Tugas Besar IF2224 Teori Bahasa Formal & Otomata

Syntax Analysis (parser)

Oleh:

Kelompok JJK - JanganJanganKecompile

Muhammad Raihaan Perdana	13523124
Danendra Shafi Athallah	13523136
Nathanael Rachmat	13523142
M. Abizzar Gamadrian	13523155

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika - Institut Teknologi Bandung

Jl. Ganesha 10, Bandung 40132

2025

Daftar Isi

Bab I
Landasan Teori

1.1 Kompiler dan Tahapannya

Bahasa pemrograman merupakan jembatan komunikasi antara manusia dengan komputer. Agar instruksi yang ditulis dalam bahasa tingkat tinggi seperti Pascal-S dapat dimengerti dan dieksekusi oleh mesin, diperlukan sebuah program penerjemah yang disebut **kompiler** (*compiler*). Kompiler secara sistematis menerjemahkan kode sumber (*source code*) menjadi kode mesin (*machine code*) atau format tingkat rendah lainnya tanpa mengubah logika fungsionalitasnya. Proses ini tidak hanya memungkinkan eksekusi program, tetapi juga berperan penting dalam optimasi dan deteksi kesalahan dasar, sehingga menghasilkan aplikasi yang efisien dan andal.

Proses kompilasi secara umum terbagi menjadi beberapa tahapan utama yang berurutan, di mana keluaran dari satu tahap menjadi masukan bagi tahap berikutnya. Tahapan-tahapan tersebut adalah:

1. **Analisis Leksikal (*Lexical Analysis*):** Tahap awal yang memecah kode sumber menjadi unit-unit leksikal terkecil yang disebut *token*.
2. **Analisis Sintaksis (*Syntax Analysis*):** Memeriksa struktur gramatikal dari urutan *token* untuk memastikan sesuai dengan aturan sintaks bahasa pemrograman, biasanya dalam bentuk *parse tree*.
3. **Analisis Semantik (*Semantic Analysis*):** Memeriksa makna dan konsistensi dari program, seperti memeriksa kesesuaian tipe data dalam sebuah operasi.
4. **Generasi Kode Perantara (*Intermediate Code Generation*):** Menghasilkan representasi program dalam bentuk abstrak yang mudah dioptimasi dan diterjemahkan ke berbagai arsitektur mesin.
5. **Interpretasi (*Interpreter*):** Mengeksekusi kode, baik kode perantara maupun kode mesin, untuk menghasilkan *output* akhir.

Milestone kedua dalam tugas besar ini berfokus secara eksklusif pada implementasi tahap kedua, yaitu Analisis Sintaksis.

1.2 Analisis Leksikal

Analisis leksikal adalah fondasi dari proses kompilasi. Pada tahap ini, rangkaian karakter mentah dari kode sumber dibaca dan dikelompokkan menjadi satuan-satuan bermakna yang disebut *token*. Komponen yang bertanggung jawab untuk melakukan tugas ini dikenal sebagai **penganalisis leksikal**, atau lebih umum disebut ***lexer*** atau ***scanner***.

Tujuan utama dari analisis leksikal adalah menyederhanakan tugas bagi tahap selanjutnya, yaitu analisis sintaksis (parsing). Daripada memproses karakter satu per satu, *parser* akan menerima aliran *token* yang lebih terstruktur, sehingga lebih mudah untuk mengenali struktur program. Selain itu, *lexer* juga bertugas untuk membuang elemen-elemen yang tidak relevan dengan logika program, seperti spasi, *tab*, dan baris baru.

1.3 Token

Token adalah unit leksikal terkecil dalam sebuah bahasa pemrograman yang memiliki makna. Setiap *token* yang dihasilkan oleh *lexer* umumnya terdiri dari dua komponen utama:

1. **Tipe Token (*Token Type*):** Kategori dari unit leksikal tersebut. Contohnya, KEYWORD, IDENTIFIER, NUMBER, atau ARITHMETIC_OPERATOR.
2. **Nilai Token (*Token Value*):** Rangkaian karakter aktual (disebut juga *lexeme*) yang cocok dengan pola tipe *token*. Contohnya, untuk *lexeme* "program", tipe *token*-nya adalah KEYWORD. Untuk *lexeme* "x", tipe *token*-nya adalah IDENTIFIER.

Sebagai ilustrasi, jika *lexer* memindai baris kode a := 5;, maka ia akan menghasilkan urutan *token* sebagai berikut: IDENTIFIER(a), ASSIGN_OPERATOR(:=), NUMBER(5), dan SEMICOLON(;)

1.4 Deterministic Finite Automaton (DFA)

Untuk mengenali pola-pola karakter dan mengelompokkannya menjadi *token*, *lexer* menggunakan model komputasi yang disebut **Deterministic Finite Automaton (DFA)**. Dalam teori bahasa formal, himpunan *token* dalam sebuah bahasa pemrograman dapat direpresentasikan sebagai **bahasa reguler**, yang secara matematis dapat dikenali oleh *finite automata*.

Secara formal, sebuah DFA didefinisikan sebagai 5-tuple $(Q, \Sigma, \delta, q_0, F)$, di mana:

- **Q** adalah himpunan hingga dari *state* (keadaan).

- **Σ** adalah himpunan hingga dari simbol input (alfabet).
- **δ** adalah fungsi transisi ($Q \times \Sigma \rightarrow Q$).
- $\mathbf{q_0}$ adalah *state* awal.
- **F** adalah himpunan *state* akhir (penerima).

Dalam konteks *lexer*, DFA bekerja dengan membaca kode sumber karakter demi karakter. Dimulai dari *state* awal, *lexer* akan berpindah dari satu *state* ke *state* lain berdasarkan karakter yang dibaca dan fungsi transisi yang telah didefinisikan. Ketika *lexer* mencapai *state* akhir, itu menandakan bahwa serangkaian karakter yang telah dibaca berhasil dikenali sebagai sebuah *token* yang valid. Implementasi *lexer* berbasis DFA adalah contoh nyata bagaimana model teoretis dari automata digunakan untuk memecahkan masalah praktis dalam konstruksi kompiler.

Bab II
Perancangan & Implementasi

2.1 Perancangan Grammar dan Parser

Parser merupakan komponen kedua dalam proses kompilasi. Setelah lexer menghasilkan deretan token, parser menerima input tersebut dan memverifikasi apakah urutan token sesuai dengan aturan sintaksis bahasa Pascal-S. Proses ini menggunakan context-free grammar yang menggambarkan struktur hierarkis dari program.

Implementasi parser dalam tugas besar ini menggunakan algoritma Recursive Descent Parsing. Algoritma ini merupakan pendekatan top-down yang dimulai dari simbol awal grammar, kemudian secara rekursif menurunkan setiap aturan produksi hingga mencapai token terminal. Setiap aturan produksi dalam grammar diimplementasikan sebagai sebuah fungsi terpisah dalam kode program. Grammar Pascal-S yang diimplementasikan terdiri dari hierarki aturan sebagai berikut:

Program dan Struktur Utama:

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

Deklarasi:

```
<declaration-part> ::= (<const-declaration>)* (<type-declaration>)*
(<var-declaration>)* (<subprogram-declaration>)*
<const-declaration> ::= KEYWORD(konstanta) (IDENTIFIER RELATIONAL_OPERATOR(=)
<const-value> SEMICOLON)+
<const-value> ::= NUMBER | STRING_LITERAL | IDENTIFIER
<type-declaration> ::= KEYWORD(tipe) (IDENTIFIER RELATIONAL_OPERATOR(=) <type>
SEMICOLON)+
<var-declaration> ::= KEYWORD(variabel) (<identifier-list> COLON <type>
SEMICOLON)+
<identifier-list> ::= IDENTIFIER (COMMA IDENTIFIER)*
```

Tipe Data:

```
<type> ::= KEYWORD(integer|real|boolean|char) | <array-type> | IDENTIFIER
<array-type> ::= KEYWORD(larik) LBRACKET <range> RBRACKET KEYWORD(dari) <type>
```

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

Subprogram:

```
<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> ::= (KEYWORD(variabel))? <identifier-list> COLON <type>
```

Statement:

```
<compound-statement> ::= KEYWORD(mulai) <statement-list> KEYWORD(selesai)
<statement-list> ::= <statement> (SEMICOLON <statement>)*
<statement> ::= <assignment-statement>
              | <if-statement>
              | <while-statement>
              | <for-statement>
              | <repeat-statement>
              | <case-statement>
              | <procedure-call>
              | <compound-statement>
              | <empty-statement>
<assignment-statement> ::= IDENTIFIER 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>
<repeat-statement> ::= KEYWORD(ulangi) <statement-list> KEYWORD(sampai)
<expression>
<case-statement> ::= KEYWORD(kasus) <expression> KEYWORD(dari) (<expression>
COLON <statement> SEMICOLON)+ KEYWORD(selesai)
<procedure-call> ::= IDENTIFIER (LPARENTHESIS <parameter-list> RPARENTHESIS)?
<parameter-list> ::= <expression> (COMMA <expression>)*
```

```
<empty-statement> ::=
```

Ekspresi:

```
<expression> ::= <simple-expression> (RELATIONAL_OPERATOR
<simple-expression>)?
<simple-expression> ::= (ARITHMETIC_OPERATOR(+|-))? <term>
((ARITHMETIC_OPERATOR(+|-)|LOGICAL_OPERATOR(atau)) <term>)*
<term> ::= <factor> ((ARITHMETIC_OPERATOR(*|/|bagi|mod)|LOGICAL_OPERATOR(dan))
<factor>)*
<factor> ::= IDENTIFIER
           | NUMBER
           | STRING_LITERAL
           | LPARENTHESIS <expression> RPARENTHESIS
           | LOGICAL_OPERATOR(tidak) <factor>
           | <function-call>
<function-call> ::= IDENTIFIER LPARENTHESIS (<parameter-list>)? RPARENTHESIS
```

Auxiliary Rules:

```
<identifier-list> ::= IDENTIFIER (COMMA IDENTIFIER)*
<parameter-list> ::= <expression> (COMMA <expression>)*
<block> ::= <declaration-part> <compound-statement>
```

