IF2124 Teori Bahasa Formal dan Otomata

Milestone 2
Syntax Analysis

Laporan Tugas Besar

Disusun untuk memenuhi tugas besar mata kuliah IF 2124 Teori Bahasa Formal dan Otomata
pada Semester I Tahun Akademik 2025/2026

Oleh

Raka Daffa Iftikhaar	13523018
Aliya Husna Fayyaza	13523062
Ahsan Malik Al Farisi	13523074
Bevinda Vivian	13523120

Kelompok Ahsan Et Al (NTB)

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA - KOMPUTASI
PROGRAM STUDI TEKNIK INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG

2025

Daftar Isi

BAB I

Landasan Teori

1.1. Compiler dan Fase-Fase Kompilasi

Compiler adalah sebuah program komputer yang menerjemahkan *source code* yang ditulis dalam bahasa pemrograman tingkat tinggi menjadi bahasa mesin atau *object code* yang dapat dieksekusi langsung oleh komputer. Proses kompilasi merupakan serangkaian transformasi yang kompleks dan terstruktur, yang secara umum dibagi menjadi enam fase utama.

1) Fase pertama adalah analisis leksikal (*lexical analysis*), di mana kode sumber dibaca karakter demi karakter dan dikelompokkan menjadi unit-unit leksikal yang disebut token.
2) Fase kedua adalah analisis sintaks (*syntax analysis atau parsing*), yang memeriksa apakah urutan token membentuk struktur yang sesuai dengan tata bahasa (*grammar*) bahasa pemrograman.
3) Fase ketiga adalah analisis semantik (*semantic analysis*), yang memeriksa konsistensi makna dari program, seperti pengecekan tipe data dan deklarasi variabel.
4) Fase keempat adalah pembangkitan kode antara (*intermediate code generation*), yang menghasilkan representasi abstrak dari program yang lebih mudah dioptimasi.
5) Fase kelima adalah optimasi kode (*code optimization*), yang bertujuan memperbaiki kode antara agar lebih efisien tanpa mengubah fungsionalitas program.
6) Fase terakhir adalah pembangkitan kode *(code generation*), yang menghasilkan kode mesin atau assembly yang dapat dieksekusi.

Pada Milestone 2 ini, fokus utama adalah pada fase kedua, yaitu analisis sintaks, yang merupakan tahap lanjutan dari analisis leksikal. Analisis sintaks berfungsi untuk melakukan pengecekan validitas dari urutan token yang ada atau memverifikasi struktur program berdasarkan grammar Pascal-S yang telah didefinisikan di dokumen spesifikasi *milestone* 2. Selain itu analisis sintaks akan menghasilkan *Parse Tree* menggunakan metode *Recursive Descent Parsing.*

1.2. Analisis Sintaks dan Peran Parser

Analisis sintaks adalah proses evaluasi urutan token yang dihasilkan oleh lexer untuk membentuk struktur bahasa yang valid sesuai dengan grammar formal.

Grammar akan mendefinisikan aturan tata bahasa yang sah, misalnya bagaimana deklarasi, ekspresi, dan statement dituliskan. Peran parser meliputi:

1. Memastikan struktur program valid.

   Parser mengecek apakah token mengikuti aturan produksi grammar. Jika terjadi pelanggaran, parser akan menghasilkan pesan syntax error.
2. Membangun Parse Tree.

   Parse Tree adalah struktur pohon yang menunjukkan bagaimana token dihasilkan dari grammar. Parser mengeksekusi aturan-aturan grammar secara rekursif hingga seluruh input diproses.
3. Menjadi dasar bagi analisis semantik.

   Struktur hirarkis yang dihasilkan parser akan digunakan oleh fase berikutnya, seperti pengecekan tipe variabel, scoping identifier, hingga evaluasi ekspresi.

Jika lexer memetakan karakter menjadi token, maka parser memetakan token menjadi struktur sintaksis.

1.3. Teori Bahasa Formal dan Hierarki Chomsky

Untuk memahami bagaimana parser bekerja, kita perlu memahami konsep teori bahasa formal. Bahasa formal adalah himpunan string yang dibentuk dari alfabet tertentu menurut aturan-aturan yang terdefinisi dengan jelas. Dalam konteks kompilasi, bahasa formal digunakan untuk mendefinisikan struktur leksikal dan sintaksis dari bahasa pemrograman yang disusun berdasarkan Hirarki Chomsky.

Noam Chomsky mengklasifikasikan bahasa formal menjadi empat tingkatan dalam Hierarki Chomsky, dari yang paling umum hingga yang paling terbatas:

1. Type-0 (*Unrestricted Grammar*) memiliki aturan produksi tanpa batasan dan setara dengan bahasa yang dapat diterjemahkan oleh mesin Turing.
2. Type-1 (*Context-Sensitive Grammar*) memiliki aturan produksi bergantung pada konteks simbol sekelilingnya dan digunakan untuk bahasa yang membutuhkan struktur kompleks.
3. Type-2 (*Context-Free Grammar*) memiliki aturan produksi berbentuk A -> a, dengan A adalah non-terminal dan ‘a’ adalah rangkaian terminal. Selain itu mayoritas bahasa pemrograman modern dinyatakan dengan CFG.
4. Type-3 (*Regular Grammar*). adalah bahasa yang diwakili oleh regular expression atau finite automata dan digunakan oleh lexer karena memiliki kemampuan yang lemah untuk mendeskripsikan struktur program yang bersarang.

Untuk analisis sintaks, grammar yang digunakan adalah Context-Free Grammar, karena CFG mampu mewakili struktur rekursif seperti blok dalam blok, ekspresi aritmetika bersarang, statement bertingkat, dan struktur prosedur dan fungsi.

1.4. Context-Free Grammar

Context-Free Grammar adalah representasi formal dari struktur tata bahasa suatu bahasa pemrograman. CFG terdiri dari empat komponen, yaitu:

1. V : himpunan non-terminal
2. Σ : himpunan terminal (token)
3. R : himpunan aturan produksi
4. S : simbol awal (start symbol)

Berikut juga aturan produksi CFG untuk bahasa Pascal-S, seperti berikut:

program → program-header declaration-part compound-statement DOT
program-header → KEYWORD(program) IDENTIFIER SEMICOLON
statement → assignment-statement | if-statement | while-statement | ...
expression → simple-expression (relational-operator simple-expression)?

Grammar ini akan menjadi acuan dalam membangun parse tree menggunakan metode Recursive Descent.

1.5. Parse Tree

Parse tree merupakan representasi struktur dari token input dari hasil lexer yang mengikuti aturan produksi CFG. Setiap node pada Parse Tree merepresentasikan hasil ekspresi dari sebuah non-terminal. Adapun ciri-ciri parse tree dari Pascal-S seperti berikut:

- Root akan selalu dimulai dengan simbol awal grammar yaitu “<program>”.
- Node internal adalah non-terminal.
- Node daun adalah terminal yang berasal dari token lexer.
- Struktur tree mengikut langkah derivasi grammar dari atas ke bawah.

1.6. Recursive Descent Parser

Recursive Descent adalah teknik parsing *top-down* menggunakan satu fungsi rekursif untuk setiap non-terminal pada grammar. Teknik ini cocok digunakan pada grammar Pascal-S karena:

- Struktur grammar jelas dan deterministik.
- Produksi dapat dianalisis tanpa backtracking jika grammar sudah disesuaikan agar bebas dari *left recursion.*
- Implementasi yang intuitif dan mudah dimodularisasi.

Setiap fungsi dalam parser akan memeriksa token saat ini, kemudian memastikan kesesuaian dengan aturan produksi. Setelah itu, jika valid maka akan dibangun node parse tree, kemudian berpindah ke token berikutnya dan mengeluarkan error jika token tidak sesuai.

Recursive Descent akan memudahkan pembacaan logika grammar karena struktur fungsi mencerminkan langsung aturan produksi.

BAB 2

Perancangan & Implementasi

2.1. Gambaran Umum Implementasi Parsing

Secara garis besar, proses *parsing* dimulai setelah *lexer* menghasilkan daftar token. Alur kerjanya adalah sebagai berikut:

1. Inisialisasi dengan main.cpp yang membuat objek Parser dan memberinya token (*std::vector<Token>*) lalu memanggil fungsi utama *parser.pars_program()*
2. Top-Down Recursive Descent yaitu *pars_program()* bertindak sebagai fungsi dengan level tertinggi dan bertugas untuk mengurai bagian-bagian utama program seperti *program_header*, *declaration_part*, dan *compound_statement*
3. Konstruksi *parse tree*: saat *parse_program()* perlu mengurai *declaration_part*, ia memanggil fungsi *pars_declaration_part()* yang akan mengurai semua deklarasi *variabel* dan mengembalikan sebuah *std::unique_ptr<DeclarationPartNode>* yang sudah terisi
4. ProgramNode (yang dibuat oleh *pars_program()*) kemudian menyimpan *pointer* ke DeclarationPartNode tersebut
5. Proses ini akan terus berlanjut secara rekursif ke bawah. *pars_compound_statement()* memanggil *pars_statement_list()*, yang memanggil *pars_statement()*, yang memanggil *pars_if_statement()*, yang memanggil *pars_expression()*, dan seterusnya.
6. Hasil akhirnya adalah semua token terkonsumsi (atau *error* ditemukan), *pars_program()* akan mengembalikan satu *std::unique_ptr<ProgramNode>* ke main.cpp. *Node* ini adalah akar dari keseluruhan Abstract Syntax Tree yang berisi semua *node* anak yang telah dibangun.

2.2. Adaptasi Bahasa

Untuk memenuhi spesifikasi, *keyword* diubah dari Bahasa Inggris menjadi Bahasa Indonesia. Perubahan ini terjadi pada bagian *char_classes.cpp* berikut:

```
const std::unordered_set<std::string> KEYWORDS = {
      "program", "konstanta", "tipe", "variabel", "prosedur", "fungsi",
      "mulai",  "selesai",  "jika",  "maka",  "selain-itu",  "selama",
      "lakukan",
      "untuk", "ke", "turun-ke", "ulangi", "sampai", "kasus", "dari",
      "larik", "rekaman",
      "integer", "real", "boolean", "char", "string",
```

```
        "true", "false", "writeln", "write", "readln", "read"
};
const std::unordered_set<std::string> LOGICAL_WORDS = {
        "dan", "atau", "tidak"
};
const std::unordered_set<std::string> ARITH_WORDS = {
        "bagi", "mod"
};
```

2.3. Implementasi Modul AST

Modul ini bertugas untuk menyediakan definisi bentuk dari *parse tree* yang akan muncul sebagai *output*. Modul ini diimplementasikan dalam *ast_nodes.cpp* dan *ast_nodes.hpp*. Setiap simpul yang ada di *tree* mewakili sebuah konstruksi dalam kode. Sebagai *base class*, terdapat kelas ASTNode yang mendefinisikan *interface* umum untuk semua jenis simpul AST. Selanjutnya, untuk mengakomodasi tiap konstruksi *grammar*, dibuat beberapa kelas turunan yaitu:

- ProgramNode: Simpul akar yang menyimpan *header* program, bagian deklarasi, dan *compound statement*
- DeclarationPartNode: Menyimpan daftar untuk semua bagian deklarasi seperti ConstDeclaration, TypeDeclarationNode, VariableDeclarationNode, SubprogramDeclarationNode
- VariableDeclarationNode: Menyimpan IdentifierListNode dan TypeNode
- CompoundStatementNode: Menyimpan daftar *statements* yang berada di antara mulai dan selesai
- IdentifierListNode: Menyimpan daftar token identifier
- TypeNode: Menyimpan token tipe data

Berikut adalah cuplikan kode implementasinya

```
class ASTNode {
        public:
        virtual ~ASTNode() = default;
        virtual std::string toString() const { return "ASTNode"; }
        virtual std::vector<ASTNode*> getChildren() const { return {}; }
};

class ProgramNode : public ASTNode {
        public:
        std::string pars_program_name;
        std::unique_ptr<ASTNode> pars_program_header;
        std::unique_ptr<class DeclarationPartNode> pars_declaration_part;
        std::unique_ptr<class                          CompoundStatementNode>
        pars_compound_statement;
        Token dot_token; // DOT token at end
```

```
        std::string toString() const override { return "<program>"; }
        std::vector<ASTNode*> getChildren() const override;
};

class DeclarationPartNode : public ASTNode {
        public:
        std::vector<std::unique_ptr<class            ConstDeclarationNode>>
        pars_const_declaration_list;
        std::vector<std::unique_ptr<class             TypeDeclarationNode>>
        pars_type_declaration_list;
        std::vector<std::unique_ptr<class         VariableDeclarationNode>>
        pars_variable_declaration_list;
        std::vector<std::unique_ptr<class       SubprogramDeclarationNode>>
        pars_subprogram_declaration_list;
        std::string     toString()     const     override     {     return
        "<declaration-part>"; }
        std::vector<ASTNode*> getChildren() const override;
};


class ProgramHeaderNode : public ASTNode {
        public:
        Token program_keyword; // KEYWORD(program)
        Token program_name; // IDENTIFIER(name)
        Token semicolon; // SEMICOLON(;)
        std::string toString() const override { return "<program-header>";
        }
        std::vector<ASTNode*> getChildren() const override { return {}; }
};

        class VariableDeclarationNode : public ASTNode {
        public:
        Token var_keyword; // KEYWORD(variabel)
        std::unique_ptr<class IdentifierListNode> pars_identifier_list;
        Token colon; // COLON(:)
        std::unique_ptr<ASTNode> pars_type;
        Token semicolon; // SEMICOLON(;)
        std::string     toString()     const     override     {     return
        "<var-declaration>"; }
        std::vector<ASTNode*> getChildren() const override;
};
```

```
class IdentifierListNode : public ASTNode {
	public:
	std::vector<std::string> pars_identifier_list;
	std::vector<Token> identifier_tokens; // Store actual tokens
	std::vector<Token> comma_tokens; // Store comma tokens
	std::string toString() const override { return
	"<identifier-list>"; }
	std::vector<ASTNode*> getChildren() const override;
};

class TypeNode : public ASTNode {
	public:
	std::string pars_type_name;
	Token type_keyword;
	std::string toString() const override { return "<type>"; }
	std::vector<ASTNode*> getChildren() const override;
};

class CompoundStatementNode : public ASTNode {
	public:
	Token mulai_keyword; // KEYWORD(mulai)
	std::vector<std::unique_ptr<ASTNode>> pars_statement_list;
	Token selesai_keyword; // KEYWORD(selesai)
	std::string toString() const override { return
	"<compound-statement>"; }
	std::vector<ASTNode*> getChildren() const override;
};

class StatementListNode : public ASTNode {
	public:
	std::vector<std::unique_ptr<ASTNode>> pars_statements;
	std::string toString() const override { return "<statement-list>";
	}
	std::vector<ASTNode*> getChildren() const override;
};


class TokenNode : public ASTNode {
	public:
	Token token;
	TokenNode(const Token& t) : token(t) {}
	std::string toString() const override {
```

```
        return token.toString();
        }
        std::vector<ASTNode*> getChildren() const override { return {}; }
};
```

