

LAPORAN TUGAS BESAR
IF2224 TEORI BAHASA FORMAL DAN OTOMATA
MILESTONE 2 - Syntax Analysis



Disusun oleh
Kelompok 9 - TMP

Bertha Soliany Frandi	13523026
Brian Albar Hadian	13523048
Muhammad Izzat Jundy	13523092
Michael Alexander Angkawijaya	13523102

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

DAFTAR ISI

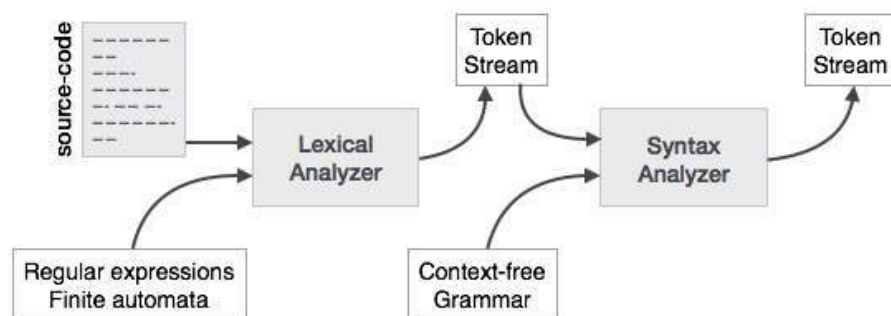
DAFTAR ISI.....	2
BAB I.....	3
LANDASAN TEORI.....	3
1.1 Syntax Analyzer.....	3
1.2 Context Free Grammar.....	3
1.3 Parse Tree.....	4
1.4 Algoritma Recursive Descent.....	4
1.5 Error Handling pada Parser.....	5
BAB II.....	6
PERANCANGAN & IMPLEMENTASI.....	6
2.1 Perancangan.....	6
2.2 Implementasi.....	6
2.2.1 Program.....	7
BAB III.....	11
PENGUJIAN.....	11
3.1 Kasus 1.....	11
3.2 Kasus 2.....	12
3.3 Kasus 3.....	13
3.4 Kasus 4.....	13
3.5 Kasus 5.....	14
BAB IV.....	15
KESIMPULAN DAN SARAN.....	15
4.1 Kesimpulan.....	15
4.2 Saran.....	15
REFERENSI.....	16
LAMPIRAN.....	17

BAB I

LANDASAN TEORI

1.1 Syntax Analyzer

Syntax analyzer atau *parser* merupakan tahap kedua dalam proses kompilasi setelah *lexical analysis*. Tugas utamanya adalah memeriksa apakah rangkaian token yang dihasilkan oleh *lexer* sesuai dengan aturan tata bahasa (grammar) yang didefinisikan untuk bahasa tersebut.



Gambar Tahapan Syntax Analyzer

(Sumber: https://www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.htm)

Parser mengidentifikasi struktur sintaksis dari program sumber, memeriksa kesalahan sintaks, serta menghasilkan representasi struktur—biasanya berupa *parse tree* atau *abstract syntax tree* (Aho et al., 2007). Jika terjadi kesalahan, *parser* harus mampu memberikan laporan kesalahan yang informatif sekaligus melakukan *error recovery* agar proses kompilasi dapat dilanjutkan.

Secara umum, parser berperan untuk menguji validitas sintaks program, membangun struktur hierarkis sesuai aturan grammar, dan menjadi penghubung antara analisis leksikal dan analisis semantik.

1.2 Context Free Grammar

Untuk mendeskripsikan struktur sintaksis sebuah bahasa pemrograman, digunakan suatu bentuk tata bahasa formal. Bentuk yang paling sering dipakai adalah Context-Free Grammar (CFG).

