Milestone 2 IF2224 Teori Bahasa Formal dan Otomata
Syntax 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 tahap *lexical analysis* selesai, source code telah diubah menjadi deretan token yang merepresentasikan unit-unit dasar dari bahasa pemrograman, seperti identifier, keyword, operator, dan simbol khusus. Token-token inilah yang menjadi masukan utama untuk tahap berikutnya dalam proses kompilasi, yaitu *Syntax Analysis* atau *parsing*.

Definisi dan Peran Parser

Parser adalah komponen pada compiler atau interpreter yang bertugas memeriksa urutan token dari lexer dan menentukan apakah token-token tersebut membentuk struktur sintaksis yang valid sesuai dengan aturan tata bahasa (*context-free grammar*) dari bahasa pemrograman. Dengan kata lain, parser melakukan proses untuk:

1. Memverifikasi struktur sintaks berdasarkan grammar.
2. Mendeteksi dan melaporkan kesalahan sintaks, seperti penggunaan simbol yang tidak sesuai atau struktur perintah yang tidak lengkap.
3. Membangun representasi sintaksis, umumnya berupa *parse tree* atau *abstract syntax tree (AST)*, yang nantinya akan digunakan dalam tahap analisis semantik maupun proses penerjemahan berikutnya.

Parser merupakan penghubung penting antara token mentah yang dihasilkan oleh lexer dan representasi semantik yang lebih tinggi yang diperlukan compiler.

Keterkaitan antara Lexer dan Parser

Lexer dan parser bekerja secara berurutan namun saling bergantung:

- Lexer memecah input berupa karakter menjadi token-token yang lebih bermakna.

- Parser menggunakan token-token tersebut untuk menilai apakah program mengikuti aturan grammar.

Hubungan keduanya dapat disederhanakan sebagai berikut:

```
Source Code → Lexer → Token Stream → Parser → Parse Tree / AST
```

Lexer menangani detail-level rendah seperti pengenalan kata kunci, angka, atau operator, sedangkan parser menangani struktur-level tinggi seperti ekspresi, pernyataan kondisi, deklarasi, atau blok program. Jika lexer salah mengelompokkan token, parser juga tidak akan mampu membentuk struktur sintaks yang benar.

Algoritma Recursive Descent Parsing

Salah satu teknik parsing yang paling umum dan mudah diimplementasikan adalah Recursive Descent Parsing. Ini adalah metode *top-down parsing* yang menyusun parse tree dengan memulai dari simbol awal grammar dan mencoba menurunkannya (*derive*) sesuai aturan produksi grammar.

Ciri-ciri utama *recursive descent parser*:

1. Setiap non-terminal dalam grammar diimplementasikan sebagai satu fungsi.
   Fungsi tersebut akan memanggil fungsi lain sesuai aturan produksi.
2. Parser bekerja secara rekursif.
   Struktur grammar yang bersarang diterjemahkan menjadi pemanggilan fungsi yang juga bersarang.
3. Mudah diimplementasikan, terutama untuk grammar yang bebas dari *left recursion*.
4. Membaca input secara urut, memproses token dari awal hingga akhir sambil membangun struktur sintaks.

Setiap fungsi mencocokkan token dan memanggil fungsi lain untuk membentuk pohon sintaks dari grammar tersebut.

Parse Tree

Parse tree adalah representasi hierarkis yang menunjukkan bagaimana token dari lexer disusun untuk membentuk struktur program yang sesuai dengan grammar. Dalam parse tree:

- Node internal mewakili simbol non-terminal (misalnya: Expr, Statement, Program).
- Node daun mewakili simbol terminal atau token nyata (misalnya: identifier, angka, operator, tanda baca).

Parse tree menampilkan secara lengkap seluruh langkah produksi grammar yang digunakan untuk membentuk input, sehingga pohon ini cenderung lebih detail. Parse tree berbeda dari *Abstract Syntax Tree (AST)* — parse tree menyimpan seluruh elemen grammar, sedangkan AST biasanya menyimpan informasi yang lebih ringkas dan relevan untuk proses semantik.

Contoh parse tree (diadaptasi dari berbagai referensi seperti Ruslan Spivak dan Arpeggio) menunjukkan bagaimana rangkaian token “3 + 5 * 2” diuraikan menjadi struktur hierarkis sesuai aturan grammar.

Perancangan

Struktur Program

Struktur program terdiri dari file-file yang digunakan untuk lexical analysis dengan tambahan parser.rs sebagai file yang menangani parser

Struktur program

```
CGK-TUBES-IF2224
├── Cargo.lock
├── Cargo.toml
├── dfa_rules.json
├── doc
├── LICENSE
├── README.md
├── src
│   ├── dfa.rs
│   ├── lexer.rs
│   ├── main.rs
│   ├── node.rs
│   ├── parser.rs
│   └── token.rs
├── test
│   └── milestone1
│   │   ├── input-1.pas
│   │   ├── input-2.pas
│   │   ├── input-3.pas
│   │   ├── input-4.pas
│   │   └── input-5.pas
│   └── milestone2
│       ├── input-1.pas
│       ├── input-2.pas
│       ├── input-3.pas
│       ├── input-4.pas
│       └── input-5.pas
└── test.pas
```

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 parser.

Nama	Tipe	Deskripsi	Justifikasi
Parser	struct	Struct ini menyimpan state parsing saat ini (daftar token dan indeks posisi current). Ia bertanggung jawab	Mengenkapsulasi state parsing dalam satu objek memungkinkan kontrol penuh atas aliran token dan memudahkan

		untuk menjalankan algoritma Recursive Descent.	pengelolaan lifecycle parsing.
ParseNode	struct	Node pembentuk Parse Tree. Setiap node memiliki tipe (NodeType) dan daftar anak (children).	Struktur rekursif (Vec<ParseNode>) dipilih karena Parse Tree memiliki kedalaman yang tidak diketahui dan dinamis
NodeType	enum	Mendefinisikan semua kemungkinan jenis node dalam Parse Tree, baik itu Non-Terminal (aturan grammar seperti <program>, <expression>) maupun Terminal (leaf).	Penggunaan enum di Rust menjamin type safety dan memastikan kita hanya menggunakan label yang valid sesuai spesifikasi grammar.
ParseError	struct	Error handling, objek khusus untuk melaporkan kesalahan sintaks. Menyimpan pesan error informatif dan token penyebab kesalahan.	Memisahkan tipe error memungkinkan pelaporan kesalahan dengan informasi lengkap (lokasi dan konteks) tanpa mengganggu logika utama parser.
parse()	fn	Metode publik utama pada Parser yang memulai proses analisis dari aturan grammar teratas (<program>) dan memastikan seluruh input terkonsumsi.	Menyediakan antarmuka publik yang bersih bagi main.rs untuk berinteraksi dengan parser tanpa perlu mengetahui detail internalnya.
consume()	fn	Helper method yang memverifikasi apakah token saat ini sesuai dengan yang diharapkan (wajib). Jika sesuai, ia menggeser posisi maju; jika tidak, ia mengembalikan ParseError.	Mengimplementasikan logika match-and-advance secara terpusat mengurangi duplikasi kode dan menstandardisasi cara penanganan token terminal.
peek() / check()	fn	Lookahead (LL(1)). Metode untuk melihat token saat ini tanpa mengonsumsinya. Digunakan untuk mengambil keputusan percabangan dalam	Parser LL(1) membutuhkan kemampuan untuk melihat satu token ke depan (lookahead) guna menentukan aturan produksi mana yang harus dipilih.

		grammar.	

Alur Kerja Program

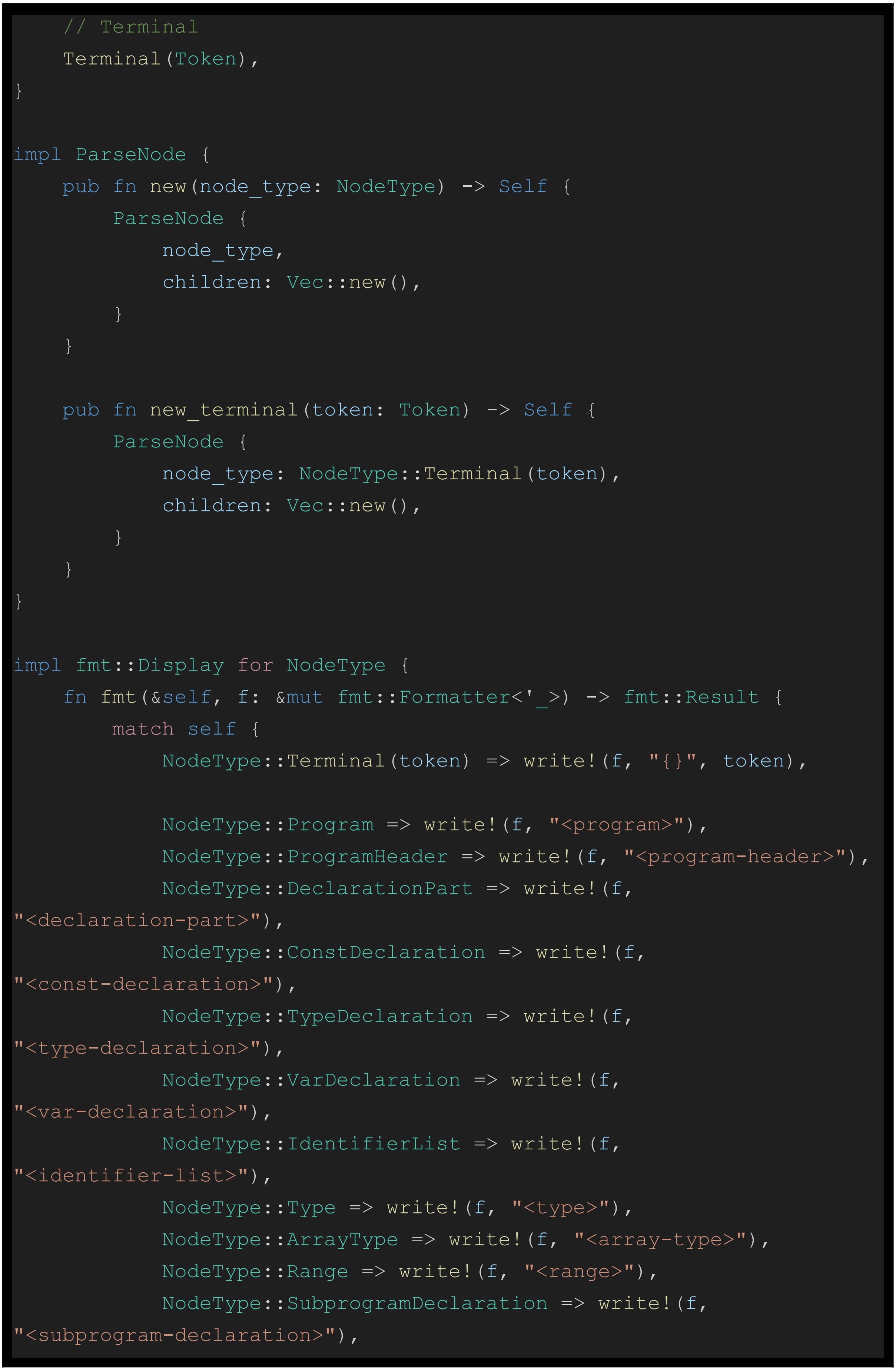
Alur kerja program mengikut konsep Top-Down Parsing menggunakan algoritma Recursive Descent. Berikut adalah tahapan eksekusi program:

1. Program main.rs membaca file .pas, menjalankan Lexer (Milestone 1), dan menghasilkan vektor token (Vec<Token>)
2. Objek Parser dibuat dengan menerima kepemilikan (ownership) dari vektor token tersebut. Penunjuk posisi (current) diatur ke indeks 0.
3. Fungsi parse() dipanggil, yang kemudian memanggil fungsi grammar root yaitu parse_program().
4. Setiap fungsi parse_X (misalnya parse_program_header, parse_expression) merepresentasikan satu aturan Non-Terminal pada grammar. Fungsi ini akan membuat ParseNode baru sebagai simpul induk. Fungsi kemudian memanggil fungsi parsing lain atau mengonsumsi token terminal sesuai urutan aturan grammar.
5. Setelah aturan <program> selesai diproses, parser memeriksa apakah masih ada token tersisa. Jika ada, error dilemparkan karena input tidak valid sepenuhnya.
6. Jika sukses, Program mengembalikan Ok(ParseNode) yang merupakan root dari Parse Tree. Pohon ini kemudian dicetak ke terminal dan file menggunakan implementasi fmt::Display yang diformat dengan indentasi rekursif.
7. Jika di tengah proses ditemukan ketidaksesuaian token (misalnya mengharapkan ; tapi menemukan keyword), parser mengembalikan Err(ParseError) dan proses berhenti (fail-fast). Pesan error ditampilkan ke pengguna.

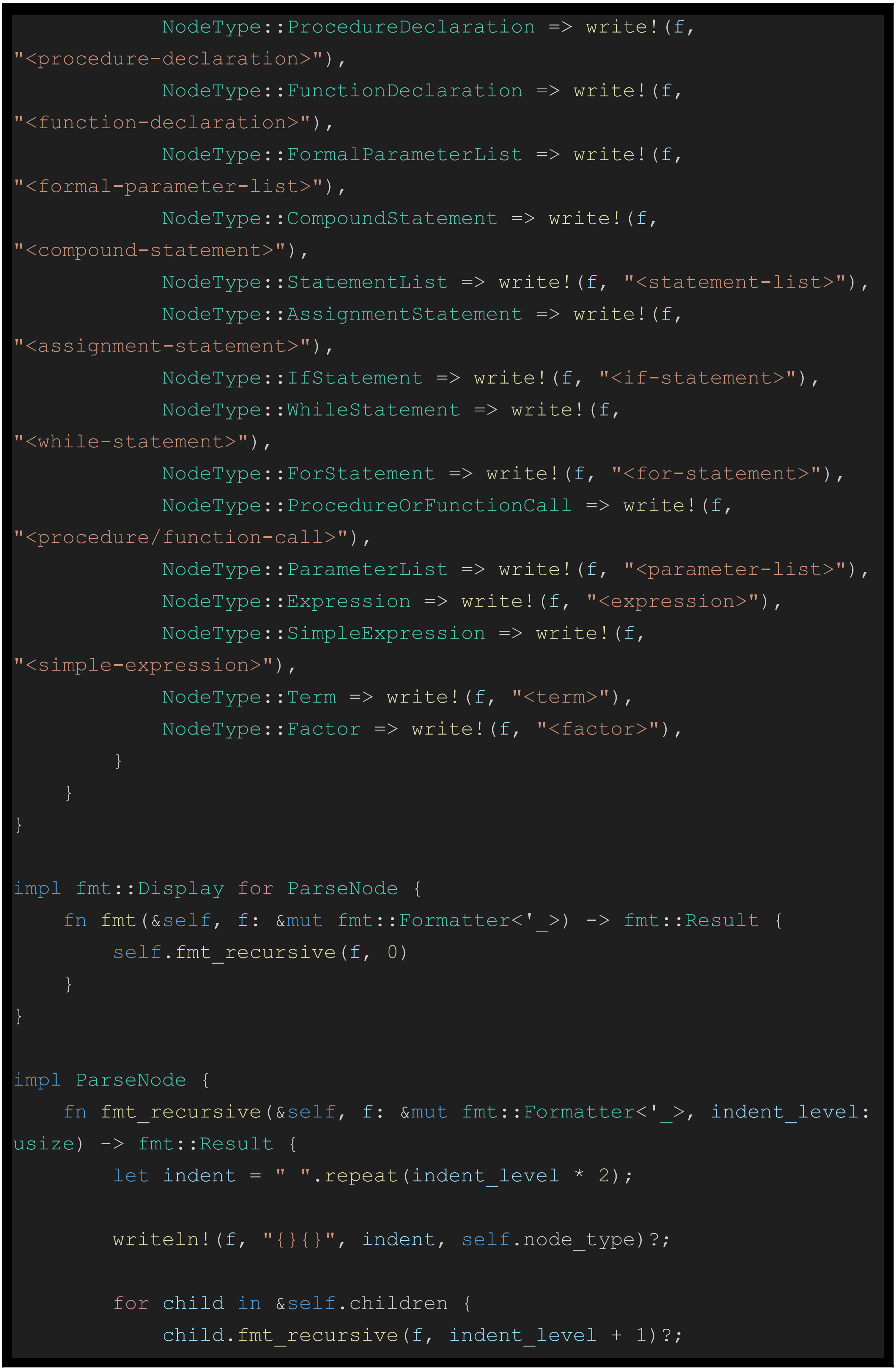
Implementasi

node.rs

```rust
use crate::token::Token;
use std::fmt;

#[derive(Debug)]
pub struct ParseNode {
    pub node_type: NodeType,
    pub children: Vec<ParseNode>,
}

#[derive(Debug)]
pub enum NodeType {
    // Non-Terminal Grammar Rules
    Program,
    ProgramHeader,
    DeclarationPart,
    ConstDeclaration,
    TypeDeclaration,
    VarDeclaration,
    IdentifierList,
    Type,
    ArrayType,
    Range,
    SubprogramDeclaration,
    ProcedureDeclaration,
    FunctionDeclaration,
    FormalParameterList,
    CompoundStatement,
    StatementList,
    AssignmentStatement,
    IfStatement,
    WhileStatement,
    ForStatement,
    ProcedureOrFunctionCall,
    ParameterList,
    Expression,
    SimpleExpression,
    Term,
    Factor,
```



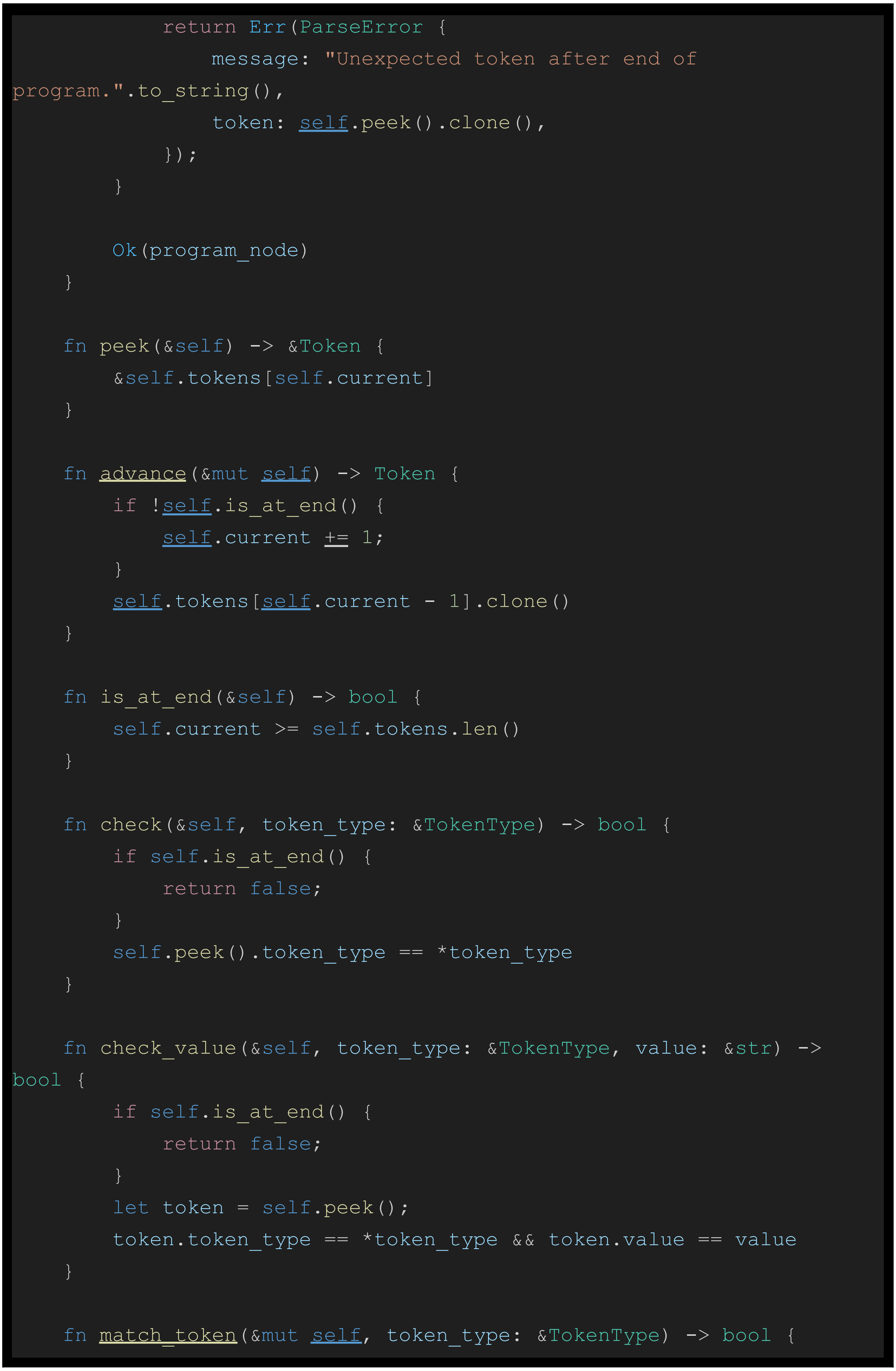
```
    // Terminal
    Terminal(Token),
}

impl ParseNode {
    pub fn new(node_type: NodeType) -> Self {
        ParseNode {
            node_type,
            children: Vec::new(),
        }
    }

    pub fn new_terminal(token: Token) -> Self {
        ParseNode {
            node_type: NodeType::Terminal(token),
            children: Vec::new(),
        }
    }
}

impl fmt::Display for NodeType {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        match self {
            NodeType::Terminal(token) => write!(f, "{}", token),

            NodeType::Program => write!(f, "<program>"),
            NodeType::ProgramHeader => write!(f, "<program-header>"),
            NodeType::DeclarationPart => write!(f,
"<declaration-part>"),
            NodeType::ConstDeclaration => write!(f,
"<const-declaration>"),
            NodeType::TypeDeclaration => write!(f,
"<type-declaration>"),
            NodeType::VarDeclaration => write!(f,
"<var-declaration>"),
            NodeType::IdentifierList => write!(f,
"<identifier-list>"),
            NodeType::Type => write!(f, "<type>"),
            NodeType::ArrayType => write!(f, "<array-type>"),
            NodeType::Range => write!(f, "<range>"),
            NodeType::SubprogramDeclaration => write!(f,
"<subprogram-declaration>"),
```



