

Milestone 2

IF2224 Teori Bahasa Formal dan Otomata

Syntax Analysis



Kelompok MOE

Disusun oleh:

Ivant Samuel Silaban 13523129
Rafa Abdussalam Danadyaksa 13523133
Muhamad Nazih Najmudin 13523144
Anas Ghazi Al Gifari 13523159
Muhammad Rizain Firdaus 13523164

PROGRAM STUDI TEKNIK INFORMATIKA
SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2025

DAFTAR ISI

DAFTAR ISI.....	1
DAFTAR GAMBAR.....	3
BAB I	
LANDASAN TEORI.....	4
1.1 Syntax Analysis.....	4
1.2 Context-Free Grammar.....	5
1.3 Parse Tree.....	7
1.4 Recursive Descent Parsing.....	11
1.5 LL(1) Parsing.....	13
1.6 Operator Precedence dan Associativity.....	15
BAB II	
PERANCANGAN DAN IMPLEMENTASI.....	18
2.1 Arsitektur Sistem.....	18
2.2 Aturan Produksi.....	19
2.2.1 Program dan Blok.....	19
2.2.2 Deklarasi.....	19
2.2.3 Tipe.....	21
2.2.4 Konstanta.....	22
2.2.5 Subprogram.....	23
2.2.6 Statement.....	26
2.2.7 Parameter Aktual.....	29
2.2.8 Ekspresi.....	30
2.2.9 Faktor dan Variabel.....	31
2.2.10 Operator.....	32
2.3 Implementasi Algoritma.....	33
2.3.1 Struktur File.....	33
2.3.2 Kelas dan Fungsi.....	34
BAB III	
PENGUJIAN.....	46
3.1 Rencana Pengujian.....	46
3.2 Test Case.....	46
3.2.1 Test Case 1: Hyphenated.....	46
3.2.2 Test Case 2: Minus Edge Cases.....	49
3.2.3 Test Case 3: Error Checking.....	53
3.2.4 Test Case 4: Test Declaration.....	54
3.2.5 Test Case 5: Test Subprogram.....	62
3.3 Ringkasan Hasil Pengujian.....	68

BAB IV**KESIMPULAN DAN SARAN.....70**

4.1 Kesimpulan.....70

4.2 Saran.....70

LAMPIRAN.....72**REFERENSI.....77**

DAFTAR GAMBAR

Gambar 1.1. Parse tree.....	4
Gambar 1.2. Salah satu representasi himpunan aturan produksi dalam sebuah tree.....	7
Gambar 1.3. Ekuivalensi dari beberapa pernyataan tentang grammar.....	9
Gambar 1.4. Parse tree yang merepresentasikan derivation dari suatu ekspresi dalam grammar kalkulator sederhana.....	10
Gambar 1.5. Contoh Pembuatan Parse Tree dan Perbedaannya dengan AST (Abstract Syntax Tree).....	11
Gambar 1.6. Parse Tree yang merepresentasikan ekspresi $a^*(a+b)^0$	17
Gambar 2.1. Arsitektur sistem.....	18

BAB I

LANDASAN TEORI

1.1 Syntax Analysis

Pada Milestone 1 Lexical Analysis, *source code* sudah dibentuk menjadi token-token tertentu. Token-token inilah yang kemudian menjadi input bagi tahap selanjutnya, yaitu Syntax Analysis atau yang sering disebut parser.

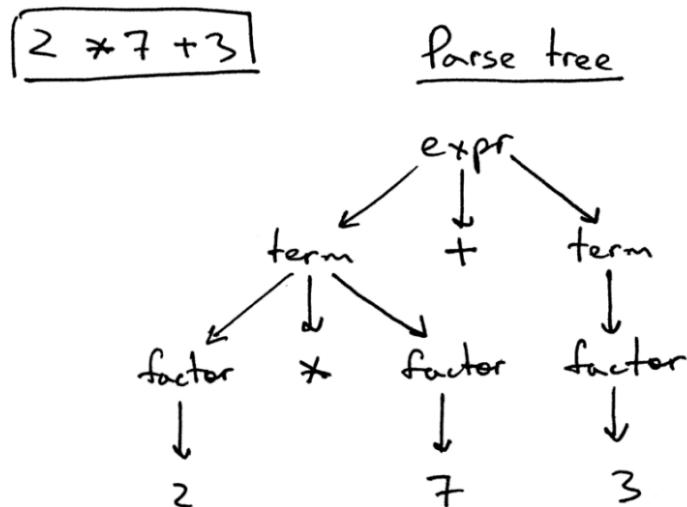
Analisis sintaksis (Syntax Analysis) atau parser bertugas untuk memeriksa urutan token yang dihasilkan oleh *lexical analyzer* untuk memastikan bahwa token-token tersebut sesuai dengan aturan tata bahasa (*grammar*) dari bahasa pemrograman yang dikompilasi.

Parser kemudian membangun struktur sintaksis seperti Parse Tree yang merepresentasikan struktur hierarkis antar elemen program sumber berdasarkan *grammar* yang telah didefinisikan.

Tahapan parser bertujuan untuk:

- Memastikan bahwa urutan token membentuk struktur sintaks yang valid.
- Mendeteksi dan melaporkan kesalahan sintaksis (*syntax error*).
- Mempersiapkan representasi struktural, seperti Parse Tree, yang menggambarkan struktur hierarkis program dan digunakan pada tahap berikutnya.

Untuk menggambarkan proses tersebut, Gambar 1.1 menunjukkan contoh Parse Tree yang terbentuk dari ekspresi aritmetika $2 * 7 + 3$. Parse Tree ini merepresentasikan bagaimana ekspresi tersebut diuraikan berdasarkan aturan *grammar* yang mendefinisikan hubungan antara *expression*, *term*, dan *factor*.



Gambar 1.1. Parse tree
(Sumber: <https://ruslanspivak.com/lsbasi-part7>)

1.2 Context-Free Grammar

Dalam teori bahasa formal, Context-Free Grammar (CFG) adalah suatu notasi formal yang digunakan untuk mengekspresikan aturan tata bahasa atau sintaksis yang berulang. CFG memungkinkan penyusunan struktur bahasa yang kompleks dapat dijelaskan melalui himpunan aturan yang lebih sederhana.

Secara formal, bentuk CFG dapat direpresentasikan sebagai

$$G = \{V, T, P, S\}$$

dengan

- V = Himpunan simbol nonterminal
- T = Himpunan simbol terminal
- P = Himpunan aturan produksi
- S = Simbol *start*

Simbol nonterminal atau yang disebut variabel adalah simbol yang merepresentasikan kategori sintaksis tertentu dan masih dapat diturunkan menjadi rangkaian simbol lain menggunakan aturan produksi. Sebaliknya, simbol terminal adalah simbol yang tidak dapat diturunkan lagi dan merupakan elemen akhir dalam *string* bahasa yang dihasilkan. Aturan produksi dalam CFG memiliki bentuk umum

$$A \rightarrow \alpha$$

dengan A adalah satu simbol nonterminal, sedangkan α merupakan deretan terminal, nonterminal, atau kombinasi keduanya. Produksi ini digunakan secara rekursif untuk membentuk *string* dalam bahasa yang didefinisikan. Proses penurunan selalu dimulai dari simbol *start* S yang menjadi titik awal pembentukan seluruh struktur bahasa.

CFG menentukan *string* apa saja yang valid atau tidak valid dalam suatu bahasa berdasarkan himpunan aturan produksinya. Aturan-aturan tersebut memberikan bagaimana simbol tersebut dapat dibentuk dari simbol-simbol lainnya. Sebagai contoh, diberikan aturan produksi berikut.

$$E \rightarrow I$$

Menyatakan bahwa simbol nonterminal E dapat diturunkan menjadi simbol I . Dalam hal ini, E merupakan variabel pada sisi kiri produksi, sedangkan I merupakan deretan simbol pada sisi kanan produksi. Dengan demikian, aturan produksi memiliki tujuan untuk mendefinisikan bagaimana simbol nonterminal dapat diperluas menjadi struktur sintaksis lain hingga membentuk rangkaian terminal.

Akan dieksplorasi CFG yang merepresentasikan ekspresi aritmatika sederhana dalam bahasa pemrograman menggunakan operator + untuk penjumlahan dan * untuk perkalian.

Simbol terminal yang diizinkan di sini hanyalah huruf a dan b , serta angka 0 dan 1. Setiap *identifier* harus dimulai dengan a atau b , diikuti oleh *string* apapun dalam $\{a, b, 0, 1\}^*$.

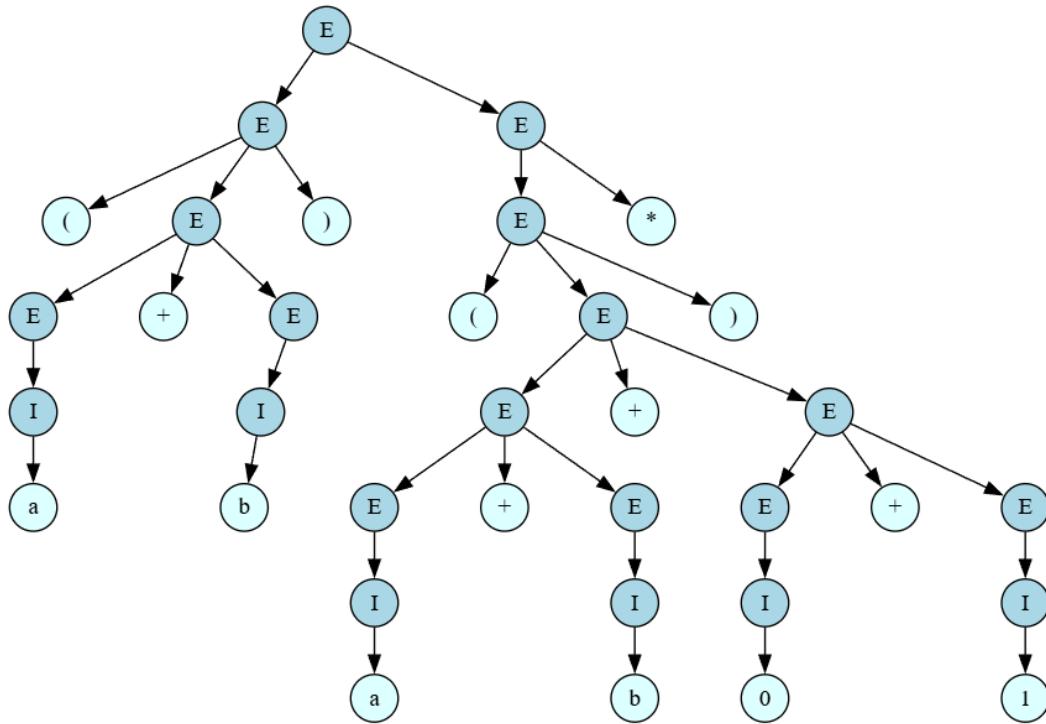
Simbol *start* pada CFG mulai menjelaskan bahasa yang diwakili melalui kumpulan aturan produksi tersebut. Misalkan E adalah simbol *start* yang merepresentasikan ekspresi dari bahasa yang didefinisikan dan I adalah variabel yang merepresentasikan *identifier*. Bahasa yang didefinisikan oleh *grammar* ini setara dengan *regular expression* berikut.

$$(a + b)(a + b + 0 + 1)^*$$

Meskipun demikian, *regular expressions* tidak akan digunakan secara langsung dalam *grammar*. Sebagai gantinya, akan digunakan himpunan aturan produksi yang menyatakan bahasa yang sama dengan *regular expression* tersebut.

1. $E \rightarrow I$
2. $E \rightarrow E + E$
3. $E \rightarrow E * E$
4. $E \rightarrow (E)$
5. $I \rightarrow a$
6. $I \rightarrow b$
7. $I \rightarrow Ia$
8. $I \rightarrow Ib$
9. $I \rightarrow I0$
10. $I \rightarrow I1$

Grammar dari ekspresi-ekspresi ini dapat dinyatakan secara formal dengan $G = (\{E, I\}, T, P, E)$ dengan T adalah himpunan simbol terminal $\{+, *, (,), a, b, 0, 1\}$ dan P adalah himpunan aturan produksi yang ditunjukkan di atas. Himpunan aturan produksi ini dapat direpresentasikan dalam sebuah *tree* seperti pada Gambar 1.2 yang menunjukkan bagaimana sebuah *string* dari bahasa tersebut diturunkan. *Grammar* ini mendefinisikan bahasa yang sama dengan *regular expression* sebelumnya.



Gambar 1.2. Salah satu representasi himpunan aturan produksi dalam sebuah tree
(Sumber: dokumen kelompok)

CFG disebut sebagai *Context-Free* karena aturan produksi yang dapat digunakan dapat dibuat tanpa memerlukan informasi tentang konteksnya untuk dapat diterapkan. Artinya, aturan produksi CFG bisa diterapkan pada setiap kemunculan simbol variabel dalam sebuah kalimat tanpa memerlukan posisi simbol dalam kalimat tersebut.

1.3 Parse Tree

Parse Tree adalah representasi lengkap dari struktur sintaks program berdasarkan aturan Context-Free Grammar (CFG). Hubungan *parent-child* dalam *tree* mewujudkan setiap langkah dalam derivasi *grammar*. Parse Tree menunjukkan struktur sintaksis yang lengkap dari *string*, memperjelas bagaimana sebuah input dihasilkan oleh *grammar* dan sekaligus memastikan input tersebut valid.

Untuk setiap Parse Tree dari *grammar* $G = (V, T, P, S)$, berlaku kondisi-kondisi berikut.

1. Setiap *node* internal dilabeli oleh sebuah variabel (simbol nonterminal) yang merupakan anggota dari V .
2. Setiap *leaf* dilabeli oleh variabel, simbol terminal, atau ϵ . Namun, jika *leaf* dilabeli ϵ , *leaf* tersebut harus menjadi satu-satunya *child* dari *parent*-nya.
3. Jika sebuah *node* internal dilabel A , dan *children*-nya dilabel

$$X_1, X_2, \dots, X_k$$

secara berurutan dari kiri, berlaku bahwa $A \rightarrow X_1 X_2 \dots X_k$ adalah sebuah aturan produksi dalam P . Perhatikan bahwa satu-satunya momen ketika salah satu dari X 's dapat berupa ϵ adalah jika X tersebut adalah label dari satu-satunya *child* dari *parent* tersebut. Ini sesuai dengan produksi $A \rightarrow \epsilon$.

Dalam konstruksi *tree*, urutan *children* dari sebuah *node* selalu ditetapkan dari kiri ke kanan yang secara langsung mengikuti urutan simbol pada sisi kanan (Right-Hand Side atau RHS) dari aturan produksi yang digunakan. Konkatenasi dari kiri ke kanan pada sebuah Parse Tree dari *grammar* yang digunakan tersebut memberikan sebuah *string* yang diturunkan dari variabel *root*. Hal ini mengarah pada dua kondisi berikut yang penting bagi Parse Tree yang menghasilkan bahasa dari *grammar* terkait.

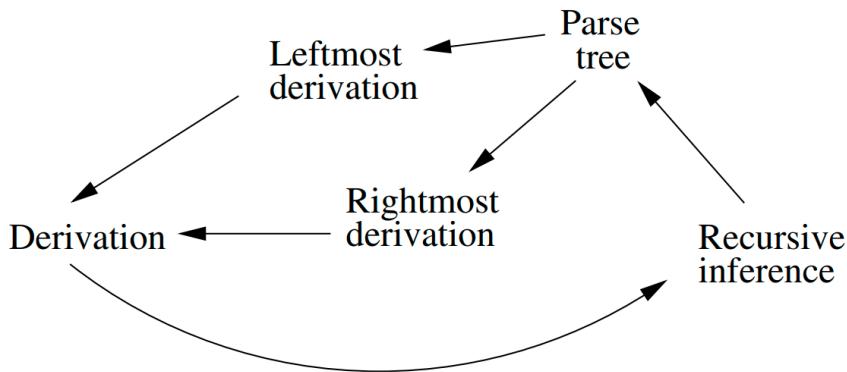
1. Hasil dari Parse Tree tersebut adalah *string* terminal. Artinya, semua *leaf* dilabeli dengan simbol terminal atau ϵ .
2. *root* dari Parse Tree tersebut dilabeli dengan simbol *start S* dari *grammar* tersebut.

Parse Tree yang hasilnya adalah *string* terminal dan *root*-nya adalah simbol *start* merepresentasikan *string* yang termasuk dalam bahasa dari *grammar* yang mendasarinya. Dengan demikian, cara lain untuk mendefinisikan bahasa dari suatu *grammar* adalah sebagai himpunan hasil dari semua Parse Tree yang memiliki simbol *start* di akarnya.

Diberikan *grammar* $G = (V, T, P, S)$, dapat ditunjukkan bahwa pernyataan-pernyataan berikut adalah ekuivalen jika w adalah *string* terminal.

1. Prosedur *recursive inference* menentukan bahwa *string* terminal w berada dalam bahasa dari variabel A .
2. Terdapat *derivation* dari A ke w ($A \xrightarrow{*} w$).
3. Terdapat *leftmost derivation* dari A ke w ($A \xrightarrow{lm} w$)
4. Terdapat *rightmost derivation* dari A ke w ($A \xrightarrow{rm} w$)
5. Terdapat Parse Tree dengan *root A* dari hasil w

Ekuivalensi ini berlaku tidak hanya untuk string terminal w , tetapi juga untuk *string* yang mengandung variabel, kecuali untuk penggunaan *recursive inference* yang hanya didefinisikan untuk *string* terminal. Ekuivalensi antar pernyataan-pernyataan ini dapat digambarkan melalui sebuah diagram seperti ada Gambar 1.3.



Gambar 1.3. Ekuivalensi dari beberapa pernyataan tentang grammar

(Sumber: Hopcroft, Motwani, dan Ullman, "Introduction to Automata Theory, Languages, and Computation", 3rd Edition, 2006)

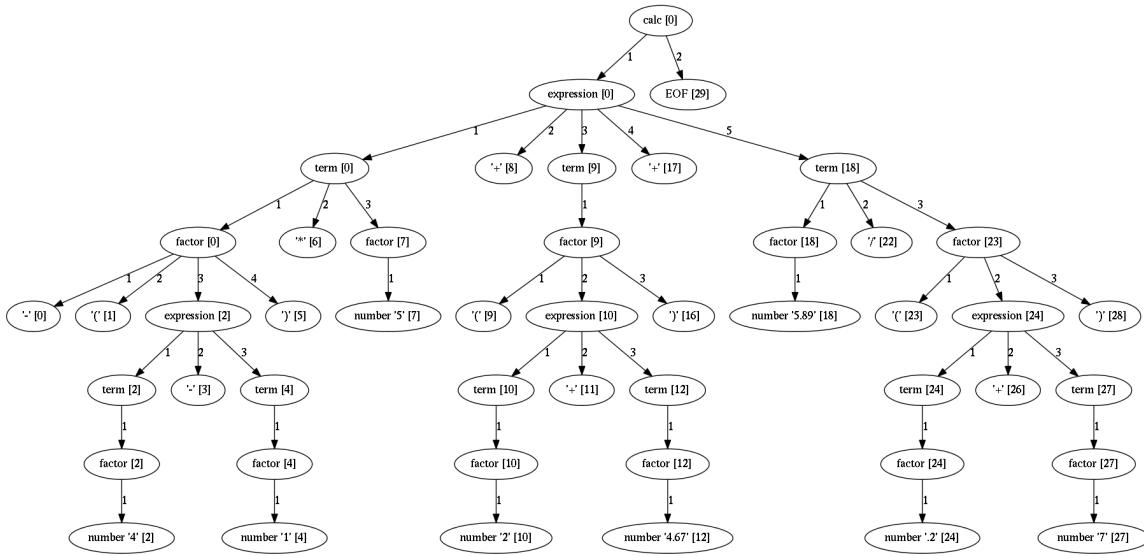
Diagram tersebut mengindikasikan bahwa jika w memenuhi kondisi pada pangkal panah, w juga memenuhi kondisi pada ujung panah. Sebagai contoh, jika w memiliki *leftmost derivation* dari A , sudah pasti memiliki *derivation* biasa dari A karena *leftmost derivation* adalah sebuah penurunan. Bukti yang lebih sulit dari ekuivalensi ini melibatkan teorema-teorema yang menunjukkan hubungan bolak-balik antara Parse Tree dan prosedur *inference* atau *derivation*.

