

IF2224 TEORI BAHASA FORMAL DAN OTOMATA

**IMPLEMENTASI SYNTAX ANALYSIS PADA PASCAL-S COMPILER
BERBASIS PARSE TREE**

Laporan Milestone 2

Disusun untuk memenuhi tugas mata kuliah Teori Bahasa Formal dan Otomata pada
Semester 5 (lima)

Tahun Akademik 2025/2026



Disusun Oleh:

Brian Ricardo Tamin 13523126

Jovandra Otniel P. S. 13523141

Andrew Tedjapratama 13523148

Theo Kurniady 13523154

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2025

DAFTAR ISI

DAFTAR ISI.....	2
BAB I	
DESKRIPSI persoalan.....	3
BAB II	
LANDASAN TEORI.....	6
2.1 Context Free Grammar (CFG).....	6
2.2 Syntax Analysis dan Peran Parser.....	7
2.3 Token.....	7
2.4 Algoritma Recursive Descent.....	9
2.5 Parse Tree.....	10
2.6 Hubungan Lexer & Parser.....	14
BAB III	
PERANCANGAN PROGRAM.....	15
3.1 Arsitektur dan Teknologi yang Digunakan.....	15
3.2 Struktur Program dan Kelas Utama.....	15
3.3 Alur Kerja Program.....	18
3.5 Aturan Grammar CFG Parser.....	19
BAB IV	
IMPLEMENTASI PROGRAM.....	24
4.1 Parse Node.....	24
4.2 Parse Error.....	25
4.3 Parser.....	25
BAB V	
TEST CASE.....	38
5.1 Test Case Lanjutan Milestone-1.....	38
5.2 Test Case Tambahan Milestone-2.....	52
BAB VI	
KESIMPULAN & SARAN.....	66
6.1 Kesimpulan.....	66
6.2 Saran.....	66
REFERENSI.....	67
LAMPIRAN.....	68

BAB I

DESKRIPSI PERSOALAN

1.1 Pendahuluan

Bahasa pemrograman Pascal-S merupakan sebuah subset dari bahasa Pascal yang dirancang dengan aturan sintaks yang lebih terbatas. Pada tahap kedua pembangunan compiler Pascal-S, fokus utama akan bergeser dari proses lexical analysis menjadi syntax analysis, yaitu proses memeriksa susunan token dan memastikan bahwa urutan tersebut sesuai dengan aturan tata bahasa (grammar) Pascal-S. Milestone 2 ini menuntut implementasi parser yang mampu membaca list token hasil lexer, kemudian membentuk parse tree yang mencerminkan struktur sintaks program secara hierarkis.

Berbeda dengan tahap lexical analysis sebelumnya yang hanya mengelompokkan karakter menjadi token, syntax analysis memerlukan pemeriksaan hubungan antar token, urutan kemunculan, pemilihan aturan produksi grammar, serta penyusunan struktur blok program seperti deklarasi, pernyataan, ekspresi, prosedur, dan fungsi. Dengan demikian, parser harus mampu mengenali konstruksi bahasa Pascal-S secara menyeluruh, bukan hanya token individual.

1.2 Rumusan Masalah

Adapun permasalahan yang dihadapi pada milestone ini adalah:

1. Bagaimana cara membangun parser yang mengikuti grammar Pascal-S secara deterministik, tanpa menghasilkan ambiguitas atau konflik aturan produksi, dan dapat menangani struktur sintaks seperti deklarasi tipe, deklarasi prosedur/fungsi, assignment, statement bersarang, control flow, dan ekspresi dengan tingkat prioritas operator yang tepat?
2. Bagaimana menghasilkan parse tree yang benar-benar menggambarkan struktur hirarki program sesuai grammar?
3. Bagaimana mendeteksi Syntax Error (eror syntax) secara akurat dan memberikan pesan kesalahan yang informatif (line, column, dan highlight sumber kesalahan)?

Untuk menjawab permasalahan tersebut, milestone ini dibangun untuk membuat suatu parser dengan pendekatan Recursive Descent Parsing, sebuah metode top-down yang cocok untuk grammar yang relatif terstruktur seperti Pascal-S.

1.3 Tujuan (Milestone 2)

Milestone ini bertujuan untuk:

1. Mengimplementasikan syntax analyzer yang mampu membaca list token hasil lexer.
2. Membangun parse tree berdasarkan grammar lengkap Pascal-S.
3. Membuat syntax error checker secara akurat dan berguna.
4. Menerapkan parsing top-down dengan metode recursive descent pada seluruh produksi grammar.
5. Menyediakan pondasi bagi tahap berikutnya seperti semantic analysis dan code generation.

1.4 Tantangan Parsing Pascal-S

Berdasarkan analisis persoalan dan rumusan masalah, parsing memiliki beberapa tantangan, sebagai berikut:

1. Prioritas operator

Artinya ekspresi aritmatika dan logika memiliki tingkatan prioritas yang berbeda (factor -> term -> simple-expression -> expression). Parser harus membangun struktur ini dengan benar.

2. Grammar hierarkis dan panjang

Pascal-S memiliki banyak konstruksi yang harus di-parse secara berurutan, serta keperluan nested statements blok. Hal ini karena parser tidak hanya memproses elemen sederhana seperti identifier dan literal, tetapi juga harus mampu mengenali berbagai konstruksi sintaks tingkat tinggi, seperti deklarasi konstanta, deklarasi tipe yang mencakup range, array, maupun record, serta deklarasi variabel dengan tipe tunggal atau kompleks. Selain itu, Pascal-S mendukung prosedur dan fungsi yang

dapat bersarang, sehingga parser perlu menangani blok deklarasi dan blok program secara rekursif.

3. Type-definition yang fleksibel

Lexer hanya mem-break token-type pada milestone sebelumnya, parser baru memahami strukturnya dan ada beberapa aturan lebih terkait tipe. Ada kemungkinan tipe berupa:

- range
- array
- record
- builtin type
- custom type

4. Ambiguitas dalam pemanggilan, contohnya seperti assignment dan function/procedure call, yang sama-sama diwali oleh identifier. Parser harus melakukan *lookahead* untuk membedakan keduanya.
5. Beberapa tambahan parsing lain seperti ambiguitas assignment vs function call, error handling parser yang akurat dan informatif.

BAB II

LANDASAN TEORI

2.1 Context Free Grammar (CFG)

Context Free Grammar adalah sekumpulan aturan yang dipakai untuk menjabarkan sintaks (susunan) dari bahasa Context Free Language (CFL). CFG terdiri dari variabel, baik terminal maupun non-terminal, dan sejumlah aturan produksi yang memungkinkan substitusi simbol non-terminal menjadi simbol lain.

Secara matematis, CFG direpresentasikan sebagai 4-tuple sebagai berikut:

$$CFG = (V, \Sigma, R, S)$$

dimana:

V : Himpunan simbol non-terminal

Σ : Himpunan simbol terminal (token lexer)

R : Aturan produksi

S : Simbol awal (*start symbol*)

Pada *syntax analysis*, CFG dipakai untuk menentukan kesesuaian sebuah string (susunan token) dengan aturan produksi. Berbeda dengan context sensitive grammar, penggantian simbol non-terminal CFG tidak bergantung pada simbol lain di sekitarnya. Selain itu, CFG dapat menggambarkan pola-pola hierarkis seperti *parse tree* dan *syntax tree*. Oleh karena itu, CFG dipakai sebagai tata bahasa dalam struktur bahasa pemrograman Pascal-S.

Berikut ini contoh aturan produksi CFG:

$$<\text{expression}> \rightarrow <\text{term}> | <\text{expression}> + <\text{term}>$$

Pada Pascal-S, grammar untuk header program adalah:

$$<\text{program-header}> \rightarrow \text{KEYWORD}(\text{program}) \text{ IDENTIFIER}() \text{ SEMICOLON}(;) \quad \text{dapat dituliskan}$$

2.2 Syntax Analysis dan Peran Parser

Analisis sintaks adalah proses yang memeriksa kesesuaian susunan token yang dihasilkan oleh lexer terhadap grammar yang didefinisikan di parser. Parser menjalankan tugasnya setelah lexer. Parser membaca token secara berurutan, kemudian mencocokkan urutannya dengan aturan produksi *grammar* yang berlaku. Hasil dari proses parsing adalah *parse tree*. Jika token tidak sesuai dengan *grammar*, parser berhenti dan mengeluarkan pesan error. Simbol terminal direpresentasikan sebagai token yang dihasilkan sebelumnya dari lexer. Sementara itu, simbol non-terminal direpresentasikan node dalam parse tree (termasuk root <program>) yang dapat diproses secara rekursif oleh parser.

Parser adalah komponen compiler yang memeriksa apakah susunan token sesuai aturan grammar. Parser bekerja setelah lexer menghasilkan token. Lexer hanya memecah karakter menjadi token, sedangkan parser menentukan struktur sintaksis menggunakan token sebagai input. Tujuan utama parser adalah menentukan struktur sintaks, memvalidasi grammar, dan menghasilkan parse tree.

2.3 Token

Token merupakan unit bahasa terkecil yang memiliki arti dalam bahasa pemrograman. Umumnya, token terbentuk dari 2 bagian utama, yaitu tipe (*type*) dan nilai (*value*). Tipe menunjukkan kategori token seperti *identifier*, *keyword*, dan *operator*. Nilai berisi satu atau kumpulan simbol aktual dari token tersebut. Pada milestone ini, value dari token diubah menjadi Bahasa Indonesia. Berikut ini adalah tabel token dalam Pascal-S yang dipakai:

Tabel 2.3.1 Daftar Token dalam Pascal-S

No	Tipe (type)	Nilai (value)	Keterangan
1	KEYWORD	program, variabel, mulai, selesai, jika, maka, selain_itu, selama, lakukan, untuk, ke, turun_ke, integer, real, boolean, char, larik, dari, prosedur,	Kata kunci yang sudah didefinisikan oleh bahasa Pascal-S dan memiliki fungsi khusus dalam struktur program.

		fungsi, konstanta, tipe	
2	IDENTIFIER	x, y, z, sum, avg, count	Nama yang didefinisikan oleh pengguna, misalnya nama variabel, prosedur, atau fungsi.
3	ARITHMETIC_OPERATOR	+, -, *, /, div, mod	-
4	RELATIONAL_OPERATOR	=, <>, <, <=, >, >=	-
5	LOGICAL_OPERATOR	dan, atau, tidak	-
6	ASSIGN_OPERATOR	:=	Operator penugasan yang digunakan untuk memberi nilai ke variabel
7	NUMBER	22, 3, 2018	Bilangan berupa integer atau ril
8	CHAR_LITERAL	'a', 'b', 'c'	-
9	STRING_LITERAL	'tbfo', 'seru sekali'	-
10	SEMICOLON	;	-
11	COMMA	,	-
12	COLON	:	-
13	DOT	.	-
14	LPARENTHESIS	(-
15	RPARENTHESIS)	-
16	LBRACKET	[-

17	RBRACKET]	-
18	RANGE_OPERATOR	..	-

2.4 Algoritma Recursive Descent

Recursive descent parsing adalah teknik parsing yang dilakukan secara *top-down* untuk membangun sebuah parse tree berdasarkan pemanggilan fungsi parsing secara rekursif. Setiap simbol non-terminal pada grammar direpresentasikan sebagai fungsi. Ketika parser menemukan aturan produksi yang sesuai, parser memanggil fungsi yang sesuai untuk menangani bagian tersebut. Misalnya, pada grammar:

```
<program> -> <program-header> <declaration-part> <compound-statement> DOT(.)
```

Parser akan memanggil fungsi dengan gambaran seperti berikut:

parseProgram():

```
    parseProgramHeader()
    parseDeclarationPart()
    parseCompoundStatement()
    match(DOT(.))
```

Metode parsing ini mempunyai beberapa kelebihan, yaitu keselarasan algoritma dengan CFG dan kemudahan menelusuri node secara rekursif hingga *leaf node* (simbol terminal) ditemukan. Namun, terdapat *trade-off* yang harus ditanggulangi supaya algoritma ini dapat berjalan dengan optimal, yaitu *grammar* tidak boleh mengandung *left recursion* untuk menghindari *infinite recursion*. Terlepas dari itu, metode recursive descent dipilih karena sangat sesuai dengan struktur grammar Pascal-S yang cenderung terstruktur dan tidak ambigu. Grammar Pascal-S memiliki bentuk yang dapat dipecah menjadi fungsi-fungsi kecil sehingga setiap non-terminal dapat dipetakan secara langsung menjadi sebuah fungsi dalam kode. Misalnya, <expression> diproses oleh parseExpression(), <term> oleh parseTerm(), <factor> oleh parseFactor(), dan seterusnya. Pemetaan ini membuat implementasi parser lebih modular, mudah dibaca, dan mudah ditelusuri saat debugging.

Untuk dapat memilih aturan produksi yang tepat, parser juga membutuhkan mekanisme lookahead, yaitu membaca token berikutnya tanpa mengonsumsi token tersebut. Pada program ini, digunakan fungsi seperti peek(). Lookahead diperlukan untuk membedakan struktur yang memiliki awalan token sama, misalnya membedakan antara assignment ($x := 5$) dan function call ($x(5)$), karena keduanya diawali IDENTIFIER. Recursive descent sangat mendukung mekanisme ini karena proses parsing dilakukan secara manual dan fleksibel.

Agar recursive descent bekerja secara optimal, grammar yang digunakan harus bebas dari left recursion untuk menghindari pemanggilan rekursif tak berhingga. Oleh karena itu, grammar Pascal-S yang digunakan pada proyek ini sudah disusun ulang ke dalam bentuk yang tidak memiliki left recursion, serta diurutkan secara eksplisit sehingga dapat diproses top-down tanpa konflik.

2.5 Parse Tree

Parse tree adalah representasi pohon (*tree*) yang menunjukkan susunan token yang terbentuk sesuai *grammar*. Node akar (*root*) merepresentasikan simbol awal (*start symbol*) grammar. Sementara itu, node internal mewakili simbol non-terminal grammar. Node daun (*leaf node*) pada tree merupakan token yang dihasilkan lexer.