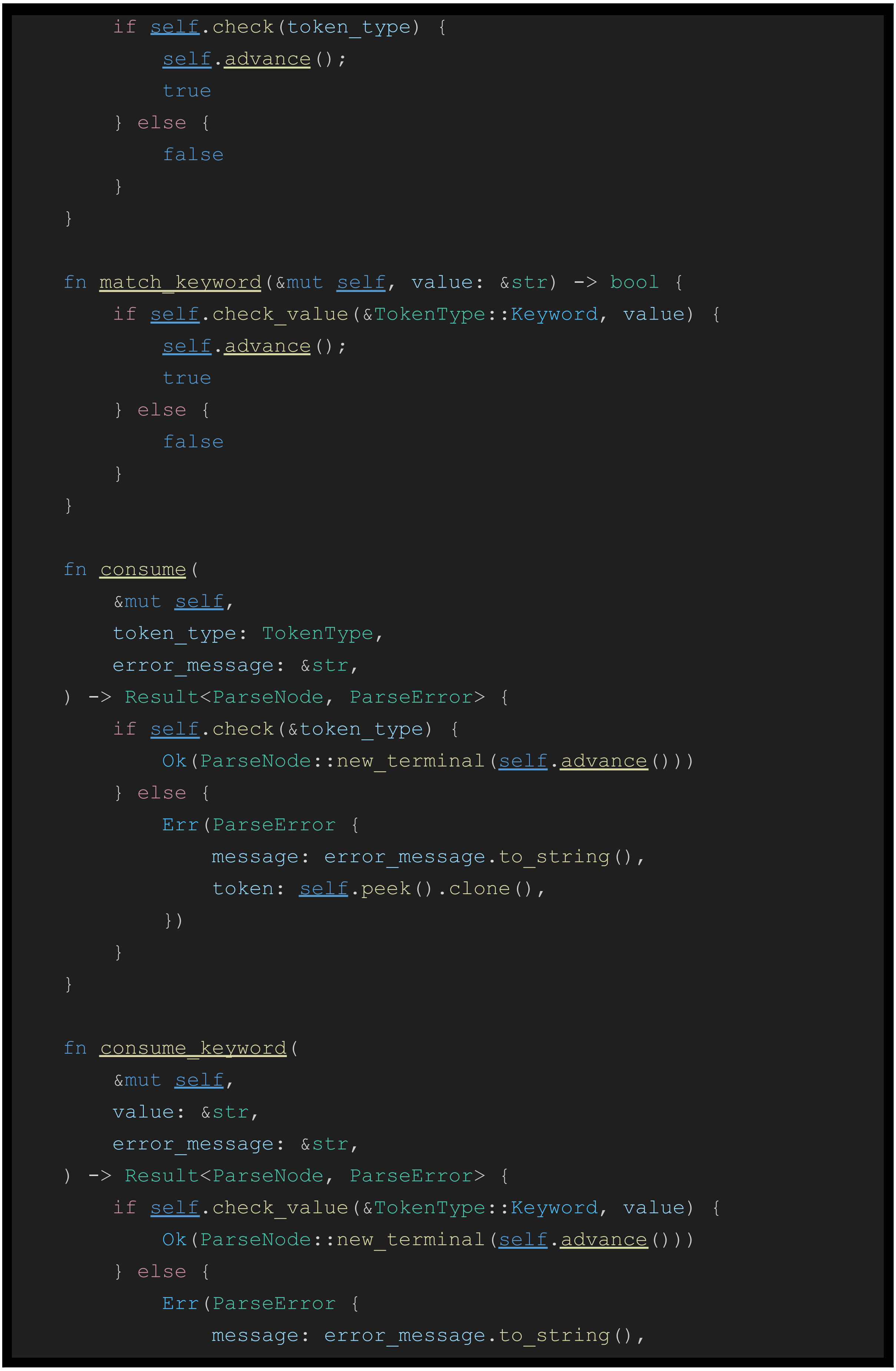
```
            NodeType::ProcedureDeclaration => write!(f,
"<procedure-declaration>"),
            NodeType::FunctionDeclaration => write!(f,
"<function-declaration>"),
            NodeType::FormalParameterList => write!(f,
"<formal-parameter-list>"),
            NodeType::CompoundStatement => write!(f,
"<compound-statement>"),
            NodeType::StatementList => write!(f, "<statement-list>"),
            NodeType::AssignmentStatement => write!(f,
"<assignment-statement>"),
            NodeType::IfStatement => write!(f, "<if-statement>"),
            NodeType::WhileStatement => write!(f,
"<while-statement>"),
            NodeType::ForStatement => write!(f, "<for-statement>"),
            NodeType::ProcedureOrFunctionCall => write!(f,
"<procedure/function-call>"),
            NodeType::ParameterList => write!(f, "<parameter-list>"),
            NodeType::Expression => write!(f, "<expression>"),
            NodeType::SimpleExpression => write!(f,
"<simple-expression>"),
            NodeType::Term => write!(f, "<term>"),
            NodeType::Factor => write!(f, "<factor>"),
        }
    }
}

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

impl ParseNode {
    fn fmt_recursive(&self, f: &mut fmt::Formatter<'_>, indent_level:
usize) -> fmt::Result {
        let indent = " ".repeat(indent_level * 2);

        writeln!(f, "{}{}", indent, self.node_type)?;

        for child in &self.children {
            child.fmt_recursive(f, indent_level + 1)?;
        }

        Ok(())
    }
}
```

parser.rs

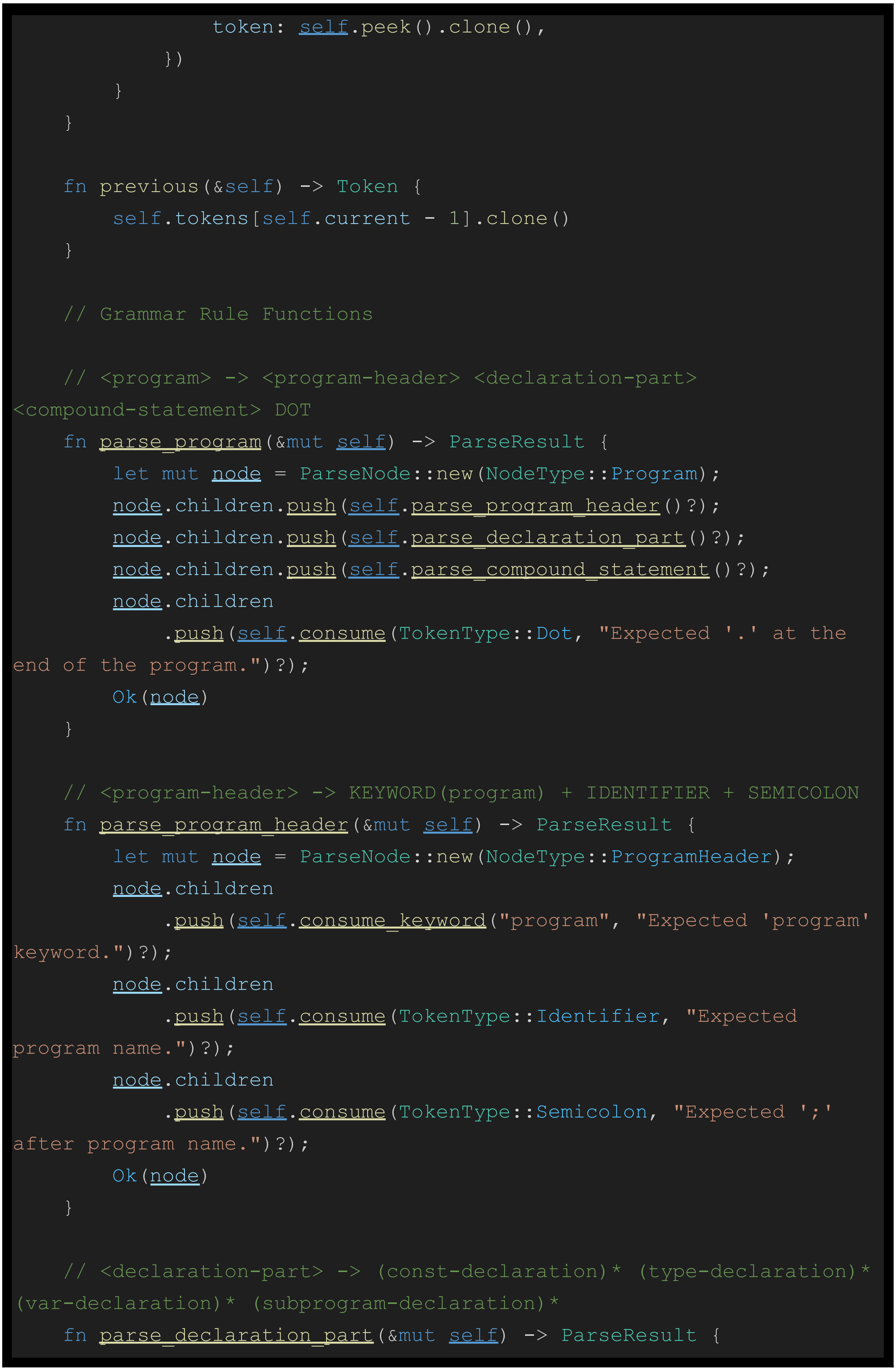
```
use crate::node::{NodeType, ParseNode};
use crate::token::{Token, TokenType};
use std::fmt;

#[derive(Debug)]
pub struct ParseError {
    pub message: String,
    pub token: Token,
}

impl fmt::Display for ParseError {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "Syntax error: {} (found {})", self.message,
self.token)
    }
}

type ParseResult = Result<ParseNode, ParseError>;

pub struct Parser {
    tokens: Vec<Token>,
    current: usize,
}

impl Parser {
    pub fn new(tokens: Vec<Token>) -> Self {
        Parser { tokens, current: 0 }
    }

    pub fn parse(&mut self) -> ParseResult {
        let program_node = self.parse_program()?;

        if !self.is_at_end() {
```



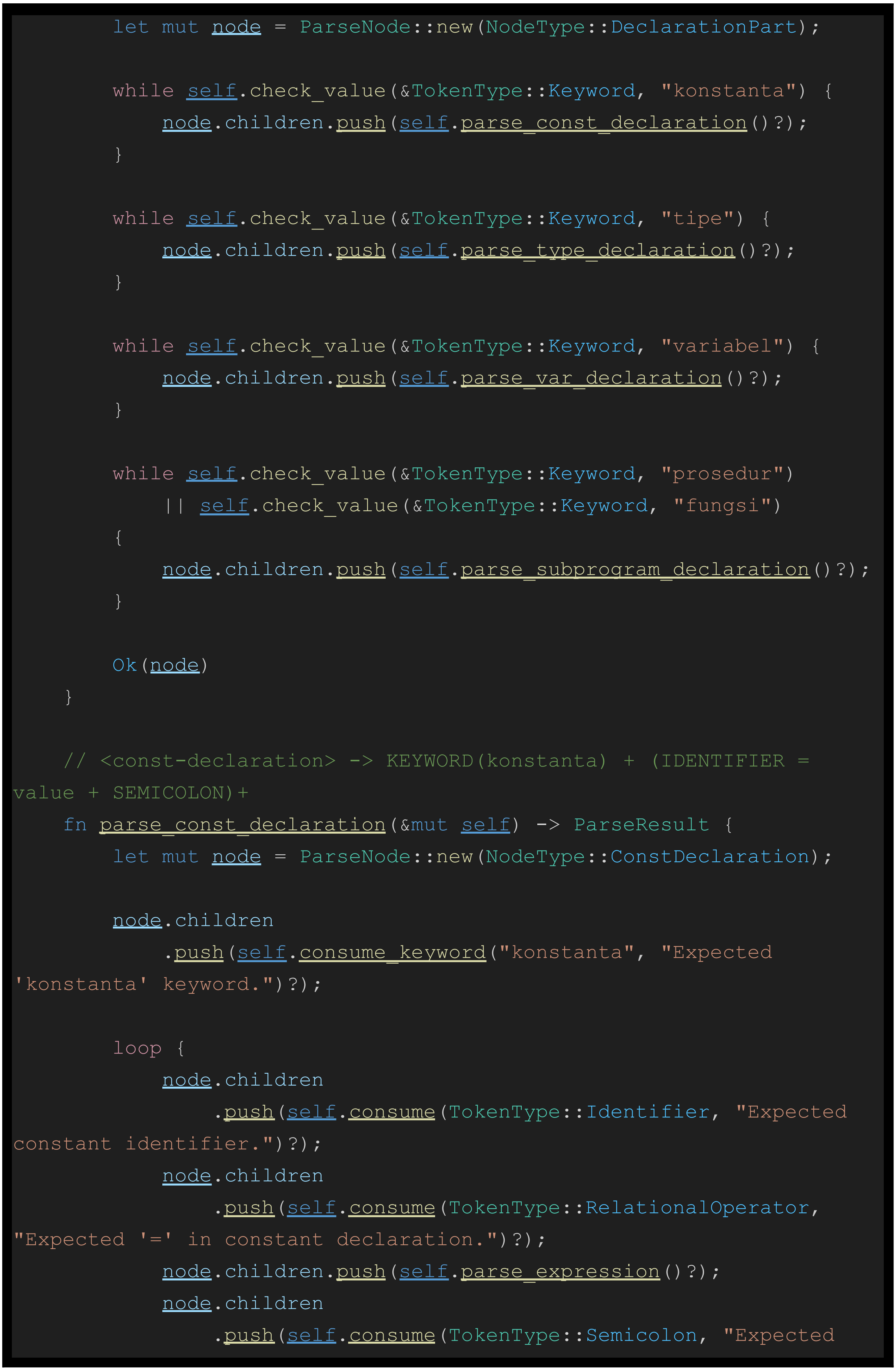
```
            return Err(ParseError {
                message: "Unexpected token after end of
program.".to_string(),
                token: self.peek().clone(),
            });
        }

        Ok(program_node)
    }

    fn peek(&self) -> &Token {
        &self.tokens[self.current]
    }

    fn advance(&mut self) -> Token {
        if !self.is_at_end() {
            self.current += 1;
        }
        self.tokens[self.current - 1].clone()
    }

    fn is_at_end(&self) -> bool {
        self.current >= self.tokens.len()
    }

    fn check(&self, token_type: &TokenType) -> bool {
        if self.is_at_end() {
            return false;
        }
        self.peek().token_type == *token_type
    }

    fn check_value(&self, token_type: &TokenType, value: &str) ->
bool {
        if self.is_at_end() {
            return false;
        }
        let token = self.peek();
        token.token_type == *token_type && token.value == value
    }

    fn match_token(&mut self, token_type: &TokenType) -> bool {
```



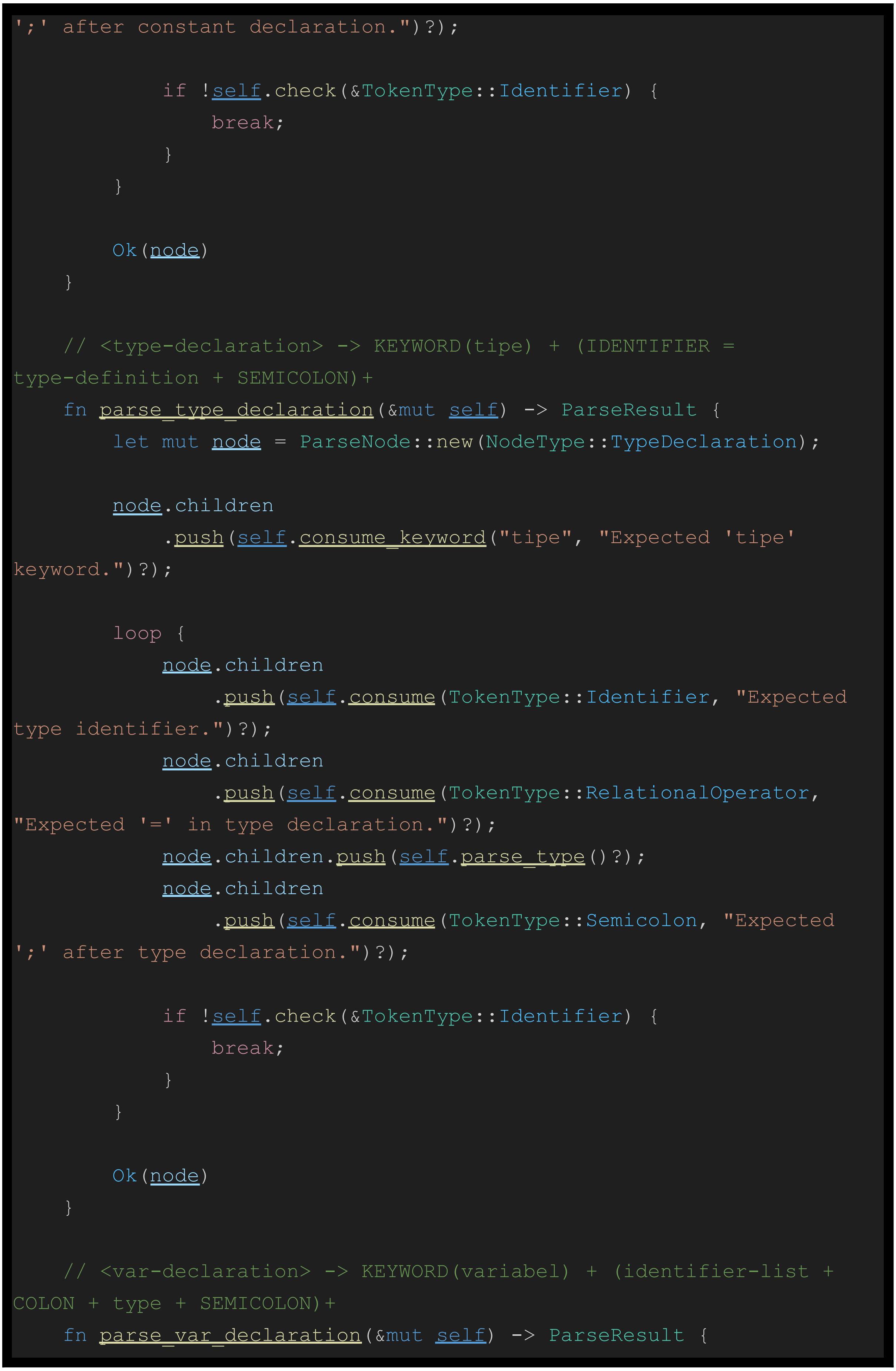
```
        if self.check(token_type) {
            self.advance();
            true
        } else {
            false
        }
    }

    fn match_keyword(&mut self, value: &str) -> bool {
        if self.check_value(&TokenType::Keyword, value) {
            self.advance();
            true
        } else {
            false
        }
    }

    fn consume(
        &mut self,
        token_type: TokenType,
        error_message: &str,
    ) -> Result<ParseNode, ParseError> {
        if self.check(&token_type) {
            Ok(ParseNode::new_terminal(self.advance()))
        } else {
            Err(ParseError {
                message: error_message.to_string(),
                token: self.peek().clone(),
            })
        }
    }

    fn consume_keyword(
        &mut self,
        value: &str,
        error_message: &str,
    ) -> Result<ParseNode, ParseError> {
        if self.check_value(&TokenType::Keyword, value) {
            Ok(ParseNode::new_terminal(self.advance()))
        } else {
            Err(ParseError {
                message: error_message.to_string(),
```



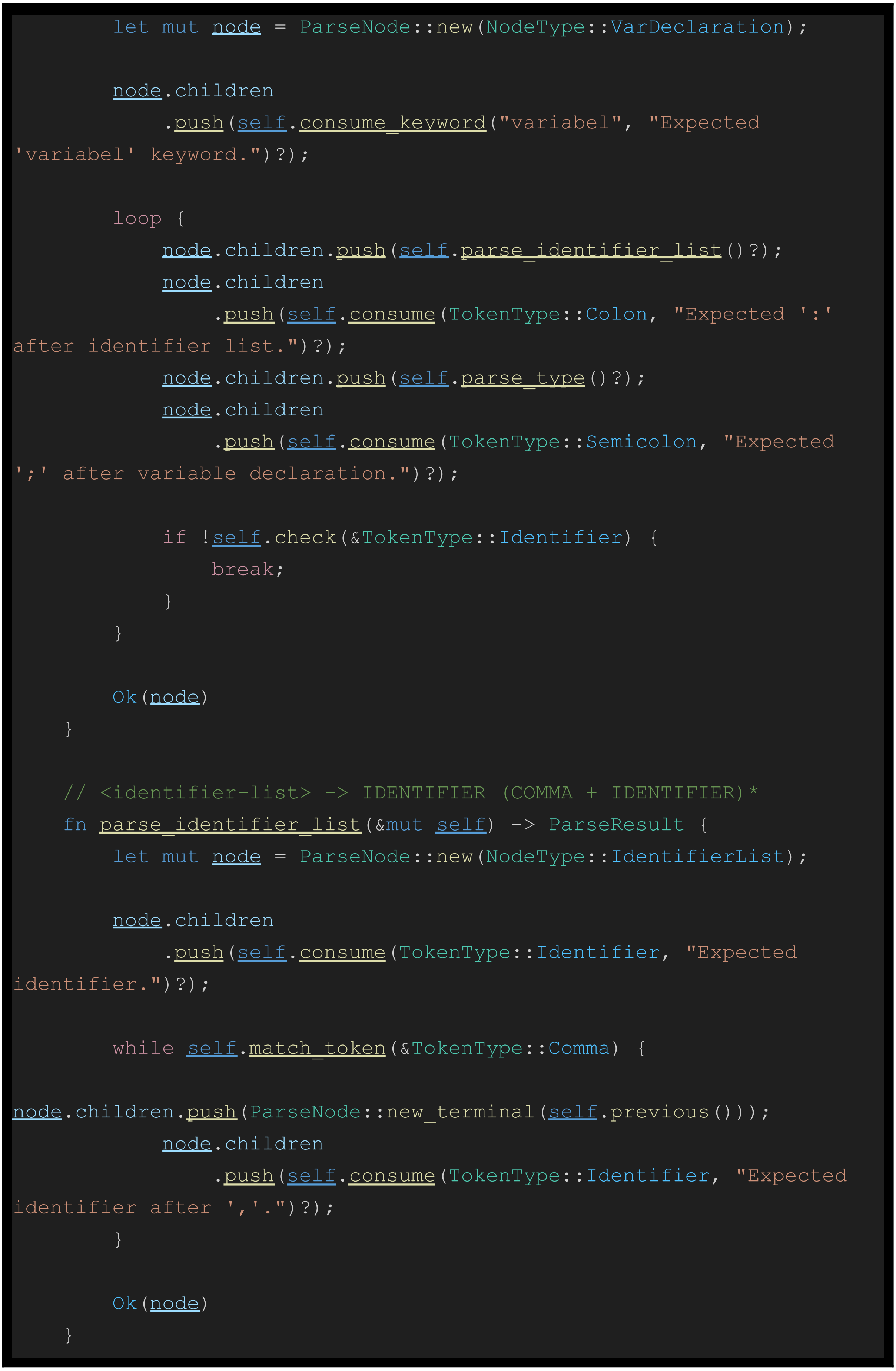
```
                token: self.peek().clone(),
            })
        }
    }

    fn previous(&self) -> Token {
        self.tokens[self.current - 1].clone()
    }

    // Grammar Rule Functions

    // <program> -> <program-header> <declaration-part>
<compound-statement> DOT
    fn parse_program(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::Program);
        node.children.push(self.parse_program_header()?);
        node.children.push(self.parse_declaration_part()?);
        node.children.push(self.parse_compound_statement()?);
        node.children
            .push(self.consume(TokenType::Dot, "Expected '.' at the
end of the program.")?);
        Ok(node)
    }

    // <program-header> -> KEYWORD(program) + IDENTIFIER + SEMICOLON
    fn parse_program_header(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::ProgramHeader);
        node.children
            .push(self.consume_keyword("program", "Expected 'program'
keyword.")?);
        node.children
            .push(self.consume(TokenType::Identifier, "Expected
program name.")?);
        node.children
            .push(self.consume(TokenType::Semicolon, "Expected ';'
after program name.")?);
        Ok(node)
    }

    // <declaration-part> -> (const-declaration)* (type-declaration)*
(var-declaration)* (subprogram-declaration)*
    fn parse_declaration_part(&mut self) -> ParseResult {
```