- Teorema 5.12: Jika $G = (V, T, P, S)$ adalah CFG dan sebuah *string* terminal w dapat dibuktikan berada dalam bahasa dari variabel A melalui *recursive inference*, terdapat Parse Tree untuk G dengan *root* A dan hasil w .
- Teorema 5.14: Jika $G = (V, T, P, S)$ adalah CFG dan terdapat Parse Tree dengan *root* A dan hasil w , terdapat *leftmost derivation* dari A ke w ($A \xrightarrow{lm} w$) dalam G .

Dengan adanya ekuivalensi ini, Parse Tree dapat digunakan sebagai representasi struktur sintaks yang lengkap karena keberadaannya menjamin bahwa *string* tersebut dapat diturunkan menggunakan aturan *grammar*.

Dalam pengembangan *compiler*, seperti pada *compiler* Pascal-S yang menggunakan CFG, Parse Tree yang dihasilkan oleh *parser* menyimpan seluruh simbol terminal (token) dan simbol non-terminal sesuai dengan *grammar* yang telah ditentukan. Sehingga parse tree dapat digunakan sebagai representasi struktur sintaks yang lengkap. Secara umum, Parse Tree digunakan dalam proses *compiling* untuk hal-hal berikut.

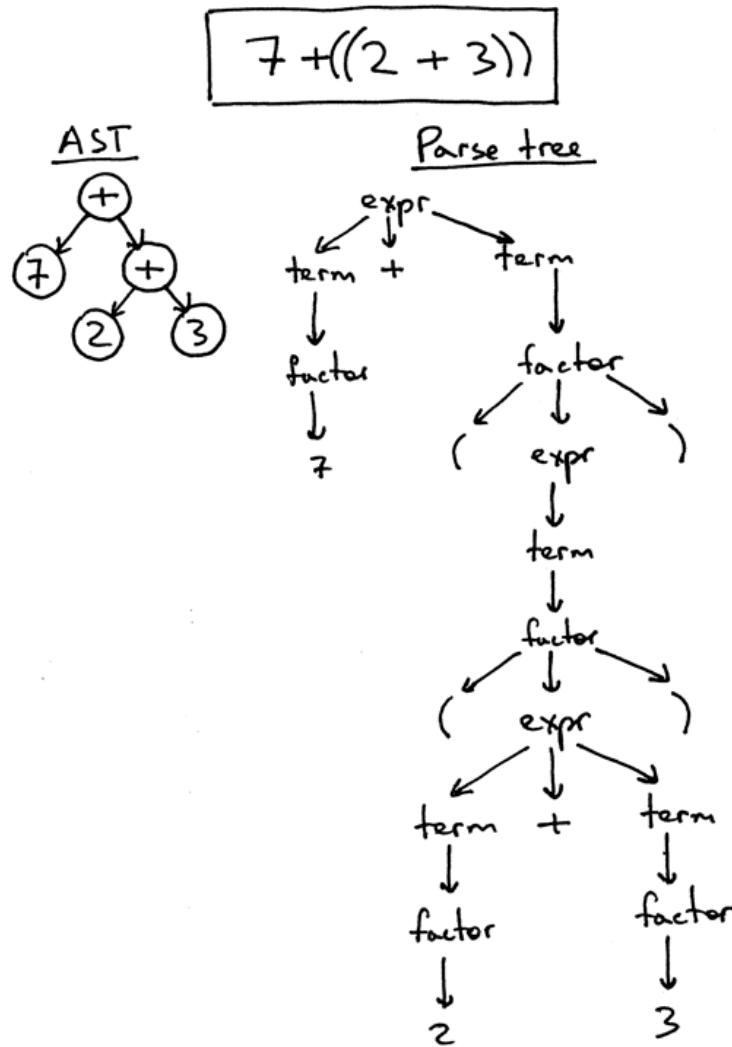
1. Memverifikasi bahwa urutan token dari program sumber benar-benar mengikuti aturan *grammar* bahasa tersebut.
2. Menyediakan struktur hierarkis bagi tahap selanjutnya (semantic analysis, AST generation).
3. Mendeteksi kesalahan sintaks secara terstruktur dan akurat karena setiap penyimpangan dari aturan *grammar* akan mencegah konstruksi pohon yang valid.



Gambar 1.4. Parse tree yang merepresentasikan *derivation* dari suatu ekspresi dalam grammar kalkulator sederhana

(Sumber: https://textx.github.io/Arpeggio/latest/images/calc_parse_tree.dot.png)

Parse Tree yang dihasilkan oleh *parser* untuk grammar kalkulator seperti pada Gambar 1.4 menunjukkan *derivation* yang dimulai dari *node* calc[0] hingga menjadi *string* terminal di bagian *leaf*. *Tree* ini dimulai dari *root* calc[0] dan langsung diturunkan menjadi expression[0] dan EOF (End of File) yang menunjukkan kerangka dasar program. Struktur hierarkis kemudian menyebar melalui *node-node* internal seperti expression, term, dan factor yang merupakan simbol nonterminal yang digunakan untuk mengelompokkan operasi. Simbol terminal atau token seperti +, *, dan number 'x' hanya muncul sebagai *leaf*. *Tree* ini menunjukkan bahwa seluruh *string* input tersebut, jika dibaca dari *leaf*-nya, secara sintaksis valid dan memberikan urutan operasi yang jelas untuk evaluasi ekspresi aritmatika.



Gambar 1.5. Contoh Pembuatan Parse Tree dan Perbedaannya dengan AST (Abstract Syntax Tree)

(Sumber: <https://ruslanspivak.com/lbasi-part7>)

1.4 Recursive Descent Parsing

Recursive Descent Parsing adalah metode *top-down parsing* yang membangun parse tree dengan memanggil satu fungsi untuk setiap non terminal dalam grammar. Parser berjalan dari simbol awal (start symbol) dan menurunkan input token secara rekursif mengikuti aturan produksi grammar. Pendekatan ini termasuk dalam keluarga LL parsing, karena membaca input secara Left-to-right, dan membentuk Leftmost derivation.

Menurut teori CFG dan parse tree sebelumnya, recursive descent secara langsung membangun parse tree yang ekivalen dengan derivasi kiri, dimana setiap fungsi non-terminal membentuk satu node, dan memanggil fungsi lain untuk membentuk anak-anaknya. Karakteristik utamanya adalah sebagai berikut.

1. Satu fungsi per non-terminal

Jika grammar memiliki non-terminal `<expression>`, maka parser menyediakan: `parse_expression()`. Setiap fungsi memeriksa token lookahead dan memilih produksi yang sesuai.

2. Top-Down: mulai dari Start Symbol

Parsing dimulai dari nonterminal awal grammar, misalnya `<program>`. Fungsi ini memanggil sub-fungsi aturan grammar

3. Leftmost derivation

Implementasi recursive descent selalu menghasilkan derivasi kiri, berdasarkan teorema hubungan derivasi dan parse tree.

4. Lookahead tunggal (umumnya 1 token)

Recursive descent bergantung pada kemampuan memilih aturan produksi berdasarkan token berikutnya. Grammar harus berbentuk LL(1) (tidak ambigu, tidak left-recursive, FIRST/FOLLOW tidak bertabrakan).

5. Error Handling terstruktur

Jika lookahead tidak cocok dengan produksi mana pun maka parser mengeluarkan syntax error (“unexpected token ... expected ...”)

Kelebihan dari Recursive Descent Parsing adalah

- Sederhana diimplementasikan, dimana struktur program mengikuti langsung struktur grammar
- Mudah dibaca dan di-maintenance (setiap non-terminal = satu fungsi)
- Sangat cocok untuk bahasa dengan grammar LL(1), seperti Pascal-S

Kekurangan dari Recursive Descent Parsing adalah

- Tidak dapat menangani grammar left-recursive secara langsung
- Memerlukan grammar yang “factorable” agar dapat dipilih berdasarkan satu token lookahead.
- Bisa memerlukan penyesuaian grammar manual agar bebas ambiguitas

Contoh alur kerja yaitu dengan aturan grammar:

```
program → program-header declaration-part compound-statement DOT
```

Maka parser memiliki fungsi seperti ini

```
parse_program() :  
    parse_program_header()
```

```

    parse_declaraction_part()
    parse_compound_statement()
    match(DOT)

```

Setiap pemanggilan fungsi akan menghasilkan node pada parse tree, konsisten dengan definisi parse tree, dimana setiap node sesuai produksi grammar.

1.5 LL(1) Parsing

LL(1) parsing adalah teknik top-down parsing yang membaca input Left-to-right, membentuk derivasi Leftmost, dan hanya membutuhkan 1 token look ahead untuk menentukan produksi mana yang harus dipakai. Parser jenis ini menjadi dasar dari recursive descent, dan digunakan ketika grammar berada dalam bentuk yang kompatibel dengan LL(1). Materi teoretis LL(1) mengandalkan konsep FIRST, FOLLOW, dan eliminasi ambiguitas/left-recursion. LL(1) ini penting karena ini akan menentukan apakah grammar bisa diparse secara deterministik oleh recursive descent.

Sebuah grammar dikatakan LL(1) jika untuk setiap nonterminal A dengan beberapa alternatif produksi

$$A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$$

harus memenuhi dua kondisi berikut.

1. $\text{FIRST}(\alpha_i)$ dan $\text{FIRST}(\alpha_j)$ harus saling terpisah (disjoint) untuk setiap $i \neq j$. Parser harus memilih produksi hanya berdasarkan 1 lookahead token.
2. Jika suatu produksi dapat menghasilkan ϵ , berlaku

$$\text{FIRST}(\alpha i) \cap \text{FOLLOW}(A) = \emptyset$$

supaya parser tidak ambigu saat memutuskan kapan memakai produksi.

Jika syarat ini terpenuhi, tidak ada konflik pilihan, dan grammar deterministik untuk parsing LL(1).

Karakteristik LL(1) Parsing adalah:

1. Deterministik: Tidak ada backtracking. Produksi dipilih berdasarkan token berikut (lookahead).
2. Prediktif: Parser dapat “memprediksi” produksi mana yang benar menggunakan FIRST/FOLLOW.
3. Tidak boleh left-recursive: Produksi seperti $A \rightarrow A\alpha$ harus dihilangkan atau ditulis ulang.
4. Tidak boleh ambigu: Grammar ambigu tidak dapat dipecahkan oleh LL(1)

Hubungan LL(1) dengan Recursive Descent adalah Recursive descent adalah implementasi langsung dari LL(1). Setiap fungsi non terminal melakukan

- cek FIRST set produksi
- pilih salah satu produksi
- consume token
- panggil fungsi-fungsi lain sesuai RHS produksi

Karena recursive descent hanya memakai 1 token lookahead, grammar yang digunakan harus LL(1). Contoh nyata di Pascal-S:

```

statement →
    assignment-statement
    | if-statement
    | while-statement
    | for-statement
    | procedure/function-call

```

First set:

- assignment-statement → IDENTIFIER
- function-call → IDENTIFIER (diikuti LPAREN)
- if-statement → KEYWORD(jika)
- while-statement → KEYWORD(selama)
- for-statement → KEYWORD(untuk)

Karena dua produksi sama-sama dimulai dengan IDENTIFIER, grammar memakai disambiguasi berbasis lookahead ke-2:

```

IDENTIFIER '(' → function call
IDENTIFIER not '(' → assignment

```

Ini teknik standar di grammar LL(1) yang mengizinkan kondisi lookahead selektif ketika diperlukan

Keuntungan Grammar LL(1) ini adalah:

- Parsing sederhana, cepat, tanpa backtracking
- Struktur parser langsung mengikuti bentuk grammar
- Error handling mudah karena konflik parsing jarang terjadi
- Cocok untuk bahasa terstruktur seperti Pascal-S.

1.6 Operator Precedence dan Associativity

Pada analisis sintaks bahasa pemrograman, ekspresi aritmatika atau logika tidak dapat diparse secara sembarang. Parser harus mengetahui operator mana yang harus dievaluasi lebih dahulu dan apakah operator tersebut bersifat left-associative (mengelompokkan dari kiri) atau right-associative (mengelompokkan dari kanan). Dua konsep ini dinamakan Operator Precedence dan Operator Associativity.

Operator precedence menentukan tingkat prioritas masing-masing operator dalam sebuah ekspresi. Operator dengan prioritas lebih tinggi harus dikelompokkan lebih dahulu dalam parse tree. Contoh umum urutan prioritas (tinggi ke rendah) adalah

1. faktor: literal, identifier, function-call, parenthesis
2. operator unary: -, +, tidak
3. operator perkalian: *, /, mod, bagi, dan
4. operator penjumlahan: +, -, atau
5. operator relasional: =, <, <=, >=, \diamond

Grammar Pascal-S sudah secara eksplisit membentuk hirarki precedence ini melalui pembagian non-terminal:

```

expression → simple-expression (rel-op simple-expression) ?
simple-expression → ('+' | '-')? term (additive-op term)*
term → factor (multiplicative-op factor)*
factor → ... (literal, identifier, parenthesis, unary, call)

```

Struktur grammar tersebut secara otomatis memaksa parser memproses:

- actor lebih dulu daripada term,
- term lebih dulu daripada simple-expression,
- simple-expression lebih dulu daripada expression.

Dengan demikian, precedence diatur oleh bentuk grammar, bukan oleh tabel terpisah.

Operator Associativity menentukan arah pengelompokan jika dua operator dengan precedence yang sama muncul berurutan. Contoh klasik:

a - b - c

Jika — bersifat left-associative, maka:

(a - b) - c

Jika right-associative:

$$a - (b - c)$$

Pada grammar Pascal-S, strukur:

```
term (additive-operator term) *
```

dan

```
factor (multiplicative-operator factor) *
```

Membuat operator-operator tersebut left-associative, karena parser akan membangun Parse Tree melalui pola berikut.

1. parse left operand
2. selama ada operator berikutnya, gabungkan menjadi node kiri
3. ulangi

Pola ini selalu membentuk ekspresi kiri seperti:

```
Node ('-', Node ('-', a, b), c)
```

yang merupakan bentuk left-associative.

Unary, seperti $-x$, $+x$, $\text{not } x$, selalu diparse terlebih dahulu sebelum binary operator, karena grammar menempatkannya pada level simple-expression (untuk + dan -) dan factor (untuk NOT). Ini mencegah ambiguitas seperti:

$$- a * b$$

yang benar ditafsirkan sebagai:

$$(-a) * b$$

Dalam recursive descent:

- precedence diwujudkan dengan memecah grammar menjadi beberapa fungsi (`parse_expression()`, `parse_simple_expression()`, `parse_term()`, `parse_factor()`)
- associativity diwujudkan dengan loop pada bagian (`op operand`)*

Contoh pola umum:

```
left = parse_term()
while lookahead in ADDITIVE_OPERATOR:
```

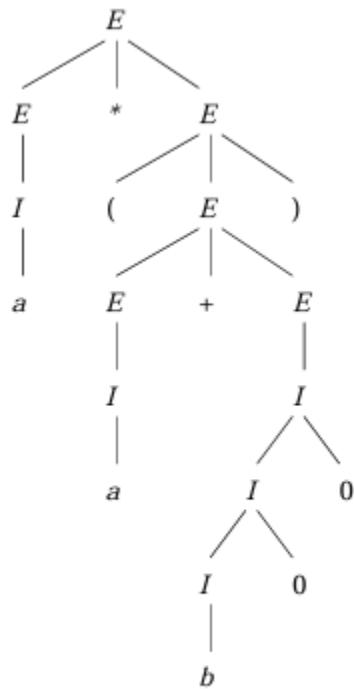
```

op = consume()
right = parse_term()
left = Node(op, left, right)
return left

```

Pola ini menghasilkan associative kiri secara alami. Parse tree menampilkan precedence dan associativity secara eksplisit dalam bentuk struktur pohon:

- operator dengan precedence lebih tinggi muncul lebih dekat ke leaf (lebih dalam)
- operator left-associative menghasilkan struktur bercabang ke kiri



The yield is $a * (a + b00)$.

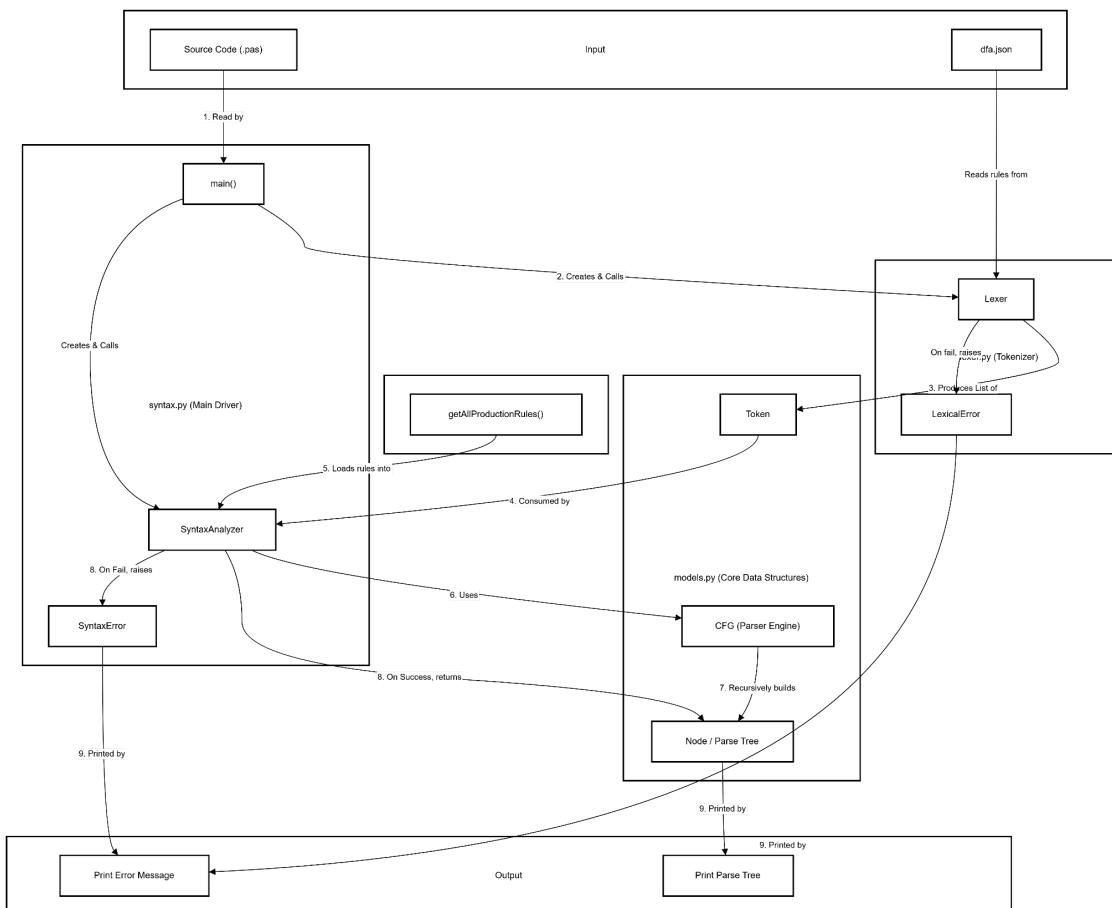
Gambar 1.6. Parse Tree yang merepresentasikan ekspresi $a * (a + b00)$
(Sumber: Hopcroft, Motwani, dan Ullman, "Introduction to Automata Theory, Languages, and Computation", 3rd Edition, 2006)

BAB II

PERANCANGAN DAN IMPLEMENTASI

2.1 Arsitektur Sistem

Entry point sistem analisis sintaksis terdapat pada kelas SyntaxAnalyzer yang menerima daftar token terurut hasil analisis leksikal dari kelas Lexer. Hasilnya adalah parse tree yang direpresentasikan dalam kelas-kelas Node yang terangkai secara rekursif. Parse tree memiliki leaf node berupa token-token, serta non-leaf node berupa simbol non-terminal dalam CFG. Aturan produksi CFG dibuat dengan menerapkan factory pattern yang di-hardcode dan menghasilkan callable procedure sebagai RHS yang kemudian didaftarkan dengan simbol non-terminalnya sebagai LHS dalam sebuah pasangan key-value LHS-RHS. Setiap prosedur dapat memanggil prosedur lainnya dengan mengakses kumpulan key-value tersebut. Proses parsing selalu dimulai dari simbol non-terminal <Program> yang akan memanggil secara rekursif prosedur-prosedur aturan produksi dari kumpulan key-value tersebut dan menghasilkan parse tree. Setelah selesai, parse tree akan dicetak ke layar.



