Milestone 2 Syntax Analysis

IF2224 *Formal Language and Automata Theory*

Laboratorium Ilmu Rekayasa dan Komputasi

Oleh:

Kelompok HJE - HidupJalaninAje

Sebastian Enrico Nathanael (13523134)

Jonathan Kenan Budianto (13523139)

Mahesa Fadhillah Andre (13523140)

Muhammad Farrel Wibowo (13523153)

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika - Institut Teknologi Bandung

Jl. Ganesha 10, Bandung 40132

2025

Daftar Isi

BAB I
Landasan Teori

Proses kompilasi merupakan suatu rangkaian tahapan yang saling terhubung dan berurutan. Setiap tahapan menghasilkan keluaran yang menjadi input untuk tahap berikutnya. Oleh karena itu, sangat penting untuk memastikan bahwa hasil dari setiap langkah sudah benar agar tidak mengganggu kelancaran proses selanjutnya. Dalam tugas ini, kami memanfaatkan lima tahap utama dalam proses kompilasi, yaitu ***Lexical Analysis, Syntax Analysis, Semantic Analysis, Intermediate Code Generation, dan Interpreter.*** Berikut adalah penjelasan lebih lanjut mengenai setiap tahapan tersebut:

1.1 *Syntax Analysis*

Parser adalah komponen dalam compiler yang bertugas melakukan *syntax analysis*, yaitu memeriksa apakah urutan token yang dihasilkan oleh lexer sudah sesuai dengan tata bahasa (*grammar*) bahasa pemrograman. Jika lexer bekerja di level karakter dan hanya mengubah deretan karakter menjadi token-token seperti *KEYWORD*, *IDENTIFIER*, *NUMBER*, atau *OPERATOR*, maka parser bekerja di level struktur. Parser memastikan token-token tersebut tersusun menjadi bentuk yang sah, misalnya program lengkap, bagian deklarasi, rangkaian pernyataan, hingga ekspresi. Di dalam proses ini, parser juga memegang peran penting dalam mendeteksi dan melaporkan kesalahan sintaks: ketika urutan token tidak mengikuti aturan grammar, parser harus dapat menghentikan proses dan memberi pesan kesalahan yang jelas. Selain memvalidasi sintaks, parser biasanya membangun representasi struktur program (seperti *parse tree*) yang akan digunakan pada tahap analisis semantik dan tahap-tahap kompilasi berikutnya.

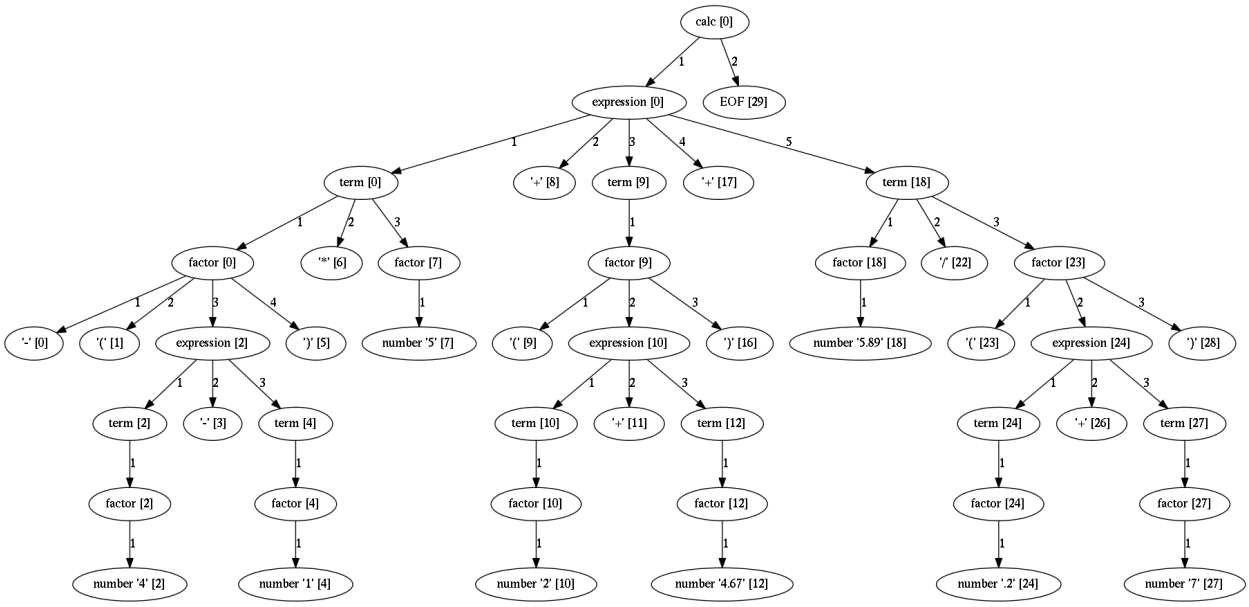
1.2 Keterkaitan dengan lexer

Lexer dan parser bekerja berurutan dan saling bergantung. Lexer terlebih dahulu membaca source code mentah dan memecahnya menjadi deretan token yang sudah diklasifikasikan, misalnya KEYWORD(program), IDENTIFIER(namaProgram), SEMICOLON(;), dan seterusnya. Parser kemudian tidak lagi berurusan dengan karakter satu per satu, melainkan hanya dengan aliran token yang dihasilkan lexer. Karena itu, definisi kategori token di lexer harus konsisten dengan grammar yang digunakan parser. Ketidaksesuaian, misalnya lexer mengelompokkan suatu lexeme ke jenis token yang salah, dapat membuat parser gagal mengenali struktur yang sebenarnya valid, atau justru menerima struktur yang seharusnya ditolak. Pemisahan peran ini menjadikan sistem lebih terstruktur: kesalahan pada level penulisan simbol (seperti string yang tidak ditutup) ditangani lexer, sedangkan kesalahan pada level

susunan bahasa (seperti titik koma yang hilang atau urutan kata kunci yang tidak tepat) menjadi tanggung jawab parser.

1.3 Parse Tree

Parse tree adalah representasi pohon yang menggambarkan secara lengkap bagaimana suatu program “dihasilkan” dari grammar. Akar pohon adalah simbol awal grammar (misalnya <program>), node-node internal merepresentasikan non-terminal seperti <declaration-part>, <compound-statement>, atau <expression>, sedangkan daun pohon berisi simbol terminal berupa token-token konkret seperti KEYWORD(program), IDENTIFIER, NUMBER, atau SEMICOLON. Dengan susunan seperti ini, parse tree memperlihatkan secara hierarkis struktur sintaks sebuah program: bagian mana yang menjadi header, mana yang termasuk deklarasi, mana kumpulan pernyataan, dan bagaimana ekspresi disusun dari operator dan operand. Dalam konteks compiler, parse tree tidak hanya berfungsi sebagai bukti bahwa program valid secara sintaks, tetapi juga menjadi jembatan menuju tahap berikutnya, seperti analisis semantik atau pembentukan abstract syntax tree (AST) yang lebih ringkas. Struktur pohon yang benar dan konsisten akan sangat membantu dalam memeriksa makna program serta dalam mengimplementasikan transformasi atau optimasi di tahap selanjutnya.



Contoh Parse Tree

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

1.4 Algoritma *Recursive Descent*

Recursive Descent adalah salah satu teknik top-down parsing yang mengimplementasikan grammar secara langsung dalam bentuk fungsi-fungsi rekursif. Ide utamanya adalah: setiap non-terminal di grammar direpresentasikan oleh satu prosedur atau fungsi dalam program. Misalnya, non-terminal <program> diwakili oleh fungsi parseProgram(), <statement> oleh parseStatement(), dan <expression> oleh parseExpression(). Proses parsing dimulai dari simbol awal grammar, lalu fungsi-fungsi tersebut saling memanggil secara rekursif mengikuti aturan produksi yang ada. Di dalam setiap fungsi, parser akan memeriksa token saat ini, membandingkannya dengan token yang diharapkan oleh grammar, mengonsumsi token yang sesuai, lalu melanjutkan ke bagian berikutnya. Pendekatan ini mudah dipahami karena struktur kode sangat mirip dengan struktur grammar. Namun, grammar yang digunakan perlu disesuaikan agar cocok untuk parsing top-down, misalnya menghilangkan left recursion langsung dan menata ulang produksi agar keputusan cabang dapat diambil hanya dengan melihat token berikutnya.

BAB II
Perencanaan & Implementasi

2.1 Pemilihan bahasa dan Justifikasi

Proyek compiler Pascal-S ini diimplementasikan menggunakan Python 3 sebagai bahasa pemrograman utama. Pemilihan bahasa ini didasarkan pada berbagai pertimbangan yang mendukung kemudahan dalam pengembangan, efisiensi dalam implementasi struktur data, serta kecepatan dalam siklus pengembangan. Berikut adalah justifikasi terkait pemilihan Python:

1. **Kemudahan Implementasi Struktur Data Kompleks**
   Python menyediakan berbagai struktur data built-in yang sangat mendukung pengelolaan informasi dalam proyek compiler ini. Misalnya, dalam implementasi DFA (Deterministic Finite Automaton), struktur *dictionary* digunakan untuk menyimpan transisi antar state. Hal ini mempermudah pemodelan dan pemrosesan data dalam bentuk yang lebih mudah dipahami dan dikelola. Selain itu, penggunaan *list* untuk menyimpan urutan token serta *dictionary* untuk memetakan kata kunci dan identifier pada simbol tabel juga meningkatkan efisiensi dalam pengelolaan data yang diperlukan.
2. **Manipulasi String dan File yang Efisien**
   Python dilengkapi dengan berbagai library built-in yang mendukung manipulasi string dan pengelolaan file dengan mudah. Misalnya, penggunaan modul *os* untuk File I/O mempermudah pembacaan file kode Pascal-S dan penulisan hasil output.
3. **Penanganan JSON untuk Konfigurasi DFA**
   Dalam proyek ini, aturan DFA disimpan dalam file eksternal dalam format JSON. Python menyediakan dukungan native untuk parsing JSON, sehingga memudahkan pembacaan aturan DFA dan penggunaan aturan tersebut dalam pemrosesan kode sumber.

2.2 Struktur Program

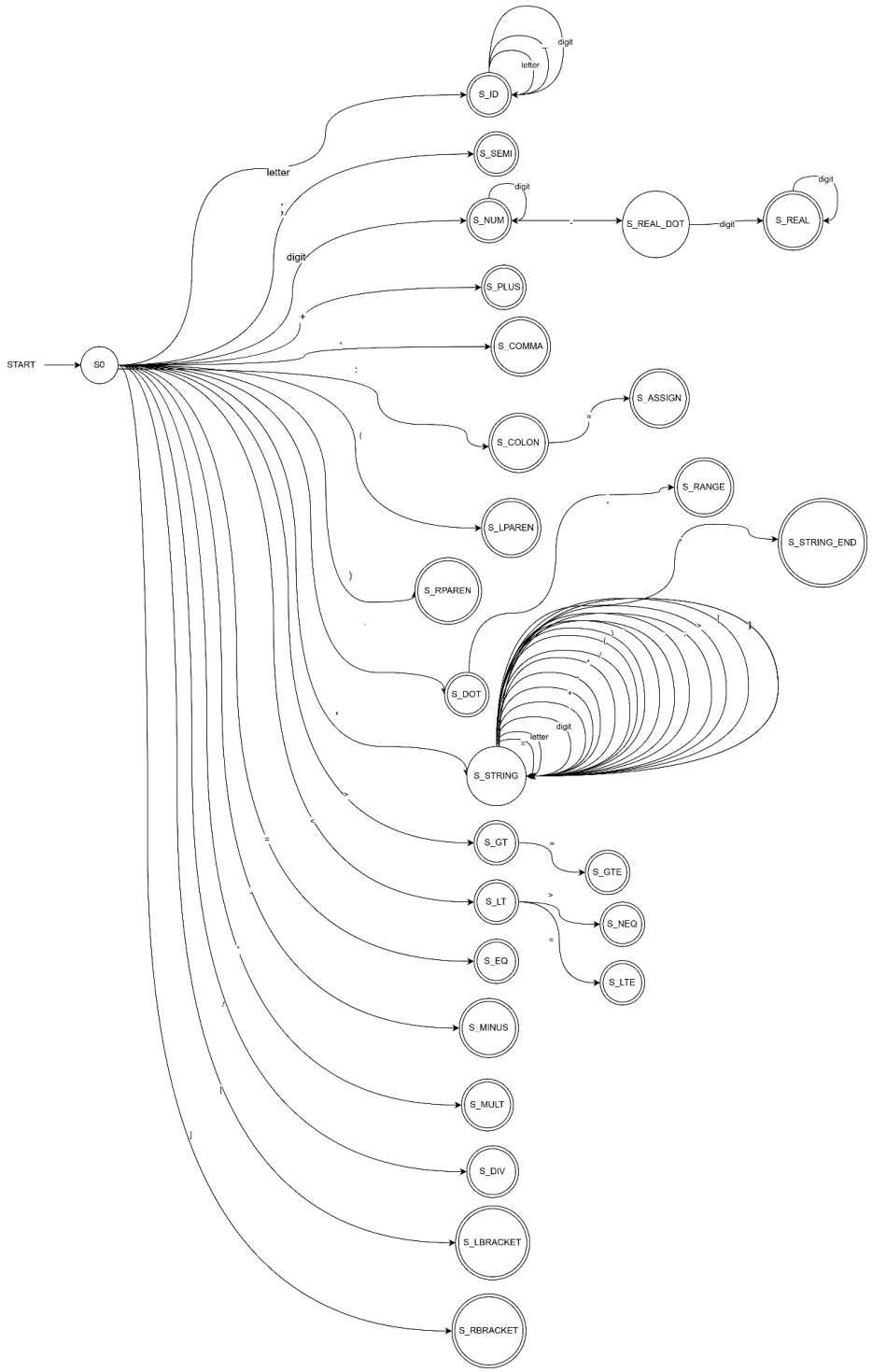
```
HJE-TUBES-IF2224/
│
├── doc/
│   ├── gambar diagram/
│   │   └── diagram.png
│   └── laporan/
│   └── Laporan-1-HJE.pdf
│   └── Laporan-2-HJE.pdf
│
├── src/
│   ├── compiler.md
```

```
|   ├── compiler.py
|   ├── dfa_rules.json
|   ├── lexer.py
|   ├── parser.py
|   └── pascal_token.py
|
├── test/
|   └── milestone-1/
|   |   ├── input/
|   |   └── output/
|   └── milestone-2/
|       ├── input/
|       └── output/
├── README.md
```

Penjelasan Struktur:

1. **doc/:**
    a. Berisi file dokumentasi dan referensi seperti gambar yang dapat digunakan untuk penjelasan dalam laporan atau presentasi.
2. **src/:**
    a. compiler.py: Merupakan file utama untuk menjalankan proses kompilasi kode Pascal-S.
    b. dfa_rules.json: Berisi aturan DFA yang diperlukan oleh lexer untuk melakukan analisis leksikal.
    c. lexer.py: Modul yang menangani pemindaian dan analisis leksikal dari kode sumber untuk menghasilkan token.
    pascal_token.py: Berisi definisi dan representasi token yang digunakan dalam kode Pascal-S.
    d. Pascal_token.py: Berisi kelas Token, sebuah struktur data sederhana untuk merepresentasikan token individual (tipe, nilai, baris, kolom).
    e. parser.py: Berisi kelas parser dan node. Kelas parser bertanggung jawab penuh atas analisis sintaksis menggunakan metode *Recursive Descent* untuk membangun *Parse Tree*.
3. **test/:**
    a. Folder ini digunakan untuk menyimpan file input (kode Pascal-S) dan output yang dihasilkan oleh compiler untuk pengujian.
4. **README.md:**
    a. File dokumentasi utama yang menjelaskan tentang proyek ini, cara menjalankannya, dan dependensi yang diperlukan.

2.3 Penjelasan Diagram DFA



Link: [dfa diagram]

2.3.1. Start State (S0)

Diagram DFA dimulai pada start state, yaitu S0. Ini adalah titik awal dalam proses scanning kode sumber. Pada state ini, DFA menunggu untuk menerima input pertama dari kode sumber, seperti huruf, angka, atau simbol lainnya. Berdasarkan input pertama, DFA akan bergerak ke state berikutnya sesuai dengan aturan transisi yang sudah didefinisikan.

2.3.2. Final States

Final states dalam diagram DFA adalah state yang menandakan bahwa token tertentu telah dikenali. Setiap final state ini memiliki makna spesifik berdasarkan jenis token yang terdeteksi. Misalnya:

- S_ID: Menandakan token IDENTIFIER, yaitu variabel atau nama fungsi.
- S_NUM: Menandakan token NUMBER, yang bisa berupa bilangan bulat.
- S_REAL: Menandakan token REAL NUMBER atau bilangan desimal.
- S_SEMI: Menandakan token SEMICOLON, yaitu tanda titik koma.
- S_ASSIGN: Menandakan token ASSIGN_OPERATOR seperti :=.

Final states ini digunakan untuk memberi tahu DFA bahwa sebuah token telah berhasil dikenali dan siap untuk diproses lebih lanjut dalam langkah-langkah berikutnya.

2.3.3. Transisi antar State

Transisi antar state dalam DFA ini sangat tergantung pada simbol input yang diterima. Setiap simbol input (huruf, angka, operator, atau tanda baca) akan memicu transisi dari satu state ke state berikutnya. Misalnya:

- Dari S0 (start state), jika karakter yang dibaca adalah letter (huruf), DFA akan bergerak ke S_ID, yang menandakan bahwa input berikutnya adalah sebuah identifier.
- Jika karakter yang dibaca adalah digit (angka), DFA akan bergerak ke S_NUM, untuk mengenali bilangan.
- Jika karakter yang dibaca adalah operator +, DFA akan bergerak ke S_PLUS untuk mengenali operator aritmatika.

Setiap state dalam diagram DFA ini memiliki transisi yang jelas, yang memungkinkan DFA untuk memindai dan mengidentifikasi token dalam kode dengan efisien.

2.3.4. State Khusus untuk Literals dan Operators

Diagram DFA ini juga memiliki transisi khusus untuk mengenali berbagai jenis simbol dalam kode Pascal-S. Misalnya:

- S_STRING dan S_STRING_END: Digunakan untuk mengenali string literal dalam kode Pascal-S. Ketika DFA membaca tanda kutip tunggal ('), DFA akan memasuki state S_STRING dan melanjutkan pemindaian karakter string. Ketika ditemukan tanda kutip penutup ('), DFA akan bertransisi ke S_STRING_END, yang menandakan akhir dari string literal.
- S_REAL_DOT: Setelah membaca angka (S_NUM), jika ditemukan simbol titik ("."), DFA akan bertransisi ke S_REAL_DOT, yang menunjukkan bahwa input tersebut adalah bagian dari angka desimal.