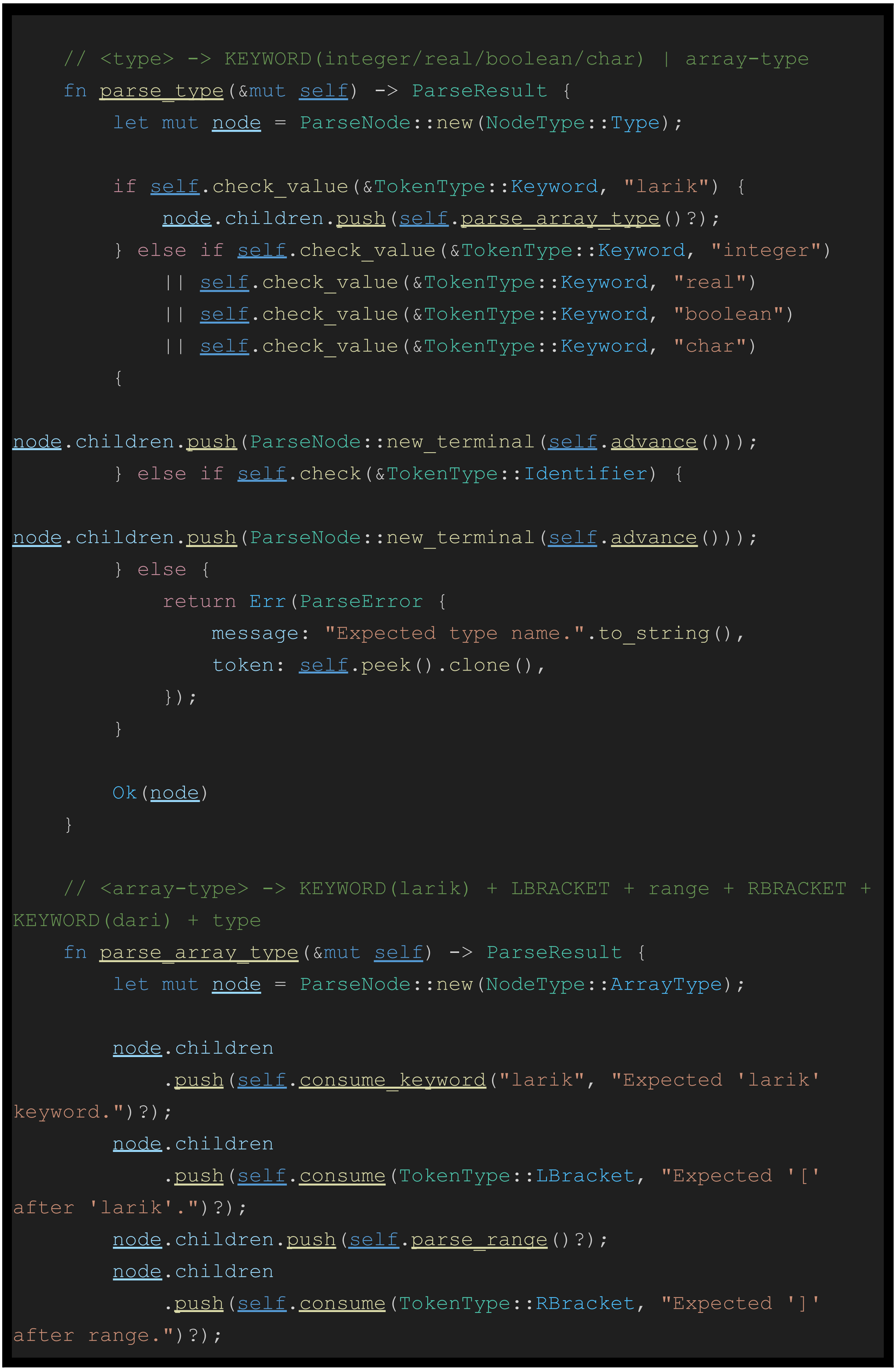
```
        let mut node = ParseNode::new(NodeType::DeclarationPart);

        while self.check_value(&TokenType::Keyword, "konstanta") {
            node.children.push(self.parse_const_declaration()?);
        }

        while self.check_value(&TokenType::Keyword, "tipe") {
            node.children.push(self.parse_type_declaration()?);
        }

        while self.check_value(&TokenType::Keyword, "variabel") {
            node.children.push(self.parse_var_declaration()?);
        }

        while self.check_value(&TokenType::Keyword, "prosedur")
            || self.check_value(&TokenType::Keyword, "fungsi")
        {
            node.children.push(self.parse_subprogram_declaration()?);
        }

        Ok(node)
    }

    // <const-declaration> -> KEYWORD(konstanta) + (IDENTIFIER =
value + SEMICOLON)+
    fn parse_const_declaration(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::ConstDeclaration);

        node.children
            .push(self.consume_keyword("konstanta", "Expected
'konstanta' keyword.")?);

        loop {
            node.children
                .push(self.consume(TokenType::Identifier, "Expected
constant identifier.")?);
            node.children
                .push(self.consume(TokenType::RelationalOperator,
"Expected '=' in constant declaration.")?);
            node.children.push(self.parse_expression()?);
            node.children
                .push(self.consume(TokenType::Semicolon, "Expected
```



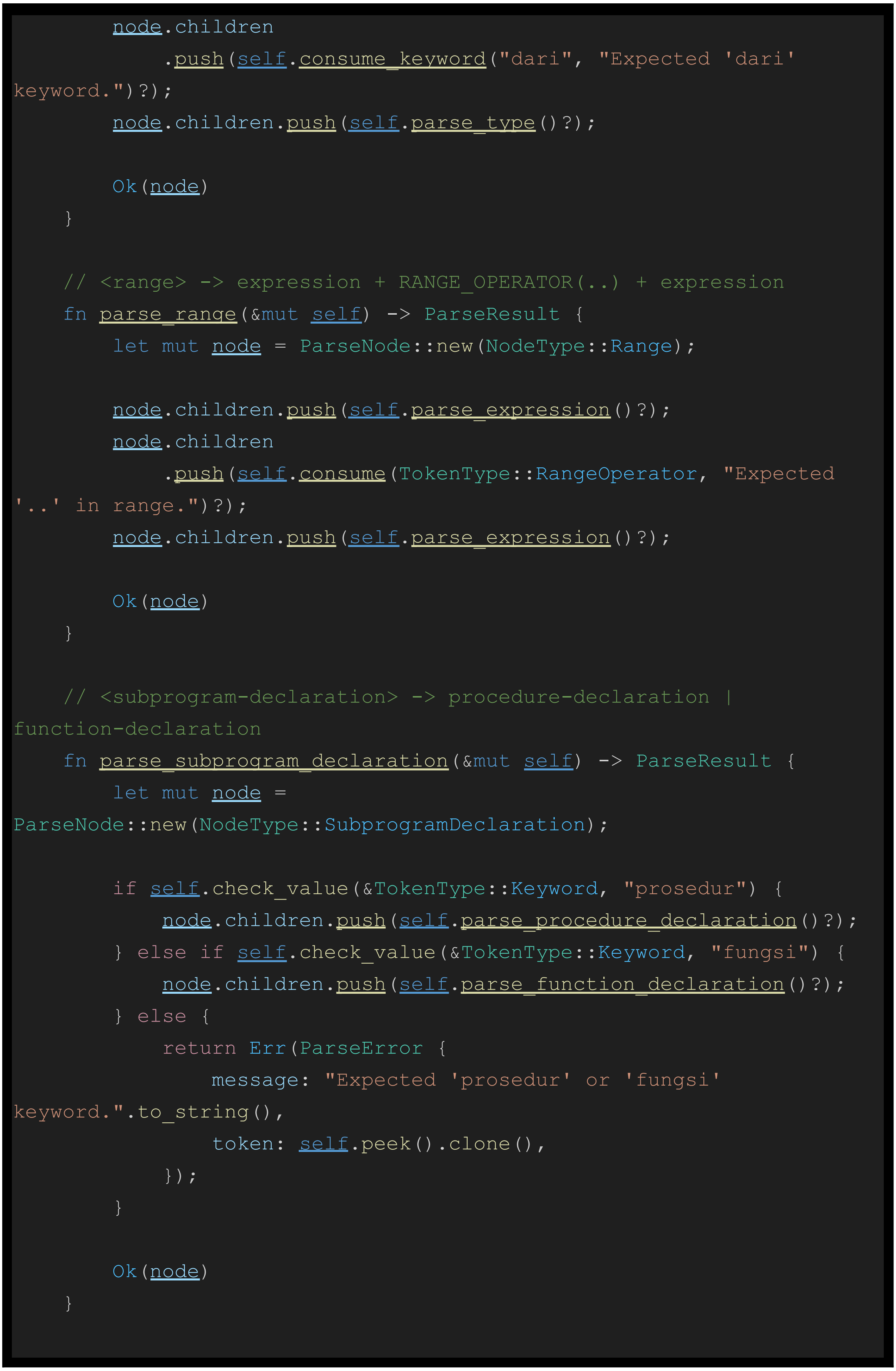
```
';' after constant declaration.")?);

            if !self.check(&TokenType::Identifier) {
                break;
            }
        }

        Ok(node)
    }

    // <type-declaration> -> KEYWORD(tipe) + (IDENTIFIER =
type-definition + SEMICOLON)+
    fn parse_type_declaration(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::TypeDeclaration);

        node.children
            .push(self.consume_keyword("tipe", "Expected 'tipe'
keyword.")?);

        loop {
            node.children
                .push(self.consume(TokenType::Identifier, "Expected
type identifier.")?);
            node.children
                .push(self.consume(TokenType::RelationalOperator,
"Expected '=' in type declaration.")?);
            node.children.push(self.parse_type()?);
            node.children
                .push(self.consume(TokenType::Semicolon, "Expected
';' after type declaration.")?);

            if !self.check(&TokenType::Identifier) {
                break;
            }
        }

        Ok(node)
    }

    // <var-declaration> -> KEYWORD(variabel) + (identifier-list +
COLON + type + SEMICOLON)+
    fn parse_var_declaration(&mut self) -> ParseResult {
```



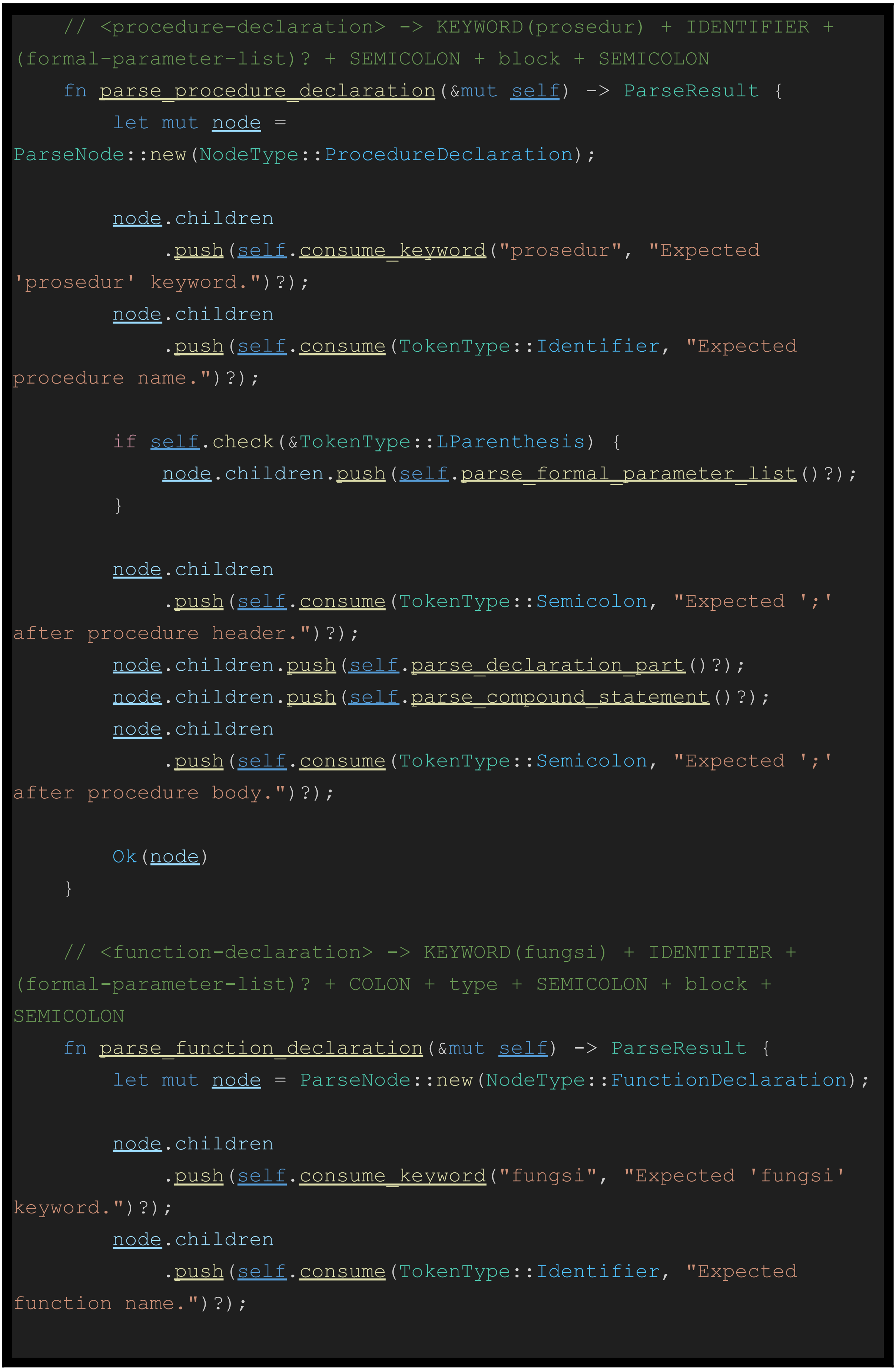
```
        let mut node = ParseNode::new(NodeType::VarDeclaration);

        node.children
            .push(self.consume_keyword("variabel", "Expected 
'variabel' keyword.")?);

        loop {
            node.children.push(self.parse_identifier_list()?);
            node.children
                .push(self.consume(TokenType::Colon, "Expected ':' 
after identifier list.")?);
            node.children.push(self.parse_type()?);
            node.children
                .push(self.consume(TokenType::Semicolon, "Expected 
';' after variable declaration.")?);

            if !self.check(&TokenType::Identifier) {
                break;
            }
        }

        Ok(node)
    }

    // <identifier-list> -> IDENTIFIER (COMMA + IDENTIFIER)*
    fn parse_identifier_list(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::IdentifierList);

        node.children
            .push(self.consume(TokenType::Identifier, "Expected 
identifier.")?);

        while self.match_token(&TokenType::Comma) {

node.children.push(ParseNode::new_terminal(self.previous()));
            node.children
                .push(self.consume(TokenType::Identifier, "Expected 
identifier after ','.")?);
        }

        Ok(node)
    }
```



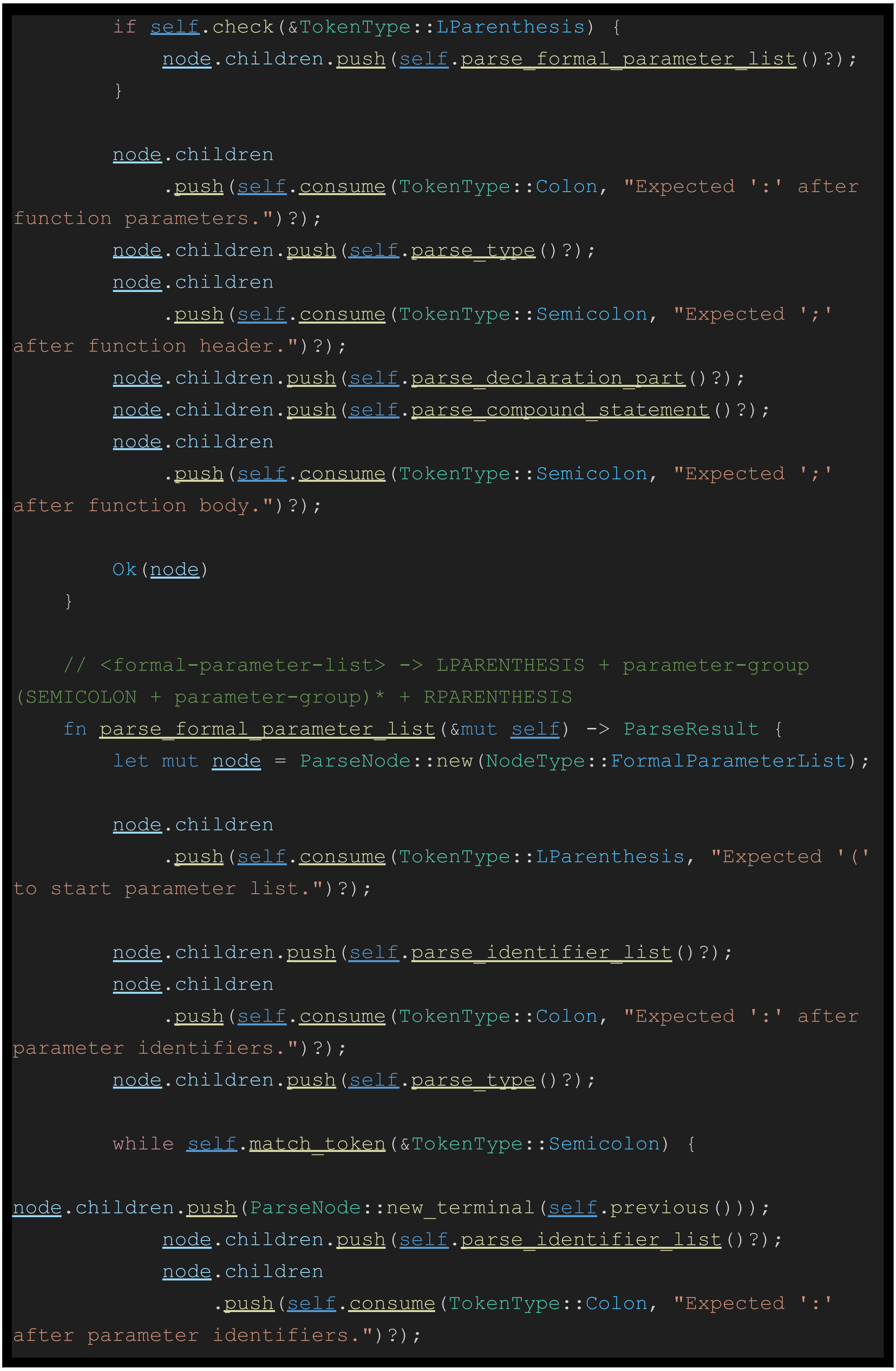
```
    // <type> -> KEYWORD(integer/real/boolean/char) | array-type
    fn parse_type(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::Type);

        if self.check_value(&TokenType::Keyword, "larik") {
            node.children.push(self.parse_array_type()?);
        } else if self.check_value(&TokenType::Keyword, "integer")
            || self.check_value(&TokenType::Keyword, "real")
            || self.check_value(&TokenType::Keyword, "boolean")
            || self.check_value(&TokenType::Keyword, "char")
        {

node.children.push(ParseNode::new_terminal(self.advance()));
        } else if self.check(&TokenType::Identifier) {

node.children.push(ParseNode::new_terminal(self.advance()));
        } else {
            return Err(ParseError {
                message: "Expected type name.".to_string(),
                token: self.peek().clone(),
            });
        }

        Ok(node)
    }

    // <array-type> -> KEYWORD(larik) + LBRACKET + range + RBRACKET +
KEYWORD(dari) + type
    fn parse_array_type(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::ArrayType);

        node.children
            .push(self.consume_keyword("larik", "Expected 'larik'
keyword.")?);
        node.children
            .push(self.consume(TokenType::LBracket, "Expected '['
after 'larik'.")?);
        node.children.push(self.parse_range()?);
        node.children
            .push(self.consume(TokenType::RBracket, "Expected ']'
after range.")?);
```



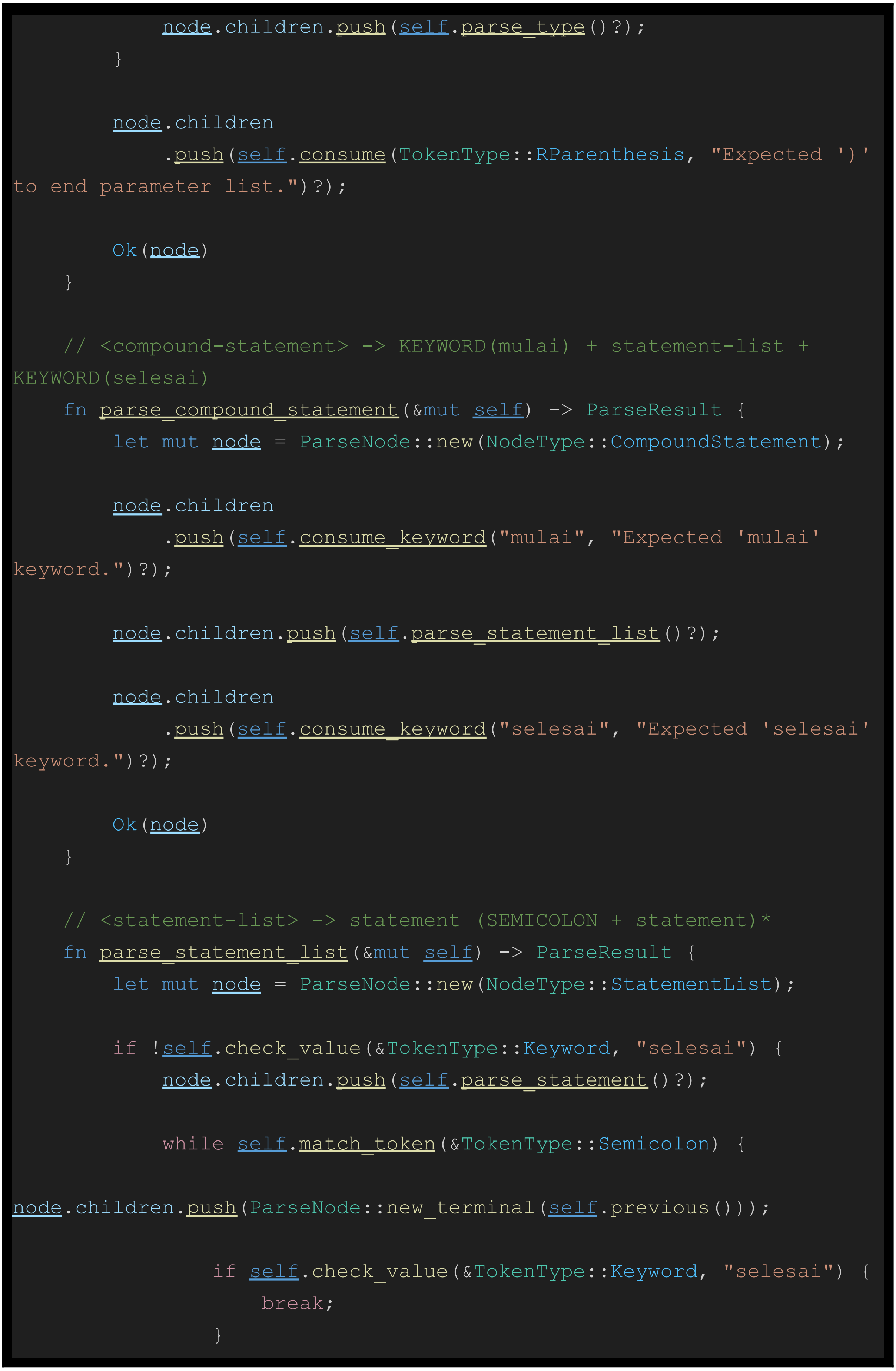
```
        node.children
            .push(self.consume_keyword("dari", "Expected 'dari' keyword.")?);
        node.children.push(self.parse_type()?);

        Ok(node)
    }

    // <range> -> expression + RANGE_OPERATOR(..) + expression
    fn parse_range(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::Range);

        node.children.push(self.parse_expression()?);
        node.children
            .push(self.consume(TokenType::RangeOperator, "Expected '..' in range.")?);
        node.children.push(self.parse_expression()?);

        Ok(node)
    }

    // <subprogram-declaration> -> procedure-declaration | function-declaration
    fn parse_subprogram_declaration(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::SubprogramDeclaration);

        if self.check_value(&TokenType::Keyword, "prosedur") {
            node.children.push(self.parse_procedure_declaration()?);
        } else if self.check_value(&TokenType::Keyword, "fungsi") {
            node.children.push(self.parse_function_declaration()?);
        } else {
            return Err(ParseError {
                message: "Expected 'prosedur' or 'fungsi' keyword.".to_string(),
                token: self.peek().clone(),
            });
        }

        Ok(node)
    }
```