Token Terminals:

```
KEYWORD: program, konstanta, tipe, variabel, prosedur, fungsi, mulai, selesai,
jika, maka, selain-itu, selama, lakukan, untuk, ke, turun-ke, ulangi, sampai,
kasus, dari, integer, real, boolean, char, larik
IDENTIFIER: [a-zA-Z_][a-zA-Z0-9_]*
NUMBER: [0-9]+ | [0-9]+.[0-9]+ (integer atau real literal)
STRING_LITERAL: '...' (karakter dalam single quotes)
ARITHMETIC_OPERATOR: + - * / bagi mod
RELATIONAL_OPERATOR: = <> < <= > >=
LOGICAL_OPERATOR: dan atau tidak
ASSIGN_OPERATOR: :=
PUNCTUATION: ; : , . .. ( ) [ ]
```

Berikut adalah spesifikasi lengkap grammar Pascal-S yang diimplementasikan dalam parser. Grammar ditulis dalam Extended Backus-Naur Form (EBNF) dengan konvensi berikut:

- <non-terminal> untuk simbol non-terminal
- TERMINAL untuk simbol terminal (token)
- ::= untuk definisi produksi
- | untuk alternatif
- ()? untuk optional (0 atau 1 kali)
- ()* untuk repetition (0 atau lebih kali)
- ()+ untuk repetition (1 atau lebih kali)
- () untuk grouping

Berikut detail catatan Grammar:

1. Grammar bersifat context-free dengan struktur hierarkis yang jelas dari program ke statement ke expression.
2. Operator precedence diimplementasikan melalui level hierarki: relational di expression, additive/OR di simple-expression, multiplicative/AND di term.
3. Binary operators bersifat left-associative melalui loop while yang menangani repetition di simple-expression dan term
4. Unary operator "tidak" memiliki precedence tertinggi karena diparsing di level factor.
5. Parentheses dapat digunakan untuk override precedence dengan parsing expression di dalam factor.
6. Empty statement diizinkan untuk menangani trailing semicolon sebelum keyword "selesai".
7. Keyword dalam bahasa Indonesia menggantikan keyword Pascal standar dengan detail padanan kata sebagai berikut:

Kata Bahasa Inggris	Kata Bahasa Indonesia
`and`	`dan`
`array`	`larik`
`begin`	`mulai`
`case`	`kasus`
`const`	`konstanta`
`div`	`bagi`
`do`	`lakukan`
`downto`	`turun-ke`

else	selain-itu
end	selesai
for	untuk
function	fungsi
if	jika
mod	mod
not	tidak
of	dari
or	atau
procedure	prosedur
program	program
record	rekaman
repeat	ulangi
then	maka
to	ke
type	tipe
until	sampai
var	variabel
while	selama

8. Grammar mendukung nested structures seperti nested if, nested loop, dan nested compound statement.
9. Function dan procedure dapat memiliki formal parameter dengan passing by value (default) atau by reference (dengan keyword "variabel" sebelum parameter).
10. Array type menggunakan range dengan operator ".." untuk menspesifikasikan index bounds.

Pendekatan Recursive Descent memiliki karakteristik implementasi yang straightforward. Setiap fungsi parsing bertanggung jawab mengenali satu aturan produksi. Fungsi tersebut membaca token dari input stream, memverifikasi kecocokannya dengan aturan grammar, dan memanggil fungsi parsing lain secara rekursif untuk aturan non-terminal. Jika terjadi ketidaksesuaian, parser melaporkan syntax error dengan informasi posisi dan token yang diharapkan. Parser menghasilkan Parse Tree sebagai representasi

struktural dari program. Setiap node dalam tree mewakili sebuah konstruksi sintaksis, baik terminal (token) maupun non-terminal (simbol grammar). Tree ini memvisualisasikan bagaimana program diurai menurut hierarki grammar. Parse tree berbeda dari Abstract Syntax Tree (AST) karena menyimpan seluruh detail parsing termasuk keyword dan delimiter.

Keunggulan Recursive Descent adalah kemudahan implementasi dan debugging. Alur eksekusi parser mengikuti struktur grammar secara langsung. Setiap fungsi memiliki tanggung jawab yang jelas. Error dapat dilacak dengan mudah karena call stack menunjukkan jalur parsing yang sedang dilakukan. Parser juga dapat memberikan error message yang deskriptif dengan informasi konteks yang lengkap. Implementasi parser ini menggunakan lookahead satu token. Parser melihat token saat ini untuk menentukan jalur parsing yang harus diambil. Dalam beberapa kasus seperti membedakan assignment dan procedure call, parser perlu melihat token berikutnya. Teknik backtracking sederhana digunakan untuk menangani kasus tersebut dengan menyimpan posisi parser sebelum melakukan lookahead.

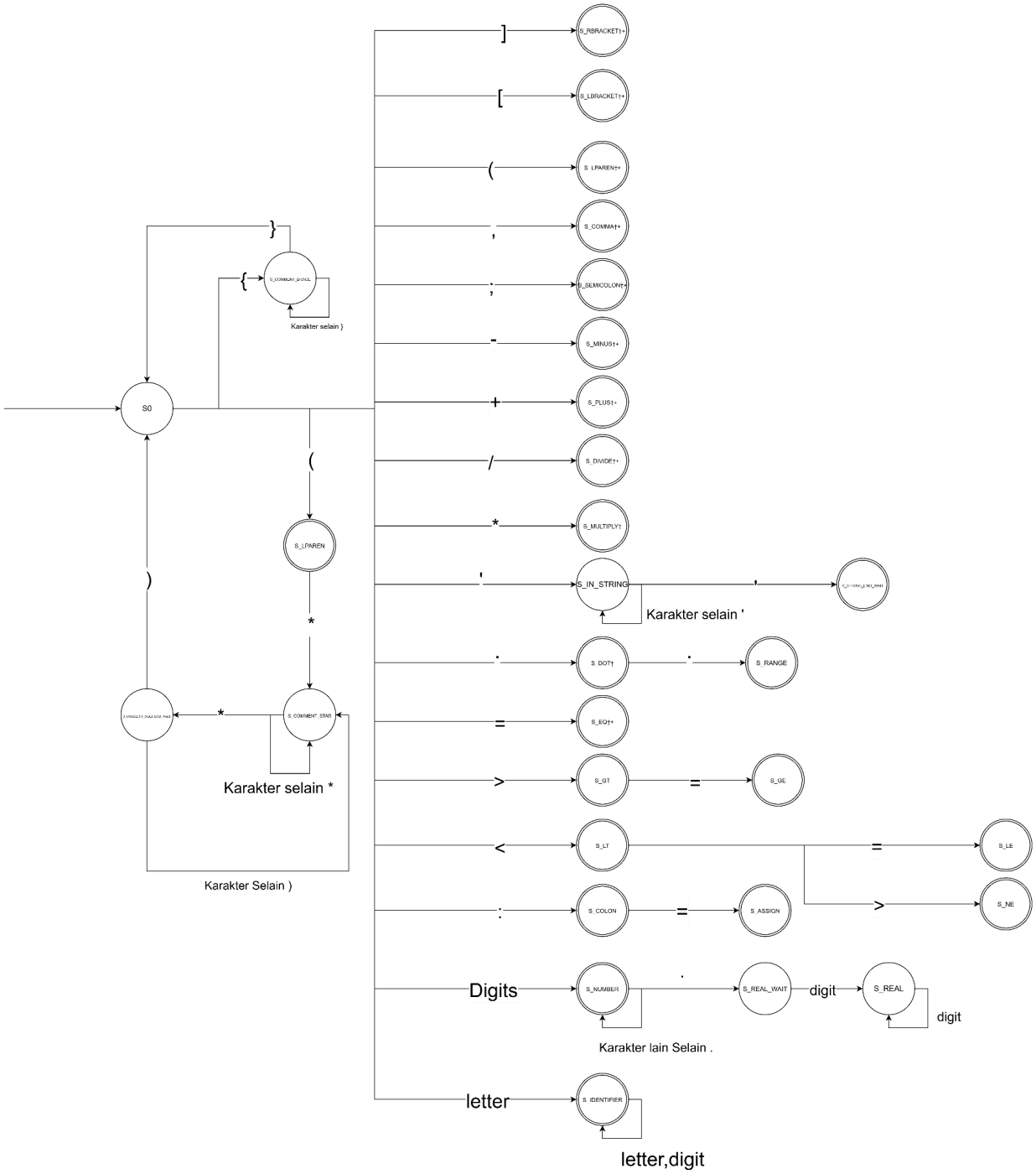


Diagram 1: Visualisasi Alur Transisi DFA

[GAMBAR]
Diagram 2: Visualisasi Alur Syntax Analysis

2.2 Struktur Program

Implementasi Milestone 2 memodifikasi struktur program dari Milestone 1 dengan menambahkan komponen parser. Struktur program terdiri dari empat file utama dengan peran masing-masing sebagai berikut:

1. **compiler.py:** Bertindak sebagai *driver* atau titik masuk program. Skrip ini bertanggung jawab untuk memproses argumen *command-line*, membaca file sumber .pas, dan mengorkestrasi panggilan fungsi dari modul *lexer*.
2. **lexer.py:** Merupakan mesin utama dari *lexer*. Modul ini berisi semua logika inti, termasuk implementasi mesin DFA, fungsi untuk memuat aturan DFA dari file JSON, dan proses *tokenisasi* itu sendiri.
3. **dfa_rules.json:** File konfigurasi yang menyimpan definisi DFA untuk lexer. File ini berisi *start state, final states, transition table,* daftar *keywords, logical operators*, dan *arithmetic keywords* dalam bahasa Indonesia. Pemisahan konfigurasi DFA ke file JSON memudahkan modifikasi aturan lexical tanpa mengubah kode program.
4. **parser.py**: File ini merupakan inti dari Milestone 2. File berisi implementasi Recursive Descent Parser dengan beberapa kelas penting. Kelas Token merepresentasikan satu unit token dengan atribut tipe dan nilai. Kelas NodePohonParsing merepresentasikan node dalam parse tree dengan atribut nama node dan list anak-anaknya. Kelas Parser berisi logika parsing dengan method untuk setiap aturan grammar. Kelas SyntaxError digunakan untuk melaporkan kesalahan sintaksis dengan pesan error yang deskriptif.

Alur kerja program secara keseluruhan berjalan sebagai berikut.

1. Pengguna menjalankan compiler.py dengan argumen berupa path file input dan optional flag.
2. Program memuat aturan DFA dari dfa_rules.json ke dalam memori.
3. Isi dari file kode .pas dibaca sebagai sebuah string tunggal.
4. Fungsi tokenize dari lexer.py dipanggil untuk memproses string kode tersebut karakter per karakter menggunakan mesin DFA yang telah dimuat.
5. Mesin DFA berpindah *state* sesuai dengan karakter input. Ketika sebuah token selesai dikenali, token tersebut ditambahkan ke dalam sebuah daftar.

