LAPORAN MILESTONE 2
IF2224 TEORI BAHASA FORMAL DAN OTOMATA
SYNTAX ANALYSIS

Kelompok 6 DFAntastic (DFC)

Mayla Yaffa Ludmilla 13523050
Anella Utari Gunadi 13523078
Muhammad Edo Raduputu Aprima 13523096
Athian Nugraha Muarajuang 13523106

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2025

DAFTAR ISI

1. DESKRIPSI TUGAS

Pada milestone ini, kami diminta untuk membuat tahapan kedua dari compiler, yaitu syntax analysis. Parser berfungsi melakukan analisis sintaksis (syntax analysis) dengan menggunakan Recursive Descent untuk mengenali struktur gramatikal dalam deretan token.

Tahapan ini bertujuan untuk menganalisis struktur sintaksis dari list of tokens yang dihasilkan lexer dan membangun Parse Tree yang merepresentasikan hierarki struktur program Pascal-S sesuai dengan grammar bahasa. Parse Tree ini akan menjadi masukan bagi tahap semantic analysis dalam proses kompilasi.

Masukan	List Token (misalnya VAR(X), ASSIGN(=), OPERATOR(+))
Keluaran	Parse Tree
Algoritma	Recursive Descent

Program harus menggunakan Recursive Descent dengan grammar yang di-hardcode dalam program dan keseluruhan grammar dijelaskan kembali dalam laporan yang dibuat. Input awal dari program tetap merupakan source code PASCAL-S sehingga sebelum dimasukkan ke parser, kode dimasukkan terlebih dahulu ke lexer.

Input Command (contoh yang digunakan menggunakan Python)
Format: python [Compiler] [Kode Pascal]

```
python compiler.py program.pas
```

Isi program.pas

```
program Hello;

variabel
  a, b: integer;

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

program.pas ketika diubah menjadi token

```
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(;)
KEYWORD(writeln)
LPARENTHESIS(()
STRING_LITERAL('Result = ')
COMMA(,)
IDENTIFIER(b)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(selesai)
DOT(.)
```

Proses (grammar belum tentu benar)

```
program → program-header declaration-part compound-statement DOT
program-header → KEYWORD(program) IDENTIFIER SEMICOLON
declaration-part → ...
compound-statement → ...
... grammar lainnya

Berdasarkan grammar “program”, non-terminal yang pertama ditemukan adalah
“program-header”. Setelah itu di dalam “program-header” dilakukan pengecekan
token input.

KEYWORD(program)
IDENTIFIER(Hello)
SEMICOLON(;)

Karena token input sesuai, berarti “program-header” valid. Sehingga output
sementara akan seperti ini

<program>
 ├── <program-header>
 │      ├── KEYWORD(program)
 │      ├── IDENTIFIER(Hello)
 │      └── SEMICOLON(;)
... sisa output

Kemudian sisa pengecekan dilakukan untuk “declaration-part”,
“compound-statement”, dan DOT.

Kemudian juga disini dilakukan error checking di level parser. Contohnya
```

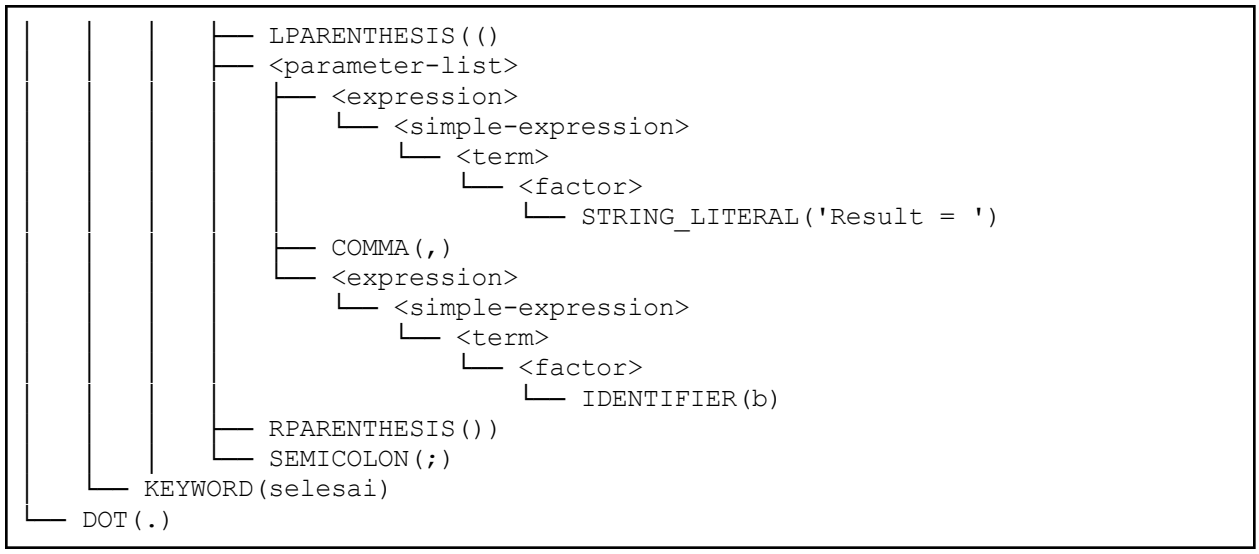
adalah misal untuk “program-header”, token yang dibaca adalah seperti berikut.

```
KEYWORD(program)
IDENTIFIER(Hello)
DOT(.)
```

Maka parser akan mengeluarkan error. Untuk pesan error dibebaskan yang penting informatif. Contohnya adalah “Syntax error: unexpected token DOT(.), expected SEMICOLON(;)”.

Keluaran (output)

```
<program>
├── <program-header>
│   ├── KEYWORD(program)
│   ├── IDENTIFIER(Hello)
│   └── SEMICOLON(;)
├── <declaration-part>
│   └── <var-declaration>
│       ├── KEYWORD(variabel)
│       ├── <identifier-list>
│       │   ├── IDENTIFIER(a)
│       │   ├── COMMA(,)
│       │   └── IDENTIFIER(b)
│       ├── COLON(:)
│       ├── <type>
│       │   └── KEYWORD(integer)
│       └── SEMICOLON(;)
├── <compound-statement>
│   ├── KEYWORD(mulai)
│   ├── <statement-list>
│   │   ├── <assignment-statement>
│   │   │   ├── IDENTIFIER(a)
│   │   │   ├── ASSIGN_OPERATOR(:=)
│   │   │   └── <expression>
│   │   │       └── <simple-expression>
│   │   │           └── <term>
│   │   │               └── <factor>
│   │   │                   └── NUMBER(5)
│   │   ├── SEMICOLON(;)
│   │   ├── <assignment-statement>
│   │   │   ├── IDENTIFIER(b)
│   │   │   ├── ASSIGN_OPERATOR(:=)
│   │   │   └── <expression>
│   │   │       ├── <simple-expression>
│   │   │       │   ├── <term>
│   │   │       │   │   └── <factor>
│   │   │       │   │       └── IDENTIFIER(a)
│   │   │       │   ├── ARITHMETIC_OPERATOR(+)
│   │   │       │   └── <term>
│   │   │       │       └── <factor>
│   │   │       │           └── NUMBER(10)
│   │   ├── SEMICOLON(;)
│   │   ├── <procedure-call>
│   │   │   ├── KEYWORD(writeln)
```


LPARENTHESIS(()
<parameter-list>
<expression>
<simple-expression>
<term>
<factor>
STRING_LITERAL('Result = ')
COMMA(,)
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(b)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(selesai)
DOT(.)

2. LANDASAN TEORI

2.1. Syntax Analyzer

Syntax Analyzer adalah tahap setelah Lexical Analysis dalam proses kompilasi, yang bertugas memeriksa apakah token-token yang dihasilkan oleh lexer tersusun sesuai dengan aturan grammar bahasa pemrograman. Sederetan token yang tidak mengikuti aturan sintaks akan dilaporkan sebagai kesalahan sintaks (syntax error). Secara logika deretan token yang bersesuaian dengan sintaks tertentu akan dinyatakan sebagai pohon parsing (parse tree).

Tahap-tahap dalam syntax analyzer mencakup:

1. Parsing, dimana syntax analyzer nganalisis token berdasarkan aturan grammar dan membangun Parse Tree.
2. Error handling, dimana syntax analyzer menemukan dan melaporkan error sesuai lokasinya.
3. Symbol table creation, dimana syntax analyzer menyimpan informasi simbol yang telah dibaca.

2.2. Parse Tree

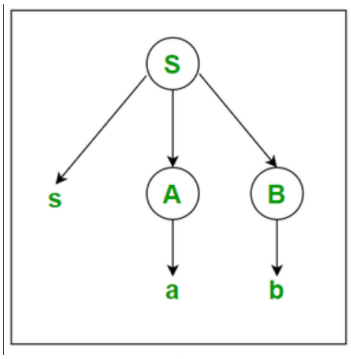
Parse tree adalah sebuah struktur pohon yang digunakan dalam compiler untuk menggambarkan bagaimana sebuah string atau kode program dibentuk berdasarkan aturan grammar tertentu. Struktur ini dibuat pada tahap parsing. Daun pohon dalam parse tree harus berupa simbol terminal, sementara node lainnya adalah simbol non-terminal. Jika pohon dibaca dari kiri ke kanan, pohon akan menghasilkan kembali input hasilnya.

Parse tree memiliki beberapa kegunaan penting dalam proses kompilasi. Struktur ini membantu melakukan analisis sintaks dengan menunjukkan secara jelas bagaimana input mengikuti aturan grammar bahasa. Selain itu, parse tree menyediakan representasi data di memori yang terorganisasi sesuai dengan struktur bahasa, sehingga memudahkan tahap-tahap lanjutan dalam compiler. Keuntungan lainnya adalah parse tree memungkinkan compiler melakukan beberapa kali pemrosesan terhadap data tanpa perlu mengulang proses parsing dari awal.

Sebagai contoh, ada grammar sebagai berikut