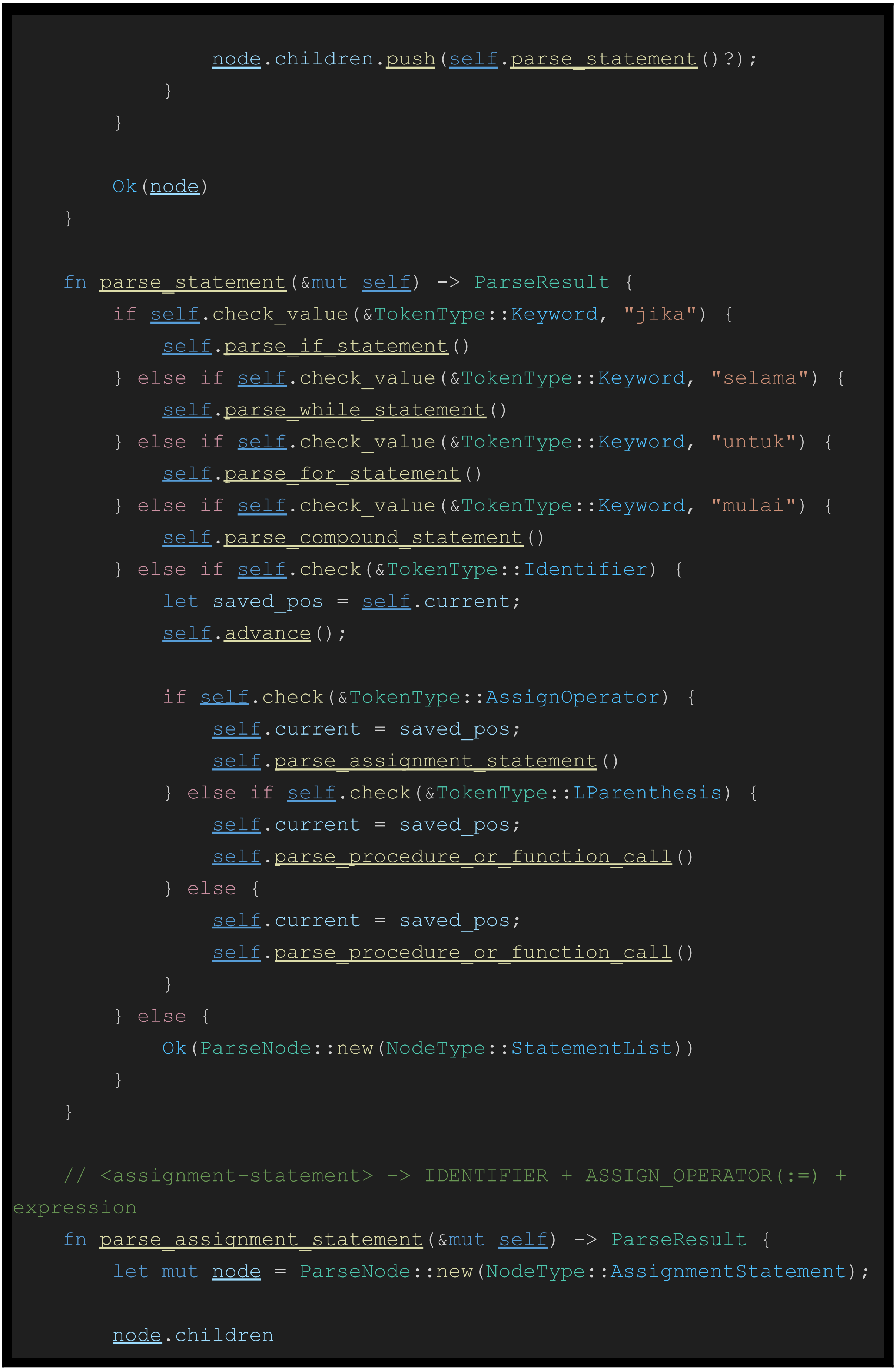
```
    // <procedure-declaration> -> KEYWORD(prosedur) + IDENTIFIER +
(formal-parameter-list)? + SEMICOLON + block + SEMICOLON
    fn parse_procedure_declaration(&mut self) -> ParseResult {
        let mut node =
ParseNode::new(NodeType::ProcedureDeclaration);

        node.children
            .push(self.consume_keyword("prosedur", "Expected
'prosedur' keyword.")?);
        node.children
            .push(self.consume(TokenType::Identifier, "Expected
procedure name.")?);

        if self.check(&TokenType::LParenthesis) {
            node.children.push(self.parse_formal_parameter_list()?);
        }

        node.children
            .push(self.consume(TokenType::Semicolon, "Expected ';'
after procedure header.")?);
        node.children.push(self.parse_declaration_part()?);
        node.children.push(self.parse_compound_statement()?);
        node.children
            .push(self.consume(TokenType::Semicolon, "Expected ';'
after procedure body.")?);

        Ok(node)
    }

    // <function-declaration> -> KEYWORD(fungsi) + IDENTIFIER +
(formal-parameter-list)? + COLON + type + SEMICOLON + block +
SEMICOLON
    fn parse_function_declaration(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::FunctionDeclaration);

        node.children
            .push(self.consume_keyword("fungsi", "Expected 'fungsi'
keyword.")?);
        node.children
            .push(self.consume(TokenType::Identifier, "Expected
function name.")?);
```



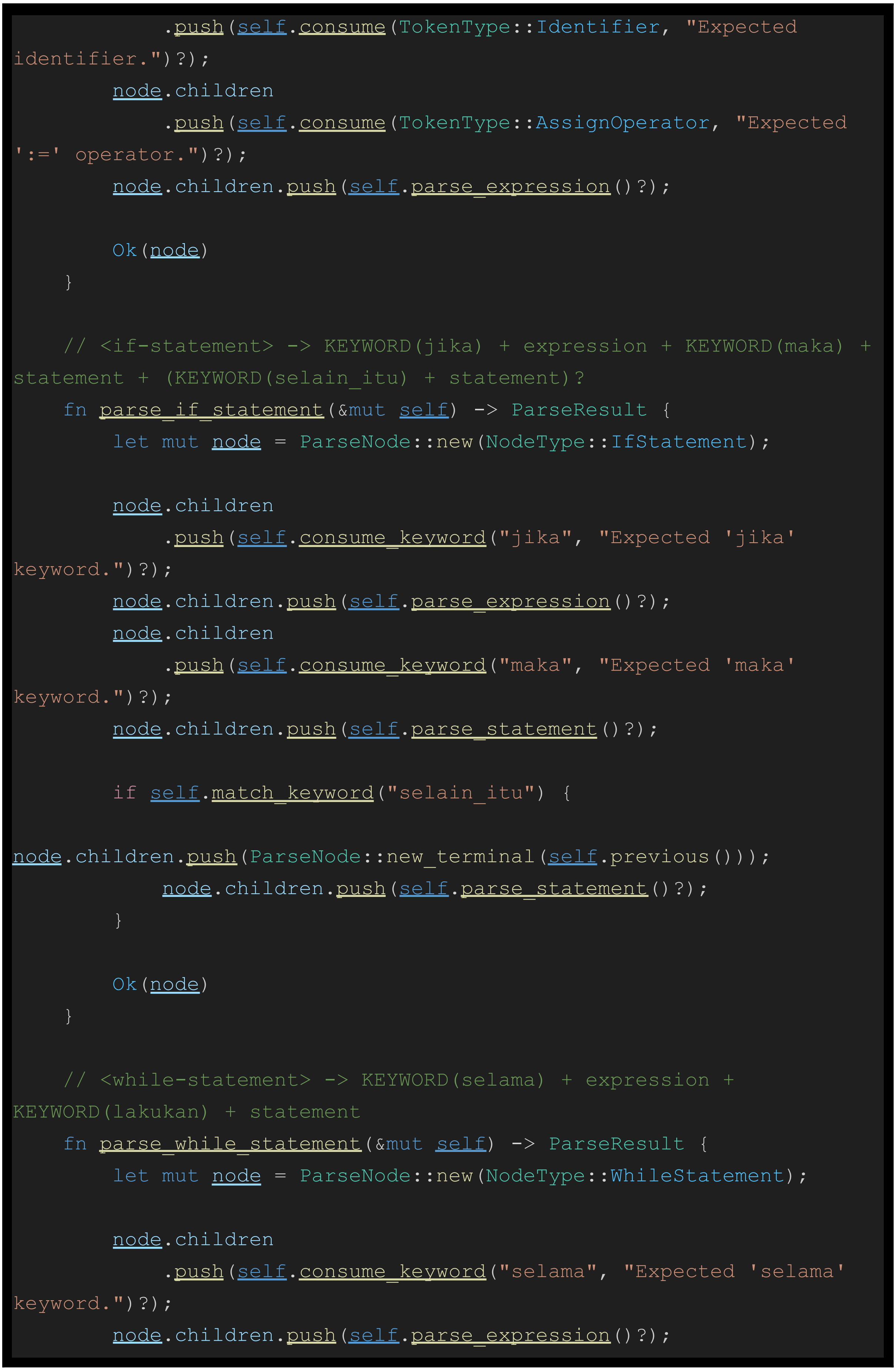
```
        if self.check(&TokenType::LParenthesis) {
            node.children.push(self.parse_formal_parameter_list()?);
        }

        node.children
            .push(self.consume(TokenType::Colon, "Expected ':' after
function parameters.")?);
        node.children.push(self.parse_type()?);
        node.children
            .push(self.consume(TokenType::Semicolon, "Expected ';'
after function header.")?);
        node.children.push(self.parse_declaration_part()?);
        node.children.push(self.parse_compound_statement()?);
        node.children
            .push(self.consume(TokenType::Semicolon, "Expected ';'
after function body.")?);

        Ok(node)
    }

    // <formal-parameter-list> -> LPARENTHESIS + parameter-group
(SEMICOLON + parameter-group)* + RPARENTHESIS
    fn parse_formal_parameter_list(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::FormalParameterList);

        node.children
            .push(self.consume(TokenType::LParenthesis, "Expected '('
to start parameter list.")?);

        node.children.push(self.parse_identifier_list()?);
        node.children
            .push(self.consume(TokenType::Colon, "Expected ':' after
parameter identifiers.")?);
        node.children.push(self.parse_type()?);

        while self.match_token(&TokenType::Semicolon) {

node.children.push(ParseNode::new_terminal(self.previous()));
            node.children.push(self.parse_identifier_list()?);
            node.children
                .push(self.consume(TokenType::Colon, "Expected ':'
after parameter identifiers.")?);
```



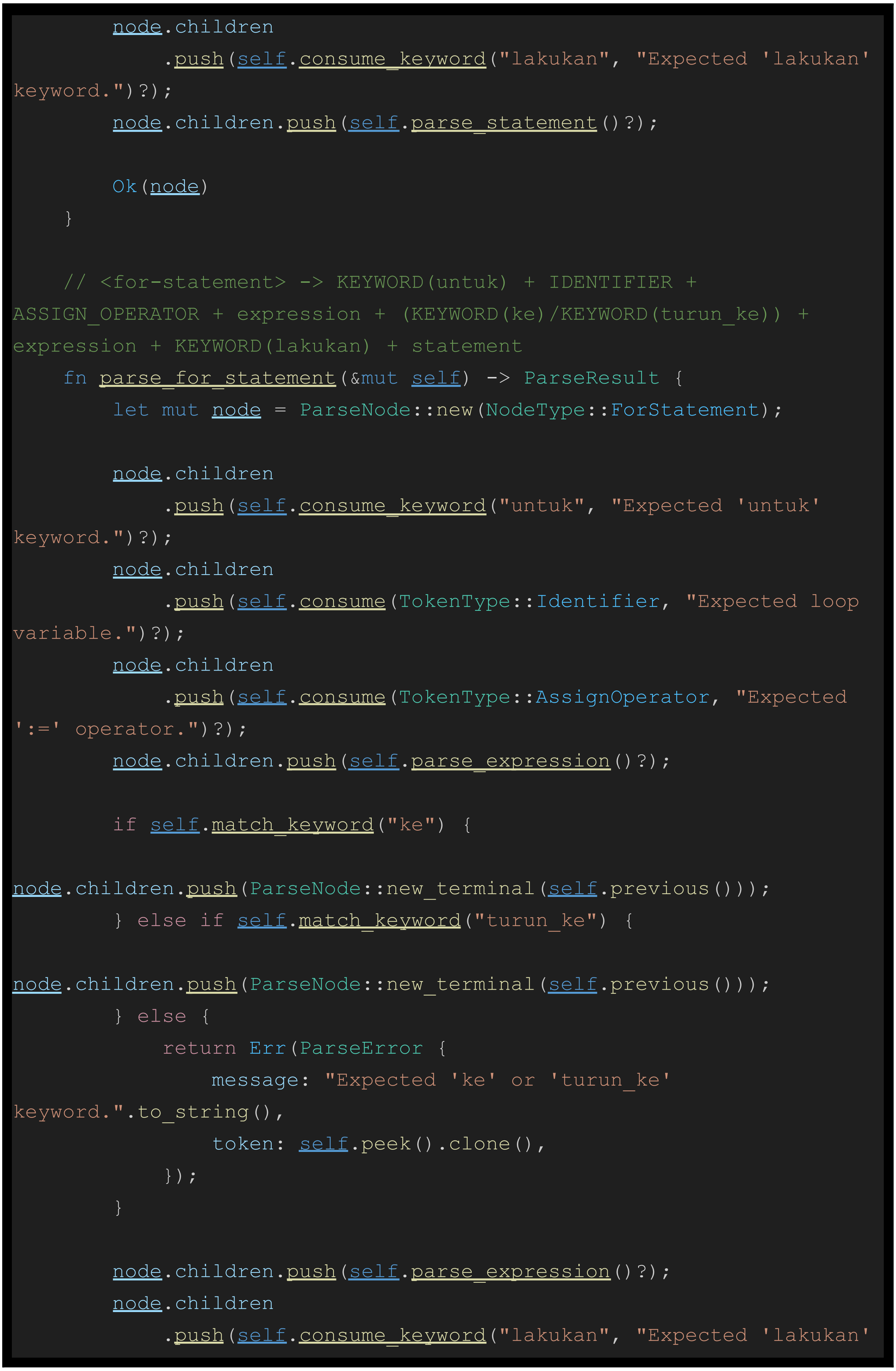
```
            node.children.push(self.parse_type()?);
        }

        node.children
            .push(self.consume(TokenType::RParenthesis, "Expected ')'
to end parameter list.")?);

        Ok(node)
    }

    // <compound-statement> -> KEYWORD(mulai) + statement-list +
KEYWORD(selesai)
    fn parse_compound_statement(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::CompoundStatement);

        node.children
            .push(self.consume_keyword("mulai", "Expected 'mulai'
keyword.")?);

        node.children.push(self.parse_statement_list()?);

        node.children
            .push(self.consume_keyword("selesai", "Expected 'selesai'
keyword.")?);

        Ok(node)
    }

    // <statement-list> -> statement (SEMICOLON + statement)*
    fn parse_statement_list(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::StatementList);

        if !self.check_value(&TokenType::Keyword, "selesai") {
            node.children.push(self.parse_statement()?);

            while self.match_token(&TokenType::Semicolon) {

node.children.push(ParseNode::new_terminal(self.previous()));

                if self.check_value(&TokenType::Keyword, "selesai") {
                    break;
                }
```



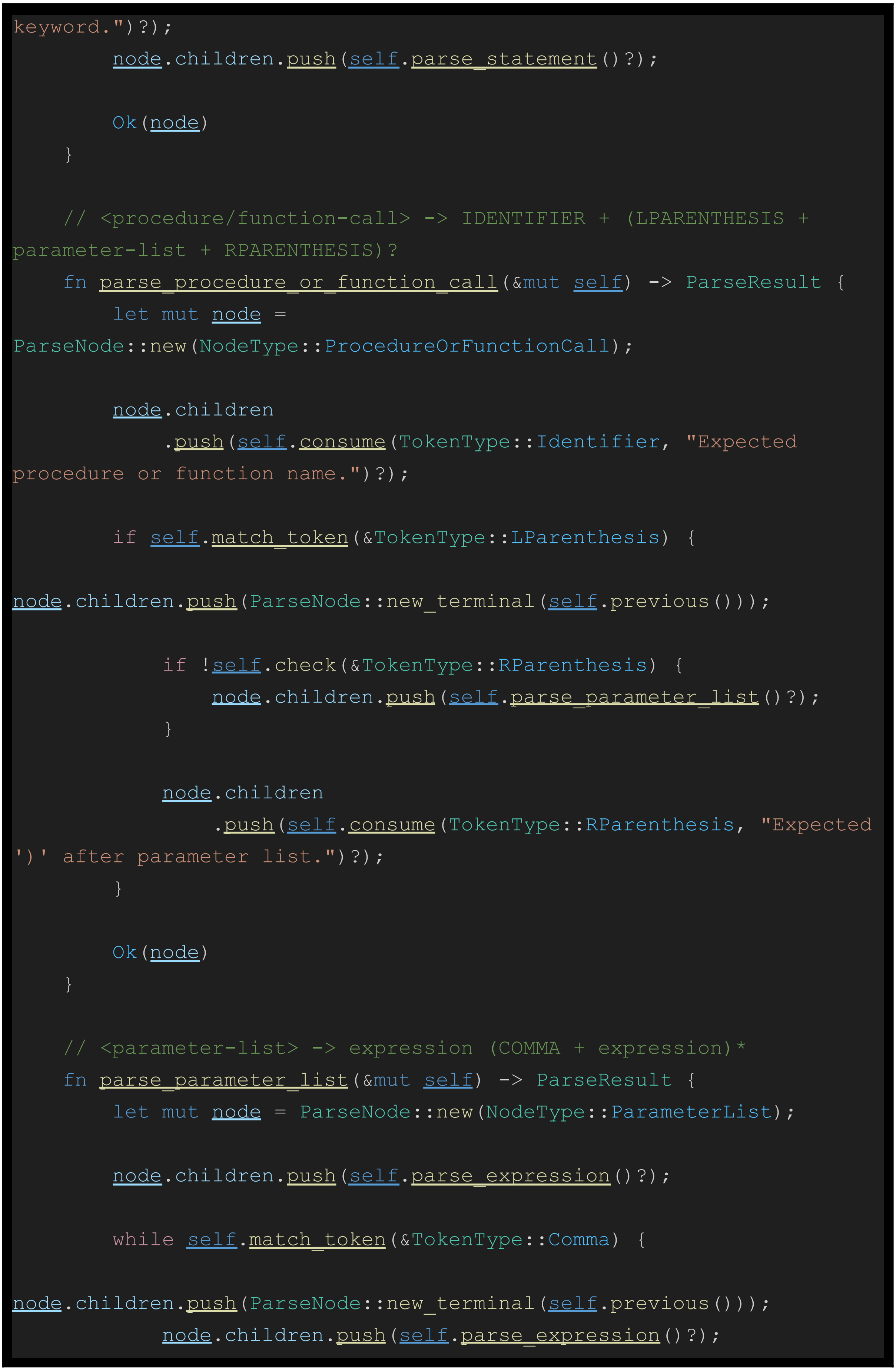
```
                node.children.push(self.parse_statement()?);
            }
        }

        Ok(node)
    }

    fn parse_statement(&mut self) -> ParseResult {
        if self.check_value(&TokenType::Keyword, "jika") {
            self.parse_if_statement()
        } else if self.check_value(&TokenType::Keyword, "selama") {
            self.parse_while_statement()
        } else if self.check_value(&TokenType::Keyword, "untuk") {
            self.parse_for_statement()
        } else if self.check_value(&TokenType::Keyword, "mulai") {
            self.parse_compound_statement()
        } else if self.check(&TokenType::Identifier) {
            let saved_pos = self.current;
            self.advance();

            if self.check(&TokenType::AssignOperator) {
                self.current = saved_pos;
                self.parse_assignment_statement()
            } else if self.check(&TokenType::LParenthesis) {
                self.current = saved_pos;
                self.parse_procedure_or_function_call()
            } else {
                self.current = saved_pos;
                self.parse_procedure_or_function_call()
            }
        } else {
            Ok(ParseNode::new(NodeType::StatementList))
        }
    }

    // <assignment-statement> -> IDENTIFIER + ASSIGN_OPERATOR(:=) +
expression
    fn parse_assignment_statement(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::AssignmentStatement);

        node.children
```



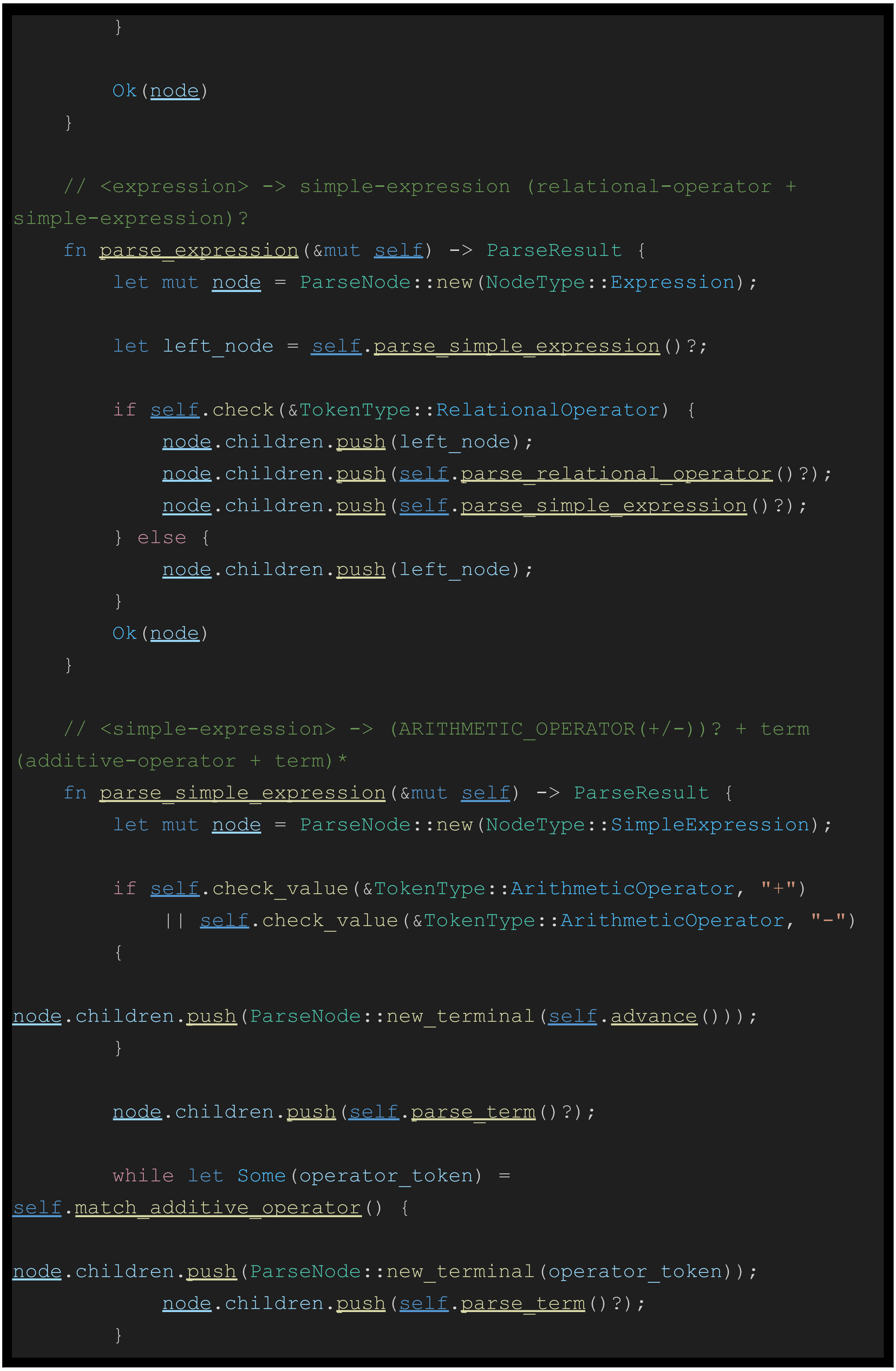
```
            .push(self.consume(TokenType::Identifier, "Expected
identifier.")?);
        node.children
            .push(self.consume(TokenType::AssignOperator, "Expected
':=' operator.")?);
        node.children.push(self.parse_expression()?);

        Ok(node)
    }

    // <if-statement> -> KEYWORD(jika) + expression + KEYWORD(maka) +
statement + (KEYWORD(selain_itu) + statement)?
    fn parse_if_statement(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::IfStatement);

        node.children
            .push(self.consume_keyword("jika", "Expected 'jika'
keyword.")?);
        node.children.push(self.parse_expression()?);
        node.children
            .push(self.consume_keyword("maka", "Expected 'maka'
keyword.")?);
        node.children.push(self.parse_statement()?);

        if self.match_keyword("selain_itu") {

node.children.push(ParseNode::new_terminal(self.previous()));
            node.children.push(self.parse_statement()?);
        }

        Ok(node)
    }

    // <while-statement> -> KEYWORD(selama) + expression +
KEYWORD(lakukan) + statement
    fn parse_while_statement(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::WhileStatement);

        node.children
            .push(self.consume_keyword("selama", "Expected 'selama'
keyword.")?);
        node.children.push(self.parse_expression()?);
```