2.3.5. Relational dan Arithmetic Operators

Diagram DFA ini juga menangani berbagai relational operators (seperti >, <, =, >=, <=, !=) dan arithmetic operators (seperti +, -, *, /). Misalnya:

- S_GT, S_LT, dan S_EQ: Menandakan operator relasional seperti greater than, less than, dan equal to.
- S_PLUS, S_MINUS, S_MULT, S_DIV: Menandakan operator aritmatika.

Pada diagram ini, transisi antara state disesuaikan dengan simbol yang terdeteksi, memastikan bahwa operator-operator ini dikenali dan diproses dengan benar.

2.4 Implementasi Parser

2.4.1. Kelas Node

Untuk membangun *Parse Tree*, kami mengimplementasikan kelas Node. Setiap *node* dalam pohon merepresentasikan sebuah simbol (baik terminal maupun *non-terminal*).

- Setiap Node memiliki atribut name (misalnya, <program>, <expression>, atau IDENTIFIER(a)) dan sebuah *list* children.
- Terdapat metode add_child(node) untuk menambahkan *node* anak ke dalam *list children*.
- Kami juga mengimplementasikan metode print_tree() yang mencetak struktur pohon ke konsol dengan format visual (menggunakan awalan ├── dan └──) agar mudah dibaca dan diverifikasi.

2.4.2. Kelas Parser

Kelas ini adalah implementasi algoritma *Recursive Descent*.

- Inisialisasi: *Parser* diinisialisasi dengan list of tokens yang diterima dari Lexer. Kami menggunakan sebuah *pointer* (self.token_index) dan self.current_token untuk melacak token yang sedang diproses.
- Fungsi Pembantu: Terdapat tiga fungsi pembantu utama:
    1. advance(): Untuk mengonsumsi token saat ini dan memajukan *pointer* ke token berikutnya.
    2. peek(type, value): Untuk "mengintip" *token* saat ini tanpa mengonsumsinya. Ini sangat penting untuk mengambil keputusan, misalnya untuk mengecek keberadaan klausa selain_itu opsional pada if-statement.
    3. expect(type, value): Ini adalah fungsi validasi utama. Fungsi ini memeriksa apakah current_token sesuai dengan tipe (dan nilai, jika perlu) yang diharapkan oleh aturan grammar. Jika tidak sesuai, fungsi ini akan me-*raise* SyntaxError.
- Fungsi Grammar: Setiap *non-terminal* pada grammar PASCAL-S diimplementasikan sebagai satu metode di dalam kelas Parser.
    - Proses dimulai dari parse() yang memanggil program().
    - Metode program() kemudian memanggil program_header(), declaration_part(), compound_statement(), dan expect("DOT", ".") secara berurutan, sesuai dengan aturan produksinya.
    - Prioritas Operator: Prioritas operator aritmatika dan logika ditangani secara alami oleh struktur pemanggilan fungsi: expression() (prioritas terendah) memanggil simple_expression(), yang memanggil term(), yang memanggil factor() (prioritas tertinggi). Tanda kurung () di dalam factor() akan memanggil kembali expression(), memulai ulang siklus prioritas secara rekursif.

2.5 Alur kerja compiler.py

1. Input: Program menerima argumen *command-line* berupa *path* ke file PASCAL-S (.pas).
2. Membaca File: compiler.py membaca isi *source code* dari file tersebut.
3. Tahap Lexer:
    - Sebuah *instance* Lexer dibuat, dengan dfa_rules.json sebagai konfigurasinya.
    - Metode lexer.run_scanner() dipanggil untuk memproses *source code* mentah.
    - Hasilnya adalah sebuah list of tokens.
4. Tahap Parser:

- Sebuah *instance* Parser dibuat, dengan list of tokens dari *lexer* sebagai inputnya.
- Metode parser.parse() dipanggil dalam sebuah blok try...except.

5. Output:
    - Jika berhasil: Metode parse() mengembalikan sebuah parse_tree (sebuah Node *root*). Metode parse_tree.print_tree() kemudian dipanggil untuk menampilkan hasilnya ke konsol. Hasil ini juga ditulis ke dalam file parsetree-X.txt.
    - Jika gagal: Jika *parser* menemukan kesalahan sintaks (melalui expect()), sebuah SyntaxError akan dilempar. Blok except di compiler.py akan menangkap *error* ini dan mencetak pesan kesalahan yang informatif ke konsol, lalu menghentikan eksekusi.

2.6 Error Handling

Penanganan Syntax Error Handling diimplementasikan secara terpusat pada metode expect(token_type, value) di dalam kelas Parser.

Saat sebuah fungsi grammar (contoh: program_header()) mengharapkan *token* tertentu (contoh: SEMICOLON setelah IDENTIFIER), ia akan memanggil self.expect("SEMICOLON", ";").

- Kondisi berhasil: Jika self.current_token adalah SEMICOLON, fungsi expect() akan memanggil advance() dan program berjalan normal.
- Kondisi gagal: Jika self.current_token adalah hal lain (misalnya, KEYWORD(mulai)), fungsi expect() akan langsung me-*raise* SyntaxError. Pesan *error* ini diformat secara informatif untuk memberi tahu pengguna *token* apa yang diharapkan dan *token* apa yang sebenarnya ditemukan, contoh: Error Sintaks: Diharapkan token SEMICOLON, tetapi ditemukan KEYWORD(mulai) pada posisi X.

Error ini akan menghentikan proses *parsing* dan ditangkap oleh blok try...except di compiler.py, sehingga program tidak *crash* dan pengguna mendapatkan umpan balik/output yang jelas.

2.7 Implementasi

compiler.py

```
import sys
import os
import io
from lexer import Lexer
from pascal_token import Token
from parser import Parser

# KEYWORD Pascal-S
```

```
PASCAL_S_KEYWORDS = [
    # Keyword
    "program", "var", "begin", "end", "if", "then", "else", "while", "do",
    "for", "to", "downto", "integer", "real", "boolean", "char", "array",
    "of", "procedure", "function", "const", "type", "true", "false",
    # Padanan Bahasa Indonesia
    "program", "variabel", "mulai", "selesai", "jika", "maka", "selain_itu", "selama",
"lakukan",
    "untuk", "ke", "turun_ke", "integer", "real", "boolean", "char", "larik",
    "dari", "prosedur", "fungsi", "konstanta", "tipe", "true", "false"
]

def main():
    #Penerimaan Input File
    if len(sys.argv) != 2:
        print("Penggunaan: python compiler.py [Kode Pascal]")
        return

    pascal_file = sys.argv[1]

    if not os.path.exists(pascal_file) or not os.path.isfile(pascal_file):
            print(f"File input '{pascal_file}' tidak ditemukan atau bukan file yang valid.")
            raise SystemExit(1)

    #Membaca kode Pascal-S
    try:
        with open(pascal_file, 'r') as f:
            source_code = f.read()
    except Exception as e:
        print(f"Gagal membaca file input: {e}")
        return

    #Inisialisasi Lexer
    dfa_path = os.path.join(os.path.dirname(__file__), "dfa_rules.json")
    try:
        lexer = Lexer(dfa_path, PASCAL_S_KEYWORDS)
    except SystemExit:
        return

    # 4. Melakukan Scanning
    tokens = lexer.run_scanner(source_code)

    if tokens:
        # 5. Penghasilan Output Token ke File (Sesuai Milestone 1)
        input_path = pascal_file
        input_filename = os.path.basename(input_path)

        milestone_dir = os.path.dirname(os.path.dirname(input_path))
```

```
        test_number = "".join(filter(str.isdigit, input_filename))
        output_filename = f"output-{test_number}.txt"

        output_dir = os.path.join(milestone_dir, "output")
        os.makedirs(output_dir, exist_ok=True)

        output_path = os.path.join(output_dir, output_filename)

        try:
            with open(output_path, 'w') as f:
                for token in tokens:
                    f.write(str(token) + '\n')
            print(f"Output berhasil ditulis ke: {output_path}")
        except Exception as e:
            print(f"Gagal menulis file output token: {e}")

        # 6. Inisialisasi dan Jalankan Parser (Syntax Analysis)
        print("\nLexer selesai. Memulai parser...")
        parser = Parser(tokens)
        try:
            parse_tree = parser.parse()

            if parse_tree:
                print("\nParse Tree berhasil dibuat:")
                parse_tree.print_tree()

                try:
                    parsetree_filename = f"parsetree-{test_number}.txt"
                    parsetree_output_path = os.path.join(output_dir, parsetree_filename)

                    f_buffer = io.StringIO()
                    original_stdout = sys.stdout
                    sys.stdout = f_buffer

                    parse_tree.print_tree()

                    sys.stdout = original_stdout
                    tree_string = f_buffer.getvalue()

                    with open(parsetree_output_path, 'w', encoding='utf-8') as f:
                        f.write(tree_string)
                    print(f"Parse tree (format tree) berhasil ditulis ke:
{parsetree_output_path}")

                except Exception as e:
                    print(f"Gagal menulis file parse tree: {e}")
            else:
                print("Tidak ada output dari parser.")
```

```
        except SyntaxError as e:
            print(f"\n[PARSING GAGAL] {e}")

    else:
        print("Tidak ada token yang dihasilkan oleh lexer.")

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

Function	*Class/Entity*	Deskripsi
main()	*Global Constant*	Fungsi utama yang dieksekusi. Sebagai alur program dari input hingga output. Input: Memeriksa argumen command line (sys.argv), memverifikasi path file Pascal-S, dan membaca konten file menjadi string source_code. Inisialisasi: Menginisialisasi objek Lexer dengan path aturan DFA dan list PASCAL_S_KEYWORDS. Eksekusi: Memanggil lexer.run_scanner(source_code). Output: Menulis daftar token yang dihasilkan ke file output (output-[n].txt) di direktori yang sesuai, dan menampilkan *comment* ketika berhasil/gagal
PASCAL_S_KEYWORDS		Fungsi utamanya adalah sebagai kamus leksikal yang digunakan oleh method get_token_type di class Lexer

		untuk menyelesaikan ambiguitas DFA (*Deterministic Finite Automata*). Dengan kata lain, karena DFA tidak dapat membedakan antara keyword dan identifier hanya dari pola karakter, Lexer memeriksa lexeme yang ditemukan terhadap daftar ini untuk mengklasifikasikannya dengan benar.

lexer.py

```
import json
import os
from pascal_token import Token

class Lexer:
    """
    Melakukan analisis leksikal menggunakan DFA yang dimuat dari file.
    """

    def __init__(self, dfa_file_path, keyword_list):
        self.keywords = keyword_list
        self.dfa = self.load_dfa(dfa_file_path)

        self.current_index = 0
        self.current_line = 1
        self.current_coloumn = 1

    def load_dfa(self, file_path):
        """
        Membaca dan memparsing file aturan DFA (JSON atau TXT).
        """
        #print(f"Menginisialisasi DFA dari: {file_path}")
        try:
            with open(file_path, 'r') as f:
                dfa_data = json.load(f)

                if 'start_state' not in dfa_data or 'final_states' not in dfa_data or
'transitions' not in dfa_data:
                    print("File aturan DFA tidak lengkap")
                    return None
            return dfa_data
```

```
        except json.JSONDecodeError as e:
            print(f"Format File DFA tidak valid {e}")
            return None

        except FileNotFoundError:
            print(f"File tidak ditemukan di {file_path}")
            return None

        except ValueError as e:
            print(f"Error pada konfigurasi DFA {e}")
            return None

    def get_token_type(self, lexeme, final_state):
        """
        Menentukan tipe token, termasuk mengecek apakah lexeme adalah KEYWORD atau
IDENTIFIER.
        """
        token_type = self.dfa['final_states'].get(final_state)
        lexeme_lower = lexeme.lower()

        if token_type == "IDENTIFIER_CANDIDATE":
            if lexeme_lower in ["and", "or", "not", "dan", "atau", "tidak"]:
                return "LOGICAL_OPERATOR"
            elif lexeme_lower in ["div", "mod", "bagi"]:
                return "ARITHMETIC_OPERATOR"
            elif lexeme_lower in [k.lower() for k in self.keywords]:
                return "KEYWORD"
            else:
                return "IDENTIFIER"

        if token_type == "STRING_LITERAL":
            # Jika panjangnya 3 (contoh: 'a'), itu adalah CHAR_LITERAL
            if len(lexeme) == 3 or len(lexeme) == 2:
                return "CHAR_LITERAL"
            else:
                return "STRING_LITERAL"

        return token_type if token_type else "UNKNOWN_TOKEN"

    def advance_past_comment(self, source_code, index):
        """
        Melakukan scanning untuk melewati seluruh blok komentar (misalnya: (*...*) atau
{...}).
        Mengembalikan index setelah akhir komentar dan penyesuaian baris/kolom.
        """
        return index, 0, 0

    def classify_char_input(self, char):
        if char.isalpha():
```

```
            return "letter"
        if char.isdigit():
            return "digit"
        return char

    def run_scanner(self, source_code):
        """
        Melakukan scanning kode sumber huruf demi huruf menggunakan logika DFA.
        """
        tokens = []



        while self.current_index < len(source_code):
            token_start_line = self.current_line
            token_start_coloumn = self.current_coloumn

            char = source_code[self.current_index]

            #Buat handle whitespace
            if char.isspace():
                if char == '\n':
                    self.current_line += 1
                    self.current_coloumn = 1
                else:
                    self.current_coloumn += 1
                self.current_index += 1
                continue

            # Handle comment
            if char == '{' or (char == '(' and self.current_index + 1 < len(source_code) and
source_code[self.current_index + 1] == '*'):
                # Skip comment content
                if char == '{':
                    self.current_index += 1
                    self.current_coloumn += 1
                    while self.current_index < len(source_code) and
source_code[self.current_index] != '}':
                        if source_code[self.current_index] == '\n':
                            self.current_line += 1
                            self.current_coloumn = 1
                        else:
                            self.current_coloumn += 1
                        self.current_index += 1
                    if self.current_index < len(source_code):
                        self.current_index += 1
                        self.current_coloumn += 1
                else:  # Handle (* ... *) comments
                    self.current_index += 2 #skip (*
                    self.current_coloumn += 2
```

```
                    while self.current_index + 1 < len(source_code):
                        if (self.current_index + 1 < len(source_code) and
                            source_code[self.current_index:self.current_index+2] == '*)'):
                            self.current_index += 2
                            self.current_coloumn += 2
                            break
                        if source_code[self.current_index] == '\n':
                            self.current_line += 1
                            self.current_coloumn = 1
                        else:
                            self.current_coloumn += 1
                        self.current_index += 1
                continue

            #Scanning Token
            current_state = self.dfa['start_state']
            lexeme =""
            longest_finalstate = None
            longest_lexeme = ""
            last_valid_index = self.current_index

            temp_index = self.current_index

            while temp_index < len(source_code):
                temp_char = source_code[temp_index]
                input_symbol = self.classify_char_input(temp_char)

                # Cek ada transisi ga
                if current_state in self.dfa['transitions'] and input_symbol in
self.dfa['transitions'][current_state]:
                    current_state = self.dfa['transitions'][current_state][input_symbol]
                    lexeme += temp_char
                    temp_index += 1

                    # Cek current state nya final atau ga
                    if current_state in self.dfa['final_states']:
                        longest_lexeme = lexeme
                        longest_finalstate = current_state
                        last_valid_index = temp_index
                else:
                    # No valid transition, stop
                    break

            #Buat tokennya
            if longest_lexeme:
                token_type = self.get_token_type(longest_lexeme, longest_finalstate)
                tokens.append(Token(token_type, longest_lexeme, token_start_line,
token_start_coloumn))
```

```
                #terus majuin poinnya ke posisi setelah token found
                self.current_coloumn += len(longest_lexeme)
                self.current_index = last_valid_index
            else:
                # Lexical error unknown symbol
                tokens.append(Token("LEXICAL_ERROR", char, token_start_line,
token_start_coloumn))
                print(f"Simbol unknown '{char}' pada baris {token_start_line}")
                self.current_index += 1
                self.current_coloumn += 1

        return tokens
```

Function	Class	Deskripsi
__init__(...)	Lexer	*Constructor* untuk *class*. Inisiasi state untuk lexer Ia akan memanggil load_dfa, menyimpan list keyword, dan mengatur variabel scanning awal.
load_dfa(file_path)		Untuk memuat dan memverifikasi file aturan DFA. Ia akan membaca file JSON, memeriksa kunci-kunci wajib (start_state, final_states, transitions), dan mengembalikan struktur data DFA. Juga menangani error seperti file tidak ditemukan.
get_token_type(...)		Menentukan tipe token yang sebenarnya dari kandidat Lexer. Jika mencapai state IDENTIFIER_CANDIDATE, ia akan memeriksa lexeme untuk menentukan apakah itu LOGICAL_OPERATOR, ARITHMETIC_OPERATOR (mod/div), KEYWORD, atau fallback ke IDENTIFIER.

		Juga membedakan CHAR_LITERAL dari STRING_LITERAL berdasarkan panjang lexeme.
classify_char_input(char)		Mengubah karakter aktual menjadi simbol input yang digunakan dalam tabel transisi DFA . Mengubah 'A', 'B',... menjadi "letter". Mengubah '1', '2',... menjadi "digit". Karakter lain dikembalikan apa adanya (misalnya '+', ';').
advance_past_comment(...)		*Comment handler*, melewati konten di dalam blok komentar yang valid ({...} atau (*...*)). Menemukan karakter penutup dan memperbarui current_index, current_line, dan current_coloumn tanpa menghasilkan token apa pun.
run_scanner(source_code)		Fungsi utama yang menjalankan analisis leksikal. 1. Main Loop: Iterasi melalui source_code. 2. Pembersihan: Menangani dan melewati whitespace serta komentar. 3. *Greedy Match*: Menggunakan DFA untuk mencari *longest valid* lexeme. 4. Tokenisasi: Memanggil get_token_type dan menambahkan objek Token ke list tokens. 5. *Error Handling*: Melaporkan dan melanjutkan setelah menemukan simbol yang tidak dikenal.

pascal_token.py

```
class Token:
    """
    Merepresentasikan satu unit makna tunggal (Token).
    """
    def __init__(self, type, value, line=None, column=None):
        # inisiasi objek token
        self.type = type
        self.value = value
        self.line = line
        self.column = column

    def __str__(self):
        """
        print representasi string Token sesuai format output yang diminta
        """
        return f"{self.type}({self.value})"

        """
        return f"{self.type}({self.value})"
```

Function	Class	Deskripsi
__init__(...)	Token	Konstruktor, Menyimpan detail token (tipe, nilai, posisi).
__str__()		Format Output, Mendefinisikan bagaimana objek dicetak. Mengembalikan representasi string token dalam format TYPE(VALUE) (misalnya, KEYWORD(program)), yang digunakan untuk output akhir.

parser.py