```
S -> sAB
A -> a
B -> b
```

Jika input string adalah “sab”, parse treenya berbentuk



Gambar 2.1 Parse tree untuk input “sab”

Sumber: [4]

2.3. Recursive Descent

Algoritma recursive descent parser adalah algoritma parser top-down yang menggunakan fungsi-fungsi rekursif, satu fungsi untuk setiap aturan grammar. Algoritma ini akan membaca input dari kiri ke kanan dan mencoba mencocokkannya dengan produksi grammar untuk membentuk parse tree. Algoritma ini mudah diimplementasikan sehingga sering digunakan untuk compiler kecil atau interpreter sederhana.

```
S()
{     Choose any S production, S ->X1X2…..Xk;
      for (i = 1 to k)
      {
          If ( Xi is a non-terminal)
          Call procedure Xi();
          else if ( Xi equals the current input, increment input)
          Else /* error has occurred, backtrack and try another
possibility */
      }
}
```

III. PERANCANGAN & IMPLEMENTASI

3.1. Perancangan

3.1.1. Aturan Grammar

Notasi:

`<non-terminal>`	Simbol non-terminal
`TERMINAL`	Simbol terminal (Token dari Lexer)
`::=`	“Didefinisikan sebagai”
`|`	Pilihan (OR)
`[ ... ]`	Opsional (0 atau 1 kali)
`{ ... }`	Repetisi (0 atau lebih kali)
`( ... )`	Pengelompokan
ε	Empty string / tidak ada

1. **Struktur Program Utama**

   Aturan-aturan ini mendefinisikan struktur keseluruhan dari sebuah program PASCAL-S.

```
<program> ::= <program-header> <block> DOT

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

<block> ::= <declaration-part> <compount-statement>
```

2. **Bagian Deklarasi**

   Aturan-aturan untuk mendeklarasikan konstanta, tipe, variabel, dan subprogram.

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

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

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

<var-declaration> ::=
     KEYWORD(variabel) (<identifier-list> COLON <type>
     SEMICOLON)+

<identifier-list> ::= IDENTIFIER { COMMA IDENTIFIER }
```

3. Definisi Tipe

Aturan untuk mendefinisikan tipe data, baik sederhana maupun larik (*array*).

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

<simple-type> ::=
     KEYWORD(integer)
     | KEYWORD(real)
     | KEYWORD(boolean)
     | KEYWORD(char)

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

<range> ::= <expression> RANGE_OPERATOR <expression>
```

4. Deklarasi Subprogram (Prosedur & Fungsi)

Aturan untuk mendefinisikan prosedur dan fungsi.

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

<procedure-declaration> ::=
     KEYWORD(prosedur) IDENTIFIER [ <formal-parameter-list>
     ] SEMICOLON <block> SEMICOLON

<function-declaration> ::=
```

```
     KEYWORD(fungsi) IDENTIFIER [ <formal-parameter-list> ]
     COLON <type> SEMICOLON <block> SEMICOLON

<formal-parameter-list> ::=
     LPARENTHESIS <parameter-group> { SEMICOLON
     <parameter-group> } RPARENTHESIS

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

5. Pernyataan (Statements)

Aturan-aturan yang mendefinisikan berbagai jenis pernyataan yang dapat dieksekusi.

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

<statement-list> ::= <statement> { SEMICOLON <statement> }

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

<empty-statement> ::= ε

<assignment-statement> ::=
     IDENTIFIER ASSIGN_OPERATOR <expression>

<procedure-call> ::=
     IDENTIFIER [ LPARENTHESIS <parameter-list> RPARENTHESIS
     ]

<parameter-list> ::= <expression> { COMMA <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>
     <direction> <expression> KEYWORD(lakukan) <statement>

<direction> ::= KEYWORD(ke) | KEYWORD(turun-ke)
```

6. Ekspresi (Expressions)

Aturan-aturan yang mendefinisikan ekspresi, operator, dan operand, dengan memperhatikan *operator precedence*.

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

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

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

<factor> ::=
     IDENTIFIER
     | NUMBER
     | CHAR_LITERAL
     | STRING_LITERAL
     | <function-call>
     | LPARENTHESIS <expression> RPARENTHESIS
     | LOGICAL_OPERATOR(tidak) <factor>

<function-call> ::=
     IDENTIFIER LPARENTHESIS [ <parameter-list> ]
     RPARENTHESIS

<sign> ::=
     ARITHMETIC_OPERATOR(+) | ARITHMETIC_OPERATOR(-)

<relational-operator> ::=
     RELATIONAL_OPERATOR(=)
     | RELATIONAL_OPERATOR(<>)
     | RELATIONAL_OPERATOR(<)
     | RELATIONAL_OPERATOR(<=)
     | RELATIONAL_OPERATOR(>)
     | RELATIONAL_OPERATOR(>=)

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

```
<multiplicative-operator> ::=
      ARITHMETIC_OPERATOR(*)
      | ARITHMETIC_OPERATOR(/)
      | ARITHMETIC_OPERATOR(bagi)
      | ARITHMETIC_OPERATOR(mod)
      | LOGICAL_OPERATOR(dan)
```

3.2. Implementasi

- node.py

Kelas node adalah representasi dasar untuk membangun parse tree. Setiap objek node memiliki label yang menyimpan nama node, token yang menyimpan objek token terkait dengan node, dan children yang menyimpan daftar anak yang dimiliki node saat ini.

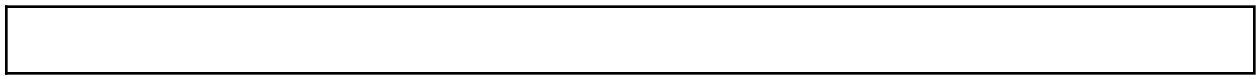
```python
from src.common.pascal_token import Token

class Node:
    def __init__(self, label, token=None):
        self.label: str = label
        self.token: Token | None = token
        self.children: list[Node] = []

    def add_children(self, node):
        if node:
            self.children.append(node)
        return node

    def print_tree(self, prefix: str = "", is_last: bool = True):
        connector = "└── " if is_last else "├── "
        if self.token:
            print(prefix + connector + f"{self.label}({self.token.value})")
        else:
            print(prefix + connector + f"{self.label}")

        child_prefix = prefix + ("    " if is_last else "│   ")
        for i, child in enumerate(self.children):
            is_last_child = (i == len(self.children) - 1)
            child.print_tree(child_prefix, is_last_child)
```

- Fungsi dasar kelas parser

```
class Parser:
    def __init__(self, tokens: list[Token], raise_on_error: bool = False):
        self.tokens = tokens
        self.current_index = 0
        self.errors = []
        self.raise_on_error = raise_on_error

    def peek(self) -> Token | None:
        # lihat token saat ini tanpa mengonsumsi
        if self.current_index < len(self.tokens):
            return self.tokens[self.current_index]
        return None

    def consume_token(self) -> Token | None:
        # ngambil token saat ini dan berpindah ke token berikutnya
        token = self.peek()
        if token:
            self.current_index += 1
        return token

    def match_token(self, expected_type: str, expected_value: str | None = None) ->
Token | None:
        # mastiin token saat ini sesuai dengan yg diharapkan.
        # kalau ngga, muncul error syntax
        token = self.peek()
        if token is None:
            self.error(f"{expected_type}({expected_value})" if expected_value else
expected_type, None)
            return None

        if token.token_type == expected_type and (expected_value is None or
token.value.lower() == expected_value.lower()):
            return self.consume_token()
        else:
            self.error(
                f"{expected_type}({expected_value})" if expected_value else
expected_type,
                token
            )
            return None

    def error(self, expected: str, actual_token: Token | None):
        actual_desc = self._fmt_token(actual_token)
        line, col = (actual_token.line, actual_token.column) if actual_token else
(None, None)
```