6. Setelah seluruh kode diproses, daftar *token* yang dihasilkan akan ditulis ke dalam sebuah file output (misalnya, output-1.txt) di direktori yang sama dengan file input. Jika input berupa file .txt, program langsung membaca token dari file tersebut.
7. Setelah token tersedia, program membuat instance Parser dengan memasukkan list token.
8. Parser dijalankan dengan memanggil method parse yang mengembalikan root node dari parse tree.
9. Parse tree dicetak ke console menggunakan method cetak yang menghasilkan visualisasi tree dengan indentasi dan simbol cabang.
10. Jika terjadi syntax error, program menangkap exception dan menampilkan pesan error dengan informasi posisi dan token yang diharapkan.

Modifikasi dari Milestone 1 meliputi penambahan import Parser dan SyntaxError di compiler.py, penambahan logika untuk menjalankan syntax analysis setelah lexical analysis, penambahan exception handling untuk SyntaxError, dan penambahan mode input dari file .txt untuk memudahkan testing parser secara terpisah. Struktur ini memisahkan concern dengan jelas antara lexical analysis dan syntax analysis sesuai prinsip modular design.

2.3 Implementasi Detail

Implementasi parser terdiri dari beberapa komponen kunci yang bekerja sama untuk memverifikasi sintaksis program Pascal-S.

Potongan Kode 1: Struktur Kelas Parser dan Token Matching

```
class Parser:
    def __init__(self, tokens):
        self.tokens = [Token(t[0], t[1]) for t in tokens]
        self.posisi = 0
          self.token_sekarang = self.tokens[0] if len(self.tokens) > 0
else None

    def maju(self):
        self.posisi += 1
        if self.posisi < len(self.tokens):
            self.token_sekarang = self.tokens[self.posisi]
        else:
            self.token_sekarang = None
```

```
    def cocokkan(self, tipe_token, nilai_token=None):
        if self.token_sekarang is None:
            raise SyntaxError(f"error: token tidak ditemukan, diharapkan
{tipe_token}")

        if self.token_sekarang.tipe != tipe_token:
                raise SyntaxError(f"error: token tidak sesuai, ditemukan
{self.token_sekarang.tipe}({self.token_sekarang.nilai}),      diharapkan
{tipe_token}")

           if nilai_token is not None and self.token_sekarang.nilai !=
nilai_token:
                  raise SyntaxError(f"error: nilai token tidak sesuai,
ditemukan {self.token_sekarang.nilai}, diharapkan {nilai_token}")

                                                                node       =
NodePohonParsing(f"{self.token_sekarang.tipe}({self.token_sekarang.nilai
})")
        self.maju()
        return node
```

Kelas Parser mengelola state parsing dengan menyimpan list token, posisi saat ini, dan token yang sedang dibaca. Method maju memindahkan posisi ke token berikutnya. Method cocokkan melakukan verifikasi token dengan memeriksa tipe dan optional nilai token. Jika token tidak sesuai, method melempar SyntaxError dengan pesan deskriptif. Jika sesuai, method membuat node untuk parse tree dan memajukan posisi.

Potongan Kode 2: Parsing Statement dengan Dispatching

```
def parse_statement(self):
    if not self.token_sekarang:
        return NodePohonParsing("<empty-statement>")

    if self.token_sekarang.tipe == "KEYWORD":
        if self.token_sekarang.nilai == "jika":
            return self.parse_if_statement()
        elif self.token_sekarang.nilai == "selama":
```

```
            return self.parse_while_statement()
        elif self.token_sekarang.nilai == "untuk":
            return self.parse_for_statement()
        elif self.token_sekarang.nilai == "mulai":
            return self.parse_compound_statement()
        elif self.token_sekarang.nilai == "ulangi":
            return self.parse_repeat_statement()
        elif self.token_sekarang.nilai == "kasus":
            return self.parse_case_statement()

    if self.token_sekarang.tipe == "IDENTIFIER":
        if self.posisi + 1 < len(self.tokens):
            token_berikut = self.tokens[self.posisi + 1]
            if token_berikut.tipe == "ASSIGN_OPERATOR":
                return self.parse_assignment_statement()
                        elif token_berikut.tipe == "LPARENTHESIS" or
token_berikut.tipe == "SEMICOLON":
                return self.parse_procedure_call()
        else:
            return self.parse_procedure_call()

    return NodePohonParsing("<empty-statement>")
```

Method parse_statement bertindak sebagai dispatcher yang menentukan jenis statement berdasarkan token saat ini. Untuk keyword, method langsung memetakan ke parsing function yang sesuai. Untuk identifier, method melakukan lookahead satu token untuk membedakan assignment statement dan procedure call. Pendekatan ini mengimplementasikan LL(1) parsing dengan lookahead terbatas.

Potongan Kode 3: Parsing Expression dengan Precedence

```
def parse_expression(self):
    node = NodePohonParsing("<expression>")
    node.tambah_anak(self.parse_simple_expression())
            if self.token_sekarang and self.token_sekarang.tipe ==
"RELATIONAL_OPERATOR":
        node.tambah_anak(self.cocokkan("RELATIONAL_OPERATOR"))
```

```
        node.tambah_anak(self.parse_simple_expression())
    return node

def parse_simple_expression(self):
    node = NodePohonParsing("<simple-expression>")

          if  self.token_sekarang  and  self.token_sekarang.tipe  ==
"ARITHMETIC_OPERATOR" and self.token_sekarang.nilai in ["+", "-"]:
        node.tambah_anak(self.cocokkan("ARITHMETIC_OPERATOR"))

    node.tambah_anak(self.parse_term())
    while self.token_sekarang and (
                (self.token_sekarang.tipe  ==  "ARITHMETIC_OPERATOR"  and
self.token_sekarang.nilai in ["+", "-"]) or
                  (self.token_sekarang.tipe  ==  "LOGICAL_OPERATOR"  and
self.token_sekarang.nilai == "atau")
    ):
        if self.token_sekarang.tipe == "ARITHMETIC_OPERATOR":
            node.tambah_anak(self.cocokkan("ARITHMETIC_OPERATOR"))
        else:
            node.tambah_anak(self.cocokkan("LOGICAL_OPERATOR"))
        node.tambah_anak(self.parse_term())
    return node

def parse_term(self):
    node = NodePohonParsing("<term>")
    node.tambah_anak(self.parse_factor())
    while self.token_sekarang and (
                (self.token_sekarang.tipe  ==  "ARITHMETIC_OPERATOR"  and
self.token_sekarang.nilai in ["*", "/", "bagi", "mod"]) or
                  (self.token_sekarang.tipe  ==  "LOGICAL_OPERATOR"  and
self.token_sekarang.nilai == "dan")
    ):
        if self.token_sekarang.tipe == "ARITHMETIC_OPERATOR":
            node.tambah_anak(self.cocokkan("ARITHMETIC_OPERATOR"))
        else:
            node.tambah_anak(self.cocokkan("LOGICAL_OPERATOR"))
```

```
        node.tambah_anak(self.parse_factor())
    return node
```

Parsing expression menggunakan hierarki tiga level untuk mengimplementasikan operator precedence. Level tertinggi adalah expression yang menangani relational operator. Level tengah adalah simple_expression yang menangani additive operator dan logical OR. Level terendah adalah term yang menangani multiplicative operator dan logical AND. Struktur ini memastikan operator dengan precedence lebih tinggi dievaluasi terlebih dahulu. Loop while pada simple_expression dan term mengimplementasikan left-associativity untuk operator biner.

Potongan Kode 5: Parse Tree Construction dan Visualization

```
class NodePohonParsing:
    def __init__(self, nama, anak=None):
        self.nama = nama
        self.anak = anak if anak is not None else []

    def tambah_anak(self, node):
        self.anak.append(node)

    def cetak(self, level=0, prefix=""):
        hasil = ""
        if level == 0:
            hasil += f"{prefix}{self.nama}\n"
        else:
            hasil += f"{prefix}{self.nama}\n"

        jumlah_anak = len(self.anak)
        for i, anak in enumerate(self.anak):
            adalah_terakhir = (i == jumlah_anak - 1)
            if level == 0:
                prefix_baru = ""
            else:
                prefix_baru = prefix

            if adalah_terakhir:
                hasil += anak.cetak(level + 1, prefix_baru + "└── ")
```

```
            else:
                hasil += anak.cetak(level + 1, prefix_baru + "├── ")

        return hasil
```

Kelas NodePohonParsing merepresentasikan node dalam parse tree. Setiap node menyimpan nama (label) dan list anak-anak. Method tambah_anak menambahkan child node. Method cetak menghasilkan visualisasi tree dengan format ASCII art menggunakan simbol box-drawing. Tree dicetak secara rekursif dengan indentasi dan simbol cabang yang menunjukkan struktur hierarkis. Simbol "├──" digunakan untuk child yang bukan terakhir, dan "└──" untuk child terakhir.

Potongan Kode 6: Error Handling dan Reporting

```
class SyntaxError(Exception):
    def __init__(self, pesan):
        self.pesan = pesan
        super().__init__(self.pesan)

def parse(self):
    try:
        pohon = self.parse_program()
        if self.token_sekarang is not None:
                raise SyntaxError(f"error: masih ada token yang tersisa
setelah parsing selesai")
        return pohon
    except SyntaxError as e:
        raise e
```

Kelas SyntaxError merupakan custom exception untuk melaporkan kesalahan sintaksis. Method parse di kelas Parser merupakan entry point yang membungkus proses parsing dengan exception handling. Setelah parsing selesai, method memverifikasi bahwa semua token telah dikonsumsi. Jika masih ada token tersisa, berarti ada konten yang tidak dikenali setelah end of program. Error message mencakup informasi posisi token dan token yang diharapkan untuk memudahkan debugging.