CFG adalah sebuah 4-tuple $G=(V,T,P,S)$ yang terdiri atas:

V: himpunan non-terminal

T: himpunan terminal (biasanya token)

P: himpunan aturan produksi

S: simbol awal (start symbol)

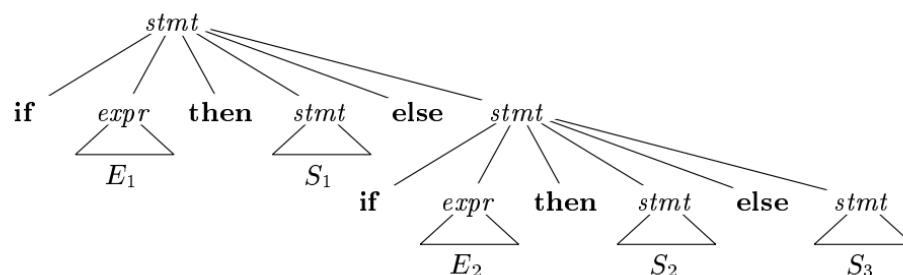
Setiap produksi memiliki bentuk $A \rightarrow \alpha$, di mana $A \in V$ dan $\alpha \in (V \cup T)^*$

CFG digunakan secara luas untuk mendefinisikan sintaks bahasa pemrograman karena kemampuannya menggambarkan struktur rekursif, seperti ekspresi berpasangan kurung, blok program, atau pernyataan bersarang (Tutorialspoint, GfG). Grammar menjadi acuan parser untuk menentukan apakah suatu rangkaian token valid, menentukan bagaimana struktur sintaks disusun, dan memandu proses konstruksi parse tree.

Grammar yang baik harus bebas dari ambiguitas dan telah disesuaikan agar kompatibel dengan teknik parsing tertentu (mis., LL(1), LR(1), dll.)—Aho et al. (2007) menyebutnya sebagai grammar transformations.

1.3 Parse Tree

Parse tree adalah struktur pohon yang menunjukkan bagaimana token-token hasil lexical analysis membentuk sebuah konstruksi program berdasarkan aturan grammar. Dalam parse tree, setiap node non-terminal menggambarkan suatu aturan produksi, sedangkan node terminal berada pada daun dan mewakili token asli dari program. Parse tree memvisualisasikan proses bagaimana simbol awal grammar secara bertahap menurunkan rangkaian token melalui aturan-aturan produksi.



Contoh Gambar Parse Tree untuk *Conditional Statement*
(Sumber: Aho et al., 2007, hlm. 210)

Struktur ini memberikan gambaran lengkap tentang bagaimana sebuah pernyataan atau ekspresi disusun sesuai grammar yang digunakan. Karena itu, parse tree sangat membantu untuk memahami struktur program dan memastikan bahwa urutan token mengikuti aturan sintaks. Parse tree juga sering menjadi dasar pembentukan struktur sintaks lanjutan seperti Abstract Syntax Tree (AST), yang lebih ringkas dan digunakan pada tahap kompilasi berikutnya.

1.4 Algoritma Recursive Descent

Recursive Descent merupakan salah satu teknik parsing yang bersifat top-down, di mana proses pembacaan struktur program dimulai dari simbol awal grammar dan diturunkan secara bertahap mengikuti aturan produksi yang berlaku. Pada metode ini, setiap non-terminal dalam grammar direpresentasikan oleh sebuah fungsi di dalam parser. Fungsi tersebut bertugas memeriksa urutan token dan memastikan bahwa pola yang muncul sesuai dengan aturan produksi non-terminal

tersebut. Jika grammar menyatakan bahwa sebuah non-terminal terdiri dari beberapa simbol lain, maka fungsi yang bersangkutan akan memanggil fungsi-fungsi lain yang mewakili simbol-simbol tersebut.

Cara kerja Recursive Descent cukup langsung dan mudah diikuti. Parser menyimpan token yang sedang dilihat (lookahead token), kemudian mencocokkan token tersebut dengan simbol yang diharapkan sesuai grammar. Jika cocok, parser akan melanjutkan pembacaan token berikutnya. Jika tidak, parser akan melaporkan kesalahan sintaks dengan informasi yang menunjukkan bagian program mana yang tidak sesuai aturan. Karena setiap fungsi hanya bertanggung jawab pada satu non-terminal, struktur parser menjadi jelas, terorganisasi, dan mudah ditelusuri.

