Laporan Tugas Besar Milestone 2

IF2224 - Teori Bahasa Formal Otomata 2025/2026

Disusun oleh :

Kelompok AutoRejectAtas

Muhammad Syarafi Akmal	13522076
Kenneth Poenadi	13523040
Bob Kunanda	13523086
M. Zahran Ramadhan Ardiana	13523104

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

JL. GANESHA 10, BANDUNG 40132

2025

Daftar Isi

BAB 1
Landasan Teori

1.1. Tahapan dalam Proses Kompilasi

Kompilasi merupakan proses penerjemahan kode sumber (*source code*) dari bahasa pemrograman tingkat tinggi menjadi bentuk yang dapat dijalankan oleh mesin. Proses ini dilakukan secara bertahap melalui beberapa komponen utama yang dikenal sebagai compiler phases. Secara umum, tahap-tahap tersebut terdiri dari:

1. Lexical Analysis (Analisis Leksikal)

   Bertugas mengenali urutan karakter dalam source code dan mengelompokkan mereka menjadi unit-unit bermakna yang disebut token, seperti keyword, identifier, operator, literal, atau delimiter. Tahapan ini menghasilkan list of tokens yang akan menjadi masukan bagi tahap berikutnya.

   Contoh:

```
variabel a, b: integer;
a := 5;
```

   Akan diterjemahkan menjadi:

```
KEYWORD(variabel)
IDENTIFIER(a)
COMMA(,)
IDENTIFIER(b)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(a)
ASSIGN_OPERATOR(:=)
NUMBER(5)
```

```
SEMICOLON(;)
```

Fungsi utamanya:

- Menghapus spasi, tab, dan komentar yang tidak relevan.
- Mengidentifikasi token valid sesuai aturan regular expression.
- Menghasilkan urutan token (token stream) untuk tahap parser.
- Melaporkan kesalahan leksikal jika ada simbol yang tidak dikenal.

Lexical analyzer bekerja berdasarkan Finite Automata dan Regular Expression (RE). Setiap pola token (misalnya angka, identifier, atau operator) direpresentasikan dalam bentuk RE yang diterjemahkan menjadi Deterministic Finite Automaton (DFA) untuk proses pencocokan karakter. Keluaran dari tahap ini berupa list of tokens yang akan menjadi masukan bagi Syntax Analyzer.

2. Syntax Analysis (Analisis Sintaksis)

Tahapan ini memeriksa apakah urutan token yang dihasilkan oleh lexer membentuk struktur yang valid secara sintaksis sesuai aturan tata bahasa (*grammar*) bahasa pemrograman. Aturan tersebut biasanya didefinisikan dalam bentuk Context-Free Grammar (CFG).

Fungsi Utama:

- Membangun struktur hierarkis dari program sumber dalam bentuk *Parse Tree* atau *Abstract Syntax Tree* (AST).
- Mendeteksi dan melaporkan *syntax error*.
- Menjadi dasar bagi tahap Semantic Analysis.

Parser bekerja dengan prinsip teori automata pushdown (PDA), yang mampu mengenali bahasa bebas konteks (Context-Free Languages).

Contoh CFG sederhana untuk Pascal-S:

```
<program> → <program-header> <declaration-part>
<compound-statement> DOT
<program-header> → KEYWORD(program) IDENTIFIER SEMICOLON
```

Parser dapat dibagi menjadi dua kelompok besar:

- Top-Down Parsing: membangun pohon sintaks dari akar ke daun (contoh: Recursive Descent, LL Parser).
- Bottom-Up Parsing: membangun pohon sintaks dari daun ke akar (contoh: LR, SLR, LALR Parser).

Keluaran dari tahap ini adalah Parse Tree yang menunjukkan struktur program secara hierarkis dan menjadi masukan bagi Semantic Analyzer.

3. Semantic Analysis (Analisis Semantik)

Tahapan ini memeriksa makna logis (semantik) dari program yang telah dinyatakan valid secara sintaks. Parser memastikan struktur benar, sementara semantic analyzer memastikan isi struktur bermakna dan konsisten secara semantik.

Fungsi Utama:

- Memastikan bahwa setiap variabel telah dideklarasikan sebelum digunakan.
- Memeriksa tipe data pada ekspresi (type checking).
- Memastikan aturan lingkup (scope) dan visibilitas simbol.
- Membangun symbol table yang menyimpan informasi deklarasi variabel, tipe, konstanta, fungsi, dan prosedur.

Contoh Kasus:

```
variabel x: integer;
mulai
  x := true;
selesai.
```

Meskipun secara sintaks benar, kode di atas akan menghasilkan semantic error karena true adalah tipe boolean, bukan integer. Keluaran tahap ini berupa annotated syntax tree atau symbol table yang akan digunakan pada tahap berikutnya.

4. Intermediate Code Generation:

   Tahapan ini menerjemahkan *syntax tree* atau *symbol table* menjadi representasi kode menengah (intermediate code) yang lebih dekat ke bahasa mesin namun masih bersifat *portable*. Tujuannya adalah membuat bentuk kode yang mudah di optimasi dan tidak bergantung pada arsitektur *hardware* tertentu.

   Bentuk Umum *Intermediate Code*:

   - Three-Address Code (TAC)

     ```
     t1 := a + b
     t2 := t1 * c
     ```

   - Quadruples dan Triples
   - Intermediate Representation (IR) seperti DAG (*Directed Acyclic Graph*) atau SSA (*Static Single Assignment*)

   Fungsi Utama:
   - Mempermudah analisis dan optimisasi program.
   - Menjadi bentuk perantara sebelum diterjemahkan ke bahasa mesin.

5. Optimization

   Optimisasi adalah proses memperbaiki kode menengah agar lebih efisien, baik dari sisi waktu eksekusi maupun penggunaan memori, tanpa mengubah hasil logis dari program.

   Jenis Optimisasi:
   - Local Optimization:
     Perbaikan pada *basic block* (contohnya *constant folding*, *strength reduction*, *dead code elimination*).

   - Global Optimization:
     Perbaikan di seluruh fungsi atau modul (*common subexpression elimination*, *loop invariant code motion*).

- Peephole Optimization:
  Penyederhanaan pada potongan kecil kode (*instruction-level optimization*).

Fungsi:

- Mengurangi waktu eksekusi program.
- Menghemat memori dan register.
- Meningkatkan performa keseluruhan tanpa mengubah output program.

6. Code Generation dan Linking

   Tahapan terakhir kompilasi adalah pembuatan kode mesin (*machine code*) dan *linking* dengan pustaka eksternal agar program dapat dijalankan secara mandiri. Code Generation menerjemahkan *intermediate code* menjadi assembly code atau machine code spesifik arsitektur. Linking kemudian menggabungkan semua modul yang dikompilasi (termasuk pustaka dan fungsi eksternal) menjadi *executable file*.

   Tantangan dalam Code Generation:

   - Alokasi register yang efisien.
   - Penjadwalan instruksi agar tidak terjadi pipeline stall.
   - Manajemen memori (stack, heap, data segment).

   Keluaran akhirnya berupa *object code* atau *executable* binary yang dapat dijalankan oleh sistem operasi.

1.2. Grammar dan Context-Free Grammar (CFG)

Grammar adalah seperangkat aturan produksi yang menentukan bagaimana simbol-simbol dalam suatu bahasa dapat dibentuk. Dalam teori automata, grammar dinyatakan sebagai 4-tuple:

$$G=(V_T, V_N, P, S)$$

dengan:

V_T : himpunan simbol terminal (token yang tidak dapat diuraikan lagi, seperti if, ; , :=, dsb),

V_N : himpunan simbol non-terminal (seperti <program>, <statement>, <expression>, dll),

P : himpunan aturan produksi berbentuk $A \rightarrow \alpha$, di mana $A \in V_N$ dan

$\alpha \in (V_T \cup V_N)$

S : simbol awal (start symbol).

1.3. Parse Tree

Parse Tree adalah representasi pohon dari struktur sintaks program berdasarkan aturan grammar yang digunakan. Setiap node pada pohon mewakili suatu simbol (terminal atau non-terminal), sedangkan cabang pohon menunjukkan aturan produksi yang diterapkan.

Perbedaan dengan Abstract Syntax Tree:

- Parse Tree memuat semua simbol terminal dan non-terminal sesuai grammar.
- Abstract Syntax Tree (AST) lebih ringkas dan fokus pada struktur logis program, dengan menghilangkan simbol-simbol yang tidak berpengaruh terhadap semantik (misalnya tanda kurung, titik koma).

