LAPORAN MILESTONE 2: SYNTAX ANALYSIS / PARSER

IF2224 Teori Bahasa Formal dan Automata

Disusun oleh:

Kelompok JYP, K-02

Noumisyifa Nabila Nareswari	13523058
M. Ghifary Komara Putra	13523066
Sabilul Huda	13523072
Diyah Susan Nugrahani	13523080
Henry Filberto Shinelo	13523108

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2025

DAFTAR ISI

A. LANDASAN TEORI

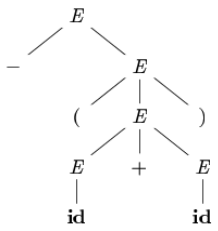
1. *Syntax Analysis*

Syntax analysis merupakan tahapan setelah *lexical analysis.* Pada *lexical analysis,* input diolah menjadi token. Token-token ini kemudian akan diperiksa pada *syntax analysis* agar tersusun sesuai dengan aturan bahasa formal *(grammar)*. Hasil utama dari proses *syntax analysis* adalah berupa struktur pohon yang menggambarkan struktur hierarki dari program. Ketika kode tidak mengikuti aturan *grammar* maka parser akan mendeteksi hal tersebut dan menangani kesalahan *syntax* tersebut sehingga dapat mendeteksi *syntax error.*

2. *Parse Tree*

Parse tree merupakan struktur pohon yang merepresentasikan proses derivasi berdasarkan aturan tata bahasa bebas konteks *(context-free grammar). Parse tree* menunjukkan bagaimana suatu kalimat dapat dibentuk dari simbol awal dengan menerapkan aturan-aturan produksi *grammar* secara bertahap. Pada penerapannya, *parse tree* tidak memperlihatkan urutan penerapan aturan derivasi melainkan menampilkan hasil akhir dalam bentuk struktur hierarkis yang menunjukkan hubungan antar simbol-simbol nonterminal dan terminal.

Setiap simpul pada *parse tree* menunjukkan bahwa pada titik tersebut sebuah aturan produksi dipakai untuk memperluas suatu nonterminal. Simpul diberi label dengan nonterminal dan menjadi *head of the production*. Anak-anak dari simpul merepresentasikan simbol-simbol yang muncul dan dituliskan dari kiri ke kanan sesuai dengan urutan dalam aturan. Daun dari kiri ke kanan merepresentasikan string terminal akhir yang dihasilkan *grammar.*



Gambar 1. Parse Tree

Sumber: Compilers - Principles, Techniques, and Tools (2006)

3. Context-Free Grammars

Context-free grammar (CFG) adalah sekumpulan aturan produksi yang secara rekursif menjelaskan bagaimana string yang valid dalam bahasa dapat dibentuk. *Context-free* memiliki arti aturan produksi yang digunakan tidak bergantung pada konteks kemunculan simbol sebelum atau sesudah. *CFG* digunakan oleh parser untuk menghasilkan *parse tree* yang kemudian akan dianalisis lebih lanjut oleh kompiler.

4. Recursive Descent Parser

Recursive Descent Parser adalah teknik *parsing top-down* yang membangun *parse tree* dari simbol awal grammar dengan memanggil fungsi secara rekursif untuk setiap nonterminal. Umumnya, implementasi dilakukan dengan cara merepresentasikan setiap aturan produksi sebagai sebuah prosedur, sehingga proses parsing mengikuti struktur grammar secara langsung. Recursive Descent Parser banyak digunakan dalam pembuatan compiler sederhana, meskipun memiliki keterbatasan seperti tidak dapat menangani grammar yang *left-recursive* tanpa transformasi tambahan.

B. PERANCANGAN DAN IMPLEMENTASI

1. Deskripsi Umum

S*yntax analysis* merupakan tahap kedua dari proses pembuatan compiler. Tahapan ini bertujuan untuk memeriksa struktur sintaksis dari deretan token yang telah dihasilkan dari tahap sebelumnya yaitu *lexical analysis.* Proses analisis sintaksis ini dilakukan dengan membangun parser menggunakan pendekatan *recursive descent parsing* yang diimplementasikan menggunakan bahasa pemrograman C++. Metode tersebut dipilih karena relatif lebih mudah diimplementasikan untuk grammar yang berbentuk *context-free grammar* sehingga parser dapat mengenali struktur gramatikal program dengan cara yang terstruktur.

Parser yang dibangun akan menganalisis urutan token berdasarkan aturan grammar dari bahasa Pascal-S dan memastikan setiap konstruksi program sesuai dengan aturan sintaksis yang telah ditentukan. Ketika susunan token memenuhi aturan *grammar,* maka parser akan menghasilkan *parse tree* yang merepresentasikan hierarki dari struktur program yang menunjukkan hubungan antar simbol nonterminal dan terminal sesuai dengan *grammar.*

2. Grammar Rules

Berikut merupakan daftar *grammar rules* yang diimplementasikan beserta dengan penjelasannya. Untuk detail implementasi kodenya dapat diakses melalui *repository* github yang terlampir pada lampiran.

No	Nama Node	Deskripsi	Aturan Produksi	Contoh
1	`<program>`	Node root yang merepresentasikan keseluruhan program Pascal-S	program → program-header + declaration-part + compound-statement + DOT	Seluruh struktur program dari awal hingga akhir
	Fungsi `<program>`, program menjadi root yang menyatukan seluruh bagian utama program. Fungsi ini memanggil tiga fungsi lain yaitu program-header, declaration-part, dan compound-state. Setelah ketiga komponen tersebut selesai diparse, fungsi akan mencocokkan token penutup berupa DOT.			
2	`<program-header>`	Bagian kepala program yang berisi nama program	KEYWORD(program) + IDENTIFIER + SEMICOLON	`program Hello;`
	Fungsi `<program-header>`, fungsi ini membuat sebuah node parse tree bertanda <program-header> dan kemudian mencocokkan token terminal yang ada. Pertama mencari token KEYWORD program dilanjut dengan IDENTIFIER dan diakhiri dengan SEMICOLON. Jika urutan token tidak sesuai, fungsi akan menghasilkan error.			
3	`<declaration-part>`	Bagian deklarasi yang dapat berisi const, type, var, procedure, function	(const-declaration)* + (type-declaration)* + (var-declaration)* + (subprogram-declaration)*	Semua deklarasi sebelum "mulai"
	Fungsi `<declaration-part>` digunakan untuk memproses seluruh bagian deklarasi dalam program. Fungsi ini membuat node `<declaration-part>` kemudian secara berulang memeriksa token saat ini untuk menentukan jenis deklarasi apa yang muncul. Jika token menunjukkan kata kunci konstanta, tipe, variabel, prosedur,			