Metode ini cocok digunakan ketika grammar berada dalam bentuk yang relatif sederhana dan tidak mengandung ambiguitas yang memerlukan backtracking. Dalam konteks Pascal-S, pendekatan Recursive Descent dapat diterapkan secara langsung karena bentuk grammar-nya memungkinkan pemetaan non-terminal ke fungsi-fungsi parser tanpa penanganan konflik produksi yang rumit. Selain memvalidasi urutan token, parser juga dapat membangun parse tree dengan menambahkan node baru pada setiap pemanggilan fungsi non-terminal, sehingga struktur sintaks dapat direpresentasikan secara lengkap sesuai dengan grammar yang digunakan.

1.5 Error Handling pada Parser

Parser harus mampu mendeteksi dan menangani kesalahan sintaks agar proses kompilasi tidak terhenti sepenuhnya. Aho et al. (2007) menjelaskan beberapa strategi umum untuk error handling, antara lain:

1. Panic-mode recovery: melewati token sampai menemukan titik “aman” seperti ‘;’ atau ‘}’.
2. Phrase-level recovery: menambah atau menghapus token tertentu untuk memulihkan struktur.
3. Error productions: menambahkan aturan produksi khusus untuk menangani kemungkinan kesalahan umum.
4. Global correction: mencoba memodifikasi input seminimal mungkin agar menjadi sintaks yang valid.

BAB II

PERANCANGAN & IMPLEMENTASI

2.1 Perancangan

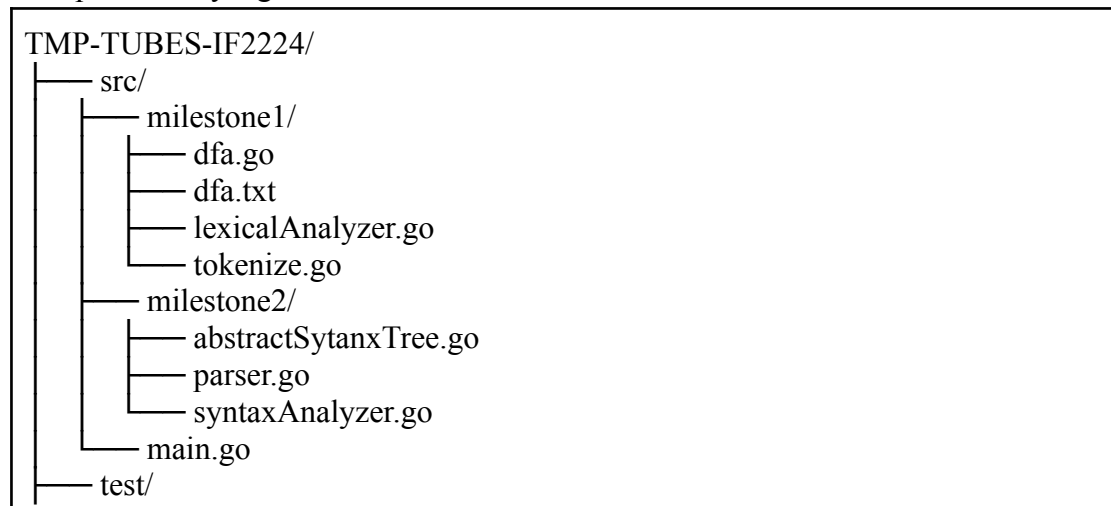
Perancangan untuk menerapkan algoritma *Recursive Descent* dilakukan dengan melakukan dekomposisi pekerjaan menjadi beberapa bagian, yakni membuat kelas Parser yang bertanggung jawab untuk menyimpan token dan melakukan pencocokan terhadap kecocokan komponen tersebut dan implementasi seluruh production rule dalam format fungsi yang dapat memberikan keluaran yang berupa struktur data `AbstractSyntaxTree` dan error. Struktur data `AbstractSyntaxTree` merupakan struktur data yang menyimpan atribut `TerminalValue`, yakni nama dari suatu node apabila merupakan suatu node terminal dari Node Abstract Syntax Tree (AST), `NonTerminalValue`, yakni nama dari suatu node, serta `Children`, yakni larik yang menyimpan seluruh node dari Abstract Syntax Tree yang diperlukan berdasarkan ekspansi node tersebut.