Bab III

Pengujian

3.1 Kasus Pengujian Unik

3.1.1 Uji Program Sederhana

test1_simple.pas

```
program Hello;

variabel
  a, b: integer;

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

Test case ini menguji struktur program paling dasar dengan deklarasi variabel sederhana, assignment statement, dan ekspresi aritmatika. Program mendeklarasikan dua variabel integer, melakukan assignment dengan literal dan ekspresi, kemudian memanggil procedure writeln. Parser harus mengenali program header, variable declaration dengan identifier list, compound statement berisi assignment, dan procedure call dengan multiple parameters.

3.1.2 Uji If Statement

test2_if_statement.pas

```
program IfTest;

variabel
  x, y: integer;

mulai
  x := 10;
  jika x > 5 maka
```

```
    y := 1
  selain-itu
    y := 0;
  writeln(y);
selesai.
```

Test case ini menguji if-else statement dengan relational expression. Program melakukan conditional assignment berdasarkan perbandingan nilai variabel. Parser harus mengenali struktur if-statement lengkap dengan keyword "jika", "maka", dan "selain-itu". Ekspresi kondisi menggunakan relational operator. Statement di branch if dan else keduanya berupa assignment.

3.1.3 Uji Perulangan For Loop

test3_loop.pas

```
program LoopTest;

variabel
  i, sum: integer;

mulai
  sum := 0;
  untuk i := 1 ke 10 lakukan
    sum := sum + i;
  writeln('Sum = ', sum);
selesai.
```

Test case ini menguji for-loop statement dengan iterasi ascending. Program menghitung sum dari 1 sampai 10 menggunakan loop. Parser harus mengenali struktur for-statement dengan keyword "untuk", "ke", dan "lakukan". Parsing harus menangani loop variable, initial value, final value, dan body statement. Loop body berupa single assignment statement.

3.1.4 Uji Kasus Prosedur dengan Parameter

test4_procedure.pas

```
program ProcedureTest;
```

```
variabel
  x: integer;

prosedur print(nilai: integer);
mulai
  writeln('Nilai: ', nilai);
selesai;

mulai
  x := 42;
  print(x);
selesai.
```

Test case ini menguji deklarasi dan pemanggilan prosedur dengan parameter formal. Parser harus mengenali procedure declaration dengan formal parameter list. Declaration part di level program mengandung variable declaration dan subprogram declaration. Body prosedur berisi compound statement. Main program memanggil prosedur dengan actual parameter. Parser harus membedakan procedure call dari assignment statement.

3.1.5 Uji Kasus Ekspresis Kompleks

test10_complex_expression.pas

```
program ComplexExpressionTest;

variabel
  a, b, c, d: integer;
  x, y: real;
  valid: boolean;

mulai
  a := 10;
  b := 20;
  c := 3;
  d := 5;
```

```
  x := (a + b) * c bagi (d - 2) + a mod c;
  y := a * b + c - d bagi 2;

  valid := (a > b) atau ((c < d) dan (x <> y));

  jika (a + b > c * d) dan tidak (x = y) maka
    writeln('Kondisi kompleks terpenuhi')
  selain-itu
    writeln('Kondisi tidak terpenuhi');
selesai.
```

Test case ini menguji parsing ekspresi dengan multiple operators dan precedence levels. Ekspresi mengandung arithmetic operators (tambah, kali, bagi, mod), relational operators (lebih besar, lebih kecil, tidak sama), logical operators (dan, atau, tidak), dan parentheses untuk grouping. Parser harus membangun parse tree yang mencerminkan precedence dan associativity yang benar. Ekspresi nested dan kombinasi operator boolean menguji robustness parser.

3.1.6 Uji Kasus Error Missing Semicolon

test_error1_missing_semicolon.pas

```
program ErrorTest1;

variabel
  x, y: integer;

mulai
  x := 10
  y := 20;
  writeln(x + y);
selesai.
```

Test case ini menguji error detection untuk missing semicolon. Statement pertama di compound statement tidak diakhiri semicolon. Parser harus mendeteksi error ini dan melaporkan posisi serta token yang diharapkan. Error message harus informatif dengan menyebutkan bahwa SEMICOLON diharapkan setelah assignment statement.

3.1.7 Uji Kasus Error Missing Keyword

test_error2_missing_maka.pas

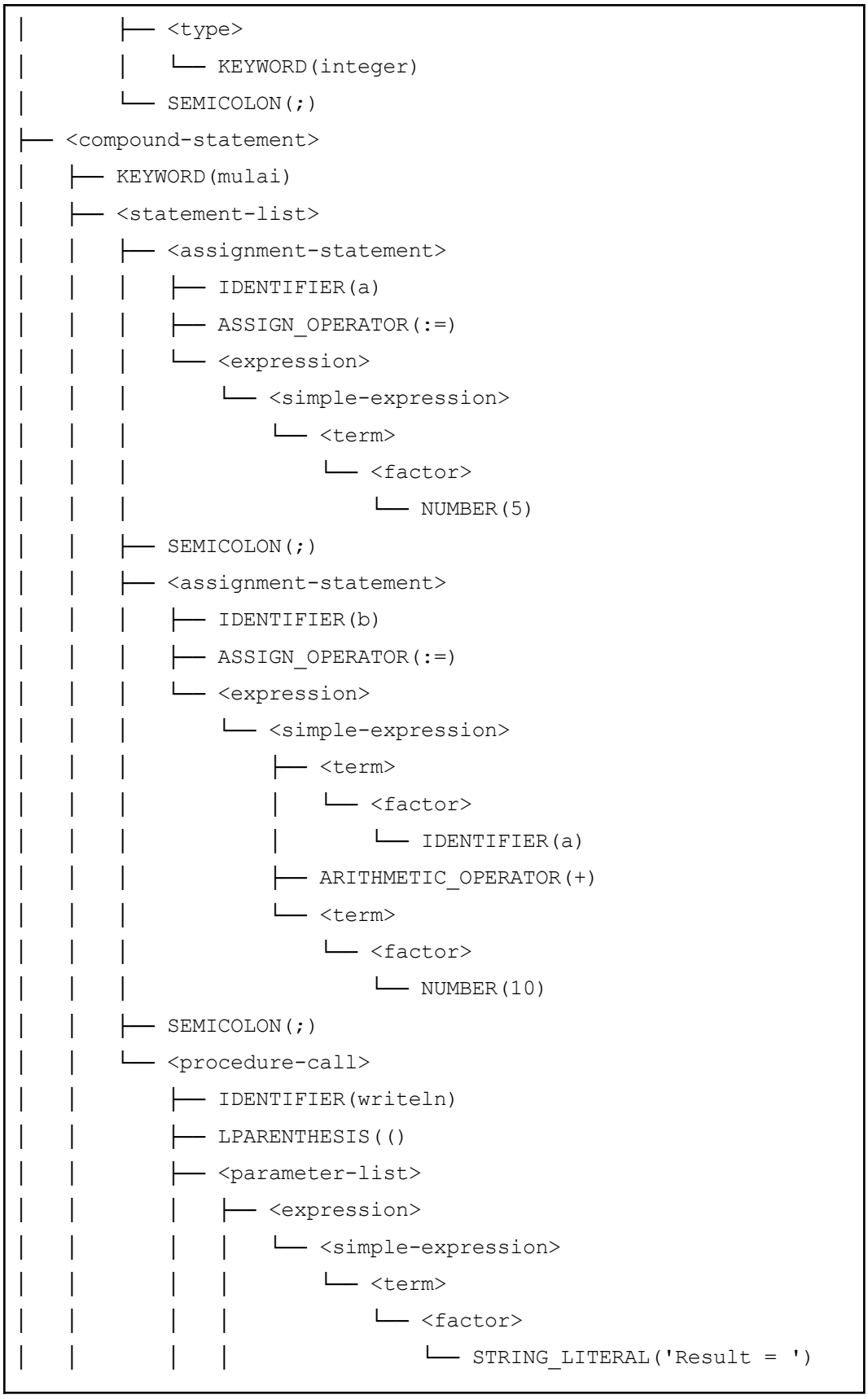
```
program ErrorTest2;

variabel
  x: integer;

mulai
  x := 10;
  jika x > 5
    writeln(x);
selesai.
```

Test case ini menguji error detection untuk missing keyword "maka" dalam if-statement. Parser harus mendeteksi bahwa setelah expression dalam if-statement, keyword "maka" wajib ada. Error message harus menyebutkan token yang ditemukan dan token yang diharapkan dengan informasi posisi yang akurat.

3.2 Hasil dan Pembahasan Pengujian

3.2.1 Hasil Uji Program Sederhana

```
Parsing berhasil!