	atau fungsi, maka parser memanggil fungsi deklarasi yang sesuai.			
4	<const-declaration>	Deklarasi konstanta	KEYWORD(konstanta) + (IDENTIFIER := value + SEMICOLON)+	konstanta MAX := 100;
	Fungsi <const-declaration> menangani parsing deklarasi konstanta. Setelah membuat node <const-declaration>, fungsi ini akan mencocokkan keyword konstanta dan mencocokkan identifier beserta valuenya dan diakhiri dengan semicolon.			
5	<type-declaration>	Deklarasi tipe data baru	KEYWORD(tipe) + (IDENTIFIER := type-definition + SEMICOLON)+	tipe Range := 1..10;
	Fungsi <type-declaration> memproses deklarasi tipe. Setelah mencocokkan keyword tipe, fungsi masuk ke loop do–while untuk menangani satu atau lebih deklarasi tipe. Di setiap iterasi, parser mencocokkan identifier sebagai nama tipe, operator assignment, kemudian memanggil type_definition() untuk mem-parse definisi tipe yang dirujuk. Lalu diakhiri dengan mencocokkan tanda SEMICOLON.			
6	<var-declaration>	Deklarasi variabel	KEYWORD(variabel) + (identifier-list + COLON + type + SEMICOLON)+	variabel a, b: integer;
	Fungsi <var-declaration> memproses deklarasi variabel. Setelah mencocokkan keyword variabel, fungsi masuk ke loop do–while untuk menangani beberapa deklarasi variabel berurutan. Di setiap iterasi, parser memanggil identifier_list() untuk membaca satu atau lebih nama variabel, mencocokkan tanda titik dua sebagai pemisah, lalu memanggil type_spec() untuk membaca spesifikasi tipe variabel tersebut. Lalu diakhiri dengan mencocokkan tanda SEMICOLON.			
7	<identifier-list>	Daftar identifier yang dipisahkan koma	IDENTIFIER (COMMA + IDENTIFIER)*	a, b, c
	Fungsi <identifier-list> digunakan untuk memproses daftar identifier yang dipisahkan koma, seperti a, b, c. Fungsi ini pertama-tama mencocokkan satu identifier, kemudian masuk ke loop while yang akan terus berjalan selama token berikutnya adalah koma. Setiap kali menemukan koma, parser menambahkannya sebagai child, maju ke token berikutnya, lalu mencocokkan identifier tambahan.			

8	<type>	Tipe data (integer, real, boolean, char, array)	KEYWORD(integer/real/boolean/char) atau array-type	integer, larik[1..10] dari integer
	Fungsi <type> menangani parsing spesifikasi tipe, baik tipe dasar maupun tipe larik. Jika token saat ini adalah salah satu keyword tipe dasar (integer, real, boolean, atau char), parser langsung menambahkannya sebagai child dan maju ke token berikutnya. Jika token berupa keyword larik, fungsi memanggil array_type() untuk memproses deklarasi tipe larik. Apabila token tidak cocok dengan keduanya, parser menampilkan pesan error.			
9	<array-type>	Definisi tipe array	KEYWORD(larik) + LBRACKET + range + RBRACKET + KEYWORD(dari) + type	larik[1..10] dari integer
	Fungsi <array-type> memproses deklarasi tipe larik yang mengikuti pola larik [range] dari <type>. Setelah mencocokkan keyword larik, parser membaca tanda [lalu memanggil range() untuk memproses batas indeks larik. Setelah tanda], parser mencocokkan keyword dari sebagai penghubung, lalu memanggil type_spec() untuk menentukan tipe elemen larik.			
10	<range>	Rentang nilai untuk array atau subrange	expression + RANGE_OPERATOR(..) + expression	1..10, 'a'..'z'
	Fungsi <range> memproses batas indeks larik yang berbentuk <expression> .. <expression>. Di dalamnya, parser memanggil expression() untuk membaca batas awal, kemudian mencocokkan token operator rentang, dan kembali memanggil expression() untuk membaca batas akhir.			
11	<subprogram-declaration>	Deklarasi prosedur atau fungsi	procedure-declaration atau function-declaration	prosedur print(x: integer);
	Fungsi <subprogram-declaration> menentukan apakah deklarasi subprogram yang sedang diparse merupakan prosedur atau fungsi. Berdasarkan nilai cur_tok.lexeme, parser memanggil procedure_declaration() jika token adalah prosedur, atau function_declaration() jika token adalah fungsi.			
12	<procedure-declaration>	Deklarasi prosedur	KEYWORD(prosedur) + IDENTIFIER	Prosedur dengan parameter

			+ (formal-parameter-list)? + SEMICOLON + block + SEMICOLON	opsional
	Fungsi `<procedure-declaration>` memproses deklarasi prosedur. Setelah mencocokkan keyword prosedur dan nama prosedur, parser memeriksa apakah terdapat tanda kurung buka; jika ada, ia memanggil formal_parameter_list() untuk memproses parameter formal. Setelah itu parser mencocokkan tanda SEMICOLON, memanggil block() untuk membaca isi prosedur, dan diakhiri dengan tanda SEMICOLON terakhir.			
13	`<function-declaration>`	Deklarasi fungsi	KEYWORD(fungsi) + IDENTIFIER + (formal-parameter-list)? + COLON + type + SEMICOLON + block + SEMICOLON	Fungsi yang mengembalikan nilai
	Fungsi `<function-declaration>` memproses deklarasi fungsi. Setelah mencocokkan keyword fungsi dan identifier sebagai nama fungsi, parser memeriksa apakah terdapat parameter formal. Jika ada, ia memanggil formal_parameter_list(). Kemudian parser mencocokkan tanda : untuk memulai spesifikasi tipe hasil, memanggil type_spec() untuk menentukan tipe fungsi, mencocokkan tanda SEMICOLON, lalu membaca isi fungsi melalui block(). Deklarasi ditutup dengan tanda SEMICOLON terakhir.			
14	`<formal-parameter-list>`	Daftar parameter formal	LPARENTHESIS + parameter-group (SEMICOLON + parameter-group)* + RPARENTHESIS	`(x, y: integer; z: real)`
	Fungsi `<formal-parameter-list>` memproses daftar parameter formal yang berada di dalam tanda kurung. Pertama, parser mencocokkan tanda kurung buka, kemudian memanggil parameter_group() untuk membaca satu kelompok parameter. Jika terdapat tanda ;, parser menganggap masih ada kelompok parameter berikutnya dan memprosesnya dalam loop. Setelah seluruh kelompok parameter selesai dibaca, parser mencocokkan tanda kurung tutup.			
15	`<compound-statement>`	Blok statement yang diawali	KEYWORD(mulai) + statement-list	`mulai ... selesai`

		begin dan diakhiri end	+ KEYWORD(selesai)	
	Fungsi <compound-statement> memproses blok pernyataan yang diawali keyword mulai dan diakhiri selesai. Setelah mencocokkan keyword pembuka, parser memanggil statement_list() untuk membaca satu atau lebih pernyataan di dalam blok. Terakhir, parser mencocokkan keyword penutup selesai.			
16	<statement-list>	Daftar statement yang dipisahkan semicolon	statement (SEMICOLON + statement)*	Urutan statement dalam blok
	Fungsi <statement-list> memproses satu atau lebih pernyataan yang dipisahkan oleh tanda SEMICOLON. Pertama, parser mengecek apakah token saat ini dapat memulai sebuah pernyataan; jika ya, fungsi memanggil statement() untuk memprosesnya. Selanjutnya, selama parser menemukan token SEMICOLON, token tersebut ditambahkan sebagai child dan parser kembali memeriksa apakah token berikutnya merupakan awal pernyataan lain.			
17	<assignment-statement>	Statement penugasan nilai ke variabel	IDENTIFIER + ASSIGN_OPERATOR(:=) + expression	`x := 5, y := x + 10`
	Fungsi <assignment-statement> memproses pernyataan penugasan dengan format `identifier := expression`. Parser mencocokkan identifier sebagai sisi kiri penugasan, kemudian mencocokkan operator assignment, dan akhirnya memanggil expression() untuk membaca ekspresi di sisi kanan.			
18	<if-statement>	Statement kondisional	KEYWORD(jika) + expression + KEYWORD(maka) + statement + (KEYWORD(selain-itu) + statement)?	`jika x > 0 maka y := 1 selain-itu y := 0`
	Fungsi <if-statement> memproses struktur percabangan yang mengikuti pola jika <expression> maka <statement> (selain-itu <statement>)?. Parser terlebih dahulu mencocokkan keyword jika, membaca ekspresi kondisinya, lalu mencocokkan keyword maka dan memproses statement yang menjadi cabang utama. Apabila terdapat bagian opsional selain-itu, parser mencocokkan tanda SEMICOLON, mencocokkan keyword selain-itu, lalu memproses statement cabang alternatif.			
19	<while-statem	Perulangan	KEYWORD(selama	`selama x < 10`