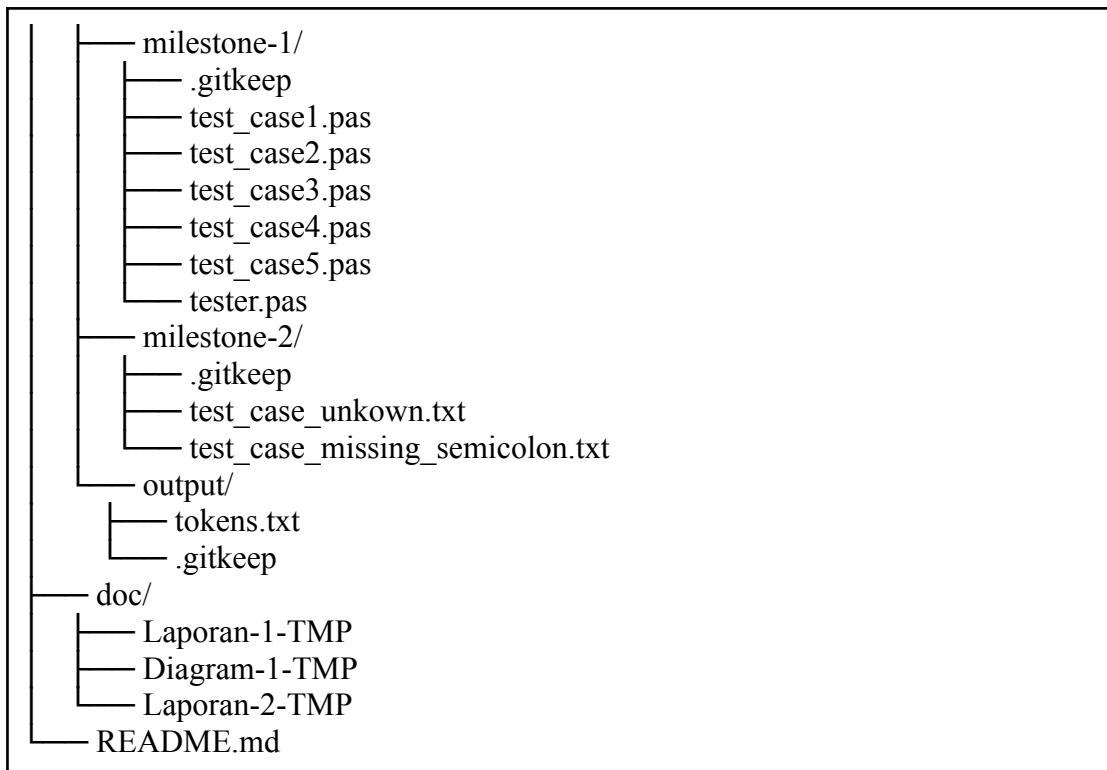
Selanjutnya, diterapkan juga semua fungsi pembantu (*helper*) yang diperlukan untuk melakukan pencocokan *node* terminal, mengatasi error, dan pemindahan variabel *counter*.

2.2 Implementasi

Implementasi dilakukan dengan menggunakan bahasa Go. Penggunaan bahasa Go dimotivasi dari kecepatannya dalam melakukan kompilasi dibandingkan dengan bahasa lainnya dan penggunaan bahasa ini dapat memberikan kemampuan konkurensi yang cenderung sama cepatnya dengan bahasa C, tetapi memiliki *syntax* penulisan yang bersifat *high-level* sehingga dapat mendorong performa *compiler* dan memberikan kesempatan lebih untuk pengembangan selanjutnya.

Berikut adalah struktur folder dari program. Folder `src` menyimpan segala program yang akan dibuat dan folder `test` berisi semua test case. Terdapat pula folder `doc` yang akan berisi laporan ini dalam bentuk pdf dan diagram DFA. File `dfa.txt` merupakan file yang berisi aturan DFA.





