui mekanisme visitor function yang memisahkan struktur daata dari operasi semantik sehingga setiap node diperiksa dan diperkaya secara modular. Hasil akhir dari semantic analysis adalah struktur abstract syntax tree yang telah diverifikasi secara semantik.

Saran

1. Pengujian menggunakan banyak kasus unik
2. Pastikan lexical analyzer dan parser berjalan dengan baik

Lampiran

Link Release:
https://github.com/V-Kleio/CGK-Tubes-IF2224/releases/tag/v0.3.0

Pembagian Tugas:

NIM	Tugas	Persentase Kontribusi
13523128	Semantic analysis impl, Laporan	40%
13523145	Laporan	15%
13523146	Laporan	15%
13523152	Update Parser dari M2, Testing, dan Laporan	30%

Referensi

“Analisis Semantik dalam Kompilasi”
https://www.scribd.com/document/353902415/321748-Analisis-Semantik

“Attributed Grammars in Compiler Design”
https://www.tutorialspoint.com/compiler_design/compiler_design_attributed_grammars.htm

“Symbol Table in Compiler”
https://www.geeksforgeeks.org/compiler-design/symbol-table-compiler/