1.4. Recursive Descent Parsing

Recursive Descent Parsing adalah metode top-down parsing di mana setiap non-terminal pada grammar direpresentasikan oleh satu fungsi rekursif. Fungsi ini akan memanggil fungsi lain sesuai dengan aturan produksi yang berlaku.

Cara Kerja:

- Membaca token satu per satu dari hasil lexer.
- Mencocokkan token tersebut dengan simbol yang diharapkan sesuai grammar.

- Jika cocok, parser melanjutkan proses ke simbol berikutnya.
- Jika tidak cocok, parser melaporkan error sintaks.

1.5. Error Handling dalam Parser

Error handling sangat penting agar program dapat memberikan pesan kesalahan yang informatif ketika menemui token yang tidak sesuai grammar.

Beberapa teknik umum:

- Panic Mode Recovery
  Melewati token hingga menemukan titik sinkronisasi seperti ; atau end untuk melanjutkan parsing.

- Phrase-Level Recovery
  Mencoba memperbaiki kesalahan kecil seperti menambahkan token yang hilang atau menghapus token yang tidak diharapkan.

- Error Productions
  Menambahkan aturan produksi khusus untuk menangani kesalahan umum dalam grammar.

BAB 2
PERANCANGAN & IMPLEMENTASI

2.1. Alur Kerja Program

Program compiler ini bekerja dalam 3 tahap utama:

1. Input Processing

    - Program menerima input berupa file .pas (source code Pascal) atau .txt (token list).
    - Jika input .pas, file akan di-tokenize menggunakan lexer.
    - Jika input .txt, file langsung dibaca sebagai daftar token yang sudah ada.
    - Auto-detection encoding dilakukan untuk mendukung UTF-8 dan UTF-16-LE

2. Lexical Analysis (hanya untuk file .pas)

    - Lexer membaca source code karakter per karakter menggunakan DFA (Deterministic Finite Automaton).
    - Comment {...} dan (*...*) di-skip sebelum tokenizing.
    - Token dihasilkan dalam format tuple (type, value).
    - Compound keywords seperti "selain-itu" dan "turun-ke" digabungkan setelah tokenizing.
    - Hasilnya adalah list of tokens: [("KEYWORD", "program"), ("IDENTIFIER", "Hello"), ...].

3. Syntax Analysis (Parsing)

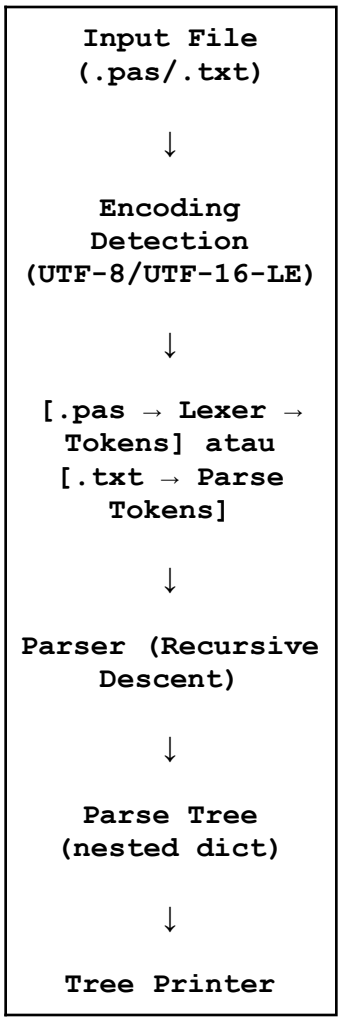
    - Parser menerima list of tokens dari lexer.
    - Menggunakan algoritma Recursive Descent untuk membangun parse tree.
    - Setiap aturan grammar diimplementasikan sebagai fungsi terpisah (31 fungsi)
    - Parse tree direpresentasikan sebagai nested dictionary dengan struktur:

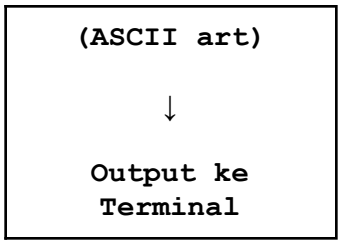
```
{
    "type": "<node-name>",
    "children": [child1, child2, ...]
}
```

4. Output Generation

- Parse tree dicetak ke terminal menggunakan ASCII art formatting.
- Format tree menggunakan karakter ├── dan └── untuk menunjukkan hierarki.
- Terminal node (token) ditampilkan sebagai TYPE(value).
- Non-terminal node ditampilkan sebagai <node-name>

Diagram Alur:





2.2. Modifikasi Lexer

Lexer dari Milestone 1 dimodifikasi dengan beberapa penambahan untuk Milestone 2 yaitu:

1. Comment Handling

```
def skip_comment(text, pos):
    # skip comment { } atau (* *) dan return posisi setelah
comment
    if text[pos] == '{':
        # cari closing }
        pos += 1
        while pos < n and text[pos] != '}':
            pos += 1
        return pos + 1 if pos < n else n

    if text[pos:pos+2] == '(*':
        # cari closing *)
        pos += 2
        while pos + 1 < n:
            if text[pos] == '*' and text[pos + 1] == ')':
                return pos + 2
            pos += 1
        return n
```

- Mendukung 2 style comment Pascal: {...} dan (*...*).
- Multi-line comment didukung.
- Unclosed comment akan di-skip sampai akhir file

2. Compound Keywords Merging

```
    def merge_compound_keywords(tokens):
        # gabungin token yang pisah jadi compound keyword
        # selain - itu → selain-itu
```

```
        # turun - ke → turun-ke
```

- Menggabungkan 3 token menjadi 1 token untuk compound keywords.
- Pattern matching: IDENTIFIER + "-" + IDENTIFIER/KEYWORD.
- Diperlukan karena DFA tidak bisa mengenali - di tengah keyword.

3. Keyword Classification

- Keywords bahasa Indonesia: program, variabel, mulai, selesai, jika, maka, selain-itu, untuk, ke, turun-ke, dll.
- Logical operators: dan, atau, tidak.
- Arithmetic operators: bagi, mod.
- Reklasifikasi dilakukan setelah tokenizing berdasarkan lookup table

4. Integrasi dengan Parser

- Lexer menghasilkan list of tuples: [(type, value), ...].
- Parser mengonversi tuples menjadi objek Token untuk kemudahan akses.
- Interface: tokenize_from_file(dfa_path, source_path) → list of tuples

2.3. Struktur Data (Kelas Node)

Token Class

```
    class Token:
        def __init__(self, token_type, value):
            self.type = token_type      # tipe token (KEYWORD,
    IDENTIFIER, dll)
            self.value = value           # nilai token
    (program, Hello, dll)

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

- Berfungsi sebagai wrapper untuk tuple (type, value) dari lexer.
- Memudahkan akses dengan token.type dan token.value instead of token[0] dan token[1].
- Digunakan di parser untuk tracking posisi dan matching

Parse Tree Node (Dictionary)

```
node = {
    "type": "<node-name>",
    "children": [child1, child2, ...]
}
```

Jenis Node:

1. Non-Terminal Node - node dengan children (production rules)

```
{
    "type": "<program>",
    "children": [
        {"type": "<program-header>", "children": [...]},
        {"type": "<declaration-part>", "children": [...]},
        {"type": "<compound-statement>", "children": [...]},
        Token("DOT", ".")
    ]
}
```

2. Terminal Node - objek Token (leaf nodes)

```
Token("KEYWORD", "program")
Token("IDENTIFIER", "Hello")
Token("NUMBER", "42")
```

2.4. Implementasi Parser (Recursive Descent)

Konsep Recursive Descent:

- Setiap non-terminal dalam grammar diimplementasikan sebagai fungsi.
- Fungsi memanggil fungsi lain sesuai production rules.
- Top-down parsing: mulai dari start symbol <program> turun ke terminal

Struktur Umum Fungsi Parse:

```
def parse_<construct>(self):
    # 1. buat node baru
    node = {"type": "<construct>", "children": []}

    # 2. consume terminal yang expected
    node["children"].append(self.expect("TOKEN_TYPE", "value"))

    # 3. panggil fungsi parse lain untuk non-terminal
    node["children"].append(self.parse_<sub_construct>())

    # 4. return node yang udah lengkap
    return node
```

Fungsi Helper:

1. expect() → consume token yang expected, error jika tidak sesuai