2.2.1 Program

Program memiliki alur utama berupa menerima input hasil program Lexical Analyzer berupa list of tokens, kemudian memeriksa kesesuaian urutan token dengan CFG menggunakan algoritma recursive decent, dengan membuat abstract syntax tree . Apabila sesuai, akan dilakukan penambahan node dan apabila node tersebut masih merupakan node non-terminal, akan dilakukan ekspansi dan dilakukan aksi rekursif terhadap node tersebut. Sementara itu, apabila telah mencapai node terminal, akan dilakukan eliminasi dan eksekusi akan berjalan ke node selanjutnya yang belum diekspansi dan/atau dilakukan pencocokan

Berikut adalah penjelasan mengenai fungsi yang terdapat pada program di berbagai file.

Nama Fungsi/Tipe	Penjelasan
func main()	Merupakan titik awal program yang mengatur flow utama program.
type Parser	Variable: Tokens (Array of String), Pos (integer)
func NewParser(Tokens []string)	Membuat Parser baru. Tipe keluaran: *Parser
func eof()	Method dari Parser. Untuk menentukan Parser sudah kehabisan token untuk dibaca. Tipe keluaran: boolean.

func peek()	Method dari Parser. Mendapatkan token terkini berdasarkan posisi Parser terkini bila ada. Tipe keluaran: string.
func next()	Method dari Parser. Untuk menggeser posisi Parser ke posisi selanjutnya bila ada. Tipe keluaran: string (token pada posisi sebelum digeser)
func accept(tok string)	Method dari Parser. Menentukan apakah token didahului oleh token yang sesuai dengan CFG. Tipe keluaran: boolean.
func expect(tok string)	Method dari Parser. Menentukan apakah token didahului oleh token yang sesuai dengan CFG dan mengembalikan error bila tidak. Tipe keluaran: error.
func ParseProgram()	Method dari Parser. Parsing berdasarkan aturan node 1. Tipe keluaran: *AbstractSyntaxTree, error
func ParseProgramHeader()	Method dari Parser. Parsing berdasarkan aturan node 2. Tipe keluaran: *AbstractSyntaxTree, error
func ParseDeclarationPart()	Method dari Parser. Parsing berdasarkan aturan node 3. Tipe keluaran: *AbstractSyntaxTree, error
func ParseConstDeclaration()	Method dari Parser. Parsing berdasarkan aturan node 4. Tipe keluaran: *AbstractSyntaxTree, error
func ParseTypeDeclaration()	Method dari Parser. Parsing berdasarkan aturan node 5. Tipe keluaran: *AbstractSyntaxTree, error
func ParseVarDeclaration()	Method dari Parser. Parsing berdasarkan aturan node 6. Tipe keluaran: *AbstractSyntaxTree, error
func ParseIdentifierList()	Method dari Parser. Parsing berdasarkan aturan node 7. Tipe keluaran: *AbstractSyntaxTree, error
func ParseType()	Method dari Parser. Parsing berdasarkan aturan node 8. Tipe keluaran: *AbstractSyntaxTree, error
func ParseArrayType()	Method dari Parser. Parsing berdasarkan