Misalkan sebuah aturan produksi seperti berikut:

$$A \rightarrow XYZ$$

maka pada *parse tree*, node A akan mempunyai tiga anak (*child*), yaitu X, Y, dan Z. Hal ini selaras dengan struktur parse tree untuk <program-header> yang digambarkan seperti berikut:

```
<program-header>
  └── KEYWORD(program)
  └── IDENTIFIER(Hello)
  └── SEMICOLON
```

Berikut ini merupakan tabel node yang mungkin muncul dalam hasil *parse tree*:

No	Nama Node	Deskripsi	Aturan Produksi	Contoh
1	<program>	Node root yang merepresentasikan keseluruhan program Pascal-S	$\text{program} \rightarrow \text{program-header} + \text{declaration-part} + \text{compound-state ment} + \text{DOT}$	Seluruh struktur program dari awal hingga akhir
2	<program-header>	Bagian kepala program yang berisi nama program	$\text{KEYWORD(program)} + \text{IDENTIFIER} + \text{SEMICOLON}$	program Hello;
3	<declaration-part>	Bagian deklarasi yang dapat berisi	$(\text{const-declaratio n})^*$	Semua deklarasi sebelum "mulai"

		const, type, var, procedure, function	(type-declaration) [*] + (var-declaration) [*] + (subprogram-dec laration) [*]	
4	<const-declar ation>	Deklarasi konstanta	KEYWORD(konsta nta) + (IDENTIFIER = value + SEMICOLON)+	konstanta MAX = 100;
5	<type-declara tion>	Deklarasi tipe data baru	KEYWORD(tipe) + (IDENTIFIER = type-definition + SEMICOLON)+	tipe Range = 1..10;
6	<var-declarat ion>	Deklarasi variabel	KEYWORD(variab el) + (identifier-list + COLON + type + SEMICOLON)+	variabel a, b: integer;
7	<identifier-l ist>	Daftar identifier yang dipisahkan koma	IDENTIFIER (COMMA + IDENTIFIER)*	a, b, c
8	<type>	Tipe data (integer, real, boolean, char, array)	KEYWORD(integer /real/boolean/ch ar) atau array-type	integer, larik[1..10] dari integer
9	<array-type>	Definisi tipe array	KEYWORD(larik) + LBRACKET + range + RBRACKET + KEYWORD(dari) + type	larik[1..10] dari integer
10	<range>	Rentang nilai untuk array atau subrange	expression + RANGE_OPERAT OR(..) + expression	1..10, 'a'..'z'
11	<subprogram-d eclaration>	Deklarasi prosedur atau fungsi	procedure-declar ation atau function-declarat ion	prosedur print (x: integer);

12	<procedure-declaration>	Deklarasi prosedur	KEYWORD(prosedur) + IDENTIFIER + (formal-parameter-list)? + SEMICOLON + block + SEMICOLON	Prosedur dengan parameter opsional
13	<function-declaration>	Deklarasi fungsi	KEYWORD(fungsi) + IDENTIFIER + (formal-parameter-list)? + COLON + type + SEMICOLON + block + SEMICOLON	Fungsi yang mengembalikan nilai
14	<formal-parameter-list>	Daftar parameter formal	L PARENTHESIS + parameter-group (SEMICOLON + parameter-group)* + R PARENTHESIS	(x, y: integer; z: real)
15	<compound-statement>	Blok statement yang diawali begin dan diakhiri end	KEYWORD(mulai) + statement-list + KEYWORD(selesai)	mulai ... selesai
16	<statement-list>	Daftar statement yang dipisahkan semicolon	statement (SEMICOLON + statement)*	Urutan statement dalam blok
17	<assignment-statement>	Statement penugasan nilai ke variabel	IDENTIFIER + ASSIGN_OPERATOR(:=) + expression	x := 5, y := x + 10
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
19	<while-statement>	Perulangan	KEYWORD(selama)	selama x < 10

	<code><ent></code>	dengan kondisi di awal	<code>) + expression + KEYWORD(lakukan) + statement</code>	<code>lakukan x := x + 1</code>
20	<code><for-statement></code>	Perulangan dengan counter	<code>KEYWORD(untuk) + IDENTIFIER + ASSIGN_OPERATOR + expression + (KEYWORD(ke)/KEYWORD(turun-k e)) + expression + KEYWORD(lakukan) + statement</code>	<code>untuk i := 1 ke 10 lakukan ...</code>
21	<code><procedure/function-call></code>	Pemanggilan prosedur atau fungsi	<code>IDENTIFIER + (LPARENTHESIS + parameter-list + RPARENTHESIS)?</code>	<code>writeln('Hello'), print(x, y)</code>
22	<code><parameter-list></code>	Daftar parameter aktual saat pemanggilan	<code>expression (COMMA + expression)*</code>	<code>'Result = ', b, 100</code>
23	<code><expression></code>	Ekspresi yang menghasilkan nilai	<code>simple-expression (relational-operator + simple-expression)?</code>	<code>x + y, a > b, 5 * (x + 1)</code>
24	<code><simple-expression></code>	Ekspresi tanpa operator relasional	<code>(ARITHMETIC_OPERATOR(+/-))? term (additive-operator + term)*</code>	<code>x + y - z, 5 * 2</code>
25	<code><term></code>	Bagian ekspresi dengan prioritas lebih tinggi	<code>factor (multiplicative-operator + factor)*</code>	<code>x * y, a bagi b</code>
26	<code><factor></code>	Unit terkecil dalam ekspresi	<code>IDENTIFIER / NUMBER / CHAR_LITERAL / STRING_LITERAL / (LPARENTHESIS + expression + RPARENTHESIS) / LOGICAL_OPERA</code>	<code>x, 5, 'a', (x+y), flag "tidak"</code>

			TOR(tidak) + factor / function-call	
27	<function-call †>	Pemanggilan fungsi yang mengembalikan nilai	IDENTIFIER + LPARENTHESIS + (parameter-list)? RPARENTHESIS	sqrt(16), max(a, b)
28	<relational-operator>	Operator perbandingan	=, <>, <, <=, >, >=	x = 5, y > 10
29	<additive-operator>	Operator penjumlahan/pengurangan	+, -, atau	x + y, a atau b
30	<multiplicative-operator>	Operator perkalian/pembagian	*, /, bagi, mod, dan	x * y, a bagi b, p dan q

2.6 Hubungan Lexer & Parser

Lexer dan parser bekerja secara urutan pada dua tahap pertama compiler. Lexer pada milestone-1 mengolah kode pascal-S menjadi kumpulan token yang sudah dimapping melalui DFA. Hasil list token tersebut adalah bahasa yang dikenali oleh parser. Parser akan mengambil list token tersebut dan membangun parse tree berdasarkan aturan grammar bahasa pascal-S. Artinya keduanya saling melengkapi dan menjadi langkah awal dalam kompilasi sebuah bahasa pemrograman, yang pada kasus ini adalah pascal-S.

BAB III

PERANCANGAN PROGRAM

Penjelasan ini merinci rancangan program *syntax analyzer* (parser) untuk *compiler* Pascal-S, mencakup teknologi, struktur, alur kerja, dan justifikasi atas pilihan desain yang diambil.

3.1 Arsitektur dan Teknologi yang Digunakan

Program ini dibangun murni menggunakan bahasa pemrograman Python 3. Tidak ada *library* eksternal yang digunakan untuk logika *parsing* inti. Pemilihan ini didasarkan pada kemudahan implementasi, keterbacaan kode, dan kapabilitas bawaan Python dalam menangani struktur data (seperti *list* untuk token dan *class* untuk *node* pohon) serta *exception handling* yang mumpuni.

3.2 Struktur Program dan Kelas Utama

```
BBC-TUBES-IF2224
├── doc
│   ├── Diagram-1-BBC.png
│   └── Laporan-1-BBC.pdf
├── main.py
└── README.md
src
├── compiler.py
├── config
│   ├── states.json
│   ├── token_maps.json
│   └── transitions.json
├── dfa
│   ├── dfa_config.py
│   ├── dfa_engine.py
│   ├── __init__.py
│   └── __pycache__
│       ├── dfa_config.cpython-313.pyc
│       ├── dfa_engine.cpython-313.pyc
│       └── __init__.cpython-313.pyc
├── __init__.py
└── lexer
    ├── __init__.py
    ├── lexer_config.py
    ├── lexer.py
    ├── lexical_error.py
    └── __pycache__
        ├── __init__.cpython-313.pyc
        ├── lexer_config.cpython-313.pyc
        ├── lexer.cpython-313.pyc
        ├── lexical_error.cpython-313.pyc
        └── token.cpython-313.pyc
```

```
    └── token.py
  └── parser
    ├── parse_error.py
    ├── parse_node.py
    ├── parser.py
    └── pycache_
      ├── parse_error.cpython-313.pyc
      ├── parse_node.cpython-313.pyc
      └── parser.cpython-313.pyc
  └── pycache_
    ├── compiler.cpython-313.pyc
    ├── __init__.cpython-313.pyc
    └── utils.cpython-313.pyc
└── utils.py
test
└── milestone-1
  └── input
    ├── input-1.pas
    ├── input-2.pas
    ├── input-3.pas
    ├── input-4.pas
    └── input-5.pas
  └── input-indo
    ├── input-1.pas
    ├── input-2.pas
    ├── input-3.pas
    ├── input-4.pas
    ├── input-5.pas
    ├── input-kasus.pas
    └── input-rekaman.pas
  └── output
    ├── output-1.txt
    ├── output-2.txt
    ├── output-3.txt
    ├── output-4.txt
    ├── output-5.txt
    └── output.txt
└── milestone-2
  └── input
    ├── input-1.pas
    ├── input-2.pas
    ├── input-3.pas
    ├── input-4.pas
    ├── input-5.pas
    ├── input-6.pas
    ├── input-7.pas
    ├── input-8.pas
    ├── input-9.pas
    ├── input-kasus.pas
    └── input-rekaman.pas
  └── output
    └── output.txt
```

19 directories, 65 files

Rancangan program dibagi menjadi beberapa *file* modular untuk memisahkan tanggung jawab (*separation of concerns*):

1. Parser (parser.py):

Ini adalah kelas utama yang menampung seluruh logika parsing.

- Metode Utilitas:

1. advance(): untuk maju satu token dari stream.
2. peek(offset=1): untuk mengintip token berikutnya tanpa mengkonsumsinya (dipakai untuk sistem *lookahead*)
3. check(type, expected_value?): untuk memverifikasi apakah token saat ini sesuai dengan tipe-nilai yang diharapkan.
4. except(type, expected_value?): untuk memverifikasi dan mengkonsumsi token. Jika token tidak sesuai, function ini akan melempar ParseError.

- Metode Parsing (Recursive Descent):

1. Setiap aturan non-terminal dalam *Context Free Grammar* (CFG) Pascal-S diimplementasikan sebagai sebuah metode di dalam kelas Parser. (parse_program(), parse_declaraction_part(), parse_statement(), parse_expression(), parse_term(), parse_factor(), dan lain-lain.)

- ParseNode(parse_node.py)

1. Sebuah kelas struktur data sederhana yang merepresentasikan satu *node* di dalam *parse tree*.
2. Setiap node memiliki atribut type (nama aturan grammar, misal: <program> atau IDENTIFIER) dan child (sebuah list yang berisi ParseNode atau token anak).
3. Memiliki metode str yang diformat khusus untuk mencetak keseluruhan parse tree secara hierarkis dan visual.

- ParseError(parse_error.py)

1. Sebuah kelas struktur data sederhana yang membantu dalam mendeteksi dan menuliskan error message setiap kali ada kesalahan sintaks
2. Pesan error memiliki atribut message (isi pesan error), line (nomor baris kesalahan), column (nomor kolom kesalahan), value (nilai token), dan full_source_text (isi seluruh teks).
3. Memiliki metode str yang diformat khusus untuk mencetak error message dengan format yang jelas.

3.3 Alur Kerja Program

Alur kerja *syntax analysis* mengikuti pola *top-down* yang ketat:

1. Inisialisasi : *Lexer* menghasilkan daftar (list) token. Objek Parser kemudian diinisialisasi dengan daftar token tersebut.
2. Memulai Parsing: Proses dimulai dengan pemanggilan metode `parse()`, yang kemudian memanggil `parse_program()` sebagai *start symbol* dari grammar.
3. Recursive Descent: `parse_program()` akan memanggil metode lain sesuai aturan produksinya, secara berurutan:
 - Memanggil `parse_program_header()`.
 - Memanggil `parse_declaraction_part()`.
 - Memanggil `parse_compound_statement()`.
 - Mengharapkan token DOT di akhir.
4. Penanganan Aturan: Setiap metode *parsing* (misal `parse_statement`) menggunakan `check()` untuk menentukan aturan produksi mana yang harus diambil (misalnya, membedakan antara jika atau selama). Setelah aturan diidentifikasi, `expect()` digunakan untuk mengonsumsi token-token yang membentuk aturan tersebut.
5. Operator Precedence: Prioritas operator ditangani secara implisit melalui struktur pemanggilan rekursif: `parse_expression()` memanggil `parse_simple_expression()`, yang memanggil `parse_term()`, yang memanggil `parse_factor()`. Ini memastikan bahwa factor (prioritas tertinggi, seperti literal atau tanda kurung) dievaluasi terlebih dahulu, diikuti oleh term (*, /, mod), dan terakhir simple-expression (+, -).
6. Pembangunan Pohon: Selama proses, setiap kali sebuah aturan grammar (fungsi) berhasil di-*parse*, sebuah *ParseNode* dibuat. Token yang dikonsumsi dan *node* lain yang dihasilkan dari pemanggilan rekursif ditambahkan sebagai *child* ke *node* tersebut.
7. Hasil:
 - Sukses: Jika *parser* berhasil mencapai akhir *file* tanpa sisa token, metode `parse()` akan mengembalikan *ParseNode* akar (root) yang merepresentasikan seluruh *parse tree* program.
 - Gagal: Jika `expect()` gagal atau tidak ada aturan yang cocok, *ParseError* akan dilempar (raised), menghentikan proses *parsing* dan menampilkan pesan kesalahan yang telah diformat.

3.4 Justifikasi Pilihan Implementasi

- Recursive Descent (RD) Parsing: Metode ini dipilih karena merupakan salah satu teknik parsing top-down yang paling intuitif. Keuntungan utamanya adalah keterbacaan kode, di mana setiap aturan CFG dapat dipetakan langsung ke satu fungsi Python. Hal ini membuat logika parser mudah diikuti, di-debug, dan dikelola.

- Modularitas (Kelas Parser, ParseNode, ParseError): Memisahkan logika parser dari struktur data tree dan mekanisme error handling adalah praktik desain yang baik. Ini memungkinkan setiap komponen untuk diuji dan dimodifikasi secara independen.
- Custom Error (ParseError): Menggunakan exception kustom sangat penting. Daripada hanya gagal, parser dapat memberikan umpan balik yang sangat spesifik kepada pengguna (lokasi baris dan kolom) yang fundamental untuk pengalaman debugging yang baik.
- expect() vs check(): Penggunaan dua utilitas ini adalah pola standar dalam RD parser. check() memungkinkan pengambilan keputusan (misalnya, untuk parsing if vs while), sementara expect() menegakkan aturan tata bahasa secara ketat.