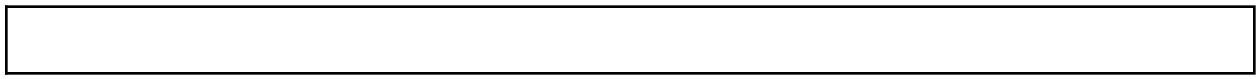
```
        msg = f"Syntax error: expected {expected}, but got {actual_desc}"
        logging.error(msg)
        self.errors.append(msg)
        if self.raise_on_error:
            raise TokenUnexpectedError(expected, actual_desc, line, col)

    def _fmt_token(self, tok: Token | None) -> str:
        if tok is None:
            return "EOF"
        return f"{tok.token_type}({tok.value}) @ {tok.line}:{tok.column}
```

- blabla

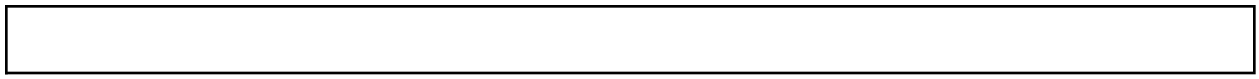


- blabla

- blabla



- blabla



IV. Pengujian

4.1. Pengujian 0

Tujuan pengujian: Untuk memvalidasi bahwa *parser* dapat mengenali struktur program PASCAL-S yang paling minimal. Pengujian ini memastikan integrasi dasar antara fungsi parse_program, parse_program_header, dan parse_compound_statement (kosong) berjalan dengan sukses tanpa *crash*.

File: 0-smoke-test.pas

Input

```
program tes;
mulai
selesai.
```

Output

```
KEYWORD(program)
IDENTIFIER(tes)
SEMICOLON(;)
KEYWORD(mulai)
KEYWORD(selesai)
DOT(.)

START PARSING...

└── <program>
    ├── <program-header>
    │   ├── KEYWORD(program)
    │   ├── IDENTIFIER(tes)
    │   └── SEMICOLON(;)
    ├── <compound-statement>
    │   ├── KEYWORD(mulai)
    │   └── KEYWORD(selesai)
    └── DOT(.)

FINISH PARSING...
```

4.2. Pengujian 1

Tujuan pengujian: Untuk memvalidasi kemampuan *parser* dalam mem-parsing <declaration-part> yang berisi variabel sederhana. Pengujian mengujia <compound-statement> yang berisi satu <assigment-statement>, serta memastikan penanganan *statement list* dan semicolon dengan benar.

File: 1-basic.pas

Input

```
program Basic;
variabel
   x: integer;
mulai
   x := 10;
selesai.
```

Output

```
KEYWORD(program)
IDENTIFIER(Basic)
SEMICOLON(;)
KEYWORD(variabel)
IDENTIFIER(x)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(mulai)
IDENTIFIER(x)
ASSIGN_OPERATOR(:=)
NUMBER(10)
SEMICOLON(;)
KEYWORD(selesai)
DOT(.)

START PARSING...

└── <program>
    ├── <program-header>
    │   ├── KEYWORD(program)
    │   ├── IDENTIFIER(Basic)
    │   └── SEMICOLON(;)
    ├── <declaration-part>
    │   └── <var-declaration>
    │       ├── KEYWORD(variabel)
    │       ├── <identifier-list>
    │       │   └── IDENTIFIER(x)
    │       ├── COLON(:)
    │       ├── <type>
    │       │   └── KEYWORD(integer)
    │       └── SEMICOLON(;)
    ├── <compound-statement>
    │   ├── KEYWORD(mulai)
    │   ├── <assignment-statement>
    │   │   ├── IDENTIFIER(x)
    │   │   ├── ASSIGN_OPERATOR(:=)
    │   │   └── <expression>
    │   │       └── <simple-expression>
    │   │           └── <term>
    │   │               └── <factor>
    │   │                   └── NUMBER(10)
    │   ├── SEMICOLON(;)
    │   └── KEYWORD(selesai)
```

```
    └── DOT(.)

FINISH PARSING...
```

4.3. Pengujian 2

Tujuan pengujian: Untuk memvalidasi kemampuan *parser* dalam menangani ekspresi aritmatika dan logika. Pengujian ini secara spesifik menguji implementasi operator precedence dan *parsing* ekspresi di dalam tanda kurung. Pengujian ini juga memvalidasi *parsing* struktur kontrol jika-maka-selain-itu.