Gambar 2.1. Arsitektur sistem
(Sumber: dokumen kelompok)

2.2 Aturan Produksi

Grammar Pascal-S yang diterapkan pada implementasi ini terdiri dari total 67 aturan produksi. Aturan-aturan produksi ini dikategorikan ke dalam beberapa kelompok fungsional untuk memudahkan pemahaman struktur bahasa tersebut.

2.2.1 Program dan Blok

Didefinisikan tiga aturan produksi fundamental yang membentuk kerangka dasar dari setiap program Pascal-S, yaitu <Program>, <ProgramHeader>, dan <Block>.

1. <Program> → <ProgramHeader> <Block> .

Aturan produksi pertama dan terpenting mendefinisikan bahwa sebuah Program (<Program>) terdiri dari tiga komponen wajib yang berurutan: header program (<ProgramHeader>), diikuti oleh blok (<Block>), dan wajib diakhiri dengan simbol titik (.). Ini memastikan bahwa setiap program memiliki pembukaan, isi utama, dan penutup yang jelas.

2. <ProgramHeader> → program IDENTIFIER ;

Aturan ini menjelaskan bagaimana *header* program (<ProgramHeader>) dibangun. Sebuah *header* program harus dimulai dengan kata kunci program yang diikuti oleh token bertipe IDENTIFIER yang berfungsi sebagai nama program dan diakhiri dengan titik koma (;).

3. <Block> → <DeclarationPart> <StatementPart>

Terakhir, aturan ini mendefinisikan struktur dari blok (<Block>), yaitu bagian utama dari program yang memuat logika dan data. Sebuah Blok terdiri dari dua bagian non-terminal yang berurutan: bagian deklarasi (<DeclarationPart>) yang berisi semua definisi variabel, konstanta, atau tipe, diikuti oleh bagian *statement* (<StatementPart>) yang berisi semua instruksi atau logika program yang akan dieksekusi.

Dengan tiga aturan ini, kerangka sintaksis program Pascal-S terjamin di mana program harus dinamai, memiliki deklarasi walaupun mungkin kosong, dan berisi serangkaian *statement* yang akan dijalankan.

2.2.2 Deklarasi

Didefinisikan aturan-aturan produksi yang membentuk bagian deklarasi dalam sebuah blok Pascal-S. Bagian deklarasi berfungsi sebagai tempat pendefinisan konstanta, tipe data, variabel, dan subprogram yang akan digunakan dalam program. Karena keseluruhan bagian deklarasi bersifat opsional dalam Pascal-S, grammar menggunakan banyak produksi yang dapat menghasilkan epsilon (kosong) untuk mencerminkan fleksibilitas ini.

1. $\langle \text{DeclarationPart} \rangle \rightarrow \langle \text{ConstDeclOpt} \rangle \langle \text{TypeDeclOpt} \rangle \langle \text{VarDeclOpt} \rangle$

Aturan produksi ini mendefinisikan struktur keseluruhan bagian deklarasi. $\langle \text{DeclarationPart} \rangle$ terdiri atas tiga komponen yang harus muncul dalam urutan tertentu: deklarasi konstanta opsional ($\langle \text{ConstDeclOpt} \rangle$), diikuti deklarasi tipe opsional ($\langle \text{TypeDeclOpt} \rangle$), dan deklarasi variabel opsional ($\langle \text{VarDeclOpt} \rangle$). Urutan wajib ini mencerminkan struktur standar Pascal. Bagian deklarasi subprogram berada di luar aturan ini dan menyusul setelah $\langle \text{DeclarationPart} \rangle$.

2. $\langle \text{ConstDeclOpt} \rangle \rightarrow \text{konstanta } \langle \text{ConstList} \rangle \mid \epsilon$

Aturan ini menyatakan bahwa deklarasi konstanta bersifat opsional. Jika bagian ini digunakan, deklarasi harus dimulai dengan kata kunci konstanta dan diikuti oleh satu atau lebih pasangan nama-nilai dalam $\langle \text{ConstList} \rangle$. Alternatif epsilon memungkinkan tidak adanya deklarasi konstanta apabila program tidak membutuhkannya.

3. $\langle \text{ConstList} \rangle \rightarrow \text{IDENTIFIER} = \langle \text{Constant} \rangle ; \langle \text{ConstList} \rangle \mid \epsilon$

Aturan produksi ini mendefinisikan daftar deklarasi konstanta. Setiap entri dalam daftar terdiri atas IDENTIFIER sebagai nama konstanta, operator relasional sama dengan (=), sebuah $\langle \text{Constant} \rangle$ sebagai nilainya, dan diakhiri dengan titik koma. Aturan bersifat rekursif sehingga dapat mendeklarasikan lebih dari satu konstanta secara berurutan, atau menghasilkan epsilon untuk mengakhiri daftar.

4. $\langle \text{TypeDeclOpt} \rangle \rightarrow \text{tipe } \langle \text{TypeList} \rangle \mid \epsilon$

Aturan ini menjelaskan bahwa deklarasi tipe data juga bersifat opsional. Jika digunakan, bagian ini harus diawali kata kunci tipe dan diikuti oleh $\langle \text{TypeList} \rangle$. Alternatif epsilon memungkinkan program tidak memiliki deklarasi tipe tambahan buatan pengguna.

5. $\langle \text{TypeList} \rangle \rightarrow \text{IDENTIFIER} = \langle \text{Type} \rangle ; \langle \text{TypeList} \rangle \mid \epsilon$

Aturan produksi ini mendefinisikan struktur deklarasi tipe. Masing-masing deklarasi terdiri atas IDENTIFIER sebagai nama tipe baru, simbol sama dengan, dan $\langle \text{Type} \rangle$ yang menjelaskan bentuk atau struktur tipe tersebut. Deklarasi kemudian diakhiri dengan titik koma. Aturan ini bersifat rekursif untuk memungkinkan beberapa tipe dideklarasikan secara berurutan, atau berhenti melalui epsilon.

6. $\langle \text{VarDeclOpt} \rangle \rightarrow \text{variabel } \langle \text{VarDeclList} \rangle \mid \epsilon$

Aturan ini menyatakan bahwa deklarasi variabel merupakan bagian opsional dari deklarasi. Jika digunakan, bagian ini dimulai dengan kata kunci variabel dan diikuti daftar deklarasi variabel ($\langle \text{VarDeclList} \rangle$). Alternatif epsilon memastikan bahwa blok dapat berfungsi meskipun tidak memiliki variabel.

7. $\langle \text{VarDeclList} \rangle \rightarrow \langle \text{VarDeclaration} \rangle ; \langle \text{VarDeclList} \rangle \mid \epsilon$

Aturan ini mendefinisikan daftar deklarasi variabel. Setiap entri harus berbentuk satu $\langle \text{VarDeclaration} \rangle$ yang kemudian diakhiri dengan titik koma. Aturan ini rekursif sehingga memungkinkan banyak deklarasi variabel dituliskan secara berurutan.

8. $\langle \text{VarDeclaration} \rangle \rightarrow \langle \text{IdentifierList} \rangle : \langle \text{Type} \rangle$

Aturan ini menyatakan bahwa satu deklarasi variabel terdiri atas daftar identifier ($\langle \text{IdentifierList} \rangle$) sebagai nama-nama variabel yang ingin didefinisikan, diikuti tanda titik dua, dan $\langle \text{Type} \rangle$ sebagai tipe datanya. Dengan demikian, satu pernyataan deklarasi dapat mencakup beberapa variabel yang memiliki tipe sama.

9. $\langle \text{IdentifierList} \rangle \rightarrow \text{IDENTIFIER } \langle \text{IdentifierListPrime} \rangle$

Aturan ini mendefinisikan bahwa daftar identifier selalu diawali oleh satu IDENTIFIER dan dapat diikuti oleh lanjutan daftar melalui $\langle \text{IdentifierListPrime} \rangle$.

10. $\langle \text{IdentifierListPrime} \rangle \rightarrow , \text{IDENTIFIER } \langle \text{IdentifierListPrime} \rangle \mid \epsilon$

Aturan produksi ini memberikan kelanjutan opsional bagi daftar identifier. Daftar dapat diperluas dengan menambahkan koma diikuti IDENTIFIER tambahan, dan aturan dapat berulang secara rekursif. Alternatif epsilon menandakan bahwa daftar identifier dapat berakhir setelah satu nama saja.

Dengan rangkaian aturan produksi ini, struktur deklarasi dalam Pascal-S terbentuk secara lengkap dan fleksibel. Grammar mengakomodasi pendefinisan konstanta, tipe, dan variabel dalam urutan yang baku tetapi tetap opsional, sehingga pemrogram dapat menyesuaikan kebutuhan deklarasi tanpa melanggar aturan sintaksis bahasa.

2.2.3 Tipe

Didefinisikan aturan-aturan produksi yang membentuk sistem tipe data dalam Pascal-S. Tipe data menentukan bentuk dan batasan nilai yang dapat disimpan dalam variabel, serta memastikan konsistensi operasi dalam program. Grammar berikut mengatur tiga kategori utama tipe, yaitu tipe sederhana, tipe array, dan tipe record, sekaligus mendefinisikan rentang nilai yang dapat digunakan dalam struktur array.

1. $\langle \text{Type} \rangle \rightarrow \langle \text{SimpleType} \rangle \mid \langle \text{ArrayType} \rangle \mid \langle \text{RecordType} \rangle$

Aturan produksi ini mendefinisikan bahwa sebuah tipe ($\langle \text{Type} \rangle$) dapat berada dalam salah satu dari tiga bentuk: tipe sederhana ($\langle \text{SimpleType} \rangle$), tipe array ($\langle \text{ArrayType} \rangle$), atau tipe record ($\langle \text{RecordType} \rangle$). Struktur ini memberikan fleksibilitas bagi pemrogram untuk menggunakan tipe dasar maupun tipe terstruktur yang lebih kompleks. Dengan demikian, aturan ini

menjadi dasar bagi sistem tipe data dalam Pascal-S yang mencakup berbagai bentuk representasi nilai.

2. $\text{<SimpleType>} \rightarrow \text{integer} \mid \text{real} \mid \text{boolean} \mid \text{char} \mid \text{IDENTIFIER}$

Aturan ini menjelaskan bahwa tipe sederhana meliputi empat tipe dasar bawaan—integer, real, boolean, dan char—serta IDENTIFIER yang merujuk pada tipe yang telah didefinisikan sebelumnya. Dengan menyediakan opsi IDENTIFIER, grammar memungkinkan pendefinisian tipe buatan pengguna, sehingga meningkatkan modularitas dan keterbacaan program.

3. $\text{<ArrayType>} \rightarrow \text{larik} \mid \text{<Range>} \mid \text{dari } \text{<Type>}$

Aturan produksi ini mendefinisikan bentuk tipe array. Kata kunci larik harus diikuti tanda kurung siku yang memuat sebuah <Range> sebagai batas indeks array, kemudian kata kunci dari, dan diakhiri dengan <Type> yang menentukan tipe elemen array. Struktur ini memastikan bahwa setiap array memiliki rentang indeks yang jelas dan tipe elemen yang konsisten.

4. $\text{<RecordType>} \rightarrow \text{rekaman } \text{<VarDeclOpt>} \text{ selesai}$

Aturan ini menetapkan bentuk tipe record, yaitu struktur data yang terdiri dari beberapa field dengan nama dan tipe masing-masing. Kata kunci rekaman harus diikuti bagian deklarasi variabel opsional (<VarDeclOpt>) yang mewakili field-field record, dan diakhiri dengan kata kunci selesai. Dengan aturan ini, grammar menyediakan mekanisme pembuatan tipe terstruktur yang dapat menampung berbagai data secara bersamaan.

5. $\text{<Range>} \rightarrow \text{<Constant>} \dots \text{<Constant>}$

Aturan produksi ini mendefinisikan batas rentang yang digunakan pada array. Nonterminal <Range> terdiri atas dua konstanta yang dipisahkan oleh operator dua titik (..), masing-masing menentukan nilai minimum dan maksimum indeks array. Struktur ini memastikan bahwa setiap tipe array memiliki batas yang eksplisit dan dapat divalidasi secara sintaksis.

Dengan keseluruhan aturan produksi ini, sistem tipe dalam Pascal-S tersusun secara lengkap dan terorganisasi. Grammar mendukung penggunaan tipe skalar, array berdimensi tetap, serta record dengan struktur fleksibel, sehingga pemrogram dapat membangun representasi data yang sesuai kebutuhan sambil tetap mengikuti kaidah sintaksis bahasa.

2.2.4 Konstanta

Didefinisikan aturan-aturan produksi yang membentuk struktur konstanta dalam Pascal-S. Konstanta merupakan nilai tetap yang tidak berubah selama program berjalan, dan grammar menyediakan beberapa bentuk representasi nilai, baik berupa angka, karakter, string, maupun nilai boolean. Aturan-aturan berikut menjelaskan bagaimana

konstanta dituliskan serta bagaimana tanda positif atau negatif dapat diterapkan pada nilai numerik.

1. $\langle \text{Constant} \rangle \rightarrow \langle \text{SignOpt} \rangle \langle \text{UnsignedConstant} \rangle \mid \text{STRING_LITERAL} \mid \text{CHAR_LITERAL} \mid \text{true} \mid \text{false}$

Aturan produksi ini mendefinisikan berbagai bentuk konstanta yang dapat digunakan dalam ekspresi. Nonterminal $\langle \text{Constant} \rangle$ dapat berupa sebuah konstanta numerik yang diawali tanda opsional ($\langle \text{SignOpt} \rangle$), sebuah literal string (STRING_LITERAL), literal karakter (CHAR_LITERAL), atau salah satu dari nilai boolean true dan false. Dengan menyediakan beberapa alternatif bentuk, aturan ini memungkinkan program merepresentasikan nilai tetap dari berbagai tipe dasar yang umum digunakan.

2. $\langle \text{UnsignedConstant} \rangle \rightarrow \text{NUMBER} \mid \text{IDENTIFIER}$

Aturan ini menjelaskan bahwa sebuah konstanta numerik tanpa tanda dapat berupa sebuah NUMBER atau IDENTIFIER. Kehadiran IDENTIFIER sebagai alternatif merepresentasikan konstanta yang sebelumnya telah dideklarasikan. Struktur ini memastikan grammar dapat membedakan antara penggunaan nilai angka langsung dan penggunaan nama konstanta yang merujuk pada nilai tertentu.

3. $\langle \text{SignOpt} \rangle \rightarrow \langle \text{Sign} \rangle \mid \epsilon$

Aturan produksi ini mendefinisikan tanda opsional yang dapat ditempelkan pada konstanta numerik. Nonterminal $\langle \text{SignOpt} \rangle$ dapat menghasilkan sebuah tanda sesuai aturan $\langle \text{Sign} \rangle$ atau menghasilkan epsilon (kosong). Hal ini membuat tanda positif atau negatif bersifat opsional tanpa mempengaruhi bentuk dasar konstanta.

4. $\langle \text{Sign} \rangle \rightarrow + \mid -$

Aturan ini menjelaskan bahwa tanda yang dapat digunakan untuk konstanta numerik hanyalah tanda tambah (+) atau tanda minus (-). Aturan ini memberikan batasan eksplisit mengenai bentuk tanda yang diperbolehkan dan menjadi dasar bagi penulisan nilai bertanda dalam ekspresi.

Dengan keempat aturan ini, struktur konstanta dalam Pascal-S ditetapkan secara komprehensif. Grammar mengakomodasi berbagai jenis nilai tetap dan mendukung penggunaan tanda opsional pada nilai numerik, sehingga penulisan konstanta dalam program dapat dilakukan secara fleksibel namun tetap sesuai dengan kaidah sintaksis bahasa.

2.2.5 Subprogram

Didefinisikan aturan-aturan produksi yang membentuk struktur subprogram dalam Pascal-S. Subprogram, yang terdiri dari prosedur dan fungsi, memungkinkan

pemrogram menyusun program secara modular dengan memisahkan bagian-bagian logika ke dalam unit yang dapat dipanggil ulang. Grammar berikut mengatur bagaimana deklarasi, parameter formal, serta struktur internal subprogram dituliskan secara konsisten.

1. $\langle\text{SubprogDeclList}\rangle \rightarrow \langle\text{SubprogramDeclaration}\rangle ; \langle\text{SubprogDeclList}\rangle | \epsilon$

Aturan produksi ini mendefinisikan daftar deklarasi subprogram. Nonterminal $\langle\text{SubprogDeclList}\rangle$ dapat menghasilkan satu deklarasi subprogram yang diikuti tanda titik koma dan berlanjut kembali ke $\langle\text{SubprogDeclList}\rangle$, atau menghasilkan epsilon (kosong). Dengan demikian, grammar mengizinkan program memiliki banyak subprogram secara berurutan maupun tidak memiliki subprogram sama sekali.

2. $\langle\text{SubprogramDeclaration}\rangle \rightarrow \langle\text{ProcedureDeclaration}\rangle | \langle\text{FunctionDeclaration}\rangle$

Aturan ini menyatakan bahwa sebuah deklarasi subprogram dapat berupa deklarasi prosedur atau deklarasi fungsi. Pemisahan dua bentuk ini penting karena masing-masing memiliki struktur dan tujuan yang berbeda: prosedur tidak mengembalikan nilai, sedangkan fungsi menghasilkan nilai.

3. $\langle\text{ProcedureDeclaration}\rangle \rightarrow \text{prosedur IDENTIFIER} \langle\text{FormalParamOpt}\rangle ; \langle\text{Block}\rangle$

Aturan produksi ini menjelaskan bentuk deklarasi prosedur. Kata kunci prosedur harus diikuti sebuah IDENTIFIER sebagai nama prosedur, kemudian bagian parameter formal opsional ($\langle\text{FormalParamOpt}\rangle$), tanda titik koma, dan sebuah $\langle\text{Block}\rangle$ yang berisi implementasi prosedur. Struktur ini memastikan bahwa prosedur memiliki nama dan blok eksekusi yang jelas, serta dapat menampung parameter bila diperlukan.

4. $\langle\text{FunctionDeclaration}\rangle \rightarrow \text{fungsi IDENTIFIER} \langle\text{FormalParamOpt}\rangle : \langle\text{SimpleType}\rangle ; \langle\text{Block}\rangle$

Aturan ini mendefinisikan bentuk deklarasi fungsi. Fungsi dituliskan dengan kata kunci fungsi, diikuti IDENTIFIER sebagai nama fungsi, parameter formal opsional, tanda titik dua yang diikuti $\langle\text{SimpleType}\rangle$ sebagai tipe nilai balik, lalu tanda titik koma dan sebuah $\langle\text{Block}\rangle$. Dengan struktur ini, grammar menetapkan bahwa fungsi tidak hanya memiliki logika internal, tetapi juga tipe nilai yang harus dihasilkan ketika fungsi dipanggil.