```
self.expect("KEYWORD", "program")  # harus "program"
self.expect("SEMICOLON")           # harus ";"
```

2. match() → check token tanpa consume (lookahead)

```
if self.match("KEYWORD", "jika"):
    return self.parse_if_statement()
```

3. advance() → pindah ke token selanjutnya

```
self.advance()  # self.pos += 1
```

4. peek() → lihat token di depan tanpa consume

```
next_token = self.peek(1)  # lihat 1 token ke depan
```

Contoh Implementasi untuk <program>:

```
# grammar: <program> ::= <program-header> <declaration-part>
<compound-statement>
def parse_program(self):
    node = {"type": "<program>", "children": []}
    node["children"].append(self.parse_program_header())  #
rekursif
    node["children"].append(self.parse_declaration_part()) #
rekursif
    node["children"].append(self.parse_compound_statement()) #
rekursif
    node["children"].append(self.expect("DOT"))  # terminal
    return node
```

Handling Pilihan (OR):

```
# <statement> ::= <assignment> | <if> | <while> | ...
def parse_statement(self):
    if self.match("KEYWORD", "jika"):
        return self.parse_if_statement()
    elif self.match("KEYWORD", "selama"):
        return self.parse_while_statement()
    elif self.match("KEYWORD", "untuk"):
        return self.parse_for_statement()
    # ...
```

Handling Repetisi (Loop):

```
# <identifier-list> ::= <identifier> {, <identifier>}
def parse_identifier_list(self):
    node = {"type": "<identifier-list>", "children": []}
    node["children"].append(self.expect("IDENTIFIER"))

    # loop selama ada koma
    while self.match("COMMA"):
        node["children"].append(self.expect("COMMA"))
        node["children"].append(self.expect("IDENTIFIER"))

    return node
```

Total 31 Fungsi Parse:

```
parse_program, parse_program_header
parse_declaration_part, parse_const_declaration,
parse_type_declaration, parse_var_declaration
parse_identifier_list, parse_type, parse_array_type, parse_range
parse_subprogram_declaration, parse_procedure_declaration,
parse_function_declaration
parse_formal_parameter_list, parse_parameter_group
parse_block, parse_compound_statement, parse_statement_list
parse_statement, parse_assignment_statement
parse_if_statement, parse_while_statement, parse_for_statement,
parse_repeat_statement
parse_procedure_call, parse_parameter_list
parse_expression, parse_simple_expression, parse_term,
parse_factor
parse_function_call
```

2.5. Implementasi Grammar

Program Structure:

<program> ::= <program-header> <declaration-part> <compound-statement> "."
<program-header> ::= "program" <identifier> ";"

Declaration Part

<declaration-part> ::= { <const-declaration> | <type-declaration> | <var-declaration> | <subprogram-declaration> }

<const-declaration> ::= "konstanta" <const-definition> { <const-definition> }

<const-definition> ::= <identifier> "=" <constant-value> ";"

<constant-value> ::= <number> | <char-literal> | <string-literal> | <identifier>

<type-declaration> ::= "tipe" <type-definition> { <type-definition> }

<type-definition> ::= <identifier> "=" <type> ";"

<var-declaration> ::= "variabel" <variable-definition> { <variable-definition> }

<variable-definition> ::= <identifier-list> ":" <type> ";"

<identifier-list> ::= <identifier> { "," <identifier> }

Type Definitions