	ent>	dengan kondisi di awal	) + expression + KEYWORD(lakukan) + statement	lakukan x := x + 1
	Fungsi <while-statement> memproses struktur perulangan selama <expression> lakukan <compound-statement>. Parser pertama-tama mencocokkan keyword selama, lalu memproses sebuah expression() sebagai kondisi perulangan. Setelah itu, fungsi memastikan adanya keyword lakukan sebagai penanda awal blok yang akan diulang. Bagian tubuh perulangannya direpresentasikan menggunakan compound_statement(), sehingga loop selalu berisi satu blok terstruktur yang dimulai dengan mulai dan diakhiri selesai.			
20	<for-statement>	Perulangan dengan counter	KEYWORD(untuk) + IDENTIFIER + ASSIGN_OPERATOR + expression + (KEYWORD(ke)/KEYWORD(turun-ke)) + expression + KEYWORD(lakukan) + statement	untuk i := 1 ke 10 lakukan ...
	Fungsi <for-statement> memproses struktur perulangan for. Parser terlebih dahulu mencocokkan keyword untuk, kemudian membaca sebuah identifier sebagai variabel iterasi, diikuti operator penugasan dan sebuah expression() sebagai nilai awal. Setelah itu, fungsi mencoba mencocokkan keyword ke atau turun-ke. Jika salah satu ditemukan, token tersebut ditambahkan ke node. Parser lalu memproses expression() kedua sebagai batas akhir iterasi, memastikan adanya keyword lakukan, dan akhirnya memaksa tubuh perulangan berbentuk compound_statement().			
21	<procedure/function call>	Pemanggilan prosedur atau fungsi	IDENTIFIER + (LPARENTHESIS + parameter-list + RPARENTHESIS)?	writeln('Hello'), print(x, y)
	Fungsi <procedure/function call> bertugas mem-parse pemanggilan prosedur atau fungsi. Parser pertama mencocokkan sebuah IDENTIFIER sebagai nama prosedur/fungsi, lalu memastikan ada tanda kurung buka. Jika token berikutnya bukan kurung tutup, parser memproses daftar argumen melalui parameter_list(). Setelah itu, parsing ditutup dengan mencocokkan kurung tutup.			
22	<parameter-list>	Daftar parameter aktual saat pemanggilan	expression (COMMA + expression)*	'Result = ', b, 100

	Fungsi <parameter-list> memproses daftar parameter aktual yang dipisahkan tanda koma, misalnya pada pemanggilan prosedur atau fungsi. Parser pertama memanggil expression() untuk membaca parameter pertama, lalu selama token berikutnya adalah koma, fungsi menambahkannya sebagai child, maju ke token berikutnya, dan kembali memanggil expression() untuk parameter berikutnya.			
23	<expression>	Ekspresi yang menghasilkan nilai	simple-expression (relational-operator + simple-expression)?	x + y, a > b, 5 * (x + 1)
	Fungsi <expression> memproses ekspresi yang dapat berupa satu simple_expression saja atau dua simple_expression yang dihubungkan operator relasional. Pertama, parser selalu membaca simple_expression() sebagai bagian kiri. Jika token berikutnya adalah salah satu operator relasional seperti =, <>, <, <=, >, atau >=, maka parser menambahkan operator tersebut melalui relational_operator() lalu memproses simple_expression() kedua.			
24	<simple-expression>	Ekspresi tanpa operator relasional	(ARITHMETIC_OPERATOR(+/-))? term (additive-operator + term)*	x + y - z, 5 * 2
	Fungsi <simple-expression> memproses ekspresi aritmatika tingkat atas, yang dapat diawali dengan tanda + atau - opsional. Setelah itu parser membaca sebuah term() sebagai komponen utama ekspresi. Selama token berikutnya adalah operator penjumlahan/pengurangan atau keyword logika atau, parser memanggil additive_operator() lalu kembali memproses term() berikutnya.			
25	<term>	Bagian ekspresi dengan prioritas lebih tinggi	factor (multiplicative-operator + factor)*	x * y, a bagi b
	Fungsi <term> memproses bagian ekspresi yang melibatkan operasi perkalian, pembagian, modulo, atau operator logika dan. Parser pertama mengambil sebuah factor() sebagai komponen awal. Selama token berikutnya adalah salah satu operator tersebut (*, /, bagi, mod, atau dan), parser memanggil multiplicative_operator() lalu membaca factor() berikutnya.			

26	<factor>	Unit terkecil dalam ekspresi	IDENTIFIER / NUMBER / CHAR_LITERAL / STRING_LITERAL / (LPARENTHESIS + expression + RPARENTHESIS) / LOGICAL_OPERATOR(tidak) + factor / function-call	x, 5, 'a', (x+y), flag "tidak"
	Fungsi <factor> digunakan untuk memproses komponen paling dasar dari sebuah ekspresi, yaitu elemen yang tidak lagi dapat dipecah oleh aturan prioritas operator. Fungsi ini menangani beberapa bentuk faktor, termasuk identifier, literal, operator unary, dan ekspresi dalam tanda kurung. Ketika token saat ini berupa identifier dan token berikutnya adalah tanda kurung buka maka identifier tersebut dianggap sebagai pemanggilan prosedur atau fungsi dan diproses melalui procedure_function_call(). Jika tidak, identifier diperlakukan sebagai nilai biasa. Literal seperti angka, karakter, dan string juga langsung diterima sebagai faktor			
27	<relational-operator>	Operator perbandingan	=, <>, <, <=, >, >=	x = 5, y > 10
	Fungsi <relational-operator> digunakan untuk melakukan parsing operator relasional yang dapat muncul di dalam sebuah ekspresi. Fungsi mengecek apakah token saat ini adalah operator yang valid, lalu menambahkannya sebagai child dan maju ke token berikutnya. Jika token tidak valid, fungsi menampilkan pesan error dan tetap melakukan advance() agar parsing tidak berhenti.			
28	<additive-operator>	Operator penjumlahan/pengurangan	+, -, atau	x + y, a atau b
	Fungsi <additive-operator> melakukan parsing operator penjumlahan tingkat atas seperti +, -, dan keyword logika atau. Jika token saat ini cocok dengan salah satu operator tersebut, fungsi membuat node <additive-operator>, menambah token itu sebagai child, lalu maju ke token berikutnya. Jika token bukan operator yang valid, fungsi menampilkan pesan error tetapi tetap melakukan advance().			
29	<multiplicative-operator>	Operator perkalian/pembagian	*, /, bagi, mod, dan	x * y, a bagi b, p dan q
	Fungsi <multiplicative-operator> melakukan parsing operator level menengah seperti *, /, bagi, mod, dan operator logika dan. Jika token cocok			

	salah satu operator tersebut, node <multiplicative-operator> dibuat lalu token ditambahkan sebagai child dan parser maju. Jika tidak cocok, parser mencetak error tetapi tetap melanjutkan parsing.