5. $\langle\text{FormalParamOpt}\rangle \rightarrow \langle\text{FormalParameterList}\rangle | \epsilon$

Aturan ini menyatakan bahwa bagian parameter formal dapat ada atau tidak. Penggunaan epsilon memungkinkan deklarasi prosedur maupun fungsi

tanpa parameter, sementara <FormalParameterList> digunakan bila subprogram memerlukan daftar parameter.

6. $\text{<FormalParameterList>} \rightarrow (\text{<ParamSectionList>})$

Aturan produksi ini menjelaskan bahwa daftar parameter formal harus dituliskan dalam tanda kurung dan terdiri dari satu atau lebih bagian parameter. Struktur ini memastikan format parameter tetap konsisten pada deklarasi subprogram apa pun.

7. $\text{<ParamSectionList>} \rightarrow \text{<ParamSection>} \text{<ParamSectionListPrime>}$

Aturan ini menyatakan bahwa daftar bagian parameter selalu dimulai dengan satu <ParamSection> dan dapat diikuti oleh kelanjutan yang ditentukan oleh <ParamSectionListPrime>. Dengan demikian, grammar mendukung satu maupun beberapa kelompok parameter.

8. $\text{<ParamSectionListPrime>} \rightarrow ; \text{<ParamSection>}$

$\text{<ParamSectionListPrime>} | \epsilon$

Aturan produksi ini memberikan kelanjutan opsional untuk daftar bagian parameter. Nonterminal ini dapat menghasilkan titik koma diikuti bagian parameter berikutnya, dan berulang secara rekursif. Alternatif epsilon memungkinkan daftar parameter berhenti setelah satu bagian saja, memberikan fleksibilitas jumlah parameter dalam subprogram.

9. $\text{<ParamSection>} \rightarrow \text{<VarKeywordOpt>} \text{<IdentifierList>} : \text{<SimpleType>}$

Aturan ini menjelaskan struktur satu bagian parameter. Sebuah <ParamSection> dapat diawali dengan kata kunci variabel secara opsional (<VarKeywordOpt>) yang menandakan parameter diturunkan secara by reference, diikuti daftar identifier yang menjadi nama parameter, tanda titik dua, dan tipe sederhana. Aturan ini memastikan setiap parameter memiliki nama dan tipe yang jelas, serta mendefinisikan cara penurunannya.

10. $\text{<VarKeywordOpt>} \rightarrow \text{variabel} | \epsilon$

Aturan ini mendefinisikan bahwa parameter dapat diberi kata kunci variabel bila ingin diteruskan sebagai by reference, atau menghasilkan epsilon bila parameter diteruskan secara by value. Struktur ini penting untuk membedakan efek pemanggilan terhadap nilai parameter.

Dengan keseluruhan aturan produksi ini, struktur subprogram dalam Pascal-S tersusun secara lengkap dan modular. Grammar menyediakan mekanisme untuk mendeklarasikan prosedur dan fungsi beserta parameter formalnya, memastikan bahwa subprogram dapat ditulis dan dipanggil dengan cara yang terstandarisasi, fleksibel, dan sesuai dengan kaidah bahasa.

2.2.6 Statement

Didefinisikan aturan-aturan produksi yang mengontrol struktur pernyataan dalam Pascal-S. Bagian ini merupakan inti dari alur eksekusi program, karena setiap perintah, percabangan, maupun pengulangan direpresentasikan melalui statement. Grammar berikut memformalkan berbagai bentuk pernyataan, mulai dari blok komposit, penugasan, pemanggilan prosedur, hingga konstruksi kontrol alur seperti if, while, for, repeat, dan case. Dengan aturan-aturan ini, perilaku program dapat ditentukan secara sistematis dan konsisten.

1. $\langle \text{StatementPart} \rangle \rightarrow \langle \text{CompoundStatement} \rangle$

Aturan ini menyatakan bahwa bagian pernyataan utama suatu blok selalu direpresentasikan melalui $\langle \text{CompoundStatement} \rangle$. Dengan demikian, setiap blok program memiliki satu kesatuan pernyataan yang terorganisasi secara eksplisit, selalu dibuka dan ditutup dengan batas sintaksis yang tetap.

2. $\langle \text{CompoundStatement} \rangle \rightarrow \text{mulai } \langle \text{StatementList} \rangle \text{ selesai}$

Aturan produksi ini mendefinisikan struktur pernyataan komposit. Sebuah $\langle \text{CompoundStatement} \rangle$ diawali kata kunci mulai, diikuti satu atau lebih pernyataan yang tercakup dalam $\langle \text{StatementList} \rangle$, dan diakhiri kata kunci selesai. Bentuk ini memungkinkan pengelompokan pernyataan dalam blok bersarang yang menjadi dasar struktur program.

3. $\langle \text{StatementList} \rangle \rightarrow \langle \text{Statement} \rangle \langle \text{StatementListPrime} \rangle \mid \epsilon$

Aturan ini menentukan bahwa sebuah daftar pernyataan dapat terdiri dari satu entri $\langle \text{Statement} \rangle$ yang diikuti bagian rekursif $\langle \text{StatementListPrime} \rangle$, atau dapat pula kosong. Dengan demikian, grammar mengizinkan blok yang memiliki satu pernyataan, sejumlah pernyataan, maupun blok kosong.

4. $\langle \text{StatementListPrime} \rangle \rightarrow ; \langle \text{Statement} \rangle \langle \text{StatementListPrime} \rangle \mid \epsilon$

Aturan ini mengatur pemisahan pernyataan dalam suatu daftar menggunakan titik koma. Mekanisme rekursi memungkinkan jumlah pernyataan yang tidak terbatas, sehingga $\langle \text{StatementList} \rangle$ dapat berkembang sesuai kebutuhan struktur program.

5. $\langle \text{Statement} \rangle \rightarrow \langle \text{AssignmentStatement} \rangle \mid \langle \text{ProcedureCall} \rangle \mid \langle \text{CompoundStatement} \rangle \mid \langle \text{IfStatement} \rangle \mid \langle \text{WhileStatement} \rangle \mid \langle \text{ForStatement} \rangle \mid \langle \text{RepeatStatement} \rangle \mid \langle \text{CaseStatement} \rangle \mid \epsilon$

Aturan produksi ini mendefinisikan himpunan bentuk dasar sebuah pernyataan. Nonterminal $\langle \text{Statement} \rangle$ dapat menghasilkan salah satu dari berbagai konstruksi sintaksis yang mencakup aksi (penugasan dan pemanggilan) serta kendali alur (if, while, for, repeat, dan case), termasuk kemungkinan kemunculan pernyataan kosong.

6. $\langle \text{AssignmentStatement} \rangle \rightarrow \text{IDENTIFIER } \langle \text{VariableTail} \rangle := \langle \text{Expression} \rangle$

Aturan ini mendeskripsikan bentuk pernyataan penugasan. IDENTIFIER memberikan nama variabel yang menjadi target, $\langle \text{VariableTail} \rangle$ memungkinkan akses tersegmentasi seperti indeks array atau field rekaman, dan operator $:=$ menghubungkannya dengan nilai hasil evaluasi $\langle \text{Expression} \rangle$. Struktur ini memastikan bahwa penugasan selalu ditujukan pada lokasi data yang terdefinisi.

7. $\langle \text{ProcedureCall} \rangle \rightarrow \text{IDENTIFIER } \langle \text{ParameterListOpt} \rangle$

Aturan ini mendefinisikan bentuk pemanggilan prosedur. Nama prosedur direpresentasikan oleh IDENTIFIER, sedangkan $\langle \text{ParameterListOpt} \rangle$ menyediakan daftar argumen aktual secara opsional. Dengan demikian, pemanggilan prosedur dapat muncul baik dengan parameter maupun tanpa parameter.

8. $\langle \text{IfStatement} \rangle \rightarrow \text{jika } \langle \text{Expression} \rangle \text{ maka } \langle \text{Statement} \rangle \langle \text{ElsePart} \rangle$

Aturan produksi ini membentuk struktur percabangan. Bagian $\langle \text{Expression} \rangle$ menentukan kondisi yang menjadi dasar pemilihan jalur eksekusi, $\langle \text{Statement} \rangle$ merepresentasikan cabang utama, sementara $\langle \text{ElsePart} \rangle$ menyediakan opsi cabang alternatif.

9. $\langle \text{ElsePart} \rangle \rightarrow \text{selain-itu } \langle \text{Statement} \rangle \mid \epsilon$

Aturan ini menyatakan bahwa bagian else bersifat opsional. Jika hadir, kata kunci selain-itu diikuti sebuah $\langle \text{Statement} \rangle$ sebagai cabang alternatif; jika tidak muncul, struktur percabangan tetap sah sebagai if tanpa else.

10. $\langle \text{WhileStatement} \rangle \rightarrow \text{selama } \langle \text{Expression} \rangle \text{ lakukan } \langle \text{Statement} \rangle$

Aturan ini mendefinisikan bentuk pengulangan while. Nonterminal $\langle \text{Expression} \rangle$ menentukan syarat iterasi, dan $\langle \text{Statement} \rangle$ menyatakan tubuh pengulangan. Mekanisme evaluasi kondisi sebelum eksekusi direpresentasikan secara struktural oleh urutan simbol-simbol dalam aturan ini.

11. $\langle \text{RepeatStatement} \rangle \rightarrow \text{ulangi } \langle \text{StatementList} \rangle \text{ sampai } \langle \text{Expression} \rangle$

Aturan produksi ini mendeskripsikan konstruksi repeat-until. $\langle \text{StatementList} \rangle$ dijalankan terlebih dahulu sebagai tubuh pengulangan, kemudian kondisi penghentian dievaluasi melalui $\langle \text{Expression} \rangle$. Urutan ini secara sintaksis memastikan bahwa pengulangan berjalan minimal satu kali.

12. $\langle \text{ForStatement} \rangle \rightarrow \text{untuk } \text{IDENTIFIER } := \langle \text{Expression} \rangle \langle \text{ForDirection} \rangle \langle \text{Expression} \rangle \text{ lakukan } \langle \text{Statement} \rangle$

Aturan ini menentukan struktur pengulangan for dengan variabel kontrol. IDENTIFIER berperan sebagai variabel iterasi, dua $\langle \text{Expression} \rangle$ memberikan batas awal dan batas akhir, sedangkan $\langle \text{ForDirection} \rangle$ menentukan arah

perubahan nilai variabel kontrol. Tubuh iterasi direpresentasikan melalui **<Statement>** yang mengikuti kata kunci lakukan.

13. $\text{<ForDirection>} \rightarrow \text{ke} \mid \text{turun-ke}$

Aturan ini menetapkan arah iterasi. Kata kunci ke menyatakan iterasi menaik, sedangkan turun-ke menyatakan iterasi menurun. Dengan demikian, arah perubahan variabel kontrol tidak ambigu dan dinyatakan secara eksplisit.

14. $\text{<CaseStatement>} \rightarrow \text{kasus } \text{<Expression>} \text{ dari } \text{<CaseList>} \text{ <CaseEndOpt>} \text{ selesai}$

Aturan produksi ini merumuskan bentuk pernyataan case. Nilai **<Expression>** dicocokkan dengan elemen-elemen dalam **<CaseList>**, dan **<CaseEndOpt>** menyediakan opsi titik koma sebelum penutup selesai. Struktur ini mewakili pemilihan berdasarkan kecocokan nilai yang bersifat diskrit.

15. $\text{<CaseList>} \rightarrow \text{<CaseElement>} \text{ <CaseListPrime>}$

Aturan ini mendefinisikan daftar elemen kasus yang terdiri dari satu **<CaseElement>** awal yang dapat diikuti oleh elemen-elemen tambahan melalui bagian rekursif **<CaseListPrime>**.

16. $\text{<CaseListPrime>} \rightarrow ; \text{ <CaseElement>} \text{ <CaseListPrime>} \mid \epsilon$

Aturan ini mengizinkan penambahan elemen case berikutnya, dipisahkan oleh titik koma. Penggunaan rekursi memungkinkan daftar elemen yang panjang dan fleksibel.

17. $\text{<CaseEndOpt>} \rightarrow ; \mid \epsilon$

Aturan ini menyediakan titik koma opsional sebelum penutup pernyataan case. Kehadirannya tidak wajib, tetapi grammar mengizinkan variasi penulisan tersebut.

18. $\text{<CaseElement>} \rightarrow \text{<Constant>} : \text{<Statement>}$

Aturan ini mendefinisikan struktur setiap elemen case. Sebuah konstanta **<Constant>** berfungsi sebagai nilai pencocokan, diikuti tanda titik dua dan **<Statement>** yang dieksekusi bila nilai ekspresi case sesuai dengan konstanta tersebut.

Dengan keseluruhan aturan produksi ini, struktur pernyataan dalam Pascal-S terbentuk secara sistematis dan hierarkis. Setiap bentuk aksi dan kontrol alur direpresentasikan secara formal, sehingga urutan dan batasan eksekusi program dapat ditentukan secara konsisten dalam analisis sintaksis.

2.2.7 Parameter Aktual

Didefinisikan aturan-aturan produksi yang mengatur cara penulisan dan penyusunan parameter aktual dalam pemanggilan prosedur atau fungsi pada Pascal-S. Parameter aktual merupakan bagian penting dari ekspresi pemanggilan karena menentukan nilai-nilai yang dikirimkan ke prosedur atau fungsi untuk diproses.

1. $\langle \text{ParameterListOpt} \rangle \rightarrow \langle \text{ParameterList} \rangle \mid \epsilon$

Aturan produksi ini mendefinisikan bahwa keberadaan daftar parameter bersifat opsional. Nonterminal $\langle \text{ParameterListOpt} \rangle$ dapat menghasilkan $\langle \text{ParameterList} \rangle$ apabila sebuah pemanggilan memerlukan parameter, atau menghasilkan epsilon (kosong) ketika pemanggilan tidak membutuhkan parameter sama sekali. Fleksibilitas ini memungkinkan grammar menangani pemanggilan prosedur atau fungsi dengan atau tanpa argumen.

2. $\langle \text{ParameterList} \rangle \rightarrow (\langle \text{ExpressionList} \rangle)$

Aturan ini menjelaskan bahwa sebuah daftar parameter ($\langle \text{ParameterList} \rangle$) harus dituliskan dalam tanda kurung dan berisi sebuah $\langle \text{ExpressionList} \rangle$. Struktur ini memastikan bahwa setiap pemanggilan yang memiliki parameter mengikuti sintaks yang baku, yaitu diawali dengan tanda kurung buka, diikuti daftar ekspresi yang menjadi nilai parameter, dan diakhiri dengan tanda kurung tutup.

3. $\langle \text{ExpressionList} \rangle \rightarrow \langle \text{Expression} \rangle \langle \text{ExpressionListPrime} \rangle$

Aturan produksi ini menyatakan bahwa $\langle \text{ExpressionList} \rangle$ selalu dimulai dengan sebuah $\langle \text{Expression} \rangle$ dan dapat dilanjutkan oleh $\langle \text{ExpressionListPrime} \rangle$. Dengan demikian, aturan ini memfasilitasi keberadaan satu parameter ataupun lebih, tergantung lanjutan yang dihasilkan oleh $\langle \text{ExpressionListPrime} \rangle$.

4. $\langle \text{ExpressionListPrime} \rangle \rightarrow , \langle \text{Expression} \rangle \langle \text{ExpressionListPrime} \rangle \mid \epsilon$

Aturan ini mendefinisikan kemungkinan kelanjutan daftar ekspresi dalam parameter. Nonterminal $\langle \text{ExpressionListPrime} \rangle$ dapat menghasilkan koma (,) yang diikuti ekspresi baru untuk membentuk daftar parameter yang lebih panjang, dan secara rekursif dapat berlanjut dengan struktur yang sama. Selain itu, bentuk epsilon diperbolehkan sehingga daftar ekspresi dapat berhenti setelah satu ekspresi saja. Dengan struktur ini, grammar mampu menangani daftar parameter dengan jumlah yang variatif dan fleksibel.

Dengan keempat aturan produksi ini, struktur parameter aktual dalam Pascal-S tersusun secara konsisten dan terstandarisasi. Grammar memastikan bahwa pemanggilan prosedur atau fungsi dapat ditulis dengan parameter opsional, daftar

parameter tunggal, maupun daftar parameter yang panjang, semuanya tetap mengikuti pola sintaksis yang jelas dan terdefinisi.

2.2.8 Ekspresi

Didefinisikan rangkaian aturan produksi yang mengatur pembentukan ekspresi dalam Pascal-S. Ekspresi merupakan komponen inti dari bahasa karena digunakan dalam evaluasi kondisi, perhitungan aritmetika, maupun operasi logika. Aturan-aturan berikut menyusun ekspresi secara bertingkat berdasarkan presedensi operator, dimulai dari ekspresi sederhana hingga kombinasi kompleks yang melibatkan operator relasional.

1. $\langle \text{Expression} \rangle \rightarrow \langle \text{SimpleExpression} \rangle \langle \text{ExpressionPrime} \rangle$

Aturan produksi ini mendefinisikan bahwa sebuah ekspresi ($\langle \text{Expression} \rangle$) selalu berawal dari $\langle \text{SimpleExpression} \rangle$ dan dapat diikuti oleh $\langle \text{ExpressionPrime} \rangle$. Struktur ini menggambarkan bahwa ekspresi dasar terdiri atas ekspresi sederhana, sementara bagian lanjutannya bersifat opsional dan menentukan apakah ekspresi tersebut juga mengandung operasi relasional. Dengan demikian, aturan ini menjadi fondasi bagi penyusunan ekspresi lengkap.

2. $\langle \text{ExpressionPrime} \rangle \rightarrow \langle \text{RelationalOperator} \rangle \langle \text{SimpleExpression} \rangle | \epsilon$

Aturan ini menjelaskan komponen opsional dari sebuah ekspresi. Nonterminal $\langle \text{ExpressionPrime} \rangle$ dapat menghasilkan sebuah operator relasional yang diikuti $\langle \text{SimpleExpression} \rangle$, atau menghasilkan epsilon (kosong). Kehadiran bentuk kosong ini memungkinkan sebuah ekspresi hanya berupa ekspresi sederhana tanpa memiliki perbandingan. Namun ketika operator relasional digunakan, aturan ini memastikan struktur ekspresi relasional tetap konsisten dan terdefinisi jelas.

3. $\langle \text{SimpleExpression} \rangle \rightarrow \langle \text{SignedTerm} \rangle \langle \text{SimpleExpressionPrime} \rangle$

Aturan produksi ini menyatakan bahwa sebuah ekspresi sederhana dibangun dari sebuah istilah bertanda ($\langle \text{SignedTerm} \rangle$) dan dapat dilanjutkan oleh $\langle \text{SimpleExpressionPrime} \rangle$. Dengan demikian, ekspresi sederhana dapat terdiri atas satu atau lebih komponen term yang dihubungkan operator aditif. Struktur ini juga mengatur bagaimana tanda awal (+ atau -) diperlakukan pada elemen pertama ekspresi.

4. $\langle \text{SignedTerm} \rangle \rightarrow \langle \text{SignOpt} \rangle \langle \text{Term} \rangle$

Aturan ini menjelaskan bahwa sebuah $\langle \text{SignedTerm} \rangle$ diawali oleh $\langle \text{SignOpt} \rangle$, yaitu tanda positif atau negatif yang bersifat opsional, kemudian diikuti sebuah $\langle \text{Term} \rangle$. Penggunaan tanda opsional ini memungkinkan ekspresi sederhana dimulai dengan nilai bertanda tanpa mengubah struktur keseluruhan grammar.