Parse Tree
<program>
├── <program-header>
│   ├── KEYWORD(program)
│   ├── IDENTIFIER(Hello)
│   └── SEMICOLON(;)
├── <declaration-part>
│   └── <var-declaration>
│       ├── KEYWORD(variabel)
│       ├── <identifier-list>
│       │   ├── IDENTIFIER(a)
│       │   ├── COMMA(,)
│       │   └── IDENTIFIER(b)
│       ├── COLON(:)
```

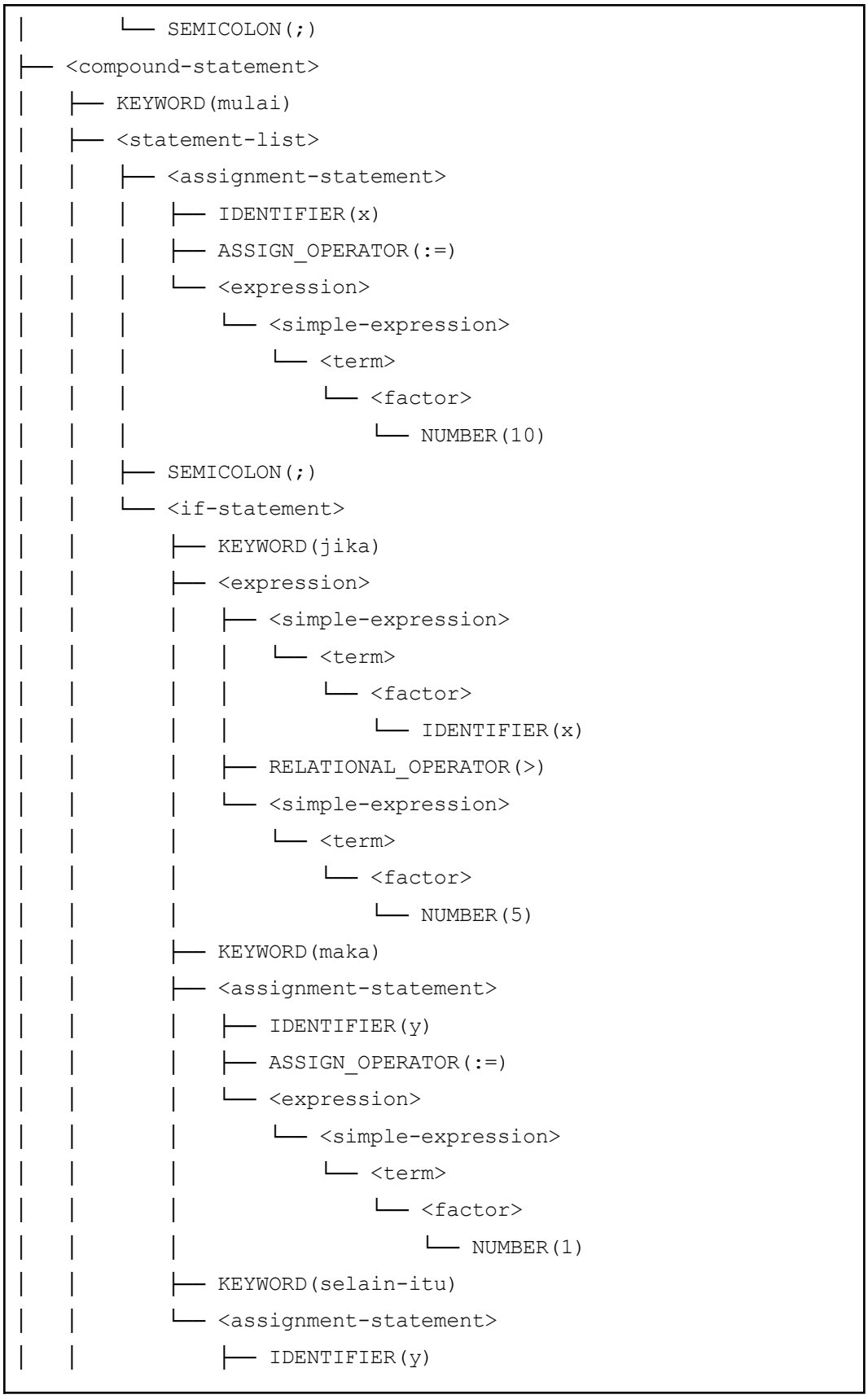

├── <type>
│ └── KEYWORD(integer)
└── SEMICOLON(;)
├── <compound-statement>
├── KEYWORD(mulai)
├── <statement-list>
├── <assignment-statement>
├── IDENTIFIER(a)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(5)
├── SEMICOLON(;)
├── <assignment-statement>
├── IDENTIFIER(b)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
├── <term>
│ └── <factor>
│ └── IDENTIFIER(a)
├── ARITHMETIC_OPERATOR(+)
└── <term>
└── <factor>
└── NUMBER(10)
├── SEMICOLON(;)
└── <procedure-call>
├── IDENTIFIER(writeln)
├── LPARENTHESIS(()
├── <parameter-list>
├── <expression>
│ └── <simple-expression>
│ └── <term>
│ └── <factor>
│ └── STRING_LITERAL('Result = ')

```
│   │       │   ├── COMMA(,)
│   │       │   └── <expression>
│   │       │       └── <simple-expression>
│   │       │           └── <term>
│   │       │               └── <factor>
│   │       │                   └── IDENTIFIER(b)
│   │       └── RPARENTHESIS())
│   └── KEYWORD(selesai)
└── DOT(.)
```

Pembahasan: Parser berhasil mengenali struktur program sederhana dengan benar. Parse tree menunjukkan hierarki lengkap dari program header hingga compound statement. Variable declaration dengan identifier list (a, b) diparsing sebagai satu group dengan tipe integer. Assignment statement pertama mengandung literal number sebagai factor dalam expression. Assignment kedua mengandung binary expression dengan operator plus. Procedure call writeln diparsing dengan parameter list berisi dua expression: string literal dan identifier. Seluruh struktur sesuai dengan grammar yang didefinisikan.

3.2.2 Hasil Uji If Statement

```
Parsing berhasil!

Parse Tree
<program>
├── <program-header>
│   ├── KEYWORD(program)
│   ├── IDENTIFIER(IfTest)
│   └── SEMICOLON(;)
├── <declaration-part>
│   └── <var-declaration>
│       ├── KEYWORD(variabel)
│       ├── <identifier-list>
│       │   ├── IDENTIFIER(x)
│       │   ├── COMMA(,)
│       │   └── IDENTIFIER(y)
│       ├── COLON(:)
│       ├── <type>
│       │   └── KEYWORD(integer)
```

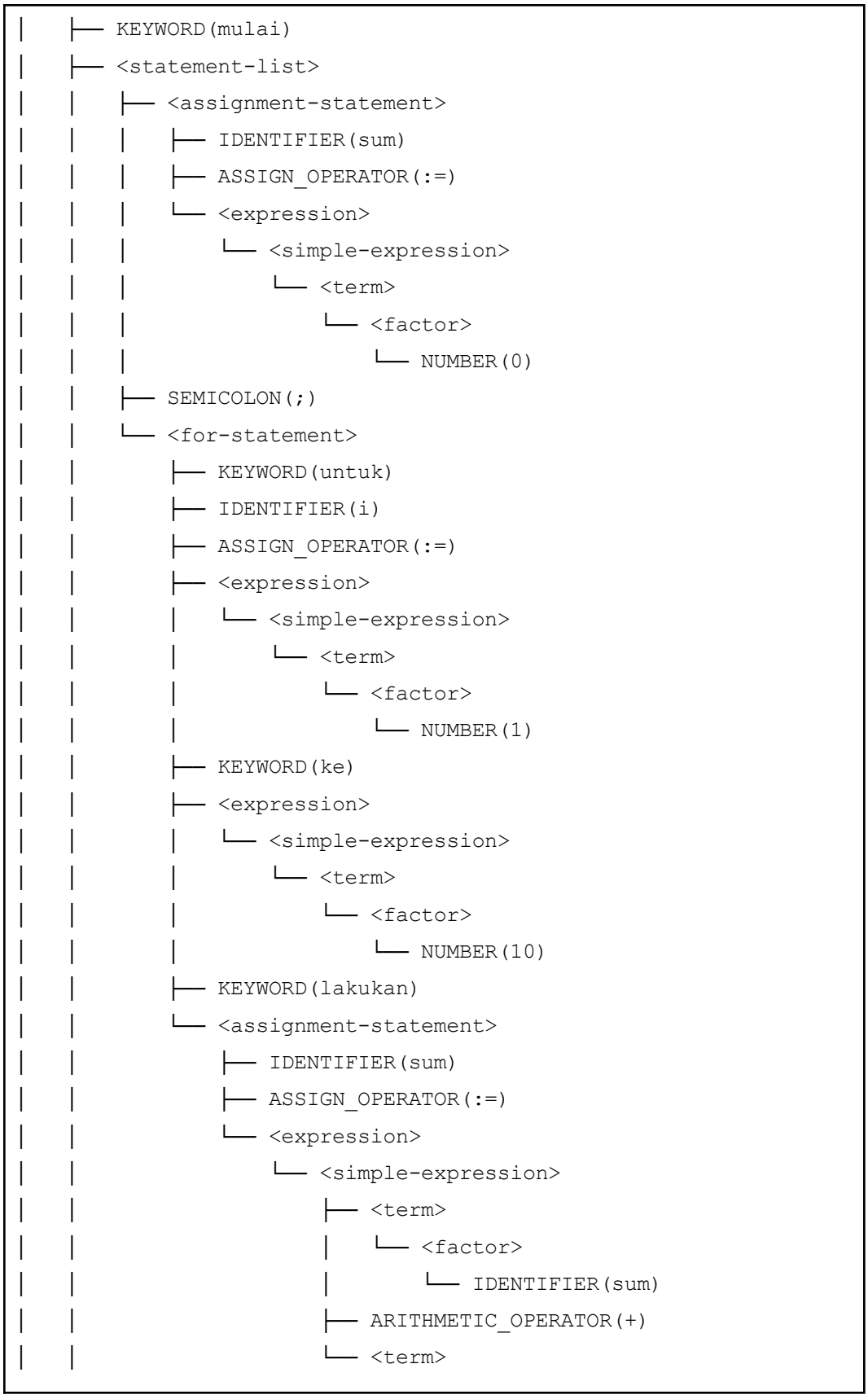

└── SEMICOLON(;)
├── <compound-statement>
├── KEYWORD(mulai)
├── <statement-list>
├── <assignment-statement>
├── IDENTIFIER(x)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(10)
├── SEMICOLON(;)
└── <if-statement>
├── KEYWORD(jika)
├── <expression>
├── <simple-expression>
└── <term>
└── <factor>
└── IDENTIFIER(x)
├── RELATIONAL_OPERATOR(>)
└── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(5)
├── KEYWORD(maka)
├── <assignment-statement>
├── IDENTIFIER(y)
├── ASSIGN_OPERATOR(:=)
└── <expression>
└── <simple-expression>
└── <term>
└── <factor>
└── NUMBER(1)
├── KEYWORD(selain-itu)
└── <assignment-statement>
├── IDENTIFIER(y)

```
│   │           ├── ASSIGN_OPERATOR(:=)
│   │           └── <expression>
│   │               └── <simple-expression>
│   │                   └── <term>
│   │                       └── <factor>
│   │                           └── NUMBER(0)
│   └── KEYWORD(selesai)
└── DOT(.)
```

Pembahasan: Parser berhasil menangani if-else statement dengan benar. Kondisi dalam if diparsing sebagai expression dengan relational operator. Expression terdiri dari dua simple-expression yang dihubungkan oleh operator relasional ">". Branch then dan else masing-masing berisi assignment statement. Parser mengenali keyword "selain-itu" sebagai penanda else branch. Struktur if-statement lengkap dengan semua komponennya terepresentasi dalam parse tree.

3.2.3 Hasil Uji For Loop

```
Parsing berhasil!