```
# src/parser.py

class Node:
    """
    Kelas untuk merepresentasikan sebuah node dalam Parse Tree.
```

```
    """
    def __init__(self, name, value=None):
        self.name = name
        self.value = value
        self.children = []

    def add_child(self, node):
        self.children.append(node)

    def __repr__(self, level=0):
        ret = "\t" * level + f"{self.name}"
        if self.value:
            ret += f"({self.value})"
        ret += "\n"
        for child in self.children:
            ret += child.__repr__(level + 1)
        return ret

    def print_tree(self, prefix="", is_last=True, is_root=True):
        if is_root:
            print(self.name)
            new_prefix = ""
        else:
            connector = "└── " if is_last else "├── "
            print(f"{prefix}{connector}{self.name}")

            new_prefix = prefix + ("    " if is_last else "│   ")

        child_count = len(self.children)
        for i, child in enumerate(self.children):
            is_last_child = (i == child_count - 1)
            child.print_tree(new_prefix, is_last_child, is_root=False)

class Parser:
    """
    Melakukan syntax analysis menggunakan metode Recursive Descent.
    """
    def __init__(self, tokens):
        self.tokens = tokens
        self.token_index = 0
        self.current_token = self.tokens[self.token_index] if self.tokens else None

    def advance(self):
        self.token_index += 1
        if self.token_index < len(self.tokens):
            self.current_token = self.tokens[self.token_index]
        else:
            self.current_token = None
```

```
    def expect(self, token_type, value=None):
        token = self.current_token
        if token and token.type == token_type and (value is None or token.value.lower() ==
value.lower()):
            self.advance()
            node_name = f"{token.type}({token.value})"
            return Node(node_name)

        expected_val = f" dengan nilai '{value}'" if value else ""
        current_val = f"'{self.current_token.value}' ({self.current_token.type})" if
self.current_token else "None"
        raise SyntaxError(
            f"Error Sintaks: Diharapkan token {token_type}{expected_val}, tetapi ditemukan
{current_val} pada posisi {self.token_index}."
        )

    def peek(self, token_type, value=None):
        return self.current_token and self.current_token.type == token_type and \
               (value is None or self.current_token.value.lower() == value.lower())

    def parse(self):
        if not self.current_token:
            return None
        return self.program()

    # --- ATURAN PRODUKSI UTAMA ---

    def program(self):
        node = Node("<program>")
        node.add_child(self.program_header())
        node.add_child(self.declaration_part())
        node.add_child(self.compound_statement())
        node.add_child(self.expect("DOT", "."))
        print("Parsing Selesai!")
        return node

    def program_header(self):
        node = Node("<program-header>")
        node.add_child(self.expect("KEYWORD", "program"))
        node.add_child(self.expect("IDENTIFIER"))
        node.add_child(self.expect("SEMICOLON", ";"))
        return node

    def declaration_part(self):
        node = Node("<declaration-part>")
        while self.peek("KEYWORD", "variabel") or \
              self.peek("KEYWORD", "konstanta") or \
              self.peek("KEYWORD", "tipe") or \
              self.peek("KEYWORD", "prosedur") or \
```

```
            self.peek("KEYWORD", "fungsi"):

        if self.peek("KEYWORD", "variabel"):
            node.add_child(self.var_declaration())
        if self.peek("KEYWORD", "konstanta"):
            node.add_child(self.const_declaration())
        if self.peek("KEYWORD", "tipe"):
            node.add_child(self.type_declaration())
        if self.peek("KEYWORD", "prosedur") or self.peek("KEYWORD", "fungsi"):
            node.add_child(self.subprogram_declaration())
    return node

def compound_statement(self):
    node = Node("<compound-statement>")
    node.add_child(self.expect("KEYWORD", "mulai"))
    node.add_child(self.statement_list())
    node.add_child(self.expect("KEYWORD", "selesai"))
    return node

# --- ATURAN PRODUKSI DEKLARASI ---

def var_declaration(self):
    node = Node("<var-declaration>")
    node.add_child(self.expect("KEYWORD", "variabel"))
    while self.peek("IDENTIFIER"):
        node.add_child(self.identifier_list())
        node.add_child(self.expect("COLON", ":"))
        node.add_child(self.type_spec())
        node.add_child(self.expect("SEMICOLON", ";"))
    return node

def const_declaration(self):
    node = Node("<const-declaration>")
    node.add_child(self.expect("KEYWORD", "konstanta"))
    while self.peek("IDENTIFIER"):
        node.add_child(self.expect("IDENTIFIER"))
        node.add_child(self.expect("RELATIONAL_OPERATOR", "="))
        node.add_child(self.expect("NUMBER"))
        node.add_child(self.expect("SEMICOLON", ";"))
    return node

def type_declaration(self):
    node = Node("<type-declaration>")
    node.add_child(self.expect("KEYWORD", "tipe"))
    while self.peek("IDENTIFIER"):
        node.add_child(self.expect("IDENTIFIER"))
        node.add_child(self.expect("RELATIONAL_OPERATOR", "="))
        node.add_child(self.type_spec())
        node.add_child(self.expect("SEMICOLON", ";"))
```

```
        return node

    def subprogram_declaration(self):
        node = Node("<subprogram-declaration>")
        while self.peek("KEYWORD", "prosedur") or self.peek("KEYWORD", "fungsi"):
            if self.peek("KEYWORD", "prosedur"):
                node.add_child(self.procedure_declaration())
            elif self.peek("KEYWORD", "fungsi"):
                node.add_child(self.function_declaration())
        return node

    def procedure_declaration(self):
        node = Node("<procedure-declaration>")
        node.add_child(self.expect("KEYWORD", "prosedur"))
        node.add_child(self.expect("IDENTIFIER"))

        if self.peek("LPARENTHESIS", "("):
            node.add_child(self.formal_parameter_list())

        node.add_child(self.expect("SEMICOLON", ";"))
        node.add_child(self.declaration_part())
        node.add_child(self.compound_statement())
        node.add_child(self.expect("SEMICOLON", ";"))
        return node

    def function_declaration(self):
        node = Node("<function-declaration>")
        node.add_child(self.expect("KEYWORD", "fungsi"))
        node.add_child(self.expect("IDENTIFIER"))

        if self.peek("LPARENTHESIS", "("):
            node.add_child(self.formal_parameter_list())

        node.add_child(self.expect("COLON", ":"))
        node.add_child(self.type_spec())
        node.add_child(self.expect("SEMICOLON", ";"))
        node.add_child(self.declaration_part())
        node.add_child(self.compound_statement())
        node.add_child(self.expect("SEMICOLON", ";"))
        return node

    def formal_parameter_list(self):
        node = Node("<formal-parameter-list>")
        node.add_child(self.expect("LPARENTHESIS", "("))
        node.add_child(self.parameter_group())
        while self.peek("SEMICOLON", ";"):
            node.add_child(self.expect("SEMICOLON", ";"))
            node.add_child(self.parameter_group())
        node.add_child(self.expect("RPARENTHESIS", ")"))
```

```
        return node

    def parameter_group(self):
        node = Node("<parameter-group>")
        node.add_child(self.identifier_list())
        node.add_child(self.expect("COLON", ":"))
        node.add_child(self.type_spec())
        return node

    def identifier_list(self):
        node = Node("<identifier-list>")
        node.add_child(self.expect("IDENTIFIER"))
        while self.peek("COMMA"):
            node.add_child(self.expect("COMMA", ","))
            node.add_child(self.expect("IDENTIFIER"))
        return node

    def type_spec(self):
        node = Node("<type>")
        if self.peek("KEYWORD", "integer"):
            node.add_child(self.expect("KEYWORD", "integer"))
        elif self.peek("KEYWORD", "real"):
            node.add_child(self.expect("KEYWORD", "real"))
        elif self.peek("KEYWORD", "boolean"):
            node.add_child(self.expect("KEYWORD", "boolean"))
        elif self.peek("KEYWORD", "char"):
            node.add_child(self.expect("KEYWORD", "char"))
        elif self.peek("KEYWORD", "larik"):
            node.add_child(self.expect("KEYWORD", "larik"))
            node.add_child(self.expect("LBRACKET", "["))
            node.add_child(self.range_spec())
            node.add_child(self.expect("RBRACKET", "]"))
            node.add_child(self.expect("KEYWORD", "dari"))
            node.add_child(self.type_spec())
        else:
            expr_node = self.expression()

            if self.peek("RANGE_OPERATOR", ".."):
                subrange_node = Node("<subrange-type>")
                subrange_node.add_child(expr_node)
                subrange_node.add_child(self.expect("RANGE_OPERATOR", ".."))
                subrange_node.add_child(self.expression()) # Ambil expression kedua
                node.add_child(subrange_node)
            else:
                node.add_child(expr_node)
        return node

    def range_spec(self):
        node = Node("<range>")
```

```
        node.add_child(self.expression())
        node.add_child(self.expect("RANGE_OPERATOR", ".."))
        node.add_child(self.expression())
        return node

    # --- ATURAN PRODUKSI STATEMENT ---

    def statement_list(self):
        node = Node("<statement-list>")
        node.add_child(self.statement())
        while self.peek("SEMICOLON"):
            node.add_child(self.expect("SEMICOLON", ";"))
            if not self.peek("KEYWORD", "selesai"):
                 node.add_child(self.statement())
            else:
                break
        return node

    def statement(self):
            # 1. Cek Compound Statement (Blok mulai ... selesai)
            if self.peek("KEYWORD", "mulai"):
                return self.compound_statement()

            # 2. Cek If Statement (Percabangan)
            elif self.peek("KEYWORD", "jika"):
                return self.if_statement()

            # 3. Cek While Statement (Perulangan)
            elif self.peek("KEYWORD", "selama"):
                return self.while_statement()

            # 4. Cek For Statement (Perulangan Counter)
            elif self.peek("KEYWORD", "untuk"):
                return self.for_statement()

            # 5. Cek Identifier (Bisa Assignment ATAU Procedure Call)
            elif self.peek("IDENTIFIER"):
                next_token_idx = self.token_index + 1

                is_assignment = False
                if next_token_idx < len(self.tokens):
                    next_type = self.tokens[next_token_idx].type
                    # Assignment ditandai dengan ':=' ATAU '[' (untuk array)
                    if next_type == "ASSIGN_OPERATOR" or next_type == "LBRACKET":
                        is_assignment = True

                if is_assignment:
                    return self.assignment_statement()
                else:
```

```
                    return self.procedure_call()

            # 6. Handle Empty Statement (titik koma berlebih)
            elif self.peek("SEMICOLON"):
                return Node("<empty-statement>")

            else:
                raise SyntaxError(f"Error Sintaks: Diharapkan statement, ditemukan
'{self.current_token.value}'")

    def assignment_statement(self):
        # Grammar: ID [ '[' expression ']' ] := expression
        node = Node("<assignment-statement>")

        # 1. Nama Variabel
        node.add_child(self.expect("IDENTIFIER"))

        # 2. Cek apakah ini akses Array? (Opsional)
        if self.peek("LBRACKET", "["):
             node.add_child(self.expect("LBRACKET", "["))
             node.add_child(self.expression()) # Indeks
             node.add_child(self.expect("RBRACKET", "]"))

        # 3. Operator Assignment
        node.add_child(self.expect("ASSIGN_OPERATOR", ":="))

        # 4. Nilai Baru
        node.add_child(self.expression())

        return node

    # ATURAN PRODUKSI EKSPRESI

    def expression(self):
        """
        Aturan produksi: <expression> -> <simple-expression> [ <relational-operator>
<simple-expression> ]
        Saat ini hanya mengimplementasikan bagian pertama.
        """
        node = Node("<expression>")
        node.add_child(self.simple_expression())
        # Cek Operator Relasional
        if self.current_token and self.current_token.type == "RELATIONAL_OPERATOR":
            node.add_child(self.expect("RELATIONAL_OPERATOR"))
            node.add_child(self.simple_expression())

        return node

    def simple_expression(self):
```

```
        """
        Aturan produksi: <simple-expression> -> <term> ( <additive-operator> <term> )*
        """
        node = Node("<simple-expression>")

        # Handle unary operator (+/-) di depan angka (misal: -5)
        if self.peek("ARITHMETIC_OPERATOR", "+") or self.peek("ARITHMETIC_OPERATOR", "-"):
             node.add_child(self.expect("ARITHMETIC_OPERATOR"))

        node.add_child(self.term()) # Selalu dimulai dengan term

        # Loop jika ada operator tambah/kurang
        while self.current_token and self.current_token.value in ['+', '-', 'atau']:
            op_token = self.current_token
            if op_token.value == '+':
                node.add_child(self.expect("ARITHMETIC_OPERATOR", "+"))
                node.add_child(self.term())
            elif op_token.value == '-':
                 node.add_child(self.expect("ARITHMETIC_OPERATOR", "-"))
                 node.add_child(self.term())
            elif op_token.value.lower() == 'atau':
                 node.add_child(self.expect("LOGICAL_OPERATOR", "atau"))
                 node.add_child(self.term())

        return node

    def term(self):
        """
        Aturan produksi: <term> -> <factor> ( <multiplicative-operator> <factor> )*
        """
        node = Node("<term>")
        node.add_child(self.factor()) # Selalu dimulai dengan factor

        # Loop jika ada operator kali/bagi
        while self.current_token and self.current_token.value in ['*', '/', 'bagi', 'mod',
'dan']:
            if self.peek("ARITHMETIC_OPERATOR", "*"):
                node.add_child(self.expect("ARITHMETIC_OPERATOR", "*"))
                node.add_child(self.factor())
            elif self.peek("ARITHMETIC_OPERATOR", "/"):
                node.add_child(self.expect("ARITHMETIC_OPERATOR", "/"))
                node.add_child(self.factor())
            elif self.peek("ARITHMETIC_OPERATOR", "bagi"): # div
                node.add_child(self.expect("ARITHMETIC_OPERATOR", "bagi"))
                node.add_child(self.factor())
            elif self.peek("ARITHMETIC_OPERATOR", "mod"): # mod
                node.add_child(self.expect("ARITHMETIC_OPERATOR", "mod"))
                node.add_child(self.factor())
            elif self.peek("LOGICAL_OPERATOR", "dan"): # and
```

```
                node.add_child(self.expect("LOGICAL_OPERATOR", "dan"))
                node.add_child(self.factor())
            else:
                break

        return node

    def factor(self):
        """
        Aturan produksi: <factor> -> IDENTIFIER | NUMBER | STRING | CHAR | ( <expression> )
| true | false
        """
        node = Node("<factor>")

        if self.peek("IDENTIFIER"):
            # Cek apakah ini Function Call (ID diikuti kurung buka)
            next_token_idx = self.token_index + 1

            # Kasus 1: Function Call -> nama_fungsi(...)
            if next_token_idx < len(self.tokens) and self.tokens[next_token_idx].type ==
"LPARENTHESIS":
                node.add_child(self.function_call())

            # Kasus 2: Array Access -> nama_array[indeks]
            elif next_token_idx < len(self.tokens) and self.tokens[next_token_idx].type ==
"LBRACKET":
                node.add_child(self.expect("IDENTIFIER"))
                node.add_child(self.expect("LBRACKET", "["))
                node.add_child(self.expression()) # Indeks array
                node.add_child(self.expect("RBRACKET", "]"))

            # Kasus 3: Variabel Biasa
            else:
                node.add_child(self.expect("IDENTIFIER"))

        elif self.peek("NUMBER"):
            node.add_child(self.expect("NUMBER"))

        elif self.peek("STRING_LITERAL"):
            node.add_child(self.expect("STRING_LITERAL"))

        elif self.peek("CHAR_LITERAL"):
            node.add_child(self.expect("CHAR_LITERAL"))

        elif self.peek("KEYWORD", "true"):
            node.add_child(self.expect("KEYWORD", "true"))

        elif self.peek("KEYWORD", "false"):
            node.add_child(self.expect("KEYWORD", "false"))
```

```
        elif self.peek("LOGICAL_OPERATOR", "tidak"): # Operator NOT
            node.add_child(self.expect("LOGICAL_OPERATOR", "tidak"))
            node.add_child(self.factor())

        elif self.peek("LPARENTHESIS", "("):
            node.add_child(self.expect("LPARENTHESIS", "("))
            node.add_child(self.expression())
            node.add_child(self.expect("RPARENTHESIS", ")"))

        else:
            val = self.current_token.value if self.current_token else "EOF"
            raise SyntaxError(f"Error Sintaks: Diharapkan factor, ditemukan '{val}'")

        return node

    # Control Flow Statements
    def if_statement(self):
        # Grammar: jika <expression> maka <statement> [selain_itu <statement>]
        node = Node("<if-statement>")
        node.add_child(self.expect("KEYWORD", "jika"))
        node.add_child(self.expression())
        node.add_child(self.expect("KEYWORD", "maka"))
        node.add_child(self.statement())

        # Cek apakah ada 'selain_itu' (else)
        if self.peek("KEYWORD", "selain_itu"):
            node.add_child(self.expect("KEYWORD", "selain_itu"))
            node.add_child(self.statement())

        return node

    def while_statement(self):
        # Grammar: selama <expression> lakukan <statement>
        node = Node("<while-statement>")
        node.add_child(self.expect("KEYWORD", "selama"))
        node.add_child(self.expression())
        node.add_child(self.expect("KEYWORD", "lakukan"))
        node.add_child(self.statement())
        return node

    def for_statement(self):
        # Grammar: untuk <id> := <expr> ke/turun_ke <expr> lakukan <statement>
        node = Node("<for-statement>")
        node.add_child(self.expect("KEYWORD", "untuk"))
        node.add_child(self.expect("IDENTIFIER"))
        node.add_child(self.expect("ASSIGN_OPERATOR", ":="))
        node.add_child(self.expression())
```

```
        # Cek arah loop
        if self.peek("KEYWORD", "ke"):
            node.add_child(self.expect("KEYWORD", "ke"))
        elif self.peek("KEYWORD", "turun_ke"):
            node.add_child(self.expect("KEYWORD", "turun_ke"))
        else:
            raise SyntaxError("Error Sintaks: Diharapkan 'ke' atau 'turun_ke' dalam loop
'untuk'.")

        node.add_child(self.expression())
        node.add_child(self.expect("KEYWORD", "lakukan"))
        node.add_child(self.statement())
        return node

    # Procedure Call
    def procedure_call(self):
        # Grammar: IDENTIFIER ( [ <parameter-list> ] )
        node = Node("<procedure-call>")
        node.add_child(self.expect("IDENTIFIER"))
        node.add_child(self.expect("LPARENTHESIS", "("))

        if not self.peek("RPARENTHESIS", ")"):
             node.add_child(self.parameter_list())

        node.add_child(self.expect("RPARENTHESIS", ")"))

        return node

    def parameter_list(self):
        # Grammar: <expression> (, <expression>)*
        node = Node("<parameter-list>")
        node.add_child(self.expression())

        while self.peek("COMMA", ","):
            node.add_child(self.expect("COMMA", ","))
            node.add_child(self.expression())

        return node

    # Function Call
    def function_call(self):
        # Mirip procedure call tapi mengembalikan nilai (bagian dari factor)
        node = Node("<function-call>")
        node.add_child(self.expect("IDENTIFIER"))
        node.add_child(self.expect("LPARENTHESIS", "("))