File: 2-expression.pas

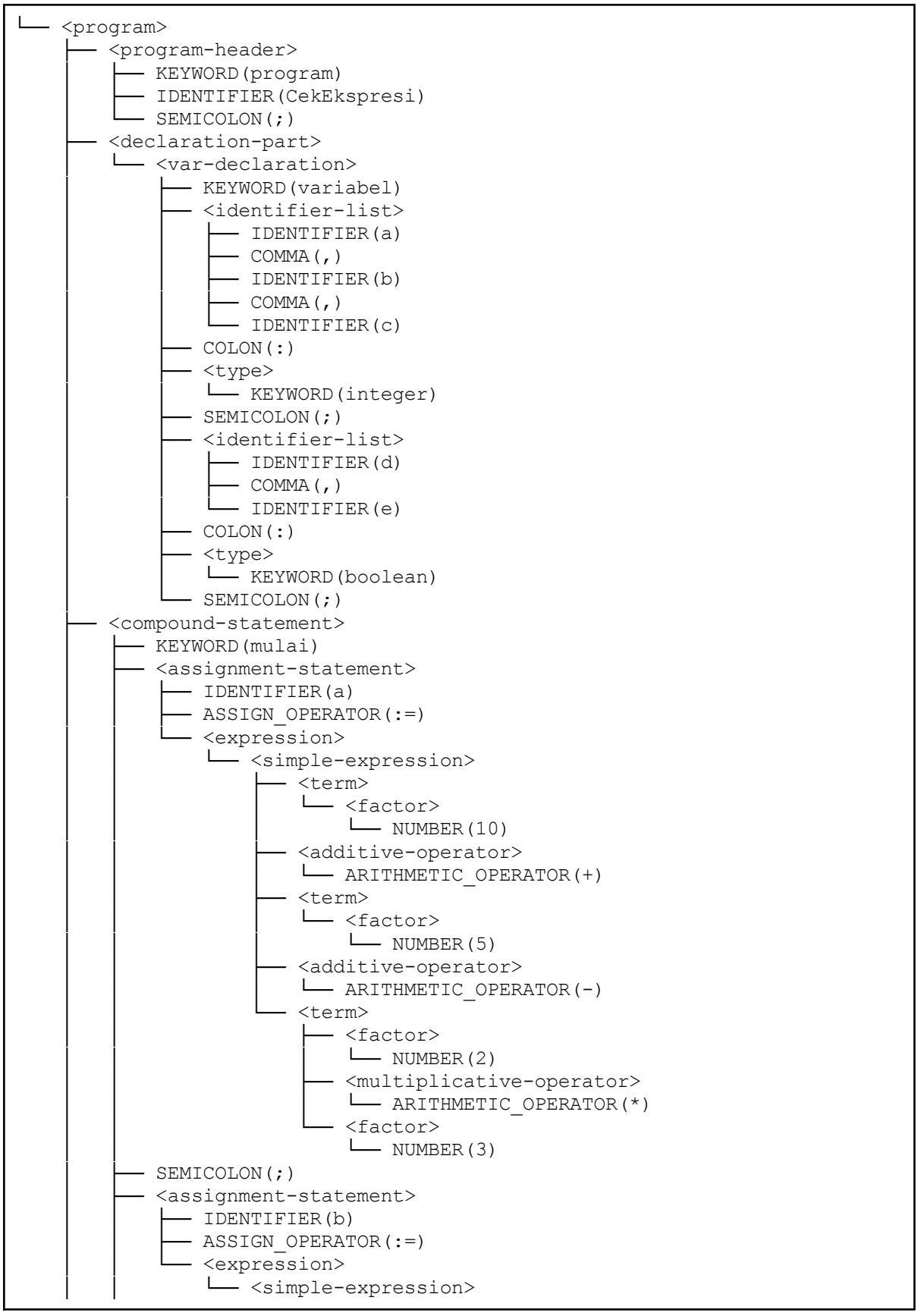
Input

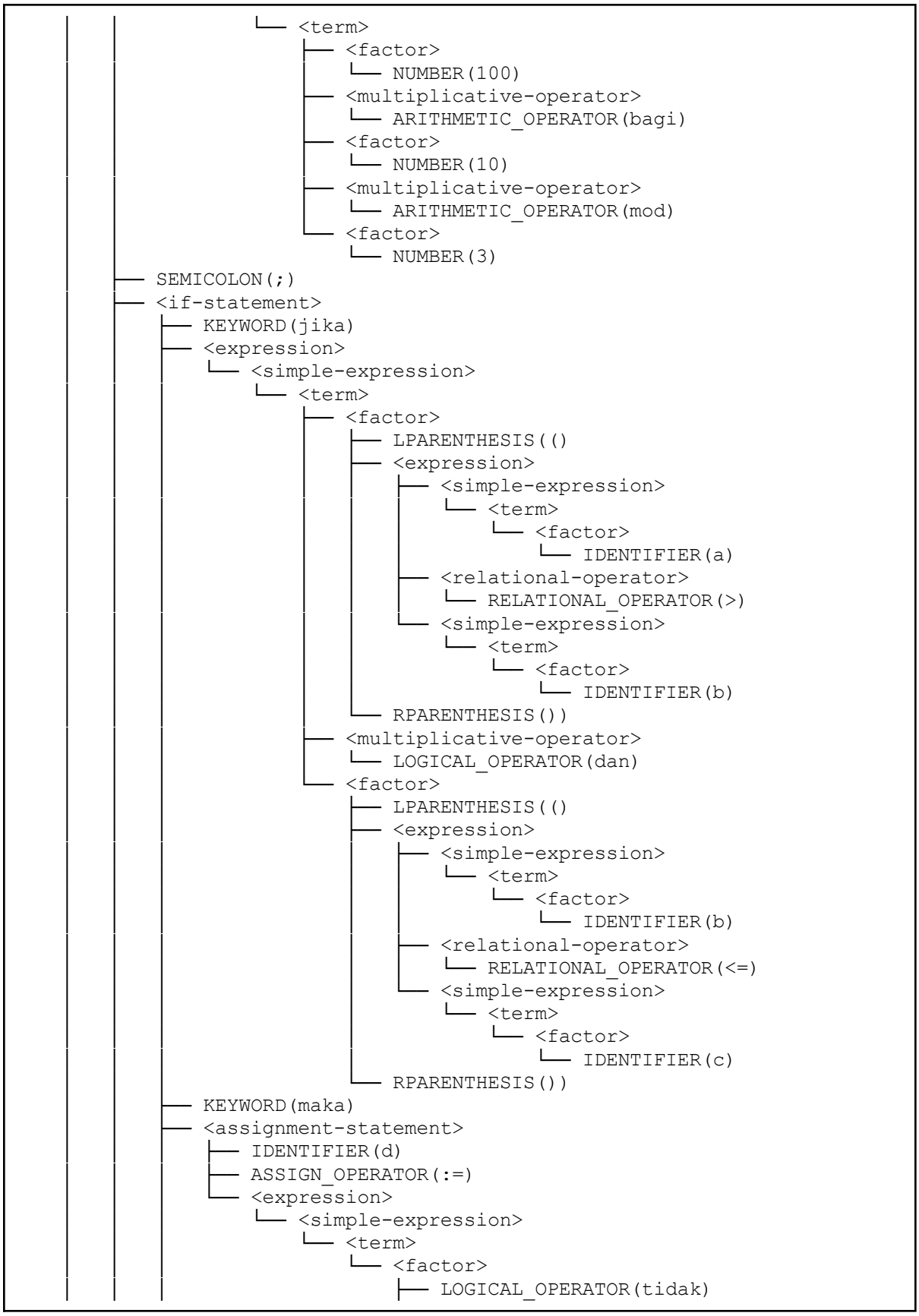
```
program CekEkspresi;
variabel
   a, b, c: integer;
   d, e: boolean;
mulai
   { Tes Aritmatika }
   a := 10 + 5 - 2 * 3;
   b := 100 bagi 10 mod 3;

   { Tes Relasional & Logika }
   jika (a > b) dan (b <= c) maka
      d := tidak e
   selain_itu
      e := (a <> c) atau (c = 10);
selesai.
```

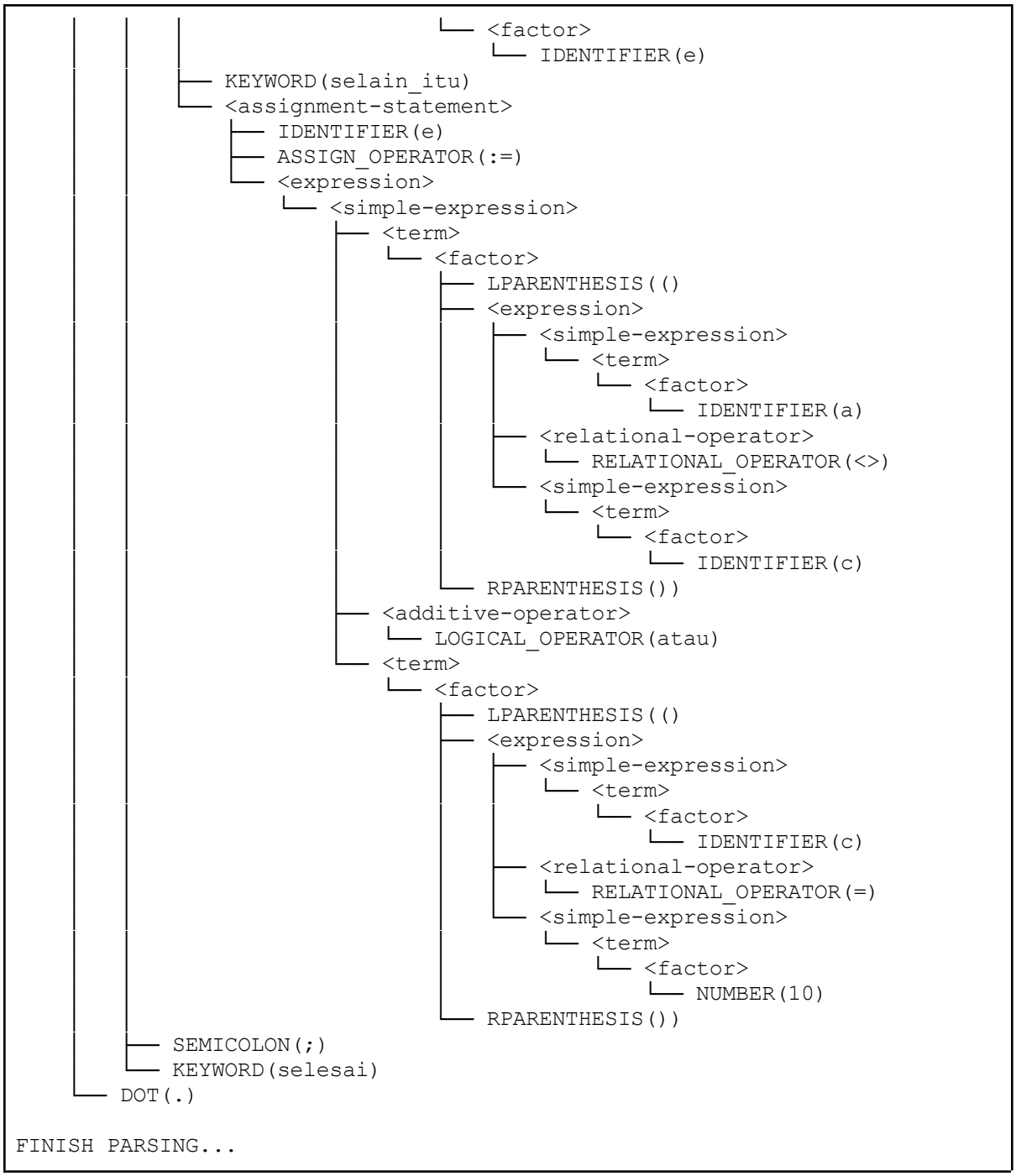
Output

```
KEYWORD(program)
IDENTIFIER(CekEkspresi)
SEMICOLON(;)
KEYWORD(variabel)
IDENTIFIER(a)
COMMA(,)
IDENTIFIER(b)
COMMA(,)
IDENTIFIER(c)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(d)
COMMA(,)
IDENTIFIER(e)
COLON(:)
KEYWORD(boolean)
SEMICOLON(;)
```

```
KEYWORD(mulai)
IDENTIFIER(a)
ASSIGN_OPERATOR(:=)
NUMBER(10)
ARITHMETIC_OPERATOR(+)
NUMBER(5)
ARITHMETIC_OPERATOR(-)
NUMBER(2)
ARITHMETIC_OPERATOR(*)
NUMBER(3)
SEMICOLON(;)
IDENTIFIER(b)
ASSIGN_OPERATOR(:=)
NUMBER(100)
ARITHMETIC_OPERATOR(bagi)
NUMBER(10)
ARITHMETIC_OPERATOR(mod)
NUMBER(3)
SEMICOLON(;)
KEYWORD(jika)
LPARENTHESIS(()
IDENTIFIER(a)
RELATIONAL_OPERATOR(>)
IDENTIFIER(b)
RPARENTHESIS())
LOGICAL_OPERATOR(dan)
LPARENTHESIS(()
IDENTIFIER(b)
RELATIONAL_OPERATOR(<=)
IDENTIFIER(c)
RPARENTHESIS())
KEYWORD(maka)
IDENTIFIER(d)
ASSIGN_OPERATOR(:=)
LOGICAL_OPERATOR(tidak)
IDENTIFIER(e)
KEYWORD(selain_itu)
IDENTIFIER(e)
ASSIGN_OPERATOR(:=)
LPARENTHESIS(()
IDENTIFIER(a)
RELATIONAL_OPERATOR(<>)
IDENTIFIER(c)
RPARENTHESIS())
LOGICAL_OPERATOR(atau)
LPARENTHESIS(()
IDENTIFIER(c)
RELATIONAL_OPERATOR(=)
NUMBER(10)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(selesai)
DOT(.)

START PARSING...
```