3.5 Aturan Grammar CFG Parser

CFG didefinisikan sebagai 4 tuple:

$$CFG = (V, \Sigma, P, S)$$

dengan:

$$V = \{$$

<program>, <program-header>, <declaration-part>,
<const-declaration>,
<type-declaration>, <type-definition>,
<var-declaration>,
<subprogram-declaration>, <procedure-declaration>, <function-declaration>,
<block>, <formal-parameter-list>, <parameter-group>,
<identifier-list>, <type>, <array-type>, <range>,
<compound-statement>, <statement-list>, <statement>,
<assignment-statement>, <if-statement>, <while-statement>, <for-statement>,
<procedure-call>, <parameter-list>, <empty-statement>,
<expression>, <simple-expression>, <term>, <factor>,
<function-call>, <relational-operator>, <additive-operator>,
<multiplicative-operator>, <logical-not>

}

$$\Sigma = \{$$

KEYWORD(program), KEYWORD(variabel), KEYWORD(mulai), KEYWORD(selesai),
KEYWORD(jika), KEYWORD(maka), KEYWORD(selain_itu), KEYWORD(selama),
KEYWORD(lakukan), KEYWORD(untuk), KEYWORD(ke), KEYWORD(turun_ke),
KEYWORD(integer), KEYWORD(real), KEYWORD(boolean), KEYWORD(char),
KEYWORD(larik), KEYWORD(dari), KEYWORD(true), KEYWORD(false),
KEYWORD(prosedur), KEYWORD(fungsi), KEYWORD(konstanta), KEYWORD(tipe),
KEYWORD(sampai), KEYWORD(ulangi), KEYWORD(rekaman), KEYWORD(kasus),

ARITHMETIC_OPERATOR(+), ARITHMETIC_OPERATOR(-),
ARITHMETIC_OPERATOR(*)

```

ARITHMETIC_OPERATOR(/), ARITHMETIC_OPERATOR(bagi),
ARITHMETIC_OPERATOR(mod),

ASSIGN_OPERATOR(:=),

RELATIONAL_OPERATOR(=), RELATIONAL_OPERATOR(<),
RELATIONAL_OPERATOR(>),
RELATIONAL_OPERATOR(<=), RELATIONAL_OPERATOR(>=),
RELATIONAL_OPERATOR(<>),

LOGICAL_OPERATOR(dan), LOGICAL_OPERATOR(atau),
LOGICAL_OPERATOR(tidak),
RANGE_OPERATOR(..),

SEMICOLON(;), COLON(:), COMMA(), DOT(.),
LPARENTHESIS(), RPARENTHESIS(),
LBRACKET([), RBRACKET(]),

IDENTIFIER(...),
IDENTIFIER(count_value),
NUMBER(...),
STRING_LITERAL(...),
CHAR_LITERAL(...)

}

```

$P = \{$

$\langle \text{program} \rangle \rightarrow \langle \text{program-header} \rangle \langle \text{declaration-part} \rangle \langle \text{compound-statement} \rangle \text{ DOT}$

$\langle \text{program-header} \rangle \rightarrow \text{KEYWORD(program)} \text{ IDENTIFIER SEMICOLON}$

$\langle \text{declaration-part} \rangle \rightarrow \langle \text{const-declaration} \rangle^* \langle \text{type-declaration} \rangle^* \langle \text{var-declaration} \rangle^*$

$\langle \text{subprogram-declaration} \rangle^*$

$\langle \text{const-declaration} \rangle \rightarrow \text{KEYWORD(konstanta)} (\text{IDENTIFIER}$

$\text{RELATIONAL_OPERATOR}(=) (\text{NUMBER}$

- | STRING_LITERAL
- | CHAR_LITERAL
- | IDENTIFIER
- | KEYWORD(true)
- | KEYWORD(false)) SEMICOLON)+

$\langle \text{type-declaration} \rangle \rightarrow \text{KEYWORD(tipe)} \langle \text{type-definition} \rangle^+$

<type-definition> → IDENTIFIER RELATIONAL _ OPERATOR(=) <type> SEMICOLON
 <var-declaration> → KEYWORD(variabel) (<identifier-list> COLON <type>
 SEMICOLON)+
 <identifier-list> → IDENTIFIER (COMMA IDENTIFIER)*
 <subprogram-declaration> → <procedure-declaration> | <function-declaration>
 <procedure-declaration> → KEYWORD(prosedur) IDENTIFIER <formal-parameter-list>?
 SEMICOLON <block> SEMICOLON
 <function-declaration> → KEYWORD(fungsi) IDENTIFIER <formal-parameter-list>?
 COLON <type> SEMICOLON <block> SEMICOLON
 <block> → <declaration-part> <compound-statement>
 <formal-parameter-list> → LPARENTHESIS <parameter-group> (SEMICOLON
 <parameter-group>)* RPARENTHESIS
 <parameter-group> → <identifier-list> COLON <type>
 <type> → KEYWORD(integer)
 | KEYWORD(real)
 | KEYWORD(boolean)
 | KEYWORD(char)
 | <array-type>
 | IDENTIFIER
 <array-type> → KEYWORD(larik) LBRACKET <range> RBRACKET KEYWORD(dari)
 <type>
 <range> → <simple-expression> RANGE _ OPERATOR(..) <simple-expression>
 <compound-statement> → KEYWORD(mulai) <statement-list> KEYWORD(selesai)
 <statement-list> → <statement> (SEMICOLON <statement>)*
 <statement> → <assignment-statement>
 | <if-statement>
 | <while-statement>
 | <for-statement>
 | <compound-statement>
 | <procedure-call>

| <empty-statement>

<empty-statement> → ε

<assignment-statement> → IDENTIFIER ASSIGN_OPERATOR(:=) <expression>

<if-statement> → KEYWORD(jika) <expression> KEYWORD(maka) <statement>
(KEYWORD(selain-itu) <statement>)?

<while-statement> → KEYWORD(selama) <expression> KEYWORD(lakukan)
<statement>

<for-statement> → KEYWORD(untuk) IDENTIFIER (ASSIGN_OPERATOR
<expression>)? (KEYWORD(ke) | KEYWORD(turun_ke)) <expression>
KEYWORD(lakukan) <statement>

<procedure-call> → (IDENTIFIER | KEYWORD) (LPARENTHESIS <parameter-list>?
RPARENTHESIS)?

<parameter-list> → <expression> (COMMA <expression>)*

<expression> → <simple-expression> (<relational-operator> <simple-expression>)?

<simple-expression> → (ARITHMETIC_OPERATOR(+|-))? <term> (<additive-operator>
<term>)*

<term> → <factor> (<multiplicative-operator> <factor>)*

<factor> → IDENTIFIER

- | NUMBER
- | STRING_LITERAL
- | CHAR_LITERAL
- | KEYWORD(true)
- | KEYWORD(false)
- | LPARENTHESIS <expression> RPARENTHESIS
- | <logical-not> <factor>
- | <function-call>

<function-call> → IDENTIFIER LPARENTHESIS <parameter-list>? RPARENTHESIS

<relational-operator> → RELATIONAL_OPERATOR(=)

- | RELATIONAL_OPERATOR(<>)
- | RELATIONAL_OPERATOR(<)
- | RELATIONAL_OPERATOR(<=)

| RELATIONAL_OPERATOR(>)
| RELATIONAL_OPERATOR(>=)

<additive-operator> → ARITHMETIC_OPERATOR(+)
| ARITHMETIC_OPERATOR(-)
| KEYWORD(atau)
| LOGICAL_OPERATOR(atau)

<multiplicative-operator> → ARITHMETIC_OPERATOR(*)
| ARITHMETIC_OPERATOR(/)
| KEYWORD(bagi)
| KEYWORD(mod)
| KEYWORD(dan)
| LOGICAL_OPERATOR(dan)

<logical-not> → KEYWORD(tidak) | LOGICAL_OPERATOR(tidak)
}

S = <program>

BAB IV

IMPLEMENTASI PROGRAM

4.1 Parse Node

Kelas ini menjadi struktur utama dari parser dalam membentuk dan memformat parse tree.

Class	Penjelasan Singkat
<pre>class ParseNode: def __init__(self, type: str): self.type = type self.child = [] def add_child(self, node: "ParseNode"): self.child.append(node) def __str__(self, level=0, prefix=""): '''Parse Tree Output''' lines = [] if level == 0: lines.append(f"{self.type}") else: lines.append(f"{prefix}└ {self.type}") child_count = len(self.child) for i, c in enumerate(self.child): is_last = (i == child_count - 1) connector = "└ " if is_last else " " next_prefix = prefix + (" " if is_last else " ") if isinstance(c, ParseNode): lines.append(f"{prefix}{connector}{c.type}") child_lines = c.__str__(level + 1, next_prefix).splitlines() lines.extend(child_lines[1:]) else: if hasattr(c, "type") and hasattr(c, "value"): lines.append(f"{prefix}{connector}{c.type}({c.value})") else: lines.append(f"{prefix}{connector}{str(c)}") return "\n".join(lines)</pre>	ParseNode adalah kelas yang merepresentasikan satu simpul dalam parse tree. Setiap node menyimpan jenis aturan produksi atau token yang dikenali parser, serta daftar child yang menggambarkan struktur hirarkis grammar. Node dibuat setiap kali parser memproses suatu bagian grammar, dan anak-anaknya ditambahkan sesuai urutan aturan produksi yang terbentuk. Selain menyimpan struktur, ParseNode juga menyediakan fungsi representasi teks melalui <code>__str__()</code> , yang menampilkan tree dalam format bertingkat menggunakan indentasi, sehingga membantu visualisasi hasil parsing.

4.2 Parse Error

Kelas ini mengatasi segala syntax error yang ada pada program input pascal-S

Class	Penjelasan Singkat
<pre>●●● class ParseError(Exception): def __init__(self, message, token): self.message = message self.line = token.line self.column = token.column self.value = token.value self.type = token.type self.full_source_text = None def __str__(self): if not self.full_source_text: return f"SyntaxError at line {self.line}, column {self.column}:\n{self.message} (Token: {self.value})" lines = self.full_source_text.split('\n') error_line = lines[self.line - 1] if 0 < self.line <= len(lines) else "" error_line = error_line.rstrip() line_num_str = str(self.line) code_prefix = f"{' ' * len(line_num_str)} " empty_prefix = f"{' ' * len(line_num_str)} " caret_prefix = f"{' ' * len(line_num_str)} " caret_padding = " " * max(0, self.column - 1) return (f"SyntaxError: {self.message}\n" f" --> ({line_num_str}, {self.column})\n" f" {empty_prefix}{error_line}\n" f" {caret_prefix}{error_line}{caret_padding}^"))</pre>	ParseError adalah kelas exception khusus yang digunakan parser untuk melaporkan kesalahan sintaks saat token yang dibaca tidak sesuai dengan aturan grammar. Objek ini menyimpan informasi detail mengenai error, seperti pesan, jenis token, nilai token, serta posisi baris dan kolom tempat error terjadi. Ketika error dicetak, kelas ini membentuk pesan yang informatif lengkap dengan highlight posisi kesalahan pada source code, sehingga memudahkan proses debugging dan identifikasi letak kesalahan dalam input Pascal-S pengguna.

4.3 Parser

Pada aplikasi syntax analysis Pascal-S, parser merupakan komponen utama dalam memecah list token menjadi sebuah parse tree.

Functions	Penjelasan Singkat
<pre>●●● def advance(self): '''Advance 1 offset dari tokens list''' self.pos += 1 if self.pos < len(self.tokens): self.current_token = self.tokens[self.pos] else: self.current_token = None</pre>	Digunakan untuk memindahkan pointer token ke posisi berikutnya dengan memperbarui current_token.
<pre>●●● def peek(self, offset=1): '''Melihat token selanjutnya tanpa consume''' peek_pos = self.pos + offset if peek_pos < len(self.tokens): return self.tokens[peek_pos] return None</pre>	Digunakan untuk melihat token di depan tanpa mengonsumsi token tersebut.

```

● ● ●

def expect(self, expected_type, expected_value=None):
    '''Expected token untuk suatu aturan produksi'''
    token = self.current_token
    if token is None:
        raise ParseError(f"Unexpected end of input, expected {expected_type}", self.tokens[-1])

    if token.type != expected_type:
        raise ParseError(f"Unexpected token {token.type} ({token.value}), expected {expected_type}", token)

    if expected_value and token.value.lower() != expected_value.lower():
        raise ParseError(f"Unexpected value '{token.value}', expected '{expected_value}'", token)

    self.advance()
    return token

```

Digunakan untuk memverifikasi bahwa token saat ini sesuai tipe/nilai yang diharapkan sebelum dikonsumsi.

```

● ● ●

def check(self, expected_type, expected_value=None):
    '''Compare current token with type and/or value'''
    if self.current_token is None:
        return False
    if self.current_token.type != expected_type:
        return False
    if expected_value and self.current_token.value.lower() != expected_value.lower():
        return False
    return True

```

Digunakan untuk mengecek apakah token saat ini cocok dengan tipe/nilai tertentu tanpa konsumsi.

```

● ● ●

def parse(self):
    '''Main function caller'''
    node = self.parse_program()
    if self.current_token is not None:
        raise ParseError(f"Unexpected token {self.current_token.type} ({self.current_token.value})", self.current_token)
    return node

```

Digunakan untuk memulai seluruh proses parsing dan memastikan tidak ada token tersisa setelah program selesai di-parse.

```

● ● ●

def parse_program(self):
    '''Root node parse: header + declaration + compound statement + ending dot'''
    node = ParseNode("<program>")
    node.add_child(self.parse_program_header())
    node.add_child(self.parse_declaration_part())
    node.add_child(self.parse_compound_statement())
    node.add_child(self.expect("DOT"))
    return node

```

Digunakan untuk mem-parse struktur program utama berupa header, deklarasi, blok utama, dan tanda titik akhir program.

```

● ● ●

def parse_program_header(self):
    '''Header node parser'''
    node = ParseNode("<program-header>")
    node.add_child(self.expect("KEYWORD", "program"))
    node.add_child(self.expect("IDENTIFIER"))
    node.add_child(self.expect("SEMICOLON"))
    return node

```

Digunakan untuk mem-parse bagian header program yang terdiri atas format di bawah ini.

(program <identifier>;)

```

●●●

def parse_declaration_part(self):
    '''Strict urutan initialization dari pascal-s: const -> type -> var ->
    subprogram'''
    node = ParseNode("<declaration-part>")
    while self.check("KEYWORD", "konstanta"):
        node.add_child(self.parse_const_declaration())
    while self.check("KEYWORD", "tipe"):
        node.add_child(self.parse_type_declaration())
    while self.check("KEYWORD", "varlabel"):
        node.add_child(self.parse_var_declaration())
    while self.check("KEYWORD", "prosedur") or self.check("KEYWORD", "fungsi"):
        node.add_child(self.parse_subprogram_declaration())
    return node

```

Digunakan untuk mem-parse seluruh bagian deklarasi program secara terurut sesuai aturan Pascal-S.

```

●●●

def parse_const_declaration(self):
    '''Parse const declaration'''
    node = ParseNode("<const-declaration>")
    node.add_child(self.expect("KEYWORD", "konstanta"))

    while True:
        node.add_child(self.expect("IDENTIFIER"))

        if self.check("RELATIONAL_OPERATOR") and self.current_token.value == "=":
            node.add_child(self.expect("RELATIONAL_OPERATOR"))
        else:
            raise ParseError(f"Expected '=' in constant declaration",
                            self.current_token)

        node.add_child(self.parse_expression())
        node.add_child(self.expect("SEMICOLON"))

        if not self.check("IDENTIFIER"):
            break
    return node

```

Digunakan untuk mem-parse deklarasi konstanta berupa pasangan format di bawah ini.