Parse Tree
<program>
├── <program-header>
│   ├── KEYWORD(program)
│   ├── IDENTIFIER(LoopTest)
│   └── SEMICOLON(;)
├── <declaration-part>
│   └── <var-declaration>
│       ├── KEYWORD(variabel)
│       ├── <identifier-list>
│       │   ├── IDENTIFIER(i)
│       │   ├── COMMA(,)
│       │   └── IDENTIFIER(sum)
│       ├── COLON(:)
│       ├── <type>
│       │   └── KEYWORD(integer)
│       └── SEMICOLON(;)
├── <compound-statement>
```

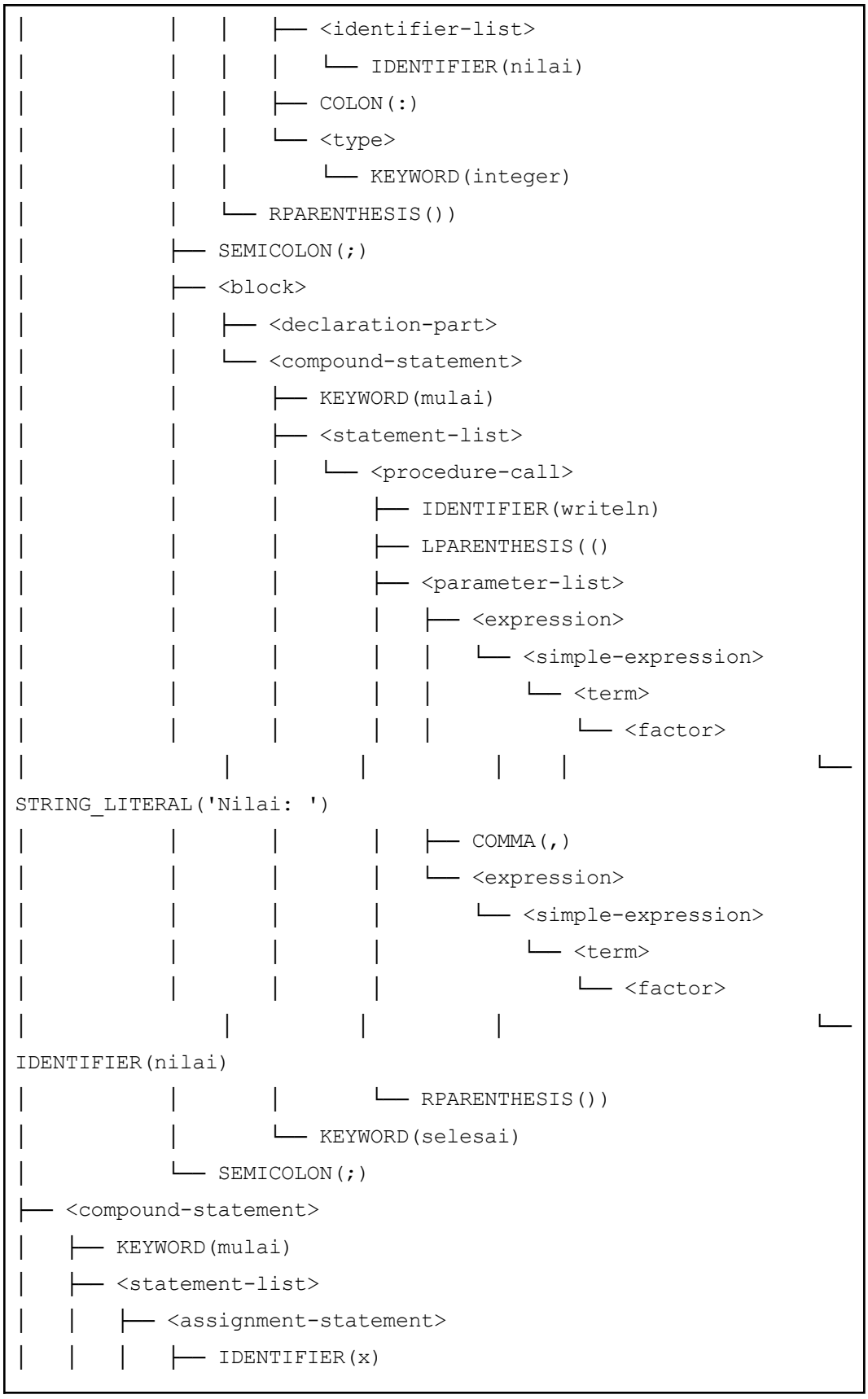

├── KEYWORD(mulai)
├── <statement-list>
│ ├── <assignment-statement>
│ │ ├── IDENTIFIER(sum)
│ │ ├── ASSIGN_OPERATOR(:=)
│ │ └── <expression>
│ │ └── <simple-expression>
│ │ └── <term>
│ │ └── <factor>
│ │ └── NUMBER(0)
│ ├── SEMICOLON(;)
│ └── <for-statement>
│ ├── KEYWORD(untuk)
│ ├── IDENTIFIER(i)
│ ├── ASSIGN_OPERATOR(:=)
│ ├── <expression>
│ │ └── <simple-expression>
│ │ └── <term>
│ │ └── <factor>
│ │ └── NUMBER(1)
│ ├── KEYWORD(ke)
│ ├── <expression>
│ │ └── <simple-expression>
│ │ └── <term>
│ │ └── <factor>
│ │ └── NUMBER(10)
│ ├── KEYWORD(lakukan)
│ └── <assignment-statement>
│ ├── IDENTIFIER(sum)
│ ├── ASSIGN_OPERATOR(:=)
│ └── <expression>
│ └── <simple-expression>
│ ├── <term>
│ │ └── <factor>
│ │ └── IDENTIFIER(sum)
│ ├── ARITHMETIC_OPERATOR(+)
│ └── <term>

```
│   │                       └── <factor>
│   │                           └── IDENTIFIER(i)
│   └── KEYWORD(selesai)
└── DOT(.)
```

Pembahasan: Parser berhasil menangani for-statement dengan komponen lengkap. Loop variable identifier, initial value expression, keyword "ke", final value expression, dan keyword "lakukan" semuanya diparsing dengan benar. Body loop berupa assignment statement yang mengakumulasi nilai sum. Parser mengenali bahwa after keyword "lakukan" harus diikuti statement. Expression untuk initial dan final value diparsing sesuai hierarki expression grammar.

3.2.4 Hasil Uji Prosedur

```
Parsing berhasil!