        # Parameter opsional untuk fungsi
        if not self.peek("RPARENTHESIS", ")"):
             node.add_child(self.parameter_list())
```

```
        node.add_child(self.expect("RPARENTHESIS", ")"))
        return node
```

Function	Class	Deskripsi
__init__()	Node	Konstruktor, Menyimpan detail token (tipe, nilai, posisi).
add_child()		Format Output, Mendefinisikan bagaimana objek dicetak. Mengembalikan representasi string token dalam format TYPE(VALUE) (misalnya, KEYWORD(program)), yang digunakan untuk output akhir.
print_tree()		Fungsi rekursif untuk mencetak *Parse Tree* ke konsol dengan format visual yang rapi (menggunakan ├── dan └──).
__init__()	Parser	Sebagai konstruktor. Menyimpan list of tokens dari lexer dan menginisialisasi pointer (self.token_index, self.current_token).
advance()		Mengonsumsi current_token dan memajukan *pointer* ke *token* berikutnya.
peek()		Memeriksa current_token *tanpa* mengonsumsinya. Krusial untuk mengambil keputusan (misal, selain_itu opsional atau membedakan *assignment* dari *procedure call*).
expect()		Validasi dan Error Handling.

		Memeriksa jika current_token sesuai. Jika ya, panggil advance(). Jika tidak, raise SyntaxError dengan pesan informatif.
parse()		Titik masuk *parser*. Memulai proses *parsing* dengan memanggil program().
program()		Mengimplementasikan aturan <program>. Memanggil program_header(), declaration_part(), compound_statement(), dan expect("DOT", ".") secara berurutan.
declaration_part()		Menggunakan *loop* while dan peek() untuk mem-parsing blok deklarasi (konstanta, tipe, variabel, fungsi, prosedur) dalam urutan apapun.
statement()		Menggunakan peek() untuk menentukan jenis *statement* yang akan di-*parse* (misal, if_statement(), while_statement(), assignment_statement(), dll.)
expression(), simple_expression(), term(), factor()		Rangkaian fungsi yang mengimplementasikan prioritas operator. expression menangani prioritas terendah (relasional), factor menangani prioritas tertinggi (angka, (...)).
procedure_call()		Mengimplementasikan aturan pemanggilan prosedur (sebagai *statement*). Mengharapkan IDENTIFIER, diikuti (, lalu memanggil parameter_list() (jika *peek* bukan)), dan diakhiri).

parameter_list()		Mengimplementasikan aturan daftar parameter aktual. Mem-parse satu expression(), lalu melakukan *loop* mem-parse COMMA dan expression() tambahan selama masih ada.
function_call()		Mengimplementasikan aturan pemanggilan fungsi (sebagai bagian dari <factor>). Logikanya identik dengan procedure_call(): mem-parse IDENTIFIER, (, parameter_list() opsional, dan).

dfa_rules.json

```
{
    "start_state": "S0",
    "final_states": {
        "S_ID": "IDENTIFIER_CANDIDATE",
        "S_SEMI": "SEMICOLON",
        "S_NUM": "NUMBER",
        "S_REAL": "NUMBER",
        "S_PLUS": "ARITHMETIC_OPERATOR",
        "S_COMMA": "COMMA",
        "S_COLON": "COLON",
        "S_ASSIGN": "ASSIGN_OPERATOR",
        "S_LPAREN": "LPARENTHESIS",
        "S_RPAREN": "RPARENTHESIS",
        "S_DOT": "DOT",
        "S_RANGE": "RANGE_OPERATOR",
        "S_STRING_END": "STRING_LITERAL",
        "S_GT": "RELATIONAL_OPERATOR",
        "S_GTE": "RELATIONAL_OPERATOR",
        "S_LT": "RELATIONAL_OPERATOR",
        "S_LTE": "RELATIONAL_OPERATOR",
        "S_NEQ": "RELATIONAL_OPERATOR",
        "S_EQ": "RELATIONAL_OPERATOR",
        "S_MINUS": "ARITHMETIC_OPERATOR",
        "S_MULT": "ARITHMETIC_OPERATOR",
        "S_DIV": "ARITHMETIC_OPERATOR",
```

```
        "S_LBRACKET": "LBRACKET",
        "S_RBRACKET": "RBRACKET"
    },
    "transitions": {
        "S0": {
            "letter": "S_ID",
            ";": "S_SEMI",
            "digit": "S_NUM",
            "+": "S_PLUS",
            ",": "S_COMMA",
            ":": "S_COLON",
            "(": "S_LPAREN",
            ")": "S_RPAREN",
            ".": "S_DOT",
            "'": "S_STRING",
            ">": "S_GT",
            "<": "S_LT",
            "=": "S_EQ",
            "-": "S_MINUS",
            "*": "S_MULT",
            "/": "S_DIV",
            "[": "S_LBRACKET",
            "]": "S_RBRACKET"
        },
        "S_ID": {
            "letter": "S_ID",
            "digit": "S_ID",
            "_": "S_ID"
        },
        "S_SEMI": {},
        "S_NUM": {
            "digit": "S_NUM",
            ".": "S_REAL_DOT"
        },
        "S_REAL_DOT": {
            "digit": "S_REAL"
        },
        "S_REAL": {
            "digit": "S_REAL"
        },
        "S_PLUS": {},
        "S_COMMA": {},
        "S_COLON": {
            "=": "S_ASSIGN"
        },
        "S_ASSIGN": {},
        "S_LPAREN": {},
        "S_RPAREN": {},
        "S_DOT": {
```

```
            ".": "S_RANGE"
        },
        "S_RANGE": {},
        "S_STRING": {
            "letter": "S_STRING",
            "digit": "S_STRING",
            " ": "S_STRING",
            "=": "S_STRING",
            "+": "S_STRING",
            "-": "S_STRING",
            "*": "S_STRING",
            "/": "S_STRING",
            "(": "S_STRING",
            ")": "S_STRING",
            ",": "S_STRING",
            ".": "S_STRING",
            ":": "S_STRING",
            ";": "S_STRING",
            ">": "S_STRING",
            "<": "S_STRING",
            "[": "S_STRING",
            "]": "S_STRING",
            "'": "S_STRING_END"
        },
        "S_STRING_END": {},
        "S_GT": {
            "=": "S_GTE"
        },
        "S_GTE": {},
        "S_LT": {
            "=": "S_LTE",
            ">": "S_NEQ"
        },
        "S_LTE": {},
        "S_NEQ": {},
        "S_EQ": {},
        "S_MINUS": {},
        "S_MULT": {},
        "S_DIV": {},
        "S_LBRACKET": {},
        "S_RBRACKET": {}
    }
}
```

1. start_state

Nilai: "S0" Mendefinisikan state awal. Setiap kali Lexer selesai mengidentifikasi satu token atau mengabaikan whitespace/komentar, ia akan kembali ke S0 untuk memulai pengenalan token berikutnya.

2. final_states

Mendefinisikan semua state yang, jika dicapai, Lexer dapat mengklaim telah berhasil menemukan lexeme yang valid dan mengeluarkan Token. Beberapa state dipetakan ke tipe Token yang sama (misalnya, S_NUM dan S_REAL keduanya dipetakan ke NUMBER) karena keduanya merupakan Literal numerik.

3. transitions

Tabel logika utama DFA. Setiap kunci di tingkat pertama adalah State Awal, dan isinya adalah aturan yang menentukan perpindahan state berdasarkan Input Simbol (letter, digit, atau karakter spesifik).

State Awal	Input Dilihat	State Selanjutnya	Tipe Token yang Dibentuk	Konsep Utama
S0	letter	S_ID	IDENTIFIER_CANDIDATE	Semua kata (identifier/key word) mulai dari sini
S0	digit	S_NUM	NUMBER	Semua bilangan dimulai dari sini
S_ID	Letter, digit, _	S_ID	IDENTIFIER_CANDIDATE	Loop untuk mengenali *identifier* atau *keyword* dengan panjang berapapun (termasuk *underscore*)

S_NUM	digit	S_NUM	NUMBER	Loop untuk mengenali bilangan *integer* multigit
S_NUM	.	S_REAL_DOT	Intermediate	Pengenalan Bilangan Real: Jika setelah angka, Lexer melihat . (dot), Lexer beralih dari integer sederhana ke real (angka desimal)
S_DOT	.	S_RANGE	RANGE_OPERATOR	Greedy Match: Jika diikuti . lagi, Lexer memprioritaskan .. daripada ..
S_REAL_DOT	digit	S_REAL	NUMBER	Setelah dot, harus ada digit agar valid. S_REAL adalah final state yang menunjukkan bilangan real

S_REAL	digit	S_REAL	NUMBER	Loop untuk mengenali sisa digit bilangan *real*
S0	‘	S_STRING	*Intermediate*	Pengenalan Literal: Lexer mulai dari tanda kutip pembuka
S0	:	S_COLON	COLON	Pola Majemuk (:=): Lexer berada pada state final (S_COLON)
S_COLON	=	S_ASSIGN	ASSIGN_OPERATOR	Greedy Match: Jika setelah : dilihat =, Lexer pindah ke state yang lebih panjang (S_ASSIGN), memprioritaskan := daripada :
S0	<	S_LT	RELATIONAL_OPERATOR	Pola Majemuk (<=, <>, <): Lexer mulai dari <

S_LT	=	S_LTE	RELATIONAL_OPERATOR	Jika diikuti =, membentuk <=
S_LT	>	S_NEQ	RELATIONAL_OPERATOR	Jika diikuti >, membentuk < >
S_LT	>	S_NEQ	RELATIONAL_OPERATOR	Greedy Match: Jika diikuti >, membentuk <>.
S0	>	S_GT	RELATIONAL_OPERATOR	Pola Relasional (Langkah 1): Lexer berada pada *final state* >.
S_GT	=	S_GT	ERELATIONAL_OPERATOR	Greedy Match: Jika diikuti =, membentuk >=.
S0	'	S_STRING	Intermediate	Pengenalan String/Char Literal: Lexer mulai dari tanda kutip pembuka

S_STRING	any_char	S_STRING	Intermediate	Tetap dalam state ini hingga menemukan '
S_STRING	'	S_STRING_END	STRING_LITERAL	Menemukan tanda kutip penutup, Token selesai
S0	+, -, *, /, =, [,], (,), ,, ;	*Final State* Masing-Masing	ARITHMETIC_OPERATOR, RELATIONAL_OPERATOR, PUNCTUATION	Simbol-simbol ini langsung dikenali sebagai token dengan satu karakter dari S0 dan tidak memiliki transisi lanjutan (kecuali yang terlibat dalam *Greedy Match*)

BAB III
Pengujian

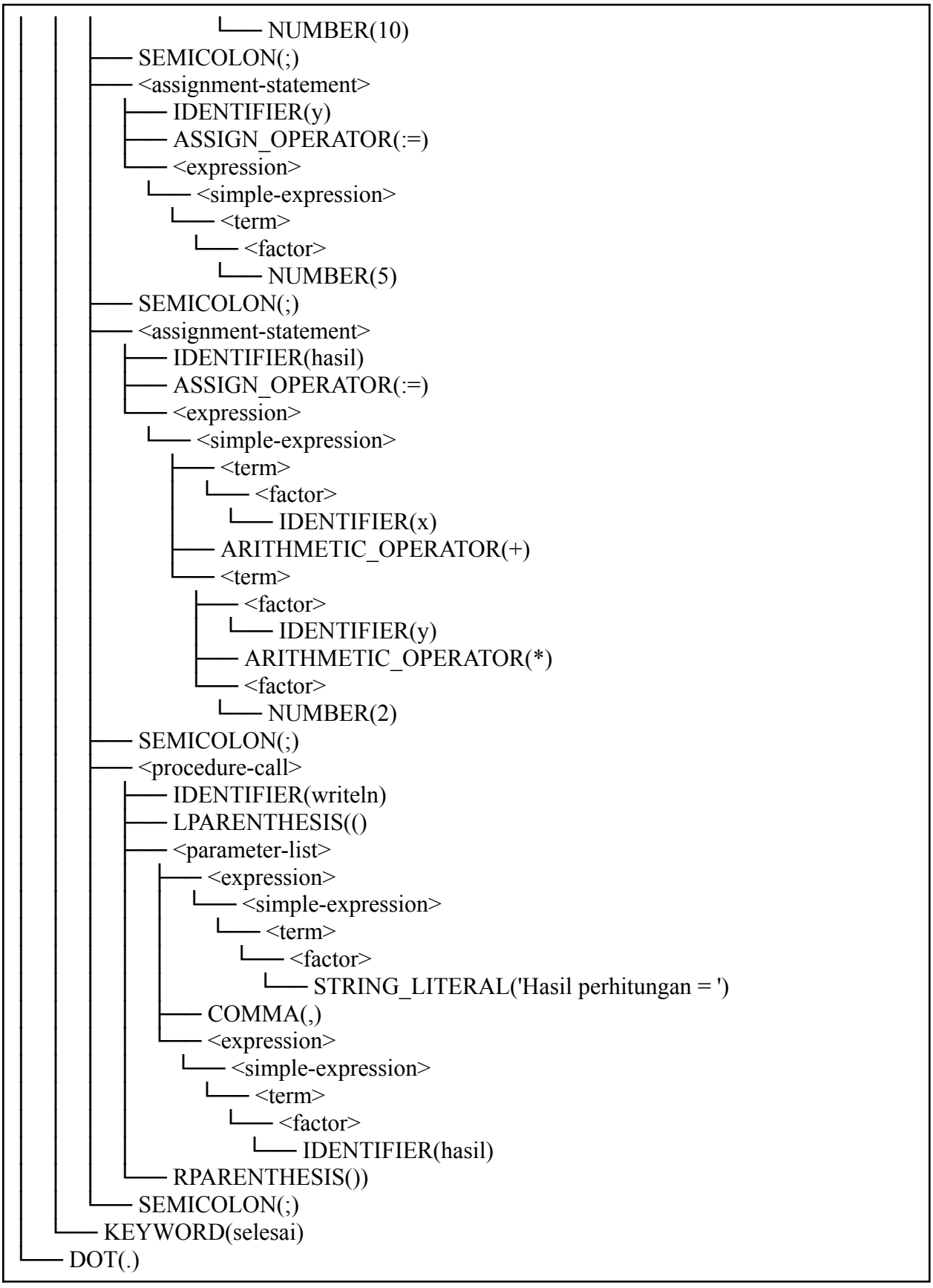
3.1 Testing 1

Input (Kode Paskal)

```
program HitungDasar;

variabel
  x, y, hasil: integer;

mulai
  x := 10;
  y := 5;
  hasil := x + y * 2;
  writeln('Hasil perhitungan = ', hasil);
selesai.
```

Output (File.txt)