5. $\langle \text{SimpleExpressionPrime} \rangle \rightarrow \langle \text{AdditiveOperator} \rangle \langle \text{Term} \rangle$
 $\langle \text{SimpleExpressionPrime} \rangle | \epsilon$

Aturan ini mendefinisikan kelanjutan opsional dari ekspresi sederhana. Nonterminal ini dapat menghasilkan operator aditif yang diikuti oleh $\langle \text{Term} \rangle$, dan secara rekursif diikuti lagi oleh $\langle \text{SimpleExpressionPrime} \rangle$. Hal ini memungkinkan deretan operasi seperti penjumlahan dan pengurangan, serta penggunaan operator logika OR, untuk dituliskan dalam satu rangkaian ekspresi. Jika tidak ada operator lanjutan, aturan ini dapat menghasilkan epsilon sehingga ekspresi sederhana dapat berhenti.

6. $\langle \text{Term} \rangle \rightarrow \langle \text{Factor} \rangle \langle \text{TermPrime} \rangle$

Aturan ini menyatakan bahwa sebuah term dibangun dari sebuah faktor ($\langle \text{Factor} \rangle$) dan dapat diikuti oleh $\langle \text{TermPrime} \rangle$. Struktur ini menjadi dasar bagi penyusunan ekspresi tingkat menengah yang melibatkan operasi perkalian, pembagian, dan operasi logika AND.

7. $\langle \text{TermPrime} \rangle \rightarrow \langle \text{MultiplicativeOperator} \rangle \langle \text{Factor} \rangle \langle \text{TermPrime} \rangle | \epsilon$

Aturan produksi ini menjelaskan kemungkinan lanjutan dari sebuah term. $\langle \text{TermPrime} \rangle$ dapat menghasilkan operator multiplikatif diikuti $\langle \text{Factor} \rangle$, dan dapat berlanjut secara rekursif untuk membentuk rangkaian operasi yang memiliki presedensi tertinggi. Dengan adanya bentuk epsilon, sebuah term dapat tetap valid walaupun tidak memiliki kelanjutan operator tambahan.

Dengan keseluruhan aturan ini, struktur ekspresi dalam Pascal-S tersusun secara hierarkis dan konsisten. Grammar memisahkan presedensi operator dengan jelas, mulai dari faktor, term, ekspresi sederhana, hingga ekspresi relasional, sehingga proses evaluasi dapat dijalankan sesuai aturan bahasa yang terstandarisasi.

2.2.9 Faktor dan Variabel

Didefinisikan aturan-aturan produksi yang membentuk struktur dasar faktor dan variabel dalam ekspresi Pascal-S. Kedua komponen ini memainkan peran penting dalam proses evaluasi ekspresi karena menjadi elemen paling dasar yang dapat berupa nilai, identifikasi variabel, pemanggilan fungsi, atau ekspresi bertingkat.

**1. $\langle \text{Factor} \rangle \rightarrow \text{IDENTIFIER} \langle \text{FactorTail} \rangle | \langle \text{Constant} \rangle | (\langle \text{Expression} \rangle) |$
tidak $\langle \text{Factor} \rangle$**

Aturan produksi ini mendefinisikan macam-macam bentuk faktor yang dapat muncul dalam sebuah ekspresi. Nonterminal $\langle \text{Factor} \rangle$ dapat berupa sebuah IDENTIFIER yang kemudian diikuti oleh $\langle \text{FactorTail} \rangle$, sebuah konstanta, sebuah ekspresi yang dibungkus dalam tanda kurung, atau sebuah faktor yang diawali oleh operator negasi logika tidak. Struktur ini memastikan bahwa faktor dapat mencakup berbagai jenis nilai, baik nilai sederhana,

pemanggilan fungsi, akses variabel, maupun negasi logika, sehingga interpretasi ekspresi dapat berjalan secara fleksibel.

2. $\langle \text{FactorTail} \rangle \rightarrow \langle \text{ParameterList} \rangle \mid \langle \text{VariableTail} \rangle$

Aturan ini menjelaskan kelanjutan dari sebuah IDENTIFIER ketika digunakan sebagai faktor. Nonterminal $\langle \text{FactorTail} \rangle$ dapat menghasilkan $\langle \text{ParameterList} \rangle$ apabila IDENTIFIER tersebut merupakan pemanggilan fungsi atau prosedur, atau $\langle \text{VariableTail} \rangle$ apabila IDENTIFIER tersebut diakses sebagai variabel. Dengan demikian, aturan ini berfungsi untuk membedakan konteks penggunaan IDENTIFIER sehingga makna penggunaannya dapat ditentukan secara tepat oleh grammar.

3. $\langle \text{VariableTail} \rangle \rightarrow [\langle \text{Expression} \rangle] \langle \text{VariableTail} \rangle \mid . \text{IDENTIFIER}$ $\langle \text{VariableTail} \rangle \mid \epsilon$

Aturan produksi ini mendefinisikan pola akses lanjutan pada sebuah variabel. Nonterminal $\langle \text{VariableTail} \rangle$ dapat menghasilkan bentuk indeks array melalui $[\langle \text{Expression} \rangle]$, atau menghasilkan akses terhadap field dalam suatu record melalui $. \text{IDENTIFIER}$, dan keduanya dapat berlanjut secara rekursif. Selain itu, aturan ini juga mengizinkan epsilon (kosong), yang berarti variabel dapat berhenti tanpa perlu akses lanjutan tambahan. Struktur ini memungkinkan grammar menangani variabel dengan tingkat kompleksitas beragam, mulai dari variabel sederhana, elemen array multidimensi, hingga field dalam record yang bertingkat.

Dengan ketiga aturan ini, struktur faktor dan variabel dalam Pascal-S tersusun secara lengkap. Grammar mampu membedakan antara nilai sederhana, pemanggilan fungsi, akses variabel kompleks, serta operasi negasi, sehingga evaluasi ekspresi dapat dilakukan secara konsisten sesuai aturan bahasa.

2.2.10 Operator

Didefinisikan aturan-aturan produksi yang berkaitan dengan operator dalam Pascal-S. Tiga kategori operator didefinisikan untuk mendukung ekspresi aritmetika maupun logika, yaitu operator relasional, operator aditif, dan operator multiplikatif. Masing-masing kategori memiliki himpunan simbol yang menentukan bagaimana nilai dievaluasi dalam suatu ekspresi.

1. $\langle \text{RelationalOperator} \rangle \rightarrow = \mid < \mid <= \mid > \mid >=$

Aturan produksi ini mendefinisikan himpunan operator yang digunakan untuk melakukan perbandingan antar nilai. Nonterminal $\langle \text{RelationalOperator} \rangle$ dapat menghasilkan salah satu dari enam simbol relasional, yaitu sama dengan ($=$), tidak sama dengan ($<$), lebih kecil dari ($<$), lebih kecil atau sama dengan ($<=$), lebih besar dari ($>$), atau lebih besar atau sama dengan ($>=$). Kehadiran

operator-operator ini memungkinkan program melakukan evaluasi kondisi yang menjadi dasar pengambilan keputusan dalam struktur kendali.

2. <AdditiveOperator> → + | - | atau

Aturan ini menjelaskan kelompok operator yang digunakan dalam operasi penjumlahan dan pengurangan, baik numerik maupun logika. Nonterminal <AdditiveOperator> dapat menghasilkan simbol tambah (+), simbol kurang (-), atau kata kunci atau. Penggunaan kata kunci atau berfungsi sebagai operator logika OR dalam ekspresi boolean, sementara simbol + dan - digunakan dalam konteks aritmetika. Dengan demikian, kategori ini memadukan fungsi aritmetika dan logika dalam satu himpunan operator.

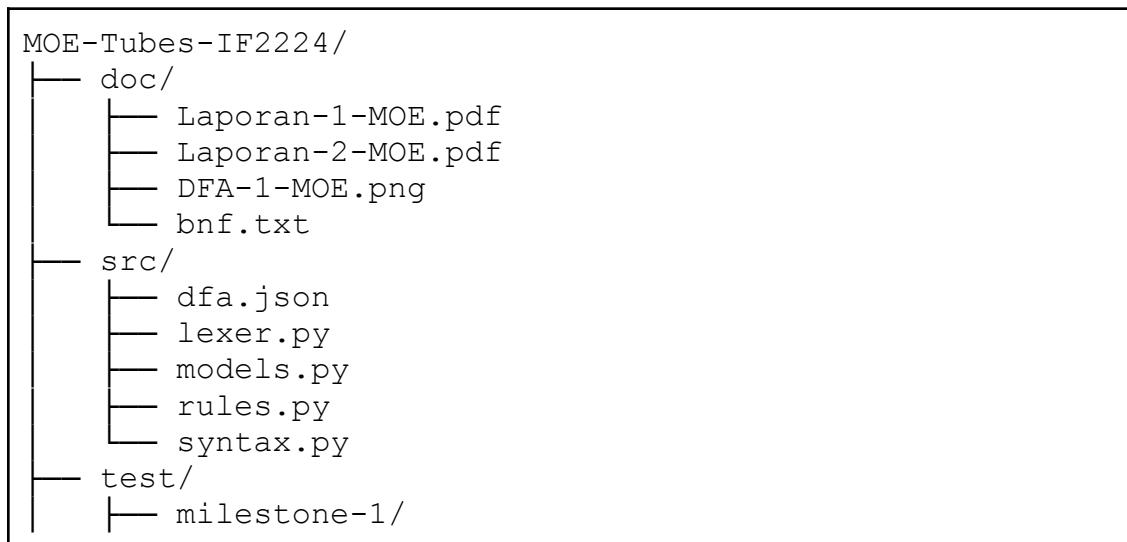
3. <MultiplicativeOperator> → * | / | bagi | mod | dan

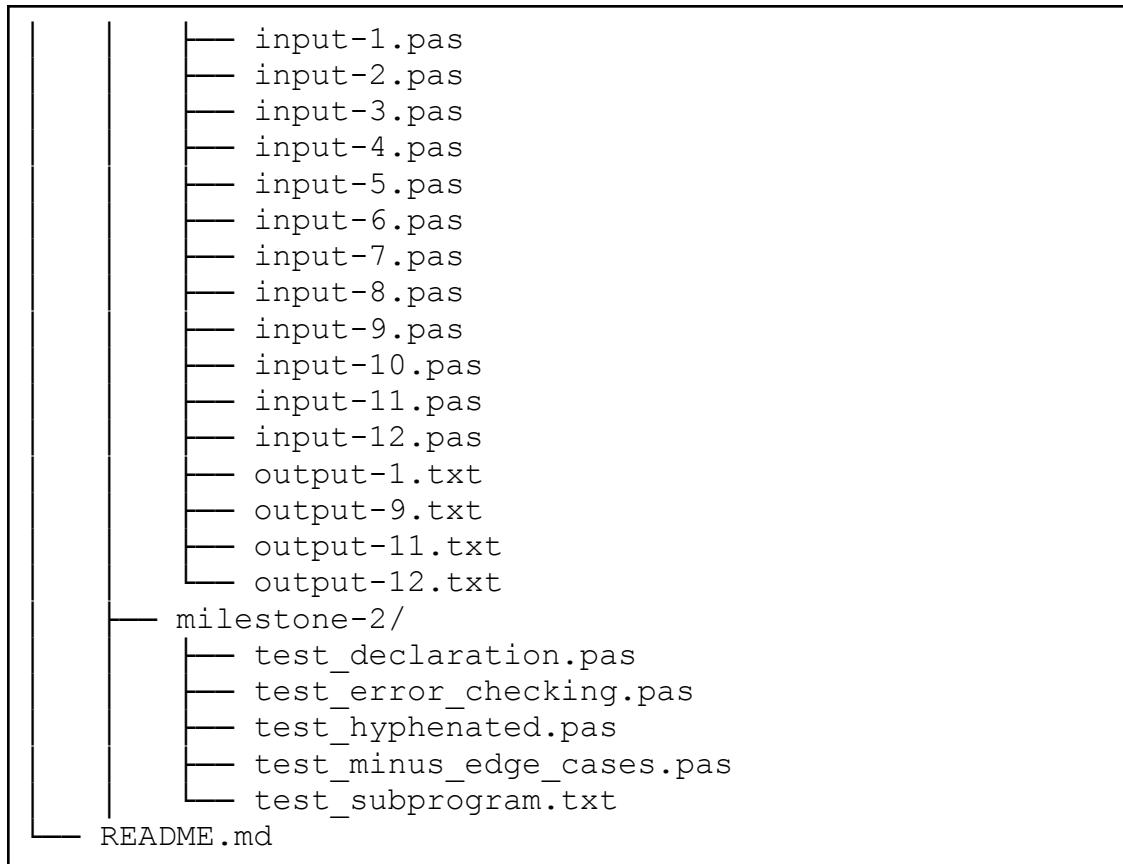
Aturan ini mendefinisikan operator-operator yang berada pada tingkat presedensi lebih tinggi dalam ekspresi. Nonterminal <MultiplicativeOperator> dapat menghasilkan simbol perkalian (*), pembagian (/), kata kunci bagi sebagai alternatif pembagian, operator modulus (mod), serta operator logika dan. Kombinasi operator aritmetika dan logika ini memungkinkan evaluasi ekspresi yang lebih kompleks di mana operasi tingkat rendah dapat dihitung sebelum operator aditif maupun relasional diproses.

Dengan ketiga aturan produksi ini, sistem operator dalam Pascal-S terbentuk secara lengkap dan terstruktur. Setiap kategori operator memiliki fungsi yang spesifik namun saling melengkapi, sehingga ekspresi dalam program dapat dievaluasi dengan aturan presedensi dan makna yang konsisten.

2.3 Implementasi Algoritma

2.3.1 Struktur File





2.3.2 Kelas dan Fungsi

Kelas dan fungsi diimplementasikan dalam bahasa pemrograman python. Adapun kelas dan fungsi yang ditulis dalam laporan ini adalah kelas dan fungsi yang berkaitan dengan analisis sintaksis tanpa menyertakan kelas dan fungsi dari milestone sebelumnya.

1. models.py: Token

```

# ===== Class for Token =====

class TokenType(str):
    pass
class Lexeme(str):
    pass

@dataclass(frozen=True)
class Token:
    token_type: TokenType
    lexeme: Lexeme
    # Posisi, default None jika tidak diberikan
    line: Optional[int] = field(default=None)
    column: Optional[int] = field(default=None)

    def __iter__(self):
        yield self.token_type
        yield self.lexeme

    def __str__(self):
        return f"{self.token_type}({self.lexeme})"
  
```

```

# Override metode __eq__ untuk perbandingan hanya berdasarkan token_type dan lexeme
def __eq__(self, other):
    if not isinstance(other, Token):
        return NotImplemented
    return (self.token_type == other.token_type) and (self.lexeme == other.lexeme)

# Override __hash__ untuk memastikan hash yang konsisten dengan __eq__
def __hash__(self):
    return hash((self.token_type, self.lexeme))

```

2. models.py: Parse Tree

```

# ===== Class for Parse Tree =====

class NonTerminal(str):
    pass

class Node:
    value: NonTerminal|Token
    children: List["Node"] = []

    def __init__(self, value: NonTerminal|Token):
        self.value = value
        self.children: List["Node"] = []

    def addChild(self, node:"Node") -> None:
        self.children.append(node)

    def addChildren(self, nodes:List["Node"]) -> None:
        self.children.extend(nodes)

    def __str__(self, level=0, prefix="", is_last=True):

        # 1. Tentukan string untuk node ini
        if level == 0:
            # Node root tidak memiliki konektor
            tree_str = f"{self.value}\n"
            child_prefix = "" # Anak dari root tidak punya awalan
        else:
            # Node anak memiliki konektor
            connector = "└─ " if is_last else "├─ "
            tree_str = f"{prefix}{connector}{self.value}\n"

        # Tentukan awalan untuk anak-anak dari nodeINI
        child_prefix = prefix + ("    " if is_last else "|   ")

        # 2. Rekursif panggil untuk semua anak
        for i, child in enumerate(self.children):
            # Cek apakah anak tersebut adalah anak terakhir
            is_child_last = (i == len(self.children) - 1)
            # Tambahkan representasi string anak ke string tree utama
            tree_str += child.__str__(level + 1, prefix=child_prefix, is_last=is_child_last)

        return tree_str

class Epsilon:
    pass

```

3. models.py: CFG

```

# ===== Class for CFG =====

class CFG:
    production_rules: Dict[NonTerminal, Callable[[], Node|None]] # Diubah
    tokens: List[Token]
    currentTokenID = 0
    # UNTUK ERROR REPORTING
    max_error_info: Dict[str, Any]

```

```

def __init__(self):
    self.production_rules = {}
    self.currentTokenID = 0

    # Inisialisasi pelacak error
    self.max_error_info = {
        'max_id': -1,           # Token ID terjauh yang gagal
        'expected': set(),     # Apa yang diharapkan di posisi itu (bisa banyak)
        'found': None          # Token apa yang ditemukan di posisi itu
    }

# GET TOKEN
def nextToken(self) -> None:
    self.currentTokenID += 1
def prevToken(self) -> None:
    self.currentTokenID -= 1
def currentToken(self) -> Token:
    # Tambahkan penjaga agar tidak error di akhir token
    if self.currentTokenID >= len(self.tokens):
        return Token(TokenType("EOF"), Lexeme("EOF")) # Token Penjaga
    return self.tokens[self.currentTokenID]

# ERROR REPORTING
def record_error(self, expected_symbol: TokenType, found_token: Token):
    """Mencatat kegagalan jika ini adalah yang 'terjauh'."""
    current_fail_id = self.currentTokenID

    # Menemukan error di posisi yang LEBIH JAUH.
    if current_fail_id > self.max_error_info['max_id']:
        self.max_error_info['max_id'] = current_fail_id
        self.max_error_info['expected'] = {expected_symbol}
        self.max_error_info['found'] = found_token

    # Menemukan error di posisi terjauh yang SAMA.
    elif current_fail_id == self.max_error_info['max_id']:
        # Tambahkan 'symbol' ini sebagai ekspektasi alternatif (contoh: expect ID or NUMBER)
        self.max_error_info['expected'].add(expected_symbol)

# PRODUCTION RULES
def addRules(self, rules: Dict[NonTerminal, List[List[NonTerminal|Token|TokenType|Epsilon]]]) -> None:
    for lhs, rhs in rules.items():

        # Rule yang didaftarkan
        # 'Kunci' nilai 'lhs' dan 'rhs' saat ini menggunakan argumen default.
        def execute_rules(current_lhs=lhs, current_rhs=rhs) -> Node|None:
            newNode = Node(current_lhs)

            # Simpan posisi token untuk backtracking
            initial_token_id = self.currentTokenID

            # Coba setiap Alternatif
            for alternative in current_rhs:
                isMatch = True
                childNodes:List[Node] = []

                # Reset token pointer untuk setiap alternatif baru
                self.currentTokenID = initial_token_id

                # Try Apply
                # print(f"Alternative Check: {alternative}")
                for symbol in alternative:
                    # print(f"Matching: symbol={symbol} token={self.currentToken()}")
                    # Inisialisasi childNode
                    childNode: Node = None

                    if isinstance(symbol, NonTerminal):
                        childNode = self.parse(symbol) # Panggil rekursif
                        # Cek apakah parse gagal
                        if childNode is None:
                            isMatch = False
                            # JANGAN RECORD ERROR DI SINI: biarkan dia cari alternatif

                    elif isinstance(symbol, Token):
                        if symbol == self.currentToken():
                            isMatch = True
                            childNode = Node(self.currentToken())
                            # KONSUMSI TOKEN
                            self.nextToken()
                        else:
                            isMatch = False

                if isMatch:
                    newNode.addChild(childNodes)
                    break

            return newNode

        self.production_rules[lhs] = execute_rules

```

```

        isMatch = False
        # RECORD ERROR
        self.record_error(symbol, self.currentToken())

    elif isinstance(symbol, TokenType):
        if symbol == self.currentToken().token_type:
            isMatch = True
            childNode = Node(self.currentToken())
            # KONSUMSI TOKEN
            self.nextToken()
        else:
            isMatch = False
            # PANGGIL PELACAK ERROR
            self.record_error(symbol, self.currentToken())

    elif isinstance(symbol, Epsilon):
        isMatch = True
        continue # Epsilon tidak menghasilkan node anak dan tidak konsumsi token

    if isMatch:
        if childNode: # Hanya tambahkan jika node berhasil dibuat
            childNodes.append(childNode)
        else:
            # Gagal di tengah alternatif, hentikan loop 'symbol' ini
            break

    # Alternatif Cocok (seluruh 'alternative' berhasil di-match)
    if isMatch:
        newNode.addChildren(childNodes)
        # Berhasil! Kembalikan node yang sudah diisi
        return newNode

    # Jika semua alternatif sudah dicoba dan TIDAK ADA yang cocok
    # Reset token ke posisi awal (sebelum 'execute_rules' ini dipanggil)
    self.currentTokenID = initial_token_id
    # Kembalikan None untuk menandakan kegagalan
    return None

    # Daftarkan Rule
    self.production_rules[lhs] = execute_rules

def parse(self, lhs:NonTerminal=NonTerminal("<Program>")) -> Node|None: # Mengembalikan None jika gagal
    # print(f'Mencoba parsing aturan: {lhs}')
    if lhs not in self.production_rules:
        raise Exception(f'Aturan produksi untuk {lhs} tidak ditemukan.')
    return self.production_rules[lhs]()

def parseToken(self, tokens:List[Token]) -> Node|None:
    self.tokens = tokens
    return self.parse()

```

4. rules.py: Aturan Produksi

```

# Tipe data untuk kejelasan, meskipun tidak divalidasi oleh 'models.py'
Symbol = Union[NonTerminal, Token, TokenType, Epsilon]
Alternative = List[Symbol]
Production = List[Alternative]
RuleDict = Dict[NonTerminal, Production]

def getAllProductionRules() -> RuleDict:
    """
    Mendefinisikan dan mengembalikan seluruh aturan produksi (CFG)
    untuk bahasa PASCAL-S dalam format yang diharapkan oleh parser CFG.

    Telah dilakukan Left-Factoring pada <Factor> dan <AssignmentStatement>
    untuk menangani ambiguitas 'IDENTIFIER' (variabel vs. fungsi).
    """

    rules: RuleDict = {

        # --- Program & Block ---

        # <Program> ::= <ProgramHeader> <Block> DOT(.)
        NonTerminal("<Program>"): [
            [NonTerminal("<ProgramHeader>"), NonTerminal("<Block>"), Token("DOT", ".")]