└── <program>
├── <program-header>
├── KEYWORD(program)
├── IDENTIFIER(CekEkspresi)
└── SEMICOLON(;)
├── <declaration-part>
└── <var-declaration>
├── KEYWORD(variabel)
├── <identifier-list>
├── IDENTIFIER(a)
├── COMMA(,)
├── IDENTIFIER(b)
├── COMMA(,)
└── IDENTIFIER(c)
├── COLON(:)
├── <type>
└── KEYWORD(integer)
├── SEMICOLON(;)
├── <identifier-list>
├── IDENTIFIER(d)
├── COMMA(,)
└── IDENTIFIER(e)
├── COLON(:)
├── <type>
└── KEYWORD(boolean)
└── SEMICOLON(;)
├── <compound-statement>
├── KEYWORD(mulai)
├── <assignment-statement>
├── IDENTIFIER(a)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
├── <term>
└── <factor>
└── NUMBER(10)
├── <additive-operator>
└── ARITHMETIC_OPERATOR(+)
├── <term>
└── <factor>
└── NUMBER(5)
├── <additive-operator>
└── ARITHMETIC_OPERATOR(-)
└── <term>
├── <factor>
└── NUMBER(2)
├── <multiplicative-operator>
└── ARITHMETIC_OPERATOR(*)
└── <factor>
└── NUMBER(3)
├── SEMICOLON(;)
├── <assignment-statement>
├── IDENTIFIER(b)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>


└── <term>
├── <factor>
└── NUMBER(100)
├── <multiplicative-operator>
└── ARITHMETIC_OPERATOR(bagi)
├── <factor>
└── NUMBER(10)
├── <multiplicative-operator>
└── ARITHMETIC_OPERATOR(mod)
└── <factor>
└── NUMBER(3)
├── SEMICOLON(;)
├── <if-statement>
├── KEYWORD(jika)
├── <expression>
└── <simple-expression>
└── <term>
├── <factor>
├── LPARENTHESIS(()
├── <expression>
├── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(a)
├── <relational-operator>
└── RELATIONAL_OPERATOR(>)
└── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(b)
└── RPARENTHESIS())
├── <multiplicative-operator>
└── LOGICAL_OPERATOR(dan)
└── <factor>
├── LPARENTHESIS(()
├── <expression>
├── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(b)
├── <relational-operator>
└── RELATIONAL_OPERATOR(<=)
└── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(c)
└── RPARENTHESIS())
├── KEYWORD(maka)
├── <assignment-statement>
├── IDENTIFIER(d)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
├── LOGICAL_OPERATOR(tidak)



4.4. Pengujian 3

Tujuan pengujian: Untuk memvalidasi kemampuan *parser* dalam mem-parsing keseluruhan <declaration-part> secara berurutan, mencakup konstanta, tipe (termasuk larik), variabel, prosedur, dan fungsi. Pengujian ini juga memvalidasi penggunaan tipe data *custom* (misalnya MyArray) dan pemanggilan prosedur serta fungsi dari dalam blok program utama.

File: 3-all-declaration.pas

Input

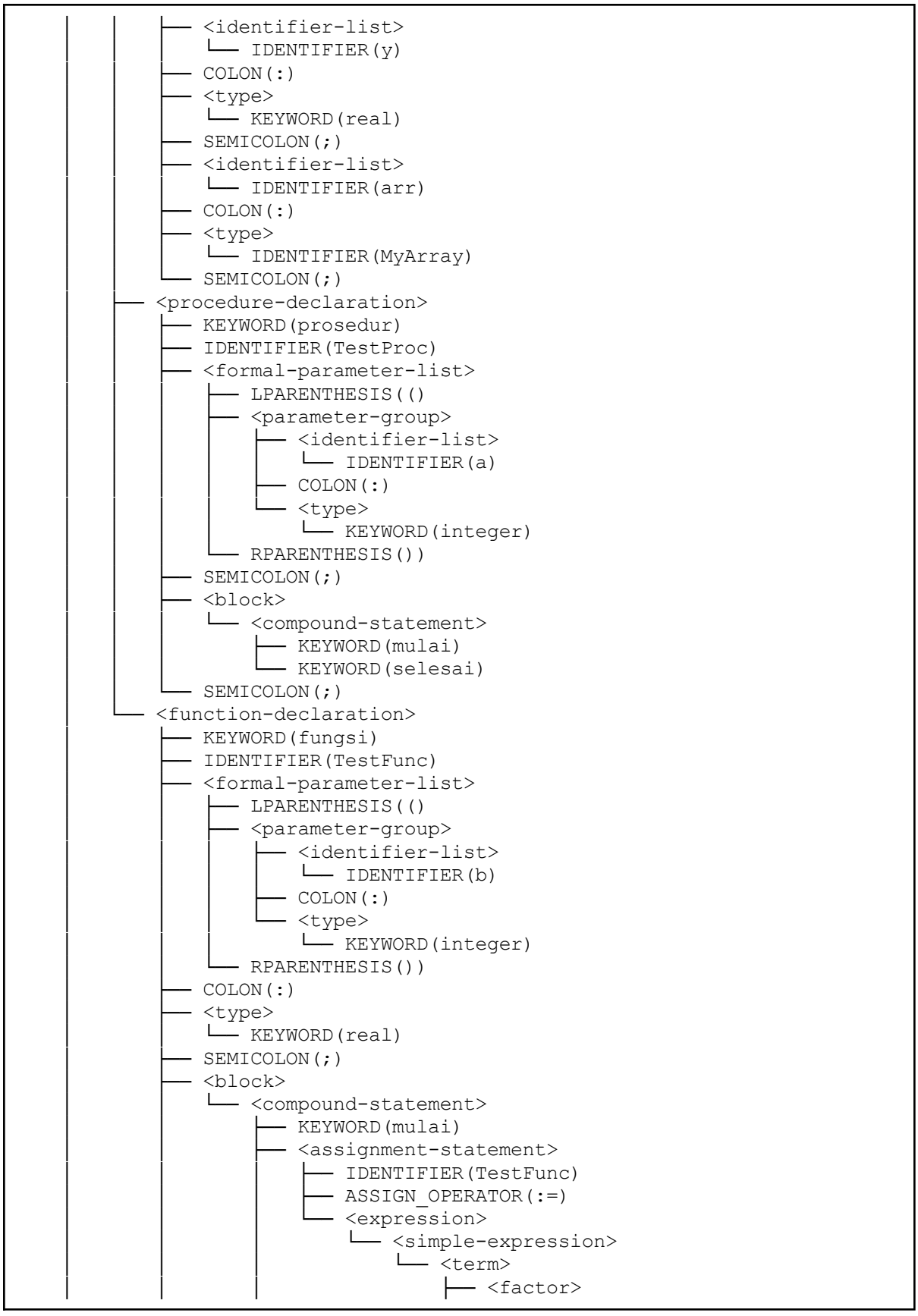
```
program AllDeclarations;
konstanta
   PI = 3.14;
tipe
   MyArray = larik [1..10] dari real;
variabel
   x: integer;
   y: real;
   arr: MyArray;

prosedur TestProc(a: integer);
mulai
   { prosedur ini tidak melakukan apa-apa }
selesai;

fungsi TestFunc(b: integer): real;
mulai
   TestFunc := b * PI;
selesai;

mulai
   { Blok utama }
   x := 10;
   y := TestFunc(x);
   TestProc(x);
selesai.
```