	aturan node 9. Tipe keluaran: *AbstractSyntaxTree, error
func ParseRange()	Method dari Parser. Parsing berdasarkan aturan node 10. Tipe keluaran: *AbstractSyntaxTree, error
func ParseSubprogramDeclaration()	Method dari Parser. Parsing berdasarkan aturan node 11. Tipe keluaran: *AbstractSyntaxTree, error
func ParseFunctionDeclaration()	Method dari Parser. Parsing berdasarkan aturan node 12. Tipe keluaran: *AbstractSyntaxTree, error
func ParseProcedureDeclaration()	Method dari Parser. Parsing berdasarkan aturan node 13. Tipe keluaran: *AbstractSyntaxTree, error
func ParseParameterGroup()	Method dari Parser. Parsing berdasarkan aturan node 14. Tipe keluaran: *AbstractSyntaxTree, error
func ParseFormalParameterList()	Method dari Parser. Parsing berdasarkan aturan node 15. Tipe keluaran: *AbstractSyntaxTree, error
func ParseCompoundStatement()	Method dari Parser. Parsing berdasarkan aturan node 16. Tipe keluaran: *AbstractSyntaxTree, error
func ParseStatement()	Method dari Parser. Parsing berdasarkan aturan node 17. Tipe keluaran: *AbstractSyntaxTree, error
func ParseStatementList()	Method dari Parser. Parsing berdasarkan aturan node 18. Tipe keluaran: *AbstractSyntaxTree, error
func ParseAssignmentStatement()	Method dari Parser. Parsing berdasarkan aturan node 19. Tipe keluaran: *AbstractSyntaxTree, error
func ParseIfStatement()	Method dari Parser. Parsing berdasarkan aturan node 20. Tipe keluaran: *AbstractSyntaxTree, error
func ParseWhileStatement()	Method dari Parser. Parsing berdasarkan aturan node 21. Tipe keluaran: *AbstractSyntaxTree, error
func ParseForStatement()	Method dari Parser. Parsing berdasarkan

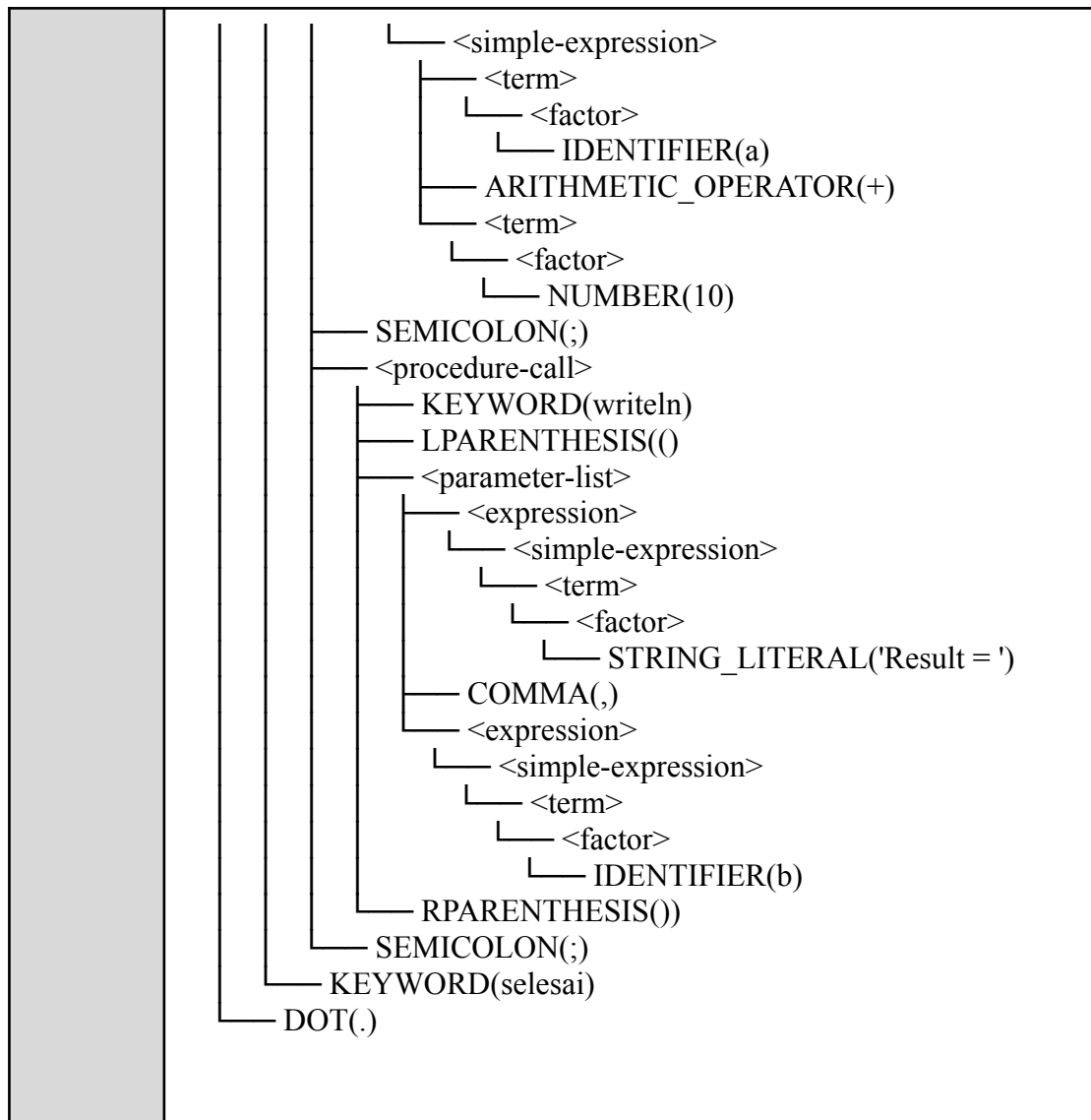
	aturan node 22. Tipe keluaran: *AbstractSyntaxTree, error
func ParseParameterList()	Method dari Parser. Parsing berdasarkan aturan node 23. Tipe keluaran: *AbstractSyntaxTree, error
func ParseExpression()	Method dari Parser. Parsing berdasarkan aturan node 24. Tipe keluaran: *AbstractSyntaxTree, error
func ParseSimpleExpression()	Method dari Parser. Parsing berdasarkan aturan node 25. Tipe keluaran: *AbstractSyntaxTree, error
func ParseTerm()	Method dari Parser. Parsing berdasarkan aturan node 26. Tipe keluaran: *AbstractSyntaxTree, error
func ParseFactor()	Method dari Parser. Parsing berdasarkan aturan node 27. Tipe keluaran: *AbstractSyntaxTree, error
func ParseRelationalOperator()	Method dari Parser. Parsing berdasarkan aturan node 28. Tipe keluaran: *AbstractSyntaxTree, error
func ParseAdditiveOperator()	Method dari Parser. Parsing berdasarkan aturan node 29. Tipe keluaran: *AbstractSyntaxTree, error
func ParseMultiplicativeOperator())	Method dari Parser. Parsing berdasarkan aturan node 30. Tipe keluaran: *AbstractSyntaxTree, error