<type> ::= <simple-type> | <array-type> | <identifier>

<simple-type> ::= "integer" | "real" | "boolean" | "char" | "string" | <range>

<array-type> ::= "larik" "[" <range> "]" "dari" <type>

<range> ::= <expression> ".." <expression>

Subprogram Declarations

<subprogram-declaration> ::= <procedure-declaration> | <function-declaration>

<procedure-declaration> ::= "prosedur" <identifier> [<formal-parameter-list>] ";" <block> ";"

<function-declaration> ::= "fungsi" <identifier> [<formal-parameter-list>] ":" <type> ";" <block> ";"

<formal-parameter-list> ::= "(" <parameter-group> { ";" <parameter-group> } ")"

<parameter-group> ::= <identifier-list> ":" <type>

<block> ::= <declaration-part> <compound-statement>

Statements

<compound-statement> ::= "mulai" <statement-list> "selesai"

<statement-list> ::= <statement> { ";" <statement> } [";"]

<statement> ::= <assignment-statement>

| <if-statement>

| <while-statement>

| <for-statement>

| <repeat-statement>

| <procedure-call>

| <compound-statement>

| <empty-statement>

<empty-statement> ::= ε

<assignment-statement> ::= <identifier> ["[" <expression> "]"] ":=" <expression>

<if-statement> ::= "jika" <expression> "maka" <statement> ["selain-itu" <statement>]

<while-statement> ::= "selama" <expression> "lakukan" <statement>

<for-statement> ::= "untuk" <identifier> ":=" <expression> ("ke" | "turun-ke") <expression> "lakukan" <statement>

<repeat-statement> ::= "ulangi" <statement-list> "sampai" <expression>

<procedure-call> ::= <identifier> "(" [<parameter-list>] ")"

Expressions

<parameter-list> ::= <expression> { "," <expression> }

<expression> ::= <simple-expression> [<relational-operator> <simple-expression>]

<relational-operator> ::= "=" | "<>" | "<" | ">" | "<=" | ">="

<simple-expression> ::= [<sign>] <term> { <additive-operator> <term> }

<sign> ::= "+" | "-"

<additive-operator> ::= "+" | "-" | "atau"

<term> ::= <factor> { <multiplicative-operator> <factor> }

<multiplicative-operator> ::= "*" | "/" | "bagi" | "mod" | "dan"

<factor> ::= <identifier> [<array-access> | <function-call>]

| <number>

| <char-literal>

| <string-literal>

| "(" <expression> ")"

| "tidak" <factor>

<array-access> ::= "[" <expression> "]"

<function-call> ::= "(" [<parameter-list>] ")"

Lexical Elements

<identifier> ::= IDENTIFIER

<number> ::= NUMBER

<char-literal> ::= CHAR_LITERAL

<string-literal> ::= STRING_LITERAL

Notasi EBNF yang Digunakan:

::= : didefinisikan sebagai

| : alternatif (pilihan)

[...] : opsional (0 atau 1 kali)

{ ... } : repetisi (0 atau lebih kali)

" ... " : simbol terminal (keyword atau operator)

<...> : simbol non-terminal

ε : string kosong (epsilon)

BAB 3
PENGUJIAN

3.1. Test Case 1 (test1.pas)

Input	Output

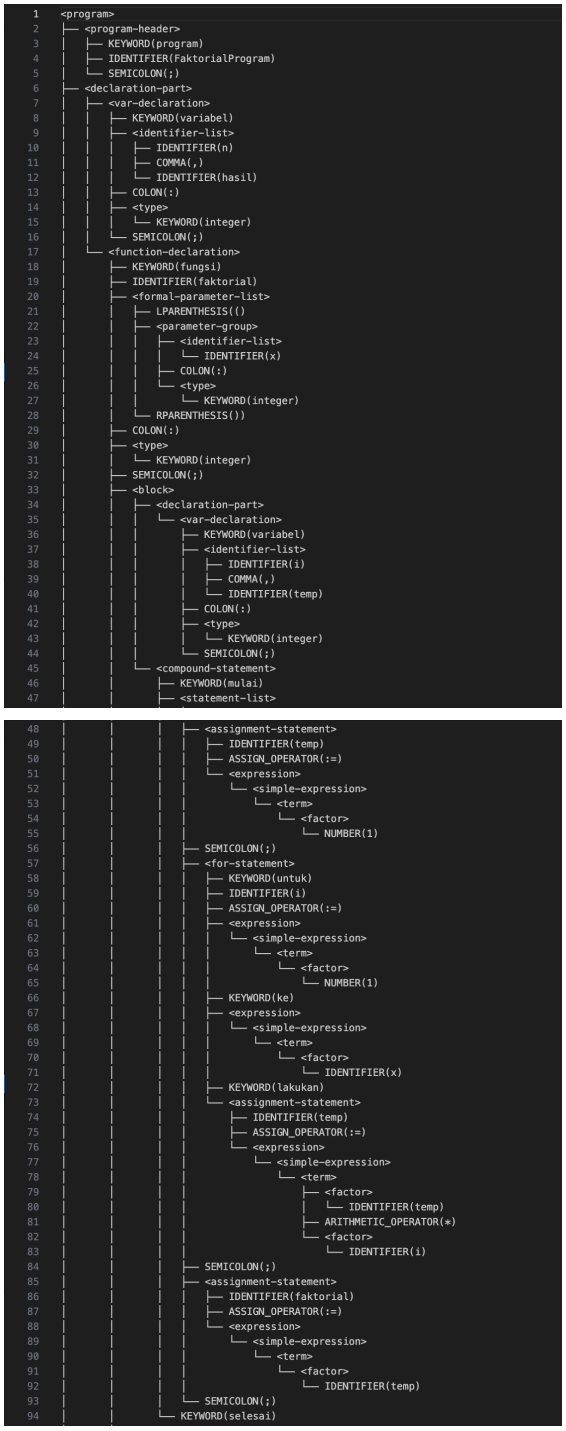
`program Hello;`
 	
`variabel`
`  a, b: integer;`
 	
`mulai`
`  a := 5;`
`  b := a + 10;`
`  writeln('Result = ', b);`
`selesai.`
	

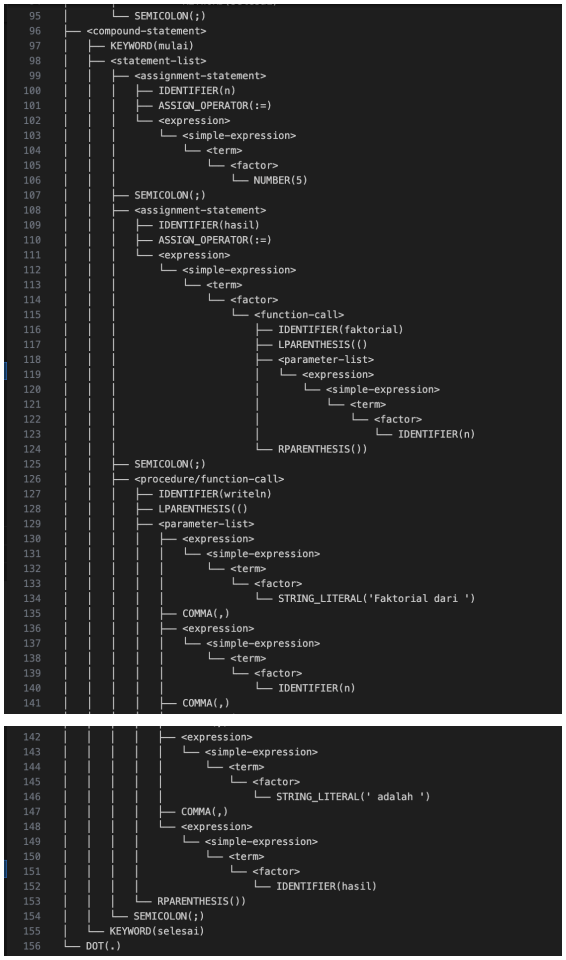
3.2. Test Case 2 (test2.pas)

Input	Output

`program FaktorialProgram;`
 	
`variabel`
`  n, hasil: integer;`
 	
`fungsi faktorial(x: integer):`	

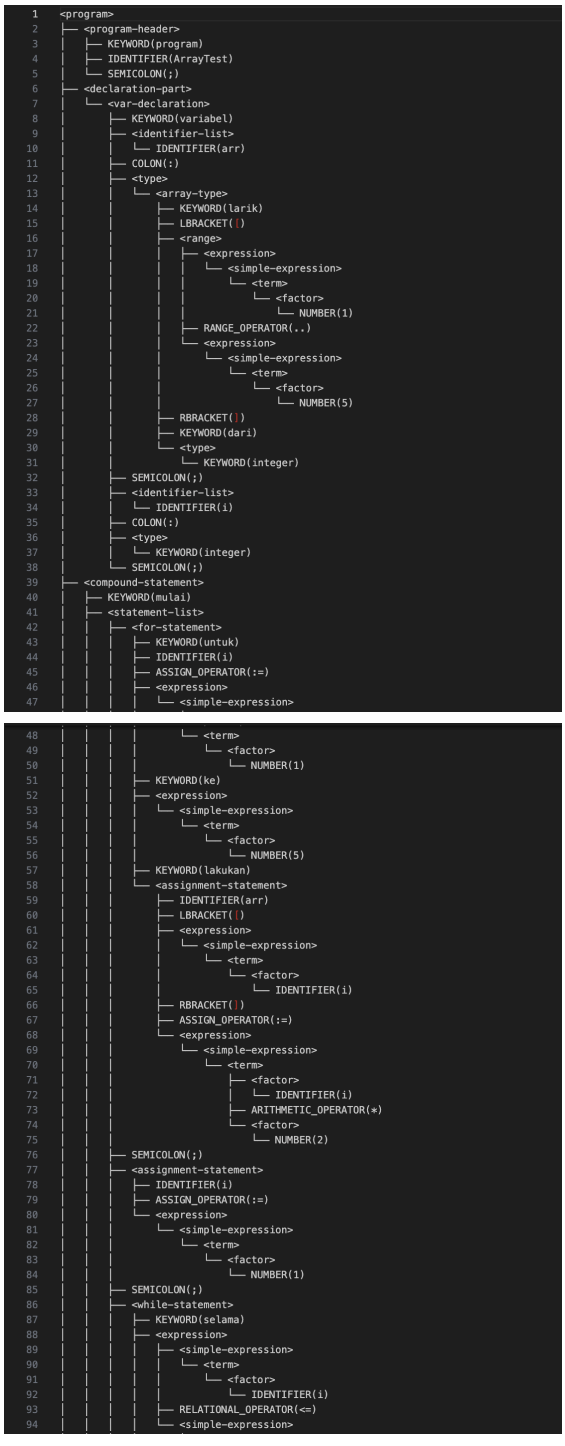
```
integer;
variabel
  i, temp: integer;
mulai
  temp := 1;
  untuk i := 1 ke x lakukan
    temp := temp * i;
  faktorial := temp;
selesai;