```
<program>
├── <program-header>
│   ├── KEYWORD(program)
│   ├── IDENTIFIER(HitungDasar)
│   └── SEMICOLON(;)
├── <declaration-part>
│   └── <var-declaration>
│       ├── KEYWORD(variabel)
│       ├── <identifier-list>
│       │   ├── IDENTIFIER(x)
│       │   ├── COMMA(,)
│       │   ├── IDENTIFIER(y)
│       │   ├── COMMA(,)
│       │   └── IDENTIFIER(hasil)
│       ├── COLON(:)
│       ├── <type>
│       │   └── KEYWORD(integer)
│       └── SEMICOLON(;)
├── <compound-statement>
│   ├── KEYWORD(mulai)
│   ├── <statement-list>
│   │   ├── <assignment-statement>
│   │   │   ├── IDENTIFIER(x)
│   │   │   ├── ASSIGN_OPERATOR(:=)
│   │   │   └── <expression>
│   │   │       └── <simple-expression>
│   │   │           └── <term>
│   │   │               └── <factor>
```


NUMBER(10)
SEMICOLON(;)
<assignment-statement>
IDENTIFIER(y)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
NUMBER(5)
SEMICOLON(;)
<assignment-statement>
IDENTIFIER(hasil)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(x)
ARITHMETIC_OPERATOR(+)
<term>
<factor>
IDENTIFIER(y)
ARITHMETIC_OPERATOR(*)
<factor>
NUMBER(2)
SEMICOLON(;)
<procedure-call>
IDENTIFIER(writeln)
LPARENTHESIS(()
<parameter-list>
<expression>
<simple-expression>
<term>
<factor>
STRING_LITERAL('Hasil perhitungan = ')
COMMA(,)
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(hasil)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(selesai)
DOT(.)

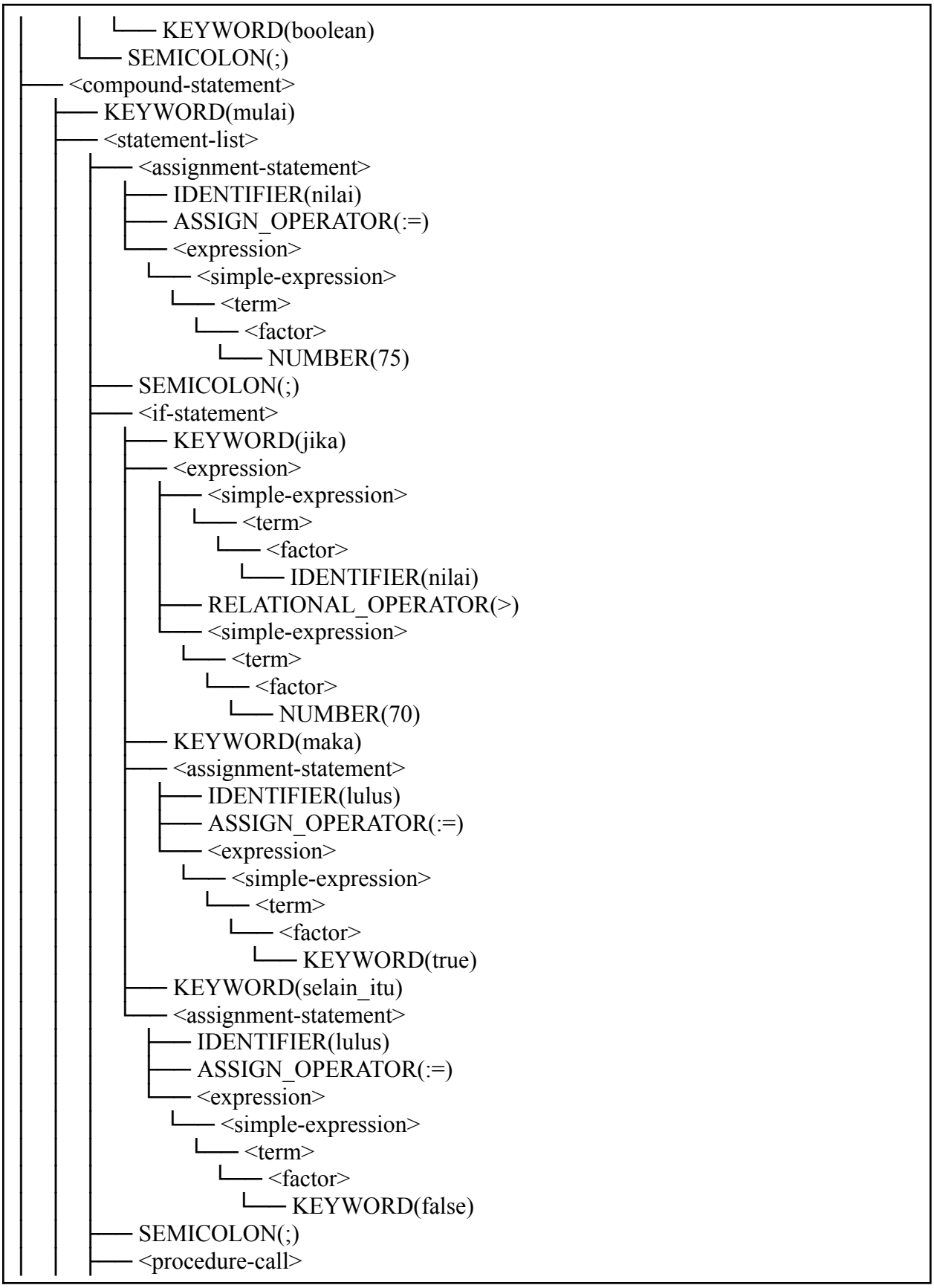
3.2 Testing 2

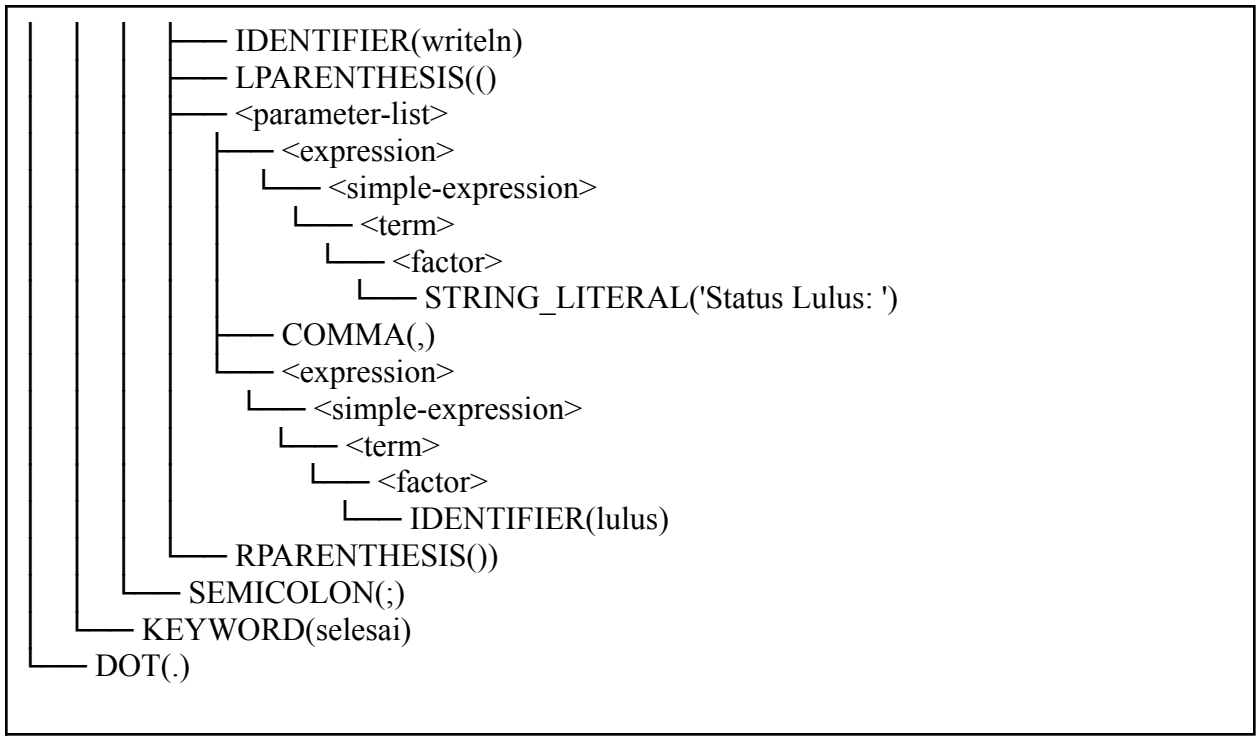
Input (Kode Paskal)

```
program CekNilai;

variabel
  nilai: integer;
  lulus: boolean;

mulai
  nilai := 75;

  jika nilai > 70 maka
    lulus := true
  selain_itu
    lulus := false;

  writeln('Status Lulus: ', lulus);
selesai.
```

Output (File.txt)

```
<program>
├── <program-header>
│   ├── KEYWORD(program)
│   ├── IDENTIFIER(CekNilai)
│   └── SEMICOLON(;)
├── <declaration-part>
│   └── <var-declaration>
│       ├── KEYWORD(variabel)
│       ├── <identifier-list>
│       │   └── IDENTIFIER(nilai)
│       ├── COLON(:)
│       ├── <type>
│       │   └── KEYWORD(integer)
│       ├── SEMICOLON(;)
│       ├── <identifier-list>
│       │   └── IDENTIFIER(lulus)
│       ├── COLON(:)
│       ├── <type>
```


KEYWORD(boolean)
SEMICOLON(;)
<compound-statement>
KEYWORD(mulai)
<statement-list>
<assignment-statement>
IDENTIFIER(nilai)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
NUMBER(75)
SEMICOLON(;)
<if-statement>
KEYWORD(jika)
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(nilai)
RELATIONAL_OPERATOR(>)
<simple-expression>
<term>
<factor>
NUMBER(70)
KEYWORD(maka)
<assignment-statement>
IDENTIFIER(lulus)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
KEYWORD(true)
KEYWORD(selain_itu)
<assignment-statement>
IDENTIFIER(lulus)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
KEYWORD(false)
SEMICOLON(;)
<procedure-call>



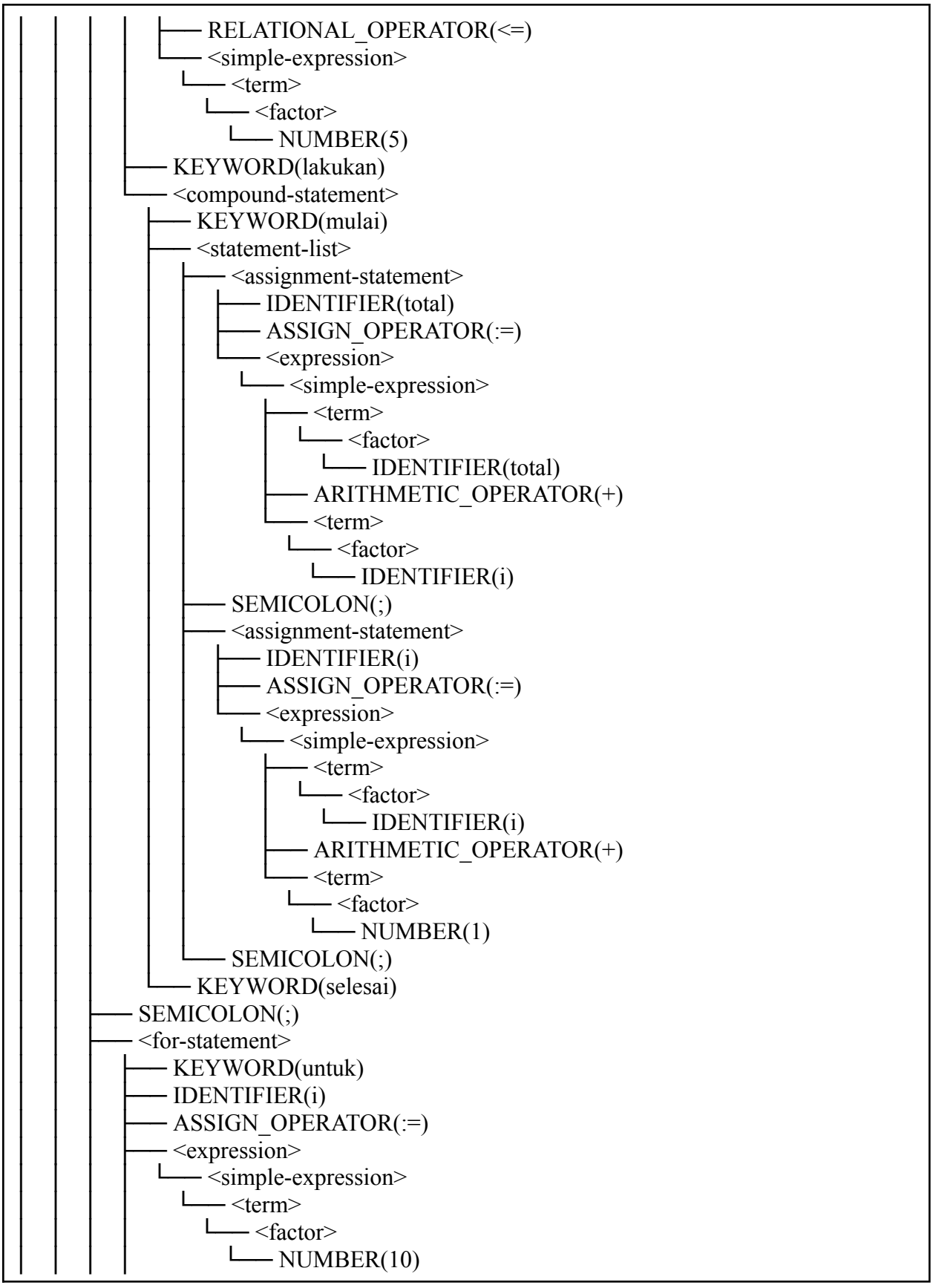
3.3 Testing 3

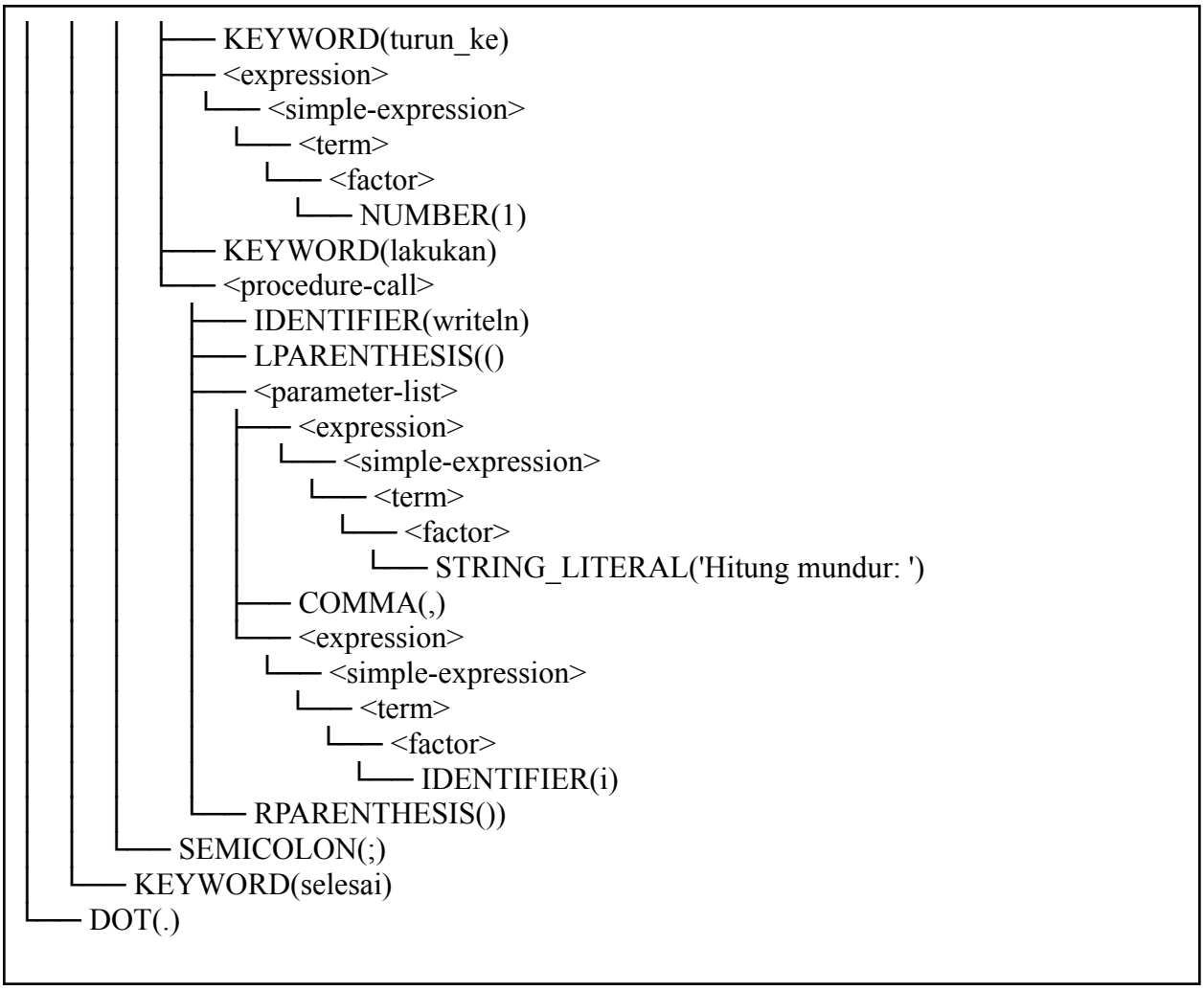
Input (Kode Paskal)

```
program LoopTest;

variabel
  i, total: integer;

mulai
  total := 0;
  i := 1;

  selama i <= 5 lakukan
  mulai
    total := total + i;
    i := i + 1;
  selesai;

  untuk i := 10 turun_ke 1 lakukan
    writeln('Hitung mundur: ', i);
selesai.
```

Output (File.txt)