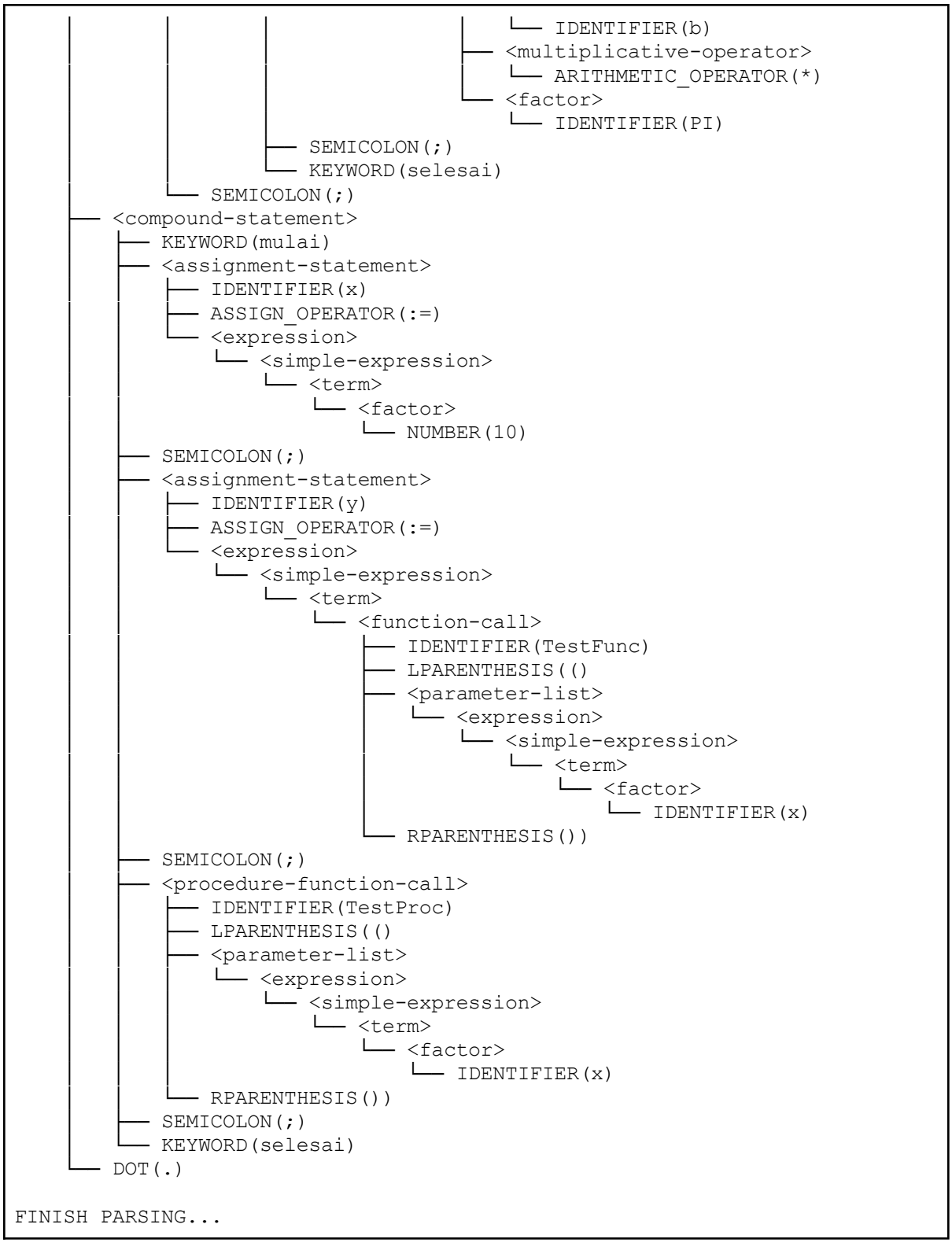
Output

```
KEYWORD(program)
IDENTIFIER(AllDeclarations)
SEMICOLON(;)
KEYWORD(konstanta)
IDENTIFIER(PI)
RELATIONAL_OPERATOR(=)
NUMBER(3.14)
SEMICOLON(;)
KEYWORD(tipe)
IDENTIFIER(MyArray)
RELATIONAL_OPERATOR(=)
KEYWORD(larik)
LBRACKET([)
NUMBER(1)
RANGE_OPERATOR(..)
NUMBER(10)
RBRACKET(])
KEYWORD(dari)
KEYWORD(real)
SEMICOLON(;)
KEYWORD(variabel)
IDENTIFIER(x)
```

```
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
IDENTIFIER(y)
COLON(:)
KEYWORD(real)
SEMICOLON(;)
IDENTIFIER(arr)
COLON(:)
IDENTIFIER(MyArray)
SEMICOLON(;)
KEYWORD(prosedur)
IDENTIFIER(TestProc)
LPARENTHESIS(()
IDENTIFIER(a)
COLON(:)
KEYWORD(integer)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(mulai)
KEYWORD(selesai)
SEMICOLON(;)
KEYWORD(fungsi)
IDENTIFIER(TestFunc)
LPARENTHESIS(()
IDENTIFIER(b)
COLON(:)
KEYWORD(integer)
RPARENTHESIS())
COLON(:)
KEYWORD(real)
SEMICOLON(;)
KEYWORD(mulai)
IDENTIFIER(TestFunc)
ASSIGN_OPERATOR(:=)
IDENTIFIER(b)
ARITHMETIC_OPERATOR(*)
IDENTIFIER(PI)
SEMICOLON(;)
KEYWORD(selesai)
SEMICOLON(;)
KEYWORD(mulai)
IDENTIFIER(x)
ASSIGN_OPERATOR(:=)
NUMBER(10)
SEMICOLON(;)
IDENTIFIER(y)
ASSIGN_OPERATOR(:=)
IDENTIFIER(TestFunc)
LPARENTHESIS(()
IDENTIFIER(x)
RPARENTHESIS())
SEMICOLON(;)
IDENTIFIER(TestProc)
LPARENTHESIS(()
IDENTIFIER(x)
```

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

START PARSING...

└── <program>
    ├── <program-header>
    │   ├── KEYWORD(program)
    │   ├── IDENTIFIER(AllDeclarations)
    │   └── SEMICOLON(;)
    ├── <declaration-part>
    │   ├── <const-declaration>
    │   │   ├── KEYWORD(konstanta)
    │   │   ├── IDENTIFIER(PI)
    │   │   ├── RELATIONAL_OPERATOR(=)
    │   │   ├── <expression>
    │   │   │   └── <simple-expression>
    │   │   │       └── <term>
    │   │   │           └── <factor>
    │   │   │               └── NUMBER(3.14)
    │   │   └── SEMICOLON(;)
    │   ├── <type-declaration>
    │   │   ├── KEYWORD(tipe)
    │   │   ├── IDENTIFIER(MyArray)
    │   │   ├── RELATIONAL_OPERATOR(=)
    │   │   ├── <type>
    │   │   │   └── <array-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(real)
    │   │   └── SEMICOLON(;)
    │   ├── <var-declaration>
    │   │   ├── KEYWORD(variabel)
    │   │   ├── <identifier-list>
    │   │   │   └── IDENTIFIER(x)
    │   │   ├── COLON(:)
    │   │   ├── <type>
    │   │   │   └── KEYWORD(integer)
    │   │   ├── SEMICOLON(;)
```


├── <identifier-list>
└── IDENTIFIER(y)
├── COLON(:)
├── <type>
└── KEYWORD(real)
├── SEMICOLON(;)
├── <identifier-list>
└── IDENTIFIER(arr)
├── COLON(:)
├── <type>
└── IDENTIFIER(MyArray)
└── SEMICOLON(;)
├── <procedure-declaration>
├── KEYWORD(prosedur)
├── IDENTIFIER(TestProc)
├── <formal-parameter-list>
├── LPARENTHESIS(()
├── <parameter-group>
├── <identifier-list>
└── IDENTIFIER(a)
├── COLON(:)
└── <type>
└── KEYWORD(integer)
└── RPARENTHESIS())
├── SEMICOLON(;)
├── <block>
└── <compound-statement>
├── KEYWORD(mulai)
└── KEYWORD(selesai)
└── SEMICOLON(;)
└── <function-declaration>
├── KEYWORD(fungsi)
├── IDENTIFIER(TestFunc)
├── <formal-parameter-list>
├── LPARENTHESIS(()
├── <parameter-group>
├── <identifier-list>
└── IDENTIFIER(b)
├── COLON(:)
└── <type>
└── KEYWORD(integer)
└── RPARENTHESIS())
├── COLON(:)
├── <type>
└── KEYWORD(real)
├── SEMICOLON(;)
├── <block>
└── <compound-statement>
├── KEYWORD(mulai)
├── <assignment-statement>
├── IDENTIFIER(TestFunc)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
└── <term>
├── <factor>


└── IDENTIFIER(b)
├── <multiplicative-operator>
└── ARITHMETIC_OPERATOR(*)
└── <factor>
└── IDENTIFIER(PI)
├── SEMICOLON(;)
└── KEYWORD(selesai)
└── SEMICOLON(;)
├── <compound-statement>
├── KEYWORD(mulai)
├── <assignment-statement>
├── IDENTIFIER(x)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(10)
├── SEMICOLON(;)
├── <assignment-statement>
├── IDENTIFIER(y)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
└── <term>
└── <function-call>
├── IDENTIFIER(TestFunc)
├── LPARENTHESIS(()
├── <parameter-list>
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(x)
└── RPARENTHESIS())
├── SEMICOLON(;)
├── <procedure-function-call>
├── IDENTIFIER(TestProc)
├── LPARENTHESIS(()
├── <parameter-list>
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(x)
└── RPARENTHESIS())
├── SEMICOLON(;)
└── KEYWORD(selesai)
└── DOT(.)
FINISH PARSING...

4.5. Pengujian 4

Tujuan pengujian: Untuk mendemonstrasikan kemampuar *parser* dalam menangani *error handling* ketika menemukan kesalahan sintaks (urutan token yang salah). Pengujian ini secara spesifik memvalidasi deteksi token yang hilang (misalnya RPARENTHESIS yang diharapkan) dan memastikan *parser* melaporkan pesan error yang informatif tanpa *crash*.

File: 4-errors.pas

Input