3. Rincian Implementasi Parse Tree

```
// ========== UTILITY ==========

Token getCurrentToken() {
    if (current < (int)tokens.size()) return tokens[current];
    return Token{"EOF", "", 0, 0};
}

Token cur_tok = getCurrentToken();

void advance() {
    if (current < tokens.size()) current++;
    cur_tok = getCurrentToken();
}

void debugEnter(const string &rule) {
    if (!gDebug) return;
    cerr << ">>> Entering rule: " << rule << " | Current token: ("
         << cur_tok.type << ", '" << cur_tok.lexeme << "' @ " <<
cur_tok.line << ":" << cur_tok.column << ")\n";
}

void debugExit(const string &rule) {
    if (!gDebug) return;
    cerr << "<<< Exiting rule: " << rule << " | Next token: ("
         << cur_tok.type << ", '" << cur_tok.lexeme << "' @ " <<
cur_tok.line << ":" << cur_tok.column << ")\n";
}
```

Bagian utility dari program ini menyediakan fungsi dasar untuk mengelola token dan proses parsing. Fungsi seperti getCurrentToken(), advance(), debugEnter(), dan debugExit() dipakai untuk mengakses token yang sedang diproses, berpindah ke token berikutnya, serta menampilkan log debugging saat parser masuk/keluar dari suatu rule.

```
ParseNode* matchType(const string &expectedType) {
    if (cur_tok.type == expectedType) {
        if (gDebug) cerr << "Matched type: " << expectedType << " (" <<
cur_tok.lexeme << " @ " << cur_tok.line << ":" << cur_tok.column << ")\n";
        Token t = cur_tok;
        advance();
        return makeTokenNode(t);
    } else {
        cerr << "Syntax error: expected type '" << expectedType
             << "' but got (" << cur_tok.type << ", '" << cur_tok.lexeme <<
"' @ " << cur_tok.line << ":" << cur_tok.column << ")\n";
        return makeNode("<missing-" + expectedType + ">");
    }
}
```

```
ParseNode* matchToken(const Token &expected) {
    if (cur_tok.type == expected.type && cur_tok.lexeme == expected.lexeme) {
        if (gDebug) cerr << "Matched token: <" << expected.type << ", '" <<
expected.lexeme << "'>\n";
        Token t = cur_tok;
        advance();
        return makeTokenNode(t);
    } else {
        cerr << "Syntax error: expected token (" << expected.type << ", '"
             << expected.lexeme << "') but got (" << cur_tok.type << ", '"
             << cur_tok.lexeme << "' @ " << cur_tok.line << ":" <<
cur_tok.column << ")\n";
        return makeNode("<missing-" + expected.type + ">");
    }
}

ParseNode* tryMatchToken(const Token &expected) {
    if (cur_tok.type == expected.type && cur_tok.lexeme == expected.lexeme) {
        Token t = cur_tok;
        advance();
        return makeTokenNode(t);
    }
    return nullptr;
}
```

Selain itu, terdapat fungsi-fungsi pembantu seperti matchType(), matchToken(), dan tryMatchToken() yang bertugas mencocokkan token tertentu sesuai grammar. Jika token sesuai, parser mengambilnya dan maju. Apabila token tidak sesuai, program mencetak error tetapi tetap membuat node placeholder agar parsing bisa dilanjutkan tanpa langsung berhenti.

```
// ========== Parse Tree ==========
ParseNode* makeNode(const string &label) {
    auto *n = new ParseNode{label, {}};
    return n;
}

ParseNode* makeTokenNode(const Token &t) {
    auto *n = new ParseNode{t.type + "(" + t.lexeme + ")", {}};
    return n;
}

void addChild(ParseNode* parent, ParseNode* child) {
    if (parent && child) parent->children.push_back(child);
}

void printTree(ParseNode* node, const string &prefix, bool isLast) {
    if (!node) return;

    static bool isRootPrinted = false;
    if (!isRootPrinted) {
        cout << node->label << "\n";
        isRootPrinted = true;
    }
    for (size_t i = 0; i < node->children.size(); ++i) {
        bool last = (i + 1 == node->children.size());
        cout << prefix << (last ? "└── " : "├── ") <<
node->children[i]->label << "\n";
        string newPrefix = prefix + (last ? "    " : "│   ");
        printTree(node->children[i], newPrefix, last);
    }
```

```
    if (prefix.empty()) {
        isRootPrinted = false;
    }
},
```

Bagian parse tree berfungsi membangun struktur pohon sintaks dari hasil parsing. Fungsi seperti makeNode(), makeTokenNode(), dan addChild() membentuk node dan menyusunnya menjadi hirarki sesuai grammar bahasa. Fungsi printTree() kemudian mencetak pohon ini dalam bentuk visual ASCII sehingga mudah untuk dianalisis.

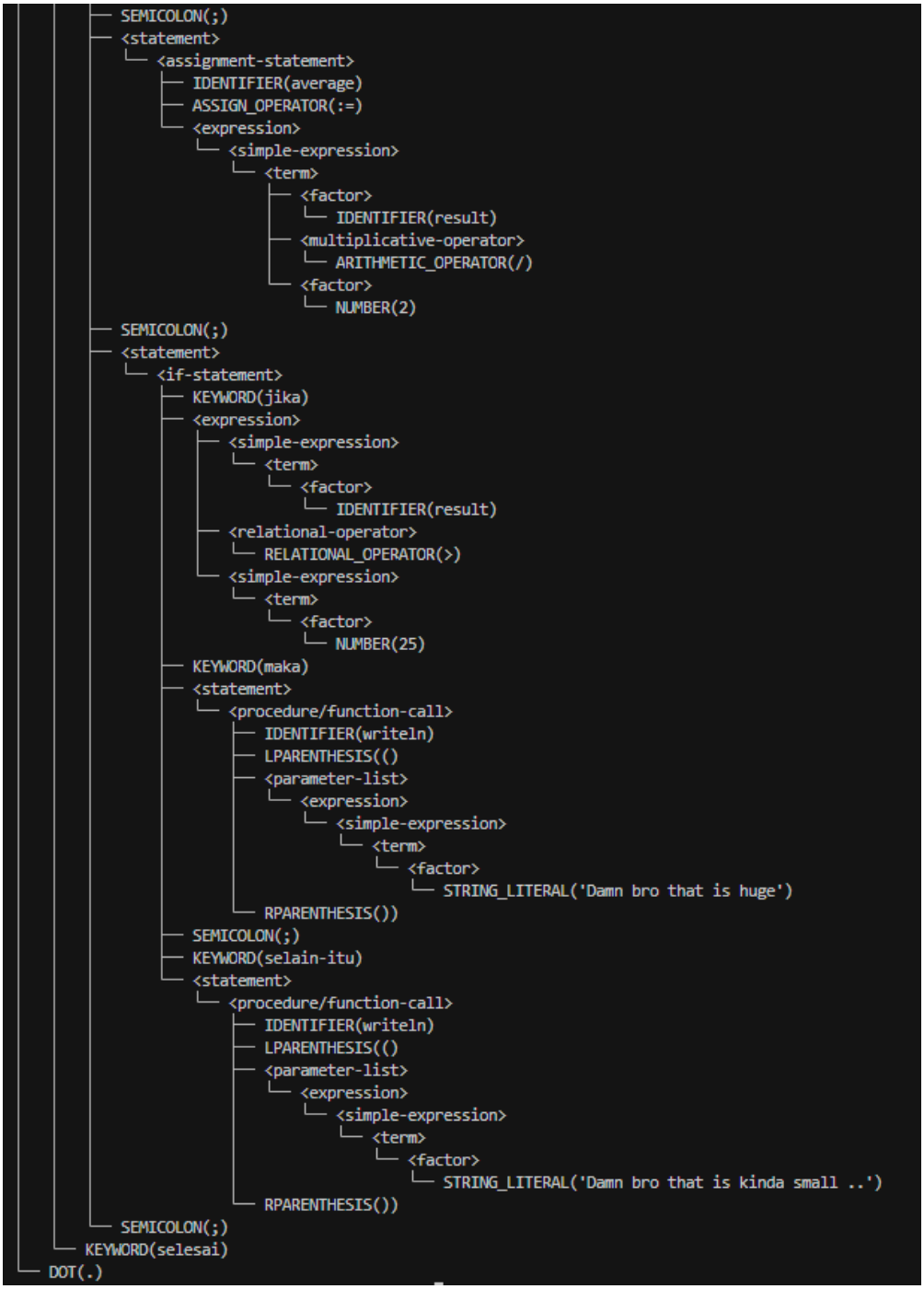
C. PENGUJIAN

Pengujian 1
Test Case

`program if_test;`
`variabel`
`    x, y, result: integer;`
`    average: real;`
`mulai`
`    x := 10;`
`    y := 20;`
`    result := x + y;`
`    average := result / 2;`
`    jika result > 25 maka`
`        writeln('Damn bro that is huge');`
`    selain-itu`
`        writeln('Damn bro that is kinda small ..');`
`selesai.`