BAB III

PENGUJIAN

3.1 Kasus 1

Input	<pre> program Hello; variabel a, b: integer; mulai a := 5; b := a + 10; writeln('Result = ', b); selesai.</pre>
Output	<pre> └─ <program> └─ <program-header> ├── KEYWORD(program) ├── IDENTIFIER(Hello) └── SEMICOLON(;) └─ <declaration-part> └─ <var-declaration> ├── KEYWORD(variabel) ├── <identifier-list> │ ├── IDENTIFIER(a) │ ├── COMMA(,) │ └── IDENTIFIER(b) ├── COLON(:) ├── <type> │ └── KEYWORD(integer) └── SEMICOLON(;) └─ <compound-statement> ├── KEYWORD(mulai) └─ <statement-list> ├── <assignment-statement> │ ├── IDENTIFIER(a) │ ├── ASSIGN_OPERATOR(:=) │ └─ <expression> │ └─ <simple-expression> │ └─ <term> │ └─ <factor> │ └─ NUMBER(5) ├── SEMICOLON(;) └─ <assignment-statement> ├── IDENTIFIER(b) ├── ASSIGN_OPERATOR(:=) └─ <expression></pre>



3.2 Kasus 2

Input	<pre> program LoopTest; variabel count: integer done: boolean; mulai count := 0; done := false; selama tidak done lakukan mulai count := count + 1; jika count >= 5 maka done := true; </pre>
--------------	--

	selesai; selesai.
Output	[Syntax Error] Syntax Error line 0: Expected ';' (Expected: ;, Got: IDENTIFIER(done)) Syntax Analysis Gagal.