```

```

],
# <ProgramHeader> ::= KEYWORD(program) IDENTIFIER SEMICOLON(;)
NonTerminal("<ProgramHeader>"): [
    [Token("KEYWORD", "program"), TokenType("IDENTIFIER"), Token("SEMICOLON", ";")]
],
# <Block> ::= <DeclarationPart> <StatementPart>
NonTerminal("<Block>"): [
    [NonTerminal("<DeclarationPart>"), NonTerminal("<StatementPart>")]
],
# --- Bagian Deklarasi ---
# <DeclarationPart> ::= <ConstDeclOpt> <TypeDeclOpt> <VarDeclOpt> <SubprogDeclList>
NonTerminal("<DeclarationPart>"): [
    [NonTerminal("<ConstDeclOpt>"), NonTerminal("<TypeDeclOpt>"), NonTerminal("<VarDeclOpt>"),
     NonTerminal("<SubprogDeclList>")]
],
# <ConstDeclOpt> ::= KEYWORD(konstanta) <ConstList> | <Epsilon>
NonTerminal("<ConstDeclOpt>"): [
    [Token("KEYWORD", "konstanta"), NonTerminal("<ConstList>"),
     [Epsilon()]]
],
# <ConstList> ::= IDENTIFIER RELATIONAL_OPERATOR(=) <Constant> SEMICOLON(); <ConstList> | <Epsilon>
NonTerminal("<ConstList>"): [
    [TokenType("IDENTIFIER"), Token("RELATIONAL_OPERATOR", "="), NonTerminal("<Constant>"),
     Token("SEMICOLON", ";"), NonTerminal("<ConstList>"),
     [Epsilon()]]
],
# <TypeDeclOpt> ::= KEYWORD(tipe) <TypeList> | <Epsilon>
NonTerminal("<TypeDeclOpt>"): [
    [Token("KEYWORD", "tipe"), NonTerminal("<TypeList>"),
     [Epsilon()]]
],
# <TypeList> ::= IDENTIFIER RELATIONAL_OPERATOR(=) <Type> SEMICOLON(); <TypeList> | <Epsilon>
NonTerminal("<TypeList>"): [
    [TokenType("IDENTIFIER"), Token("RELATIONAL_OPERATOR", "="), NonTerminal("<Type>"),
     Token("SEMICOLON", ";"), NonTerminal("<TypeList>"),
     [Epsilon()]]
],
# <VarDeclOpt> ::= KEYWORD(variabel) <VarDeclList> | <Epsilon>
NonTerminal("<VarDeclOpt>"): [
    [Token("KEYWORD", "variabel"), NonTerminal("<VarDeclList>"),
     [Epsilon()]]
],
# <VarDeclList> ::= <VarDeclaration> SEMICOLON(); <VarDeclList> | <Epsilon>
NonTerminal("<VarDeclList>"): [
    [NonTerminal("<VarDeclaration>"), Token("SEMICOLON", ";"), NonTerminal("<VarDeclList>"),
     [Epsilon()]]
],
# <VarDeclaration> ::= <IdentifierList> COLON(:) <Type>
NonTerminal("<VarDeclaration>"): [
    [NonTerminal("<IdentifierList>"), Token("COLON", ":"), NonTerminal("<Type>")]
],
# <IdentifierList> ::= IDENTIFIER <IdentifierListPrime>
NonTerminal("<IdentifierList>"): [
    [TokenType("IDENTIFIER"), NonTerminal("<IdentifierListPrime>")]
],
# <IdentifierListPrime> ::= COMMA(,) IDENTIFIER <IdentifierListPrime> | <Epsilon>
NonTerminal("<IdentifierListPrime>"): [
    [Token("COMMA", ","), TokenType("IDENTIFIER"), NonTerminal("<IdentifierListPrime>"),
     [Epsilon()]]
],
# --- Definisi Tipe ---
# <Type> ::= <SimpleType> | <ArrayType> | <RecordType>
NonTerminal("<Type>"): [
    [NonTerminal("<SimpleType>")],
    [NonTerminal("<ArrayType>")]
]

```

```

        [NonTerminal("<RecordType>")]
    ],
    # <SimpleType> ::= KEYWORD(integer) | KEYWORD(real) | ... | IDENTIFIER
    NonTerminal("<SimpleType>"): [
        [Token("KEYWORD", "integer")],
        [Token("KEYWORD", "real")],
        [Token("KEYWORD", "boolean")],
        [Token("KEYWORD", "char")],
        [TokenType("IDENTIFIER")] # Tipe kustom
    ],
    # <ArrayType> ::= KEYWORD(larik) LBRACKET([]) <Range> RBRACKET() KEYWORD(dari) <Type>
    NonTerminal("<ArrayType>"): [
        [Token("KEYWORD", "larik"), Token("LBRACKET", "["), NonTerminal("<Range>"), Token("RBRACKET", "]"), Token("KEYWORD", "dari"), NonTerminal("<Type>")]
    ],
    # <RecordType> ::= KEYWORD(rekaman) <VarDeclOpt> KEYWORD(selesai)
    NonTerminal("<RecordType>"): [
        [Token("KEYWORD", "rekaman"), NonTerminal("<VarDeclOpt>"), Token("KEYWORD", "selesai")]
    ],
    # <Range> ::= <Constant> RANGE_OPERATOR(..) <Constant>
    NonTerminal("<Range>"): [
        [NonTerminal("<Constant>"), Token("RANGE_OPERATOR", ".."), NonTerminal("<Constant>")]
    ],
    # --- Konstanta ---
    # <Constant> ::= <SignOpt> <UnsignedConstant> | STRING_LITERAL | CHAR_LITERAL | KEYWORD(true) | KEYWORD(false)
    NonTerminal("<Constant>"): [
        [NonTerminal("<SignOpt>"), NonTerminal("<UnsignedConstant>")],
        [TokenType("STRING_LITERAL")],
        [TokenType("CHAR_LITERAL")],
        [Token("KEYWORD", "true")],
        [Token("KEYWORD", "false")]
    ],
    # <UnsignedConstant> ::= NUMBER | IDENTIFIER
    NonTerminal("<UnsignedConstant>"): [
        [TokenType("NUMBER")],
        [TokenType("IDENTIFIER")] # Konstanta yg didefinisikan
    ],
    # <SignOpt> ::= <Sign> | <Epsilon>
    NonTerminal("<SignOpt>"): [
        [NonTerminal("<Sign>")],
        [Epsilon()]
    ],
    # <Sign> ::= ARITHMETIC_OPERATOR(+) | ARITHMETIC_OPERATOR(-)
    NonTerminal("<Sign>"): [
        [Token("ARITHMETIC_OPERATOR", "+")],
        [Token("ARITHMETIC_OPERATOR", "-")]
    ],
    # --- Deklarasi Subprogram ---
    # <SubprogDeclList> ::= <SubprogramDeclaration> SEMICOLON(); <SubprogDeclList> | <Epsilon>
    NonTerminal("<SubprogDeclList>"): [
        [NonTerminal("<SubprogramDeclaration>"), Token("SEMICOLON", ";"),
        NonTerminal("<SubprogDeclList>"),
        [Epsilon()]
    ],
    # <SubprogramDeclaration> ::= <ProcedureDeclaration> | <FunctionDeclaration>
    NonTerminal("<SubprogramDeclaration>"): [
        [NonTerminal("<ProcedureDeclaration>")],
        [NonTerminal("<FunctionDeclaration>")]
    ],
    # <ProcedureDeclaration> ::= KEYWORD(prosedur) IDENTIFIER <FormalParamOpt> SEMICOLON(); <Block>
    NonTerminal("<ProcedureDeclaration>"): [
        [Token("KEYWORD", "prosedur"), TokenType("IDENTIFIER"), NonTerminal("<FormalParamOpt>"),
        Token("SEMICOLON", ";"), NonTerminal("<Block>")]
    ],
    # <FunctionDeclaration> ::= KEYWORD(fungsi) IDENTIFIER <FormalParamOpt> COLON(:) <SimpleType>

```

```

SEMICOLON() <Block>
    NonTerminal("<FunctionDeclaration>"): [
        [Token("KEYWORD", "fungsi"), TokenType("IDENTIFIER"), NonTerminal("<FormalParamOpt>"),
        Token("COLON", ":"), NonTerminal("<SimpleType>"), Token("SEMICOLON", ";"), NonTerminal("<Block>")]
    ],

    # <FormalParamOpt> ::= <FormalParameterList> | <Epsilon>
    NonTerminal("<FormalParamOpt>"): [
        [NonTerminal("<FormalParameterList>")],
        [<Epsilon>]
    ],

    # <FormalParameterList> ::= LPARENTHESIS() <ParamSectionList> RPARENTHESIS()
    NonTerminal("<FormalParameterList>"): [
        [Token("LPARENTHESIS", "("), NonTerminal("<ParamSectionList>"), Token("RPARENTHESIS", ")")]
    ],

    # <ParamSectionList> ::= <ParamSection> <ParamSectionListPrime>
    NonTerminal("<ParamSectionList>"): [
        [NonTerminal("<ParamSection>"), NonTerminal("<ParamSectionListPrime>")]
    ],

    # <ParamSectionListPrime> ::= SEMICOLON(); <ParamSection> <ParamSectionListPrime> | <Epsilon>
    NonTerminal("<ParamSectionListPrime>"): [
        [Token("SEMICOLON", ";"), NonTerminal("<ParamSection>"),
        NonTerminal("<ParamSectionListPrime>"),
        [<Epsilon>]
    ],

    # <ParamSection> ::= <VarKeywordOpt> <IdentifierList> COLON(:) <SimpleType>
    NonTerminal("<ParamSection>"): [
        [NonTerminal("<VarKeywordOpt>"), NonTerminal("<IdentifierList>"), Token("COLON", ":"), NonTerminal("<SimpleType>")]
    ],

    # <VarKeywordOpt> ::= KEYWORD(variabel) | <Epsilon>
    NonTerminal("<VarKeywordOpt>"): [
        [Token("KEYWORD", "variabel")],
        [<Epsilon>]
    ],

    # --- Bagian Statement ---

    # <StatementPart> ::= <CompoundStatement>
    NonTerminal("<StatementPart>"): [
        [NonTerminal("<CompoundStatement>")]
    ],

    # <CompoundStatement> ::= KEYWORD(mulai) <StatementList> KEYWORD(selesai)
    NonTerminal("<CompoundStatement>"): [
        [Token("KEYWORD", "mulai"), NonTerminal("<StatementList>"), Token("KEYWORD", "selesai")]
    ],

    # <StatementList> ::= <Statement> <StatementListPrime> | <Epsilon>
    NonTerminal("<StatementList>"): [
        [NonTerminal("<Statement>"), NonTerminal("<StatementListPrime>")],
        [<Epsilon>] # Mengizinkan blok mulai...selesai kosong
    ],

    # <StatementListPrime> ::= SEMICOLON(); <Statement> <StatementListPrime> | <Epsilon>
    NonTerminal("<StatementListPrime>"): [
        [Token("SEMICOLON", ";"), NonTerminal("<Statement>"), NonTerminal("<StatementListPrime>")],
        [<Epsilon>]
    ],

    # <Statement> ::= <AssignmentStatement> | <ProcedureCall> | ... | <Epsilon>
    NonTerminal("<Statement>"): [
        [NonTerminal("<AssignmentStatement>")],
        [NonTerminal("<ProcedureCall>")],
        [NonTerminal("<CompoundStatement>")],
        [NonTerminal("<IfStatement>")],
        [NonTerminal("<WhileStatement>")],
        [NonTerminal("<ForStatement>")],
        [NonTerminal("<RepeatStatement>")],
        [NonTerminal("<CaseStatement>")],
        [<Epsilon>] # Statement kosong
    ],

    # --- Jenis-jenis Statement ---

```

```

# <AssignmentStatement> ::= IDENTIFIER <VariableTail> ASSIGN_OPERATOR(:=) <Expression>
# (Telah di-left-factor, <Variable> di-inline-kan)
NonTerminal("<AssignmentStatement>"): [
    [TokenType("IDENTIFIER"), NonTerminal("<VariableTail>"), Token("ASSIGN_OPERATOR", ":="),
NonTerminal("<Expression>")]
],

# <ProcedureCall> ::= IDENTIFIER <ParameterListOpt>
NonTerminal("<ProcedureCall>"): [
    [TokenType("IDENTIFIER"), NonTerminal("<ParameterListOpt>")]
],

# <IfStatement> ::= KEYWORD(jika) <Expression> KEYWORD(maka) <Statement> <ElsePart>
NonTerminal("<IfStatement>"): [
    [Token("KEYWORD", "jika"), NonTerminal("<Expression>"), Token("KEYWORD", "maka"),
NonTerminal("<Statement>"), NonTerminal("<ElsePart>")]
],

# <ElsePart> ::= KEYWORD(selain-itu) <Statement> | <Epsilon>
NonTerminal("<ElsePart>"): [
    [Token("KEYWORD", "selain-itu"), NonTerminal("<Statement>")],
    [Epsilon()]
],

# <WhileStatement> ::= KEYWORD(selama) <Expression> KEYWORD(lakukan) <Statement>
NonTerminal("<WhileStatement>"): [
    [Token("KEYWORD", "selama"), NonTerminal("<Expression>"), Token("KEYWORD", "lakukan"),
NonTerminal("<Statement>")]
],

# <RepeatStatement> ::= KEYWORD(ulangi) <StatementList> KEYWORD(sampai) <Expression>
NonTerminal("<RepeatStatement>"): [
    [Token("KEYWORD", "ulangi"), NonTerminal("<StatementList>"), Token("KEYWORD", "sampai"),
NonTerminal("<Expression>")]
],

# <ForStatement> ::= KEYWORD(until) IDENTIFIER ASSIGN_OPERATOR(:=) <Expression> <ForDirection>
<Expression> KEYWORD(lakukan) <Statement>
NonTerminal("<ForStatement>"): [
    [Token("KEYWORD", "until"), TokenType("IDENTIFIER"), Token("ASSIGN_OPERATOR", "="),
NonTerminal("<Expression>"), NonTerminal("<ForDirection>"), NonTerminal("<Expression>"), Token("KEYWORD",
"lakukan"), NonTerminal("<Statement>")]
],

# <ForDirection> ::= KEYWORD(ke) | KEYWORD(turun-ke)
NonTerminal("<ForDirection>"): [
    [Token("KEYWORD", "ke")],
    [Token("KEYWORD", "turun-ke")]
],

# <CaseStatement> ::= KEYWORD(kasus) <Expression> KEYWORD(dari) <CaseList> <CaseEndOpt>
KEYWORD(selesai)
NonTerminal("<CaseStatement>"): [
    [Token("KEYWORD", "kasus"), NonTerminal("<Expression>"), Token("KEYWORD", "dari"),
NonTerminal("<CaseList>"), NonTerminal("<CaseEndOpt>"), Token("KEYWORD", "selesai")]
],

# <CaseList> ::= <CaseElement> <CaseListPrime>
NonTerminal("<CaseList>"): [
    [NonTerminal("<CaseElement>"), NonTerminal("<CaseListPrime>")]
],

# <CaseListPrime> ::= SEMICOLON(); <CaseElement> <CaseListPrime> | <Epsilon>
NonTerminal("<CaseListPrime>"): [
    [Token("SEMICOLON", ";"), NonTerminal("<CaseElement>"), NonTerminal("<CaseListPrime>")],
    [Epsilon()]
],

# <CaseEndOpt> ::= SEMICOLON(); | <Epsilon> (Untuk ; sebelum 'selesai')
NonTerminal("<CaseEndOpt>"): [
    [Token("SEMICOLON", ";")],
    [Epsilon()]
],

# <CaseElement> ::= <Constant> COLON(:) <Statement>
NonTerminal("<CaseElement>"): [
    [NonTerminal("<Constant>"), Token("COLON", ":"), NonTerminal("<Statement>")]
],

# --- Parameter Aktual ---