Bukti Input
`diyah@LAPTOP-6C00R5DE:/mnt/e/JYP-Tubes-IF2224$ make run ARGS=test/milestone-2/tc1.pas` `./bin/main test/milestone-2/tc1.pas`
Bukti Output

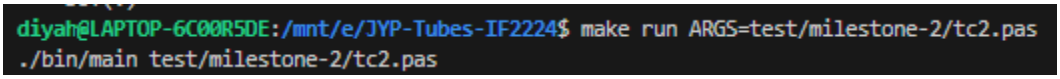

<program>
<program-header>
KEYWORD(program)
IDENTIFIER(if_test)
SEMICOLON(;)
<declaration-part>
<var-declaration>
KEYWORD(variabel)
<identifier-list>
IDENTIFIER(x)
COMMA(,)
IDENTIFIER(y)
COMMA(,)
IDENTIFIER(result)
COLON(:)
<type>
KEYWORD(integer)
SEMICOLON(;)
<identifier-list>
IDENTIFIER(average)
COLON(:)
<type>
KEYWORD(real)
SEMICOLON(;)
<compound-statement>
KEYWORD(mulai)
<statement-list>
<statement>
<assignment-statement>
IDENTIFIER(x)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
NUMBER(10)
SEMICOLON(;)
<statement>
<assignment-statement>
IDENTIFIER(y)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
NUMBER(20)
SEMICOLON(;)
<statement>
<assignment-statement>
IDENTIFIER(result)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(x)
<additive-operator>
ARITHMETIC_OPERATOR(+)
<term>
<factor>
IDENTIFIER(y)



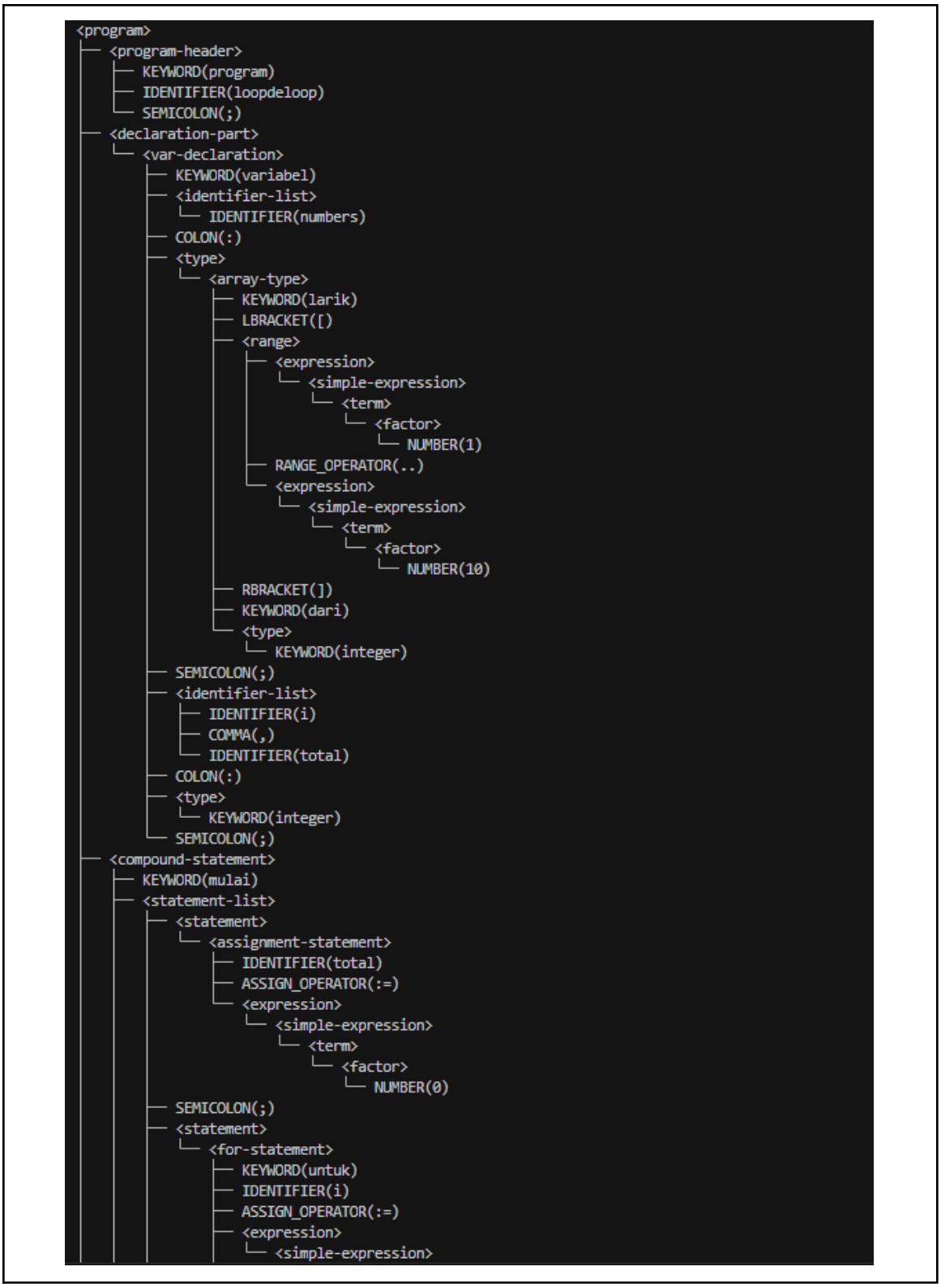
Pengujian 2
Test Case
`program loopdeloop;`