(<identifier> = <expression>)

```

●●●

def parse_type_declaration(self):
    '''Parse type declaration'''
    node = ParseNode("<tipe-declaration>")
    node.add_child(self.expect("KEYWORD", "tipe"))

    while True:
        node.add_child(self.expect("IDENTIFIER"))

        if self.check("RELATIONAL_OPERATOR") and self.current_token.value == "=":
            node.add_child(self.expect("RELATIONAL_OPERATOR"))
        else:
            raise ParseError(f"Expected '=' in type declaration", self.current_token)
        node.add_child(self.parse_type_definition())
        node.add_child(self.expect("SEMICOLON"))

        if not self.check("IDENTIFIER"):
            break
    return node

```

Digunakan untuk mem-parse deklarasi tipe baru beserta definisinya.

```

●●●

def parse_var_declaration(self):
    '''Parse variable declaration'''
    node = ParseNode("<var-declaration>")
    node.add_child(self.expect("KEYWORD", "variabel"))

    while True:
        node.add_child(self.parse_identifier_list())
        node.add_child(self.expect("COLON"))
        node.add_child(self.parse_type())
        node.add_child(self.expect("SEMICOLON"))

        if not self.check("IDENTIFIER"):
            break
    return node

```

Digunakan untuk mem-parse deklarasi variabel beserta tipe yang digunakan.

<pre><code>● ● ● def parse_subprogram_declaration(self): if self.check("KEYWORD", "prosedur"): return self.parse_procedure_declaration() elif self.check("KEYWORD", "fungsi"): return self.parse_function_declaration() else: raise ParseError(f"Expected 'prosedur' or 'fungsi'", self.current_token)</code></pre>	<p>Digunakan untuk memilih dan mem-parse deklarasi prosedur atau fungsi.</p>
<pre><code>● ● ● def parse_procedure_declaration(self): node = ParseNode("<procedure-declaration>") node.add_child(self.expect("KEYWORD", "prosedur")) node.add_child(self.expect("IDENTIFIER")) #parse parameter list if self.check("LPARENTHESIS"): node.add_child(self.parse_formal_parameter_list()) node.add_child(self.expect("SEMICOLON")) #parse block node.add_child(self.parse_block()) node.add_child(self.expect("SEMICOLON")) return node</code></pre>	<p>Digunakan untuk mem-parse deklarasi prosedur termasuk parameter, blok, dan terminator.</p>
<pre><code>● ● ● def parse_function_declaration(self): node = ParseNode("<function-declaration>") node.add_child(self.expect("KEYWORD", "fungsi")) node.add_child(self.expect("IDENTIFIER")) #parse parameter list if self.check("LPARENTHESIS"): node.add_child(self.parse_formal_parameter_list()) #parse return type node.add_child(self.expect("COLON")) node.add_child(self.parse_type()) node.add_child(self.expect("SEMICOLON")) #parse block node.add_child(self.parse_block()) node.add_child(self.expect("SEMICOLON")) return node</code></pre>	<p>Digunakan untuk mem-parse deklarasi fungsi lengkap dengan parameter, tipe kembalian, dan blok isi.</p>
<pre><code>● ● ● def parse_block(self): node = ParseNode("<block>") node.add_child(self.parse_declaration_part()) node.add_child(self.parse_compound_statement()) return node</code></pre>	<p>Digunakan untuk mem-parse satu blok program yang berisi deklarasi dan compound-statement.</p>

<pre> ● ● ● def parse_formal_parameter_list(self): node = ParseNode("<formal-parameter-list>") node.add_child(self.expect("LPARENTHESIS")) node.add_child(self.parse_parameter_group()) while self.check("SEMICOLON"): node.add_child(self.expect("SEMICOLON")) node.add_child(self.parse_parameter_group()) node.add_child(self.expect("RPARENTHESIS")) return node </pre>	<p>Digunakan untuk mem-parse daftar parameter formal pada deklarasi prosedur atau fungsi.</p>
<pre> ● ● ● def parse_parameter_group(self): '''Helper function for formal-parameter-list''' node = ParseNode("<parameter-group>") node.add_child(self.parse_identifier_list()) node.add_child(self.expect("COLON")) node.add_child(self.parse_type()) return node </pre>	<p>Digunakan untuk mem-parse satu kelompok parameter dalam format di bawah ini.</p> <p>(<identifier-list> : <type>)</p>
<pre> ● ● ● def parse_identifier_list(self): '''Identifier, Identifier, Identifier''' node = ParseNode("<identifier-list>") node.add_child(self.expect("IDENTIFIER")) while self.check("COMMA"): node.add_child(self.expect("COMMA")) node.add_child(self.expect("IDENTIFIER")) return node </pre>	<p>Digunakan untuk mem-parse daftar identifier yang dipisahkan oleh koma.</p>
<pre> ● ● ● def parse_type_definition(self): '''tipe di type-declaration saja, supports range''' node = ParseNode("<type-definition>") # RANGE TYPE (expression .. expression) if self.lookahead_is_range(): left_expr = self.parse_expression() node.add_child(left_expr) node.add_child(self.expect("RANGE_OPERATOR")) node.add_child(self.parse_expression()) return node # ARRAY TYPE if self.check("KEYWORD", "larik"): node.add_child(self.parse_array_type()) return node # RECORD TYPE if self.check("KEYWORD", "rekaman"): node.add_child(self.parse_record_type()) return node # BUILTIN TYPE if self.check("KEYWORD", "integer") or self.check("KEYWORD", "real") or \ self.check("KEYWORD", "boolean") or self.check("KEYWORD", "char"): node.add_child(self.expect("KEYWORD")) return node # CUSTOM TYPE if self.check("IDENTIFIER"): node.add_child(self.expect("IDENTIFIER")) return node raise ParseError("Invalid type-definition", self.current_token) </pre>	<p>Digunakan untuk mem-parse definisi tipe yang dapat berupa range, array, record, built-in, atau custom type.</p>

```

● ● ●

def parse_type(self):
    '''parse type di variable declaration or function return type (ga ada range)'''
    node = ParseNode("<type>")

    #array type
    if self.check("KEYWORD", "larik"):
        node.add_child(self.parse_array_type())
        return node

    #rekaman
    elif self.check("KEYWORD", "rekaman"):
        node.add_child(self.parse_record_type())
        return node

    #builtin type
    elif self.check("KEYWORD", "integer") or self.check("KEYWORD", "real") or \
        self.check("KEYWORD", "boolean") or self.check("KEYWORD", "char"):
        node.add_child(self.expect("KEYWORD"))
        return node

    #custom type (identifier)
    if self.check("IDENTIFIER"):
        node.add_child(self.expect("IDENTIFIER"))
        return node

    raise ParseError(f"Expected type", self.current_token)

```

Digunakan untuk mem-parse tipe data pada deklarasi variabel atau tipe kembalian fungsi.

```

● ● ●

def parse_array_type(self):
    node = ParseNode("<array-type>")
    node.add_child(self.expect("KEYWORD", "larik"))
    node.add_child(self.expect("LBRACKET"))
    node.add_child(self.parse_range())
    node.add_child(self.expect("RBRACKET"))
    node.add_child(self.expect("KEYWORD", "dari"))
    node.add_child(self.parse_type())
    return node

```

Digunakan untuk mem-parse tipe array dengan format di bawah ini.

(larik[range] dari <type>)

```

● ● ●

def parse_record_type(self):
    node = ParseNode("<record-type>")
    node.add_child(self.expect("KEYWORD", "rekaman"))
    node.add_child(self.parse_parameter_group())

    while self.check("SEMICOLON"):
        node.add_child(self.expect("SEMICOLON"))
        if self.check("KEYWORD", "selesai"):
            break
        node.add_child(self.parse_parameter_group())

    node.add_child(self.expect("KEYWORD", "selesai"))
    return node

```

Digunakan untuk mem-parse tipe record yang berisi field-field dan blok berakhir dengan kata “selesai”.

```

● ● ●

def parse_variable(self):
    '''parse variable, dot chaining + indexing'''
    node = ParseNode("<variable>")

    identifier = self.expect("IDENTIFIER")
    node.add_child(identifier)

    while True:
        if self.check("DOT"):
            node.add_child(self.expect("DOT"))
            node.add_child(self.expect("IDENTIFIER"))

        elif self.check("LBRACKET"):
            node.add_child(self.parse_variable_index())

        else:
            break

    return node

```

Digunakan untuk mem-parse variabel termasuk chaining field (.) dan index array ([...]).

```

● ● ●

def parse_variable_index(self):
    '''Array index access: [expr]'''
    node = ParseNode("<variable-index>")

    node.add_child(self.expect("LBRACKET"))
    node.add_child(self.parse_expression())
    node.add_child(self.expect("RBRACKET"))

    return node

```

Digunakan untuk mem-parse index array berupa [expression].

```

● ● ●

def parse_range(self):
    node = ParseNode("<range>")
    node.add_child(self.parse_expression())

    if self.check("RANGE_OPERATOR"):
        node.add_child(self.expect("RANGE_OPERATOR"))
    else:
        raise ParseError("Expected '..' for range",
                         self.current_token)

    node.add_child(self.parse_expression())
    return node

```

Digunakan untuk mem-parse range dalam format di bawah ini.

(<expression> .. <expression>)

```

● ● ●

def parse_compound_statement(self):
    node = ParseNode("<compound-statement>")
    node.add_child(self.expect("KEYWORD", "mulai"))
    node.add_child(self.parse_statement_list())
    node.add_child(self.expect("KEYWORD", "selesai"))
    return node

```

Digunakan untuk mem-parse blok mulai .. selesai, yang berisi satu atau lebih statement.

```

def parse_case_statement(self):
    node = ParseNode("<case-statement>")
    node.add_child(self.expect("KEYWORD", "kasus"))
    node.add_child(self.parse_expression())
    node.add_child(self.expect("KEYWORD", "dari"))

    case_list = ParseNode("<case-list>")

    while True:
        if (self.check("NUMBER") or self.check("CHAR_LITERAL") or self.check("STRING_LITERAL") or \
            self.check("KEYWORD", "true") or self.check("KEYWORD", "false")):
            case_list.add_child(self.expect(self.current_token.type))
        else:
            raise ParseError('Expected constant in `kasus` statement', self.current_token)

        if self.check("COMMA"):
            self.expect("COMMA")
        else:
            case_list.add_child(self.parse_statement())

            if self.check("SEMICOLON"):
                case_list.add_child(self.expect("SEMICOLON"))
                if not (self.check("NUMBER") or self.check("CHAR_LITERAL") or self.check("STRING_LITERAL") or \
                    self.check("KEYWORD", "true") or self.check("KEYWORD", "false")):
                    break
            else:
                break

    node.add_child(case_list)
    return node

```

Digunakan untuk mem-parse statement “kasus” yang berisi daftar kondisi dan statement terkaitnya.

```

def parse_statement_list(self):
    '''List of statements parser'''
    node = ParseNode("<statement-list>")
    node.add_child(self.parse_statement())

    while self.check("SEMICOLON"):
        node.add_child(self.expect("SEMICOLON"))
        if self.check("KEYWORD", "selesai"):
            break
        node.add_child(self.parse_statement())
    return node

```

Digunakan untuk mem-parse rangkaian statement yang dipisah oleh semicolon.

```

def parse_statement(self):
    '''Individual statement parser'''
    # Statement kosong
    if self.check("SEMICOLON") or self.check("KEYWORD", "selesai"):
        return ParseNode("<empty-statement>")

    # If statement
    if self.check("KEYWORD", "jika"):
        return self.parse_if_statement()
    # While statement
    if self.check("KEYWORD", "selama"):
        return self.parse_while_statement()
    # For statement
    if self.check("KEYWORD", "untuk"):
        return self.parse_for_statement()
    # Repeat statement
    if self.check("KEYWORD", "ulangi"):
        return self.parse_repeat_statement()
    # Compound statement
    if self.check("KEYWORD", "mulai"):
        return self.parse_compound_statement()
    # Case statement
    if self.check("KEYWORD", "kasus"):
        return self.parse_case_statement()
    # Caller / Assignment statement
    if self.check("IDENTIFIER"):
        next_token = self.peek()
        if next_token and next_token.type == "LPARENTHESIS":
            return self.parse_procedure_function_call()
        else:
            return self.parse_assignment_statement()
    # Built-in procedure/function
    if self.check("KEYWORD"):
        return self.parse_procedure_function_call()

    raise ParseError(f"Unexpected token in statement", self.current_token)

```

Digunakan untuk mengidentifikasi dan mem-parse satu statement sesuai token awalnya.