Selanjutnya, untuk keperluan visualisasi, setiap simpul AST mengimplementasikan fungsi *getChildren()* yang mengembalikan daftar *pointer* ke anak-anaknya. Berikut adalah cuplikan kodenya:

```
// ProgramNode getChildren implementation
std::vector<ASTNode*> ProgramNode::getChildren() const {
        std::vector<ASTNode*> children;
        if (pars_program_header) {
        children.push_back(pars_program_header.get());
        }
        if (pars_declaration_part) {
        children.push_back(pars_declaration_part.get());
        }
        if (pars_compound_statement) {
        children.push_back(pars_compound_statement.get());
        }
        return children;
}

// DeclarationPartNode getChildren implementation
std::vector<ASTNode*> DeclarationPartNode::getChildren() const {
        std::vector<ASTNode*> children;
        for (const auto& const_decl : pars_const_declaration_list) {
        if (const_decl) {
        children.push_back(const_decl.get());
        }
        }
        for (const auto& type_decl : pars_type_declaration_list) {
        if (type_decl) {
        children.push_back(type_decl.get());
        }
        }
        for (const auto& var_decl : pars_variable_declaration_list) {
        if (var_decl) {
        children.push_back(var_decl.get());
        }
        }
```

```
        for (const auto& subprog_decl : pars_subprogram_declaration_list)
        {
        if (subprog_decl) {
        children.push_back(subprog_decl.get());
}
}
return children;
}


// VariableDeclarationNode getChildren implementation
std::vector<ASTNode*> VariableDeclarationNode::getChildren() const {
        std::vector<ASTNode*> children;
        if (pars_identifier_list) {
                children.push_back(pars_identifier_list.get());
        }
        if (pars_type) {
                children.push_back(pars_type.get());
        }
        return children;
}

// IdentifierListNode getChildren implementation
std::vector<ASTNode*> IdentifierListNode::getChildren() const {
        return {};
}

// TypeNode getChildren implementation
std::vector<ASTNode*> TypeNode::getChildren() const {
        return {};
}

// CompoundStatementNode getChildren implementation
std::vector<ASTNode*> CompoundStatementNode::getChildren() const {
        std::vector<ASTNode*> children;
        for (const auto& stmt : pars_statement_list) {
                if (stmt) {
                        children.push_back(stmt.get());
                }
        }
        return children;
}
```

```
// StatementListNode getChildren implementation
std::vector<ASTNode*> StatementListNode::getChildren() const {
	std::vector<ASTNode*> children;
	for (const auto& stmt : pars_statements) {
		if (stmt) {
			children.push_back(stmt.get());
		}
	}
	return children;
}
```

2.4. Implementasi Parser

Parser adalah kelas utama yang mengonsumsi aliran token aliran token dari *lexer* dan menghasilkan AST. Kelas ini mengambil *std::vector<Token>* dari *lexer* dan menghasilkan *std::unique_ptr<ProgramNode>* yang akan menjadi akar dari AST. Parser menyimpan *state* proses *parsing*, termasuk posisi saat ini (*currrent_pos*) dan token saat ini (*current_token*).

Ada beberapa fungsi utama yang digunakan untuk mengatur aliran token yaitu:

- *advance()*: memajukan *parser* ke token berikutnya (mengonsumsi token)
- *check(type)*: memeriksa tipe token saat ini (tanpa mengonsumsinya)
- *match(type)*: jika token saat ini cocok dengan *type*, program akan memanggil *advance()* dan mengembalikan nilai *true* (dan *false* ketika tidak)
- *expect(type, message)*: mengharuskan token saat ini memiliki *type*, jika *match()* gagal, program akan melempar SyntaxError
- *peek(offset)*: melihat token di depan (sesuai *offset*) tanpa memajukan *current_pos*
- *previous()*: mengambil token yang baru saja dikonsumsi (token di *current_pos* -1)

Berikut adalah cuplikan kodenya:

```
void Parser::advance() {
	if (current_pos < tokens.size() - 1) {
	current_pos++;
	current_token = tokens[current_pos];
	}
}

bool Parser::match(const std::string& type) {
```

```
        if (check(type)) {
                advance();
                return true;
        }
        return false;
}

bool Parser::check(const std::string& type) {
        return current_token.type == type;
}

void Parser::expect(const std::string& type, const std::string& message)
{
        if (!match(type)) {
                std::stringstream ss;
                ss << "Syntax error at line " << current_token.line
                << ", column " << current_token.column
                << ": " << message
                << " (got " << current_token.type << "(" <<
                current_token.value << "))";
        throw SyntaxError(ss.str());
        }
}

Token Parser::peek(int offset) {
        size_t pos = current_pos + offset;
        if (pos < tokens.size()) {
                return tokens[pos];
        }
        return current_token;
}

Token Parser::previous() {
        if (current_pos > 0) {
                return tokens[current_pos - 1];
        }
return current_token;
}
```

Selanjutnya, ada beberapa fungsi yang membangun kerangka dari program Pascal-S berdasarkan aturan-aturan *grammar* seperti:

- *pars_program()*: *entry point* yang memanggil *pars_program_header(), pars_declaration_part(),* dan *pars_compound_statement()* secara

berurutan, mengumpulkan *node* yang fungsi-fungsi tersebut kembalikan, dan menyimpannya dalam ProgramNode yang dibuat. Terakhir, fungsi ini memeriksa adanya DOT (.) di akhir program

- *pars_declaration_part()*: bekerja dengan memeriksa satu kali untuk setipa *keyword* blok (konstanta, tipe, variabel). Jika ditemukan (misal tipe), fungsi ini akan mengonsumsi *keyword* tersebut dan masuk ke *loop* internal, memanggil fungsi *pars_type_declaration()* berulang kali sampai menemukan token yang bukan lagi IDENTIFIER.
- *pars_compound_statement()*: fungsi ini akan mengharapkan *keyword* "mulai", lalu memanggil *pars_statement_list()* untuk mengurai semua *statement* di dalam blok, lalu mengharapkan *keyword* "selesai"
- *pars_statement_list()*; fungsi ini akan memeriksa jika blok kosong (blok yang token berikutnya adalah "selesai"), memanggil *parse_statement()* untuk mengurai *statement* pertama, lalu masuk ke *while loop* yang selama match("SEMICOLON") berhasil, ia akan menyimpan token semikolon ke dalam daftar *statement*, memeriksa lagi untuk "selesai" (untuk menangani semikolon di akhir sebelum "selesai"), dan memanggil *pars_statement()* untuk mengurai *statement* berikutnya.

Berikut adalah cuplikan kodenya:

```
std::unique_ptr<ProgramNode> Parser::pars_program() {
	auto prog_node = std::make_unique<ProgramNode>();
	auto header = pars_program_header();
	if (auto* prog_header =
	dynamic_cast<ProgramHeaderNode*>(header.get())) {
		prog_node->pars_program_name =
	prog_header->program_name.value;
	}
	prog_node->pars_program_header = std::move(header);
	prog_node->pars_declaration_part = pars_declaration_part();
	prog_node->pars_compound_statement = pars_compound_statement();
	if (!check("DOT")) {
		throw SyntaxError("Expected '.' at end of program");
	}
	prog_node->dot_token = current_token;
	advance();
	return prog_node;
}

std::unique_ptr<ASTNode> Parser::pars_program_header() {
	auto header_node = std::make_unique<ProgramHeaderNode>();
	if (!check("KEYWORD") || current_token.value != "program") {
		throw SyntaxError("Expected keyword 'program'");
```

```
        }
        header_node->program_keyword = current_token;
        advance();
        if (check("IDENTIFIER")) {
                header_node->program_name = current_token; // Save token
                advance();
        } else {
                throw SyntaxError("Expected program name (identifier)");
        }
        if (!check("SEMICOLON")) {
                throw SyntaxError("Expected ';' after program name");
        }
        header_node->semicolon = current_token; // Save token
        advance();
        return header_node;
}

std::unique_ptr<DeclarationPartNode> Parser::pars_declaration_part() {
        auto decl_part_node = std::make_unique<DeclarationPartNode>();
        if (check("KEYWORD") && current_token.value == "konstanta") {
                Token const_keyword = current_token;
                advance(); // consume 'konstanta'
                while (check("IDENTIFIER")) {
                        auto const_decl = pars_const_declaration();
                        const_decl->const_keyword = const_keyword;
                        decl_part_node->pars_const_declaration_list.push_back
                        (std::move(const_decl));
                }
        }
        if (check("KEYWORD") && current_token.value == "tipe") {
                Token type_keyword = current_token;
                advance(); // consume 'tipe'
                while (check("IDENTIFIER")) {
                        auto type_decl = pars_type_declaration();
                        type_decl->type_keyword = type_keyword;
                        decl_part_node->pars_type_declaration_list.push_back(
                        std::move(type_decl));
                }
        }
        if (check("KEYWORD") && current_token.value == "variabel") {
                Token var_keyword = current_token;
                advance(); // consume 'variabel'
```

```
            while (check("IDENTIFIER")) {
                  auto var_decl = pars_variable_declaration_part();
                  var_decl->var_keyword = var_keyword;
                  decl_part_node->pars_variable_declaration_list.push_b
                  ack(std::move(var_decl));
            }
      }
      while (check("KEYWORD") && (current_token.value == "prosedur" ||
      current_token.value == "fungsi")) {
            auto subprog_decl = pars_subprogram_declaration();
            decl_part_node->pars_subprogram_declaration_list.push_back(
            std::move(subprog_decl));
      }
      return decl_part_node;
}

std::unique_ptr<VariableDeclarationNode>
Parser::pars_variable_declaration_part() {
      auto var_decl_node = std::make_unique<VariableDeclarationNode>();
      var_decl_node->pars_identifier_list = pars_identifier_list();
      if (!check("COLON")) {
            std::stringstream ss;
            ss << "Error at line " << current_token.line << ", column "
            << current_token.column
            << ": Expected ':' after variable identifier list\n"
            << " Variables: ";
            for (const auto& id :
            var_decl_node->pars_identifier_list->pars_identifier_list)
            {
            ss << id << " ";
      }
      ss << "\n Got: " << current_token.type << "(" <<
      current_token.value << ")";
      throw SyntaxError(ss.str());
      }
      var_decl_node->colon = current_token;
      advance();
      var_decl_node->pars_type = pars_type();
      if (!check("SEMICOLON")) {
            std::stringstream ss;
            ss << "Error at line " << current_token.line << ", column "
            << current_token.column
```

```
                << ": Expected ';' after variable type declaration\n"
                // << " Type: " << var_decl_node->pars_type->pars_type_name
                << "\n" // Tidak bisa akses ini lagi
                << " Got: " << current_token.type << "(" <<
                current_token.value << ")";
                throw SyntaxError(ss.str());
        }
        var_decl_node->semicolon = current_token;
        advance();
        return var_decl_node;
}

std::unique_ptr<IdentifierListNode> Parser::pars_identifier_list() {
        auto id_list_node = std::make_unique<IdentifierListNode>();
        if (check("IDENTIFIER")) {
                id_list_node->pars_identifier_list.push_back(current_token.
                value);
                id_list_node->identifier_tokens.push_back(current_token);
                advance();
                while (match("COMMA")) {
                        Token comma_token = tokens[current_pos - 1];
                        id_list_node->comma_tokens.push_back(comma_token);
                        if (check("IDENTIFIER")) {
                                id_list_node->pars_identifier_list.push_back(c
                                urrent_token.value);
                                id_list_node->identifier_tokens.push_back(curr
                                ent_token); advance();
                        } else {
                                throw SyntaxError("Expected identifier after
                        ',',");
                        }
                }
        } else {
                throw SyntaxError("Expected identifier");
        }
        return id_list_node;
}

std::unique_ptr<ASTNode> Parser::pars_type() {
        if (check("KEYWORD") && current_token.value == "larik") {
                return pars_array_type();
        }
```

```
        if (check("KEYWORD")) {
                std::string type_value = current_token.value;
                if (type_value == "integer" ||
                type_value == "real" ||
                type_value == "boolean" ||
                type_value == "char") {
                auto type_node = std::make_unique<TypeNode>();
                type_node->pars_type_name = type_value;
                type_node->type_keyword = current_token; // Save token
                advance();
                return type_node;
                }
        }

        if (check("IDENTIFIER")) {
                auto type_node = std::make_unique<TypeNode>();
                type_node->pars_type_name = current_token.value;
                type_node->type_keyword = current_token;
                advance();
                return type_node;
        }
        throw SyntaxError("Expected type (integer, real, boolean, char,
        array, or custom type identifier)");
}

std::unique_ptr<CompoundStatementNode>  Parser::pars_compound_statement()
{
        auto compound_node = std::make_unique<CompoundStatementNode>();
        if (check("KEYWORD") && current_token.value == "mulai") {
                compound_node->mulai_keyword = current_token; // Save token
                advance();
        } else {
                throw SyntaxError("Expected keyword 'mulai'");
        }
        auto stmt_list = pars_statement_list();
        compound_node->pars_statement_list                                =
        std::move(stmt_list->pars_statements);
        if (check("KEYWORD") && current_token.value == "selesai") {
                compound_node->selesai_keyword  =  current_token;  //  Save
                token
                advance();
        } else {
```

```
            throw SyntaxError("Expected keyword 'selesai'");
        }
        return compound_node;
    }

    std::unique_ptr<StatementListNode> Parser::pars_statement_list() {
        auto stmt_list_node = std::make_unique<StatementListNode>();
        if (check("KEYWORD") && current_token.value == "selesai") {
            return stmt_list_node;
        }
        auto stmt = pars_statement();
        stmt_list_node->pars_statements.push_back(std::move(stmt));
        while (match("SEMICOLON")) {
            Token semicolon_token = previous();
            stmt_list_node->pars_statements.push_back(std::make_unique<
            TokenNode>(semicolon_token));
            if (check("KEYWORD") && current_token.value == "selesai") {
            break;
        }
        auto next_stmt = pars_statement();
        stmt_list_node->pars_statements.push_back(std::move(next_stmt));
        }
        return stmt_list_node;
    }
```

Selanjutnya, berikut adalah fungsi-fungsi yang digunakan untuk mengurai *statement* di dalam blok mulai…selesai:

- *pars_statement()*: fungsi ini memandu menentukan jenis *statement* apa yang akan diurai. Ia akan memeriksa *keyword* seperti “mulai”, “jika”, “selama”, dan “untuk”. Jika cocok, ia memanggil fungsi *parser* yang sesuai (misalnya pars_if_statement()). Jika token saat ini adalah IDENTIFIER, *parser* harus menggunakan *lookahead* untuk membedakan antara *assignment* dan *procedure call* (menggunakan Token = next = peek(1), jika next.type == “ASSIGN_OPERATOR” (:=), maka akan ke *pars_assignment_statement()*, jika tidak, maka akan ke *pars_procedure_call()*). Fungsi ini juga memeriksa *keyword* yang bertindak sebagai prosedur (misalnya *writeln* atau *readln*) dan mengarahkannya ke *pars_procedure_cal()*
- *pars_assignment_statement()*: mengurutkan urutan IDENTIFIER, ASSIGN_OPERATOR, lalu memanggil *pars_expression()* untuj mengurai sisi kanan dari *assignment* (tiap token akan disimpan dalam *AssignmentStatementNode*)
- *pars_procedure_call()*: mengurai nama prosedur (bisa IDENTIFIER atau KEYWORD seperti "writeln"). Jika token berikutnya adalah LPARENTHESIS, ia

akan mengurai pars_parameter_list() secara opsional, dan terakhir mengharapkan RPARENTHESIS.