Parse Tree
<program>
├── <program-header>
│   ├── KEYWORD(program)
│   ├── IDENTIFIER(ProcedureTest)
│   └── SEMICOLON(;)
├── <declaration-part>
│   ├── <var-declaration>
│   │   ├── KEYWORD(variabel)
│   │   ├── <identifier-list>
│   │   │   └── IDENTIFIER(x)
│   │   ├── COLON(:)
│   │   ├── <type>
│   │   │   └── KEYWORD(integer)
│   │   └── SEMICOLON(;)
│   └── <subprogram-declaration>
│       └── <procedure-declaration>
│           ├── KEYWORD(prosedur)
│           ├── IDENTIFIER(print)
│           ├── <formal-parameter-list>
│           │   ├── LPARENTHESIS(()
│           │   ├── <parameter-group>
```


│ │ ├── <identifier-list>
│ │ │ └── IDENTIFIER(nilai)
│ │ ├── COLON(:)
│ │ └── <type>
│ │ └── KEYWORD(integer)
│ └── RPARENTHESIS())
├── SEMICOLON(;)
├── <block>
│ ├── <declaration-part>
│ └── <compound-statement>
│ ├── KEYWORD(mulai)
│ ├── <statement-list>
│ │ └── <procedure-call>
│ │ ├── IDENTIFIER(writeln)
│ │ ├── LPARENTHESIS(()
│ │ ├── <parameter-list>
│ │ │ ├── <expression>
│ │ │ │ └── <simple-expression>
│ │ │ │ └── <term>
│ │ │ │ └── <factor>
STRING_LITERAL('Nilai: ')
│ │ │ ├── COMMA(,)
│ │ │ └── <expression>
│ │ │ └── <simple-expression>
│ │ │ └── <term>
│ │ │ └── <factor>
IDENTIFIER(nilai)
│ │ └── RPARENTHESIS())
│ └── KEYWORD(selesai)
└── SEMICOLON(;)
├── <compound-statement>
│ ├── KEYWORD(mulai)
│ ├── <statement-list>
│ │ ├── <assignment-statement>
│ │ │ ├── IDENTIFIER(x)

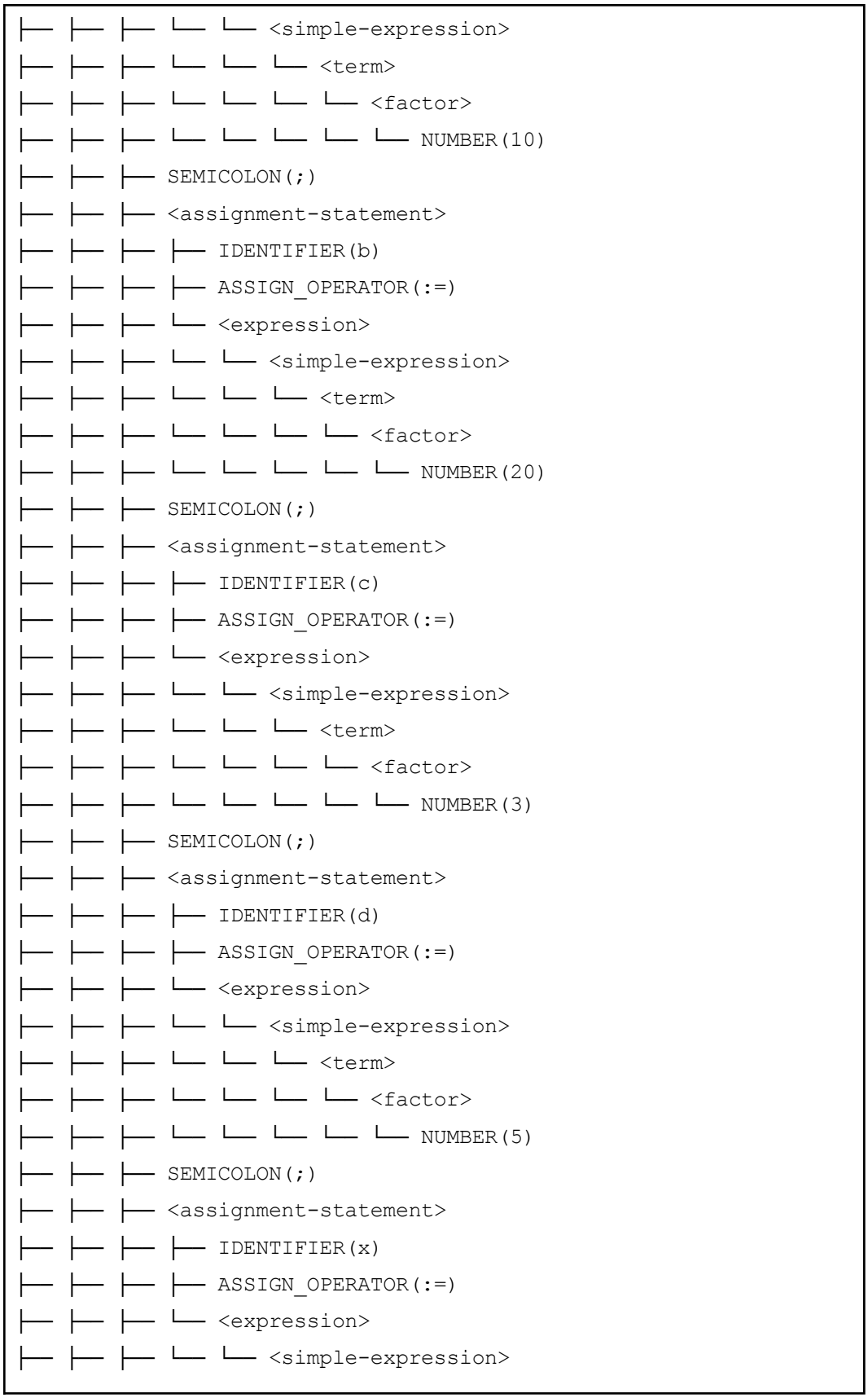
```
│   │   │   ├── ASSIGN_OPERATOR(:=)
│   │   │   └── <expression>
│   │   │       └── <simple-expression>
│   │   │           └── <term>
│   │   │               └── <factor>
│   │   │                   └── NUMBER(42)
│   │   ├── SEMICOLON(;)
│   │   └── <procedure-call>
│   │       ├── IDENTIFIER(print)
│   │       ├── LPARENTHESIS(()
│   │       ├── <parameter-list>
│   │       │   └── <expression>
│   │       │       └── <simple-expression>
│   │       │           └── <term>
│   │       │               └── <factor>
│   │       │                   └── IDENTIFIER(x)
│   │       └── RPARENTHESIS())
│   └── KEYWORD(selesai)
└── DOT(.)
```

Pembahasan: Parser berhasil menangani procedure declaration dengan formal parameter. Declaration-part di level program berisi var-declaration dan subprogram-declaration. Procedure declaration mencakup procedure name, formal parameter list dengan parameter group, dan block. Block prosedur mengandung declaration-part kosong dan compound-statement. Main program memanggil prosedur dengan actual parameter. Parser membedakan procedure call dari assignment karena ada parentheses setelah identifier.

3.2.5 Hasil Uji Ekspresi Kompleks

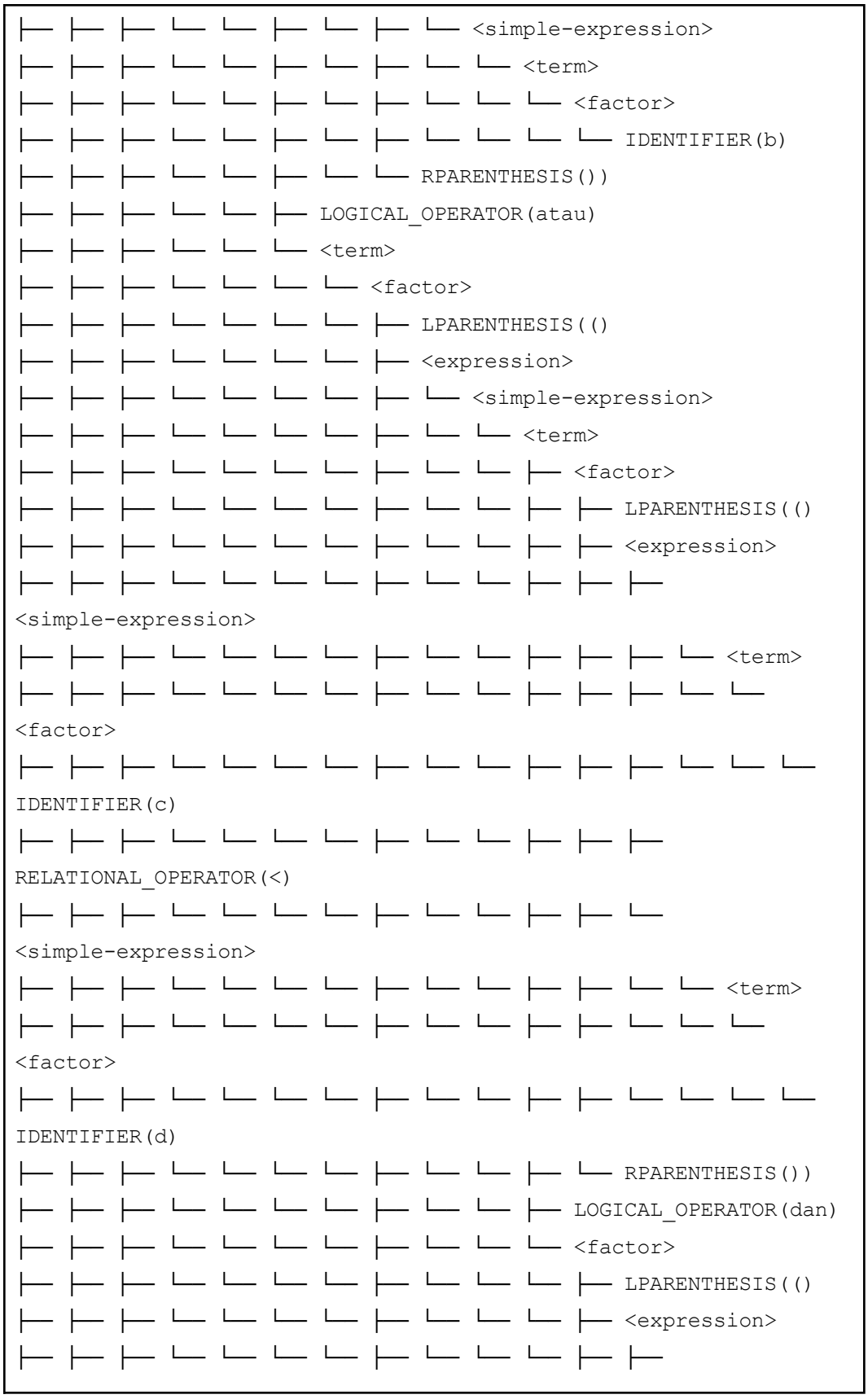
```
Parsing berhasil!

Parse Tree
<program>
├── <program-header>
├── ├── KEYWORD(program)
├── ├── IDENTIFIER(ComplexExpressionTest)
```