```

```

# <ParameterListOpt> ::= <ParameterList> | <Epsilon>
NonTerminal("<ParameterListOpt>"): [
    [NonTerminal("<ParameterList>")],
    [Epsilon()]
],

# <ParameterList> ::= LPARENTHESIS() <ExpressionList> RPARENTHESIS()
NonTerminal("<ParameterList>"): [
    [Token("LPARENTHESIS", "("), NonTerminal("<ExpressionList>"), Token("RPARENTHESIS", ")")]
],

# <ExpressionList> ::= <Expression> <ExpressionListPrime>
NonTerminal("<ExpressionList>"): [
    [NonTerminal("<Expression>"), NonTerminal("<ExpressionListPrime>")]
],

# <ExpressionListPrime> ::= COMMA(,) <Expression> <ExpressionListPrime> | <Epsilon>
NonTerminal("<ExpressionListPrime>"): [
    [Token("COMMA", ","), NonTerminal("<Expression>"), NonTerminal("<ExpressionListPrime>")],
    [Epsilon()]
],

# --- Ekspresi ---
# <Expression> ::= <SimpleExpression> <ExpressionPrime>
NonTerminal("<Expression>"): [
    [NonTerminal("<SimpleExpression>"), NonTerminal("<ExpressionPrime>")]
],

# <ExpressionPrime> ::= <RelationalOperator> <SimpleExpression> | <Epsilon>
NonTerminal("<ExpressionPrime>"): [
    [NonTerminal("<RelationalOperator>"), NonTerminal("<SimpleExpression>")],
    [Epsilon()]
],

# <SimpleExpression> ::= <SignedTerm> <SimpleExpressionPrime>
NonTerminal("<SimpleExpression>"): [
    [NonTerminal("<SignedTerm>"), NonTerminal("<SimpleExpressionPrime>")]
],

# <SignedTerm> ::= <SignOpt> <Term>
NonTerminal("<SignedTerm>"): [
    [NonTerminal("<SignOpt>"), NonTerminal("<Term>")]
],

# <SimpleExpressionPrime> ::= <AdditiveOperator> <Term> <SimpleExpressionPrime> | <Epsilon>
NonTerminal("<SimpleExpressionPrime>"): [
    [NonTerminal("<AdditiveOperator>"), NonTerminal("<Term>"),
     NonTerminal("<SimpleExpressionPrime>")],
    [Epsilon()]
],

# <Term> ::= <Factor> <TermPrime>
NonTerminal("<Term>"): [
    [NonTerminal("<Factor>"), NonTerminal("<TermPrime>")]
],

# <TermPrime> ::= <MultiplicativeOperator> <Factor> <TermPrime> | <Epsilon>
NonTerminal("<TermPrime>"): [
    [NonTerminal("<MultiplicativeOperator>"), NonTerminal("<Factor>"), NonTerminal("<TermPrime>")],
    [Epsilon()]
],

# --- Factor & Variable (Sudah di-Left-Factor) ---

# <Factor> ::= IDENTIFIER <FactorTail> | <Constant> | ( <Expression> ) | 'tidak' <Factor>
NonTerminal("<Factor>"): [
    [TokenType("IDENTIFIER"), NonTerminal("<FactorTail>")], # Variabel ATAU Panggilan Fungsi
    [NonTerminal("<Constant>")],
    [Token("LPARENTHESIS", "("), NonTerminal("<Expression>"), Token("RPARENTHESIS", ")")],
    [Token("LOGICAL_OPERATOR", "tidak"), NonTerminal("<Factor>")]
],

# <FactorTail> ::= <ParameterList> | <VariableTail>
# Memisahkan antara Panggilan Fungsi (butuh parameter list) vs Variabel
NonTerminal("<FactorTail>"): [
    [NonTerminal("<ParameterList>")], # Coba cocokkan sbg Panggilan Fungsi dulu
    [NonTerminal("<VariableTail>")] # Jika gagal, anggap sbg Variabel (bisa Epsilon)
],

```

```

# <VariableTail> ::= LBRACKET ... <VariableTail> | DOT ... <VariableTail> | <Epsilon>
# (Mendefinisikan akses array/record secara rekursif)
NonTerminal("<VariableTail>": [
    [Token("LBRACKET", "["), NonTerminal("<Expression>"), Token("RBRACKET", "]")],
    [Token("DOT", ".")], TokenType("IDENTIFIER"), NonTerminal("<VariableTail>")],
    [Epsilon()]
],)

# --- Kategori Operator ---

# <RelationalOperator> ::= = | <> | < | <= | > | >=
NonTerminal("<RelationalOperator>": [
    [Token("RELATIONAL_OPERATOR", "=")],
    [Token("RELATIONAL_OPERATOR", "<>")],
    [Token("RELATIONAL_OPERATOR", "<")],
    [Token("RELATIONAL_OPERATOR", "<=")],
    [Token("RELATIONAL_OPERATOR", ">")],
    [Token("RELATIONAL_OPERATOR", ">=")]
],)

# <AdditiveOperator> ::= + | - | 'atau'
NonTerminal("<AdditiveOperator>": [
    [Token("ARITHMETIC_OPERATOR", "+")],
    [Token("ARITHMETIC_OPERATOR", "-")],
    [Token("LOGICAL_OPERATOR", "atau")]
],)

# <MultiplicativeOperator> ::= * | / | 'bagi' | 'mod' | 'dan'
NonTerminal("<MultiplicativeOperator>": [
    [Token("ARITHMETIC_OPERATOR", "*")],
    [Token("ARITHMETIC_OPERATOR", "/")],
    [Token("ARITHMETIC_OPERATOR", "bagi")],
    [Token("ARITHMETIC_OPERATOR", "mod")],
    [Token("LOGICAL_OPERATOR", "dan")]
],)

return rules
}

```

5. syntax.py: SyntaxAnalyzer

```

class SyntaxAnalyzer():
    """Kelas untuk SyntaxAnalyzer"""
    cfg: CFG

    def __init__(self):
        self.cfg = CFG()
        self.setupProductionRules()

    def setupProductionRules(self):
        self.cfg.addRules(getAllProductionRules())

    def parse(self, tokens:List[Token]) -> Node|SyntaxError:
        parse_tree = self.cfg.parseToken(tokens)

        if parse_tree is not None:
            # Parsing berhasil, TAPI kita harus cek apakah semua token terpakai.
            final_token = self.cfg.currentToken()
            if final_token.token_type == "EOF":
                return parse_tree # success
            else:
                # Parsing selesai tapi masih ada sisa token
                raise SyntaxError(message=f"\n\tUnexpected token {final_token}", line=final_token.line,
column=final_token.column)
        else:
            # Parsing Gagal (parser mengembalikan None)
            error_info = self.cfg.max_error_info

            if error_info['max_id'] != -1 and error_info['found']:
                found = error_info['found']
                # Format 'expected' set menjadi string yang rapi
                expected_list = [str(e) for e in error_info['expected']]
                if len(expected_list) > 1:

```

```

        expected_str = "one of possible tokens: " + ", ".join(expected_list)
    elif expected_list:
        expected_str = expected_list[0]
    else:
        raise SyntaxError(message="Something went wrong", line=found.line, column=found.column)
        raise SyntaxError(message=f"\n\tUnexpected token {found}, \n\tExpected {expected_str}",
line=found.line, column=found.column)

    else:
        # Fallback jika tidak ada info error (alasan lain)
        raise SyntaxError(message="Something went wrong")

```

6. syntax.py: SyntaxError

```

class SyntaxError(Exception):
    """Custom exception untuk error sintaks."""
    def __init__(self, message:str, line:int=None, column:int=None) -> None:
        if line and column:
            super().__init__(f"Syntax Error on line {line}, column {column}: {message}")
        else:
            super().__init__(f"Syntax Error: {message}")
        self.message = message
        self.line = line
        self.column = column

```

7. syntax.py: Main Program

```

def main():
    """
    Driver utama untuk parser.
    Mengambil 1 argumen: path ke file source code .pas
    """

    # --- 1. Validasi Argumen Input ---
    if len(sys.argv) != 2:
        print("Usage: python syntax.py <source_file_path.pas>", file=sys.stderr)
        sys.exit(1)

    source_file_path = sys.argv[1]
    if not source_file_path.lower().endswith('.pas'):
        print(f"Input Error: Source file harus berekstensi .pas. Diberikan: '{source_file_path}'",
file=sys.stderr)
        sys.exit(1)

    # Tentukan path ke dfa.json (diasumsikan berada di direktori yang sama)
    script_dir = os.path.dirname(os.path.abspath(__file__))
    dfa_path = os.path.join(script_dir, "dfa.json")
    if not os.path.exists(dfa_path):
        print(f"Fatal Error: 'dfa.json' tidak ditemukan di '{script_dir}'", file=sys.stderr)
        sys.exit(1)

    # --- 2. Baca Source Code ---
    try:
        with open(source_file_path, 'r') as f:
            source_code = f.read()
    except FileNotFoundError:
        print(f"Error: Input file tidak ditemukan di '{source_file_path}'", file=sys.stderr)
        sys.exit(1)

    # --- 3. Jalankan Lexer ---
    lexer = Lexer(dfa_path)
    tokens = []
    try:
        tokens = lexer.tokenize(source_code)
    except LexicalError as e:
        print(str(e), file=sys.stderr)
        sys.exit(1) # Keluar jika ada error leksikal

    # print("\n--- Daftar Token ---")
    # for token in tokens:

```

```
#     print(token)
# print("-----\n")

# --- 4. Jalankan Parser ---
parser = SyntaxAnalyzer()
try:
    print(parser.parse(tokens=tokens))
except SyntaxError as e:
    print(e)
except Exception as e:
    print(f"\nFATAL PARSER ERROR: {e}", file=sys.stderr)
    import traceback
    traceback.print_exc()
    sys.exit(1)

if __name__ == "__main__":
    main()
```

BAB III

PENGUJIAN

3.1 Rencana Pengujian

Rencana pengujian ini bertujuan untuk memvalidasi fungsionalitas, ketahanan (robustness), dan kebenaran dari *syntax analyzer* (parser) yang telah diimplementasikan. Pengujian berfokus pada dua aspek utama: kemampuan parser untuk menerima sintaks yang valid (kasus uji positif) dan kemampuannya untuk mendeteksi serta error handling sintaksis yang tidak valid (kasus uji negatif).

Ruang lingkup pengujian mencakup:

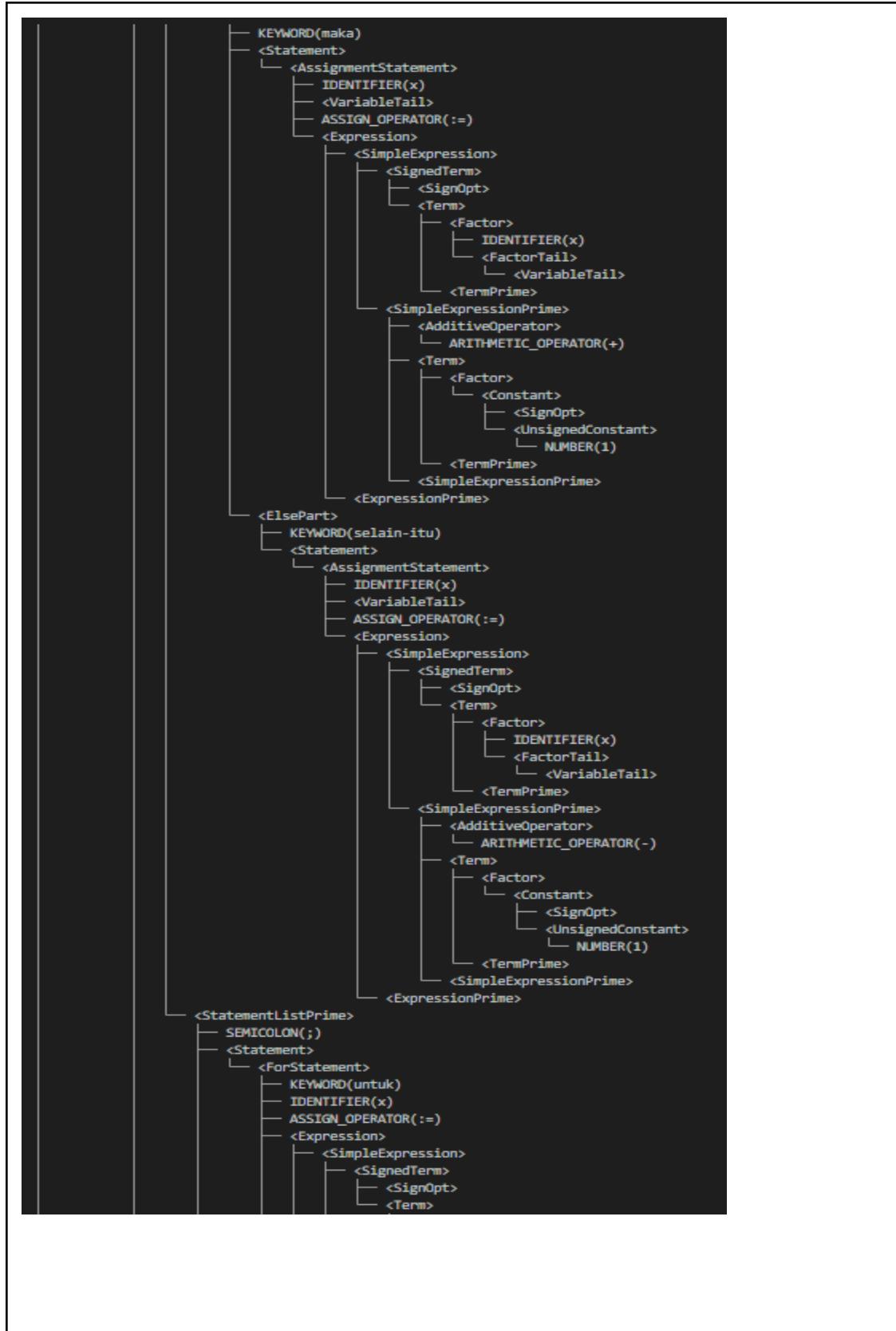
1. Struktur Program Fundamental: Memvalidasi pengenalan *<ProgramHeader>*, *<Block>*, *<DeclarationPart>*, dan *<CompoundStatement>*.
2. Kasus Uji Integrasi Lexer-Parser: Memastikan *parser* dapat menangani kasus-kasus khusus yang dihasilkan oleh *lexer*, seperti *keyword* ber-tanda hubung (selain-itu, turun-ke) dan membedakannya dari operator aritmatika (-).
3. Validasi Grammar Komprehensif: Menguji semua jenis deklarasi (konstanta, tipe, larik, prosedur, fungsi) dan semua jenis *statement* (jika, selama, for).
4. Ekspresi Kompleks: Memvalidasi hierarki dan presedensi operator (aritmatika, relasional, logika) dan penggunaan tanda kurung.
5. Error Handling: Memastikan *parser* dapat berhenti dengan benar ketika menemukan kesalahan sintaksis dan memberikan pesan yang informatif

3.2 Test Case

3.2.1 Test Case 1: Hyphenated

<pre>test_hyphenated.pas program TestHyphenated; variabel x: integer; mulai jika x > 0 maka x := x + 1 selain-itu x := x - 1; untuk x := 10 turun-ke 1 lakukan x := x selesai.</pre>
<p>Output</p>







3.2.2 Test Case 2: Minus Edge Cases

test_minus_edge_cases.pas

```

program TestMinus;
variabel a, b, selain, itu, turun: integer;
mulai
  a := 10 - 5;
  b := selain - itu;
  a := turun;

  jika a > 0 maka
  
```

```

    a := 1
selain-itu
    a := 0;

untuk b := 10 turun-ke 1 lakukan
    b := b - 1
selesai.

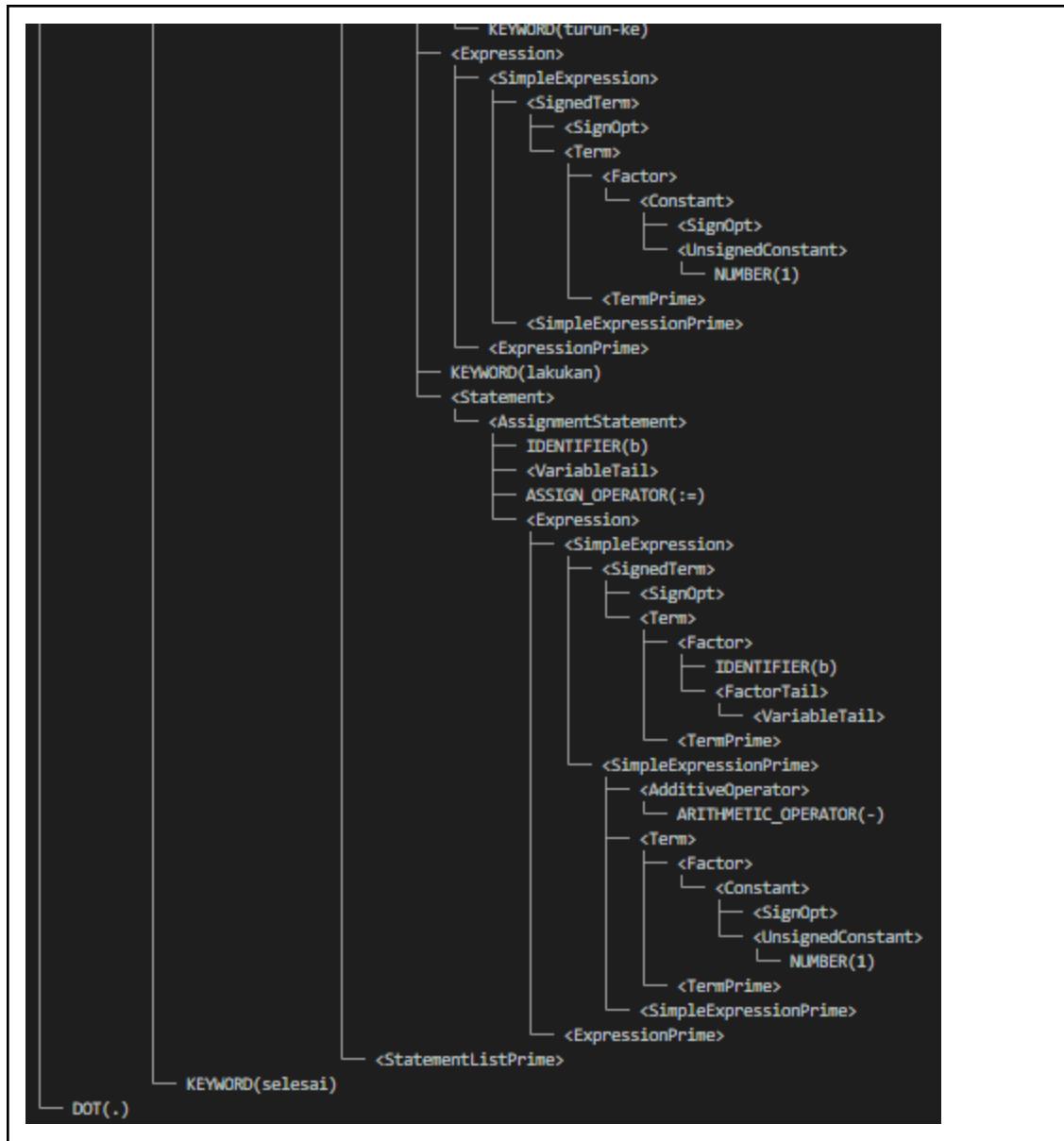
```

Output









3.2.3 Test Case 3: Error Checking

```
test_error_checking.pas
```

```
program TestErrorChecking;
variabel x, y: integer
mulai
  x := 10;
  y = x + 5;
  writeln(x, y)
selesai.
```

Output

Syntax Error on line 3, column 6:
Unexpected token KEYWORD(mulai),
Expected SEMICOLON(;)

3.2.4 Test Case 4: Test Declaration

test_declaration.pas

```
program TestDeclarations;
konstanta
  PI = 3.14;
  MAX = 100;
tipe
  TNumeros = larik [1..MAX] dari integer;
variabel
  a, b, c: integer;
  hasil: real;
  flags: boolean;
  data: TNumeros;

fungsi hitung(x: integer): real;
mulai
  hitung := (x * PI) + (x / 2)
selesai;

mulai
  a := 10;
  b := 20;
  c := (a * b) + (100 mod 3) * (5 bagi 2);
  hasil := hitung(c);
  flags := (a > b) atau (tidak (c = 0)) dan (b <> a);
  data[1] := a
selesai.
```