- *pars_if_statement()*: fungsi ini akan mengharapkan "jika" lalu memanggil *pars_expression()* untuk mengurai kondisi, lalu mengharapkan "maka dan memanggil *pars_statement()* (secara rekursif) untuk mengurai badan *then* serta memeriksa (secara opsional) *keyword* "selain-itu" (kode ini juga secara khusus menangani semikolon opsional sebeleum "selain-itu", jika "selain-itu" ditemukan, fungsi akan memanggil *pars_statement* lagi untuk mengurai badan *else*)
- *pars_while_statement()*: mengharapkan "selama", memanggil *pars_expression()* untuk kondisi, mengharapkan "lakukan", dan memanggil *pars_statement()* untuk badan *loop*
- *pars_for_statement()*: fungsi ini akan mengurai "untuk", IDENTIFIER (sebagai variabel kontrol), ASSIGN_OPERATOR, *pars_expression()* (sebagai nilai awal), *keyword* "ke" atau "turun-ke", *pars_expression()* (nilai akhir), "lakukan", dan *pars_statement()* (badan *loop*)

Berikut adalah cuplikan kodenya:

```
std::unique_ptr<ASTNode> Parser::pars_statement() {
	if (check("SEMICOLON") ||
	(check("KEYWORD") && current_token.value == "selesai")) {
		return std::make_unique<ASTNode>(); // Empty statement node
	}
	// Compound statement (nested mulai-selesai)
	if (check("KEYWORD") && current_token.value == "mulai") {
		return pars_compound_statement();
	}
	// If statement
	if (check("KEYWORD") && current_token.value == "jika") {
		return pars_if_statement();
	}
	// While statement
	if (check("KEYWORD") && current_token.value == "selama") {
		return pars_while_statement();
	}
	// For statement
	if (check("KEYWORD") && current_token.value == "untuk") {
		return pars_for_statement();
	}
	// Assignment or procedure call (both start with IDENTIFIER)
	if (check("IDENTIFIER")) {
		// Look ahead to distinguish between assignment and procedure call
		Token next = peek(1);
```

```
            if (next.type == "ASSIGN_OPERATOR") {
                // Assignment statement
                return pars_assignment_statement();
            } else if (next.type == "LPARENTHESIS" ||
            next.type == "SEMICOLON" ||
            (next.type == "KEYWORD" && next.value == "selesai")) {
                // Procedure call
                return pars_procedure_call();
            }
        }
        // Built-in procedures (writeln, write, dll)
        if (check("KEYWORD")) {
            std::string keyword = current_token.value;
            if (keyword == "writeln" || keyword == "write" || keyword ==
            "readln" || keyword == "read") {
                return pars_procedure_call();
            }
        }
        std::stringstream ss;
        ss << "Syntax error at line " << current_token.line
        << ", column " << current_token.column
        << ": Unexpected token in statement\n"
        << " Got: " << current_token.type << "(" << current_token.value << ")\n"
        << " Expected one of: assignment, procedure call, if, while, for, or
        compound statement\n"
        << " Valid statement starters: identifier, jika, selama, untuk, mulai,
        writeln, write";
        throw SyntaxError(ss.str());
}

std::unique_ptr<ASTNode> Parser::pars_assignment_statement() {
        auto assign_node = std::make_unique<AssignmentStatementNode>();
        if (!check("IDENTIFIER")) {
            throw SyntaxError("Expected identifier in assignment statement");
        }
        assign_node->identifier = current_token;
        advance();
        if (!check("ASSIGN_OPERATOR")) {
            throw SyntaxError("Expected ':=' in assignment statement");
        }
        assign_node->assign_operator = current_token;
        advance();
```

```
	assign_node->pars_expression = pars_expression();
	return assign_node;
}

std::unique_ptr<ASTNode> Parser::pars_procedure_call() {
	auto proc_call_node = std::make_unique<ProcedureFunctionCallNode>();
	if (check("IDENTIFIER") || check("KEYWORD")) {
		proc_call_node->procedure_name = current_token;
		advance();
	} else {
		throw SyntaxError("Expected procedure name");
	}
	if (check("LPARENTHESIS")) {
		proc_call_node->lparen = current_token;
		advance();
		if (!check("RPARENTHESIS")) {
			proc_call_node->pars_parameter_list                    =
		pars_parameter_list();
		}
		if (!check("RPARENTHESIS")) {
			throw SyntaxError("Expected ')' after parameter list");
		}
		proc_call_node->rparen = current_token;
		advance();
	}
	return proc_call_node;
}

std::unique_ptr<ASTNode> Parser::pars_if_statement() {
	auto if_node = std::make_unique<IfStatementNode>();
	if (!check("KEYWORD") || current_token.value != "jika") {
		throw SyntaxError("Expected keyword 'jika'");
	}
	if_node->if_keyword = current_token;
	advance();
	if_node->pars_condition = pars_expression();
	if (!check("KEYWORD") || current_token.value != "maka") {
		throw SyntaxError("Expected keyword 'maka' after condition");
	}
	if_node->then_keyword = current_token;
	advance();
	if_node->pars_then_statement = pars_statement();
```

```
        if (check("SEMICOLON")) {
                Token next = peek(1);
        if (next.type == "KEYWORD" && next.value == "selain-itu") {
                advance();
        }
        }
        if (check("KEYWORD") && current_token.value == "selain-itu") {
        if_node->else_keyword = current_token;
        advance();
        if_node->pars_else_statement = pars_statement();
        }
        return if_node;
}

std::unique_ptr<ASTNode> Parser::pars_while_statement() {
        auto while_node = std::make_unique<WhileStatementNode>();
        if (!check("KEYWORD") || current_token.value != "selama") {
                throw SyntaxError("Expected keyword 'selama'");
        }
        while_node->while_keyword = current_token;
        advance();
        while_node->pars_condition = pars_expression();
        if (!check("KEYWORD") || current_token.value != "lakukan") {
                throw SyntaxError("Expected keyword 'lakukan' after condition");
        }
        while_node->do_keyword = current_token;
        advance();
        while_node->pars_body = pars_statement();
        return while_node;
}

std::unique_ptr<ASTNode> Parser::pars_for_statement() {
        auto for_node = std::make_unique<ForStatementNode>();
        if (!check("KEYWORD") || current_token.value != "untuk") {
                throw SyntaxError("Expected keyword 'untuk'");
        }
        for_node->for_keyword = current_token;
        advance();
        if (!check("IDENTIFIER")) {
                throw SyntaxError("Expected identifier after 'untuk'");
        }
        for_node->control_variable = current_token;
```

```
        advance();
        if (!check("ASSIGN_OPERATOR")) {
              throw SyntaxError("Expected ':=' in for statement");
        }
        for_node->assign_operator = current_token;
        advance();
        for_node->pars_initial_value = pars_expression();
        if    (!check("KEYWORD")    ||    (current_token.value    !=    "ke"    &&
        current_token.value != "turun-ke")) {
              throw SyntaxError("Expected keyword 'ke' or 'turun-ke'");
        }
        for_node->direction_keyword = current_token;
        advance();
        for_node->pars_final_value = pars_expression();
        if (!check("KEYWORD") || current_token.value != "lakukan") {
              throw SyntaxError("Expected keyword 'lakukan' after final value");
        }
        for_node->do_keyword = current_token;
        advance();
        for_node->pars_body = pars_statement();
        return for_node;
}
```

Terakhir, untuk menangani operator matematika dan logika (seperti +, *, dan, atau) dengan benar sesuai urutan operasi (operator *precedence*), *parser* menggunakan serangkaian fungsi yang saling memanggil secara rekursif. *pars_expression()* (*Precedence* terendah: =, <, >) memanggil *pars_simple_expression()* (*Precedence* menengah: +, -, atau) memanggil *pars_term()* (*Precedence* tinggi: *, /, bagi, mod, dan) ↓ memanggil *pars_factor()* (*Precedence* tertinggi: Angka, Variabel, (...), tidak ...). Berikut penjelasan mendetail untuk tiap fungsinya:

- *pars_expression()*: mengurai *pars_simple_expression()* (sisi kiri), jika menemukan operator relasional (=, <, >), ia akan mengkonsumsi operator dan mengurai *pars_simple_expression() lagi* untuk sisi kanan.
- *pars_simple_expression()*: mengurai *parse_term()* pertama, lalu masuk ke *while loop* yaitu selama menemukan operator multiplikatif (*, /, dan, bagi, mod), ia akan mengkonsumsi operator dan mengurai *pars_factor()* berikutnya
- *pars_factor()*: secara langsung mengkonsumsi token literal seperti NUMBER dan STRING_LITERAL, serta KEYWORD boolean ("benar" atau "salah"). Fungsi

ini juga menangani ekspresi yang dikurung dengan mengharapkan (, memanggil *pars_expression()* secara rekursif, lalu mengharapkan). Untuk negasi, fungsi ini mengkonsumsi "tidak" dan memanggil *pars_factor()* lagi. Kasus yang paling penting adalah ketika fungsi menemukan IDENTIFIER, di mana fungsi ini harus menggunakan *lookahead* yaitu *parser* menyimpan token IDENTIFIER tersebut, maju satu langkah, dan memeriksa token berikutnya. Jika token berikutnya adalah LPARENTHESIS (menandakan *function call* seperti x(...)), parser akan "mundur" dan memanggil *pars_procedure_call()*. Jika tidak, IDENTIFIER tersebut diperlakukan sebagai variabel biasa dan tokennya disimpan.

- *pars_parameter_list()*: mengurai *pars_expression() pertam, lalu masuk while loop* untuk mengurai ekspresi berikutnya selama *match("COMMA")* benar.

```
std::unique_ptr<ASTNode> Parser::pars_expression() {
	auto expr_node = std::make_unique<ExpressionNode>();
	expr_node->pars_left = pars_simple_expression();
	if (check("RELATIONAL_OPERATOR") || check("LOGICAL_OPERATOR")) {
		std::string op_val = current_token.value;
		if (op_val == "=" || op_val == "<>" || op_val == "<" ||
		op_val == "<=" || op_val == ">" || op_val == ">=") {
			auto rel_op_node =
			std::make_unique<RelationalOperatorNode>();
			rel_op_node->op_token = current_token;
			expr_node->pars_relational_op = std::move(rel_op_node);
			advance();
			expr_node->pars_right = pars_simple_expression();
		}
	}
	return expr_node;
}

std::unique_ptr<ASTNode> Parser::pars_simple_expression() {
	auto simple_expr_node = std::make_unique<SimpleExpressionNode>();
		if (check("ARITHMETIC_OPERATOR") && (current_token.value == "+" ||
		current_token.value == "-")) {
		simple_expr_node->sign = current_token;
		advance();
	}
```

```
        simple_expr_node->pars_terms.push_back(pars_term());
        while (check("ARITHMETIC_OPERATOR") || check("LOGICAL_OPERATOR")) {
                std::string op_val = current_token.value;
                if (op_val == "+" || op_val == "-" || op_val == "atau") {
                auto add_op_node = std::make_unique<AdditiveOperatorNode>();
                add_op_node->op_token = current_token;
                simple_expr_node->pars_operators.push_back(std::move(add_op_node))
                ;
                advance();
                simple_expr_node->pars_terms.push_back(pars_term());
        } else {
        break;
        }
        }
        return simple_expr_node;
}

std::unique_ptr<ASTNode> Parser::pars_factor() {
        auto factor_node = std::make_unique<FactorNode>();
        if (check("LOGICAL_OPERATOR") && current_token.value == "tidak") {
                factor_node->not_operator = current_token;
                advance();
                factor_node->pars_expression = pars_factor();
                return factor_node;
        }
        if (check("LPARENTHESIS")) {
                Token lparen = current_token;
                advance();
                factor_node->pars_expression = pars_expression();
                if (!check("RPARENTHESIS")) {
                throw SyntaxError("Expected ')' after expression");
        }
        advance();
        return factor_node;
        }
        if (check("NUMBER") || check("CHAR_LITERAL") || check("STRING_LITERAL"))
        {
                factor_node->token = current_token;
                advance();
                return factor_node;
        }
        if (check("IDENTIFIER")) {
```

```
        Token id_token = current_token;
        advance();
        if (check("LPARENTHESIS")) {
              current_pos--;
              current_token = id_token;
              auto                  proc_func_call_node                  =
              std::make_unique<ProcedureFunctionCallNode>();
              proc_func_call_node->procedure_name = current_token;
              advance();
              proc_func_call_node->lparen = current_token;
              advance();
        if (!check("RPARENTHESIS")) {
              proc_func_call_node->pars_parameter_list                   =
        pars_parameter_list();
        }
        if (!check("RPARENTHESIS")) {
              throw SyntaxError("Expected ')' after parameter list");
        }
        proc_func_call_node->rparen = current_token;
        advance();
        factor_node->pars_procedure_function_call                          =
        std::move(proc_func_call_node);
        return factor_node;
} else {
        factor_node->token = id_token;
        return factor_node;
        }
        }
        if (check("KEYWORD")) {
        std::string kw = current_token.value;
        if (kw == "benar" || kw == "salah" || kw == "true" || kw ==
        "false") {
              factor_node->token = current_token;
              advance();
              return factor_node;
        }
}
std::stringstream ss;
ss << "Syntax error at line " << current_token.line
<< ", column " << current_token.column
<< ": unexpected token in expression " << current_token.type << "(" <<
current_token.value << ")";
```

```
        throw SyntaxError(ss.str());
}

std::unique_ptr<ASTNode> Parser::pars_parameter_list() {
        auto param_list_node = std::make_unique<ParameterListNode>();
        param_list_node->pars_parameters.push_back(pars_expression());
        while (check("COMMA")) {
                param_list_node->comma_tokens.push_back(current_token);
                advance();
                param_list_node->pars_parameters.push_back(pars_expression());
        }
        return param_list_node;
}
```