 <pre>def parse_assignment_statement(self): '''Assignment statement parser''' node = ParseNode("<assignment-statement>") node.add_child(self.parse_variable()) node.add_child(self.expect("ASSIGN_OPERATOR")) node.add_child(self.parse_expression()) return node</pre>	<p>Digunakan untuk mem-parse assignment dalam format di bawah ini.</p> <p>(<variable> := <expression>)</p>
 <pre>def parse_if_statement(self): node = ParseNode("<if-statement>") node.add_child(self.expect("KEYWORD", "jika")) node.add_child(self.parse_expression()) node.add_child(self.expect("KEYWORD", "maka")) node.add_child(self.parse_statement()) # parse else if self.check("KEYWORD", "selain_itu") or self.check("KEYWORD", "selain-itu"): node.add_child(self.expect("KEYWORD")) node.add_child(self.parse_statement()) return node</pre>	<p>Digunakan untuk mem-parse statement “jika”, “maka”, dan “selain_itu”</p>
 <pre>def parse_while_statement(self): node = ParseNode("<while-statement>") node.add_child(self.expect("KEYWORD", "selama")) node.add_child(self.parse_expression()) node.add_child(self.expect("KEYWORD", "lakukan")) node.add_child(self.parse_statement()) return node</pre>	<p>Digunakan untuk mem-parse perulangan (while loop) yang melibatkan kata “selama” dan “lakukan”</p>
 <pre>def parse_for_statement(self): node = ParseNode("<for-statement>") node.add_child(self.expect("KEYWORD", "untuk")) node.add_child(self.expect("IDENTIFIER")) node.add_child(self.expect("ASSIGN_OPERATOR")) node.add_child(self.parse_expression()) if self.check("KEYWORD", "ke"): node.add_child(self.expect("KEYWORD", "ke")) elif self.check("KEYWORD", "turun_ke"): node.add_child(self.expect("KEYWORD", "turun_ke")) else: raise ParseError("Expected 'ke' or 'turun_ke' in for-statement", self.current_token) node.add_child(self.parse_expression()) node.add_child(self.expect("KEYWORD", "lakukan")) node.add_child(self.parse_statement()) return node</pre>	<p>Digunakan untuk mem-parse perulangan (for loop) yang melibatkan kata “untuk”, “ke/turun_ke”, dan “lakukan”</p>

```

def parse_repeat_statement(self):
    node = ParseNode("<repeat-statement>")
    node.add_child(self.expect("KEYWORD", "ulangi"))
    stmt_list_node = ParseNode("<statement-list>")
    stmt_list_node.add_child(self.parse_statement())

    while self.check("SEMICOLON"):
        stmt_list_node.add_child(self.expect("SEMICOLON"))
        if self.check("KEYWORD", "sampai"):
            break
        stmt_list_node.add_child(self.parse_statement())

    node.add_child(stmt_list_node)
    node.add_child(self.expect("KEYWORD", "sampai"))
    node.add_child(self.parse_expression())
    return node

```

Digunakan untuk mem-parse loop (repeat until loop) yang melibatkan kata “ulangi” dan “sampai”, dilengkapi dengan sebuah <expression>.

```

def parse_procedure_function_call(self):
    '''Procedure/function call parser: name(args), name bisa
    identifier atau builtin keyword'''
    node = ParseNode("<procedure/function-call>")

    if self.check("IDENTIFIER"):
        node.add_child(self.expect("IDENTIFIER"))
    elif self.check("KEYWORD"):
        node.add_child(self.expect("KEYWORD"))
    else:
        raise ParseError(f"Expected procedure/function name",
                        self.current_token)

    # Parse parameter list (kalo ada args)
    if self.check("LPARENTHESIS"):
        node.add_child(self.expect("LPARENTHESIS"))
        if not self.check("RPARENTHESIS"):
            node.add_child(self.parse_parameter_list())
        node.add_child(self.expect("RPARENTHESIS"))

    return node

```

Digunakan untuk mem-parse pemanggilan prosedur atau fungsi dengan atau tanpa parameter.

```

def parse_parameter_list(self):
    node = ParseNode("<parameter-list>")
    node.add_child(self.parse_expression())

    while self.check("COMMA"):
        node.add_child(self.expect("COMMA"))
        node.add_child(self.parse_expression())
    return node

```

Digunakan untuk mem-parse daftar parameter aktual dalam pemanggilan fungsi/prosedur.

```

def parse_expression(self):
    '''Expression parser'''
    node = ParseNode("<expression>")

    node.add_child(self.parse_simple_expression())

    if self.is_relational_operator():
        node.add_child(self.parse_relational_operator())
        node.add_child(self.parse_simple_expression())
    return node

```

Digunakan untuk mem-parse ekspresi lengkap yang dapat mengandung operator relasional.

```

● ● ●

def parse_simple_expression(self):
    '''parse simple expression'''
    node = ParseNode("<simple-expression>")

    if self.check("ARITHMETIC_OPERATOR") and
    (self.current_token.value == "+" or
    self.current_token.value == "-"):

        node.add_child(self.expect("ARITHMETIC_OPERATOR"))

        node.add_child(self.parse_term())

    while self.is_additive_operator():
        node.add_child(self.parse_additive_operator())
        node.add_child(self.parse_term())

    return node

```

Digunakan untuk mem-parse ekspresi sederhana yang dapat memuat operator aditif.

```

● ● ●

def parse_term(self):
    '''Multiplicatio operations chain (* / mod dan)'''
    node = ParseNode("<term>")

    node.add_child(self.parse_factor())

    while self.is_multiplicative_operator():
        node.add_child(self.parse_multiplicative_operator())
        node.add_child(self.parse_factor())

    return node

```

Digunakan untuk mem-parse rangkaian operasi multiplikatif.

```

● ● ●

def parse_factor(self):
    '''parse factor'''
    node = ParseNode("<factor>")

    # Number literal
    if self.check("NUMBER"):
        node.add_child(self.expect("NUMBER"))

    # String literal
    elif self.check("STRING_LITERAL"):
        node.add_child(self.expect("STRING_LITERAL"))

    # Character literal
    elif self.check("CHAR_LITERAL"):
        node.add_child(self.expect("CHAR_LITERAL"))

    # Boolean literal (true/false)
    elif self.check("KEYWORD", "true") or self.check("KEYWORD", "false"):
        node.add_child(self.expect("KEYWORD"))

    # NOT operator
    elif self.check("LOGICAL_OPERATOR", "tidak") or self.check("KEYWORD", "tidak"):
        if self.check("LOGICAL_OPERATOR"):
            node.add_child(self.expect("LOGICAL_OPERATOR"))
        else:
            node.add_child(self.expect("KEYWORD"))
        node.add_child(self.parse_factor())

    # Ekspresi dalam tanda kurung
    elif self.check("LPARENTHESIS"):
        node.add_child(self.expect("LPARENTHESIS"))
        node.add_child(self.parse_expression())
        node.add_child(self.expect("RPARENTHESIS"))

    # Identifier (variable atau function/procedure call)
    elif self.check("IDENTIFIER"):
        next_token = self.peek()

        if next_token and next_token.type == "LPARENTHESIS":
            node.add_child(self.parse_procedure_function_call())
        else:
            node.add_child(self.parse_variable())
    else:
        raise ParseError(f"Unexpected token in factor", self.current_token)

    return node

```

Digunakan untuk mem-parse elemen dasar ekspresi seperti literal, variabel, kurungan, not, atau function call.

<pre><code>● ● ● def parse_relational_operator(self): if self.check("RELATIONAL_OPERATOR"): return self.expect("RELATIONAL_OPERATOR") else: raise ParseError(f"Expected relational operator", self.current_token)</code></pre>	<p>Digunakan untuk mem-parse operator relasional yang valid.</p>
<pre><code>● ● ● def parse_additive_operator(self): if self.check("ARITHMETIC_OPERATOR"): return self.expect("ARITHMETIC_OPERATOR") elif self.check("LOGICAL_OPERATOR", "atau"): return self.expect("LOGICAL_OPERATOR") else: raise ParseError(f"Expected additive operator", self.current_token)</code></pre>	<p>Digunakan untuk mem-parse operator aditif (+, -, atau)</p>
<pre><code>● ● ● def parse_multiplicative_operator(self): if self.check("ARITHMETIC_OPERATOR"): return self.expect("ARITHMETIC_OPERATOR") elif self.check("LOGICAL_OPERATOR", "dan"): return self.expect("LOGICAL_OPERATOR", "dan") else: raise ParseError(f"Expected multiplicative operator", self.current_token)</code></pre>	<p>Digunakan untuk mem-parse operator multiplikatif (*, /, mod, dan)</p>
<pre><code>● ● ● def is_relational_operator(self): if self.current_token is None: return False return self.check("RELATIONAL_OPERATOR") and self.current_token.value in ["<", ">", "<=", ">=", "=", "<>"]</code></pre>	<p>Digunakan untuk mengecek apakah token saat ini merupakan operator relasional.</p>
<pre><code>● ● ● def is_additive_operator(self): if self.current_token is None: return False return (self.check("ARITHMETIC_OPERATOR") and self.current_token.value in ["+", "-"]) or self.check("LOGICAL_OPERATOR", "atau")</code></pre>	<p>Digunakan untuk mengecek apakah token saat ini merupakan operator aditif.</p>
<pre><code>● ● ● def is_multiplicative_operator(self): if self.current_token is None: return False return (self.check("ARITHMETIC_OPERATOR") and self.current_token.value in ["*", "/", "bagi", "mod"]) or \ self.check("LOGICAL_OPERATOR", "dan")</code></pre>	<p>Digunakan untuk mengecek apakah token saat ini merupakan operator multiplicative.</p>

```
●●●

def lookahead_is_range(self):
    ''' detect range-type pattern <expr> .. <expr>, cuman dalam type-definition'''

    t1 = self.current_token
    t2 = self.peek()

    if t1 is None or t2 is None:
        return False

    # hanya allowed: NUMBER, IDENTIFIER, LPAREN
    if t1.type not in ("NUMBER", "IDENTIFIER", "LPARENTHESIS"):
        return False

    return t2.type == "RANGE_OPERATOR"
```

Digunakan untuk mendeteksi apakah pola range <expression> .. <expression> akan muncul pada type-definition.

BAB V

TEST CASE

5.1 Test Case Lanjutan Milestone-1

Tabel 5.1 Tabel test case (TC sama dengan milestone-1)

(dalam test/milestone-2/input: input-1.pas hingga input-5.pas)

No.	Input	Output
1	 <pre>program Hello; variabel a, b: integer; mulai a := 5; b := a + 10; writeln('Result = ', b); selesai.</pre>	<pre>[>] python main.py milestone-2/input/input-1.pas milestone-2 SAVED => test/milestone-2/output/output.txt ===== PARSE TREE OUTPUT ===== <program> <program-header> KEYWORD(program) IDENTIFIER(Hello) SEMICOLON(;) <declaration-part> <var-declaration> KEYWORD(variabel) <identifier-list> IDENTIFIER(a) COMMA(,) IDENTIFIER(b) COLON(:) <type> KEYWORD(integer) SEMICOLON(;) <compound-statement> KEYWORD(mulai) <statement-list> <assignment-statement> <variable> IDENTIFIER(a) ASSIGN_OPERATOR(:=) <expression> <simple-expression> <term> <factor> NUMBER(5)</pre>

		<pre> SEMICOLON(); <assignment-statement> <variable> IDENTIFIER(b) ASSIGN_OPERATOR(:=) <expression> <simple-expression> <term> <factor> <variable> IDENTIFIER(a) ARITHMETIC_OPERATOR(+) <term> <factor> NUMBER(10) SEMICOLON(); <procedure/function-call> IDENTIFIERwriteln() LPARENTHESIS() <parameter-list> <expression> <simple-expression> <term> <factor> STRING_LITERAL('Result = ') COMMA() <expression> <simple-expression> <term> <factor> <variable> IDENTIFIER(b) RPARENTHESIS() SEMICOLON(); KEYWORD(selesai) DOT(.) </pre>
2	 <pre> program Hello2; variabel a, b, i: integer; arr: larik[1 .. 5] dari integer; mulai a := 5; b := a + 10; writeln('Result = ', b); untuk i := 1 ke 5 lakukan mulai arr[i] := i + 1; writeln(arr[i]); selesai; selesai. </pre>	<pre> D] python main.py milestone-2/input/input-2.pas milestone-2 SAVED => test/milestone-2/output/output.txt ===== PARSE TREE OUTPUT ===== <program> <program-header> KEYWORD(program) IDENTIFIER(Hello2) SEMICOLON(); <declaration-part> <var-declaration> KEYWORD(variabel) <identifier-list> IDENTIFIER(a) COMMA() IDENTIFIER(b) COMMA() IDENTIFIER(i) COLON(:) <type> KEYWORD(integer) SEMICOLON(); <identifier-list> IDENTIFIER(arr) COLON(:) <type> <array-type> KEYWORD(larik) LBRAKET([) <range> <expression> <simple-expression> <term> <factor> NUMBER(1) </pre>

```
RANGE_OPERATOR(..)
  <expression>
    <simple-expression>
      <term>
        <factor>
          <NUMBER(5)>

        RBRACKET()]
        KEYWORD(dari)
        <type>
          <KEYWORD(integer)>

        SEMICOLON(;)

      <compound-statement>
        KEYWORD(mulai)
        <statement-list>
          <assignment-statement>
            <variable>
              <IDENTIFIER(a)>
            ASSIGN_OPERATOR(:=)
            <expression>
              <simple-expression>
                <term>
                  <factor>
                    <NUMBER(5)>

            SEMICOLON();
            <assignment-statement>
              <variable>
                <IDENTIFIER(b)>
              ASSIGN_OPERATOR(:=)
              <expression>
                <simple-expression>
                  <term>
                    <factor>
                      <variable>
                        <IDENTIFIER(a)>
                      ARITHMETIC_OPERATOR(+)
                  <term>
                    <factor>
                      <NUMBER(10)>

      SEMICOLON();>
      <procedure/function-call>
        <IDENTIFIERwriteln)>
        LPARENTHESIS(())
        <parameter-list>
          <expression>
            <simple-expression>
              <term>
                <factor>
                  <STRING_LITERAL('Result = ')>

            COMMA()
            <expression>
              <simple-expression>
                <term>
                  <factor>
                    <variable>
                      <IDENTIFIER(b)>
                    RPARENTHESIS())
      SEMICOLON();
      <for-statement>
        KEYWORD(until)
        <IDENTIFIER(i)>
        ASSIGN_OPERATOR(:=)
        <expression>
          <simple-expression>
            <term>
              <factor>
                <NUMBER(1)>
        KEYWORD(ke)
        <expression>
          <simple-expression>
            <term>
              <factor>
                <NUMBER(5)>
```