mulai
  n := 5;
  hasil := faktorial(n);
  writeln('Faktorial dari ',
n, ' adalah ', hasil);
selesai.
```



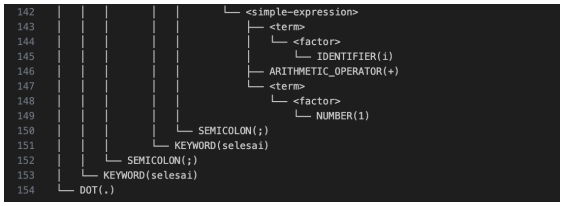

└── SEMICOLON(;)
├── <compound-statement>
├── KEYWORD(mulai)
├── <statement-list>
├── <assignment-statement>
├── IDENTIFIER(n)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(5)
├── SEMICOLON(;)
├── <assignment-statement>
├── IDENTIFIER(hasil)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── <function-call>
├── IDENTIFIER(faktorial)
├── LPARENTHESIS(()
├── <parameter-list>
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(n)
└── RPARENTHESIS())
├── SEMICOLON(;)
├── <procedure/function-call>
├── IDENTIFIER(writeln)
├── LPARENTHESIS(()
├── <parameter-list>
├── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── STRING_LITERAL('Faktorial dari ')
├── COMMA(,)
├── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(n)
├── COMMA(,)
├── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── STRING_LITERAL(' adalah ')
├── COMMA(,)
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(hasil)
└── RPARENTHESIS())
└── SEMICOLON(;)
└── KEYWORD(selesai)
└── DOT(.)

3.3. Test Case 3 (test3.pas)

Input	Output

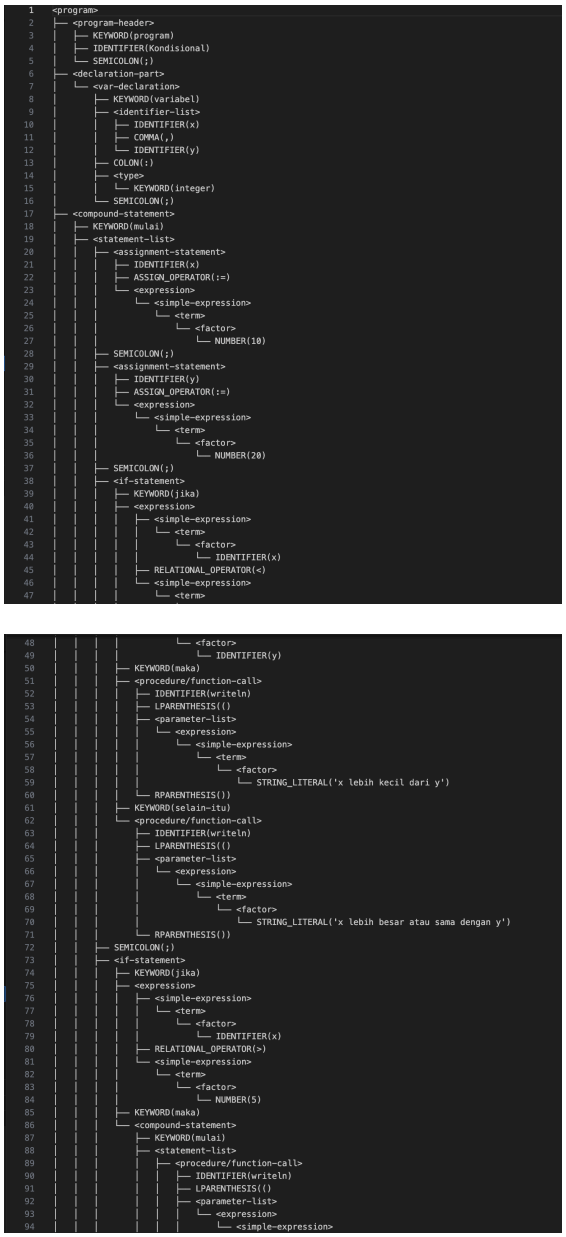
`program ArrayTest;`
 	
`variabel`
`  arr: larik[1..5] dari integer;`
`  i: integer;`
 	
`mulai`
`  untuk i := 1 ke 5 lakukan`
`  arr[i] := i * 2;`
 	
`  i := 1;`
`  selama i <= 5 lakukan`
`  mulai`
`  writeln('arr[', i, '] = ', arr[i]);`
`  i := i + 1;`
`  selesai;`
`selesai.`
	

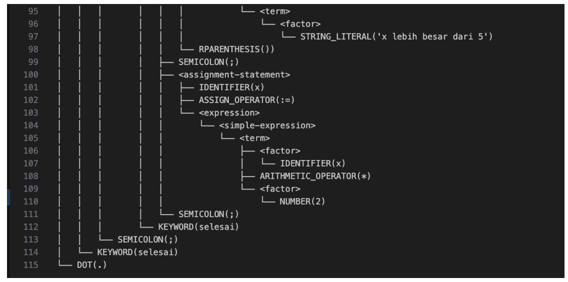
```
<program>
├── <program-header>
│   ├── KEYWORD(program)
│   ├── IDENTIFIER(ArrayTest)
│   └── SEMICOLON(;)
├── <declaration-part>
│   └── <var-declaration>
│       ├── KEYWORD(variabel)
│       ├── <identifier-list>
│       │   └── IDENTIFIER(arr)
│       ├── COLON(:)
│       ├── <type>
│       │   └── <array-type>
│       │       ├── KEYWORD(larik)
│       │       ├── LBRACKET([)
│       │       ├── <range>
│       │       │   ├── <expression>
│       │       │   │   └── <simple-expression>
│       │       │   │       └── <term>
│       │       │   │           └── <factor>
│       │       │   │               └── NUMBER(1)
│       │       │   ├── RANGE_OPERATOR(..)
│       │       │   └── <expression>
│       │       │       └── <simple-expression>
│       │       │           └── <term>
│       │       │               └── <factor>
│       │       │                   └── NUMBER(5)
│       │       ├── RBRACKET(])
│       │       ├── KEYWORD(dari)
│       │       └── <type>
│       │           └── KEYWORD(integer)
│       ├── SEMICOLON(;)
│       ├── <identifier-list>
│       │   └── IDENTIFIER(i)
│       ├── COLON(:)
│       ├── <type>
│       │   └── KEYWORD(integer)
│       └── SEMICOLON(;)
├── <compound-statement>
│   ├── KEYWORD(mulai)
│   ├── <statement-list>
│   │   ├── <for-statement>
│   │   │   ├── KEYWORD(untuk)
│   │   │   ├── IDENTIFIER(i)
│   │   │   ├── ASSIGN_OPERATOR(:=)
│   │   │   ├── <expression>
│   │   │   │   └── <simple-expression>
│   │   │   │       └── <term>
│   │   │   │           └── <factor>
│   │   │   │               └── NUMBER(1)
│   │   │   ├── KEYWORD(ke)
│   │   │   ├── <expression>
│   │   │   │   └── <simple-expression>
│   │   │   │       └── <term>
│   │   │   │           └── <factor>
│   │   │   │               └── NUMBER(5)
│   │   │   ├── KEYWORD(lakukan)
│   │   │   └── <assignment-statement>
│   │   │       ├── IDENTIFIER(arr)
│   │   │       ├── LBRACKET([)
│   │   │       ├── <expression>
│   │   │       │   └── <simple-expression>
│   │   │       │       └── <term>
│   │   │       │           └── <factor>
│   │   │       │               └── IDENTIFIER(i)
│   │   │       ├── RBRACKET(])
│   │   │       ├── ASSIGN_OPERATOR(:=)
│   │   │       └── <expression>
│   │   │           └── <simple-expression>
│   │   │               └── <term>
│   │   │                   ├── <factor>
│   │   │                   │   └── IDENTIFIER(i)
│   │   │                   ├── ARITHMETIC_OPERATOR(*)
│   │   │                   └── <factor>
│   │   │                       └── NUMBER(2)
│   │   ├── 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>
```



3.4. Test Case 4 (test4.pas)

Input	Output

`program Kondisional;`
 	
`variabel`
`  x, y: integer;`
 	
`mulai`
`  x := 10;`
`  y := 20;`
 	
`  jika x < y maka`
`  writeln('x lebih kecil dari y')`
`  selain-itu`
`  writeln('x lebih besar atau sama dengan y');`
 	
`  jika x > 5 maka`
`  mulai`
`  writeln('x lebih besar dari 5');`
`  x := x * 2;`
`  selesai;`
`selesai.`
	



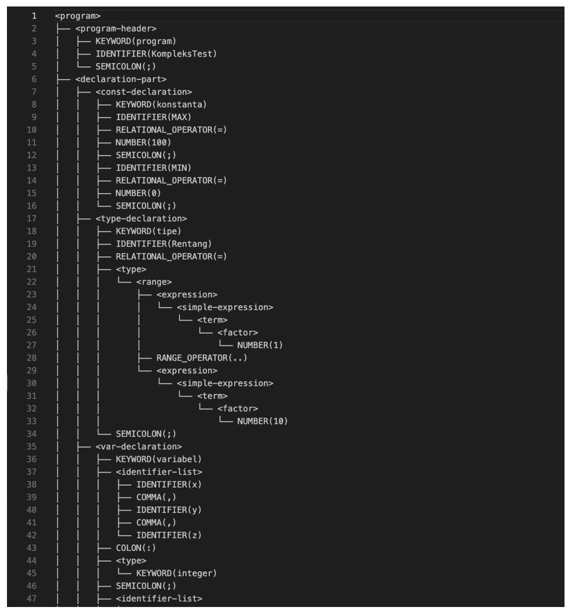
```
│   │   │       │   │   │           └── <term>
│   │   │       │   │   │               └── <factor>
│   │   │       │   │   │                   └── STRING_LITERAL('x lebih besar dari 5')
│   │   │       │   │   └── RPARENTHESIS())
│   │   │       │   ├── SEMICOLON(;)
│   │   │       │   ├── <assignment-statement>
│   │   │       │   │   ├── IDENTIFIER(x)
│   │   │       │   │   ├── ASSIGN_OPERATOR(:=)
│   │   │       │   │   └── <expression>
│   │   │       │   │       └── <simple-expression>
│   │   │       │   │           └── <term>
│   │   │       │   │               ├── <factor>
│   │   │       │   │               │   └── IDENTIFIER(x)
│   │   │       │   │               ├── ARITHMETIC_OPERATOR(*)
│   │   │       │   │               └── <factor>
│   │   │       │   │                   └── NUMBER(2)
│   │   │       │   └── SEMICOLON(;)
│   │   │       └── KEYWORD(selesai)
│   │   └── SEMICOLON(;)
│   └── KEYWORD(selesai)
└── DOT(.)
```

3.5. Test Case 5 (test5.pas)

Input

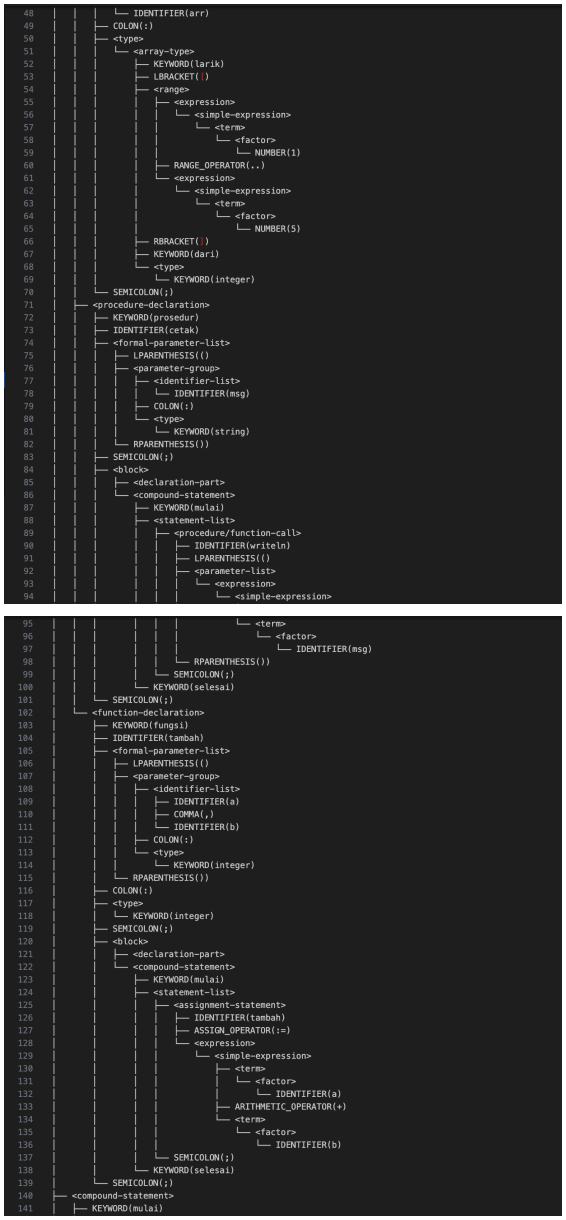
```
program KompleksTest;