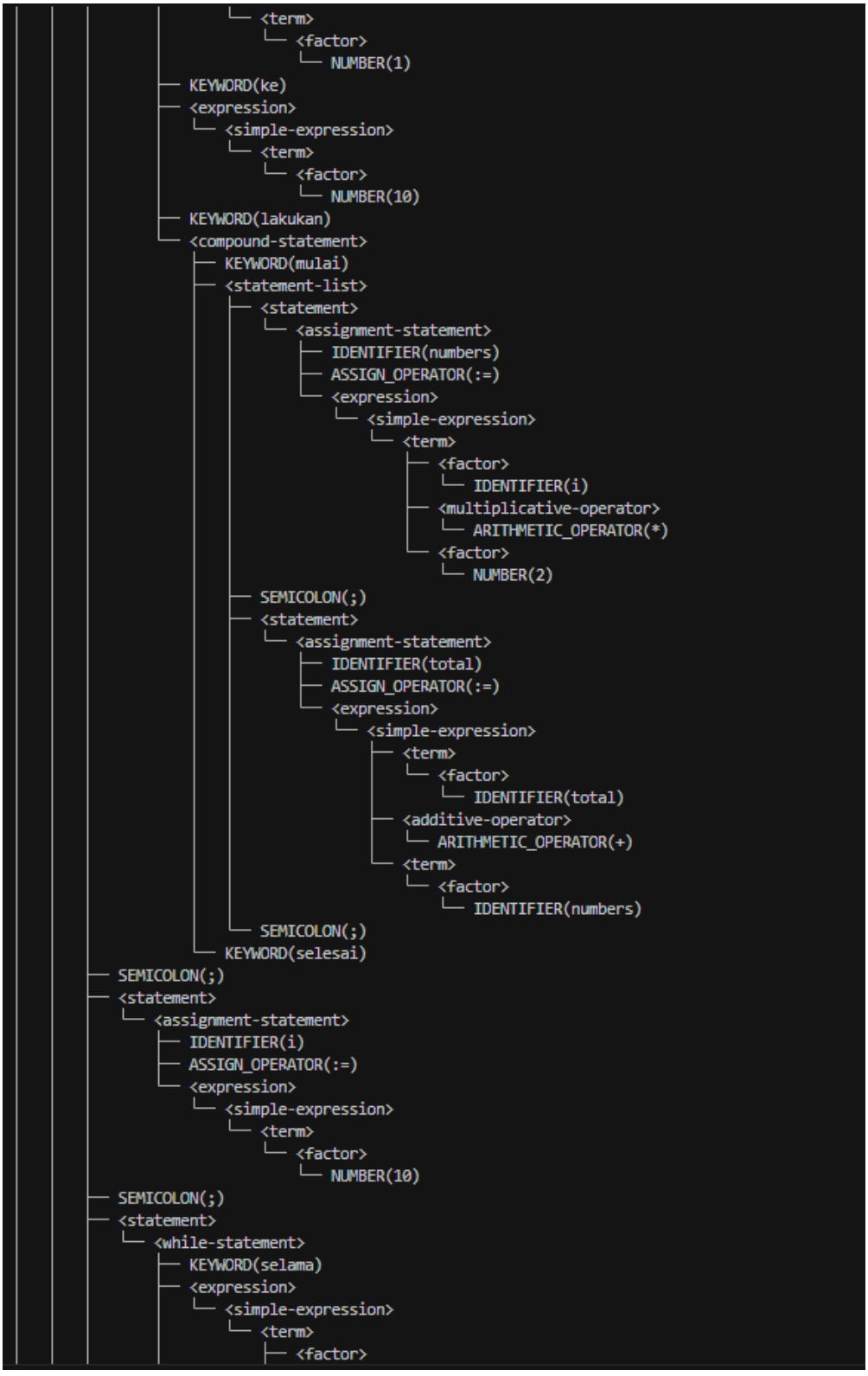
```
variabel
    numbers: larik[1 .. 10] dari integer;
    i, total: integer;
mulai
    total := 0;
    untuk i := 1 ke 10 lakukan
    mulai
        numbers := i * 2;
        total := total + numbers;
    selesai;

    i := 10;
    selama (i >= 1) dan (i <= 10) lakukan
    mulai
        writeln('Sum = ', sum);
        i := i - 1;
    selesai;
selesai.
```

Bukti Input

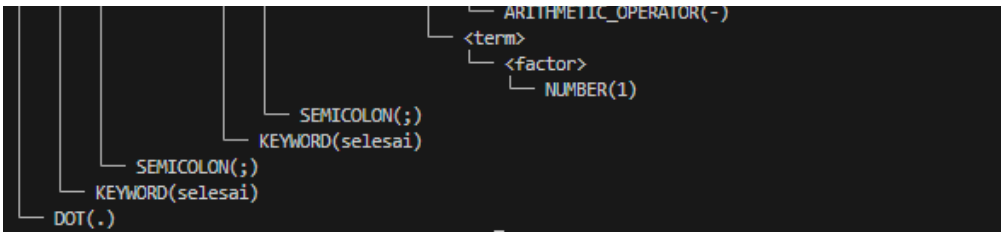


Bukti Output


<program>
<program-header>
KEYWORD(program)
IDENTIFIER(loopdeloop)
SEMICOLON(;)
<declaration-part>
<var-declaration>
KEYWORD(variabel)
<identifier-list>
IDENTIFIER(numbers)
COLON(:)
<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(;)
<identifier-list>
IDENTIFIER(i)
COMMA(,)
IDENTIFIER(total)
COLON(:)
<type>
KEYWORD(integer)
SEMICOLON(;)
<compound-statement>
KEYWORD(mulai)
<statement-list>
<statement>
<assignment-statement>
IDENTIFIER(total)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
NUMBER(0)
SEMICOLON(;)
<statement>
<for-statement>
KEYWORD(untuk)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>


└── <term>
└── <factor>
└── NUMBER(1)
├── KEYWORD(ke)
├── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(10)
├── KEYWORD(lakukan)
└── <compound-statement>
├── KEYWORD(mulai)
├── <statement-list>
├── <statement>
└── <assignment-statement>
├── IDENTIFIER(numbers)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
└── <term>
├── <factor>
└── IDENTIFIER(i)
├── <multiplicative-operator>
└── ARITHMETIC_OPERATOR(*)
└── <factor>
└── NUMBER(2)
├── SEMICOLON(;)
├── <statement>
└── <assignment-statement>
├── IDENTIFIER(total)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
├── <term>
└── <factor>
└── IDENTIFIER(total)
├── <additive-operator>
└── ARITHMETIC_OPERATOR(+)
└── <term>
└── <factor>
└── IDENTIFIER(numbers)
└── SEMICOLON(;)
└── KEYWORD(selesai)
├── SEMICOLON(;)
├── <statement>
└── <assignment-statement>
├── IDENTIFIER(i)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(10)
├── SEMICOLON(;)
├── <statement>
└── <while-statement>
├── KEYWORD(selama)
├── <expression>
└── <simple-expression>
└── <term>
├── <factor>


LPARENTHESIS(()
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(i)
<relational-operator>
RELATIONAL_OPERATOR(>=)
<simple-expression>
<term>
<factor>
NUMBER(1)
RPARENTHESIS())
<multiplicative-operator>
LOGICAL_OPERATOR(dan)
<factor>
LPARENTHESIS(()
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(i)
<relational-operator>
RELATIONAL_OPERATOR(<=)
<simple-expression>
<term>
<factor>
NUMBER(10)
RPARENTHESIS())
KEYWORD(lakukan)
<compound-statement>
KEYWORD(mulai)
<statement-list>
<statement>
<procedure/function-call>
IDENTIFIER(writeln)
LPARENTHESIS(()
<parameter-list>
<expression>
<simple-expression>
<term>
<factor>
STRING_LITERAL('Sum = ')
COMMA(,)
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(sum)
RPARENTHESIS())
SEMICOLON(;)
<statement>
<assignment-statement>
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(i)
<additive-operator>