BAB III

Pengujian

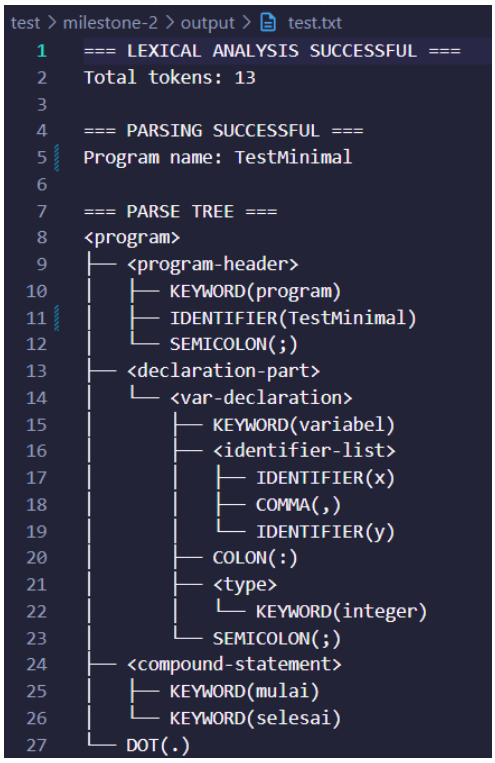
3.1. Test Case 1 - TestMinimal

Test case ini bertujuan untuk memvalidasi bahwa parser dapat mengenali program header, deklarasi variabel dengan multiple identifiers, dan compound statement kosong yaitu hanya berisi mulai dan selesai. Kasus ini ditujukan untuk memastikan struktur dasar program dapat di-parse dengan benar.

test.pas (input)

```
program TestMinimal;

variabel
  x, y: integer;

mulai
selesai.
```

Output:



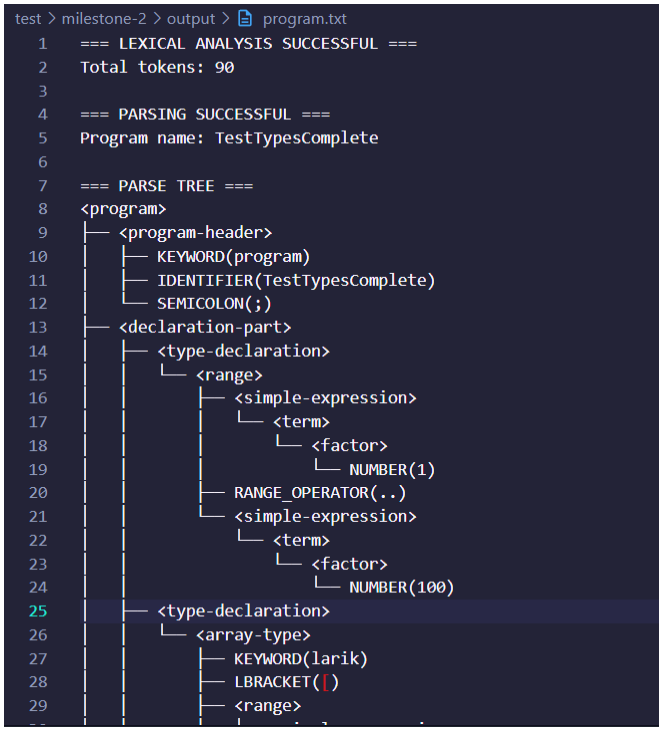
3.2. Test Case 2 - TestTypesComplete

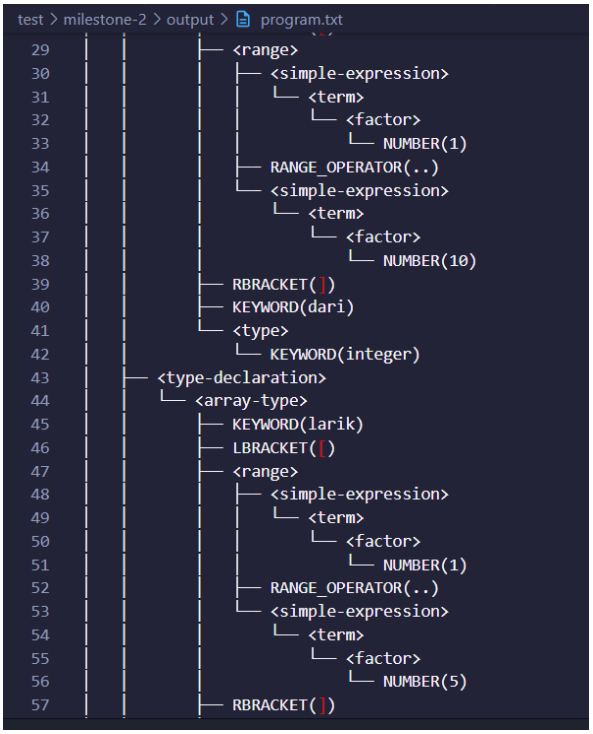
Test case ini bertujuan untuk menguji kemampuan parser dalam menangani deklarasi tipe yang kompleks termasuk subrange type (1..100), array type (larik[1..10] dari integer), serta penggunaan tipe custom dalam deklarasi variabel. Juga menguji for-statement dengan nested compound statement dan procedure call dengan parameter.

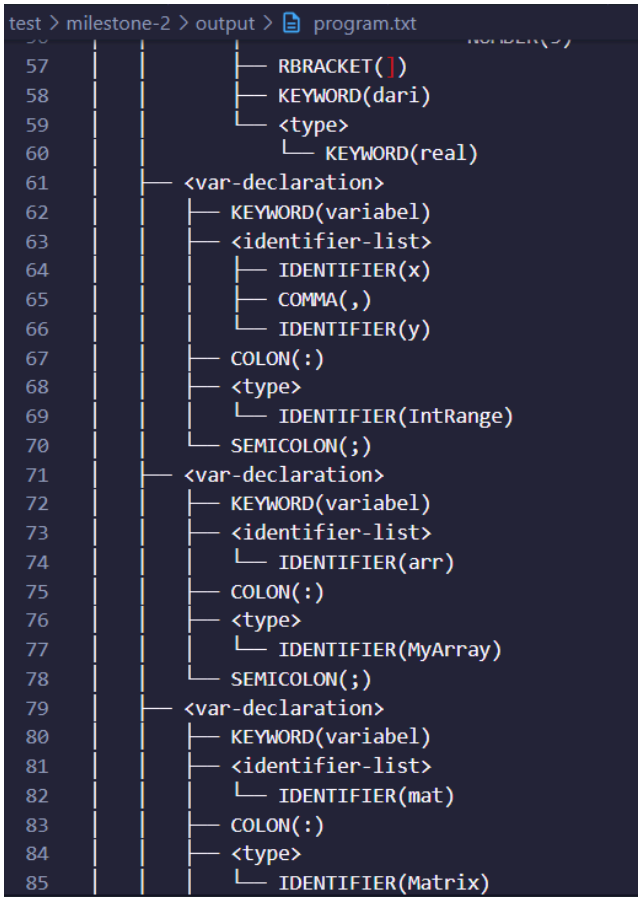
program.pas (Input)

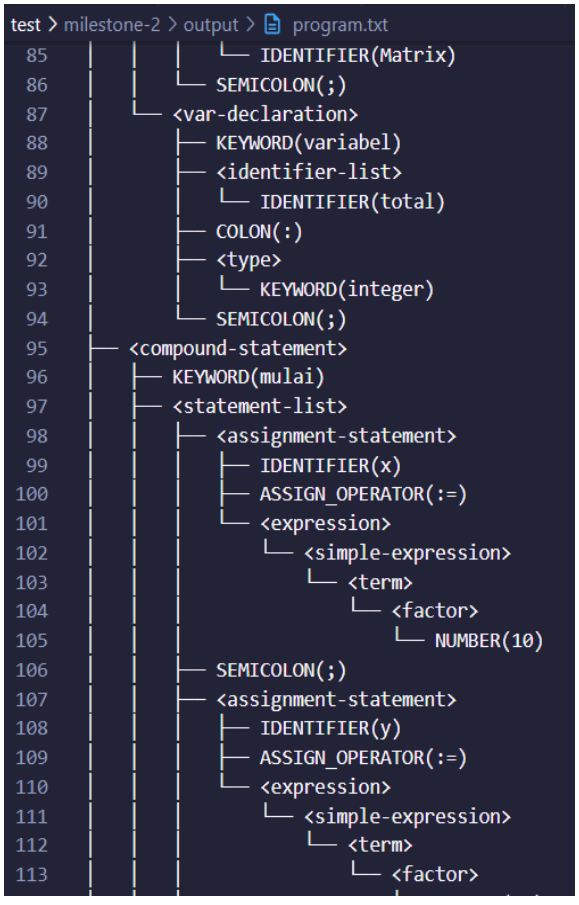
```
program TestTypesComplete;

tipe
    IntRange = 1..100;
    MyArray = larik[1..10] dari integer;
    Matrix = larik[1..5] dari real;

variabel
    x, y: IntRange;
    arr: MyArray;
    mat: Matrix;
    total: integer;

mulai
    x := 10;
    y := 20;
    total := 0;

    untuk x := 1 ke 10 lakukan mulai
        total := total + x
    selesai;

    writeln('Total: ');
    writeln(total)
selesai.
```

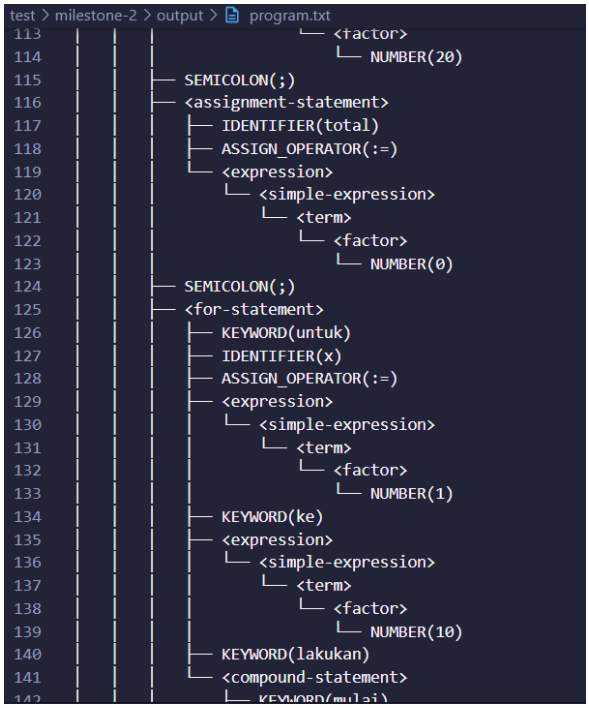
Output:


test > milestone-2 > output > program.txt
=== LEXICAL ANALYSIS SUCCESSFUL ===
Total tokens: 90
=== PARSING SUCCESSFUL ===
Program name: TestTypesComplete
=== PARSE TREE ===
<program>
├── <program-header>
│ ├── KEYWORD(program)
│ ├── IDENTIFIER(TestTypesComplete)
│ └── SEMICOLON(;)
├── <declaration-part>
│ ├── <type-declaration>
│ │ └── <range>
│ │ ├── <simple-expression>
│ │ │ └── <term>
│ │ │ └── <factor>
│ │ │ └── NUMBER(1)
│ │ ├── RANGE_OPERATOR(..)
│ │ └── <simple-expression>
│ │ └── <term>
│ │ └── <factor>
│ │ └── NUMBER(100)
│ ├── <type-declaration>
│ │ └── <array-type>
│ │ ├── KEYWORD(larik)
│ │ ├── LBRACKET([)
│ │ ├── <range>