3



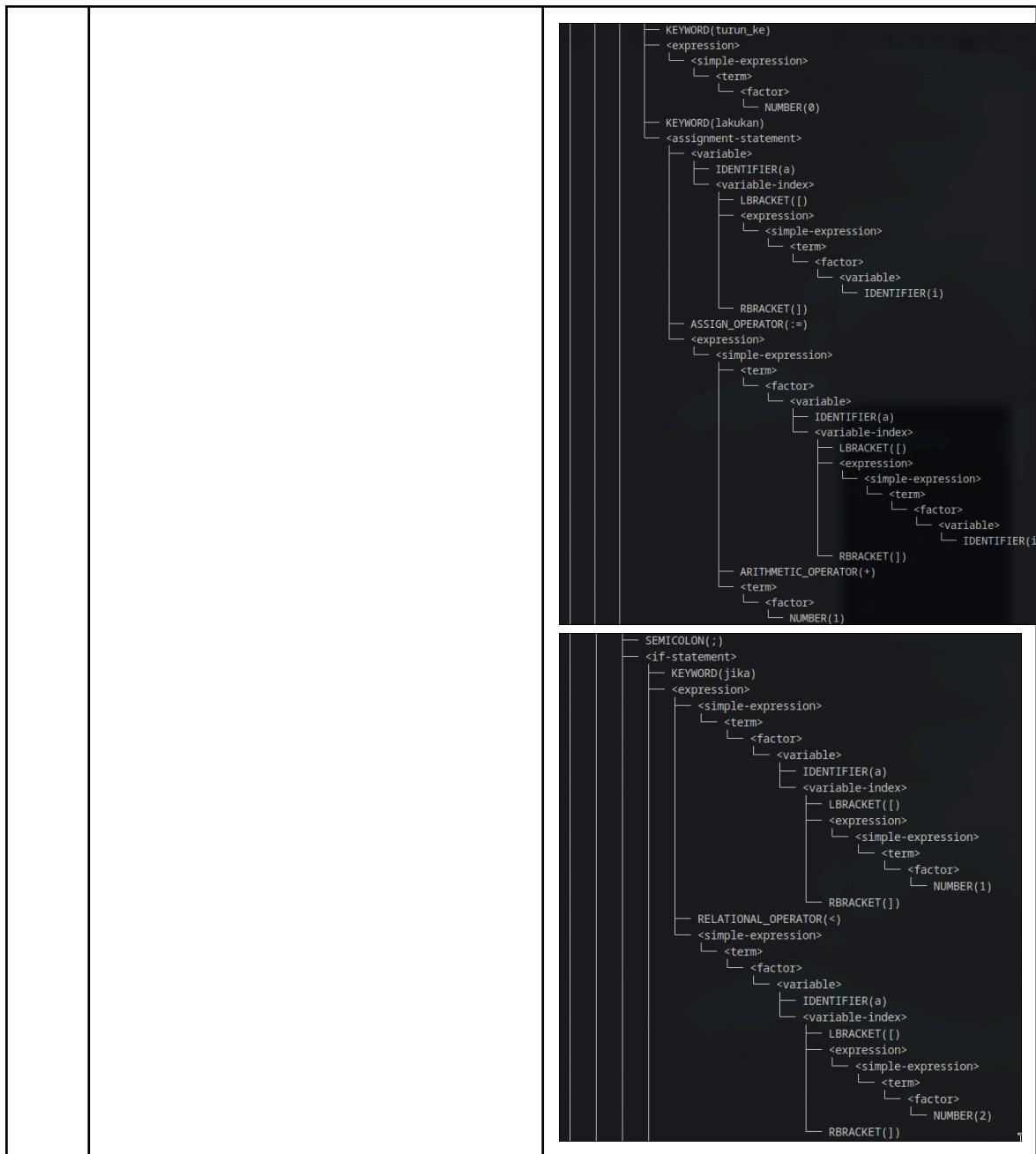
```
program RangesLoops;
variabel
  i: integer;
  a: larik[0 .. 3] dari integer;
mulai
  a[0] := 1;
  untuk i := 0 ke 3 lakukan
    a[i] := i;
  untuk i := 3 turun_ke 0 lakukan
    a[i] := a[i] + 1;
  jika a[1] < a[2] maka
    a[1] := a[2];
  selama i > 0 lakukan
    i := i - 1;
  writeln('ok', a[1]);
selesai.
```

```
[D] python main.py milestone-2/input/input-3.pas
milestone-2
SAVED    =>      test/milestone-2/output/output.txt

===== PARSE TREE OUTPUT =====

<program>
  └── <program-header>
    ├── KEYWORD(program)
    ├── IDENTIFIER(RangesLoops)
    └── SEMICOLON(;)
  └── <declaration-part>
    └── <var-declaration>
      ├── KEYWORD(variabel)
      ├── <identifier-list>
      │   └── IDENTIFIER(1)
      ├── COLON(:)
      ├── <type>
      │   └── KEYWORD(integer)
      ├── SEMICOLON(;)
      ├── <identifier-list>
      │   └── IDENTIFIER(a)
      ├── COLON(:)
      ├── <type>
      │   └── <array-type>
      │       ├── KEYWORD(larik)
      │       ├── LBRACKET([)
      │       └── <range>
      │           ├── <expression>
      │           │   └── <simple-expression>
      │           │       └── <term>
      │           │           └── <factor>
      │           │               └── NUMBER(0)
      │           └── RANGE_OPERATOR(..)
      │           └── <expression>
      │               └── <simple-expression>
      │                   └── <term>
      │                       └── <factor>
      │                           └── NUMBER(3)
```



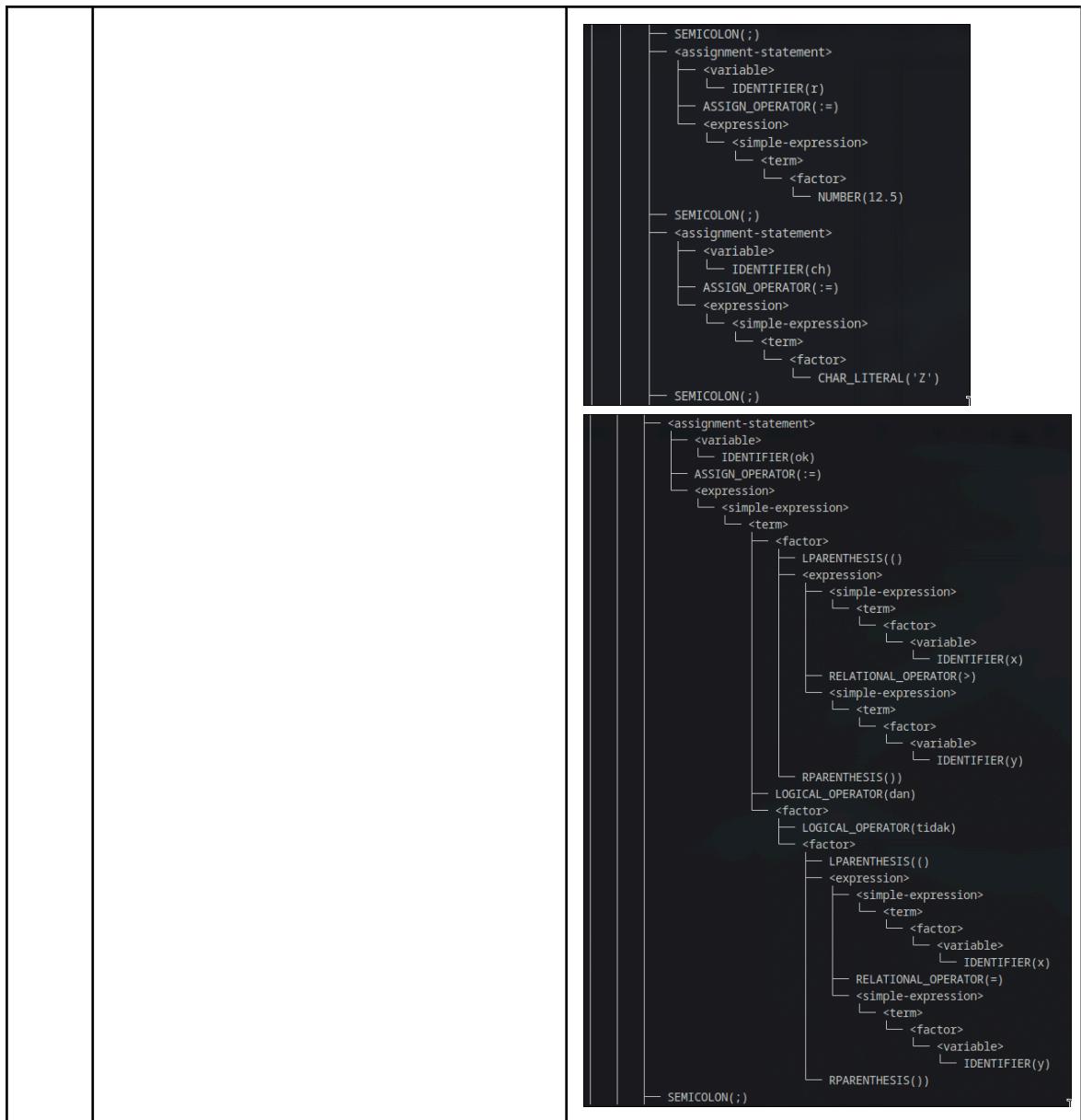


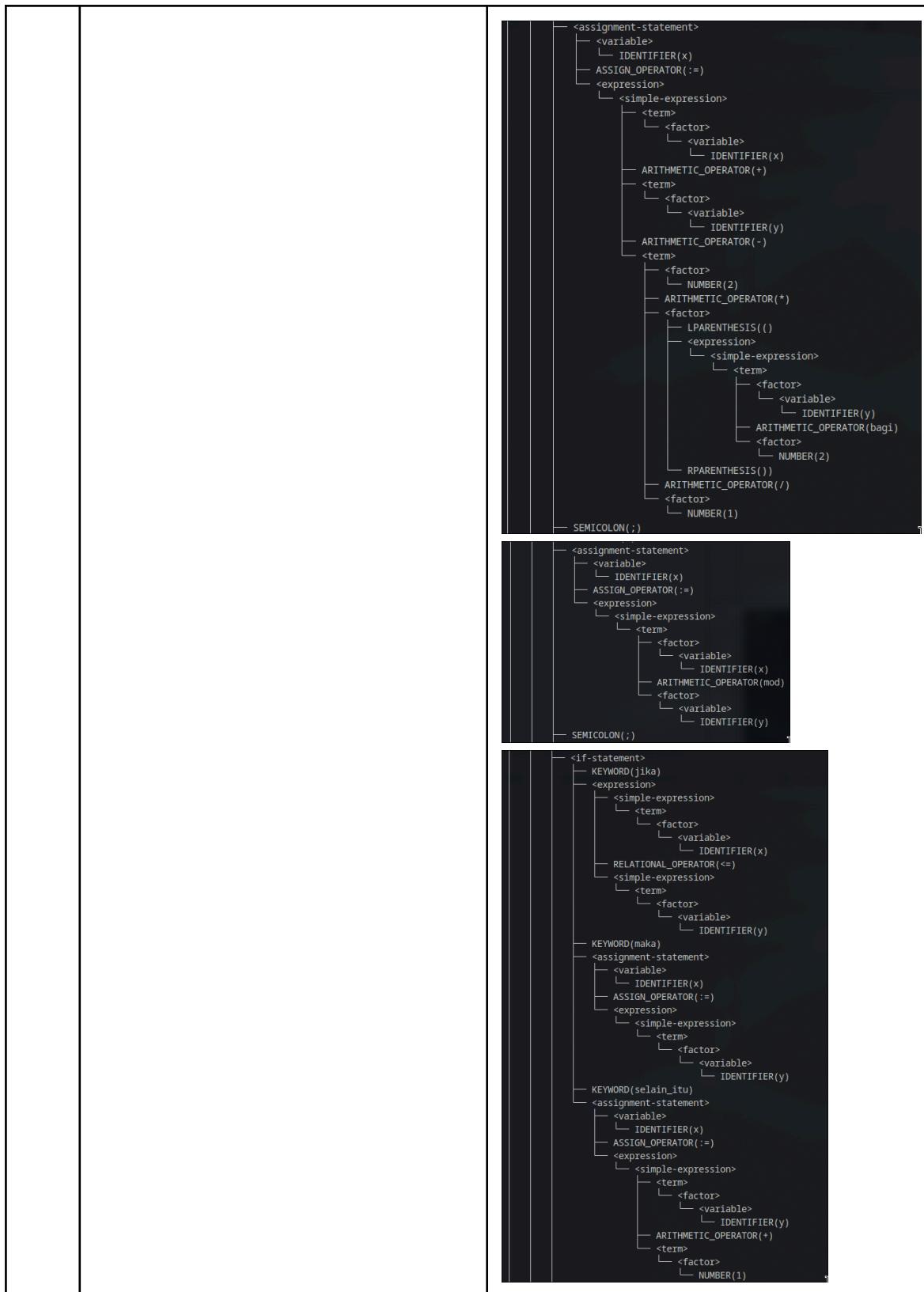


4

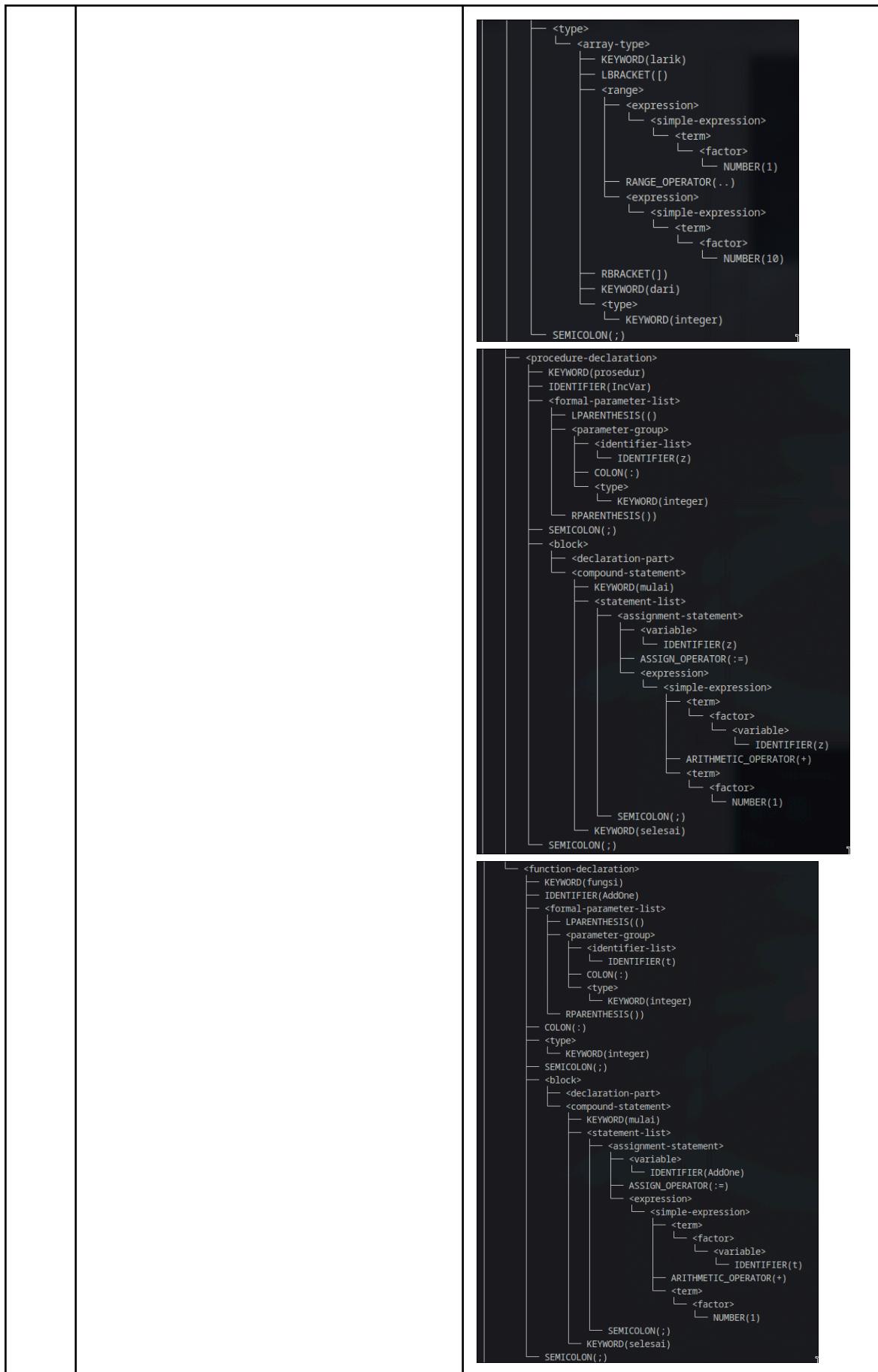
```
● ● ●  
program Ops;  
variabel  
    x, y: integer;  
    r: real;  
    ch: char;  
    ok: boolean;  
mulai  
    x := 10;  
    y := 3;  
    r := 12.5;  
    ch := 'Z';  
    ok := (x > y) dan tidak (x = y);  
    x := x + y - 2 * (y bagi 2) / 1;  
    x := x mod y;  
    jika x <= y maka  
        x := y  
    selain_itu  
        x := y + 1;  
    writeln('done ', x);  
selesai.
```

```
[!] python main.py milestone-2/input/input-4.pas  
milestone-2  
SAVED  =>      test/milestone-2/output/output.txt  
===== PARSE TREE OUTPUT =====  
  
<program>  
└── <program-header>  
    ├── KEYWORD(program)  
    ├── IDENTIFIER(Ops)  
    └── SEMICOLON();  
└── <declaration-part>  
    └── <var-declaration>  
        ├── KEYWORD(variabel)  
        ├── <identifier-list>  
        │   ├── IDENTIFIER(x)  
        │   ├── COMMA()  
        │   └── IDENTIFIER(y)  
        ├── COLON(:)  
        └── <type>  
            └── KEYWORD(integer)  
        └── SEMICOLON();  
        └── <identifier-list>  
            └── IDENTIFIER(r)  
        └── COLON(:)  
        └── <type>  
            └── KEYWORD(real)  
        └── SEMICOLON();  
        └── <identifier-list>  
            └── IDENTIFIER(ch)  
        └── COLON(:)  
        └── <type>  
            └── KEYWORD(char)  
        └── SEMICOLON();  
        └── <identifier-list>  
            └── IDENTIFIER(ok)  
        └── COLON(:)  
        └── <type>  
            └── KEYWORD(boolean)  
        └── SEMICOLON();  
  
└── <compound-statement>  
    ├── KEYWORD(mulai)  
    ├── <statement-list>  
    │   └── <assignment-statement>  
    │       └── <variable>  
    │           └── IDENTIFIER(x)  
    │       └── ASSIGN_OPERATOR(:=)  
    │       └── <expression>  
    │           └── <simple-expression>  
    │               └── <term>  
    │                   └── <factor>  
    │                       └── NUMBER(10)  
    └── SEMICOLON();  
    └── <assignment-statement>  
        └── <variable>  
            └── IDENTIFIER(y)  
        └── ASSIGN_OPERATOR(:=)  
        └── <expression>  
            └── <simple-expression>  
                └── <term>  
                    └── <factor>  
                        └── NUMBER(3)
```