```
<program>
├── <program-header>
│   ├── KEYWORD(program)
│   ├── IDENTIFIER(LoopTest)
│   └── SEMICOLON(;)
├── <declaration-part>
│   └── <var-declaration>
│       ├── KEYWORD(variabel)
│       ├── <identifier-list>
│       │   ├── IDENTIFIER(i)
│       │   ├── COMMA(,)
│       │   └── IDENTIFIER(total)
│       ├── COLON(:)
│       ├── <type>
│       │   └── KEYWORD(integer)
│       └── SEMICOLON(;)
├── <compound-statement>
│   ├── KEYWORD(mulai)
│   ├── <statement-list>
│   │   ├── <assignment-statement>
│   │   │   ├── IDENTIFIER(total)
│   │   │   ├── ASSIGN_OPERATOR(:=)
│   │   │   └── <expression>
│   │   │       └── <simple-expression>
│   │   │           └── <term>
│   │   │               └── <factor>
│   │   │                   └── NUMBER(0)
│   │   ├── SEMICOLON(;)
│   │   ├── <assignment-statement>
│   │   │   ├── IDENTIFIER(i)
│   │   │   ├── ASSIGN_OPERATOR(:=)
│   │   │   └── <expression>
│   │   │       └── <simple-expression>
│   │   │           └── <term>
│   │   │               └── <factor>
│   │   │                   └── NUMBER(1)
│   │   ├── SEMICOLON(;)
│   │   ├── <while-statement>
│   │   │   ├── KEYWORD(selama)
│   │   │   ├── <expression>
│   │   │   │   ├── <simple-expression>
│   │   │   │   │   └── <term>
│   │   │   │   │       └── <factor>
│   │   │   │   │           └── IDENTIFIER(i)
```


RELATIONAL_OPERATOR(<=)
<simple-expression>
<term>
<factor>
NUMBER(5)
KEYWORD(lakukan)
<compound-statement>
KEYWORD(mulai)
<statement-list>
<assignment-statement>
IDENTIFIER(total)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(total)
ARITHMETIC_OPERATOR(+)
<term>
<factor>
IDENTIFIER(i)
SEMICOLON(;)
<assignment-statement>
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(i)
ARITHMETIC_OPERATOR(+)
<term>
<factor>
NUMBER(1)
SEMICOLON(;)
KEYWORD(selesai)
SEMICOLON(;)
<for-statement>
KEYWORD(untuk)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
NUMBER(10)



3.4 Testing 4

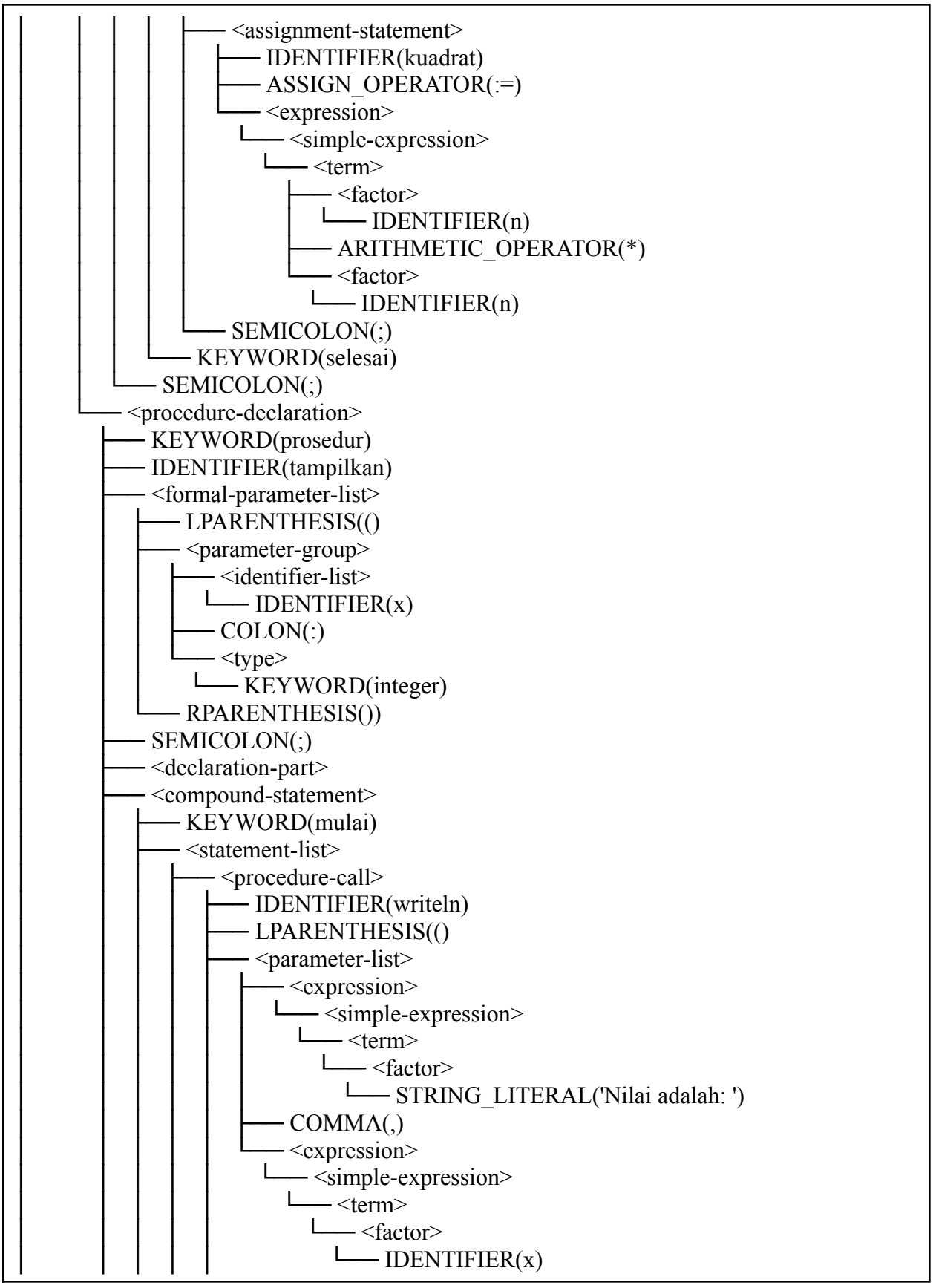
Input (Kode Paskal)

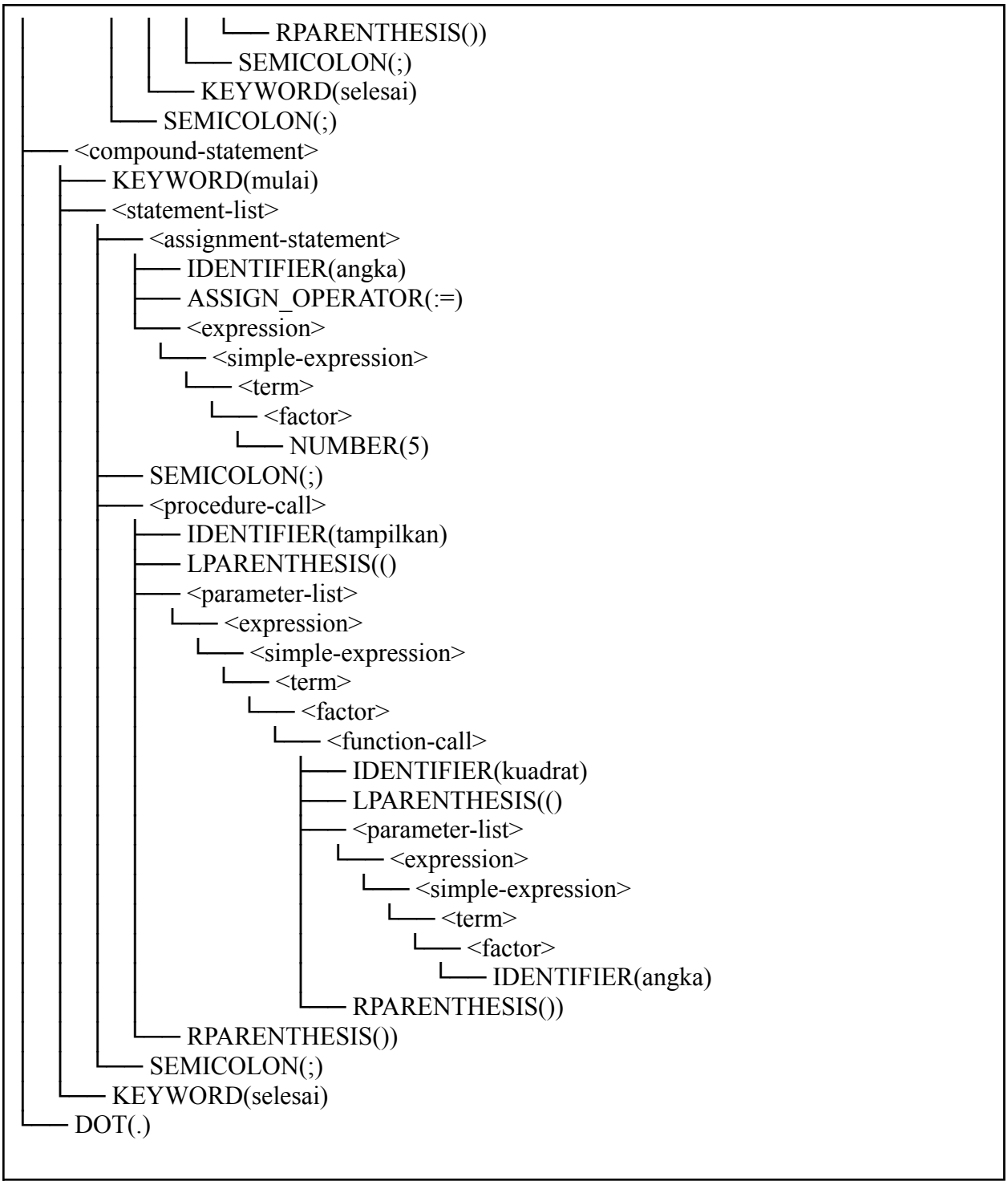
```
program SubProgramTest;

variabel
  angka: integer;

fungsi kuadrat(n: integer): integer;
mulai
  kuadrat := n * n;
selesai;

prosedur tampilkan(x: integer);
mulai
  writeln('Nilai adalah: ', x);
```

```
selesai;

mulai
  angka := 5;
  tampilkan(kuadrat(angka));
selesai.
```

Output (File.txt)

```
<program>
├── <program-header>
│   ├── KEYWORD(program)
│   ├── IDENTIFIER(SubProgramTest)
│   └── SEMICOLON(;)
├── <declaration-part>
│   ├── <var-declaration>
│   │   ├── KEYWORD(variabel)
│   │   ├── <identifier-list>
│   │   │   └── IDENTIFIER(angka)
│   │   ├── COLON(:)
│   │   ├── <type>
│   │   │   └── KEYWORD(integer)
│   │   └── SEMICOLON(;)
│   └── <subprogram-declaration>
│       ├── <function-declaration>
│       │   ├── KEYWORD(fungsi)
│       │   ├── IDENTIFIER(kuadrat)
│       │   ├── <formal-parameter-list>
│       │   │   ├── LPARENTHESIS(()
│       │   │   ├── <parameter-group>
│       │   │   │   ├── <identifier-list>
│       │   │   │   │   └── IDENTIFIER(n)
│       │   │   │   ├── COLON(:)
│       │   │   │   └── <type>
│       │   │   │       └── KEYWORD(integer)
│       │   │   └── RPARENTHESIS())
│       │   ├── COLON(:)
│       │   ├── <type>
│       │   │   └── KEYWORD(integer)
│       │   ├── SEMICOLON(;)
│       │   ├── <declaration-part>
│       │   ├── <compound-statement>
│       │   │   ├── KEYWORD(mulai)
│       │   │   ├── <statement-list>
```


├── <assignment-statement>
│ ├── IDENTIFIER(kuadrat)
│ ├── ASSIGN_OPERATOR(:=)
│ └── <expression>
│ └── <simple-expression>
│ └── <term>
│ ├── <factor>
│ │ └── IDENTIFIER(n)
│ ├── ARITHMETIC_OPERATOR(*)
│ └── <factor>
│ └── IDENTIFIER(n)
└── SEMICOLON(;)
└── KEYWORD(selesai)
└── SEMICOLON(;)
└── <procedure-declaration>
├── KEYWORD(prosedur)
├── IDENTIFIER(tampilkan)
├── <formal-parameter-list>
│ ├── LPARENTHESIS(()
│ ├── <parameter-group>
│ │ ├── <identifier-list>
│ │ │ └── IDENTIFIER(x)
│ │ ├── COLON(:)
│ │ └── <type>
│ │ └── KEYWORD(integer)
│ └── RPARENTHESIS())
├── SEMICOLON(;)
├── <declaration-part>
└── <compound-statement>
├── KEYWORD(mulai)
├── <statement-list>
│ ├── <procedure-call>
│ │ ├── IDENTIFIER(writeln)
│ │ ├── LPARENTHESIS(()
│ │ ├── <parameter-list>
│ │ │ ├── <expression>
│ │ │ │ └── <simple-expression>
│ │ │ │ └── <term>
│ │ │ │ └── <factor>
│ │ │ │ └── STRING_LITERAL('Nilai adalah: ')
│ │ │ ├── COMMA(,)
│ │ │ └── <expression>
│ │ │ └── <simple-expression>
│ │ │ └── <term>
│ │ │ └── <factor>
│ │ │ └── IDENTIFIER(x)



3.5 Testing 5

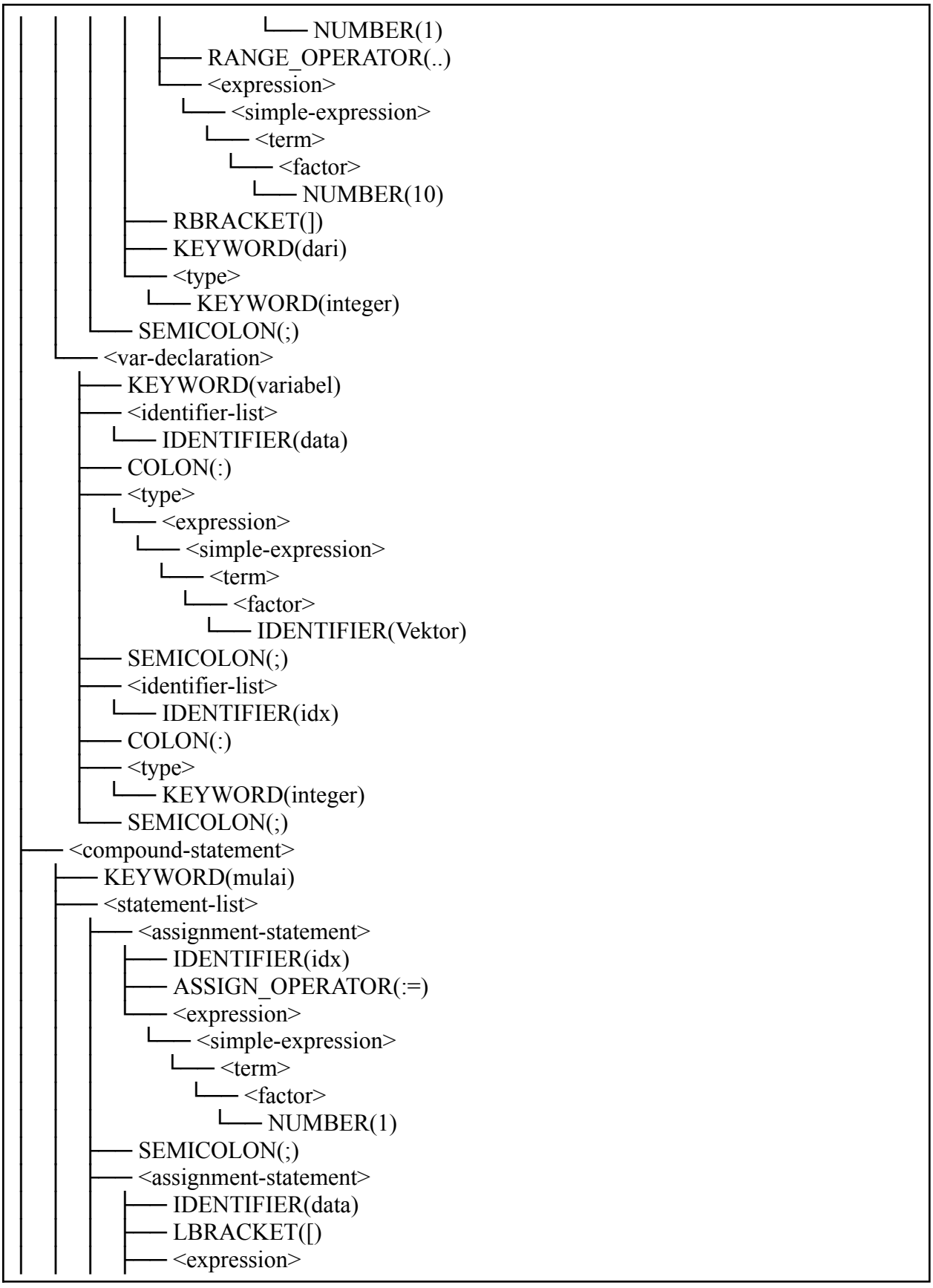
Input (Kode Paskal)

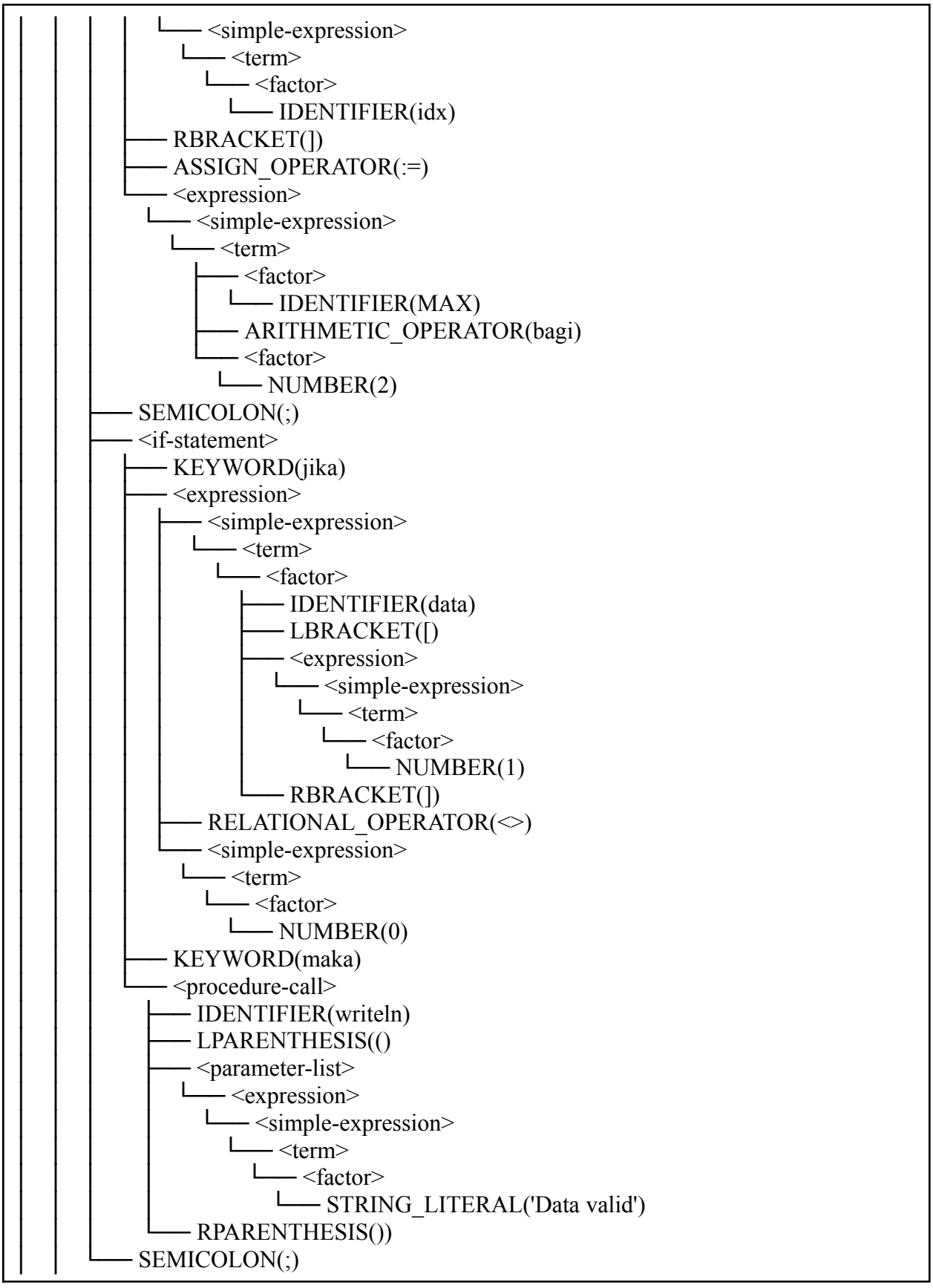
```
program ArrayKompleks;
```

```
konstanta
  MAX = 100;

tipe
  Vektor = larik [1..10] dari integer;

variabel
  data: Vektor;
  idx: integer;

mulai
  idx := 1;
  data[idx] := MAX bagi 2;

  jika data[1] <> 0 maka
    writeln('Data valid');
selesai.
```

Output (File.txt)

```
<program>
├── <program-header>
│   ├── KEYWORD(program)
│   ├── IDENTIFIER(ArrayKompleks)
│   └── SEMICOLON(;)
└── <declaration-part>
    ├── <const-declaration>
    │   ├── KEYWORD(konstanta)
    │   ├── IDENTIFIER(MAX)
    │   ├── RELATIONAL_OPERATOR(=)
    │   ├── NUMBER(100)
    │   └── SEMICOLON(;)
    ├── <type-declaration>
    │   ├── KEYWORD(tipe)
    │   ├── IDENTIFIER(Vektor)
    │   ├── RELATIONAL_OPERATOR(=)
    │   ├── <type>
    │   │   ├── KEYWORD(larik)
    │   │   ├── LBRACKET([)
    │   │   ├── <range>
    │   │   │   ├── <expression>
    │   │   │   │   └── <simple-expression>
    │   │   │   │       └── <term>
    │   │   │   │           └── <factor>
```


NUMBER(1)
RANGE_OPERATOR(..)
<expression>
<simple-expression>
<term>
<factor>
NUMBER(10)
RBRACKET(])
KEYWORD(dari)
<type>
KEYWORD(integer)
SEMICOLON(;)
<var-declaration>
KEYWORD(variabel)
<identifier-list>
IDENTIFIER(data)
COLON(:)
<type>
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(Vektor)
SEMICOLON(;)
<identifier-list>
IDENTIFIER(idx)
COLON(:)
<type>
KEYWORD(integer)
SEMICOLON(;)
<compound-statement>
KEYWORD(mulai)
<statement-list>
<assignment-statement>
IDENTIFIER(idx)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
NUMBER(1)
SEMICOLON(;)
<assignment-statement>
IDENTIFIER(data)
LBRACKET([)
<expression>


└── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(idx)
├── RBRACKET(])
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
└── <term>
├── <factor>
└── IDENTIFIER(MAX)
├── ARITHMETIC_OPERATOR(bagi)
└── <factor>
└── NUMBER(2)
├── SEMICOLON(;)
├── <if-statement>
├── KEYWORD(jika)
├── <expression>
├── <simple-expression>
└── <term>
└── <factor>
├── IDENTIFIER(data)
├── LBRACKET([)
├── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(1)
└── RBRACKET(])
├── RELATIONAL_OPERATOR(<>)
└── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(0)
├── KEYWORD(maka)
└── <procedure-call>
├── IDENTIFIER(writeln)
├── LPARENTHESIS(()
├── <parameter-list>
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── STRING_LITERAL('Data valid')
└── RPARENTHESIS())
└── SEMICOLON(;)