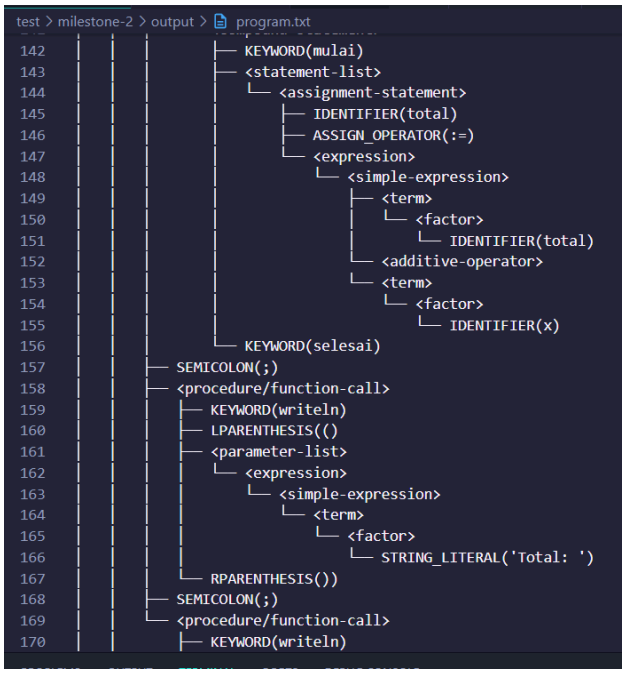

test > milestone-2 > output > program.txt
├── <range>
├── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(1)
├── RANGE_OPERATOR(..)
└── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(10)
├── RBRACKET(])
├── KEYWORD(dari)
└── <type>
└── KEYWORD(integer)
├── <type-declaration>
└── <array-type>
├── KEYWORD(larik)
├── LBRACKET([)
├── <range>
├── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(1)
├── RANGE_OPERATOR(..)
└── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(5)
├── RBRACKET(])


test > milestone-2 > output > program.txt
├── RBRACKET(])
├── KEYWORD(dari)
└── <type>
└── KEYWORD(real)
├── <var-declaration>
├── KEYWORD(variabel)
├── <identifier-list>
├── IDENTIFIER(x)
├── COMMA(,)
└── IDENTIFIER(y)
├── COLON(:)
├── <type>
└── IDENTIFIER(IntRange)
└── SEMICOLON(;)
├── <var-declaration>
├── KEYWORD(variabel)
├── <identifier-list>
└── IDENTIFIER(arr)
├── COLON(:)
├── <type>
└── IDENTIFIER(MyArray)
└── SEMICOLON(;)
├── <var-declaration>
├── KEYWORD(variabel)
├── <identifier-list>
└── IDENTIFIER(mat)
├── COLON(:)
├── <type>
└── IDENTIFIER(Matrix)

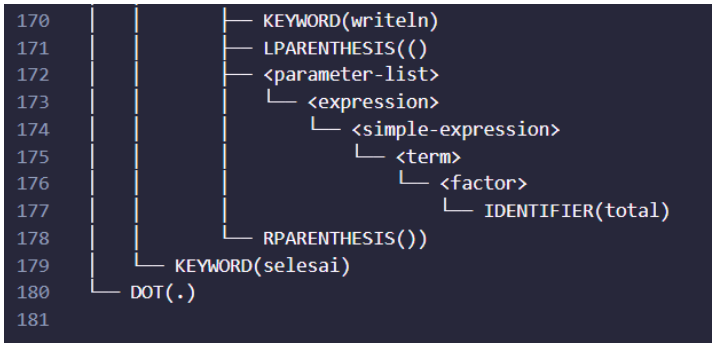

test > milestone-2 > output > program.txt
└── IDENTIFIER(Matrix)
└── SEMICOLON(;)
└── <var-declaration>
├── KEYWORD(variabel)
├── <identifier-list>
└── IDENTIFIER(total)
├── COLON(:)
├── <type>
└── KEYWORD(integer)
└── SEMICOLON(;)
├── <compound-statement>
├── KEYWORD(mulai)
├── <statement-list>
├── <assignment-statement>
├── IDENTIFIER(x)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(10)
├── SEMICOLON(;)
├── <assignment-statement>
├── IDENTIFIER(y)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>



```
test > milestone-2 > output > program.txt
113     │   │   │               └── <factor>
114     │   │   │                   └── NUMBER(20)
115     │   │   ├── SEMICOLON(;)
116     │   │   ├── <assignment-statement>
117     │   │   │   ├── IDENTIFIER(total)
118     │   │   │   ├── ASSIGN_OPERATOR(:=)
119     │   │   │   └── <expression>
120     │   │   │       └── <simple-expression>
121     │   │   │           └── <term>
122     │   │   │               └── <factor>
123     │   │   │                   └── NUMBER(0)
124     │   │   ├── SEMICOLON(;)
125     │   │   ├── <for-statement>
126     │   │   │   ├── KEYWORD(untuk)
127     │   │   │   ├── IDENTIFIER(x)
128     │   │   │   ├── ASSIGN_OPERATOR(:=)
129     │   │   │   ├── <expression>
130     │   │   │   │   └── <simple-expression>
131     │   │   │   │       └── <term>
132     │   │   │   │           └── <factor>
133     │   │   │   │               └── NUMBER(1)
134     │   │   │   ├── KEYWORD(ke)
135     │   │   │   ├── <expression>
136     │   │   │   │   └── <simple-expression>
137     │   │   │   │       └── <term>
138     │   │   │   │           └── <factor>
139     │   │   │   │               └── NUMBER(10)
140     │   │   │   ├── KEYWORD(lakukan)
141     │   │   │   └── <compound-statement>
142     │   │   │       └── KEYWORD(mulai)
```



```
test > milestone-2 > output > program.txt
142     │   │   │       ├── KEYWORD(mulai)
143     │   │   │       ├── <statement-list>
144     │   │   │       │   └── <assignment-statement>
145     │   │   │       │       ├── IDENTIFIER(total)
146     │   │   │       │       ├── ASSIGN_OPERATOR(:=)
147     │   │   │       │       └── <expression>
148     │   │   │       │           └── <simple-expression>
149     │   │   │       │               ├── <term>
150     │   │   │       │               │   └── <factor>
151     │   │   │       │               │       └── IDENTIFIER(total)
152     │   │   │       │               └── <additive-operator>
153     │   │   │       │               └── <term>
154     │   │   │       │                   └── <factor>
155     │   │   │       │                       └── IDENTIFIER(x)
156     │   │   │       └── KEYWORD(selesai)
157     │   │   ├── SEMICOLON(;)
158     │   │   ├── <procedure/function-call>
159     │   │   │   ├── KEYWORD(writeln)
160     │   │   │   ├── LPARENTHESIS(()
161     │   │   │   ├── <parameter-list>
162     │   │   │   │   └── <expression>
163     │   │   │   │       └── <simple-expression>
164     │   │   │   │           └── <term>
165     │   │   │   │               └── <factor>
166     │   │   │   │                   └── STRING_LITERAL('Total: ')
167     │   │   │   └── RPARENTHESIS())
168     │   │   ├── SEMICOLON(;)
169     │   │   └── <procedure/function-call>
170     │   │       ├── KEYWORD(writeln)
```



```
170     │   │       ├── KEYWORD(writeln)
171     │   │       ├── LPARENTHESIS(()
172     │   │       ├── <parameter-list>
173     │   │       │   └── <expression>
174     │   │       │       └── <simple-expression>
175     │   │       │           └── <term>
176     │   │       │               └── <factor>
177     │   │       │                   └── IDENTIFIER(total)
178     │   │       └── RPARENTHESIS())
179     │   └── KEYWORD(selesai)
180     └── DOT(.)
181
```

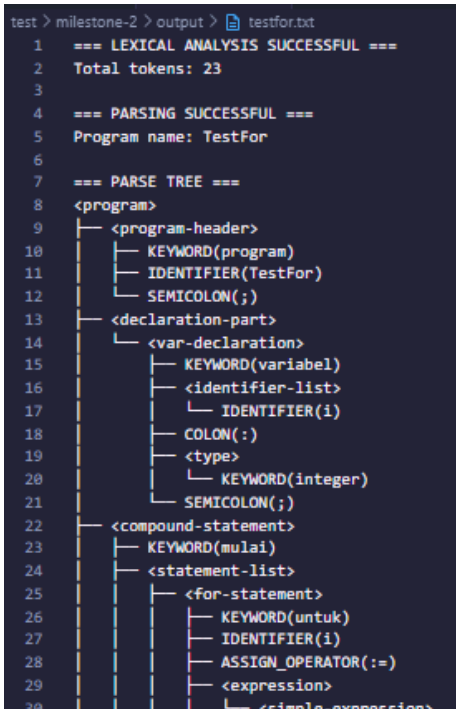
3.3. Test Case 3 - TestFor

Test case ini bertujuan untuk menguji validasi for-loop dengan direction ke (to) dan body berupa single statement (procedure call). Memastikan parser dapat menangani for-loop tanpa compound statement dan parameter passing ke procedure built-in.

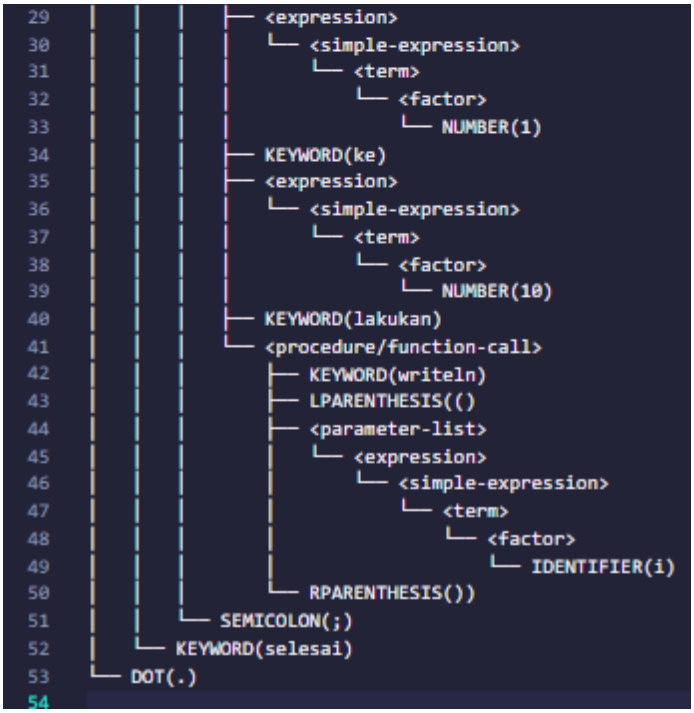
testfor.pas (Input)

```
program TestFor;
variabel i: integer;
mulai
  untuk i := 1 ke 10 lakukan
    writeln(i);
selesai.
```

Output:



```
test > milestone-2 > output > testfor.txt
1   === LEXICAL ANALYSIS SUCCESSFUL ===
2   Total tokens: 23
3
4   === PARSING SUCCESSFUL ===
5   Program name: TestFor
6
7   === PARSE TREE ===
8   <program>
9   ├── <program-header>
10  │   ├── KEYWORD(program)
11  │   ├── IDENTIFIER(TestFor)
12  │   └── SEMICOLON(;)
13  ├── <declaration-part>
14  │   └── <var-declaration>
15  │       ├── KEYWORD(variabel)
16  │       ├── <identifier-list>
17  │       │   └── IDENTIFIER(i)
18  │       ├── COLON(:)
19  │       ├── <type>
20  │       │   └── KEYWORD(integer)
21  │       └── SEMICOLON(;)
22  ├── <compound-statement>
23  │   ├── KEYWORD(mulai)
24  │   ├── <statement-list>
25  │   │   ├── <for-statement>
26  │   │   │   ├── KEYWORD(untuk)
27  │   │   │   ├── IDENTIFIER(i)
28  │   │   │   ├── ASSIGN_OPERATOR(:=)
29  │   │   │   ├── <expression>
```



```
29  │   │   │   ├── <expression>
30  │   │   │   │   └── <simple-expression>
31  │   │   │   │       └── <term>
32  │   │   │   │           └── <factor>
33  │   │   │   │               └── NUMBER(1)
34  │   │   │   ├── KEYWORD(ke)
35  │   │   │   ├── <expression>
36  │   │   │   │   └── <simple-expression>
37  │   │   │   │       └── <term>
38  │   │   │   │           └── <factor>
39  │   │   │   │               └── NUMBER(10)
40  │   │   │   ├── KEYWORD(lakukan)
41  │   │   │   └── <procedure/function-call>
42  │   │   │       ├── KEYWORD(writeln)
43  │   │   │       ├── LPARENTHESIS(()
44  │   │   │       ├── <parameter-list>
45  │   │   │       │   └── <expression>
46  │   │   │       │       └── <simple-expression>
47  │   │   │       │           └── <term>
48  │   │   │       │               └── <factor>
49  │   │   │       │                   └── IDENTIFIER(i)
50  │   │   │       └── RPARENTHESIS())
51  │   │   └── SEMICOLON(;)
52  │   └── KEYWORD(selesai)
53  └── DOT(.)
54
```

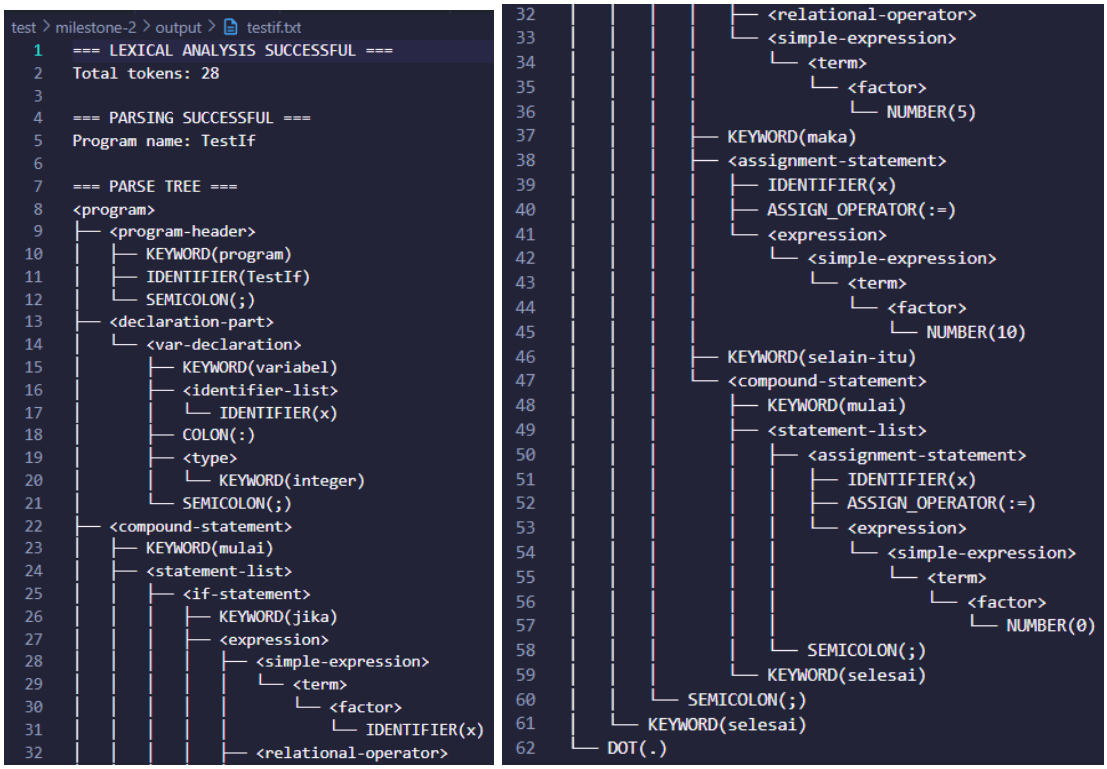
3.4. Test Case 4 - TestIf

Test case ini bertujuan untuk memvalidasi if-statement dengan kondisi relasional, then-branch berupa single statement, dan else-branch berupa compound statement. Menguji kemampuan parser mengenali keyword selain-itu (else).