konstanta
  MAX = 100;
  MIN = 0;

tipe
  Rentang = 1..10;

variabel
  x, y, z: integer;
  arr: larik[1..5] dari
integer;

prosedur cetak(msg: string);
mulai
  writeln(msg);
selesai;

fungsi tambah(a, b: integer):
integer;
mulai
  tambah := a + b;
selesai;

mulai
  x := 5;
  y := 10;
  z := tambah(x, y);

  cetak('Hasil penjumlahan:');
  writeln(z);
```

Output



```
<program>
├── <program-header>
│   ├── KEYWORD(program)
│   ├── IDENTIFIER(KompleksTest)
│   └── SEMICOLON(;)
├── <declaration-part>
│   ├── <const-declaration>
│   │   ├── KEYWORD(konstanta)
│   │   ├── IDENTIFIER(MAX)
│   │   ├── RELATIONAL_OPERATOR(=)
│   │   ├── NUMBER(100)
│   │   ├── SEMICOLON(;)
│   │   ├── IDENTIFIER(MIN)
│   │   ├── RELATIONAL_OPERATOR(=)
│   │   ├── NUMBER(0)
│   │   └── SEMICOLON(;)
│   ├── <type-declaration>
│   │   ├── KEYWORD(tipe)
│   │   ├── IDENTIFIER(Rentang)
│   │   ├── RELATIONAL_OPERATOR(=)
│   │   ├── <type>
│   │   │   └── <range>
│   │   │       ├── <expression>
│   │   │       │   └── <simple-expression>
│   │   │       │       └── <term>
│   │   │       │           └── <factor>
│   │   │       │               └── NUMBER(1)
│   │   │       ├── RANGE_OPERATOR(..)
│   │   │       └── <expression>
│   │   │           └── <simple-expression>
│   │   │               └── <term>
│   │   │                   └── <factor>
│   │   │                       └── NUMBER(10)
│   │   └── SEMICOLON(;)
│   ├── <var-declaration>
│   │   ├── KEYWORD(variabel)
│   │   ├── <identifier-list>
│   │   │   ├── IDENTIFIER(x)
│   │   │   ├── COMMA(,)
│   │   │   ├── IDENTIFIER(y)
│   │   │   ├── COMMA(,)
│   │   │   └── IDENTIFIER(z)
│   │   ├── COLON(:)
│   │   ├── <type>
│   │   │   └── KEYWORD(integer)
│   │   ├── SEMICOLON(;)
│   │   ├── <identifier-list>
```

```
  jika z > 10 maka
    cetak('z lebih besar dari
10')
  selain-itu
    cetak('z tidak lebih besar
dari 10');

  untuk x := 1 ke 5 lakukan
    arr[x] := x * x;