Pengujian 3
Test Case

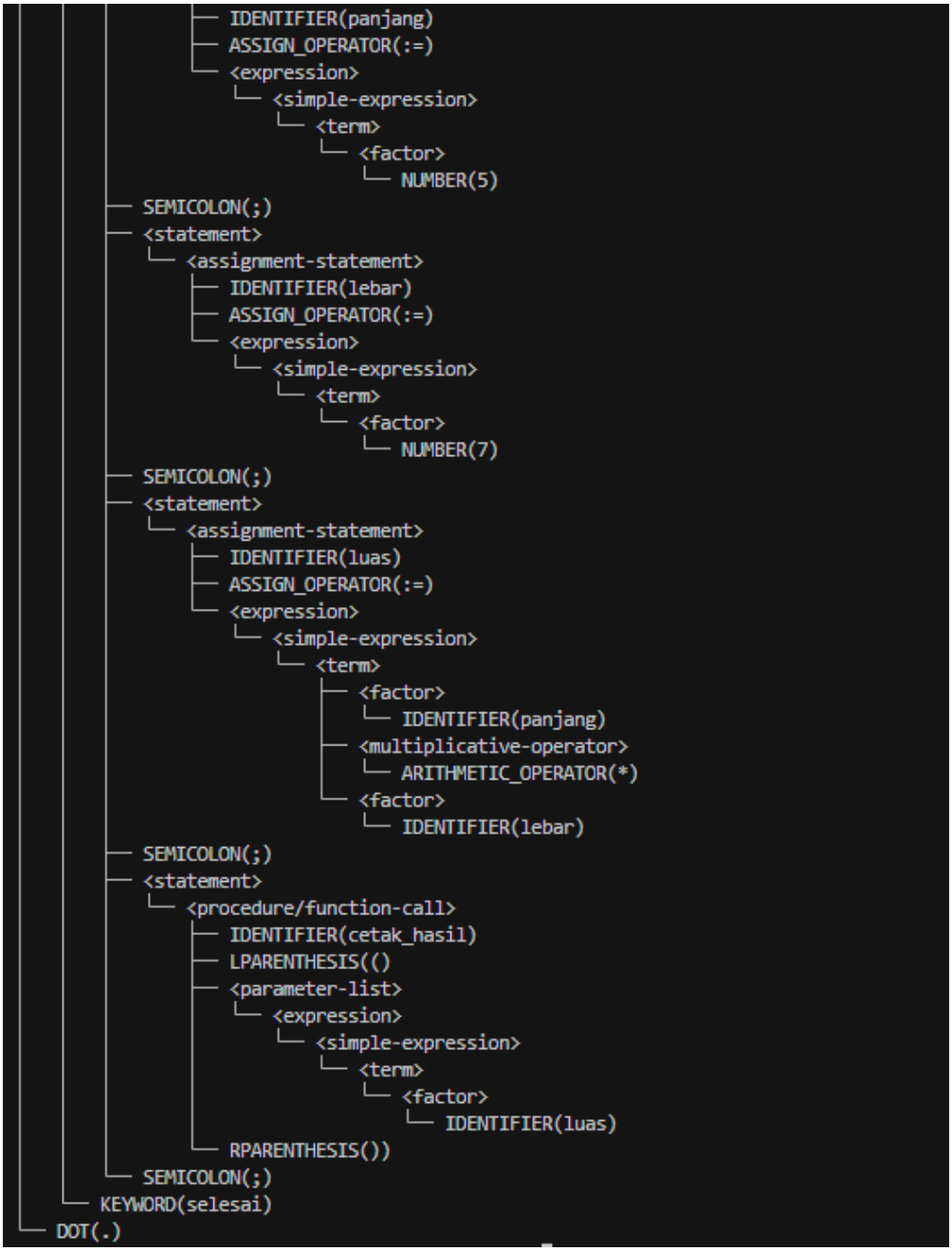
```
program hitungluas;
variabel
    panjang, lebar, luas: integer;

prosedur cetak_hasil(l:integer);
mulai
    writeln('Luas = ', l);
selesai;

mulai
    panjang := 5;
    lebar := 7;
    luas := panjang * lebar;

    cetak_hasil(luas);

selesai.
```

Bukti Input

```
diyah@LAPTOP-6C00R5DE:/mnt/e/JYP-Tubes-IF2224$ make run ARGS=test/milestone-2/tc3.pas
./bin/main test/milestone-2/tc3.pas
```

Bukti Output


<program>
<program-header>
KEYWORD(program)
IDENTIFIER(hitungluas)
SEMICOLON(;)
<declaration-part>
<var-declaration>
KEYWORD(variabel)
<identifier-list>
IDENTIFIER(panjang)
COMMA(,)
IDENTIFIER(lebar)
COMMA(,)
IDENTIFIER(luas)
COLON(:)
<type>
KEYWORD(integer)
SEMICOLON(;)
<subprogram-declaration>
<procedure-declaration>
KEYWORD(prosedur)
IDENTIFIER(cetak_hasil)
<formal-parameter-list>
LPARENTHESIS(()
<parameter-group>
<identifier-list>
IDENTIFIER(l)
COLON(:)
<type>
KEYWORD(integer)
RPARENTHESIS())
SEMICOLON(;)
<block>
<compound-statement>
KEYWORD(mulai)
<statement-list>
<statement>
<procedure/function-call>
IDENTIFIER(writeln)
LPARENTHESIS(()
<parameter-list>
<expression>
<simple-expression>
<term>
<factor>
STRING_LITERAL('Luas = ')
COMMA(,)
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(l)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(selesai)
SEMICOLON(;)
<compound-statement>
KEYWORD(mulai)
<statement-list>
<statement>
<assignment-statement>



Pengujian 4
Test Case

`program test_boolean;`
`variabel`
`    a, b: boolean;`

```
mulai
    a := benar;
    b := salah;

    jika a dan tidak b maka
        writeln('Case 1');

    jika a atau b maka
        writeln('Case 2');

selesai.
```

Bukti Input

```
diyah@LAPTOP-6C00R5DE:/mnt/e/JYP-Tubes-IF2224$ make run ARGS=test/milestone-2/tc4.pas
./bin/main test/milestone-2/tc4.pas
```

Bukti Output

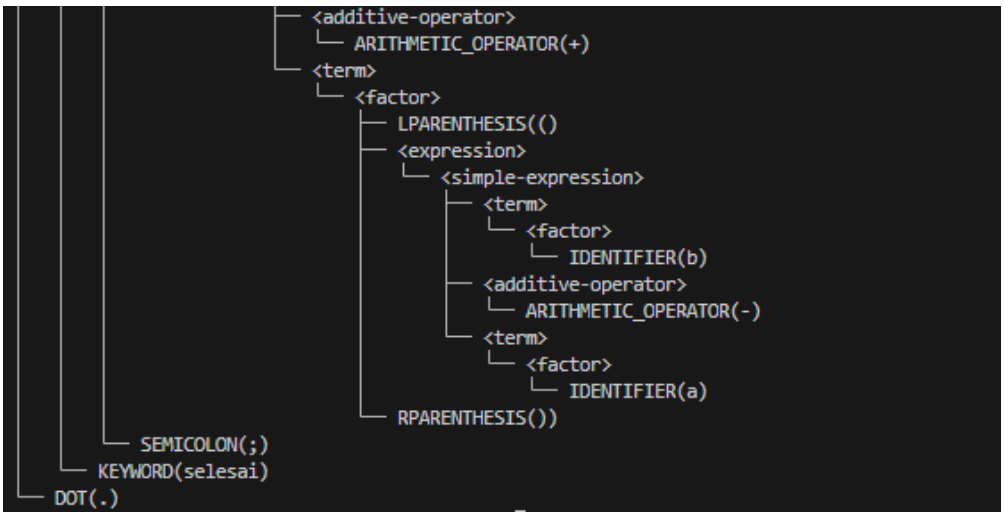

<program>
<program-header>
KEYWORD(program)
IDENTIFIER(test_boolean)
SEMICOLON(;)
<declaration-part>
<var-declaration>
KEYWORD(variabel)
<identifier-list>
IDENTIFIER(a)
COMMA(,)
IDENTIFIER(b)
COLON(:)
<type>
KEYWORD(boolean)
SEMICOLON(;)
<compound-statement>
KEYWORD(mulai)
<statement-list>
<statement>
<assignment-statement>
IDENTIFIER(a)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(benar)
SEMICOLON(;)
<statement>
<assignment-statement>
IDENTIFIER(b)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(salah)
SEMICOLON(;)
<statement>
<if-statement>
KEYWORD(jika)
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(a)
<multiplicative-operator>
LOGICAL_OPERATOR(dan)
<factor>
LOGICAL_OPERATOR(tidak)
<factor>
IDENTIFIER(b)
KEYWORD(maka)
<statement>
<procedure/function-call>
IDENTIFIER(writeln)
LPARENTHESIS(()
<parameter-list>
<expression>
<simple-expression>

Pengujian 5
Test Case

`program chain;`
`var`
`  a, b, c : integer;`
`mulai`
`  a := 5;`
`  b := a + 3;`
`  c := b * a + (b - a);`
`selesai.`

Bukti Input
`diyah@LAPTOP-6C00R5DE:/mnt/e/JYP-Tubes-IF2224$ make run ARGS=test/milestone-2/tc5.pas` `./bin/main test/milestone-2/tc5.pas`
Bukti Output