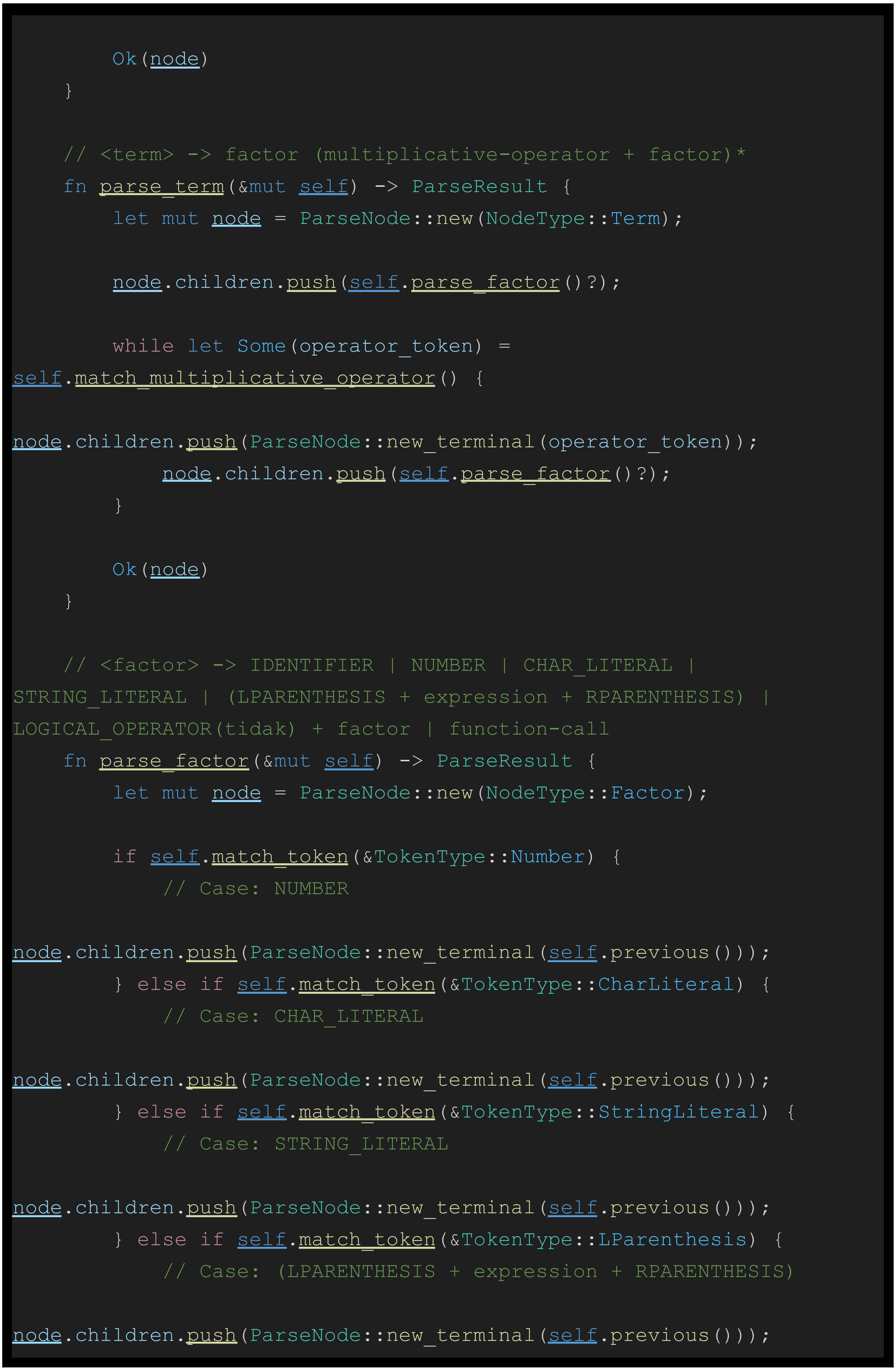
```
        node.children
            .push(self.consume_keyword("lakukan", "Expected 'lakukan'
keyword.")?);
        node.children.push(self.parse_statement()?);

        Ok(node)
    }

    // <for-statement> -> KEYWORD(untuk) + IDENTIFIER +
ASSIGN_OPERATOR + expression + (KEYWORD(ke)/KEYWORD(turun_ke)) +
expression + KEYWORD(lakukan) + statement
    fn parse_for_statement(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::ForStatement);

        node.children
            .push(self.consume_keyword("untuk", "Expected 'untuk'
keyword.")?);
        node.children
            .push(self.consume(TokenType::Identifier, "Expected loop
variable.")?);
        node.children
            .push(self.consume(TokenType::AssignOperator, "Expected
':=' operator.")?);
        node.children.push(self.parse_expression()?);

        if self.match_keyword("ke") {

node.children.push(ParseNode::new_terminal(self.previous()));
        } else if self.match_keyword("turun_ke") {

node.children.push(ParseNode::new_terminal(self.previous()));
        } else {
            return Err(ParseError {
                message: "Expected 'ke' or 'turun_ke'
keyword.".to_string(),
                token: self.peek().clone(),
            });
        }

        node.children.push(self.parse_expression()?);
        node.children
            .push(self.consume_keyword("lakukan", "Expected 'lakukan'
```



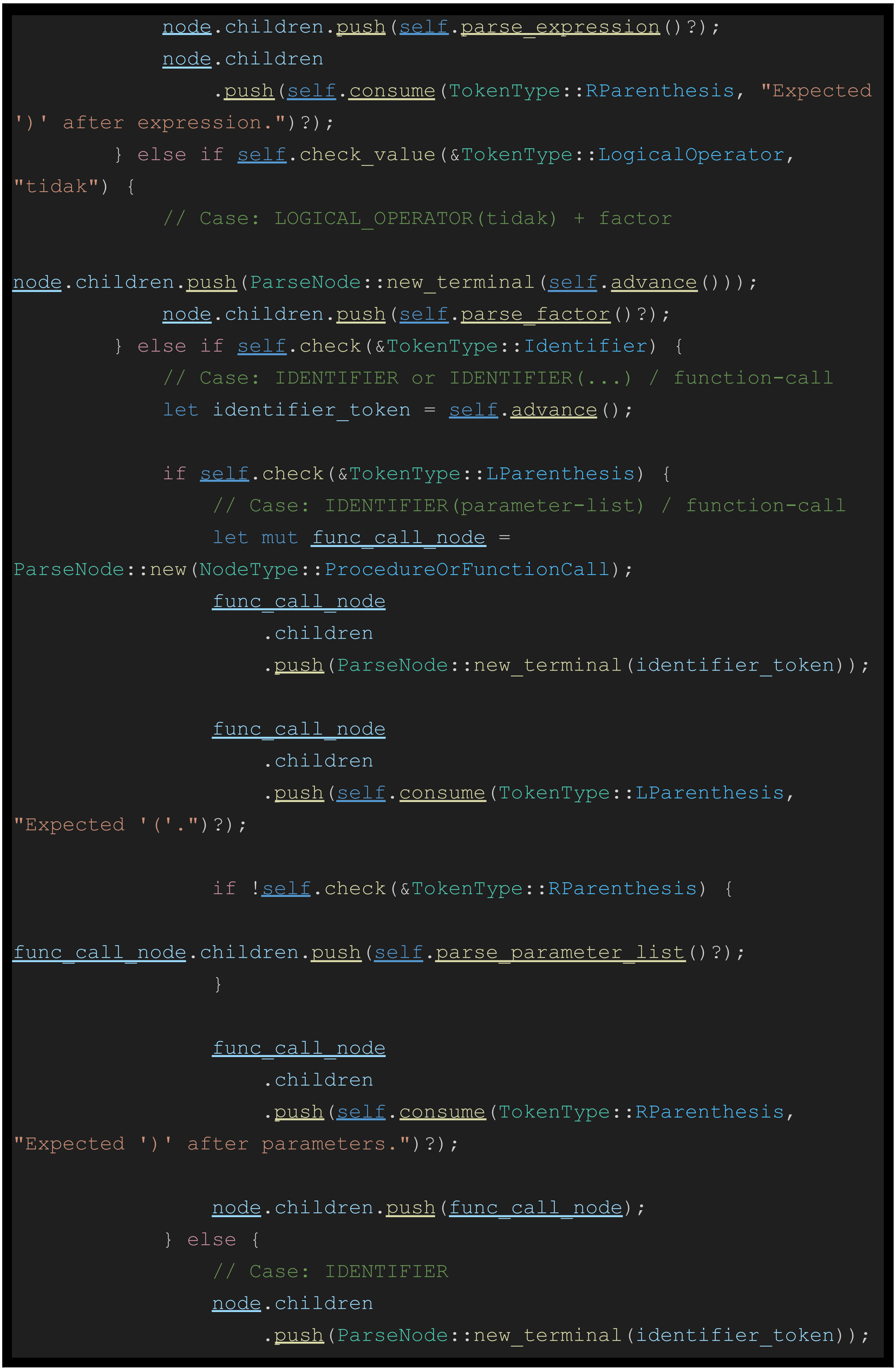
```
keyword.")?);
        node.children.push(self.parse_statement()?);

        Ok(node)
    }

    // <procedure/function-call> -> IDENTIFIER + (LPARENTHESIS +
parameter-list + RPARENTHESIS)?
    fn parse_procedure_or_function_call(&mut self) -> ParseResult {
        let mut node =
ParseNode::new(NodeType::ProcedureOrFunctionCall);

        node.children
            .push(self.consume(TokenType::Identifier, "Expected
procedure or function name.")?);

        if self.match_token(&TokenType::LParenthesis) {

node.children.push(ParseNode::new_terminal(self.previous()));

            if !self.check(&TokenType::RParenthesis) {
                node.children.push(self.parse_parameter_list()?);
            }

            node.children
                .push(self.consume(TokenType::RParenthesis, "Expected
')' after parameter list.")?);
        }

        Ok(node)
    }

    // <parameter-list> -> expression (COMMA + expression)*
    fn parse_parameter_list(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::ParameterList);

        node.children.push(self.parse_expression()?);

        while self.match_token(&TokenType::Comma) {

node.children.push(ParseNode::new_terminal(self.previous()));
            node.children.push(self.parse_expression()?);
```



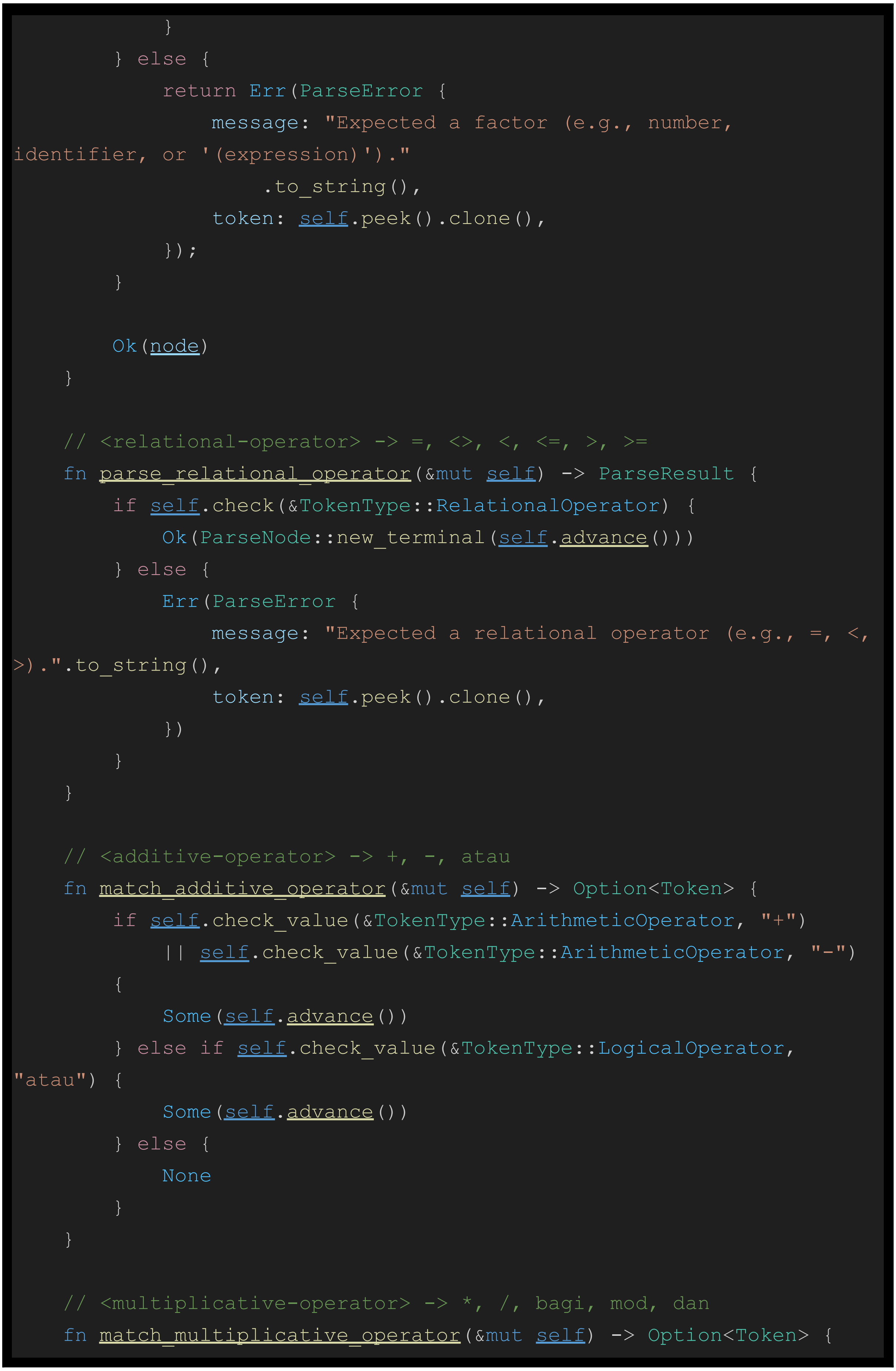
```
        }

        Ok(node)
    }

    // <expression> -> simple-expression (relational-operator +
simple-expression)?
    fn parse_expression(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::Expression);

        let left_node = self.parse_simple_expression()?;

        if self.check(&TokenType::RelationalOperator) {
            node.children.push(left_node);
            node.children.push(self.parse_relational_operator()?);
            node.children.push(self.parse_simple_expression()?);
        } else {
            node.children.push(left_node);
        }
        Ok(node)
    }

    // <simple-expression> -> (ARITHMETIC_OPERATOR(+/-))? + term
(additive-operator + term)*
    fn parse_simple_expression(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::SimpleExpression);

        if self.check_value(&TokenType::ArithmeticOperator, "+")
            || self.check_value(&TokenType::ArithmeticOperator, "-")
        {

node.children.push(ParseNode::new_terminal(self.advance()));
        }

        node.children.push(self.parse_term()?);

        while let Some(operator_token) =
self.match_additive_operator() {

node.children.push(ParseNode::new_terminal(operator_token));
            node.children.push(self.parse_term()?);
        }
```



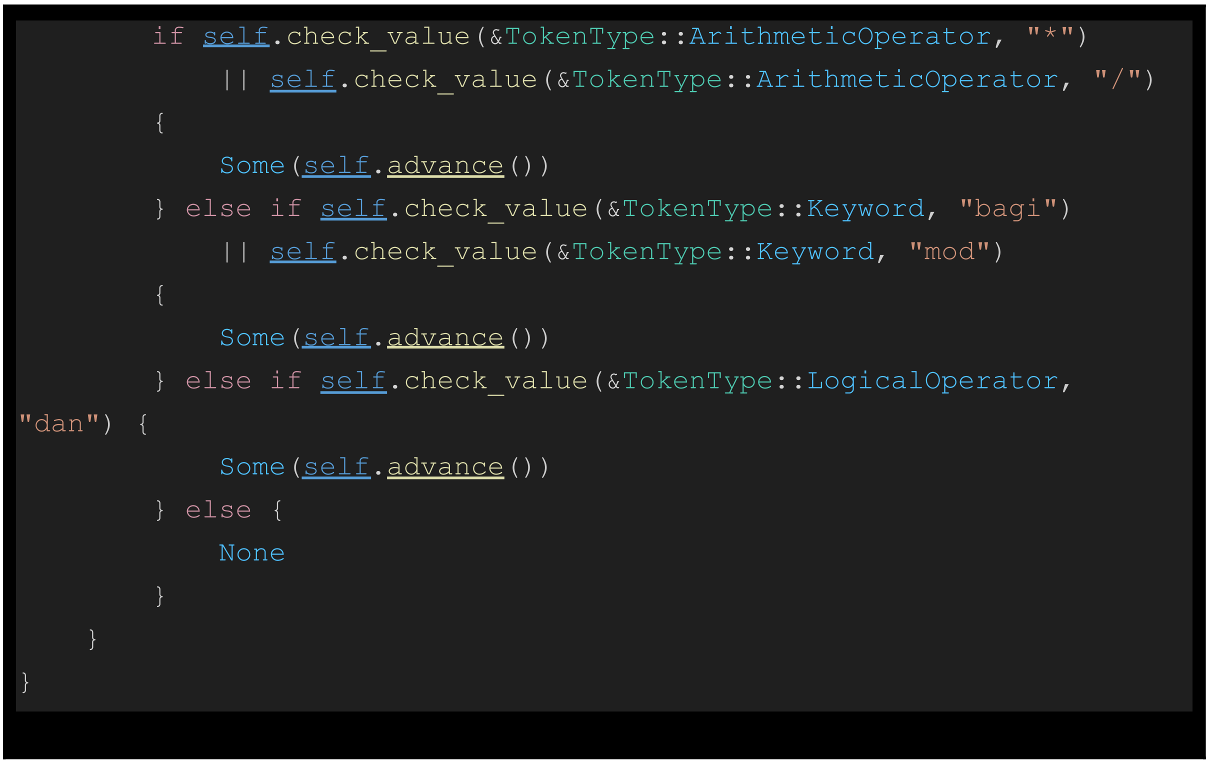
```
        Ok(node)
    }

    // <term> -> factor (multiplicative-operator + factor)*
    fn parse_term(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::Term);

        node.children.push(self.parse_factor()?);

        while let Some(operator_token) =
self.match_multiplicative_operator() {

node.children.push(ParseNode::new_terminal(operator_token));
            node.children.push(self.parse_factor()?);
        }

        Ok(node)
    }

    // <factor> -> IDENTIFIER | NUMBER | CHAR_LITERAL |
STRING_LITERAL | (LPARENTHESIS + expression + RPARENTHESIS) |
LOGICAL_OPERATOR(tidak) + factor | function-call
    fn parse_factor(&mut self) -> ParseResult {
        let mut node = ParseNode::new(NodeType::Factor);

        if self.match_token(&TokenType::Number) {
            // Case: NUMBER

node.children.push(ParseNode::new_terminal(self.previous()));
        } else if self.match_token(&TokenType::CharLiteral) {
            // Case: CHAR_LITERAL

node.children.push(ParseNode::new_terminal(self.previous()));
        } else if self.match_token(&TokenType::StringLiteral) {
            // Case: STRING_LITERAL

node.children.push(ParseNode::new_terminal(self.previous()));
        } else if self.match_token(&TokenType::LParenthesis) {
            // Case: (LPARENTHESIS + expression + RPARENTHESIS)

node.children.push(ParseNode::new_terminal(self.previous()));
```



```
            node.children.push(self.parse_expression()?);
            node.children
                .push(self.consume(TokenType::RParenthesis, "Expected
')' after expression.")?);
        } else if self.check_value(&TokenType::LogicalOperator,
"tidak") {
            // Case: LOGICAL_OPERATOR(tidak) + factor

node.children.push(ParseNode::new_terminal(self.advance()));
            node.children.push(self.parse_factor()?);
        } else if self.check(&TokenType::Identifier) {
            // Case: IDENTIFIER or IDENTIFIER(...) / function-call
            let identifier_token = self.advance();

            if self.check(&TokenType::LParenthesis) {
                // Case: IDENTIFIER(parameter-list) / function-call
                let mut func_call_node =
ParseNode::new(NodeType::ProcedureOrFunctionCall);
                func_call_node
                    .children
                    .push(ParseNode::new_terminal(identifier_token));

                func_call_node
                    .children
                    .push(self.consume(TokenType::LParenthesis,
"Expected '('.")?);

                if !self.check(&TokenType::RParenthesis) {

func_call_node.children.push(self.parse_parameter_list()?);
                }

                func_call_node
                    .children
                    .push(self.consume(TokenType::RParenthesis,
"Expected ')' after parameters.")?);

                node.children.push(func_call_node);
            } else {
                // Case: IDENTIFIER
                node.children
                    .push(ParseNode::new_terminal(identifier_token));
```