  selama y > 0 lakukan
  mulai
    y := y - 1;
    writeln(y);
  selesai;
selesai.
```




├── <statement-list>
│   ├── <assignment-statement>
│   │   ├── IDENTIFIER(x)
│   │   ├── ASSIGN_OPERATOR(:=)
│   │   └── <expression>
│   │   └── <simple-expression>
│   │   └── <term>
│   │   └── <factor>
│   │   └── NUMBER(5)
│   ├── SEMICOLON(;)
│   ├── <assignment-statement>
│   │   ├── IDENTIFIER(y)
│   │   ├── ASSIGN_OPERATOR(:=)
│   │   └── <expression>
│   │   └── <simple-expression>
│   │   └── <term>
│   │   └── <factor>
│   │   └── NUMBER(10)
│   ├── SEMICOLON(;)
│   ├── <assignment-statement>
│   │   ├── IDENTIFIER(z)
│   │   ├── ASSIGN_OPERATOR(:=)
│   │   └── <expression>
│   │   └── <simple-expression>
│   │   └── <term>
│   │   └── <factor>
│   │   └── <function-call>
│   │   ├── IDENTIFIER(tambah)
│   │   ├── LPARENTHESIS(()
│   │   ├── <parameter-list>
│   │   │   ├── <expression>
│   │   │   │   └── <simple-expression>
│   │   │   │   └── <term>
│   │   │   │   └── <factor>
│   │   │   │   └── IDENTIFIER(x)
│   │   │   ├── COMMA(,)
│   │   │   └── <expression>
│   │   │   └── <simple-expression>
│   │   │   └── <term>
│   │   │   └── <factor>
│   │   │   └── IDENTIFIER(y)
│   │   └── RPARENTHESIS())
│   ├── SEMICOLON(;)
│   ├── <procedure/function-call>
│   │   ├── IDENTIFIER(cetak)
│   │   ├── LPARENTHESIS(()
│   │   ├── <parameter-list>
│   │   │   └── <expression>
│   │   │   └── <simple-expression>
│   │   │   └── <term>
│   │   │   └── <factor>
│   │   │   └── STRING_LITERAL('Hasil penjumlahan:')
│   │   └── RPARENTHESIS())
│   ├── SEMICOLON(;)
│   ├── <procedure/function-call>
│   │   ├── IDENTIFIER(writeln)
│   │   ├── LPARENTHESIS(()
│   │   ├── <parameter-list>
│   │   │   └── <expression>
│   │   │   └── <simple-expression>
│   │   │   └── <term>
│   │   │   └── <factor>
│   │   │   └── IDENTIFIER(z)
│   │   └── RPARENTHESIS())
│   ├── SEMICOLON(;)
│   ├── <if-statement>
│   │   ├── KEYWORD(jika)
│   │   ├── <expression>
│   │   │   ├── <simple-expression>
│   │   │   │   └── <term>
│   │   │   │   └── <factor>
│   │   │   │   └── IDENTIFIER(z)
│   │   │   ├── RELATIONAL_OPERATOR(>)
│   │   │   └── <simple-expression>
│   │   │   └── <term>
│   │   │   └── <factor>
│   │   │   └── NUMBER(10)
│   │   ├── KEYWORD(maka)
│   │   ├── <procedure/function-call>
│   │   │   ├── IDENTIFIER(cetak)
│   │   │   ├── LPARENTHESIS(()
│   │   │   ├── <parameter-list>
│   │   │   │   └── <expression>
│   │   │   │   └── <simple-expression>
│   │   │   │   └── <term>
│   │   │   │   └── <factor>
│   │   │   │   └── STRING_LITERAL('z lebih besar dari 10')
│   │   │   └── RPARENTHESIS())
│   │   ├── KEYWORD(selain-itu)
│   │   └── <procedure/function-call>
│   │   ├── IDENTIFIER(cetak)
│   │   ├── LPARENTHESIS(()
│   │   ├── <parameter-list>
│   │   │   └── <expression>

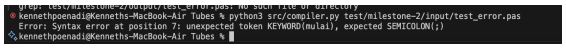
3.6. Test Case 6 (test_brutal.pas)

Test ini sangat panjang, kami hanya melampirkan link Google Drive dari test case ini yaitu:

GDrive Berikut

3.7. Test Case 7 (test_error.pas)

Input	Output

`program ErrorTest;`
 	
`variabel`
`  x: integer`
 	
`mulai`
`  x := 5;`
`  writeln(x)`
`selesai.`
	

3.8. Test Case 8 (test_unclosed_comment.pas)

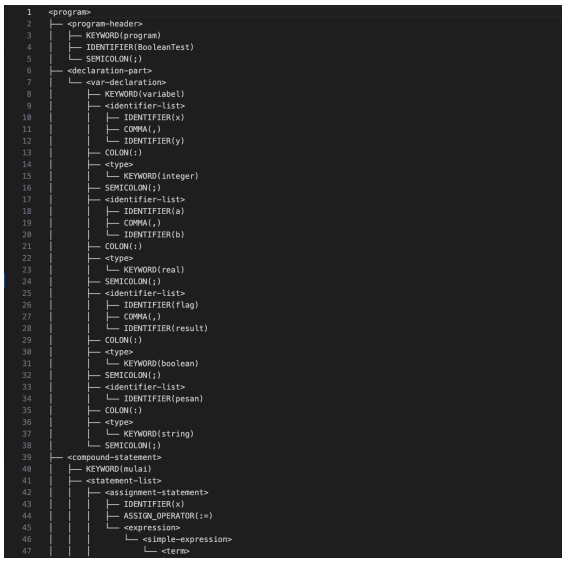
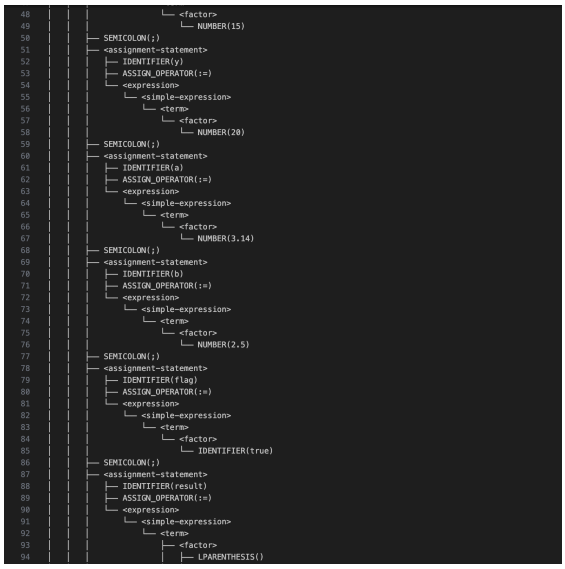
Input	Output

`program TestUnclosedComment;`
 	
`{ Test case untuk unclosed comment }`
`{ Ini test case yang seharusnya error karena ada unclosed comment }`
 	
`variabel`
`  x, y, z: integer;`
`  hasil: real;`
 	
`mulai`
`  x := 10;`
`  y := 20;`
`  z := x + y;`
 	
`  { Ini comment yang tidak ditutup`
`  hasil := z * 2.5;`
`  writeln('Hasil: ', hasil)`
`selesai.`
	

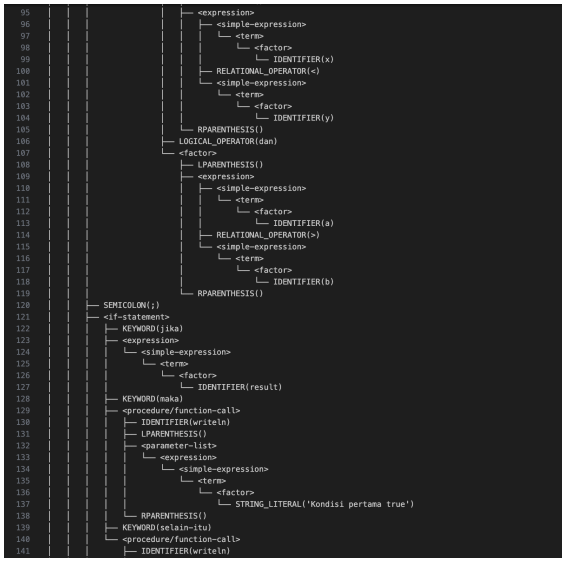
3.9. Test Case 9 (test1_tokens.txt)

Input	Output
KEYWORD(program) IDENTIFIER(Hello) SEMICOLON(;) KEYWORD(variabel) IDENTIFIER(a) COMMA(,) IDENTIFIER(b) COLON(:) KEYWORD(integer) SEMICOLON(;) KEYWORD(mulai) IDENTIFIER(a) ASSIGN_OPERATOR(:=) NUMBER(5) SEMICOLON(;) IDENTIFIER(b) ASSIGN_OPERATOR(:=) IDENTIFIER(a) ARITHMETIC_OPERATOR(+) NUMBER(10) SEMICOLON(;) IDENTIFIER(writeln) LPARENTHESIS(() STRING_LITERAL('Result = ') COMMA(,) IDENTIFIER(b) RPARENTHESIS()) SEMICOLON(;) KEYWORD(selesai) DOT(.)	

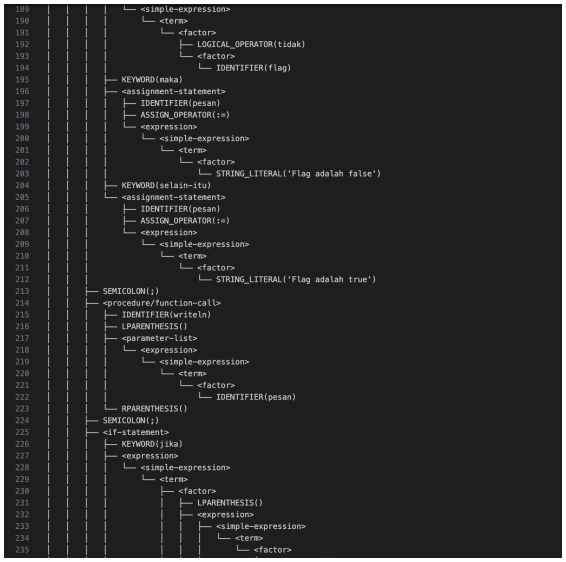
3.10. Test Case 10 (test2_tokens.txt)

Input	Output
KEYWORD(program) IDENTIFIER(BooleanTest) SEMICOLON(;) KEYWORD(variabel) IDENTIFIER(x) COMMA(,) IDENTIFIER(y) COLON(:) KEYWORD(integer) SEMICOLON(;) IDENTIFIER(a) COMMA(,) IDENTIFIER(b) COLON(:) KEYWORD(real) SEMICOLON(;) IDENTIFIER(flag) COMMA(,) IDENTIFIER(result) COLON(:) KEYWORD(boolean) SEMICOLON(;) IDENTIFIER(pesan) COLON(:) KEYWORD(string) SEMICOLON(;) KEYWORD(mulai) IDENTIFIER(x) ASSIGN_OPERATOR(:=) NUMBER(15) SEMICOLON(;) IDENTIFIER(y) ASSIGN_OPERATOR(:=) NUMBER(20) SEMICOLON(;) IDENTIFIER(a) ASSIGN_OPERATOR(:=) NUMBER(3.14) SEMICOLON(;) IDENTIFIER(b) ASSIGN_OPERATOR(:=) NUMBER(2.5) SEMICOLON(;) IDENTIFIER(flag) ASSIGN_OPERATOR(:=) IDENTIFIER(true)	 