<program>
<program-header>
KEYWORD(program)
IDENTIFIER(chain)
SEMICOLON(;)
<declaration-part>
<var-declaration>
KEYWORD(variabel)
<identifier-list>
IDENTIFIER(a)
COMMA(,)
IDENTIFIER(b)
COMMA(,)
IDENTIFIER(c)
COLON(:)
<type>
KEYWORD(integer)
SEMICOLON(;)
<compound-statement>
KEYWORD(mulai)
<statement-list>
<statement>
<assignment-statement>
IDENTIFIER(a)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
NUMBER(5)
SEMICOLON(;)
<statement>
<assignment-statement>
IDENTIFIER(b)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(a)
<additive-operator>
ARITHMETIC_OPERATOR(+)
<term>
<factor>
NUMBER(3)
SEMICOLON(;)
<statement>
<assignment-statement>
IDENTIFIER(c)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(b)
<multiplicative-operator>
ARITHMETIC_OPERATOR(*)
<factor>
IDENTIFIER(a)

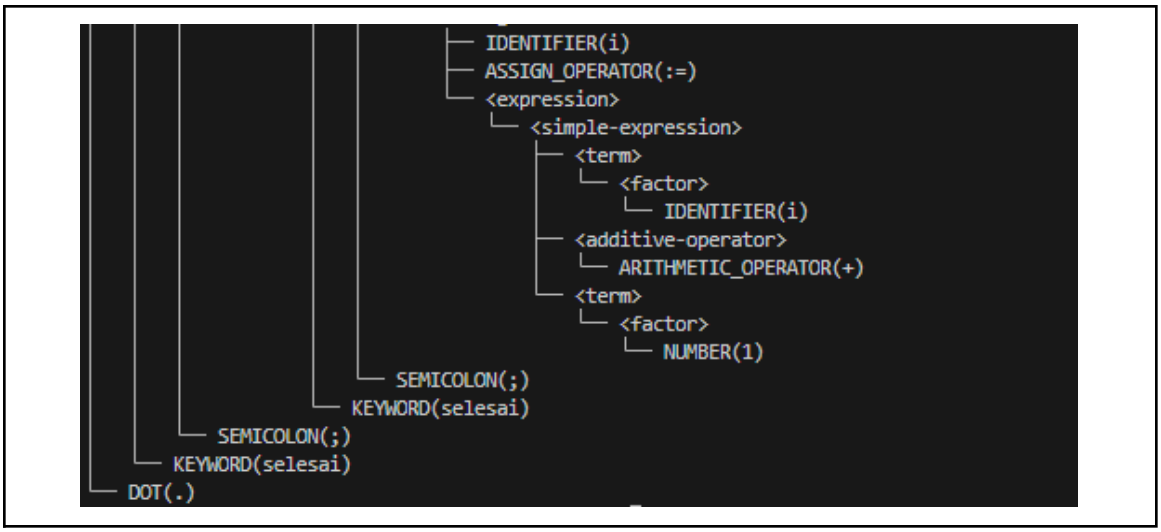


Pengujian 6 : err2.pas
Test Case

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

Bukti Input
`diyah@LAPTOP-6C00R5DE:/mnt/e/JYP-Tubes-IF2224-1$ make run ARGS=test/milestone-2/err2.pas` `./bin/main test/milestone-2/err2.pas`
Bukti Output


Syntax error: expected token (KEYWORD, 'lakukan') but got (KEYWORD, 'mulai' @ 7:5)
<program>
├── <program-header>
│   ├── KEYWORD(program)
│   ├── IDENTIFIER(MissingLakukan)
│   └── SEMICOLON(;)
├── <declaration-part>
│   └── <var-declaration>
│   ├── KEYWORD(variabel)
│   ├── <identifier-list>
│   │   └── IDENTIFIER(i)
│   ├── COLON(:)
│   ├── <type>
│   │   └── KEYWORD(integer)
│   └── SEMICOLON(;)
├── <compound-statement>
│   ├── KEYWORD(mulai)
│   ├── <statement-list>
│   │   ├── <statement>
│   │   │   └── <assignment-statement>
│   │   │   ├── IDENTIFIER(i)
│   │   │   ├── ASSIGN_OPERATOR(:=)
│   │   │   └── <expression>
│   │   │   └── <simple-expression>
│   │   │   └── <term>
│   │   │   └── <factor>
│   │   │   └── NUMBER(1)
│   │   ├── SEMICOLON(;)
│   │   ├── <statement>
│   │   │   └── <while-statement>
│   │   │   ├── KEYWORD(selama)
│   │   │   ├── <expression>
│   │   │   │   ├── <simple-expression>
│   │   │   │   │   └── <term>
│   │   │   │   │   └── <factor>
│   │   │   │   │   └── IDENTIFIER(i)
│   │   │   │   ├── <relational-operator>
│   │   │   │   │   └── RELATIONAL_OPERATOR(<)
│   │   │   │   └── <simple-expression>
│   │   │   │   └── <term>
│   │   │   │   └── <factor>
│   │   │   │   └── NUMBER(10)
│   │   │   ├── <missing-KEYWORD>
│   │   │   └── <compound-statement>
│   │   │   ├── KEYWORD(mulai)
│   │   │   ├── <statement-list>
│   │   │   │   ├── <statement>
│   │   │   │   │   └── <assignment-statement>



D. KESIMPULAN DAN SARAN

1. Kesimpulan

Pada milestone 2, implementasi *syntax analysis* atau *parser* untuk bahasa Pascal-S versi bahasa Indonesia telah berhasil diselesaikan dengan baik. Parser yang dibangun telah mampu menerjemahkan menerjemahkan konstruksi sintaks secara konsisten ke dalam bentuk *parse tree* yang terstruktur, Seluruh grammar yang diminta dalam spesifikasi tugas besar telah berhasil diimplementasikan dengan pendekatan *recursive descent.*

Selain itu, rangkaian test case yang disusun untuk berbagai kategori instruksi menunjukkan bahwa parser bekerja sesuai dengan perilaku yang diharapkan. Parser mampu menangani struktur bersarang, variasi ekspresi dengan prioritas operator yang berbeda, serta kombinasi operasi kompleks yang sering muncul. Parser juga dapat menangani kasus *syntax error* dan memberikan informasi *error* yang terjadi.

2. Saran

Terdapat beberapa keyword seperti “kasus”, “ulangi … sampai”, dan lainnya yang tidak diharuskan untuk diimplementasikan pada milestone ini, parsing yang diimplementasi terbatas pada node yang diberikan pada spesifikasi, ini dapat menjadi kesempatan untuk diimplementasikan pada kesempatan selanjutnya. Selain itu, dapat dikembangkan pesan error yang lebih informatif dengan menambahkan posisi keyword yang error yang mencakup nomor baris dan kolomnya. Pada milestone ini, hal tersebut berhasil dilakukan dengan menambahkan atribut “line” dan “column” pada struct “Token”.

E. LAMPIRAN

1. Link *Repository* Github

Berikut terlampir link *repository* github untuk tugas compiler Pascal-S kelompok JYP K-02:

https://github.com/numshv/JYP-Tubes-IF2224

2. Pembagian Tugas

Berikut terlampir pembagian tugas tiap anggota kelompok JYP K-02 pada *milestone 1*:

NIM	Nama	Tugas	Persentase
13523058	Noumisyifa Nabila Nareswari	Implementasi parser	20%
13523066	M. Ghifary Komara Putra	Menyusun laporan	20%
13523072	Sabilul Huda	Implementasi parser	20%
13523080	Diyah Susan Nugrahani	Menyusun sebagian besar dari laporan dan testing	20%
13523108	Henry Filbert Shinelo	Implementasi parser	20%