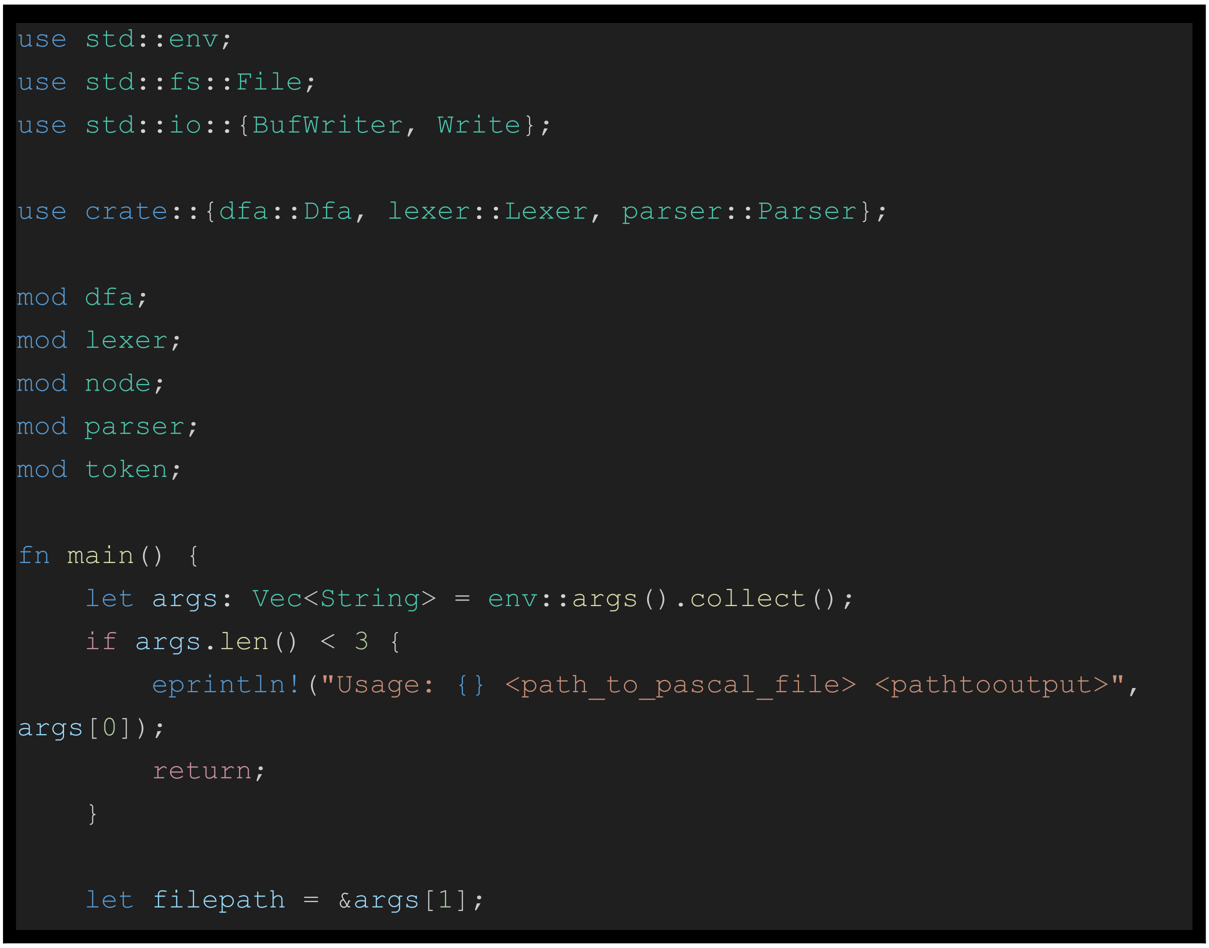


```
            }
        } else {
            return Err(ParseError {
                message: "Expected a factor (e.g., number,
identifier, or '(expression)')."
                    .to_string(),
                token: self.peek().clone(),
            });
        }

        Ok(node)
    }

    // <relational-operator> -> =, <>, <, <=, >, >=
    fn parse_relational_operator(&mut self) -> ParseResult {
        if self.check(&TokenType::RelationalOperator) {
            Ok(ParseNode::new_terminal(self.advance()))
        } else {
            Err(ParseError {
                message: "Expected a relational operator (e.g., =, <,
>).".to_string(),
                token: self.peek().clone(),
            })
        }
    }

    // <additive-operator> -> +, -, atau
    fn match_additive_operator(&mut self) -> Option<Token> {
        if self.check_value(&TokenType::ArithmeticOperator, "+")
            || self.check_value(&TokenType::ArithmeticOperator, "-")
        {
            Some(self.advance())
        } else if self.check_value(&TokenType::LogicalOperator,
"atau") {
            Some(self.advance())
        } else {
            None
        }
    }

    // <multiplicative-operator> -> *, /, bagi, mod, dan
    fn match_multiplicative_operator(&mut self) -> Option<Token> {
```

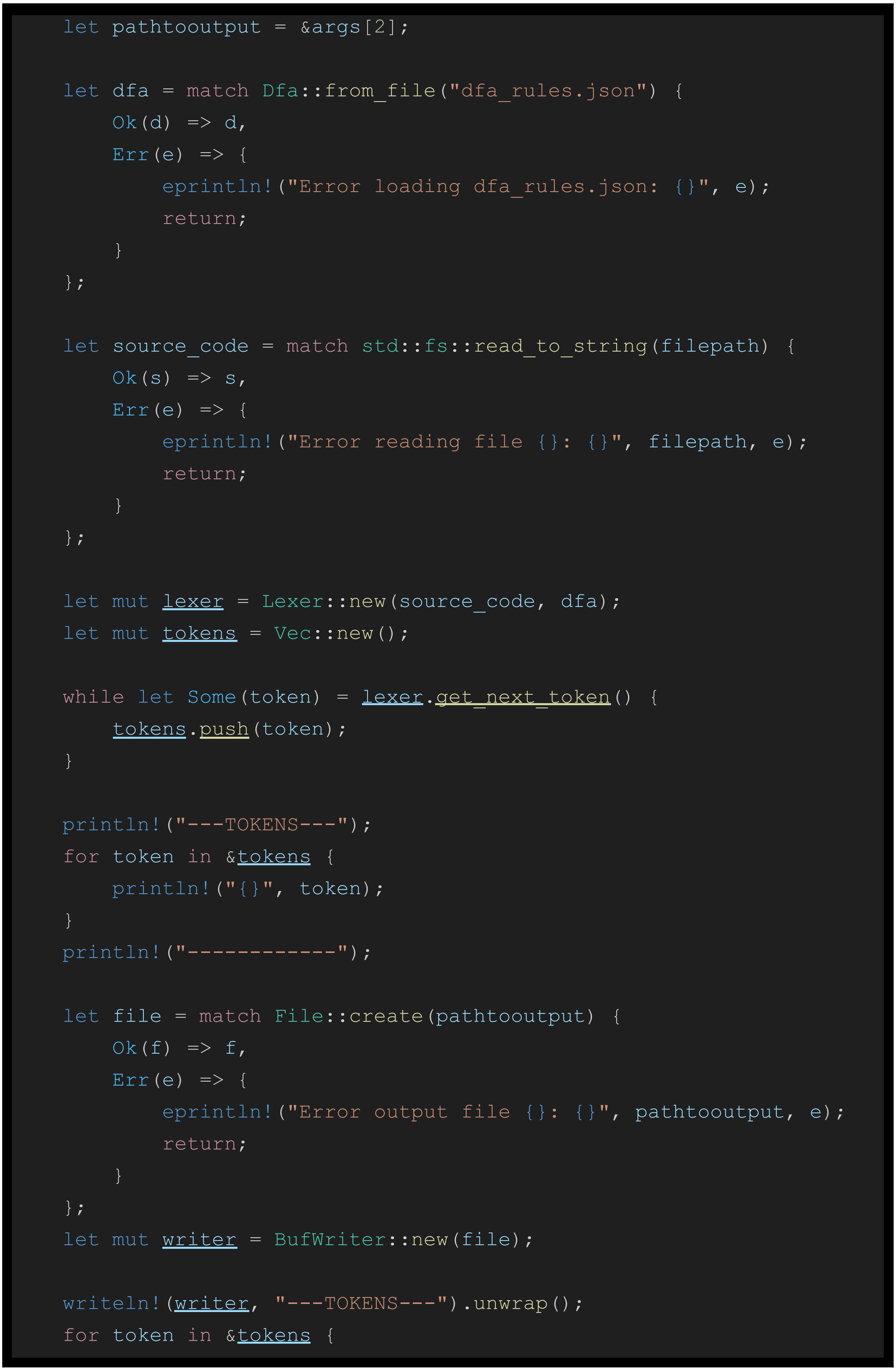


```
        if self.check_value(&TokenType::ArithmeticOperator, "*")
            || self.check_value(&TokenType::ArithmeticOperator, "/")
        {
            Some(self.advance())
        } else if self.check_value(&TokenType::Keyword, "bagi")
            || self.check_value(&TokenType::Keyword, "mod")
        {
            Some(self.advance())
        } else if self.check_value(&TokenType::LogicalOperator,
"dan") {
            Some(self.advance())
        } else {
            None
        }
    }
}
```

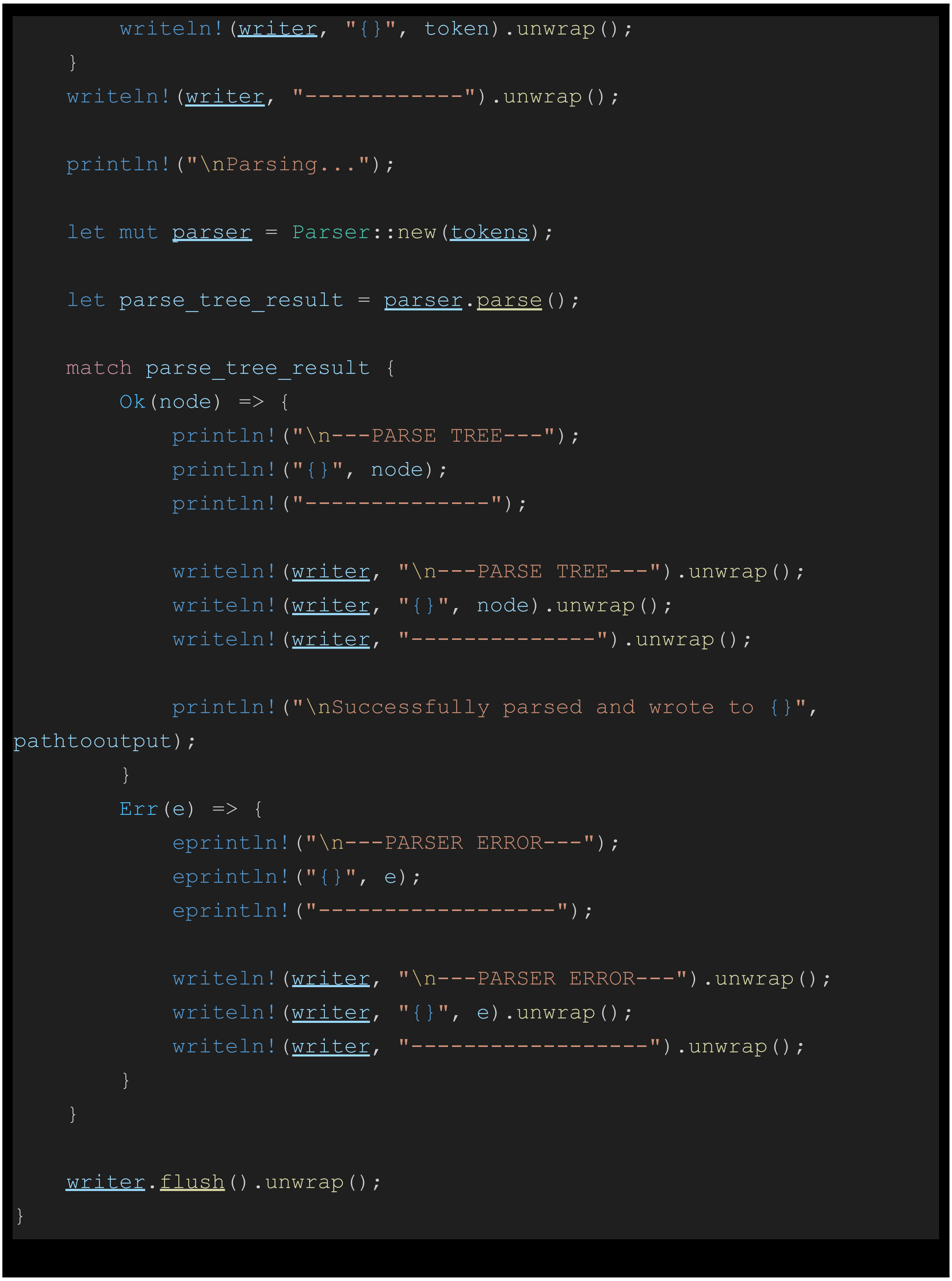
main.rs



```
use std::env;
use std::fs::File;
use std::io::{BufWriter, Write};

use crate::{dfa::Dfa, lexer::Lexer, parser::Parser};

mod dfa;
mod lexer;
mod node;
mod parser;
mod token;

fn main() {
    let args: Vec<String> = env::args().collect();
    if args.len() < 3 {
        eprintln!("Usage: {} <path_to_pascal_file> <pathtooutput>",
args[0]);
        return;
    }

    let filepath = &args[1];
```



```rust
    let pathtooutput = &args[2];

    let dfa = match Dfa::from_file("dfa_rules.json") {
        Ok(d) => d,
        Err(e) => {
            eprintln!("Error loading dfa_rules.json: {}", e);
            return;
        }
    };

    let source_code = match std::fs::read_to_string(filepath) {
        Ok(s) => s,
        Err(e) => {
            eprintln!("Error reading file {}: {}", filepath, e);
            return;
        }
    };

    let mut lexer = Lexer::new(source_code, dfa);
    let mut tokens = Vec::new();

    while let Some(token) = lexer.get_next_token() {
        tokens.push(token);
    }

    println!("---TOKENS---");
    for token in &tokens {
        println!("{}", token);
    }
    println!("------------");

    let file = match File::create(pathtooutput) {
        Ok(f) => f,
        Err(e) => {
            eprintln!("Error output file {}: {}", pathtooutput, e);
            return;
        }
    };
    let mut writer = BufWriter::new(file);

    writeln!(writer, "---TOKENS---").unwrap();
    for token in &tokens {
```



```
        writeln!(writer, "{}", token).unwrap();
    }
    writeln!(writer, "------------").unwrap();

    println!("\nParsing...");

    let mut parser = Parser::new(tokens);

    let parse_tree_result = parser.parse();

    match parse_tree_result {
        Ok(node) => {
            println!("\n---PARSE TREE---");
            println!("{}", node);
            println!("--------------");

            writeln!(writer, "\n---PARSE TREE---").unwrap();
            writeln!(writer, "{}", node).unwrap();
            writeln!(writer, "--------------").unwrap();

            println!("\nSuccessfully parsed and wrote to {}",
pathtooutput);
        }
        Err(e) => {
            eprintln!("\n---PARSER ERROR---");
            eprintln!("{}", e);
            eprintln!("------------------");

            writeln!(writer, "\n---PARSER ERROR---").unwrap();
            writeln!(writer, "{}", e).unwrap();
            writeln!(writer, "------------------").unwrap();
        }
    }

    writer.flush().unwrap();
}
```

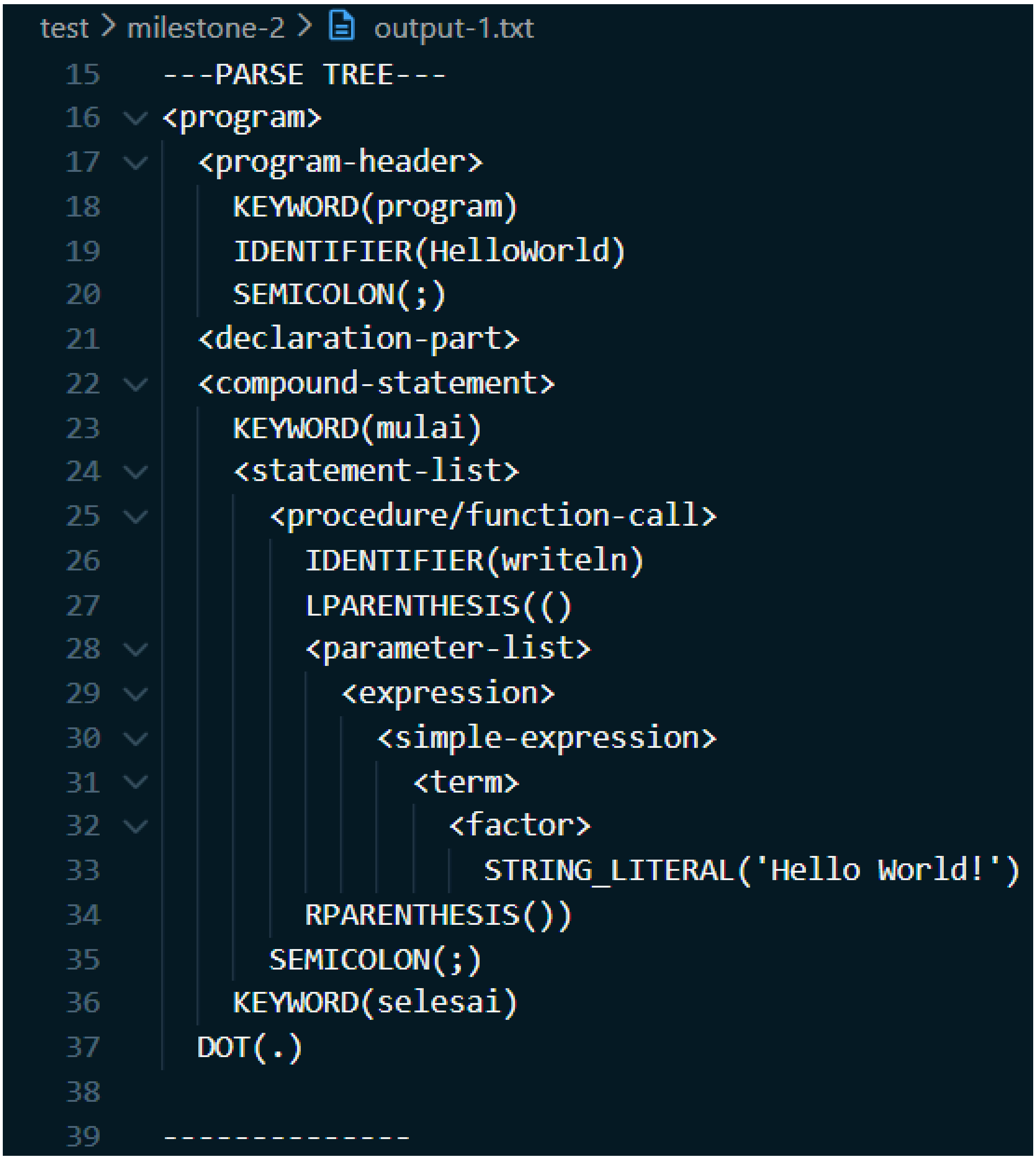
Pengujian

input-1.pas

input

```
program HelloWorld;

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

output



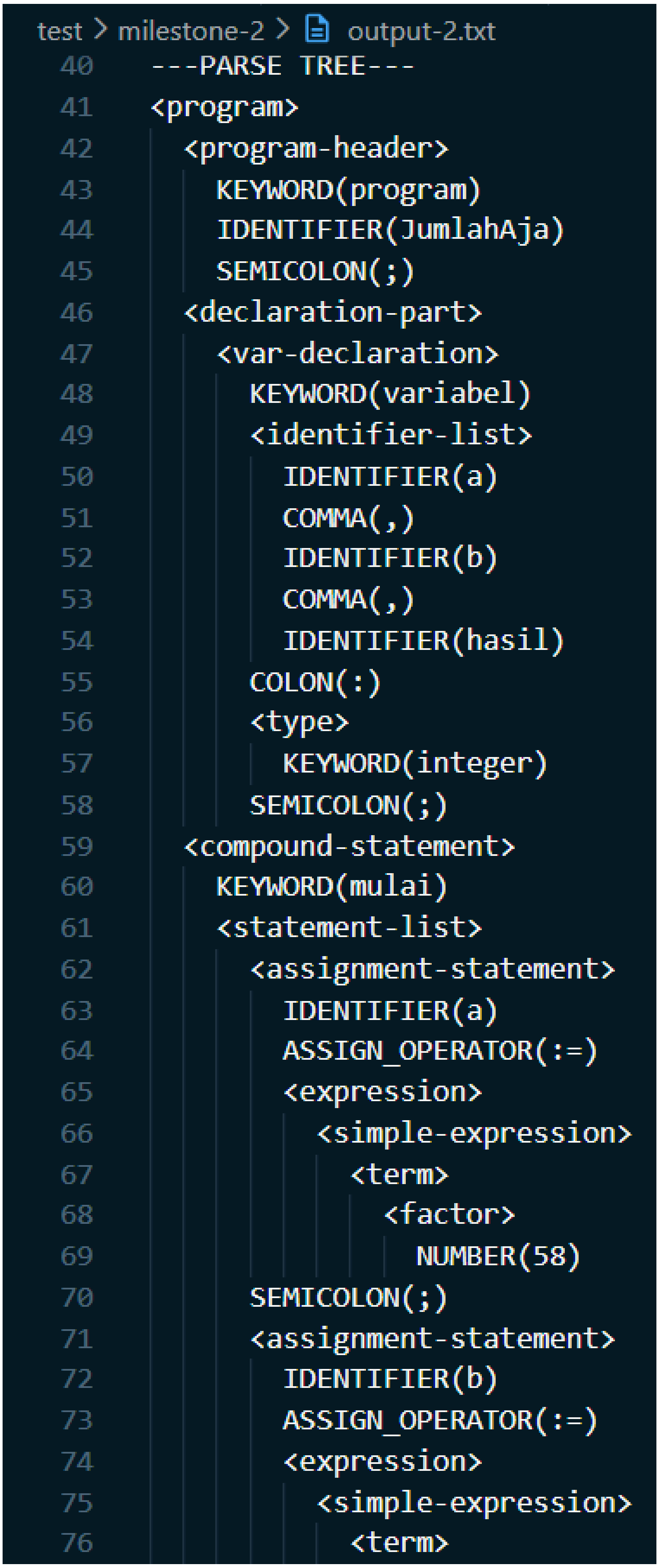
```
---PARSE TREE---
<program>
  <program-header>
    KEYWORD(program)
    IDENTIFIER(HelloWorld)
    SEMICOLON(;)
  <declaration-part>
  <compound-statement>
    KEYWORD(mulai)
    <statement-list>
      <procedure/function-call>
        IDENTIFIER(writeln)
        LPARENTHESIS(()
        <parameter-list>
          <expression>
            <simple-expression>
              <term>
                <factor>
                  STRING_LITERAL('Hello World!')
        RPARENTHESIS())
      SEMICOLON(;)
    KEYWORD(selesai)
  DOT(.)