		<pre> └── SEMICOLON(;) └── <procedure/function-call> └── IDENTIFIER(writeln) └── LPARENTHESIS() └── <parameter-list> └── <expression> └── <simple-expression> └── <term> └── <factor> └── STRING_LITERAL('done ') └── COMMA(,) └── <expression> └── <simple-expression> └── <term> └── <factor> └── <variable> └── IDENTIFIER(x) └── RPARENTHESIS() └── SEMICOLON(;) └── KEYWORD(selesai) └── DOT(..) </pre>
5	 <pre> program DeclProcFunc; konstanta K = 10; variabel i: integer; r: real; ch: char; ok: boolean; arr: larik[1 .. 10] dari integer; prosedur IncVar(z: integer); mulai z := z + 1; selesai; fungsi AddOne(t: integer): integer; mulai AddOne := t + 1; selesai; mulai r := 3.14; ch := 'A'; ok := true dan tidak false; arr[1] := 2; jika (arr[1] >= K) maka arr[1] := AddOne(arr[1]) selain_itu IncVar(arr[1]); writeln('Value: ', arr[1], '.'); selesai. </pre>	<pre> [D] python main.py milestone-2/input/input-5.pas milestone-2 SAVED => test/milestone-2/output/output.txt ===== PARSE TREE OUTPUT ===== <program> <program-header> └── KEYWORD(program) └── IDENTIFIER(DeclProcFunc) └── SEMICOLON(;) <declaration-part> <const-declaration> └── KEYWORD(konstanta) └── IDENTIFIER(K) └── RELATIONAL_OPERATOR(=) <expression> <simple-expression> <term> <factor> └── NUMBER(10) └── SEMICOLON(;) <var-declaration> └── KEYWORD(variabel) <identifier-list> └── IDENTIFIER(i) └── COLON(:) <type> └── KEYWORD(integer) └── SEMICOLON(;) <identifier-list> └── IDENTIFIER(r) └── COLON(:) <type> └── KEYWORD(real) └── SEMICOLON(;) <identifier-list> └── IDENTIFIER(ch) └── COLON(:) <type> └── KEYWORD(char) └── SEMICOLON(;) <identifier-list> └── IDENTIFIER(ok) └── COLON(:) <type> └── KEYWORD(boolean) └── SEMICOLON(;) <identifier-list> └── IDENTIFIER(arr) └── COLON(:) </pre>



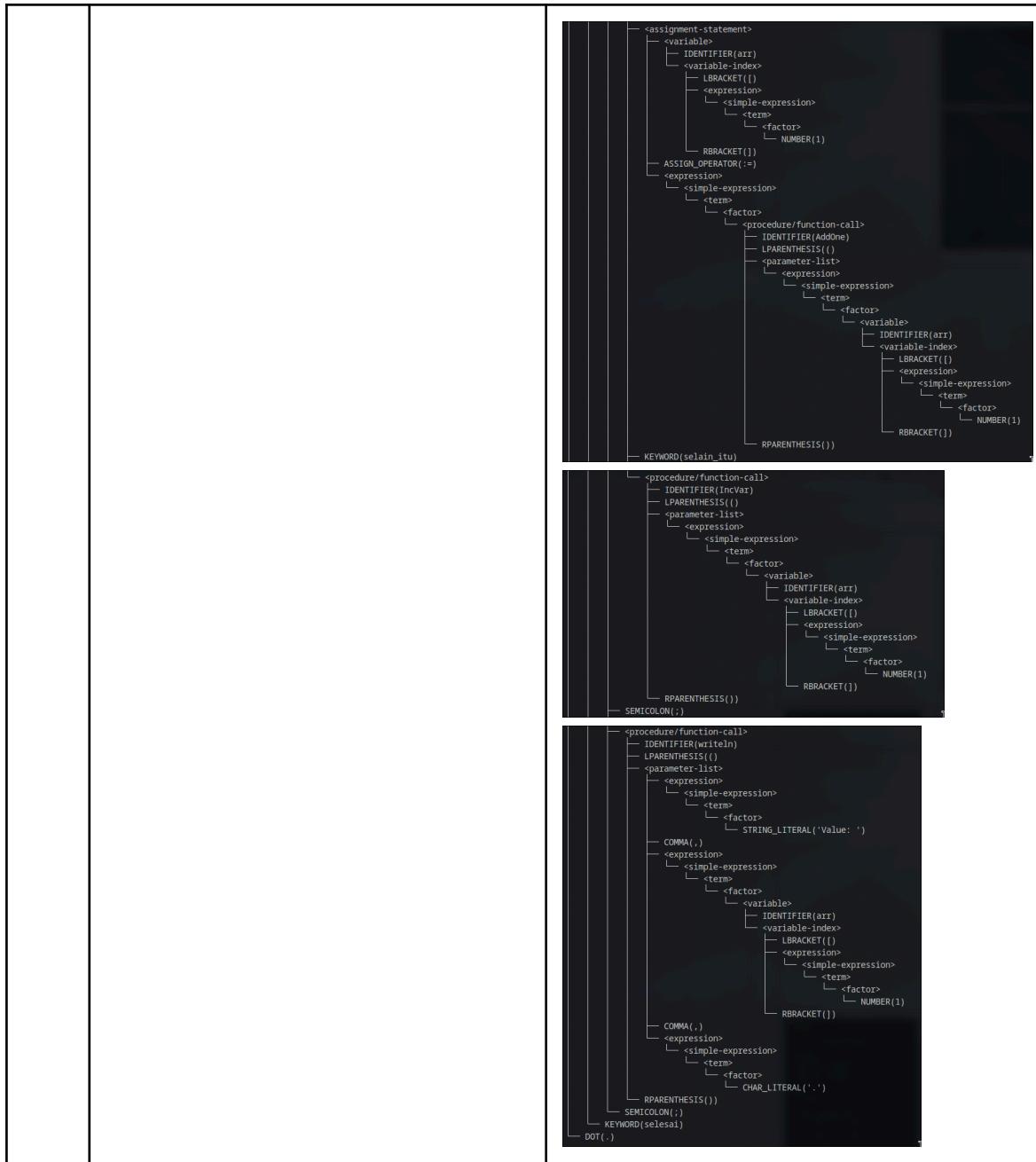
```

<compound-statement>
  └── KEYWORD(mulai)
  └── <statement-list>
    └── <assignment-statement>
      └── <variable>
        └── IDENTIFIER(x)
      └── ASSIGN_OPERATOR(:=)
      └── <expression>
        └── <simple-expression>
          └── <term>
            └── <factor>
              └── NUMBER(3.14)
    └── SEMICOLON(;)
    └── <assignment-statement>
      └── <variable>
        └── IDENTIFIER(ch)
      └── ASSIGN_OPERATOR(:=)
      └── <expression>
        └── <simple-expression>
          └── <term>
            └── <factor>
              └── CHAR_LITERAL('A')
    └── SEMICOLON(;)
    └── <assignment-statement>
      └── <variable>
        └── IDENTIFIER(ok)
      └── ASSIGN_OPERATOR(:=)
      └── <expression>
        └── <simple-expression>
          └── <term>
            └── <factor>
              └── KEYWORD(true)
            └── LOGICAL_OPERATOR(dan)
          └── <factor>
            └── LOGICAL_OPERATOR(tidak)
          └── <factor>
            └── KEYWORD(false)
    └── SEMICOLON(;)

<assignment-statement>
  └── <variable>
    └── IDENTIFIER(arr)
    └── <variable-index>
      └── LBRACKET({})
    └── <expression>
      └── <simple-expression>
        └── <term>
          └── <factor>
            └── NUMBER(1)
    └── RBRACKET({})
  └── ASSIGN_OPERATOR(:=)
  └── <expression>
    └── <simple-expression>
      └── <term>
        └── <factor>
          └── NUMBER(2)
  └── SEMICOLON(;)

<if-statement>
  └── KEYWORD(jika)
  └── <expression>
    └── <simple-expression>
      └── <term>
        └── <factor>
          └── LPARENTHESIS(())
    └── <expression>
      └── <simple-expression>
        └── <term>
          └── <factor>
            └── <variable>
              └── IDENTIFIER(arr)
              └── <variable-index>
                └── LBRACKET({})
                └── <expression>
                  └── <simple-expression>
                    └── <term>
                      └── <factor>
                        └── NUMBER(1)
                └── RBRACKET({})
            └── RELATIONAL_OPERATOR(>=)
            └── <simple-expression>
              └── <term>
                └── <factor>
                  └── <variable>
                    └── IDENTIFIER(k)
  └── RPARENTHESIS(())
  └── KEYWORD(maka)

```



5.2 Test Case Tambahan Milestone-2

Tabel 5.2 Tabel test case tambahan milestone-2 (berisi

(dalam test/milestone-2/input: input-6.pas hingga input-9.pas + input-kasus.pas dan input-rekaman.pas)

No.	Input	Output
1	input-kasus.pas <pre> ● ● ● program TestKasusDiagram; variabel pilihan: integer; hasil_A: integer; hasil_B: integer; mulai pilihan := 3; kasus pilihan dari 1: hasil_A := 100; 2: hasil_A := 200; 3: mulai hasil_A := 300; hasil_B := 333; selesai; 4: hasil_A := 400 selesai. </pre>	<pre> \$ python main.py milestone-2/input/input-kasus.pas SAVED => test/milestone-2/output/output.txt ===== PARSE TREE OUTPUT ===== <program> <program-header> └── KEYWORD(program) └── IDENTIFIER(TestKasusDiagram) └── SEMICOLON(;) <declaration-part> <var-declaration> └── KEYWORD(variabel) <identifier-list> └── IDENTIFIER(pilihan) └── COLON(:) <type> └── KEYWORD(integer) └── SEMICOLON(;) <identifier-list> └── IDENTIFIER(hasil_A) └── COLON(:) <type> └── KEYWORD(integer) └── SEMICOLON(;) <identifier-list> └── IDENTIFIER(hasil_B) └── COLON(:) <type> └── KEYWORD(integer) └── SEMICOLON(;) <compound-statement> └── KEYWORD(mulai) <statement-list> <assignment-statement> <variable> └── IDENTIFIER(pilihan) └── ASSIGN_OPERATOR(:=) <expression> <simple-expression> <term> <factor> └── NUMBER(3) └── SEMICOLON(;) <case-statement> └── KEYWORD(kasus) <expression> <simple-expression> <term> <factor> <variable> └── IDENTIFIER(pilihan) └── COLON(:) <case-list> <case> └── NUMBER(1) <colon> └── COLON(:) <assignment-statement> <variable> └── IDENTIFIER(hasil_A) └── ASSIGN_OPERATOR(:=) <expression> <simple-expression> <term> <factor> └── NUMBER(100) </pre>

```
    └── SEMICOLON();  
    └── NUMBER(2)  
    └── COLON(:)  
    └── <assignment-statement>  
        └── <variable>  
            └── IDENTIFIER(hasil_A)  
    └── ASSIGN_OPERATOR(:=)  
    └── <expression>  
        └── <simple-expression>  
            └── <term>  
                └── <factor>  
                    └── NUMBER(200)  
    └── SEMICOLON();  
    └── NUMBER(3)  
    └── COLON(:)  
    └── <compound-statement>  
        └── KEYWORD(mulai)  
        └── <statement-list>  
            └── <assignment-statement>  
                └── <variable>  
                    └── IDENTIFIER(hasil_A)  
            └── ASSIGN_OPERATOR(:=)  
            └── <expression>  
                └── <simple-expression>  
                    └── <term>  
                        └── <factor>  
                            └── NUMBER(300)  
    └── SEMICOLON();  
    └── <assignment-statement>  
        └── <variable>  
            └── IDENTIFIER(hasil_B)  
    └── ASSIGN_OPERATOR(:=)  
    └── <expression>  
        └── <simple-expression>  
            └── <term>  
                └── <factor>  
                    └── NUMBER(333)  
    └── SEMICOLON();  
    └── KEYWORD(selesai)
```

```
    └── SEMICOLON();  
    └── NUMBER(4)  
    └── COLON(:)  
    └── <assignment-statement>  
        └── <variable>  
            └── IDENTIFIER(hasil_A)  
    └── ASSIGN_OPERATOR(:=)  
    └── <expression>  
        └── <simple-expression>  
            └── <term>  
                └── <factor>  
                    └── NUMBER(400)  
    └── KEYWORD(selesai)  
    └── DOT(.)
```

2

input-rekaman.pas

```

program TestRekaman;

tipe
  DataMahasiswa = rekaman
    nama: char;
    nim, umur: integer;
    lulus: boolean
  selesai;

  Matakuliah = rekaman
    kode: char
  selesai;

variabel
  mhs1: DataMahasiswa;
  mk1: Matakuliah;

mulai

selesai.