Output



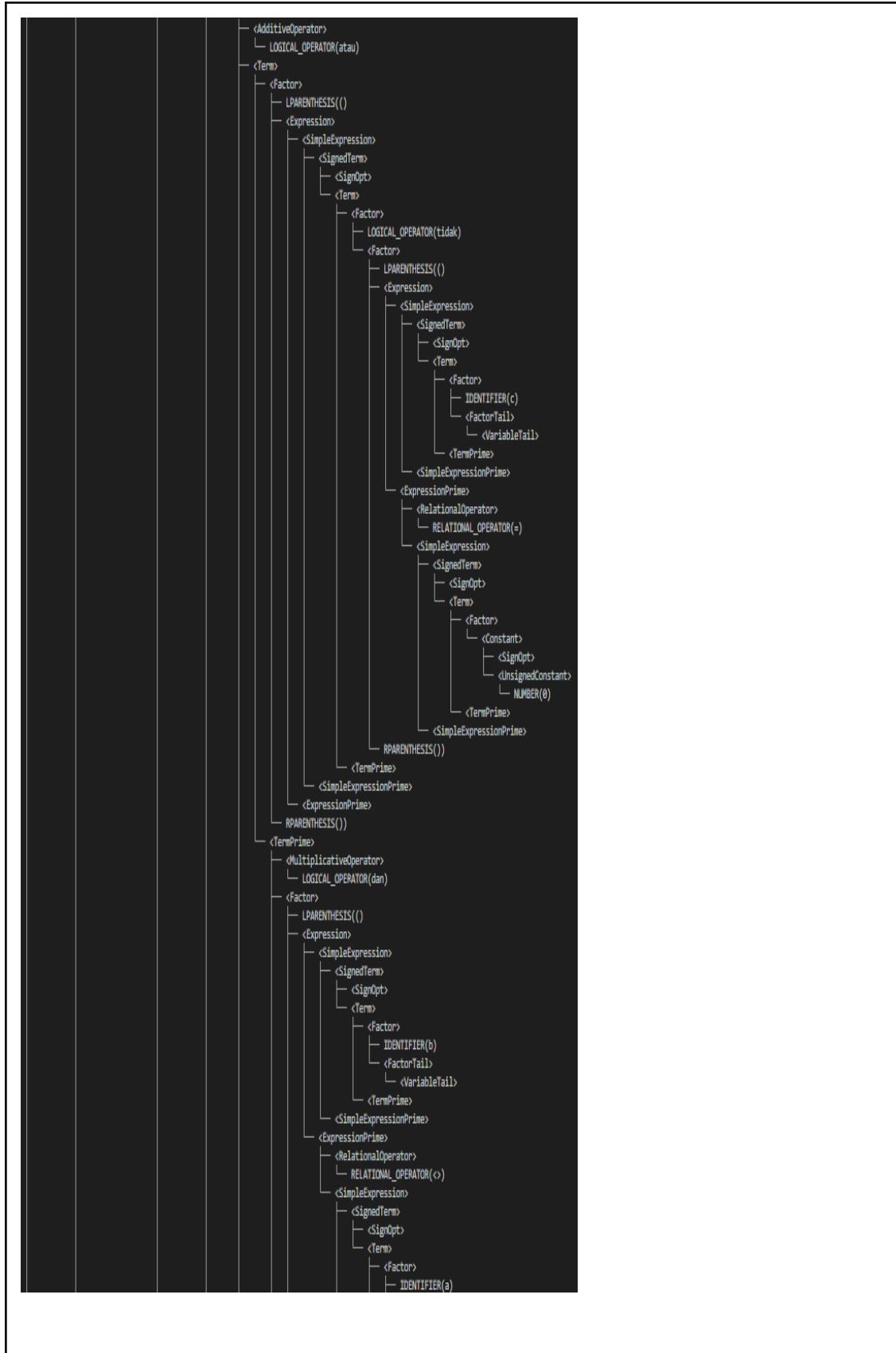














3.2.5 Test Case 5: Test Subprogram

```
test_subprogram.pas
```

```
program TestSubprograms;
variabel
  i: integer;

prosedur cetakAngka(angka: integer; batas: integer);
mulai
  selama angka < batas lakukan
```

```
mulai
    angka := angka + 1;
    jika angka = 5 maka
        angka := 6
    selesai
selesai;

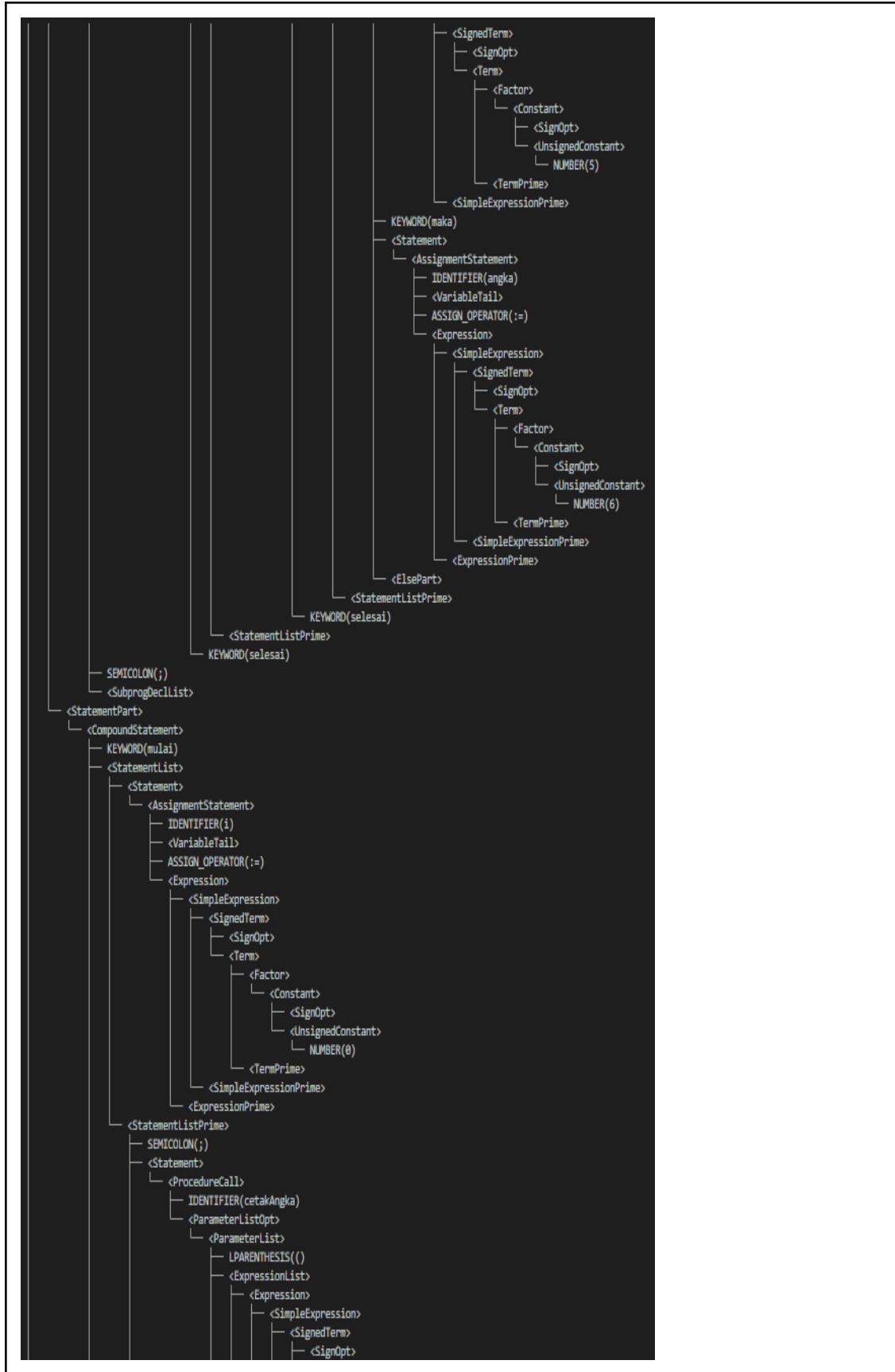
mulai
    i := 0;
    cetakAngka(i, 10);

    untuk i := 1 ke 5 lakukan
        i := i
    selesai.
```

Output









3.3 Ringkasan Hasil Pengujian

Hasil dari pengujian dirangkum dalam tabel berikut :

Kasus Uji Coba	File Input	Tujuan Pengujian	Hasil
Test Case 1	test_hyphenated.pas	Validasi selain-itu & turun-ke	Berhasil. Parser menerima token KEYWORD yang sudah digabung lexer dan parse tree
Test Case 2	test_minus_edge_cases.pas	Ambiguitas operator -	Berhasil. Parser berhasil membedakan selain - itu (3 token) dari selain-itu (1 token). Parse tree lengkap
Test Case 3	test_error_checking.pas	Error handling	Berhasil. Program berhenti. Parser mendeteksi KEYWORD(mulai) saat seharusnya SEMICOLON(;)
Test Case 4	test_declaration.pas	Uji konstanta, tipe, larik, fungsi, & ekspresi kompleks.	Berhasil. Parser berhasil mem-parsing semua blok deklarasi (konstanta, tipe, variabel, fungsi) dan ekspresi kompleks
Test Case 5	test_subprogram.pas	Uji prosedur, selama, for...ke, & blok bersarang.	Berhasil. Parser berhasil mem-parsing deklarasi dan pemanggilan prosedur, loop selama, loop untuk...ke, dan blok mulai...selesai di dalam jika.

Secara keseluruhan, parser yang diimplementasikan telah berhasil melewati semua pengujian yang disiapkan. Parser terbukti mampu menangani sintaks Pascal-S yang valid, termasuk struktur program, semua jenis deklarasi, semua statement, dan ekspresi kompleks.

Selain itu, parser juga terbukti dalam mendeteksi dan melaporkan kesalahan sintaksis secara akurat, sesuai dengan kriteria.

BAB IV

KESIMPULAN DAN SARAN

4.1 Kesimpulan

Dalam penggerjaan pada tugas besar milestone 2 ini, telah berhasil dikembangkan implementasi parser atau Syntax Analyzer dengan pendekatan bentuk teori dari Context-Free-Grammar (CFG) dan melakukan optimalisasi top-down parsing. Untuk detailnya sebagai berikut.

1. Validasi Struktur Bahasa Formal

Implementasi ini menunjukkan pembuktian bahwa sintaksis bahasa Pascal-S dapat sepenuhnya direpresentasikan dan dianalisis menggunakan Context-Free-Grammar (CFG). Keberhasilan parser dalam menerima input yang valid dan menolak yang invalid menegaskan bahwa grammar yang dirancang secara formal mendefinisikan himpunan string (program) yang benar dalam bahasa tersebut.

2. Efektivitas Metode Top-Down Parsing

Penggunaan metode Recursive Descent Parsing (turunan dari strategi Top-Down Parsing) terbukti efektif dan sesuai untuk grammar Pascal-S yang telah dirancang. Hal ini secara implisit menunjukkan bahwa

- Grammar telah dikonversi menjadi bentuk yang bebas dari Left-Recursion dan Common Prefix yang menjadikannya cocok untuk parser.
- Mekanisme prediktif (berdasarkan simbol Lookahead tunggal) yang merupakan ciri khas berhasil digunakan untuk mengarahkan fungsi-fungsi dalam membangun parse-tree program

3. Keberhasilan Tahap Kritis Compiler

Secara fungsional, tahap Syntax Analysis ini berhasil berfungsi sebagai jembatan yang kokoh antara Lexical Analysis (pembentukan token) dan Semantic Analysis. Parser telah suskses menganalisis struktur hierarkis kode program, mengubah stream token menjadi sebuah representasi terstruktur yang siap diproses pada fase kompilasi selanjutnya.

4.2 Saran

Untuk pengembangan lebih lanjut yang menguatkan pemahaman teoritis dan meningkatkan kualitas *compiler* berupa

1. Eksplorasi Metode Bottom-Up Parsing

Saran: Lakukan studi komparatif dengan mengimplementasi parser yang sama menggunakan pendekatan Bottom-Up (misalnya, SLR(1) atau LALR(1)). Hal ini akan memberikan wawasan mendalam mengenai perbedaan kompleksitas, overhead, dan kemampuan error handling antara grammar yang dapat diterima oleh kelas parser LL(1) dan kelas parser LR(k). Metode LR umumnya dikenal lebih kuat (dapat

menerima lebih banyak grammar) dan lebih baik dalam mendeteksi kesalahan lebih awal.

2. Peningkatan Error Recovery Berbasis Sets

Saran: Kembangkan mekanisme error recovery yang lebih canggih, tidak hanya sekadar deteksi dan penghentikan (panic mode), tetapi dengan strategi berbasis First dan Follow Sets yang telah dihitung untuk grammar. Pemanfaatan First dan Follow-Sets secara sistematis dapat memandu parser untuk melewati satu atau lebih token yang salah dan “melompat” ke titik sinkronisasi yang valid berikutnya, memungkinkan parser melaporkan lebih dari satu kesalahan sintaksis dalam satu kali run kompilasi.

3. Transisi ke Abstract Syntax Tree (AST)

Saran: Modifikasi parser untuk tidak hanya mem-parsing tetapi juga secara eksplisit menghasilkan Abstract Syntax Tree (AST) sebagai output resmi. AST adalah representasi struktural yang lebih ringkas dan esensial daripada Parse Tree karena menghilangkan token dan non-terminal yang tidak relevan untuk analisis semantik. AST adalah format input yang paling efisien untuk fase compiler berikutnya, yaitu Semantic Analysis dan mempermudah implementasi type checking serta intermediate code generation.

LAMPIRANLink Repository: <https://github.com/ivant8k/MOE-Tubes-IF2224/tree/main>

Releases:

- v0.1.1-Milestone 1: Lexical Analysis (19 Oktober 2025)
- v0.2.1-Milestone 2: Syntax Analysis (16 November 2025)

Pembagian Tugas

Nama	NIM	Pembagian Tugas	Persentase
Ivant Samuel Silaban	13523129	<ul style="list-style-type: none"> ● Merancang grammar dan struktur parser. ● Memperbaiki bug yang ditemukan selama development dan mengatasi error pada fungsi-fungsi parsing. ● Membuat test case. ● Menulis Landasan Teori (BAB I). 	22%
Rafa Abdussalam Danadyaksa	13523133	<ul style="list-style-type: none"> ● Merancang grammar dan struktur parser. ● Mengimplementasi Non terminal dan production rule. ● Membuat test case. ● Melakukan pengujian dan dokumentasi Hasil Pengujian (BAB III). 	19%
Muhamad Nazih Najmudin	13523144	<ul style="list-style-type: none"> ● Merancang grammar dan struktur parser. ● Mengimplementasi Parse Tree dan fungsi-fungsi parsing. ● Menentukan strategi error handling. ● Menulis Perancangan dan Implementasi (BAB II). 	23%
Anas Ghazi Al Gifari	13523159	<ul style="list-style-type: none"> ● Merancang grammar dan struktur parser. ● Membuat test case. ● Menulis Landasan Teori 	19%

		<p>(BAB I), Lampiran, dan Referensi.</p> <ul style="list-style-type: none"> • Menyusun kerangka konseptual laporan Milestone 2 secara menyeluruh. 	
Muhammad Rizain Firdaus	13523164	<ul style="list-style-type: none"> • Merancang grammar dan struktur parser. • Menulis Kesimpulan dan Saran (BAB IV). 	17%

Grammar yang digunakan

```

<Program> → <ProgramHeader> <Block> .

<ProgramHeader> → program IDENTIFIER ;

<Block> → <DeclarationPart> <StatementPart>

<DeclarationPart> → <ConstDeclOpt> <TypeDeclOpt> <VarDeclOpt>

<ConstDeclOpt> → konstanta <ConstList> | ε

<ConstList> → IDENTIFIER = <Constant> ; <ConstList> | ε

<TypeDeclOpt> → tipe <TypeList> | ε

<TypeList> → IDENTIFIER = <Type> ; <TypeList> | ε

<VarDeclOpt> → variabel <VarDeclList> | ε

<VarDeclList> → <VarDeclaration> ; <VarDeclList> | ε

<VarDeclaration> → <IdentifierList> : <Type>

<IdentifierList> → IDENTIFIER <IdentifierListPrime>

<IdentifierListPrime> → , IDENTIFIER <IdentifierListPrime> | ε

<Type> → <SimpleType> | <ArrayType> | <RecordType>

<SimpleType> → integer | real | boolean | char | IDENTIFIER

<ArrayType> → larik [ <Range> ] dari <Type>

```

```

<RecordType> → rekaman <VarDeclOpt> selesai

<Range> → <Constant> .. <Constant>

<Constant> → <SignOpt> <UnsignedConstant> | STRING_LITERAL |
CHAR_LITERAL | true | false

<UnsignedConstant> → NUMBER | IDENTIFIER

<SignOpt> → <Sign> | ε

<Sign> → + | -

<SubprogDeclList> → <SubprogramDeclaration> ; <SubprogDeclList> | ε

<SubprogramDeclaration> → <ProcedureDeclaration> | <FunctionDeclaration>

<ProcedureDeclaration> → prosedur IDENTIFIER <FormalParamOpt> ; <Block>

<FunctionDeclaration> → fungsi IDENTIFIER <FormalParamOpt> : <SimpleType> ;
<Block>

<FormalParamOpt> → <FormalParameterList> | ε

<FormalParameterList> → ( <ParamSectionList> )

<ParamSectionList> → <ParamSection> <ParamSectionListPrime>

<ParamSectionListPrime> → ; <ParamSection> <ParamSectionListPrime> | ε

<ParamSection> → <VarKeywordOpt> <IdentifierList> : <SimpleType>

<VarKeywordOpt> → variabel | ε

<StatementPart> → <CompoundStatement>

<CompoundStatement> → mulai <StatementList> selesai

<StatementList> → <Statement> <StatementListPrime> | ε

<StatementListPrime> → ; <Statement> <StatementListPrime> | ε

<Statement> → <AssignmentStatement> | <ProcedureCall> | <CompoundStatement> |
<IfStatement> | <WhileStatement> | <ForStatement> | <RepeatStatement> |
<CaseStatement> | ε

```

```
<AssignmentStatement> → IDENTIFIER <VariableTail> := <Expression>  
<ProcedureCall> → IDENTIFIER <ParameterListOpt>  
<IfStatement> → jika <Expression> maka <Statement> <ElsePart>  
<ElsePart> → selain-itu <Statement> | ε  
<WhileStatement> → selama <Expression> lakukan <Statement>  
<RepeatStatement> → ulangi <StatementList> sampai <Expression>  
<ForStatement> → untuk IDENTIFIER := <Expression> <ForDirection> <Expression>  
lakukan <Statement>  
<ForDirection> → ke | turun-ke  
<CaseStatement> → kasus <Expression> dari <CaseList> <CaseEndOpt> selesai  
<CaseList> → <CaseElement> <CaseListPrime>  
<CaseListPrime> → ; <CaseElement> <CaseListPrime> | ε  
<CaseEndOpt> → ; | ε  
<CaseElement> → <Constant> : <Statement>  
<ParameterListOpt> → <ParameterList> | ε  
<ParameterList> → ( <ExpressionList> )  
<ExpressionList> → <Expression> <ExpressionListPrime>  
<ExpressionListPrime> → , <Expression> <ExpressionListPrime> | ε  
<Expression> → <SimpleExpression> <ExpressionPrime>  
<ExpressionPrime> → <RelationalOperator> <SimpleExpression> | ε  
<SimpleExpression> → <SignedTerm> <SimpleExpressionPrime>  
<SignedTerm> → <SignOpt> <Term>  
<SimpleExpressionPrime> → <AdditiveOperator> <Term> <SimpleExpressionPrime> |  
ε  
<Term> → <Factor> <TermPrime>
```

```
<TermPrime> → <MultiplicativeOperator> <Factor> <TermPrime> | ε
<Factor> → IDENTIFIER <FactorTail> | <Constant> | ( <Expression> ) | tidak <Factor>
<FactorTail> → <ParameterList> | <VariableTail>
<VariableTail> → [ <Expression> ] <VariableTail> | . IDENTIFIER <VariableTail> | ε
<RelationalOperator> → = | <> | < | <= | > | >=
<AdditiveOperator> → + | - | atau
<MultiplicativeOperator> → * | / | bagi | mod | dan
```

REFERENSI

- [1] J. E. Hopcroft, R. Motwani, dan J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, edisi ke-3. Boston: Pearson Education, 2006