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

input-2.pas

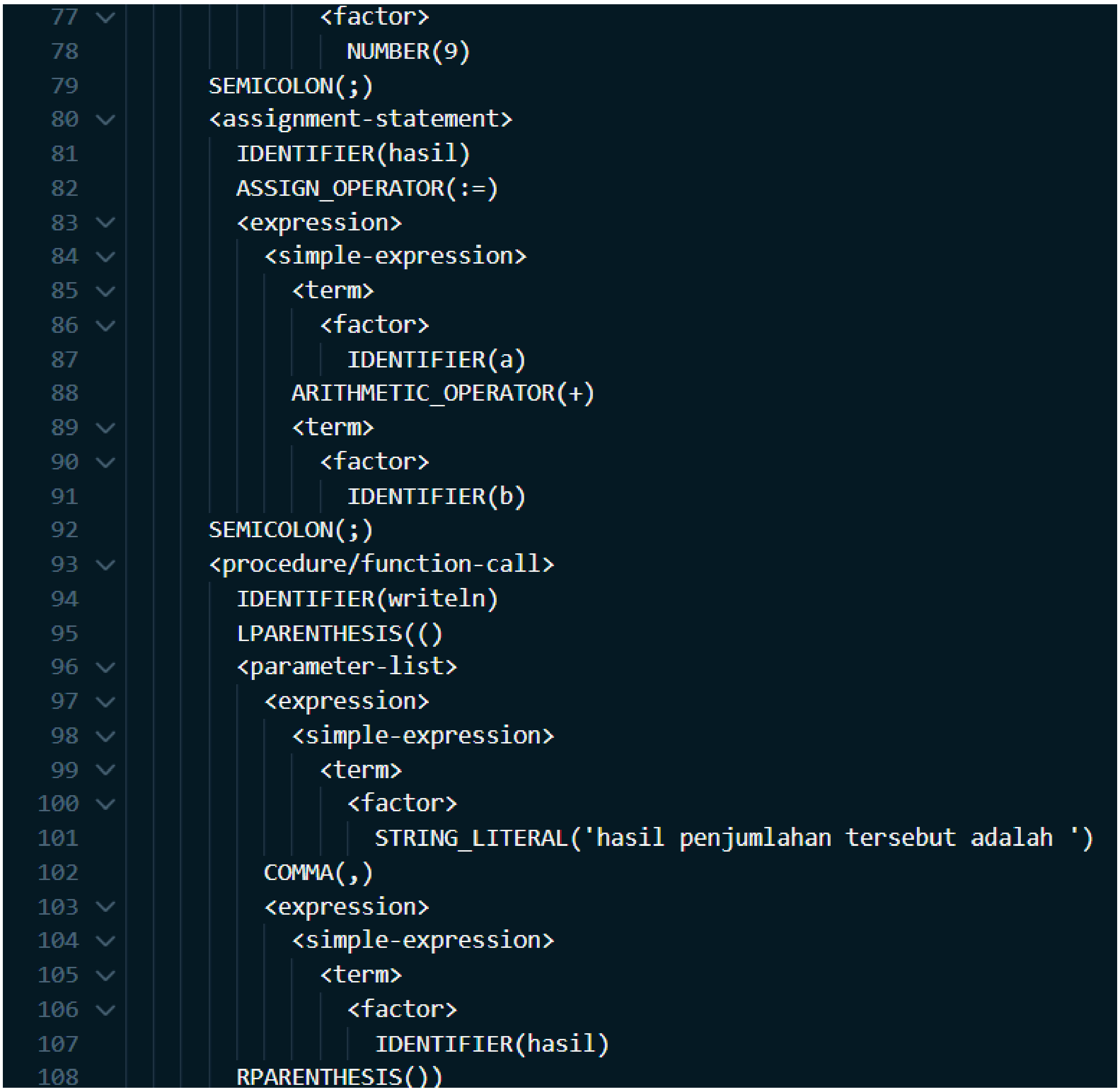
input

```
program JumlahAja;

variabel
    a, b, hasil: integer;

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

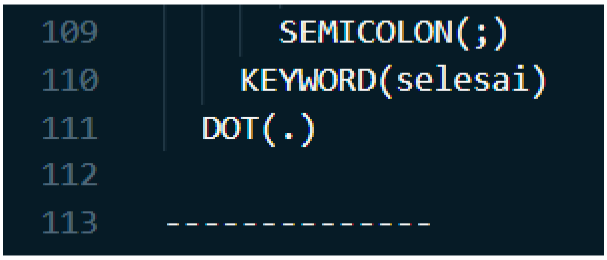
output



```
---PARSE TREE---
<program>
  <program-header>
    KEYWORD(program)
    IDENTIFIER(JumlahAja)
    SEMICOLON(;)
  <declaration-part>
    <var-declaration>
      KEYWORD(variabel)
      <identifier-list>
        IDENTIFIER(a)
        COMMA(,)
        IDENTIFIER(b)
        COMMA(,)
        IDENTIFIER(hasil)
      COLON(:)
      <type>
        KEYWORD(integer)
      SEMICOLON(;)
  <compound-statement>
    KEYWORD(mulai)
    <statement-list>
      <assignment-statement>
        IDENTIFIER(a)
        ASSIGN_OPERATOR(:=)
        <expression>
          <simple-expression>
            <term>
              <factor>
                NUMBER(58)
      SEMICOLON(;)
      <assignment-statement>
        IDENTIFIER(b)
        ASSIGN_OPERATOR(:=)
        <expression>
          <simple-expression>
            <term>
```



```
              <factor>
                NUMBER(9)
      SEMICOLON(;)
      <assignment-statement>
        IDENTIFIER(hasil)
        ASSIGN_OPERATOR(:=)
        <expression>
          <simple-expression>
            <term>
              <factor>
                IDENTIFIER(a)
            ARITHMETIC_OPERATOR(+)
            <term>
              <factor>
                IDENTIFIER(b)
      SEMICOLON(;)
      <procedure/function-call>
        IDENTIFIER(writeln)
        LPARENTHESIS(()
        <parameter-list>
          <expression>
            <simple-expression>
              <term>
                <factor>
                  STRING_LITERAL('hasil penjumlahan tersebut adalah ')
          COMMA(,)
          <expression>
            <simple-expression>
              <term>
                <factor>
                  IDENTIFIER(hasil)
        RPARENTHESIS())
```



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

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

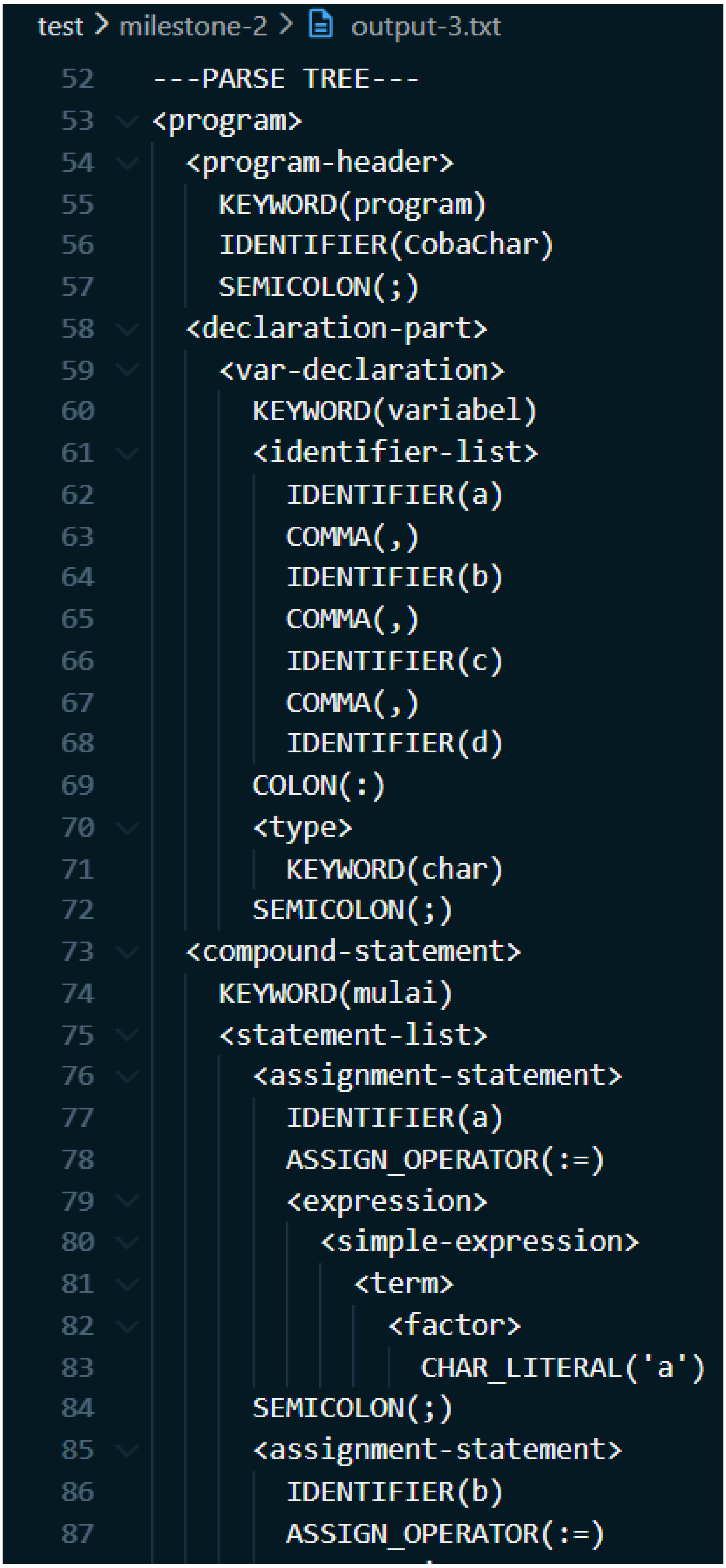
input-3.pas

input

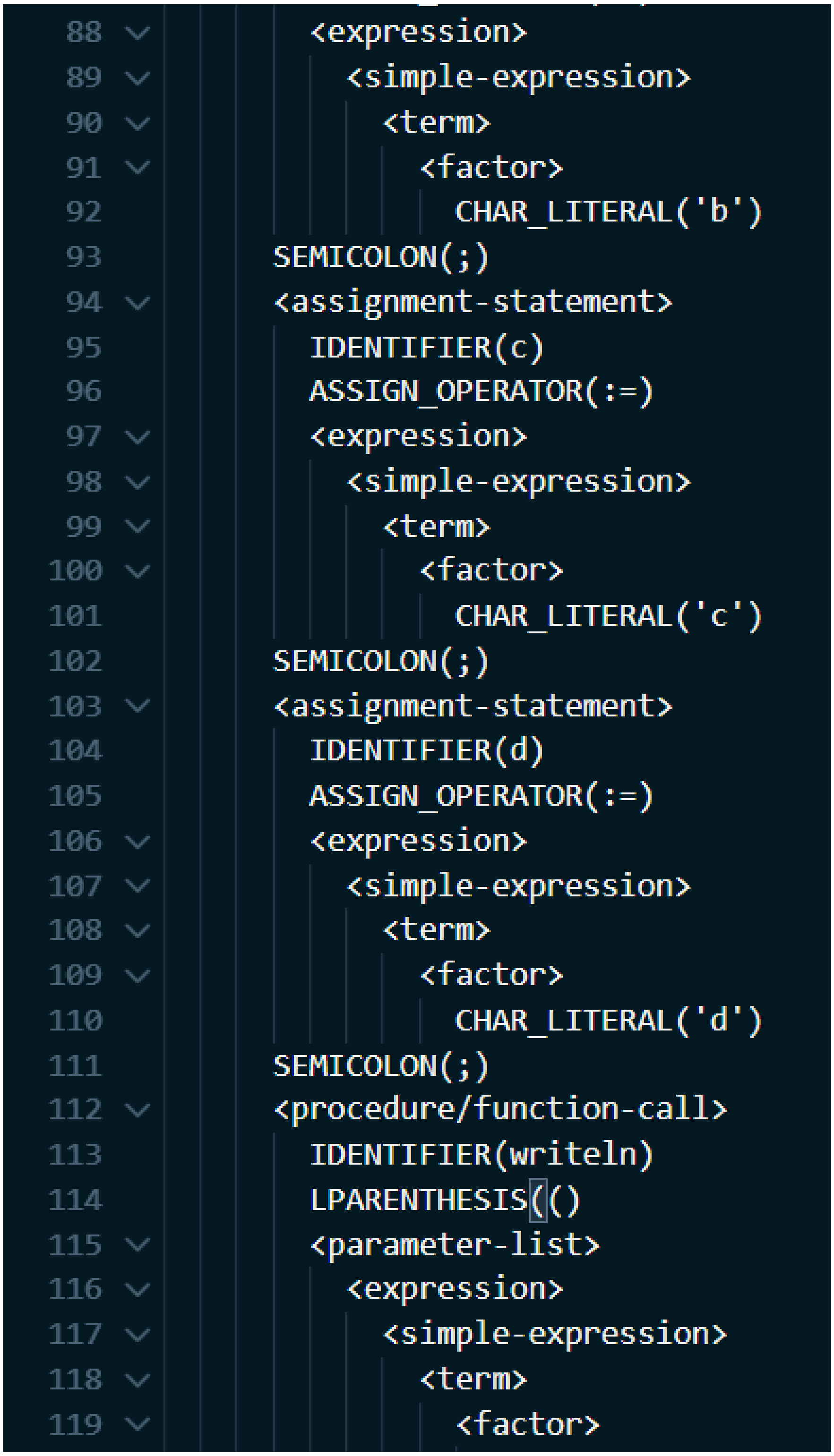
```
program CobaChar;

variabel
    a, b, c, d: char;
```

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

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

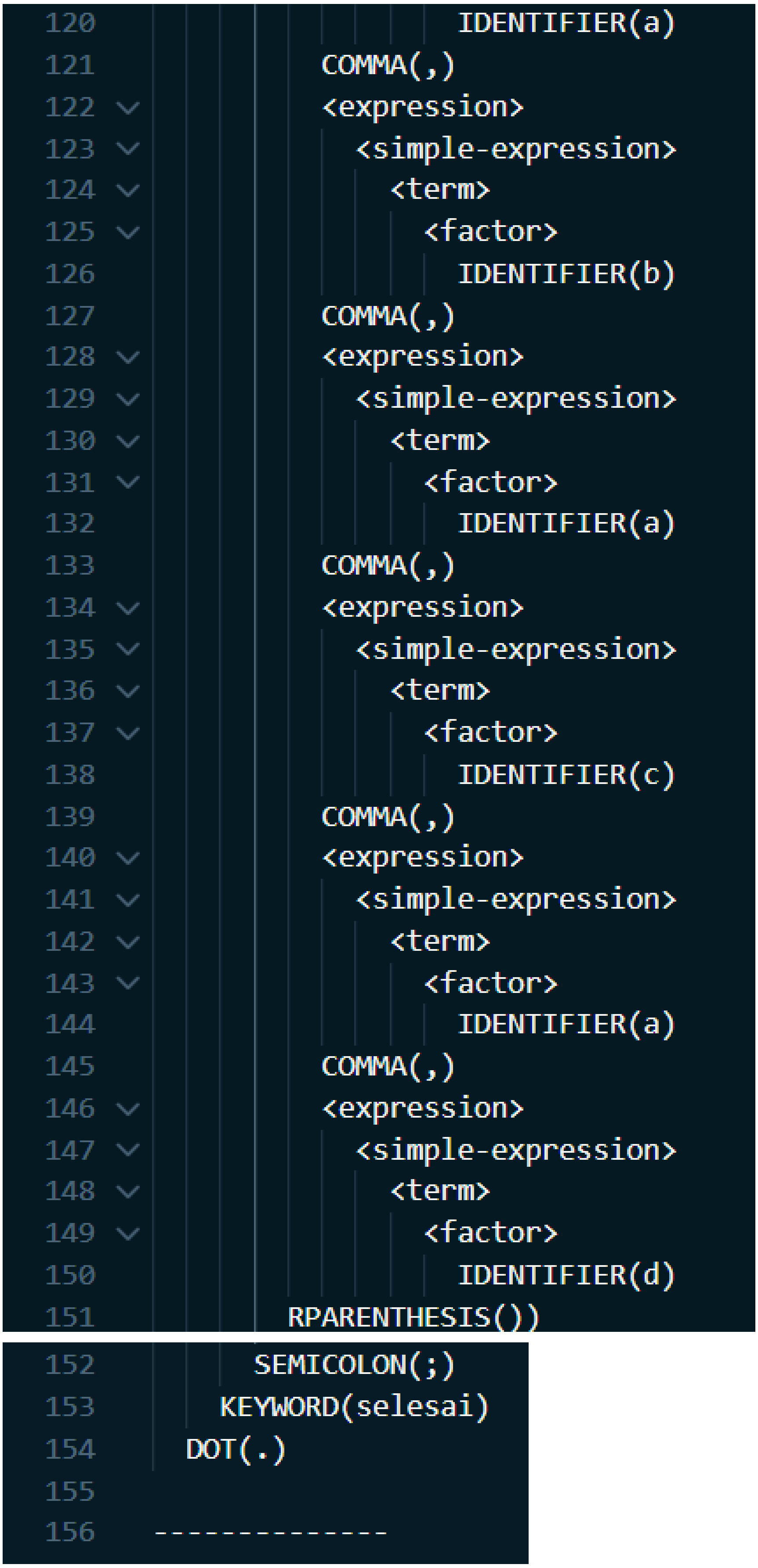
output



```
---PARSE TREE---
<program>
  <program-header>
    KEYWORD(program)
    IDENTIFIER(CobaChar)
    SEMICOLON(;)
  <declaration-part>
    <var-declaration>
      KEYWORD(variabel)
      <identifier-list>
        IDENTIFIER(a)
        COMMA(,)
        IDENTIFIER(b)
        COMMA(,)
        IDENTIFIER(c)
        COMMA(,)
        IDENTIFIER(d)
      COLON(:)
      <type>
        KEYWORD(char)
      SEMICOLON(;)
  <compound-statement>
    KEYWORD(mulai)
    <statement-list>
      <assignment-statement>
        IDENTIFIER(a)
        ASSIGN_OPERATOR(:=)
        <expression>
          <simple-expression>
            <term>
              <factor>
                CHAR_LITERAL('a')
      SEMICOLON(;)
      <assignment-statement>
        IDENTIFIER(b)
        ASSIGN_OPERATOR(:=)
```



```
  <expression>
    <simple-expression>
      <term>
        <factor>
          CHAR_LITERAL('b')
SEMICOLON(;)
<assignment-statement>
  IDENTIFIER(c)
  ASSIGN_OPERATOR(:=)
  <expression>
    <simple-expression>
      <term>
        <factor>
          CHAR_LITERAL('c')
SEMICOLON(;)
<assignment-statement>
  IDENTIFIER(d)
  ASSIGN_OPERATOR(:=)
  <expression>
    <simple-expression>
      <term>
        <factor>
          CHAR_LITERAL('d')
SEMICOLON(;)
<procedure/function-call>
  IDENTIFIER(writeln)
  LPARENTHESIS(()
  <parameter-list>
    <expression>
      <simple-expression>
        <term>
          <factor>
```



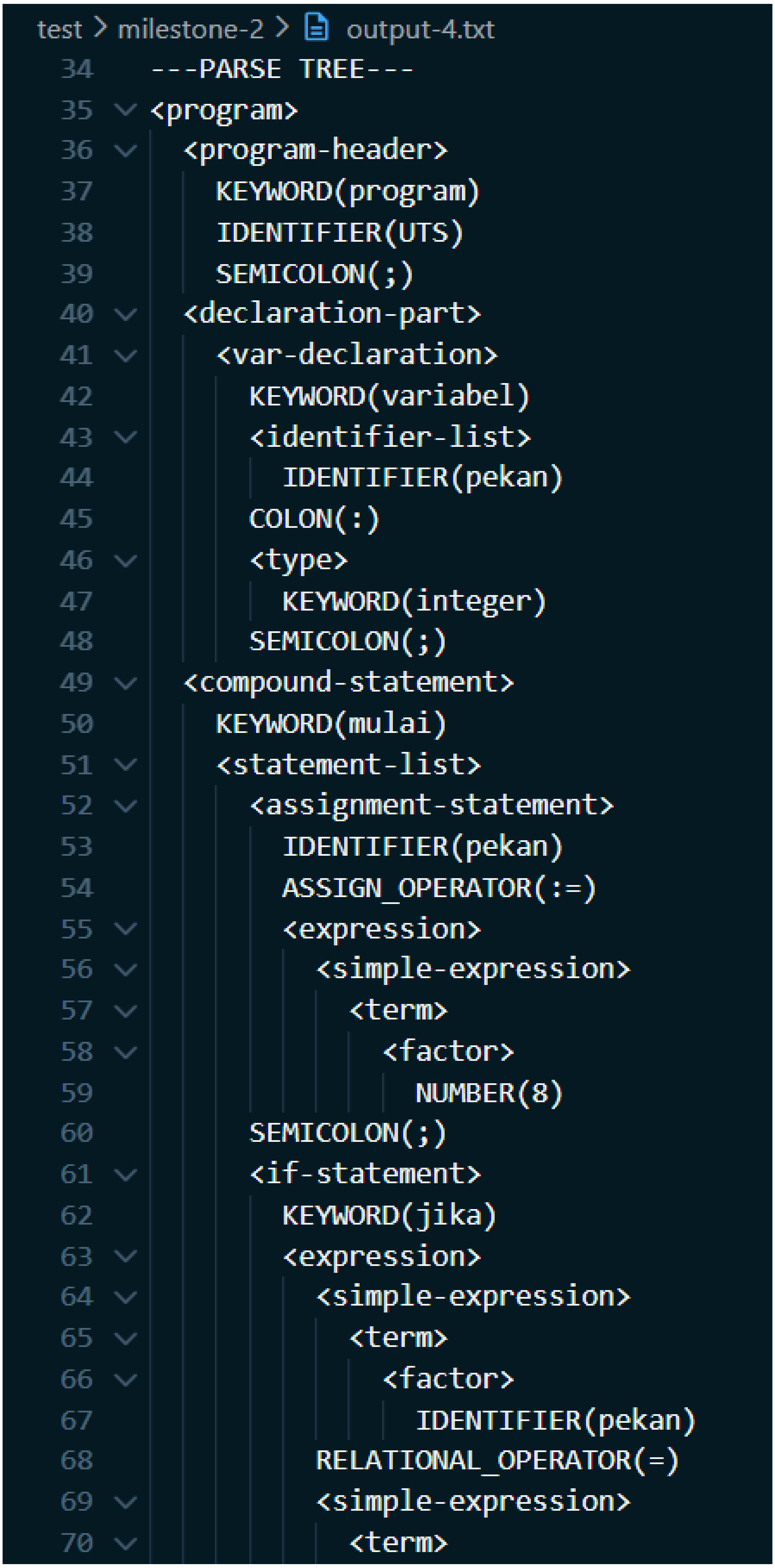
```
                  IDENTIFIER(a)
          COMMA(,)
          <expression>
            <simple-expression>
              <term>
                <factor>
                  IDENTIFIER(b)
          COMMA(,)
          <expression>
            <simple-expression>
              <term>
                <factor>
                  IDENTIFIER(a)
          COMMA(,)
          <expression>
            <simple-expression>
              <term>
                <factor>
                  IDENTIFIER(c)
          COMMA(,)
          <expression>
            <simple-expression>
              <term>
                <factor>
                  IDENTIFIER(a)
          COMMA(,)
          <expression>
            <simple-expression>
              <term>
                <factor>
                  IDENTIFIER(d)
        RPARENTHESIS())
      SEMICOLON(;)
    KEYWORD(selesai)
  DOT(.)