3.3 Kasus 3

Input	<pre> program LoopTest; variabel count: integer; done: boolean; mulai biji; count := 0; done := false; selama tidak done lakukan mulai count := count + 1; jika count >= 5 maka done := true; selesai; selesai. </pre>
Output	[Syntax Error] Unknown statement at line 0. Got: IDENTIFIER(biji) Syntax Analysis Gagal.

3.4 Kasus 4

Input	<pre> program CekNilai; variabel n: integer; mulai n := 75; jika n > 80 maka writeln('A') selain-itu jika n > 70 writeln('B') selain-itu </pre>
--------------	--

	<pre>writeln('C');</pre> <pre>selesai.</pre>
Output	<p>[Syntax Error] Syntax Error line 0: Expected 'selesai' (Expected: selesai, Got: IDENTIFIER(selain))</p> <p>Syntax Analysis Gagal.</p>

3.5 Kasus 5

Input	<pre>program CekNilai;</pre> <pre>variabel</pre> <pre> n: integer;</pre> <pre>mulai</pre> <pre> n := 75;</pre> <pre> jika n > 80 maka</pre> <pre> writeln('A')</pre> <pre> selain-itu</pre> <pre> jika n > 70 maka</pre> <pre> writeln('B')</pre> <pre> selain-itu</pre> <pre> writeln('C');</pre> <pre>selesai.</pre>
Output	<p>Menjalankan Syntax Analysis...</p> <p>[Syntax Error] Syntax Error line 0: Expected 'selesai' (Expected: selesai, Got: IDENTIFIER(selain))</p> <p>Syntax Analysis Gagal.</p>

BAB IV

KESIMPULAN DAN SARAN

4.1 Kesimpulan

Berdasarkan hasil implementasi yang telah diperoleh, dapat dinyatakan bahwa implementasi telah berhasil dengan pendekatan algoritma *recursive descent*. Pendekatan dari algoritma ini memiliki beberapa kekurangan dan kelebihan, yakni algoritma ini memiliki kelebihan berupa penyelesaian yang akan selalu mendapatkan terminal / mencapai akhir dari komputasi, sementara itu kekurangan dari algoritma ini adalah pendekatan yang menggunakan algoritma DFS akan memiliki kompleksitas yang cenderung mahal karena membutuhkan waktu yang cukup besar apabila terdapat kompleksitas dan fitur bahasa baru yang akan diterapkan selanjutnya.

Milestone 2 mengimplementasikan *syntax analyzer* untuk bahasa Pascal-S menggunakan automata Context Free Grammar (CFG). Program mampu membaca file kode sumber dan menghasilkan *Abstract Syntax Tree* (AST) sesuai spesifikasi. Tahapan ini menjadi fondasi untuk milestone selanjutnya dalam pembuatan *compiler*, yakni *semantical analyzer*.

4.2 Saran

1. Tidak menunda pekerjaan.
2. Melakukan pembagian tugas dengan jelas dan rinci.

REFERENSI

- <https://www.geeksforgeeks.org/compiler-design/introduction-to-compilers/>
- <https://www.geeksforgeeks.org/compiler-design/introduction-to-syntax-analysis-in-compiler-design/>
- <https://www.geeksforgeeks.org/compiler-design/parse-tree-in-compiler-design/>
- <http://pascal.hansotten.com/uploads/pascals/PASCAL-S%20A%20subset%20and%20its%20Implementation%20012.pdf>
- https://www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.htm
- Aho, et al. (2007). Compilers : Principles, Techniques, & Tools. New York : Pearson Education Inc.

LAMPIRAN

1. Tauran Repository Github
<https://github.com/BrianHadianSTEI23/TMP-Tubes-IF2224>
2. Tautan Release Repository Github
<https://github.com/BrianHadianSTEI23/TMP-Tubes-IF2224/tree/v0.2.2>
3. Pembagian Tugas

Nama	NIM	Tugas
Bertha Soliany Frandi	13523026	Program, testing (25%)
Brian Albar Hadian	13523048	Program (25%)
Muhammad Izzat Jundy	13523092	Program (25%)
Michael Alexander Angkawijaya	13523102	Laporan (25%)