testif.pas

```
program TestIf;
variabel x: integer;
mulai
  jika x > 5 maka
      x := 10;
  selain-itu
    mulai
      x := 0;
    selesai;
selesai.
```

Output:



```
test > milestone-2 > output > testif.txt
=== LEXICAL ANALYSIS SUCCESSFUL ===
Total tokens: 28

=== PARSING SUCCESSFUL ===
Program name: TestIf

=== PARSE TREE ===
<program>
├── <program-header>
│   ├── KEYWORD(program)
│   ├── IDENTIFIER(TestIf)
│   └── SEMICOLON(;)
├── <declaration-part>
│   └── <var-declaration>
│       ├── KEYWORD(variabel)
│       ├── <identifier-list>
│       │   └── IDENTIFIER(x)
│       ├── COLON(:)
│       ├── <type>
│       │   └── KEYWORD(integer)
│       └── SEMICOLON(;)
├── <compound-statement>
│   ├── KEYWORD(mulai)
│   ├── <statement-list>
│   │   ├── <if-statement>
│   │   │   ├── KEYWORD(jika)
│   │   │   ├── <expression>
│   │   │   │   ├── <simple-expression>
│   │   │   │   │   └── <term>
│   │   │   │   │       └── <factor>
│   │   │   │   │           └── IDENTIFIER(x)
│   │   │   │   ├── <relational-operator>
│   │   │   │   └── <simple-expression>
│   │   │   │       └── <term>
│   │   │   │           └── <factor>
│   │   │   │               └── NUMBER(5)
│   │   │   ├── KEYWORD(maka)
│   │   │   ├── <assignment-statement>
│   │   │   │   ├── IDENTIFIER(x)
│   │   │   │   ├── ASSIGN_OPERATOR(:=)
│   │   │   │   └── <expression>
│   │   │   │       └── <simple-expression>
│   │   │   │           └── <term>
│   │   │   │               └── <factor>
│   │   │   │                   └── NUMBER(10)
│   │   │   ├── KEYWORD(selain-itu)
│   │   │   └── <compound-statement>
│   │   │       ├── KEYWORD(mulai)
│   │   │       ├── <statement-list>
│   │   │       │   ├── <assignment-statement>
│   │   │       │   │   ├── IDENTIFIER(x)
│   │   │       │   │   ├── ASSIGN_OPERATOR(:=)
│   │   │       │   │   └── <expression>
│   │   │       │   │       └── <simple-expression>
│   │   │       │   │           └── <term>
│   │   │       │   │               └── <factor>
│   │   │       │   │                   └── NUMBER(0)
│   │   │       │   └── SEMICOLON(;)
│   │   │       └── KEYWORD(selesai)
│   │   └── SEMICOLON(;)
│   └── KEYWORD(selesai)
└── DOT(.)
```

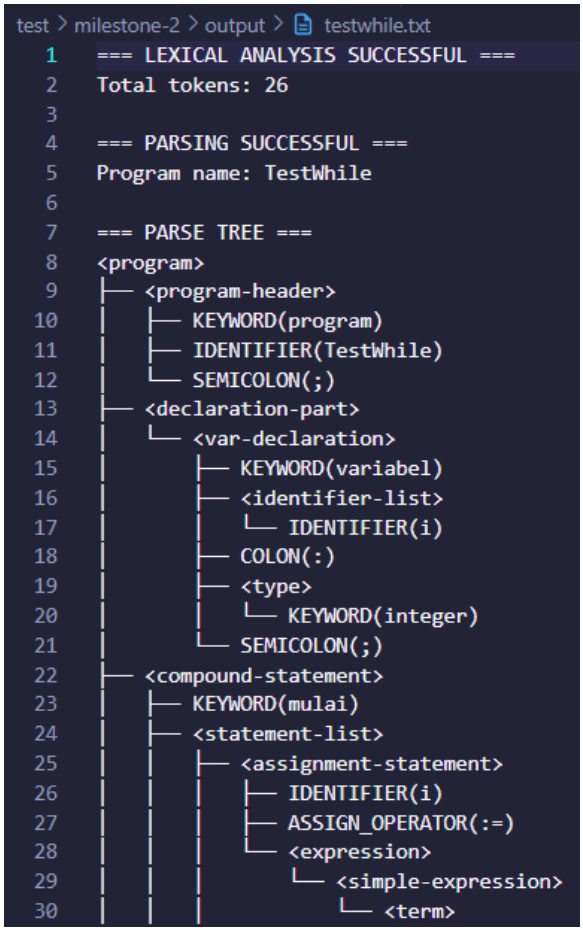
3.5. Test Case 5 - TestWhile

Test case ini bertujuan untuk memvalidasi while-loop dengan keyword selama dan lakukan, termasuk kondisi relasional dan body berupa single assignment statement dengan ekspresi aritmatika.

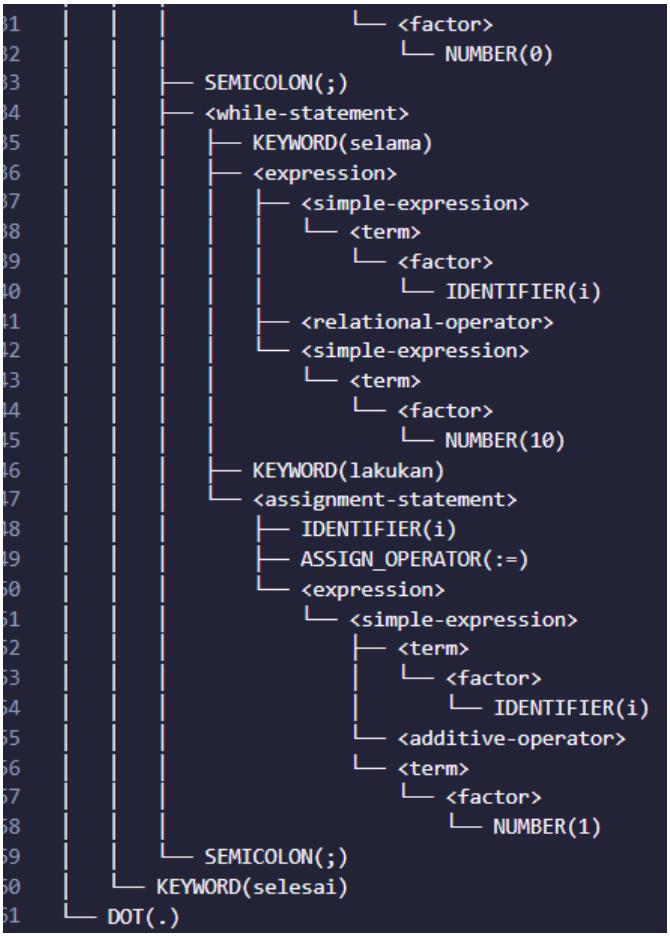
testwhile.pas

```
program TestWhile;
variabel i: integer;
mulai
  i := 0;
  selama i < 10 lakukan
    i := i + 1;
selesai.
```

Output:



```
test > milestone-2 > output > testwhile.txt
1   === LEXICAL ANALYSIS SUCCESSFUL ===
2   Total tokens: 26
3
4   === PARSING SUCCESSFUL ===
5   Program name: TestWhile
6
7   === PARSE TREE ===
8   <program>
9   ├── <program-header>
10  │   ├── KEYWORD(program)
11  │   ├── IDENTIFIER(TestWhile)
12  │   └── SEMICOLON(;)
13  ├── <declaration-part>
14  │   └── <var-declaration>
15  │       ├── KEYWORD(variabel)
16  │       ├── <identifier-list>
17  │       │   └── IDENTIFIER(i)
18  │       ├── COLON(:)
19  │       ├── <type>
20  │       │   └── KEYWORD(integer)
21  │       └── SEMICOLON(;)
22  ├── <compound-statement>
23  │   ├── KEYWORD(mulai)
24  │   ├── <statement-list>
25  │   │   ├── <assignment-statement>
26  │   │   │   ├── IDENTIFIER(i)
27  │   │   │   ├── ASSIGN_OPERATOR(:=)
28  │   │   │   └── <expression>
29  │   │   │       └── <simple-expression>
30  │   │   │           └── <term>
```



```
31  │   │   │               └── <factor>
32  │   │   │                   └── NUMBER(0)
33  │   │   ├── SEMICOLON(;)
34  │   │   ├── <while-statement>
35  │   │   │   ├── KEYWORD(selama)
36  │   │   │   ├── <expression>
37  │   │   │   │   ├── <simple-expression>
38  │   │   │   │   │   └── <term>
39  │   │   │   │   │       └── <factor>
40  │   │   │   │   │           └── IDENTIFIER(i)
41  │   │   │   │   ├── <relational-operator>
42  │   │   │   │   └── <simple-expression>
43  │   │   │   │       └── <term>
44  │   │   │   │           └── <factor>
45  │   │   │   │               └── NUMBER(10)
46  │   │   │   ├── KEYWORD(lakukan)
47  │   │   │   └── <assignment-statement>
48  │   │   │       ├── IDENTIFIER(i)
49  │   │   │       ├── ASSIGN_OPERATOR(:=)
50  │   │   │       └── <expression>
51  │   │   │           └── <simple-expression>
52  │   │   │               ├── <term>
53  │   │   │               │   └── <factor>
54  │   │   │               │       └── IDENTIFIER(i)
55  │   │   │               └── <additive-operator>
56  │   │   │               └── <term>
57  │   │   │                   └── <factor>
58  │   │   │                       └── NUMBER(1)
59  │   │   └── SEMICOLON(;)
60  │   └── KEYWORD(selesai)
61  └── DOT(.)
```

3.6. Test Case 6 - TestControlStructures

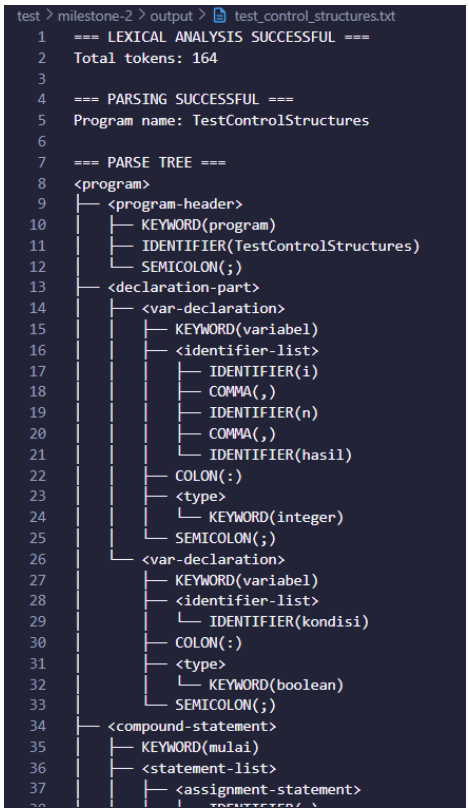
Test case ini bertujuan untuk menguji validasi semua jenis control flow seperti if-then, if-then-else, nested if, while-do, for-to, for-downto, dan kombinasi nested loop. Menguji kemampuan parser menangani multiple level nesting dan berbagai direction keyword pada for-loop.

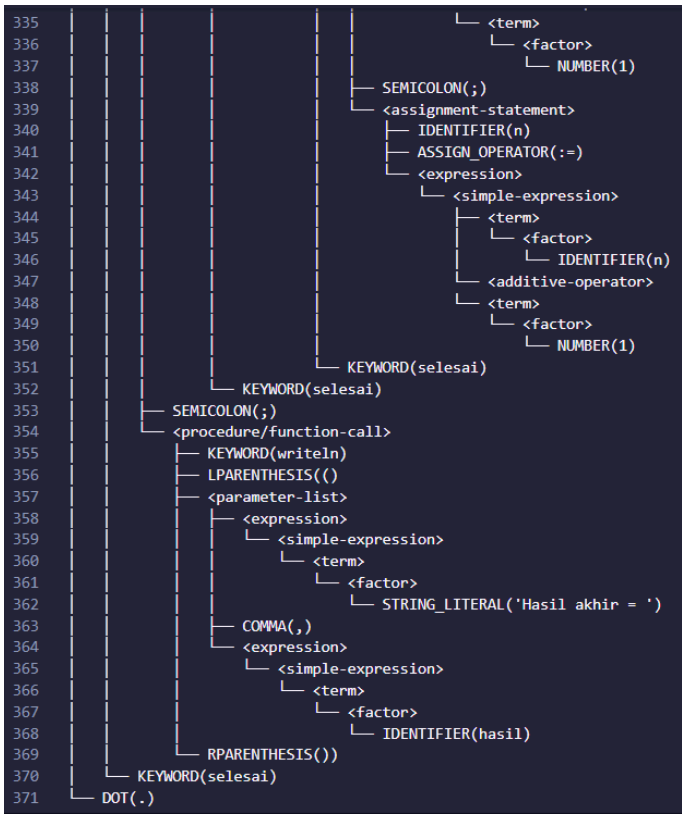
test_control_structures.pas

```
program TestControlStructures;

variabel
  i, n, hasil: integer;
  kondisi: boolean;

mulai
  n := 10;
  hasil := 0;
  jika n > 5 maka
    hasil := n * 2;
  jika n < 5 maka
    hasil := n + 10
  selain-itu
    hasil := n - 5;
  jika n > 0 maka
    jika n < 100 maka
      hasil := n
    selain-itu
      hasil := 100
  selain-itu
    hasil := 0;
  i := 1;
  selama i <= 5 lakukan
    mulai
      hasil := hasil + i;
      i := i + 1
    selesai;
  untuk i := 1 ke 10 lakukan
    hasil := hasil + i;
  untuk i := 10 turun-ke 1 lakukan
    hasil := hasil - 1;
  untuk i := 1 ke 3 lakukan
    mulai
```

```
        n := 1;
        selama n <= 2 lakukan
          mulai
            hasil := hasil + 1;
            n := n + 1
          selesai
      selesai;
    writeln('Hasil akhir = ', hasil)
  selesai.
```

Output:





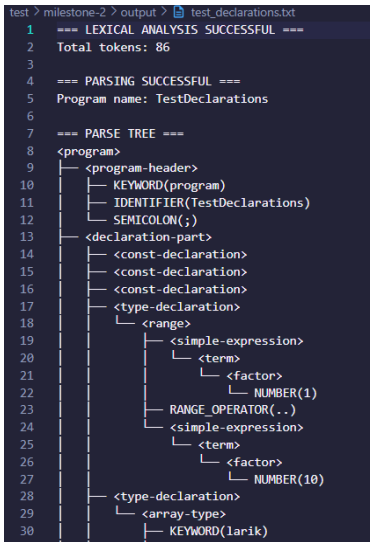
3.7. Test Case 7 - TestDeclarations

Test case ini bertujuan untuk memvalidasi kemampuan parser menangani const declaration, type declaration (subrange dan array), dan variable declaration dengan semua tipe data dasar Pascal-S: integer, real, char, boolean, custom range, dan array.