```
program ErrorTest;
mulai
   { Error: ')' hilang setelah 'Hello' }
   writeln('Hello';
selesai.
```

Output

```
ERROR:root:Syntax error: expected RPARENTHESIS()), but got SEMICOLON(;) @
4:19
ERROR:root:Syntax error: expected DOT(.), but got SEMICOLON(;) @ 4:19
KEYWORD(program)
IDENTIFIER(ErrorTest)
SEMICOLON(;)
KEYWORD(mulai)
IDENTIFIER(writeln)
LPARENTHESIS(()
STRING_LITERAL('Hello')
SEMICOLON(;)
KEYWORD(selesai)
DOT(.)

START PARSING...

└── <program>
    ├── <program-header>
    │   ├── KEYWORD(program)
    │   ├── IDENTIFIER(ErrorTest)
    │   └── SEMICOLON(;)
    └── <compound-statement>
        └── KEYWORD(mulai)

FINISH PARSING...
```

4.6. Pengujian 5

Tujuan pengujian: Untuk memvalidasi kemampuan *parser* dalam mem-parsing semua jenis *statement loop*, termasuk <while-statement> (selama…lakukan) dan <for-statement> (varian ke dan turun_ke). Pengujian ini juga secara spesifik menguji kemampuan *parser* dalam

menangani struktur bersarang (nesting), seperti <for-statement> di dalam <while-statement>, yang juga berisi blok <compound-statement> di dalamnya.

File: 5-loops.pas

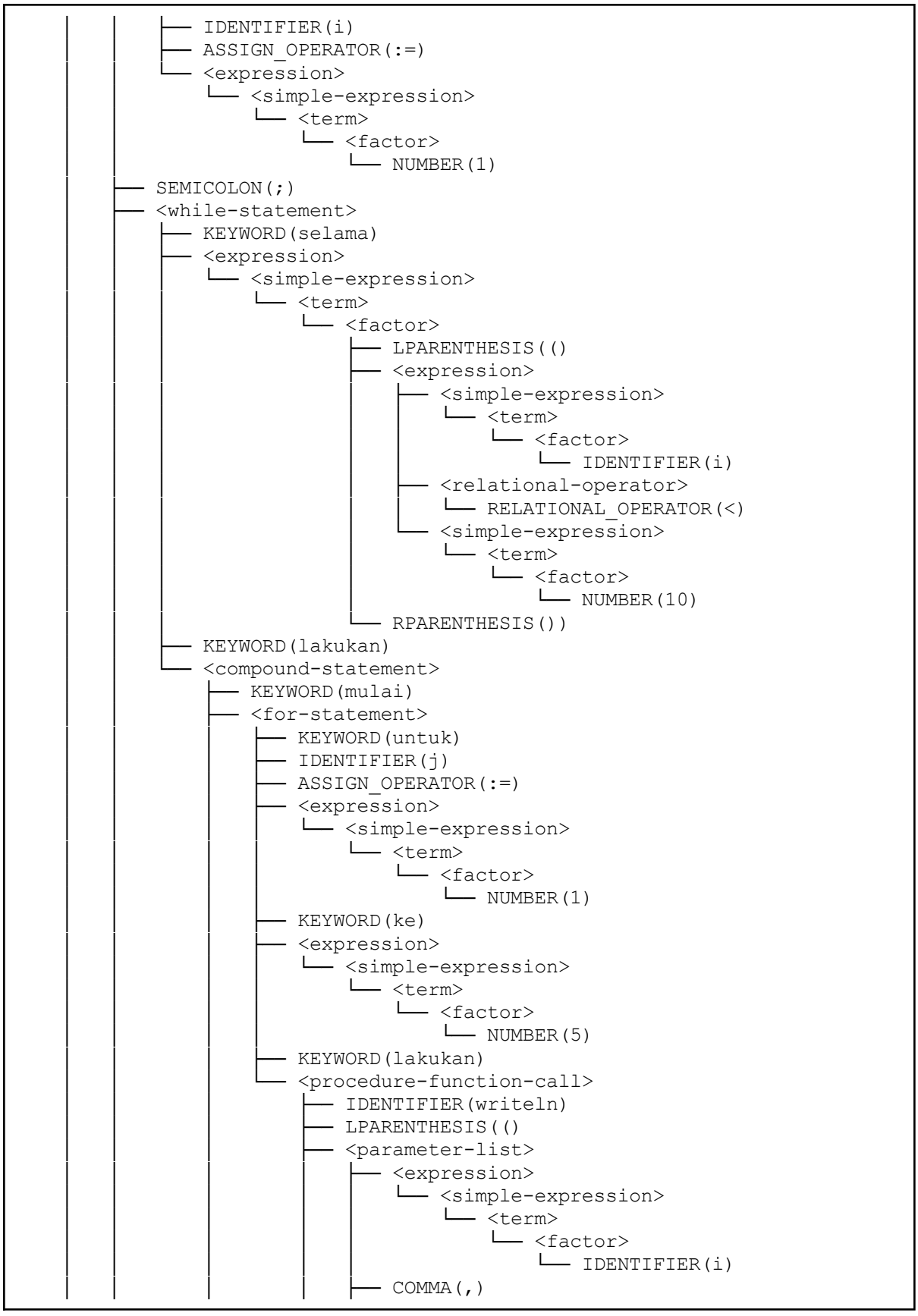
Input

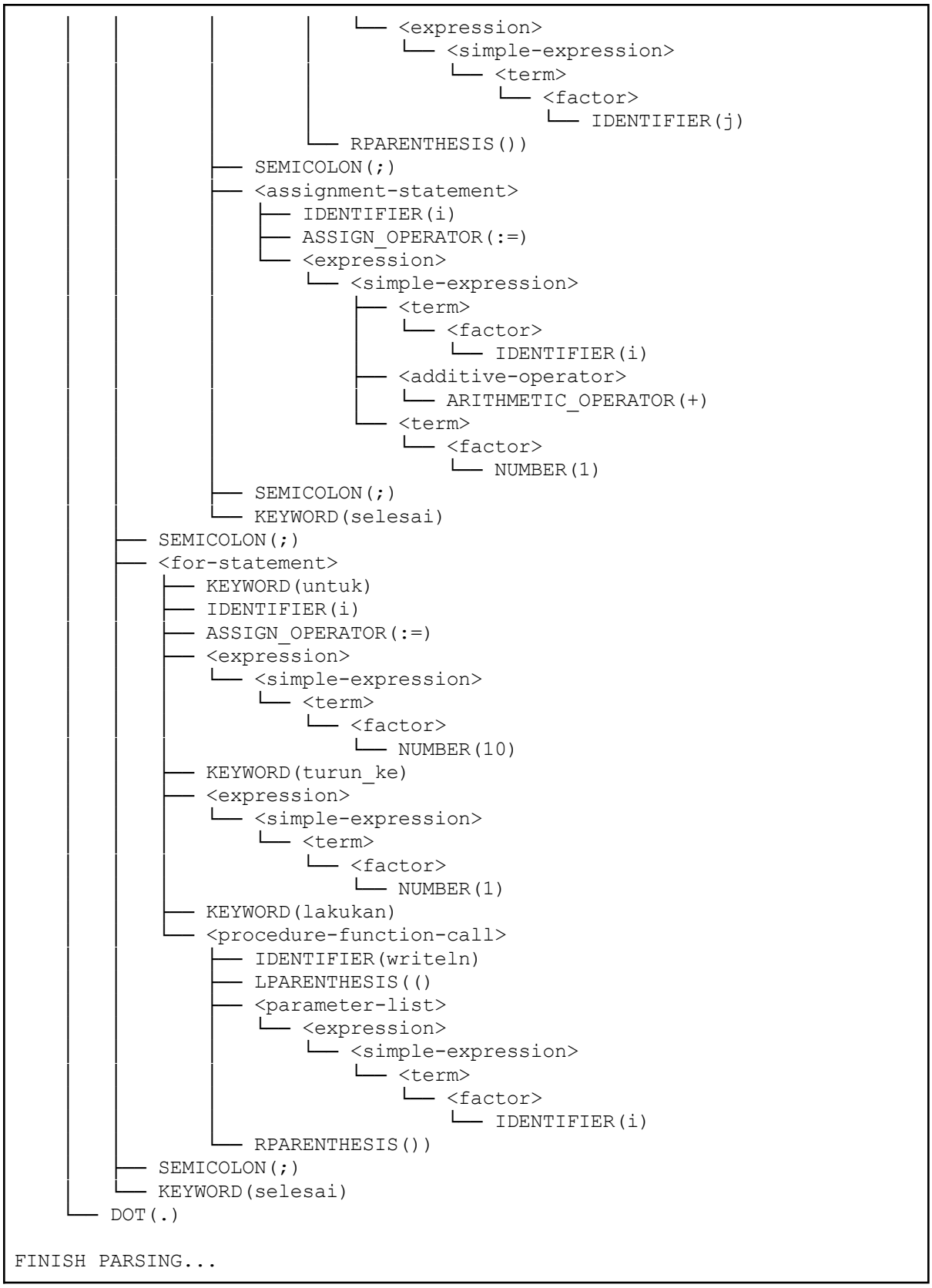
```
program TestLoops;
variabel
   i, j: integer;
mulai
   { Tes 1: <while-statement> }
   i := 1;
   selama (i < 10) lakukan
   mulai
      { Tes 2: <for-statement> (bersarang) }
      untuk j := 1 ke 5 lakukan
         writeln(i, j);