```
    └── KEYWORD(selesai)
└── DOT(.)
```

BAB IV
Kesimpulan dan Saran

4.1 Kesimpulan

Pada pengerjaan Milestone 2 ini, *Syntax Analyzer* (Parser) dengan metode Recursive Descent berhasil diimplementasikan untuk bahasa PASCAL-S. *Parser* ini berfungsi sebagai tahapan

kedua dalam alur kompilator, yang mengambil *list of tokens* dari *Lexer* (Milestone 1) sebagai input dan menghasilkan **Parse Tree** sebagai output.

Berdasarkan serangkaian pengujian yang telah dilakukan, *parser* yang kami buat telah terbukti mampu:

1. **Mem-parsing Program Lengkap:** Berhasil mengurai struktur program PASCAL-S dari program hingga selesai.
2. **Menangani Berbagai Deklarasi:** Mampu mem-parsing semua blok deklarasi, termasuk konstanta, tipe, variabel, larik (array), fungsi, dan prosedur.
3. **Mengurai Struktur Kontrol:** Berhasil mem-parsing *statement* jika-maka-selain-itu, selama-lakukan, dan untuk-ke/turun_ke, termasuk blok mulai-selesai yang bersarang di dalamnya.
4. **Mengelola Prioritas Operator:** Implementasi rantai pemanggilan expression(), simple_expression(), term(), dan factor() berhasil menangani prioritas operator (misalnya, perkalian dievaluasi sebelum penjumlahan pada Test 3.1).
5. **Menangani Kasus Kompleks:** Berhasil mengurai sintaks yang rumit seperti pemanggilan fungsi yang bersarang di dalam pemanggilan prosedur (tampilkan(kuadrat(angka)) pada Test 3.4) dan penggunaan akses *array* (data[idx]) baik sebagai target *assignment* maupun di dalam *expression*.

Implementasi *error handling* melalui fungsi expect() juga memastikan bahwa *parser* akan berhenti dan memberikan pesan yang informatif jika menemukan urutan *token* yang tidak sesuai dengan *grammar*.

4.2 Saran

Meskipun *parser* ini telah berfungsi sesuai spesifikasi Milestone 2, terdapat beberapa pengembangan yang dapat dilakukan untuk mempersiapkan Milestone 3:

1. **Transformasi ke Abstract Syntax Tree (AST):** *Parse Tree* yang dihasilkan saat ini bersifat sangat konkret dan detail (mencakup setiap titik koma, kurung, dll.). Untuk tahap *Semantic Analysis*, akan jauh lebih efisien jika *Parse Tree* ini diubah menjadi Abstract Syntax Tree (AST). AST adalah representasi yang lebih ringkas dan hanya berfokus pada makna program (misalnya, *node* IfStatement memiliki tiga anak: Kondisi, BlokMaka, BlokSelainItu), sehingga mempermudah proses *type checking* dan analisis *scope*.
2. **Pengembangan *Error Recovery*:** Saat ini, *parser* akan berhenti total (raise SyntaxError) pada kesalahan sintaks pertama yang ditemukannya. Untuk pengembangan lebih lanjut, dapat dipertimbangkan implementasi *error recovery*. Dengan teknik ini, ketika *parser* menemukan *error*, ia akan mencoba "sinkronisasi" (misalnya, meloncat ke *token* SEMICOLON atau selesai berikutnya) dan melanjutkan *parsing*. Hal ini memungkinkan *compiler* untuk melaporkan lebih dari satu kesalahan sintaks dalam satu kali kompilasi.

BAB V
Lampiran

5.1 Link Repository Github

https://github.com/iannn23/HJE-Tubes-IF2224.git

5.2 Link release Github

https://github.com/iannn23/HJE-Tubes-IF2224/releases/tag/v0.2.1

5.2 Link Workspace Diagram DFA

https://drive.google.com/file/d/1rB25CckCak9_7L3TSquK9DLXx0BWqna2/view?usp=sharing

5.3 Pembagian Tugas

Nama	NIM	Tugas
Sebastian Enrico Nathanael	13523134	Laporan, testing
Jonathan Kenan Budianto	13523139	Implementasi code, testing
Mahesa Fadhillah Andre	13523140	Implementasi code, testing
Muhammad Farrel Wibowo	13523153	Implementasi code, testing

5.4 Aturan Grammar (BNF)

1. Struktur Program Utama

```
<program> ::= <program-header> <declaration-part> <compound-statement> DOT
```

```
<program-header> ::= KEYWORD(program) IDENTIFIER SEMICOLON
```

2. Bagian Deklarasi

```
<declaration-part> ::= ( <const-declaration> | <type-declaration> | <var-declaration> |
<subprogram-declaration> )*
```

```
<const-declaration> ::= KEYWORD(konstanta) ( IDENTIFIER
RELATIONAL_OPERATOR(=) NUMBER SEMICOLON )+
```

```
<type-declaration> ::= KEYWORD(tipe) ( IDENTIFIER
RELATIONAL_OPERATOR(=) <type-spec> SEMICOLON )+
```

```
<var-declaration> ::= KEYWORD(variabel) ( <identifier-list> COLON <type-spec>
```

```
SEMICOLON )+
```

```
<identifier-list> ::= IDENTIFIER ( COMMA IDENTIFIER )*
```

3. Deklarasi Tipe (Type)

```
<type-spec> ::= KEYWORD(integer)
        | KEYWORD(real)
        | KEYWORD(boolean)
        | KEYWORD(char)
        | <array-type>
        | <subrange-type>
        | IDENTIFIER
```

```
<array-type> ::= KEYWORD(larik) LBRACKET <range-spec> RBRACKET
KEYWORD(dari) <type-spec>
```

```
<subrange-type> ::= <expression> RANGE_OPERATOR(..) <expression>
```

```
<range-spec> ::= <expression> RANGE_OPERATOR(..) <expression>
```

4. Deklarasi Subprogram (Fungsi & Prosedur)

```
<subprogram-declaration> ::= ( <procedure-declaration> | <function-declaration> )+
```

```
<procedure-declaration> ::= KEYWORD(prosedur) IDENTIFIER (
<formal-parameter-list> )? SEMICOLON <declaration-part> <compound-statement>
SEMICOLON
```

```
<function-declaration> ::= KEYWORD(fungsi) IDENTIFIER (
<formal-parameter-list> )? COLON <type-spec> SEMICOLON <declaration-part>
<compound-statement> SEMICOLON
```

```
<formal-parameter-list> ::= LPARENTHESIS <parameter-group> ( SEMICOLON
<parameter-group> )* RPARENTHESIS
```

```
<parameter-group> ::= <identifier-list> COLON <type-spec>
```

5. Statement (Pernyataan)

```
<compound-statement> ::= KEYWORD(mulai) <statement-list> KEYWORD(selesai)
```

```
<statement-list> ::= <statement> ( SEMICOLON <statement> )*
```

```
<statement> ::= <compound-statement>
        | <if-statement>
        | <while-statement>
        | <for-statement>
        | <assignment-statement>
        | <procedure-call>
        | <empty-statement>
```

```
<empty-statement> ::= (* representasi dari SEMICOLON yang tidak diikuti statement lain *)
```

```
<assignment-statement> ::= IDENTIFIER ( LBRACKET <expression> RBRACKET )? ASSIGN_OPERATOR(:=) <expression>
```

```
<if-statement> ::= KEYWORD(jika) <expression> KEYWORD(maka) <statement> ( KEYWORD(selain_itu) <statement> )?
```

```
<while-statement> ::= KEYWORD(selama) <expression> KEYWORD(lakukan) <statement>
```

```
<for-statement> ::= KEYWORD(untuk) IDENTIFIER ASSIGN_OPERATOR(:=) <expression> ( KEYWORD(ke) | KEYWORD(turun_ke) ) <expression> KEYWORD(lakukan) <statement>
```

6. Pemanggilan Fungsi & Prosedur

```
<procedure-call> ::= IDENTIFIER LPARENTHESIS ( <parameter-list> )? RPARENTHESIS
```

```
<function-call> ::= IDENTIFIER LPARENTHESIS ( <parameter-list> )?
RPARENTHESIS
```

```
<parameter-list> ::= <expression> ( COMMA <expression> )*
```

7. Ekspresi (Expressions) & Prioritas Operator

```
<expression> ::= <simple-expression> ( <relational-operator> <simple-expression> )?
```

```
<simple-expression> ::= ( ARITHMETIC_OPERATOR(+) |
ARITHMETIC_OPERATOR(-) )? <term> ( <additive-operator> <term> )*
```

```
<term> ::= <factor> ( <multiplicative-operator> <factor> )*
```

```
<factor> ::= <function-call>
        | IDENTIFIER ( LBRACKET <expression> RBRACKET )
        | IDENTIFIER
        | NUMBER
        | STRING_LITERAL
        | CHAR_LITERAL
        | KEYWORD(true)
        | KEYWORD(false)
        | LOGICAL_OPERATOR(tidak) <factor>
        | LPARENTHESIS <expression> RPARENTHESIS
```

8. Definisi Operator

```
<relational-operator> ::= RELATIONAL_OPERATOR
(* mencakup: =, <>, <, <=, >, >= *)
```

```
<additive-operator> ::= ARITHMETIC_OPERATOR(+) |
ARITHMETIC_OPERATOR(-) | LOGICAL_OPERATOR(atau)
```