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

input-4.pas

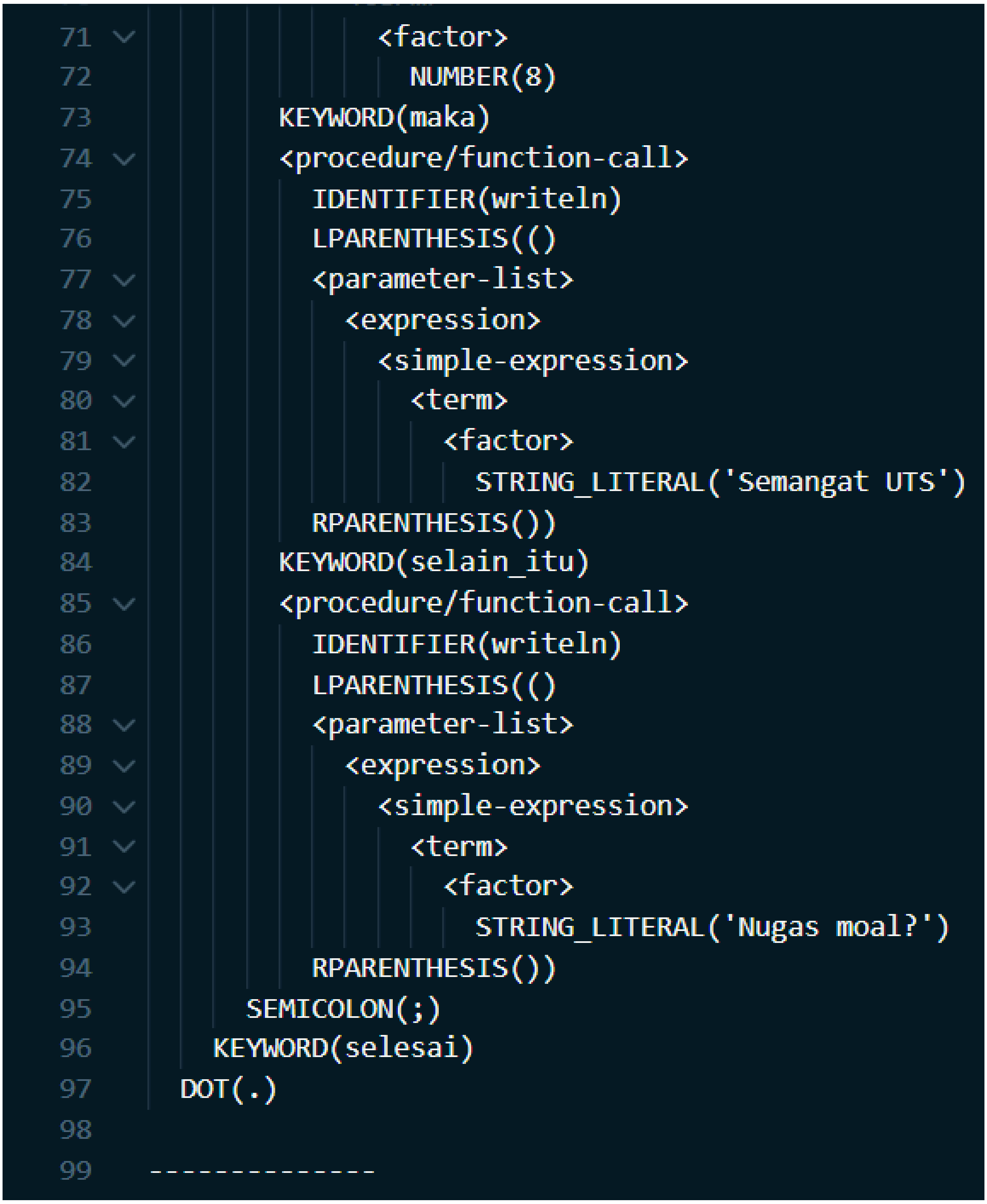
input

```
program UTS;

variabel
    pekan: integer;

mulai
    pekan := 8;

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

output



```
---PARSE TREE---
<program>
  <program-header>
    KEYWORD(program)
    IDENTIFIER(UTS)
    SEMICOLON(;)
  <declaration-part>
    <var-declaration>
      KEYWORD(variabel)
      <identifier-list>
        IDENTIFIER(pekan)
      COLON(:)
      <type>
        KEYWORD(integer)
      SEMICOLON(;)
  <compound-statement>
    KEYWORD(mulai)
    <statement-list>
      <assignment-statement>
        IDENTIFIER(pekan)
        ASSIGN_OPERATOR(:=)
        <expression>
          <simple-expression>
            <term>
              <factor>
                NUMBER(8)
      SEMICOLON(;)
      <if-statement>
        KEYWORD(jika)
        <expression>
          <simple-expression>
            <term>
              <factor>
                IDENTIFIER(pekan)
          RELATIONAL_OPERATOR(=)
          <simple-expression>
            <term>
```



```
              <factor>
                NUMBER(8)
        KEYWORD(maka)
        <procedure/function-call>
          IDENTIFIER(writeln)
          LPARENTHESIS(()
          <parameter-list>
            <expression>
              <simple-expression>
                <term>
                  <factor>
                    STRING_LITERAL('Semangat UTS')
          RPARENTHESIS())
        KEYWORD(selain_itu)
        <procedure/function-call>
          IDENTIFIER(writeln)
          LPARENTHESIS(()
          <parameter-list>
            <expression>
              <simple-expression>
                <term>
                  <factor>
                    STRING_LITERAL('Nugas moal?')
          RPARENTHESIS())
      SEMICOLON(;)
    KEYWORD(selesai)
  DOT(.)

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

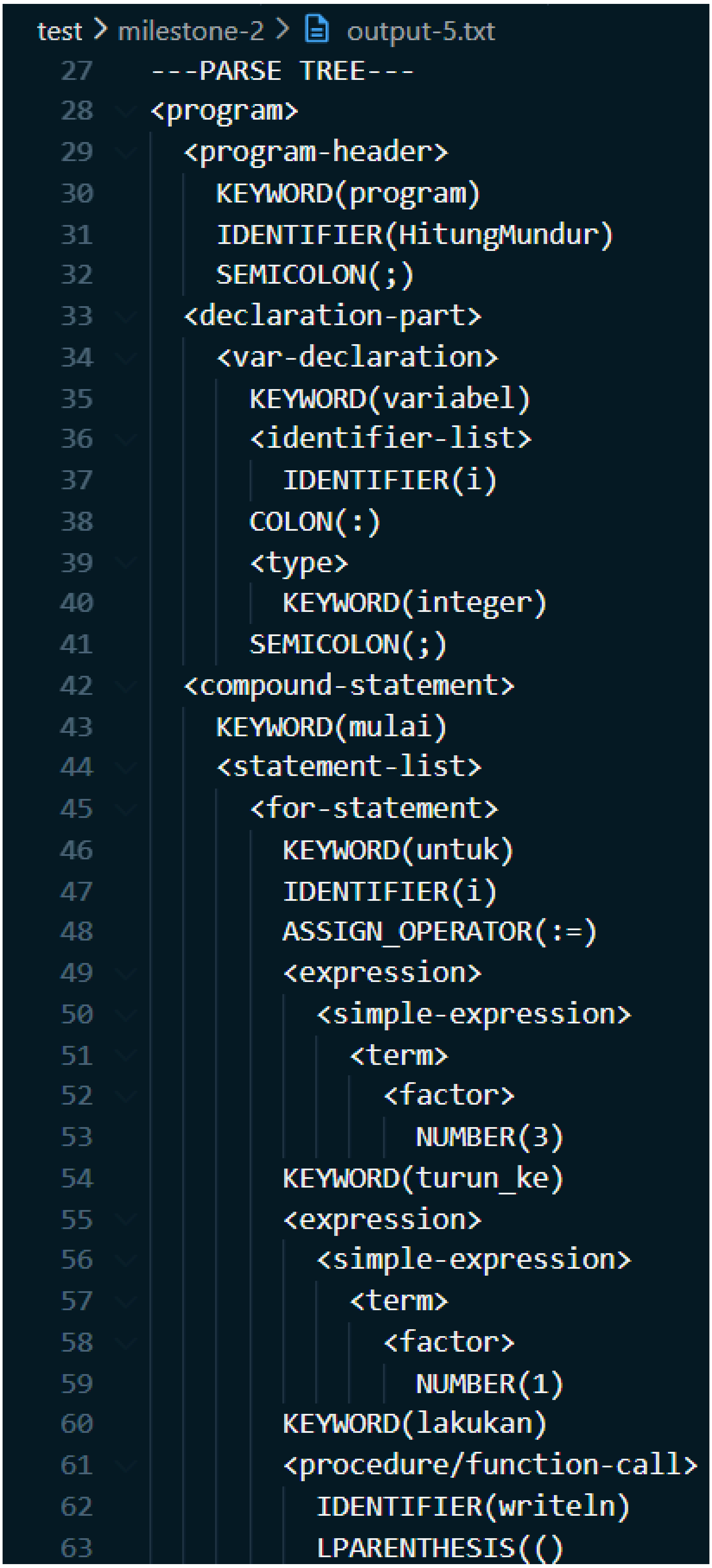
input-5.pas

input

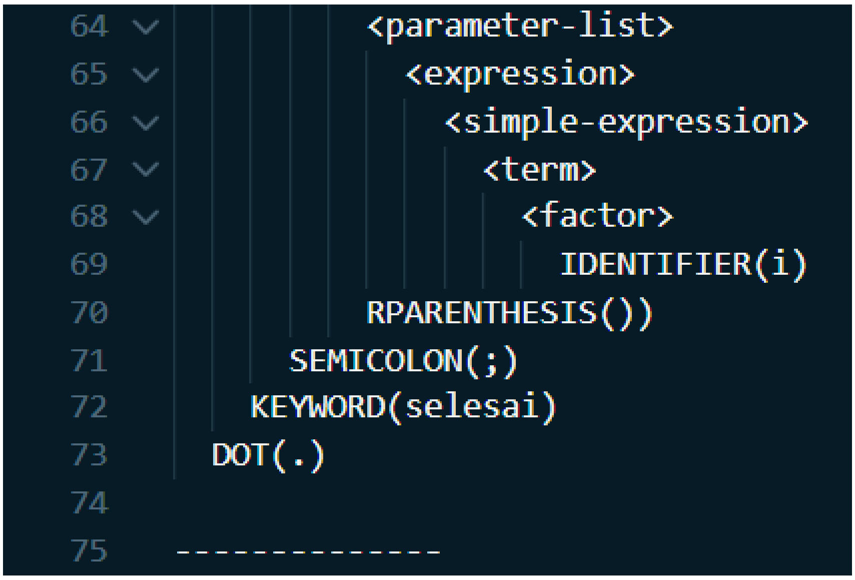
```
program HitungMundur;

variabel
```

```
    i: integer;

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

output



```
---PARSE TREE---
<program>
  <program-header>
    KEYWORD(program)
    IDENTIFIER(HitungMundur)
    SEMICOLON(;)
  <declaration-part>
    <var-declaration>
      KEYWORD(variabel)
      <identifier-list>
        IDENTIFIER(i)
      COLON(:)
      <type>
        KEYWORD(integer)
      SEMICOLON(;)
  <compound-statement>
    KEYWORD(mulai)
    <statement-list>
      <for-statement>
        KEYWORD(untuk)
        IDENTIFIER(i)
        ASSIGN_OPERATOR(:=)
        <expression>
          <simple-expression>
            <term>
              <factor>
                NUMBER(3)
        KEYWORD(turun_ke)
        <expression>
          <simple-expression>
            <term>
              <factor>
                NUMBER(1)
        KEYWORD(lakukan)
        <procedure/function-call>
          IDENTIFIER(writeln)
          LPARENTHESIS(()
```



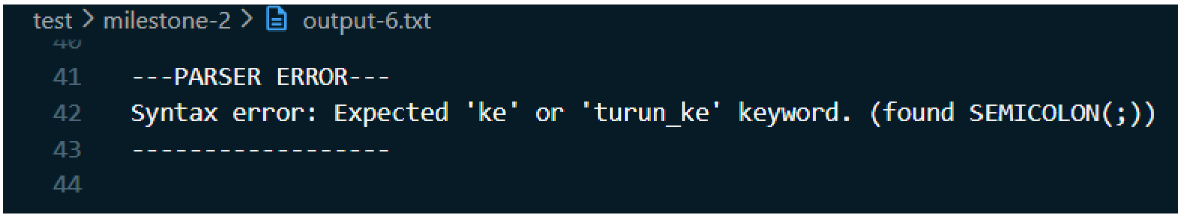
```
          <parameter-list>
            <expression>
              <simple-expression>
                <term>
                  <factor>
                    IDENTIFIER(i)
          RPARENTHESIS())
      SEMICOLON(;)
    KEYWORD(selesai)
  DOT(.)

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

input-6.pas

input

```
program UTS;

variabel
  pekan, i: integer;

mulai
  pekan := 8;

  jika pekan = 8 maka
    writeln('Semangat UTS')
  selain_itu
    writeln('Nugas moal?');
  untuk i := 3;
selesai.
```

output



```
---PARSER ERROR---
Syntax error: Expected 'ke' or 'turun_ke' keyword. (found SEMICOLON(;))
------------------
```

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

```
---PARSE TREE---
<program>
  <program-header>
    KEYWORD(program)
    IDENTIFIER(TestAllTokens)
    SEMICOLON(;)
  <declaration-part>
    <var-declaration>
      KEYWORD(variabel)
      <identifier-list>
        IDENTIFIER(my_integer)
        COMMA(,)
        IDENTIFIER(another_var)
      COLON(:)
      <type>
        KEYWORD(integer)
      SEMICOLON(;)
      <identifier-list>
        IDENTIFIER(a_real_number)
      COLON(:)
      <type>
        KEYWORD(real)
      SEMICOLON(;)
      <identifier-list>
        IDENTIFIER(is_done)
      COLON(:)
      <type>
        KEYWORD(boolean)
      SEMICOLON(;)
      <identifier-list>
        IDENTIFIER(my_char)
      COLON(:)
      <type>
        KEYWORD(char)
      SEMICOLON(;)
    <const-declaration>
      KEYWORD(konstanta)
      IDENTIFIER(PI)
      RELATIONAL_OPERATOR(=)
      <expression>
        <simple-expression>
          <term>
            <factor>
              NUMBER(3.14159)
      SEMICOLON(;)
    <type-declaration>
      KEYWORD(tipe)
      IDENTIFIER(Numbers)
      RELATIONAL_OPERATOR(=)
      <type>
        <array-type>
```

```
          KEYWORD(larik)
          LBRACKET([)
          <range>
            <expression>
              <simple-expression>
                <term>
                  <factor>
                    NUMBER(1)
            RANGE_OPERATOR(..)
            <expression>
              <simple-expression>
                <term>
                  <factor>
                    NUMBER(10)
          RBRACKET(])
          KEYWORD(dari)
          <type>
            KEYWORD(integer)
      SEMICOLON(;)
  <compound-statement>
    KEYWORD(mulai)
    <statement-list>
      <assignment-statement>
        IDENTIFIER(my_integer)
        ASSIGN_OPERATOR(:=)
        <expression>
          <simple-expression>
            <term>
              <factor>
                NUMBER(100)
      SEMICOLON(;)
      <assignment-statement>
        IDENTIFIER(another_var)
        ASSIGN_OPERATOR(:=)
        <expression>
          <simple-expression>
            <term>
              <factor>
                IDENTIFIER(my_integer)
            ARITHMETIC_OPERATOR(+)
            <term>
              <factor>
                NUMBER(20)
      SEMICOLON(;)
      <assignment-statement>
        IDENTIFIER(a_real_number)
        ASSIGN_OPERATOR(:=)
        <expression>
          <simple-expression>
            <term>
              <factor>
                IDENTIFIER(my_integer)
              ARITHMETIC_OPERATOR(/)
              <factor>
                NUMBER(3.0)
```

```
SEMICOLON(;)
<if-statement>
  KEYWORD(jika)
  <expression>
    <simple-expression>
      <term>
        <factor>
          LPARENTHESIS(()
          <expression>
            <simple-expression>
              <term>
                <factor>
                  IDENTIFIER(my_integer)
            RELATIONAL_OPERATOR(>)
            <simple-expression>
              <term>
                <factor>
                  NUMBER(50)
          RPARENTHESIS())
        LOGICAL_OPERATOR(dan)
        <factor>
          LPARENTHESIS(()
          <expression>
            <simple-expression>
              <term>
                <factor>
                  IDENTIFIER(another_var)
            RELATIONAL_OPERATOR(<>)
            <simple-expression>
              <term>
                <factor>
                  NUMBER(104)
          RPARENTHESIS())
  KEYWORD(maka)
  <compound-statement>
    KEYWORD(mulai)
    <statement-list>
      <assignment-statement>
        IDENTIFIER(is_done)
        ASSIGN_OPERATOR(:=)
        <expression>
          <simple-expression>
            <term>
              <factor>
                KEYWORD(true)
      SEMICOLON(;)
    KEYWORD(selesai)
  KEYWORD(selain_itu)
  <compound-statement>
    KEYWORD(mulai)
    <statement-list>
      <assignment-statement>
        IDENTIFIER(is_done)
        ASSIGN_OPERATOR(:=)
        <expression>
```

```
                <simple-expression>
                  <term>
                    <factor>
                      KEYWORD(false)
            SEMICOLON(;)
          KEYWORD(selesai)
      SEMICOLON(;)
      <assignment-statement>
        IDENTIFIER(my_char)
        ASSIGN_OPERATOR(:=)
        <expression>
          <simple-expression>
            <term>
              <factor>
                CHAR_LITERAL('A')
      SEMICOLON(;)
      <procedure/function-call>
        IDENTIFIER(writeln)
        LPARENTHESIS(()
        <parameter-list>
          <expression>
            <simple-expression>
              <term>
                <factor>
                  STRING_LITERAL('This is a test string
literal.')
        RPARENTHESIS())
      SEMICOLON(;)
      <if-statement>
        KEYWORD(jika)
        <expression>
          <simple-expression>
            <term>
              <factor>
                IDENTIFIER(another_var)
          RELATIONAL_OPERATOR(<=)
          <simple-expression>
            <term>
              <factor>
                NUMBER(105)
        KEYWORD(maka)
        <procedure/function-call>
          IDENTIFIER(writeln)
          LPARENTHESIS(()
          <parameter-list>
            <expression>
              <simple-expression>
                <term>
                  <factor>
                    STRING_LITERAL('Less than or equal')
          RPARENTHESIS())
      SEMICOLON(;)
    KEYWORD(selesai)
  DOT(.)
```

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

Kesimpulan

Parser menjadi komponen penting untuk compiler atau interpreter yang bertugas memeriksa urutan token dari lexer dan menentukan apakah token-token tersebut membentuk struktur sintaksis yang valid sesuai dengan aturan tata bahasa (*context-free grammar*). Parser menentukan apakah input dari user sudah sesuai atau ada kesalahan dalam sintaksis sesuai dengan bahasa yang digunakan.

Saran

- Membuat grammar yang sesuai dan mudah dimengerti untuk memudahkan implementasi

Lampiran

Link Release:

Pembagian Tugas:

NIM	Tugas	Persentase Kontribusi
13523128	Parser dan laporan	25%
13523145	parser, test, dan laporan	25%
13523146	laporan	20%
13523152	parser, test, dan laporan	30%

Grammar yang Digunakan

No	Nama Node	Deskripsi	Aturan Produksi	Contoh
1	`<program>`	Root node merepresentasikan keseluruhan program utama Pascal-S	program → <program-header + declaration-part + compound-statement + DOT	`program Hello; variabel x: integer; mulai x := 5; selesai.`
2	`<program-header>`	Header program dengan nama	KEYWORD(program) + IDENTIFIER + SEMICOLON	`program Hello;`
3	`<declaration-part>`	Bagian deklarasi (semua opsional)	(const-declaration)* + (type-declaration)* + (var-declaration)* + (subprogram-declaration)*	`konstanta MAX = 100; variabel x: integer;`
4	`<const-declaration>`	Deklarasi konstanta (minimal 1)	KEYWORD(konstanta) + (IDENTIFIER + RELOP(=) + expression + SEMICOLON)+	`konstanta PI = 3.14; MAX = 100;`
5	`<type-declaration>`	Deklarasi tipe data baru (minimal 1)	KEYWORD(tipe) + (IDENTIFIER + RELOP(=) + type-definition + SEMICOLON)+	`tipe Range = 1..10; Matrix = larik[1..5] dari integer;`
6	`<var-declaration>`	Deklarasi variabel (minimal 1)	KEYWORD(variabel) + (identifier-list + COLON + type + SEMICOLON)+	`variabel x, y: integer; nama: char;`

7	<identifier-list>	Daftar identifier yang dipisahkan koma (minimal 1)	IDENTIFIER (COMMA + IDENTIFIER)*	a, b, c, atau x
8	<type>	Tipe data primitif atau kompleks	KEYWORD(integer | real | boolean | char) | <array-type> | IDENTIFIER	integer atau larik[1..10] dari real
9	<array-type>	Definisi tipe array	KEYWORD(larik) + LBRACKET + range + RBRACKET + KEYWORD(dari) + type	larik[1..100] dari integer
10	<range>	Rentang nilai untuk array atau subrange	expression + RANGE_OPERATOR(..) + expression	1..10 atau 'a'..'z'
11	<subprogram-declaration>	Deklarasi prosedur atau fungsi	procedure-declaration atau function-declaration	prosedur print(x: integer);
12	<procedure-declaration>	Deklarasi prosedur	KEYWORD(prosedur) + IDENTIFIER + (formal-parameter-list)? + SEMICOLON + compound-statement + SEMICOLON	prosedur cetak(n: integer); mulai writeln(n); selesai;
13	<function-declaration>	Deklarasi fungsi	KEYWORD(fungsi) + IDENTIFIER + (formal-parameter-list)? + COLON + type + SEMICOLON + declaration-part + compound-statement + SEMICOLON	fungsi tambah(a, b: integer): integer; mulai tambah := a + b; selesai;
14	<formal-parameter-list>	Daftar parameter formal (minimal 1 group)	LPARENTHESIS + identifier-list + COLON + type + (SEMICOLON + identifier-list + COLON + type)* + RPARENTHESIS	(x, y: integer; z: real)
15	<compound-statement>	Blok statement	KEYWORD(mulai) + statement-list +	mulai x := 1; y := 2; selesai

		yang diawali begin dan diakhiri end	KEYWORD(selesai)	
16	<statement-list>	Daftar statement (bisa kosong, bisa trailing semicolon)	(<assignment-statement> | <if-statement> | <while-statement> | <for-statement> | <procedure/function-call> | <compound-statement>)? + (SEMICOLON + (<assignment-statement> | <if-statement> | <while-statement> | <for-statement> | <procedure/function-call> | <compound-statement>)?)*	x := 5; y := 10; atau writeln('OK');
17	<assignment-statement>	Statement penugasan nilai ke variabel	IDENTIFIER + ASSIGN_OPERATOR(:=) + expression	x := 5 atau total := a + b * c
18	<if-statement>	Statement kondisional if-then-else	KEYWORD(jika) + <expression> + KEYWORD(maka) + (<assignment-statement> | <if-statement> | <while-statement> | <for-statement> | <procedure/function-call> | <compound-statement>) + (KEYWORD(selain_itu) + (<assignment-statement> | <if-statement> | <while-statement> | <for-statement> | <procedure/function-call> | <compound-statement>))?	jika x > 0 maka y := 1 selain_itu y := 0
19	<while-statement>	Loop dengan kondisi di awal	KEYWORD(selama) + <expression> + KEYWORD(lakukan) + (<assignment-statement> | <if-statement> |	selama x < 10 lakukan x := x + 1

			<while-statement> | <for-statement> | <procedure/function-call> | <compound-statement>)	
20	`<for-statement>`	Loop dengan counter	KEYWORD(untuk) + IDENTIFIER + ASSIGN_OPERATOR + <expression> + (KEYWORD(ke) | KEYWORD(turun_ke)) + <expression> + KEYWORD(lakukan) + (<assignment-statement> | <if-statement> | <while-statement> | <for-statement> | <procedure/function-call> | <compound-statement>)	`untuk i := 1 ke 10 lakukan writeln(i)`
21	`<procedure/function-call>`	Pemanggilan prosedur/fungsi (bisa tanpa parameter atau dengan parameter)	IDENTIFIER + (LPARENTHESIS + parameter-list + RPARENTHESIS)~~?~~	`writeln('Hello')` atau `cetak(x, y)` atau `readln`
22	`<parameter-list>`	Parameter aktual (minimal 1 jika ada kurung)	expression (COMMA + expression)*	`'Result', total, 100` atau `x + 5, y * 2`
23	`<expression>`	Ekspresi dengan operator relasi opsional	simple-expression (relational-operator + simple-expression)?	`x + 5` atau `a > b` atau `(x + y) = 10`
24	`<simple-expression>`	Ekspresi dengan unary +/- opsional	(ARITHMETIC_OPERATOR(+/-))? term (additive-operator + term)*	`5 + 3 - 2` atau `a atau b` atau `-x + 10`
25	`<term>`	Ekspresi dengan prioritas lebih tinggi	factor (multiplicative-operator + factor)*	`x * y` atau `a bagi b` atau `p dan q`
26	`<factor>`	Unit terkecil	IDENTIFIER / NUMBER /	`x` atau `42` atau

		ekspresi (identifier bisa jadi function call)	CHAR_LITERAL / STRING_LITERAL / (LPARENTHESIS + expression + RPARENTHESIS) / LOGICAL_OPERATOR(tidak) + factor / function-call	'text' atau (x + y) atau tidak flag atau sqrt(16)
27	<relational-operator>	Operator perbandingan	=, <>, <, <=, >, >=	x = 5 atau a <> b atau y >= 10
28	<additive-operator>	Operator penjumlahan/pengurangan	+, -, KEYWORD(atau)	a + b atau x - y atau p atau q
29	<multiplicative-operator>	Operator perkalian/pembagian	*, /, KEYWORD(bagi), KEYWORD(mod), KEYWORD(dan)	a * b atau x / y atau n mod 2 atau p dan q

Referensi

“Parsing Expression”
https://craftinginterpreters.com/parsing-expressions.html