      i := i + 1;
   selesai;

   { Tes 3: <for-statement> dengan 'turun_ke' }
   untuk i := 10 turun_ke 1 lakukan
      writeln(i);
selesai.
```

Output

```
KEYWORD(program)
IDENTIFIER(TestLoops)
SEMICOLON(;)
KEYWORD(variabel)
IDENTIFIER(i)
COMMA(,)
IDENTIFIER(j)
COLON(:)
KEYWORD(integer)
SEMICOLON(;)
KEYWORD(mulai)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
NUMBER(1)
SEMICOLON(;)
KEYWORD(selama)
LPARENTHESIS(()
IDENTIFIER(i)
RELATIONAL_OPERATOR(<)
NUMBER(10)
RPARENTHESIS())
KEYWORD(lakukan)
KEYWORD(mulai)
KEYWORD(untuk)
IDENTIFIER(j)
```

```
ASSIGN_OPERATOR(:=)
NUMBER(1)
KEYWORD(ke)
NUMBER(5)
KEYWORD(lakukan)
IDENTIFIER(writeln)
LPARENTHESIS(()
IDENTIFIER(i)
COMMA(,)
IDENTIFIER(j)
RPARENTHESIS())
SEMICOLON(;)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
IDENTIFIER(i)
ARITHMETIC_OPERATOR(+)
NUMBER(1)
SEMICOLON(;)
KEYWORD(selesai)
SEMICOLON(;)
KEYWORD(untuk)
IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
NUMBER(10)
KEYWORD(turun_ke)
NUMBER(1)
KEYWORD(lakukan)
IDENTIFIER(writeln)
LPARENTHESIS(()
IDENTIFIER(i)
RPARENTHESIS())
SEMICOLON(;)
KEYWORD(selesai)
DOT(.)

START PARSING...

└── <program>
    ├── <program-header>
    │   ├── KEYWORD(program)
    │   ├── IDENTIFIER(TestLoops)
    │   └── SEMICOLON(;)
    ├── <declaration-part>
    │   └── <var-declaration>
    │       ├── KEYWORD(variabel)
    │       ├── <identifier-list>
    │       │   ├── IDENTIFIER(i)
    │       │   ├── COMMA(,)
    │       │   └── IDENTIFIER(j)
    │       ├── COLON(:)
    │       ├── <type>
    │       │   └── KEYWORD(integer)
    │       └── SEMICOLON(;)
    ├── <compound-statement>
    │   ├── KEYWORD(mulai)
    │   ├── <assignment-statement>
```


IDENTIFIER(i)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
NUMBER(1)
SEMICOLON(;)
<while-statement>
KEYWORD(selama)
<expression>
<simple-expression>
<term>
<factor>
LPARENTHESIS(()
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(i)
<relational-operator>
RELATIONAL_OPERATOR(<)
<simple-expression>
<term>
<factor>
NUMBER(10)
RPARENTHESIS())
KEYWORD(lakukan)
<compound-statement>
KEYWORD(mulai)
<for-statement>
KEYWORD(untuk)
IDENTIFIER(j)
ASSIGN_OPERATOR(:=)
<expression>
<simple-expression>
<term>
<factor>
NUMBER(1)
KEYWORD(ke)
<expression>
<simple-expression>
<term>
<factor>
NUMBER(5)
KEYWORD(lakukan)
<procedure-function-call>
IDENTIFIER(writeln)
LPARENTHESIS(()
<parameter-list>
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(i)
COMMA(,)


└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(j)
└── RPARENTHESIS())
├── SEMICOLON(;)
├── <assignment-statement>
├── IDENTIFIER(i)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
├── <term>
└── <factor>
└── IDENTIFIER(i)
├── <additive-operator>
└── 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)
├── KEYWORD(turun_ke)
├── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(1)
├── KEYWORD(lakukan)
└── <procedure-function-call>
├── IDENTIFIER(writeln)
├── LPARENTHESIS(()
├── <parameter-list>
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(i)
└── RPARENTHESIS())
├── SEMICOLON(;)
└── KEYWORD(selesai)
└── DOT(.)
FINISH PARSING...

V. KESIMPULAN DAN SARAN

5.1. Kesimpulan

5.2.Saran

LAMPIRAN

- Tautan *repository* GitHub : https://github.com/anellautari/DFC-Tubes-IF2224
- Tautan *workspace* diagram : Workspace Diagram DFA

PEMBAGIAN TUGAS

NAMA	NIM	TUGAS
Mayla Yaffa Ludmilla	13523050	
Anella Utari Gunadi	13523078	
Muhammad Edo Raduputu Aprima	13523096	
Athian Nugraha Muarajuang	13523106	

REFERENSI

[1] Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation. 3rd ed*, 2007. Tersedia: https://cdn-edunex.itb.ac.id/29161-Formal-Language-Theory-and-Automata/1629640939613_Introduction-to-Automata-Theory,-Languages,-and-Computation-Edition-3.pdf [Diakses: 11-Oktober-2025].

[2] Wirth, Niklaus. "PASCAL-S: A Subset and its implementation", Tersedia: http://pascal.hansotten.com/uploads/pascals/PASCAL-S%20A%20subset%20and%20its%20Implementation%20012.pdf

[3] tutorialspoint.com. *Compiler Design*. Tersedia: https://www.tutorialspoint.com/compiler_design/index.htm [Diakses 11-Oktober-2025]

[4] geeksforgeeks.com. *Introduction to Syntax Analysis in Compiler Design.* Tersedia: https://www.geeksforgeeks.org/compiler-design/introduction-to-syntax-analysis-in-compiler-design/ [Diakses 15 November 2025]

[5] geeksforgeeks.com. *Recursive Descent Parser.* Tersedia: https://www.geeksforgeeks.org/compiler-design/recursive-descent-parser/ [Diakses 16 November 2025]