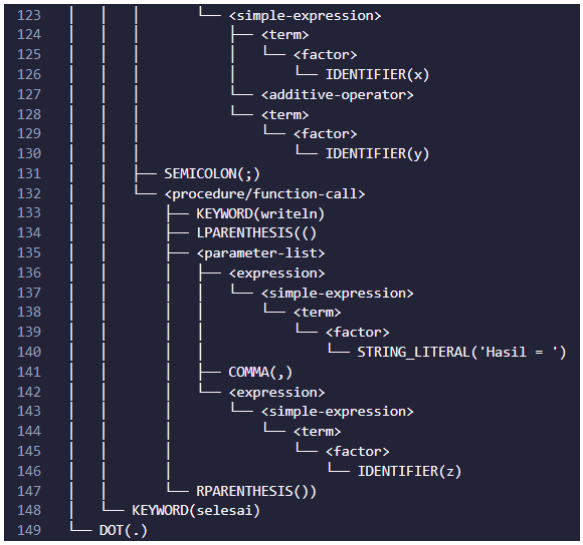
test_declarations.pas

```
program TestDeclarations;

konstanta
  MAX = 100;
  MIN = 0;
  PI = 3.14;

tipe
  Range = 1..10;
  Matrix = larik[1..5] dari integer;

variabel
  x, y, z: integer;
  nilai: real;
  huruf: char;
  valid: boolean;
  angka: Range;
  data: Matrix;

mulai
  x := 10;
  y := 20;
  z := x + y;
  writeln('Hasil = ', z)
selesai.
```

Output:



```
test > milestone-2 > output > test_declarations.txt
=== LEXICAL ANALYSIS SUCCESSFUL ===
Total tokens: 86

=== PARSING SUCCESSFUL ===
Program name: TestDeclarations

=== PARSE TREE ===
<program>
├── <program-header>
│   ├── KEYWORD(program)
│   ├── IDENTIFIER(TestDeclarations)
│   └── SEMICOLON(;)
├── <declaration-part>
│   ├── <const-declaration>
│   ├── <const-declaration>
│   ├── <const-declaration>
│   ├── <type-declaration>
│   │   └── <range>
│   │       ├── <simple-expression>
│   │       │   └── <term>
│   │       │       └── <factor>
│   │       │           └── NUMBER(1)
│   │       ├── RANGE_OPERATOR(..)
│   │       └── <simple-expression>
│   │           └── <term>
│   │               └── <factor>
│   │                   └── NUMBER(10)
│   ├── <type-declaration>
│   │   └── <array-type>
│   │       ├── KEYWORD(larik)
```



```
│   │   │       └── <simple-expression>
│   │   │           ├── <term>
│   │   │           │   └── <factor>
│   │   │           │       └── IDENTIFIER(x)
│   │   │           └── <additive-operator>
│   │   │           └── <term>
│   │   │               └── <factor>
│   │   │                   └── IDENTIFIER(y)
│   │   ├── SEMICOLON(;)
│   │   └── <procedure/function-call>
│   │       ├── KEYWORD(writeln)
│   │       ├── LPARENTHESIS(()
│   │       ├── <parameter-list>
│   │       │   ├── <expression>
│   │       │   │   └── <simple-expression>
│   │       │   │       └── <term>
│   │       │   │           └── <factor>
│   │       │   │               └── STRING_LITERAL('Hasil = ')
│   │       │   ├── COMMA(,)
│   │       │   └── <expression>
│   │       │       └── <simple-expression>
│   │       │           └── <term>
│   │       │               └── <factor>
│   │       │                   └── IDENTIFIER(z)
│   │       └── RPARENTHESIS())
│   └── KEYWORD(selesai)
└── DOT(.)
```

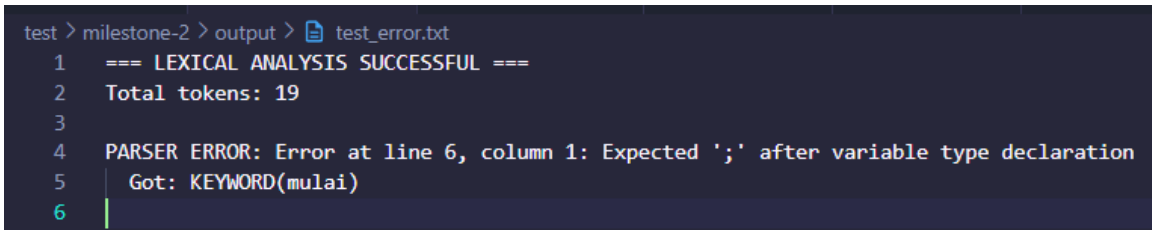
3.8. Test Case 8 - TestError

Test case ini sengaja mengandung error sintaks (kurang semicolons) untuk menguji validasi bahwa parser dapat mendeteksi kesalahan dan memberikan error message.

test_declarations.pas

```
program TestError;

variabel
  x, y: integer

mulai
  x := 5;
  y := 10
selesai.
```

Output:



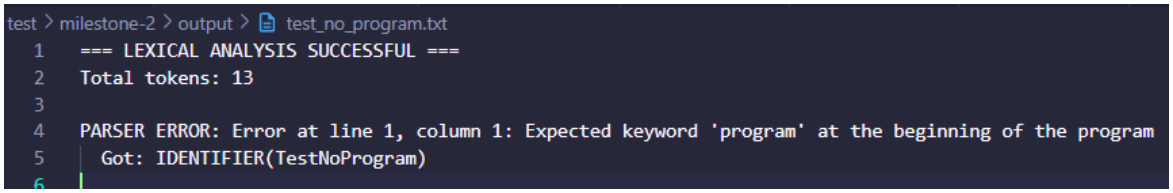
3.9. Test Case 9 - TestNoProgram

Test case ini sengaja tidak menggunakan keyword program di awal untuk memvalidasi bahwa parser dapat mendeteksi error pada program header dan memberikan pesan error yang sesuai.

test_no_program.pas

```
TestNoProgram;

variabel
  x: integer;

mulai
  x := 5
selesai.
```

Output:



3.10. Test Case 10 - TestProceduresAndFunctions

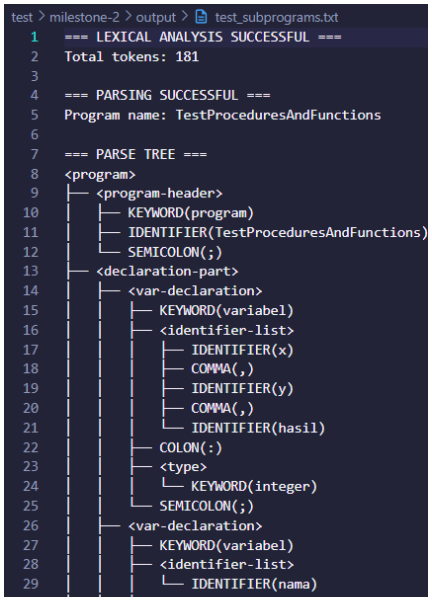
Test case ini fokus pada menguji subprogram declaration yaitu procedure dengan/tanpa parameter, function dengan return type, formal parameter lists, dan pemanggilan subprogram dengan actual parameters. Juga menguji nested function call sebagai parameter dan function assignment.

test_subprograms.pas

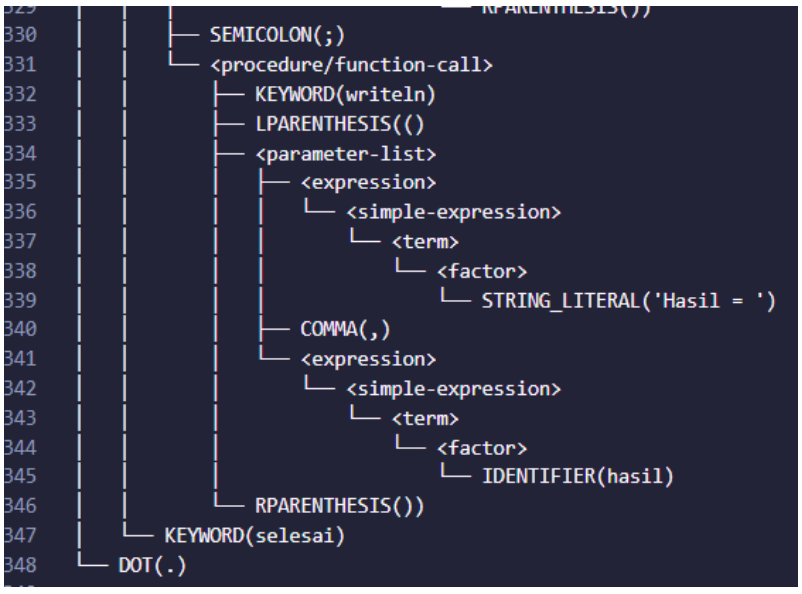
```
program TestProceduresAndFunctions;

variabel
  x, y, hasil: integer;
  nama: char;

prosedur PrintHello(msg: char);
variabel
  temp: integer;
mulai
  temp := 0;
  writeln(msg)
selesai;

prosedur Swap(a, b: integer);
variabel
  temp: integer;
mulai
  temp := a;
  a := b;
  b := temp
selesai;

fungsi Tambah(a, b: integer): integer;
variabel
```

```
  result: integer;
mulai
  result := a + b;
  Tambah := result
selesai;

fungsi Kali(x: integer; y: integer): integer;
mulai
  Kali := x * y
selesai;

mulai
  x := 10;
  y := 20;

  { Call procedures }
  PrintHello('A');
  Swap(x, y);

  { Call functions }
  hasil := Tambah(x, y);
  hasil := Kali(5, 6);
  hasil := Tambah(Kali(2, 3), 10);

  writeln('Hasil = ', hasil)
selesai.
```

Output:



```
test > milestone-2 > output > test_subprograms.txt
1  === LEXICAL ANALYSIS SUCCESSFUL ===
2  Total tokens: 181
3
4  === PARSING SUCCESSFUL ===
5  Program name: TestProceduresAndFunctions
6
7  === PARSE TREE ===
8  <program>
9  ├── <program-header>
10 │   ├── KEYWORD(program)
11 │   ├── IDENTIFIER(TestProceduresAndFunctions)
12 │   └── SEMICOLON(;)
13 ├── <declaration-part>
14 │   ├── <var-declaration>
15 │   │   ├── KEYWORD(variabel)
16 │   │   ├── <identifier-list>
17 │   │   │   ├── IDENTIFIER(x)
18 │   │   │   ├── COMMA(,)
19 │   │   │   ├── IDENTIFIER(y)
20 │   │   │   ├── COMMA(,)
21 │   │   │   └── IDENTIFIER(hasil)
22 │   │   ├── COLON(:)
23 │   │   ├── <type>
24 │   │   │   └── KEYWORD(integer)
25 │   │   └── SEMICOLON(;)
26 │   ├── <var-declaration>
27 │   │   ├── KEYWORD(variabel)
28 │   │   ├── <identifier-list>
29 │   │   │   └── IDENTIFIER(nama)
```



```
330 │   │   ├── SEMICOLON(;)
331 │   │   └── <procedure/function-call>
332 │   │       ├── KEYWORD(writeln)
333 │   │       ├── LPARENTHESIS(()
334 │   │       ├── <parameter-list>
335 │   │       │   ├── <expression>
336 │   │       │   │   └── <simple-expression>
337 │   │       │   │       └── <term>
338 │   │       │   │           └── <factor>
339 │   │       │   │               └── STRING_LITERAL('Hasil = ')
340 │   │       │   ├── COMMA(,)
341 │   │       │   └── <expression>
342 │   │       │       └── <simple-expression>
343 │   │       │           └── <term>
344 │   │       │               └── <factor>
345 │   │       │                   └── IDENTIFIER(hasil)
346 │   │       └── RPARENTHESIS())
347 │   └── KEYWORD(selesai)
348 └── DOT(.)
```

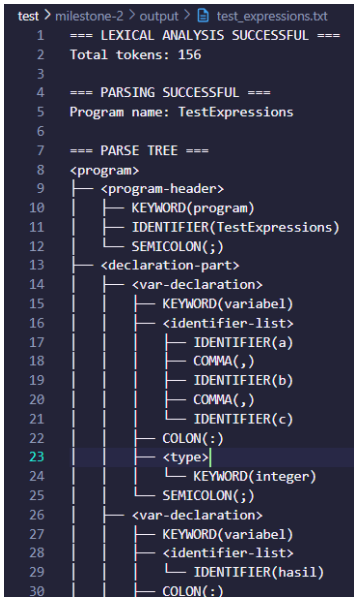
3.11. Test Case 11 - TestExpressions

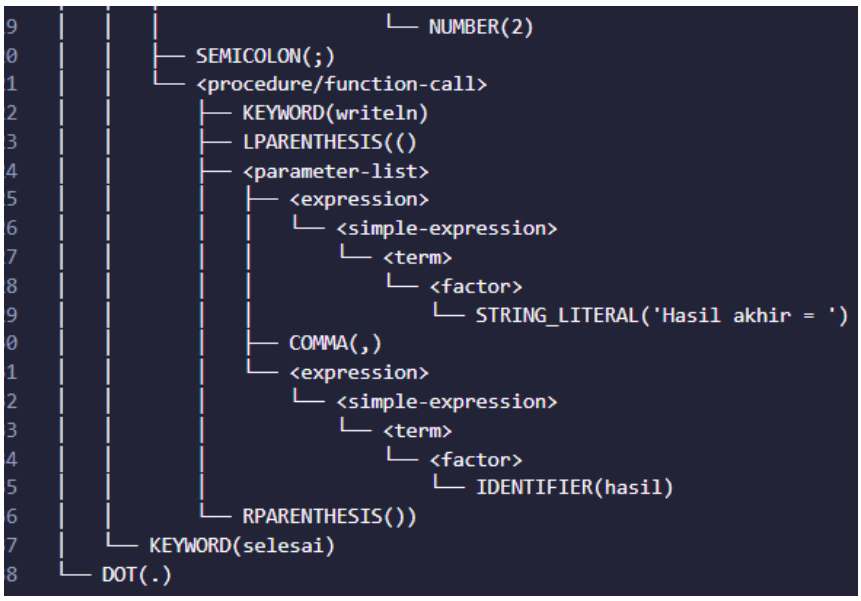
Test case ini bertujuan untuk menguji validasi expression parsing dengan semua operator: aritmatika (+, -, *, bagi, mod), relasional (=, <>, <, <=, >, >=), dan logika (dan, atau, tidak). Menguji operator precedence dan associativity, serta ekspresi parenthesized.