```

```

[D] python main.py milestone-2/input/input-rekaman.pas
milestone-2
SAVED  =>      test/milestone-2/output/output.txt

===== PARSE TREE OUTPUT =====

<program>
  <program-header>
    KEYWORD(program)
    IDENTIFIER(TestRekaman)
    SEMICOLON(:)
  <declaration-part>
    <type-declaration>
      KEYWORD(tipe)
      IDENTIFIER(DataMahasiswa)
      RELATIONAL_OPERATOR(=)
      <type-definition>
        <record-type>
          KEYWORD(rekaman)
          <parameter-group>
            <identifier-list>
              IDENTIFIER(nama)
            COLON(:)
            <type>
              KEYWORD(char)
            SEMICOLON(:)
            <parameter-group>
              <identifier-list>
                IDENTIFIER(nim)
                COMMA(,)
                IDENTIFIER(umur)
              COLON(:)
              <type>
                KEYWORD(integer)
              SEMICOLON(;)
              <parameter-group>
                <identifier-list>
                  IDENTIFIER(lulus)
                COLON(:)
                <type>
                  KEYWORD(boolean)
                KEYWORD(selesai)
  <type-declaration>
    SEMICOLON(;)
    IDENTIFIER(Matakuliah)
    RELATIONAL_OPERATOR(=)
    <type-definition>
      <record-type>
        KEYWORD(rekaman)
        <parameter-group>
          <identifier-list>
            IDENTIFIER(kode)
          COLON(:)
          <type>
            KEYWORD(char)
          KEYWORD(selesai)
        SEMICOLON(;)
      <var-declaration>
        KEYWORD(variabel)
        <identifier-list>
          IDENTIFIER(mhs1)
        COLON(:)
        <type>
          IDENTIFIER(DataMahasiswa)
        SEMICOLON(;)
        <identifier-list>
          IDENTIFIER(mk1)
        COLON(:)
        <type>
          IDENTIFIER(Matakuliah)
        SEMICOLON(;)
      <compound-statement>
        KEYWORD(mulai)
        <statement-list>
          <empty-statement>
        KEYWORD(selesai)
      DOT(..)

```

3

input-6.pas

```

program mega;
tipe
  Index = 1 .. 10;
  Pair = rekaman
    a, b: integer
  selesai;
variabel
  x, y: integer;
  arr: larik[1 .. 5] dari integer;
  rec: rekaman
    p: Pair;
    v: larik[1 .. 3] dari integer
  selesai;
prosedur foo(a: integer; b: integer);
mulai
  a := b + 1;
selesai;
fungsi bar(t: integer): integer;
mulai
  bar := t * 2;
selesai;
mulai
  rec.p.a := arr[3] + bar(2);
  rec.v[2] := rec.p.a * 3;
  jika rec.v[2] > 10 maka
    x := 1
  selain_itu
    x := 2;
  selama x < 5 lakukan
    x := x + 1;
  untuk y := 1 ke 3 lakukan
    arr[y] := arr[y] + 1;
  ulangi
    x := x - 1
  sampai x = 0;
selesai.

```

```

$ python main.py milestone-2/input/input-6.pas
milestone-2
SAVED => test/milestone-2/output/output.txt

```

===== PARSE TREE OUTPUT =====

```

<program>
  <program-header>
    KEYWORD(program)
    IDENTIFIER(mega)
    SEMICOLON();
  <declaration-part>
    <type-declaration>
      KEYWORD(tipe)
      IDENTIFIER(Index)
      RELATIONAL_OPERATOR(=)
      <type-definition>
        <expression>
          <simple-expression>
            <term>
              <factor>
                NUMBER(1)
            RANGE_OPERATOR(..)
        <expression>
          <simple-expression>
            <term>
              <factor>
                NUMBER(10)
    SEMICOLON();

```

```

  IDENTIFIER(Pair)
  RELATIONAL_OPERATOR(=)
  <type-definition>
    <record-type>
      KEYWORD(rekaman)
      <parameter-group>
        <identifier-list>
          IDENTIFIER(a)
          COMMA()
          IDENTIFIER(b)
        COLON(:)
        <type>
          KEYWORD(integer)
        KEYWORD(selesai)
      SEMICOLON();
    <var-declaration>
      KEYWORD(variabel)
      <identifier-list>
        IDENTIFIER(x)
        COMMA()
        IDENTIFIER(y)
      COLON(:)
      <type>
        KEYWORD(integer)
      SEMICOLON();
      <identifier-list>
        IDENTIFIER(arr)
      COLON(:)

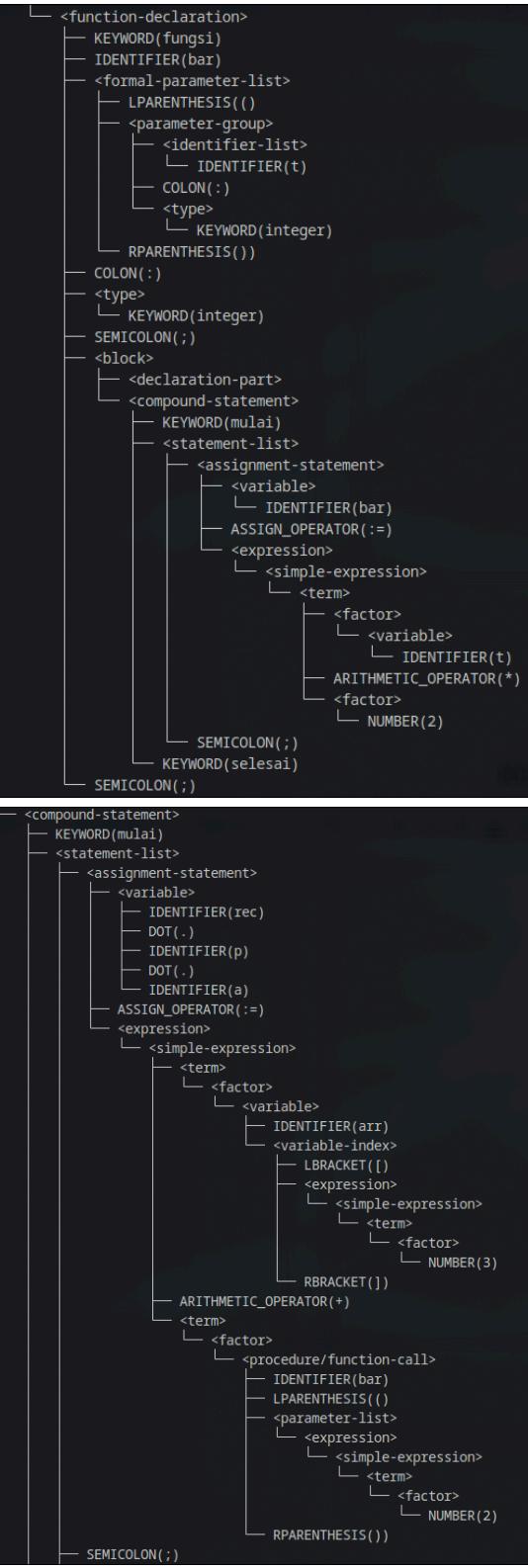
```

```

  <type>
    <array-type>
      KEYWORD(larik)
      LBRACKET([])
      <range>
        <expression>
          <simple-expression>
            <term>
              <factor>
                NUMBER(1)
            RANGE_OPERATOR(..)
        <expression>
          <simple-expression>
            <term>
              <factor>
                NUMBER(5)
            RBRACKET()
            KEYWORD(dari)
            <type>
              KEYWORD(integer)
        SEMICOLON();
        <identifier-list>
          IDENTIFIER(rec)
        COLON(:)

```









		<pre> <repeat-statement> KEYWORD(ulangi) <statement-list> <assignment-statement> <variable> IDENTIFIER(x) ASSIGN_OPERATOR(:=) <expression> <simple-expression> <term> <factor> <variable> IDENTIFIER(x) ARITHMETIC_OPERATOR(-) <term> <factor> NUMBER(1) <expression> <simple-expression> <term> <factor> <variable> IDENTIFIER(x) RELATIONAL_OPERATOR(=) <simple-expression> <term> <factor> NUMBER(0) SEMICOLON(;) KEYWORD(selesai) DOT(..) </pre>
4	input-7.pas	<pre> python main.py milestone-2/input/input-7.pas milestone-2 SAVED => test/milestone-2/output/output.txt ===== PARSE TREE OUTPUT ===== <program> <program-header> KEYWORD(program) IDENTIFIER(testall) SEMICOLON(;) <declaration-part> <type-declaration> KEYWORD(type) IDENTIFIER(Rec) RELATIONAL_OPERATOR(=) <type-definition> <record-type> KEYWORD(rekaman) <parameter-group> <identifier-list> IDENTIFIER(z) COLON(:) <type> KEYWORD(integer) SEMICOLON(;) </pre> <pre> <parameter-group> <identifier-list> IDENTIFIER(t) COLON(:) <type> <array-type> KEYWORD(larik) LBRACKET([) <range> <expression> <simple-expression> <term> <factor> NUMBER(1) RANGE_OPERATOR(..) <expression> <simple-expression> <term> <factor> NUMBER(2) RBRACKET()) KEYWORD(dari) <type> KEYWORD(integer) SEMICOLON(;) </pre>



```
└─ <procedure-declaration>
   └─ KEYWORD(prosedur)
   └─ IDENTIFIER(g)
   └─ <formal-parameter-list>
      └─ LPARENTHESIS(())
      └─ <parameter-group>
         └─ <identifier-list>
            └─ IDENTIFIER(q)
         └─ COLON(:)
         └─ <type>
            └─ KEYWORD(integer)
            └─ RPARENTHESIS(())
   └─ SEMICOLON(;)
   └─ <block>
      └─ <declaration-part>
         └─ <compound-statement>
            └─ KEYWORD(mulai)
            └─ <statement-list>
               └─ <assignment-statement>
                  └─ <variable>
                     └─ IDENTIFIER(b)
                  └─ <variable-index>
                     └─ LBRACKET(())
                     └─ <expression>
                        └─ <simple-expression>
                           └─ <term>
                              └─ <factor>
                                 └─ NUMBER(1)
                     └─ RBRACKET(())
               └─ DOT(.)
```

```
└─ IDENTIFIER(t)
   └─ <variable-index>
      └─ LBRACKET(())
      └─ <expression>
         └─ <simple-expression>
            └─ <term>
               └─ <factor>
                  └─ NUMBER(2)
   └─ RBRACKET(())
   └─ ASSIGN_OPERATOR(:=)
   └─ <expression>
      └─ <simple-expression>
         └─ <term>
            └─ <factor>
               └─ <variable>
                  └─ IDENTIFIER(q)
   └─ ARITHMETIC_OPERATOR(*)
   └─ <factor>
      └─ NUMBER(2)
   └─ SEMICOLON(;)
   └─ KEYWORD(selesai)
   └─ SEMICOLON(;)
```





5	input-8.pas (error test)  <pre>error program fail1; variabel x, y: integer mulai x := 3; selesai.</pre>	<pre>[D] python main.py milestone-2/input/input-8.pas milestone-2 SyntaxError: Unexpected token IDENTIFIER(error), expected KEYWORD --> (line 1, column 1) 1 error ^</pre>
6	input-9.pas (error test)  <pre>program fail_chain; variabel r: integer; mulai r.a. := 5; selesai.</pre>	<pre>[D] python main.py milestone-2/input/input-9.pas milestone-2 SyntaxError: Unexpected token ASSIGN_OPERATOR(:=), expected IDENTIFIER --> (line 5, column 8) 5 r.a. := 5; ^</pre>

BAB VI

KESIMPULAN & SARAN

6.1 Kesimpulan

Implementasi *syntax analyzer* untuk compiler Pascal-S telah berhasil diselesaikan pada Milestone 2 ini. Dengan menggunakan metode *Recursive Descent Parsing*, parser yang dibangun mampu memvalidasi urutan token-token yang dihasilkan oleh *lexer* dan menerjemahkannya menjadi sebuah *parse tree* yang terstruktur. Pendekatan ini terbukti efektif dalam menangani struktur hierarkis grammar Pascal-S, termasuk penanganan tantangan utama seperti prioritas operator, deklarasi tipe yang fleksibel (range, array, record), dan blok program bersarang.

Implementasi setiap aturan produksi sebagai fungsi rekursif yang independen (misalnya `parse_statement`, `parse_type_definition`, `parse_expression`) memungkinkan proses parsing *top-down* yang logis dan mudah ditelusuri. Sistem deteksi kesalahan menggunakan kelas `ParseError` juga berhasil diimplementasikan, yang mampu memberikan umpan balik informatif berupa pesan kesalahan, baris, dan kolom untuk mempermudah *debugging*. Parser ini telah berhasil memenuhi tujuan milestone dan menyediakan landasan struktur program yang kokoh untuk tahap analisis selanjutnya, yaitu analisis semantik.

6.2 Saran

Mekanisme *error handling* dapat lebih ditingkatkan. Saat ini, parser berhenti total pada *syntax error* pertama yang ditemukan (karena *exception*). Strategi *error recovery* (seperti *panic mode* atau sinkronisasi token) dapat dipertimbangkan untuk diimplementasikan. Hal ini akan memungkinkan parser untuk mencoba melanjutkan proses analisis setelah menemukan kesalahan, sehingga dapat melaporkan beberapa kesalahan sintaks sekaligus dalam satu kali kompilasi.

REFERENSI

- [1] Edunex ITB. (n.d.). *Compiler Analisis Leksikal dan Syntax*. ITB Lecture Module. Accessed: November 12, 2025. [Online]. Available: <http://edunex.itb.ac.id>
- [2] N. Wirth, PASCAL-S: A Subset and its implementation. Accessed: November 12, 2025. [Online]. Available: <http://pascal.hansotten.com/uploads/pascals/PASCAL-S%20A%20subset%20and%20its%20Implementation%20012.pdf>
- [3] Tutorials Point. (n.d.). Compiler Design - Syntax Analysis. Accessed: November 13, 2025. [Online]. Available: https://www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.htm
- [4] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Boston, MA: Addison-Wesley, 2006.

LAMPIRAN

Link Repository : [Link Repository \[GitHub\]](#)

Link Diagram : [Link Diagram \[Draw.io\]](#)

Pembagian Tugas:

Nama	NIM	Tugas	Kontribusi
Brian Ricardo Tamin	13523126	Refine modularity and lexical error, parse expression, term, factor, operators	25%
Jovandra Otniel P. S.	13523141	Parse identifier , array, range, statement, statement_list	25%
Andrew Tedjapratama	13523148	Parse Node, init Recursive Descent, Parse Variable, Parse type & type-definition, parse statement detail	25%
Theo Kurniady	13523154	Parse declarations, parameter, variable-index, rekaman, & kasus	25%