```
SEMICOLON(;)
IDENTIFIER(result)
ASSIGN_OPERATOR(:=)
LPARENTHESIS(()
IDENTIFIER(x)
RELATIONAL_OPERATOR(<)
IDENTIFIER(y)
RPARENTHESIS())
LOGICAL_OPERATOR(dan)
LPARENTHESIS(()
IDENTIFIER(a)
RELATIONAL_OPERATOR(>)
IDENTIFIER(b)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(jika)
IDENTIFIER(result)
KEYWORD(maka)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Kondisi pertama
true')
RPARENTHESIS())
KEYWORD(selain-itu)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Kondisi pertama
false')
RPARENTHESIS())
SEMICOLON(;)
IDENTIFIER(result)
ASSIGN_OPERATOR(:=)
LPARENTHESIS(()
IDENTIFIER(x)
RELATIONAL_OPERATOR(>=)
NUMBER(10)
RPARENTHESIS())
LOGICAL_OPERATOR(atau)
LPARENTHESIS(()
IDENTIFIER(y)
RELATIONAL_OPERATOR(<=)
NUMBER(15)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(jika)
LOGICAL_OPERATOR(tidak)
IDENTIFIER(flag)
KEYWORD(maka)
IDENTIFIER(pesan)
ASSIGN_OPERATOR(:=)
STRING_LITERAL('Flag adalah
```



```
false')
KEYWORD(selain-itu)
IDENTIFIER(pesan)
ASSIGN_OPERATOR(:=)
STRING_LITERAL('Flag adalah
true')
SEMICOLON(;)
IDENTIFIER(writeln)
LPARENTHESIS(()
IDENTIFIER(pesan)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(jika)
LPARENTHESIS(()
IDENTIFIER(x)
RELATIONAL_OPERATOR(<>)
IDENTIFIER(y)
RPARENTHESIS())
LOGICAL_OPERATOR(dan)
LPARENTHESIS(()
IDENTIFIER(a)
RELATIONAL_OPERATOR(<>)
IDENTIFIER(b)
RPARENTHESIS())
KEYWORD(maka)
KEYWORD(mulai)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('x tidak sama
dengan y')
RPARENTHESIS())
SEMICOLON(;)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('a tidak sama
dengan b')
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(selesai)
SEMICOLON(;)
KEYWORD(jika)
LPARENTHESIS(()
IDENTIFIER(x)
RELATIONAL_OPERATOR(=)
NUMBER(15)
RPARENTHESIS())
LOGICAL_OPERATOR(atau)
LPARENTHESIS(()
IDENTIFIER(y)
RELATIONAL_OPERATOR(=)
NUMBER(15)
```



RPARENTHESIS()) KEYWORD(maka) IDENTIFIER(writeln) LPARENTHESIS(() STRING_LITERAL('Salah satu bernilai 15') RPARENTHESIS()) SEMICOLON(;) KEYWORD(selesai) DOT(.)	

3.11. Test Case 11 (test3_tokens.txt)

Input	Output
KEYWORD(program) IDENTIFIER(SimpleLoop) SEMICOLON(;) KEYWORD(variabel) IDENTIFIER(i) COMMA(,) IDENTIFIER(total) COLON(:) KEYWORD(integer) SEMICOLON(;) KEYWORD(mulai) IDENTIFIER(total) ASSIGN_OPERATOR(:=) NUMBER(0) SEMICOLON(;) KEYWORD(untuk) IDENTIFIER(i) ASSIGN_OPERATOR(:=) NUMBER(1) KEYWORD(ke) NUMBER(5) KEYWORD(lakukan) IDENTIFIER(total) ASSIGN_OPERATOR(:=) IDENTIFIER(total) ARITHMETIC_OPERATOR(+) IDENTIFIER(i) SEMICOLON(;) IDENTIFIER(writeln) LPARENTHESIS(() STRING_LITERAL('Total: ') COMMA(,) IDENTIFIER(total)	

RPARENTHESIS()) SEMICOLON(;) KEYWORD(selesai) DOT(.)	

BAB 4

Kesimpulan dan Saran

4.1. Kesimpulan

Pada pengerjaan Milestone 2 mata kuliah IF2224 Teori Bahasa Formal Otomata ini, telah berhasil diimplementasikan sebuah syntax analyzer (parser) untuk compiler PASCAL-S. Sesuai dengan spesifikasi tugas, parser ini dikembangkan menggunakan metode Recursive Descent. Program menerima masukan berupa daftar token yang dihasilkan oleh lexer (baik dari file .pas maupun .txt) dan berhasil membangun Parse Tree sebagai keluaran.

Implementasi ini mencakup 31 fungsi parsing yang berbeda untuk menangani setiap aturan produksi non-terminal dari tata bahasa PASCAL-S, mulai dari <program> hingga <factor>. Selain itu, lexer dari Milestone 1 telah dimodifikasi untuk mendukung terjemahan keyword Bahasa Inggris ke Bahasa Indonesia (misalnya, var menjadi variabel) serta menangani multi-line comment ({...} dan (*...*)). Parse Tree direpresentasikan menggunakan struktur data kamus bersarang (nested dictionary) untuk menunjukkan hirarki program. Berdasarkan serangkaian pengujian yang telah dilakukan mencakup deklarasi variabel, struktur kondisional, perulangan, array, prosedur, fungsi, hingga penanganan *syntax error* parser yang dibuat telah mampu memvalidasi sintaks program PASCAL-S dan menghasilkan Parse Tree yang sesuai.

4.2. Saran

Parse Tree yang dihasilkan pada milestone ini akan menjadi fondasi utama untuk tahap kompilasi selanjutnya, yaitu Analisis Semantik. Untuk pengembangan di masa depan, representasi Parse Tree yang saat ini cukup lengkap dapat dioptimalkan menjadi Abstract Syntax Tree (AST). Penggunaan AST akan menyederhanakan pohon sintaks dengan menghilangkan node yang bersifat seremonial (seperti titik koma atau tanda kurung) sehingga proses analisis semantik seperti type *checking* dan manajemen *scope* dapat berjalan lebih efisien.

Selain itu, mekanisme *error handling* dapat lebih dikembangkan. Meskipun program saat ini sudah dapat melaporkan syntax error , implementasi *error recovery* (seperti *panic mode* yang melompati token hingga menemukan titik sinkronisasi seperti ; atau selesai) akan sangat bermanfaat. Hal ini akan memungkinkan parser untuk terus melanjutkan analisis dan mendeteksi beberapa kesalahan sintaks sekaligus dalam satu kali proses kompilasi, alih-alih berhenti pada kesalahan pertama yang ditemukan.

LAMPIRAN

[1] “PASCAL-S: A Subset and its implementation.” [Online]. Available: http://pascal.hansotten.com/uploads/pascals/PASCAL-S%20A%20subset%20and%20its%20Implementation%20012.pdf. [Accessed: 12-Nov-2025].

[2] R. Spivak, “Let’s Build A Simple Interpreter. Part 7: Abstract Syntax Trees,” Ruslan’s Blog, 15 Dec 2015. [Online]. Available: https://ruslanspivak.com/lsbasi-part7. [Accessed: 12-Nov-2025].

[3] “Contoh Gambar Parse Trees,” [Online]. Available: https://textx.github.io/Arpeggio/latest/images/calc_parse_tree.dot.png. [Accessed: 12-Nov-2025]

[4] “Recursive Descent Parser,” GeeksforGeeks. [Online]. Available: https://www.geeksforgeeks.org/compiler-design/recursive-descent-parser/. [Accessed: 12-Nov-2025]

[5] “Abstract Syntax Tree (AST) in Java,” GeeksforGeeks, 12 Aug 2021. [Online]. Available: https://www.geeksforgeeks.org/java/abstract-syntax-tree-ast-in-java/. [Accessed: 12-Nov-2025]

[6] “Parse Tree and Syntax Tree,” GeeksforGeeks, 9 Apr 2025. [Online]. Available: https://www.geeksforgeeks.org/compiler-design/parse-tree-and-syntax-tree/. [Accessed: 12-Nov-2025]

Link Repository Github:

https://github.com/KennethhPoenadi/ARA-Tubes-IF2224.git

Pembagian Kerja:

NIM	Nama	Pembagian Kerja
13522076	Muhammad Syarafi Akmal	25%
13523040	Kenneth Poenadi	25%
13523086	Bob Kunanda	25%
13523104	Muhammad Zahran R.A.	25%

Grammar yang digunakan:

Pada Bab 2.5