```
<multiplicative-operator> ::= ARITHMETIC_OPERATOR(*) |
```

ARITHMETIC_OPERATOR(/) | ARITHMETIC_OPERATOR(bagi) |
ARITHMETIC_OPERATOR(mod) | LOGICAL_OPERATOR(dan)

5.5 Syntax Diagram

Notes

1. Round boxes denote symbols of the language Pascal, rectangular boxes denote syntactic constructs represented by diagrams.

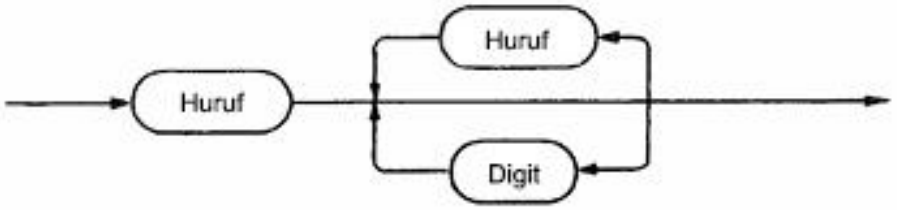
2. Separators may be inserted between any two symbols. However, no separators must occur within numbers and identifiers.

3. At least one separator must occur between consecutive identifiers, numbers, and word-symbols (such as BEGIN,END).

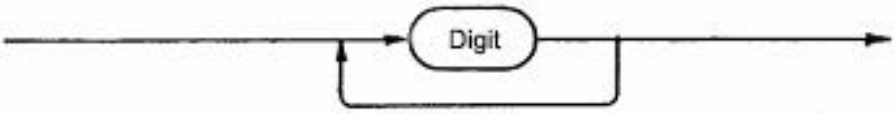
4. Separators are blanks, ends of lines, and comments. A comment is an arbitrary sequence of characters enclosed within a pair of comment brackets (* and *) .

5. The occurrence of the non-qualified word identifier in a syntax diagram implies that at this point an arbitrary, new identifier may be chosen. This identifier thereby becomes a constant-, type-, variable-, field-, function-, or procedure identifier.

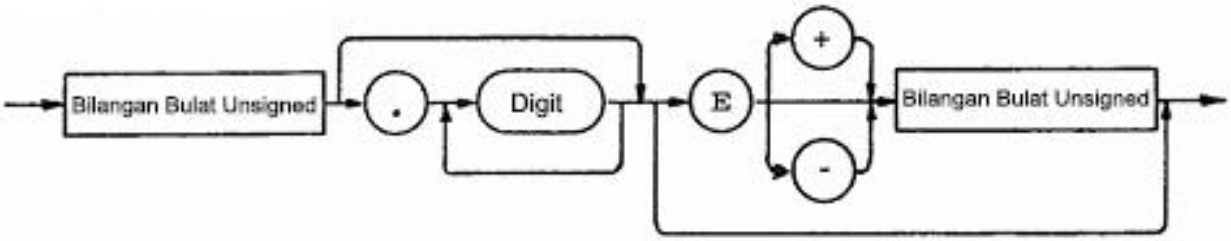
Pengenal



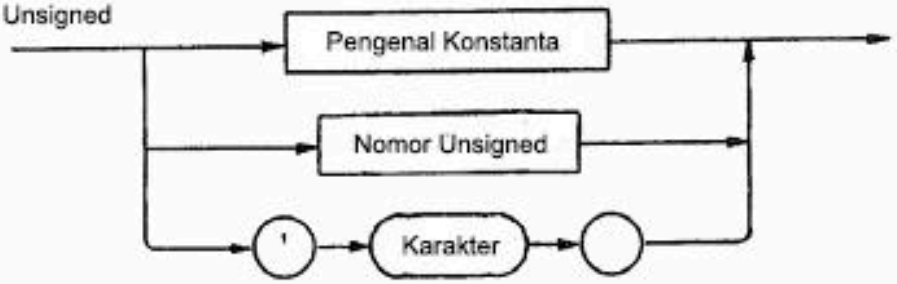
Bilangan Bulat Unsigned



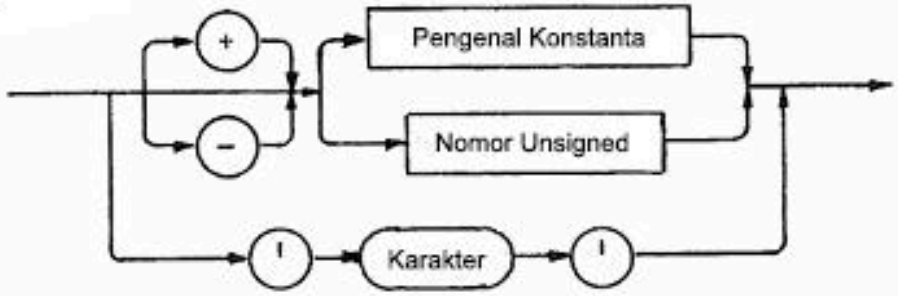
Angka Unsigned



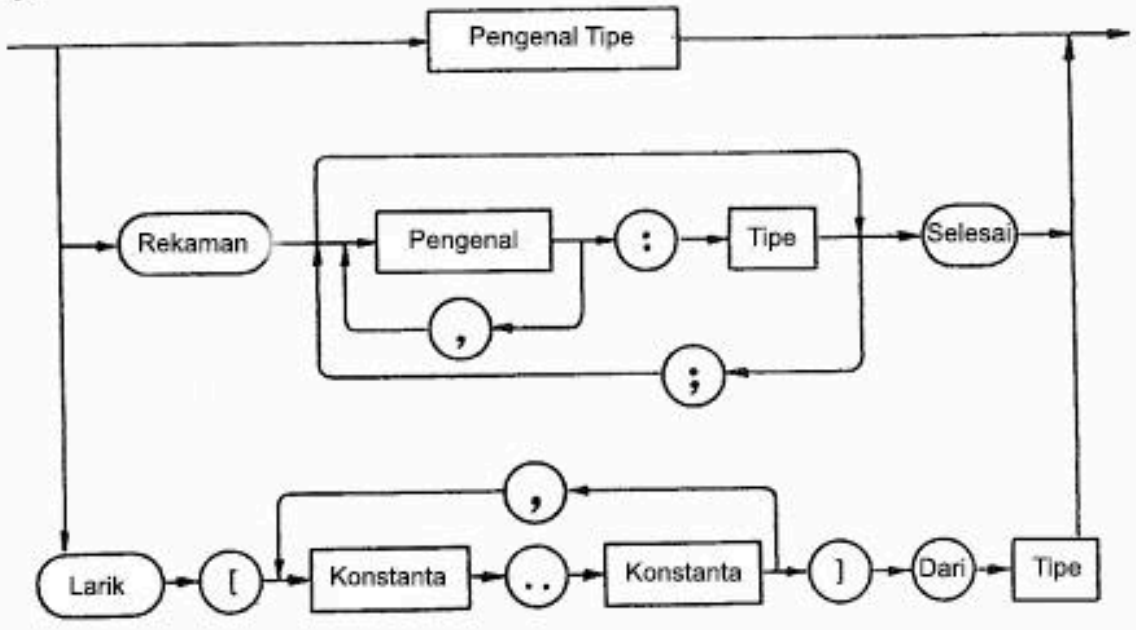
Konstanta Unsigned



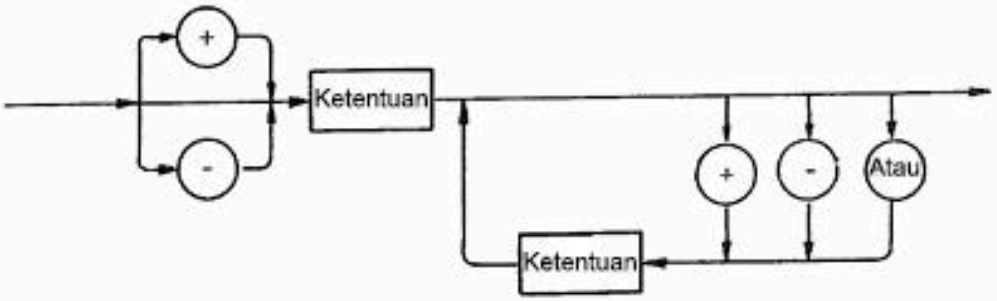
Konstanta



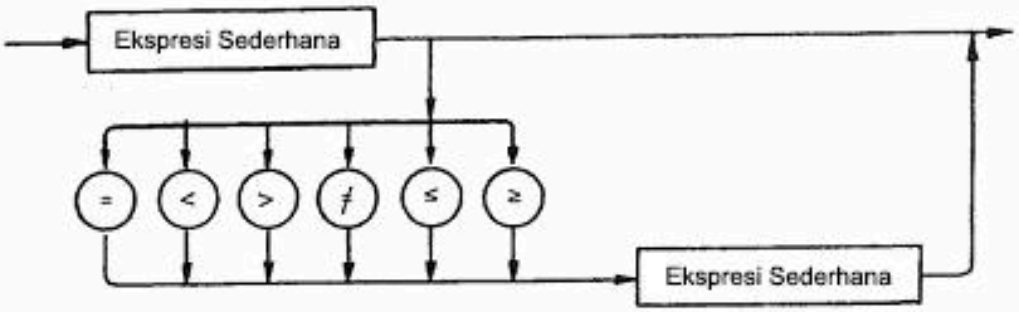
Tipe



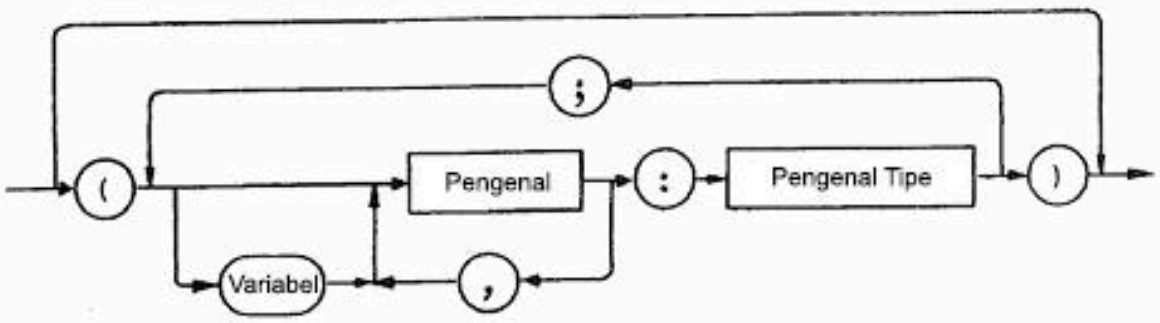
Ekspresi Sederhana



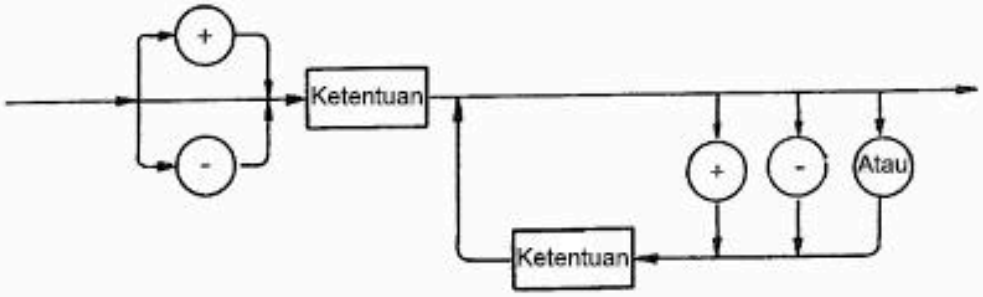
Ekspresi



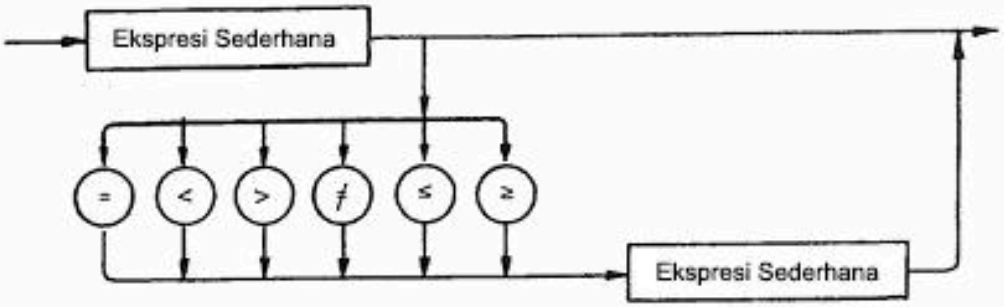
Daftar Parameter



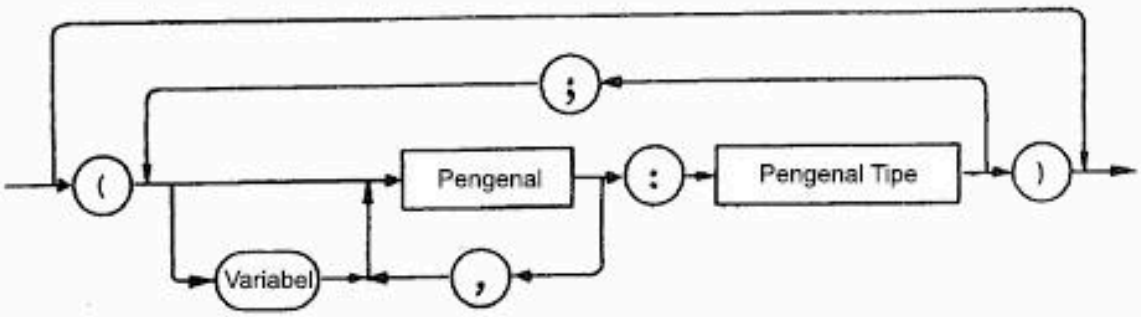
Ekspresi Sederhana



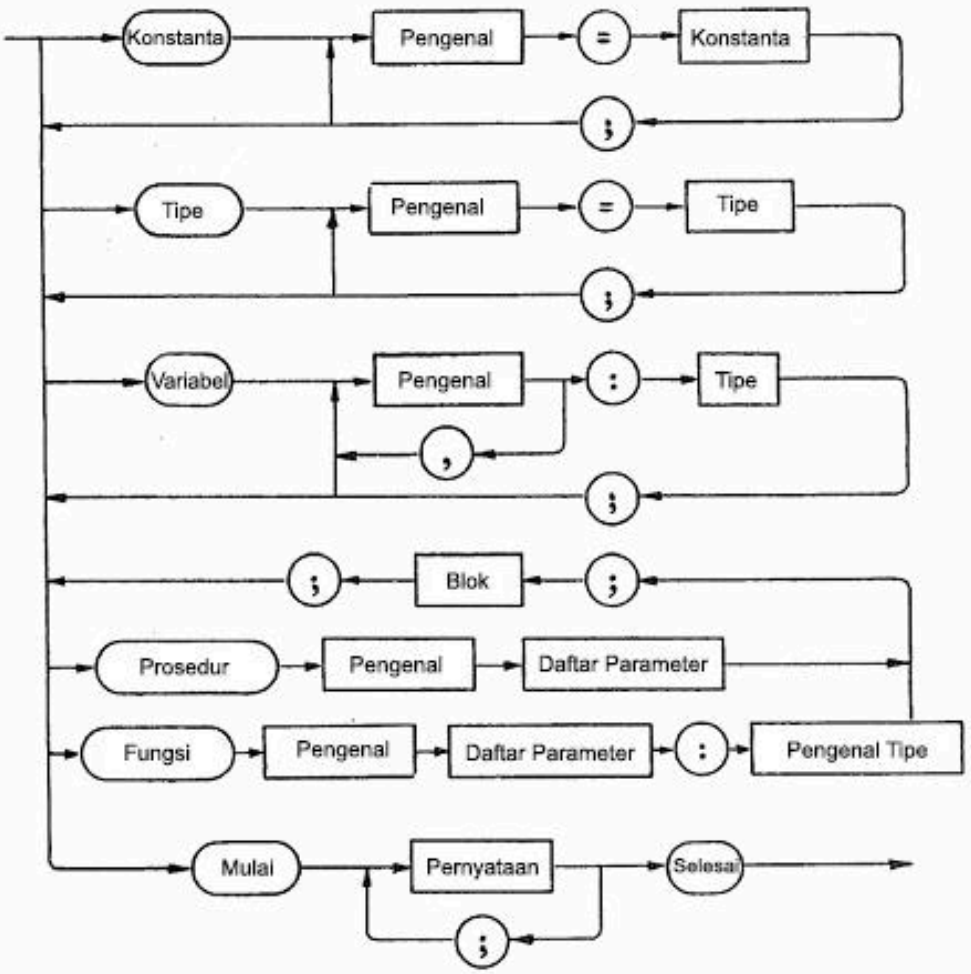
Ekspresi



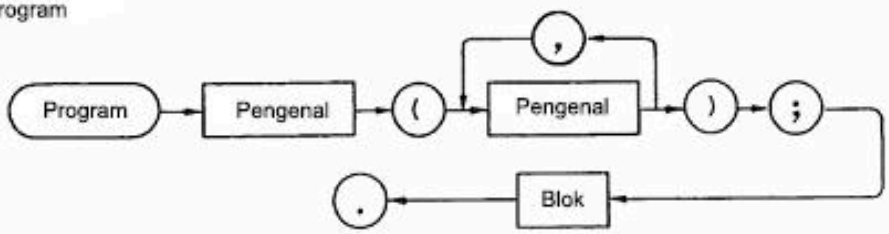
Daftar Parameter



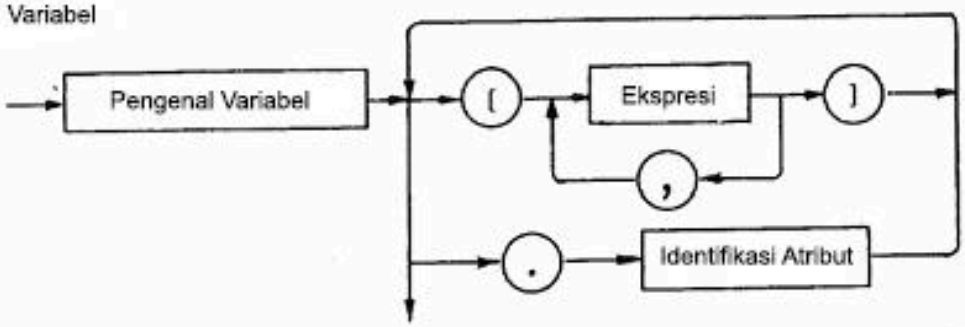
Blok



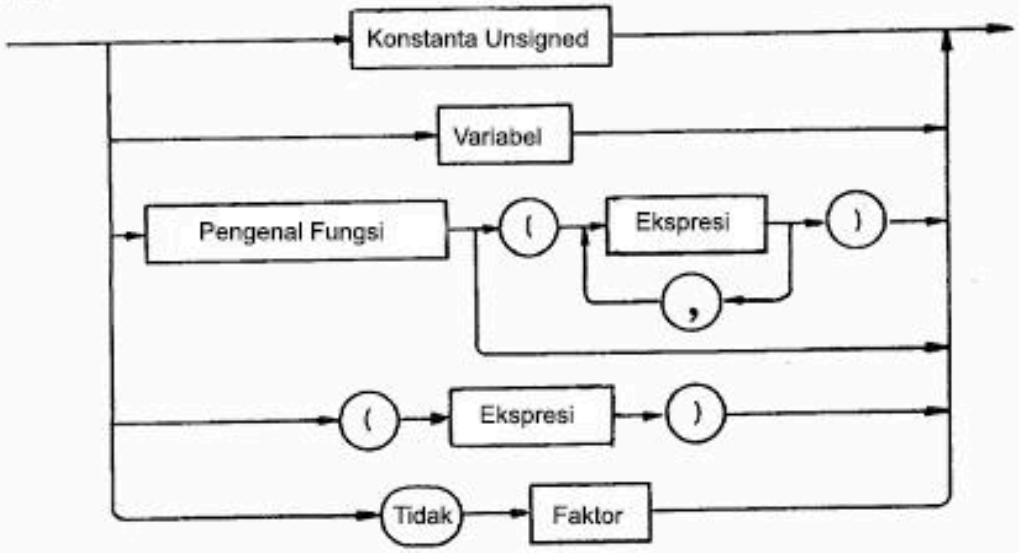
Program



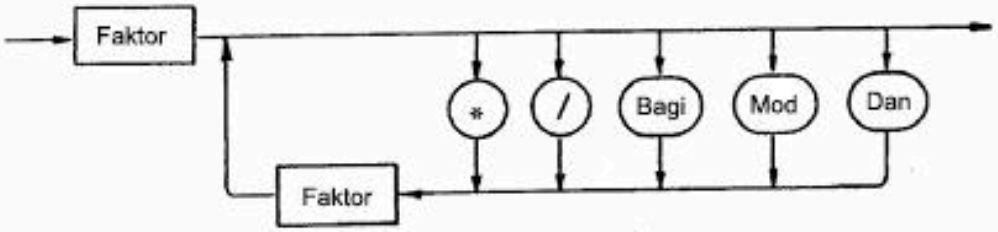
Variabel

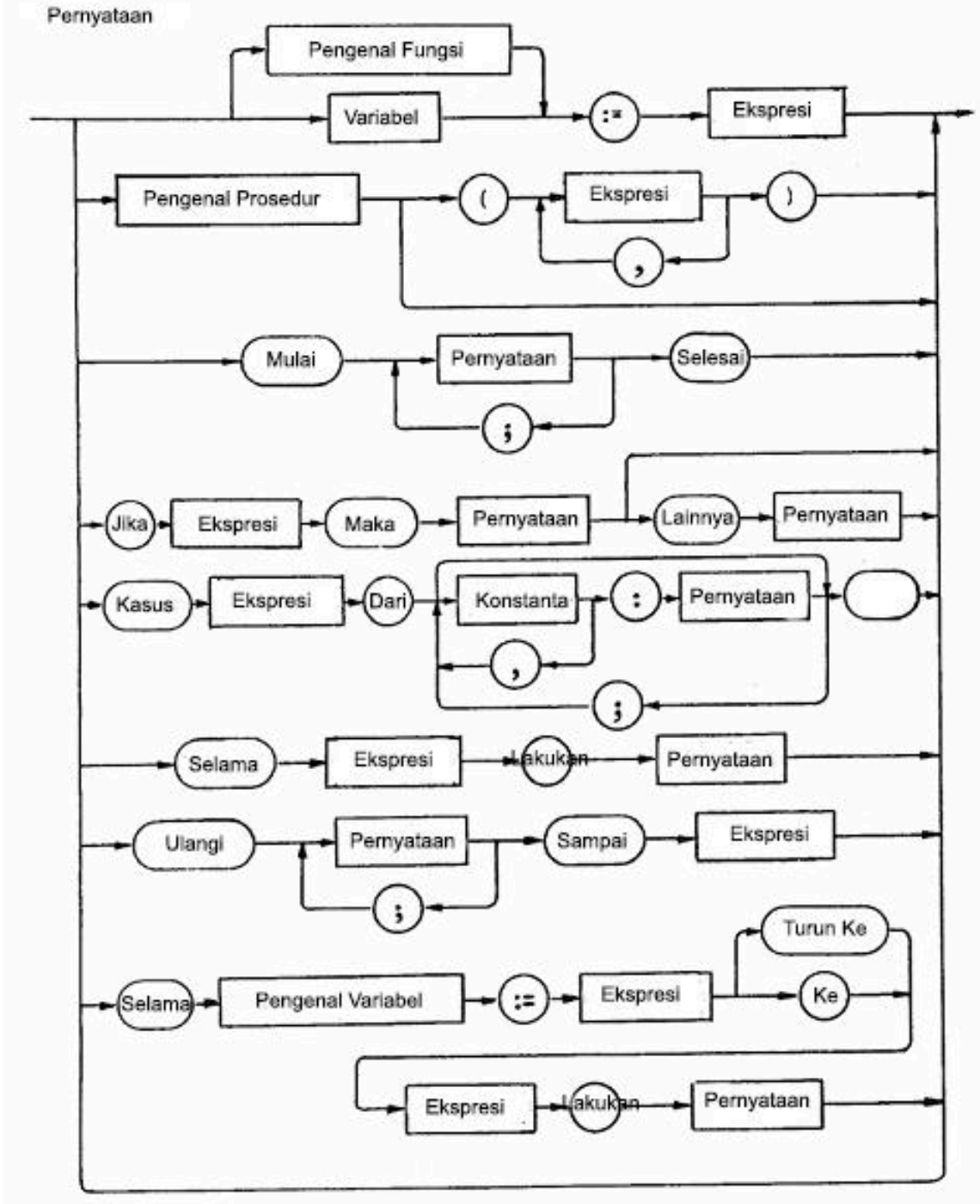


Faktor



Ketentuan




Pernyataan
Pengenal Fungsi
Variabel
:=
Ekspresi
Pengenal Prosedur
(
Ekspresi
,
)
Mulai
Pernyataan
;
Selesai
Jika
Ekspresi
Maka
Pernyataan
Lainnya
Pernyataan
Kasus
Ekspresi
Dari
Konstanta
:
Pernyataan
,
;
Selama
Ekspresi
Lakukan
Pernyataan
Ulangi
Pernyataan
;
Sampai
Ekspresi
Turun Ke
Selama
Pengenal Variabel
:=
Ekspresi
Ke
Ekspresi
Lakukan
Pernyataan

BAB VI
Referensi

- Aho, Alfred V., Sethi, Ravi, dan Ullman, Jeffrey D. Compilers. Principles, Techniques, and Tools (2006). Prentice Hall.
- Hopcroft, John E., Motwani, Rajeev, dan Ullman, Jeffrey D. Introduction to Automata Theory, Languages, and Computation,. Edisi ke-3.
- Jensen, Kathleen, dan Wirth, Niklaus. PASCAL-S: A Subset and its implementation.
Tutorialspoint. Compiler Design - Lexical Analysis.
- Institut Teknologi Bandung. Spesifikasi Tugas Besar IF2224 - Formal Language and Automata Theory, Milestone 1. https://docs.google.com/document/d/1w0GmHW5L0gKZQWbgmtJPFmOzlpSWBknNPdugucn4eII/edit?tab=t.0
- GeeksforGeeks. "Introduction of Lexical Analysis.". https://www.geeksforgeeks.org/compiler-design/introduction-of-lexical-analysis/
- TutorialsPoint. "Compiler Design - Lexical Analysis.". https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm
- “PASCAL-S: A Subset and its implementation”. http://pascal.hansotten.com/uploads/pascals/PASCAL-S%20A%20subset%20and%20its%20Implementation%20012.pdf
- “Let’s Build A Simple Interpreter. Part 7: Abstract Syntax Trees” https://ruslanspivak.com/lsbasi-part7
- “Contoh Gambar Parse Trees” https://textx.github.io/Arpeggio/latest/images/calc_parse_tree.dot.png
- Spesifikasi dan Dokumen Tugas Besar IF2224 TBFO. https://docs.google.com/document/d/1dzZKVdEjTrXSutzDch2dXB5dbkM6w2DJawScLp4VhYI/edit?tab=t.0, https://docs.google.com/document/d/1G_pC2dltQ5Q-iQ3xOpmLIl3XWLDcLfodN8jdiaJBA0Q/edit?tab=t.0#heading=h.w0hmbxg8awp6