test_expressions.pas

```
program TestExpressions;

variabel
  a, b, c: integer;
  hasil: integer;
  benar: boolean;

mulai
  a := 5;
  b := 10;
  c := a + b;
  c := a - b;
  c := a * b;
  c := a bagi b;
  c := a mod b;
  benar := a = b;
  benar := a <> b;
  benar := a < b;
  benar := a <= b;
  benar := a > b;
  benar := a >= b;
  benar := (a < b) dan (b < 20);
  benar := (a > b) atau (b > 5);
  benar := tidak (a = b);
  hasil := (a + b) * (c - 5) bagi 2;
  writeln('Hasil akhir = ', hasil)
selesai.
```

Output:





3.12. Test Case 12 - TestComprehensive

Test case bertujuan untuk mengintegrasikan hampir semua fitur Pascal-S seperti konstanta, tipe custom, variabel, fungsi, prosedur, semua jenis control flow, array indexing, nested loops, operator logika (dan, atau, tidak), dan boolean literals (benar, salah).

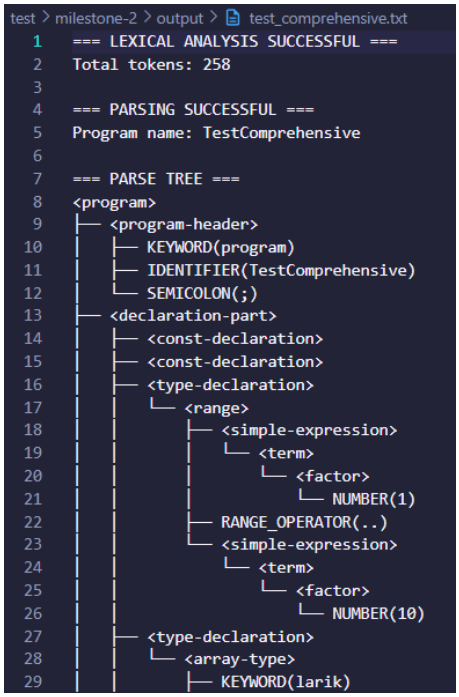
test_comprehensive.pas

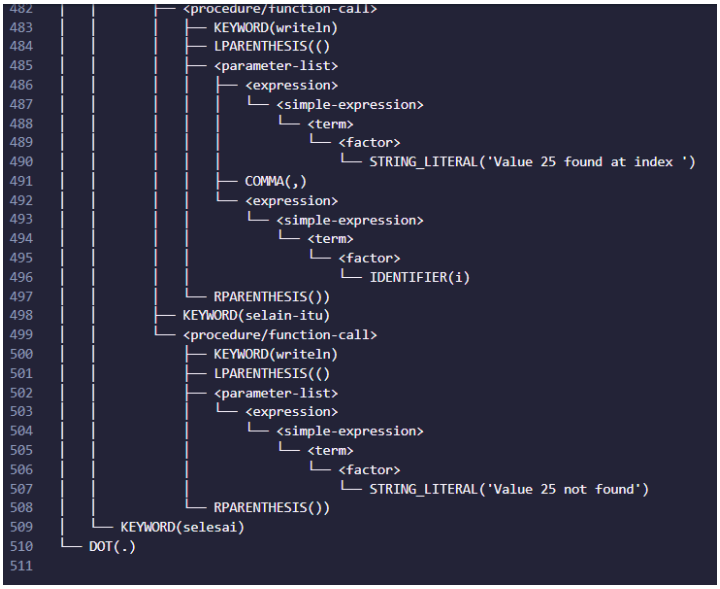
```
program TestComprehensive;

konstanta
  MAX_SIZE = 100;
  VERSION = 2;

tipe
  IndexRange = 1..10;
  DataArray = larik[1..10] dari integer;

variabel
  arr: DataArray;
  i, sum, avg: integer;
  found: boolean;

fungsi FindMax(data: DataArray; size: integer): integer;
```

```
variabel
  max, idx: integer;
mulai
  max := data;
  untuk idx := 2 ke size lakukan
    jika data > max maka
      max := data;
  FindMax := max
selesai;

prosedur PrintArray(data: DataArray; count: integer);
variabel
  j: integer;
mulai
  untuk j := 1 ke count lakukan
    write(data, ' ')
selesai;

mulai
  { Initialize array }
  sum := 0;
  untuk i := 1 ke 10 lakukan
    mulai
      arr := i * 5;
      sum := sum + arr
    selesai;

  { Calculate average }
  avg := sum bagi 10;

  { Find maximum }
  i := FindMax(arr, 10);

  { Print results }
  writeln('Array values:');
  PrintArray(arr, 10);
  writeln();
  writeln('Sum = ', sum);
  writeln('Average = ', avg);
  writeln('Maximum = ', i);

  { Search for value }
  found := salah;
  i := 1;
  selama (i <= 10) dan (tidak found) lakukan
    mulai
```

```
        jika arr = 25 maka
          found := benar
        selain-itu
          i := i + 1
      selesai;

    jika found maka
      writeln('Value 25 found at index ', i)
    selain-itu
      writeln('Value 25 not found')
  selesai.
```

Output:





BAB IV

KESIMPULAN DAN SARAN

4.1. Kesimpulan

Berdasarkan pengerjaan Milestone 2 yang telah kami lakukan, kami telah berhasil membuat sebuah penganalisis sintaks (parser) untuk bahasa Pascal-S. Parser ini berfungsi sebagai fase kedua kompilasi yang bertugas memeriksa apakah urutan token (hasil dari Milestone 1) sesuai dengan tata bahasa (grammar) yang didefinisikan, serta membangun representasi struktur program dalam bentuk Parse Tree atau Abstract Syntax Tree (AST). Implementasi ini menggunakan metode *Recursive Descent Parsing*, sebuah pendekatan *top-down*, untuk memvalidasi struktur kode berdasarkan aturan *Context-Free Grammar* (CFG) dari Pascal-S yang telah diadaptasi ke Bahasa Indonesia (misalnya, "mulai", "selesai", "jika").

Parser yang telah dibuat mampu mengenali dan memvalidasi seluruh struktur gramatikal program. Ini mencakup bagian utama seperti program-header, declaration-part (yang menangani konstanta, tipe, variabel, prosedur, dan fungsi), dan compound-statement. Selain itu, parser ini juga berhasil menguraikan semua statement kontrol alur (seperti jika-maka-selain-itu, selama-lakukan, dan untuk-ke/turun-ke) serta menangani ekspresi kompleks (aritmatika, relasional, dan logika) dengan memperhatikan urutan operasi (*operator precedence*).

Pengujian dilakukan menggunakan dua belas kasus uji yang beragam. Kasus uji ini mencakup validasi program minimal, deklarasi tipe yang kompleks, berbagai struktur kontrol, subprogram, ekspresi, dan pengujian komprehensif. Pengujian juga secara khusus memvalidasi kemampuan error handling untuk sintaks yang salah. Hasil pengujian menunjukkan bahwa program berhasil mem-parsing semua kasus uji yang valid dan mampu mendeteksi serta melaporkan kesalahan sintaks dengan benar. Dengan demikian, tahap analisis sintaks ini dapat dikatakan telah berhasil diimplementasikan sesuai dengan spesifikasi yang telah diberikan.

4.2. Saran

Tidak ada saran yang bisa kami berikan namun kami ingin mengapresiasi diri kami sendiri karena telah berhasil menyelesaikan milestone 2 tubes TBFO ini.

LAMPIRAN

Grammar yang digunakan.

Program Structure
program → program_header declaration_part compound_statement DOT
program_header → PROGRAM IDENTIFIER SEMICOLON

Declarations
declaration_part → [const_block] [type_block] [var_block]
subprogram_declaration*
const_block → KONSTANTA const_declaration+
const_declaration → IDENTIFIER EQUALS constant_value SEMICOLON
constant_value → NUMBER | STRING_LITERAL | CHAR_LITERAL | BOOLEAN
type_block → TIPE type_declaration+
type_declaration → IDENTIFIER EQUALS type_definition SEMICOLON
type_definition → array_type | simple_type | IDENTIFIER
var_block → VARIABEL variable_declaration+
variable_declaration → identifier_list COLON type SEMICOLON

Types
type → INTEGER | REAL | BOOLEAN | CHAR | array_type | IDENTIFIER
array_type → LARIK LBRACKET range RBRACKET DARI type
range → simple_expression RANGE_OPERATOR simple_expression
identifier_list → IDENTIFIER (COMMA IDENTIFIER)*

Subprograms
subprogram_declaration → procedure_declaration | function_declaration
procedure_declaration → PROSEDUR IDENTIFIER [formal_parameter_list]
SEMICOLON block SEMICOLON
function_declaration → FUNGSI IDENTIFIER [formal_parameter_list] COLON type
SEMICOLON block SEMICOLON
block → declaration_part compound_statement
formal_parameter_list → LPARENTHESIS [parameter_group (SEMICOLON
parameter_group)*] RPARENTHESIS
parameter_group → identifier_list COLON type

Statements
compound_statement → MULAI statement_list SELESAI
statement_list → [statement (SEMICOLON statement)*]
statement → assignment_statement
| procedure_call
| compound_statement
| if_statement

| while_statement
| for_statement
| empty
assignment_statement → IDENTIFIER ASSIGN_OPERATOR expression
procedure_call → (IDENTIFIER | builtin_procedure) [LPARENTHESIS [parameter_list] RPARENTHESIS]
builtin_procedure → WRITELN | WRITE | READLN | READ
if_statement → JIKA expression MAKA statement [SELAIN-ITU statement]
while_statement → SELAMA expression LAKUKAN statement
for_statement → UNTUK IDENTIFIER ASSIGN_OPERATOR expression (KE | TURUN-KE) expression LAKUKAN statement

Expressions
expression → simple_expression [relational_operator simple_expression]
relational_operator → EQUALS | NOT_EQUALS | LESS_THAN | LESS_EQUAL | GREATER_THAN | GREATER_EQUAL
simple_expression → [sign] term (additive_operator term)*
sign → PLUS | MINUS
additive_operator → PLUS | MINUS | ATAU
term → factor (multiplicative_operator factor)*
multiplicative_operator → MULTIPLY | DIVIDE | BAGI | MOD | DAN
factor → TIDAK factor
| LPARENTHESIS expression RPARENTHESIS
| NUMBER
| CHAR_LITERAL
| STRING_LITERAL
| IDENTIFIER [LPARENTHESIS parameter_list RPARENTHESIS]
| boolean_literal

boolean_literal → BENAR | SALAH | TRUE | FALSE
parameter_list → expression (COMMA expression)*

Terminal Symbols (Keywords dalam Bahasa Indonesia)
PROGRAM → "program"
KONSTANTA → "konstanta"
TIPE → "tipe"
VARIABEL → "variabel"
PROSEDUR → "prosedur"
FUNGSI → "fungsi"
MULAI → "mulai"
SELESAI → "selesai"
JIKA → "jika"
MAKA → "maka"
SELAIN-ITU → "selain-itu"

SELAMA → "selama"
UNTUK → "untuk"
KE → "ke"
TURUN-KE → "turun-ke"
LAKUKAN → "lakukan"
LARIK → "larik"
DARI → "dari"
INTEGER → "integer"
REAL → "real"
BOOLEAN → "boolean"
CHAR → "char"
BENAR → "benar"
SALAH → "salah"
DAN → "dan"
ATAU → "atau"
TIDAK → "tidak"
BAGI → "bagi"
MOD → "mod"
WRITELN → "writeln"
WRITE → "write"
READLN → "readln"
READ → "read"

Operators & Punctuation
ASSIGN_OPERATOR → ":="
RANGE_OPERATOR → ".."
EQUALS → "="
NOT_EQUALS → "<>"
LESS_THAN → "<"
LESS_EQUAL → "<="
GREATER_THAN → ">"
GREATER_EQUAL → ">="
PLUS → "+"
MINUS → "-"
MULTIPLY → "*"
DIVIDE → "/"
DOT → "."
COMMA → ","
SEMICOLON → ";"
COLON → ":"
LPARENTHESIS → "("
RPARENTHESIS → ")"
LBRACKET → "["
RBRACKET → "]"

Link Release Repository Github.

https://github.com/ahsuunn/NTB-Tubes-IF2224/releases/tag/v0.2.1

Pembagian Tugas (menunjukan persentase kontribusi)

NIM	Persentase Kerja	Tugas
13523018	25%	Mengimplementasikan nonterminal parser, menjamin alur utama berjalan sesuai grammar Pascal-S, dan membuat Bab 4 dan daftar grammar pada laporan
13523062	25%	Mengimplementasikan parser, translasi Bahasa Indonesia ke Bahasa Inggris, membuat penjelasan implementasi pada laporan
13523074	25%	Mengimplementasikan detailed parser error message, pars const, type, array, procedure, function, params list, formal params list, params group, procedure block, test parser lanjutan, dan membuat latar belakang pada laporan
13523120	25%	Mengimplementasikan parse statement, parse assignment, while, if, parse procedure call, pada parser, melakukan pengujian, dan implementasi ast node