```
├── └── SEMICOLON(;)
├── <declaration-part>
├── └── <var-declaration>
├── └── ├── KEYWORD(variabel)
├── └── ├── <identifier-list>
├── └── ├── ├── IDENTIFIER(a)
├── └── ├── ├── COMMA(,)
├── └── ├── ├── IDENTIFIER(b)
├── └── ├── ├── COMMA(,)
├── └── ├── ├── IDENTIFIER(c)
├── └── ├── ├── COMMA(,)
├── └── ├── └── IDENTIFIER(d)
├── └── ├── COLON(:)
├── └── ├── <type>
├── └── ├── └── KEYWORD(integer)
├── └── ├── SEMICOLON(;)
├── └── ├── <identifier-list>
├── └── ├── ├── IDENTIFIER(x)
├── └── ├── ├── COMMA(,)
├── └── ├── └── IDENTIFIER(y)
├── └── ├── COLON(:)
├── └── ├── <type>
├── └── ├── └── KEYWORD(real)
├── └── ├── SEMICOLON(;)
├── └── ├── <identifier-list>
├── └── ├── └── IDENTIFIER(valid)
├── └── ├── COLON(:)
├── └── ├── <type>
├── └── ├── └── KEYWORD(boolean)
├── └── └── SEMICOLON(;)
├── <compound-statement>
├── ├── KEYWORD(mulai)
├── ├── <statement-list>
├── ├── ├── <assignment-statement>
├── ├── ├── ├── IDENTIFIER(a)
├── ├── ├── ├── ASSIGN_OPERATOR(:=)
├── ├── ├── └── <expression>
```


├── ├── ├── └── └── <simple-expression>
├── ├── ├── └── └── └── <term>
├── ├── ├── └── └── └── └── <factor>
├── ├── ├── └── └── └── └── └── NUMBER(10)
├── ├── ├── SEMICOLON(;)
├── ├── ├── <assignment-statement>
├── ├── ├── ├── IDENTIFIER(b)
├── ├── ├── ├── ASSIGN_OPERATOR(:=)
├── ├── ├── └── <expression>
├── ├── ├── └── └── <simple-expression>
├── ├── ├── └── └── └── <term>
├── ├── ├── └── └── └── └── <factor>
├── ├── ├── └── └── └── └── └── NUMBER(20)
├── ├── ├── SEMICOLON(;)
├── ├── ├── <assignment-statement>
├── ├── ├── ├── IDENTIFIER(c)
├── ├── ├── ├── ASSIGN_OPERATOR(:=)
├── ├── ├── └── <expression>
├── ├── ├── └── └── <simple-expression>
├── ├── ├── └── └── └── <term>
├── ├── ├── └── └── └── └── <factor>
├── ├── ├── └── └── └── └── └── NUMBER(3)
├── ├── ├── SEMICOLON(;)
├── ├── ├── <assignment-statement>
├── ├── ├── ├── IDENTIFIER(d)
├── ├── ├── ├── ASSIGN_OPERATOR(:=)
├── ├── ├── └── <expression>
├── ├── ├── └── └── <simple-expression>
├── ├── ├── └── └── └── <term>
├── ├── ├── └── └── └── └── <factor>
├── ├── ├── └── └── └── └── └── NUMBER(5)
├── ├── ├── SEMICOLON(;)
├── ├── ├── <assignment-statement>
├── ├── ├── ├── IDENTIFIER(x)
├── ├── ├── ├── ASSIGN_OPERATOR(:=)
├── ├── ├── └── <expression>
├── ├── ├── └── └── <simple-expression>


├── ├── ├── └── └── ├── <term>
├── ├── ├── └── └── ├── ├── <factor>
├── ├── ├── └── └── ├── ├── ├── LPARENTHESIS(()
├── ├── ├── └── └── ├── ├── ├── <expression>
├── ├── ├── └── └── ├── ├── ├── └── <simple-expression>
├── ├── ├── └── └── ├── ├── ├── └── ├── <term>
├── ├── ├── └── └── ├── ├── ├── └── ├── └── <factor>
├── ├── ├── └── └── ├── ├── ├── └── ├── └── └── IDENTIFIER(a)
├── ├── ├── └── └── ├── ├── ├── └── ├── ARITHMETIC_OPERATOR(+)
├── ├── ├── └── └── ├── ├── ├── └── └── <term>
├── ├── ├── └── └── ├── ├── ├── └── └── └── <factor>
├── ├── ├── └── └── ├── ├── ├── └── └── └── └── IDENTIFIER(b)
├── ├── ├── └── └── ├── ├── └── RPARENTHESIS())
├── ├── ├── └── └── ├── ├── ARITHMETIC_OPERATOR(*)
├── ├── ├── └── └── ├── ├── <factor>
├── ├── ├── └── └── ├── ├── └── IDENTIFIER(c)
├── ├── ├── └── └── ├── ├── ARITHMETIC_OPERATOR(bagi)
├── ├── ├── └── └── ├── └── <factor>
├── ├── ├── └── └── ├── └── ├── LPARENTHESIS(()
├── ├── ├── └── └── ├── └── ├── <expression>
├── ├── ├── └── └── ├── └── ├── └── <simple-expression>
├── ├── ├── └── └── ├── └── ├── └── ├── <term>
├── ├── ├── └── └── ├── └── ├── └── ├── └── <factor>
├── ├── ├── └── └── ├── └── ├── └── ├── └── └── IDENTIFIER(d)
├── ├── ├── └── └── ├── └── ├── └── ├── ARITHMETIC_OPERATOR(-)
├── ├── ├── └── └── ├── └── ├── └── └── <term>
├── ├── ├── └── └── ├── └── ├── └── └── └── <factor>
├── ├── ├── └── └── ├── └── ├── └── └── └── └── NUMBER(2)
├── ├── ├── └── └── ├── └── └── RPARENTHESIS())
├── ├── ├── └── └── ├── ARITHMETIC_OPERATOR(+)
├── ├── ├── └── └── └── <term>
├── ├── ├── └── └── └── ├── <factor>
├── ├── ├── └── └── └── ├── └── IDENTIFIER(a)
├── ├── ├── └── └── └── ├── ARITHMETIC_OPERATOR(mod)
├── ├── ├── └── └── └── └── <factor>
├── ├── ├── └── └── └── └── └── IDENTIFIER(c)
├── ├── ├── SEMICOLON(;)


├── ├── ├── <assignment-statement>
├── ├── ├── ├── IDENTIFIER(y)
├── ├── ├── ├── ASSIGN_OPERATOR(:=)
├── ├── ├── └── <expression>
├── ├── ├── └── └── <simple-expression>
├── ├── ├── └── └── ├── <term>
├── ├── ├── └── └── ├── ├── <factor>
├── ├── ├── └── └── ├── ├── └── IDENTIFIER(a)
├── ├── ├── └── └── ├── ├── ARITHMETIC_OPERATOR(*)
├── ├── ├── └── └── ├── └── <factor>
├── ├── ├── └── └── ├── └── └── IDENTIFIER(b)
├── ├── ├── └── └── ├── ARITHMETIC_OPERATOR(+)
├── ├── ├── └── └── ├── <term>
├── ├── ├── └── └── ├── └── <factor>
├── ├── ├── └── └── ├── └── └── IDENTIFIER(c)
├── ├── ├── └── └── ├── ARITHMETIC_OPERATOR(-)
├── ├── ├── └── └── └── <term>
├── ├── ├── └── └── └── ├── <factor>
├── ├── ├── └── └── └── ├── └── IDENTIFIER(d)
├── ├── ├── └── └── └── ├── ARITHMETIC_OPERATOR(bagi)
├── ├── ├── └── └── └── └── <factor>
├── ├── ├── └── └── └── └── └── NUMBER(2)
├── ├── ├── SEMICOLON(;)
├── ├── ├── <assignment-statement>
├── ├── ├── ├── IDENTIFIER(valid)
├── ├── ├── ├── ASSIGN_OPERATOR(:=)
├── ├── ├── └── <expression>
├── ├── ├── └── └── <simple-expression>
├── ├── ├── └── └── ├── <term>
├── ├── ├── └── └── ├── └── <factor>
├── ├── ├── └── └── ├── └── ├── LPARENTHESIS(()
├── ├── ├── └── └── ├── └── ├── <expression>
├── ├── ├── └── └── ├── └── ├── ├── <simple-expression>
├── ├── ├── └── └── ├── └── ├── ├── └── <term>
├── ├── ├── └── └── ├── └── ├── ├── └── └── <factor>
├── ├── ├── └── └── ├── └── ├── ├── └── └── └── IDENTIFIER(a)
├── ├── ├── └── └── ├── └── ├── ├── RELATIONAL_OPERATOR(>)


├─ ├─ ├─ └─ └─ ├─ └─ ├─ └─ <simple-expression>
├─ ├─ ├─ └─ └─ ├─ └─ ├─ └─ └─ <term>
├─ ├─ ├─ └─ └─ ├─ └─ ├─ └─ └─ └─ <factor>
├─ ├─ ├─ └─ └─ ├─ └─ ├─ └─ └─ └─ └─ IDENTIFIER(b)
├─ ├─ ├─ └─ └─ ├─ └─ └─ RPARENTHESIS())
├─ ├─ ├─ └─ └─ ├─ LOGICAL_OPERATOR(atau)
├─ ├─ ├─ └─ └─ └─ <term>
├─ ├─ ├─ └─ └─ └─ └─ <factor>
├─ ├─ ├─ └─ └─ └─ └─ ├─ LPARENTHESIS(()
├─ ├─ ├─ └─ └─ └─ └─ ├─ <expression>
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ <simple-expression>
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ <term>
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ ├─ <factor>
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ ├─ ├─ LPARENTHESIS(()
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ ├─ ├─ <expression>
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ ├─ ├─ ├─
<simple-expression>
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ ├─ ├─ ├─ └─ <term>
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ ├─ ├─ ├─ └─ └─
<factor>
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ ├─ ├─ ├─ └─ └─ └─
IDENTIFIER(c)
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ ├─ ├─ ├─
RELATIONAL_OPERATOR(<)
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ ├─ ├─ └─
<simple-expression>
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ ├─ ├─ └─ └─ <term>
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ ├─ ├─ └─ └─ └─
<factor>
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ ├─ ├─ └─ └─ └─ └─
IDENTIFIER(d)
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ ├─ └─ RPARENTHESIS())
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ ├─ LOGICAL_OPERATOR(dan)
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ └─ <factor>
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ └─ ├─ LPARENTHESIS(()
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ └─ ├─ <expression>
├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ └─ ├─ ├─


<simple-expression>
<term>
<factor>
IDENTIFIER(x)
RELATIONAL_OPERATOR(<>)
<simple-expression>
<term>
<factor>
IDENTIFIER(y)
RPARENTHESIS())
RPARENTHESIS())
SEMICOLON(;)
<if-statement>
KEYWORD(jika)
<expression>
<simple-expression>
<term>
<factor>
LPARENTHESIS(()
<expression>
<simple-expression>
<term>
<factor>
IDENTIFIER(a)
ARITHMETIC_OPERATOR(+)
<term>
<factor>
IDENTIFIER(b)
RELATIONAL_OPERATOR(>)
<simple-expression>
<term>


├─ ├─ ├─ ├─ └─ └─ ├─ ├─ └─ └─ ├─ <factor>
├─ ├─ ├─ ├─ └─ └─ ├─ ├─ └─ └─ ├─ └─ IDENTIFIER(c)
├─ ├─ ├─ ├─ └─ └─ ├─ ├─ └─ └─ ├─
ARITHMETIC_OPERATOR(*)
├─ ├─ ├─ ├─ └─ └─ ├─ ├─ └─ └─ └─ <factor>
├─ ├─ ├─ ├─ └─ └─ ├─ ├─ └─ └─ └─ └─ IDENTIFIER(d)
├─ ├─ ├─ ├─ └─ └─ ├─ └─ RPARENTHESIS())
├─ ├─ ├─ ├─ └─ └─ ├─ LOGICAL_OPERATOR(dan)
├─ ├─ ├─ ├─ └─ └─ └─ <factor>
├─ ├─ ├─ ├─ └─ └─ └─ ├─ LOGICAL_OPERATOR(tidak)
├─ ├─ ├─ ├─ └─ └─ └─ └─ <factor>
├─ ├─ ├─ ├─ └─ └─ └─ └─ ├─ LPARENTHESIS(()
├─ ├─ ├─ ├─ └─ └─ └─ └─ ├─ <expression>
├─ ├─ ├─ ├─ └─ └─ └─ └─ ├─ ├─ <simple-expression>
├─ ├─ ├─ ├─ └─ └─ └─ └─ ├─ ├─ └─ <term>
├─ ├─ ├─ ├─ └─ └─ └─ └─ ├─ ├─ └─ └─ <factor>
├─ ├─ ├─ ├─ └─ └─ └─ └─ ├─ ├─ └─ └─ └─ IDENTIFIER(x)
├─ ├─ ├─ ├─ └─ └─ └─ └─ ├─ ├─ RELATIONAL_OPERATOR(=)
├─ ├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ <simple-expression>
├─ ├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ <term>
├─ ├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ └─ <factor>
├─ ├─ ├─ ├─ └─ └─ └─ └─ ├─ └─ └─ └─ └─ IDENTIFIER(y)
├─ ├─ ├─ ├─ └─ └─ └─ └─ └─ RPARENTHESIS())
├─ ├─ ├─ ├─ KEYWORD(maka)
├─ ├─ ├─ ├─ <procedure-call>
├─ ├─ ├─ ├─ ├─ IDENTIFIER(writeln)
├─ ├─ ├─ ├─ ├─ LPARENTHESIS(()
├─ ├─ ├─ ├─ ├─ <parameter-list>
├─ ├─ ├─ ├─ ├─ └─ <expression>
├─ ├─ ├─ ├─ ├─ └─ └─ <simple-expression>
├─ ├─ ├─ ├─ ├─ └─ └─ └─ <term>
├─ ├─ ├─ ├─ ├─ └─ └─ └─ └─ <factor>
├─ ├─ ├─ ├─ ├─ └─ └─ └─ └─ └─ STRING_LITERAL('Kondisi
kompleks terpenuhi')
├─ ├─ ├─ ├─ └─ RPARENTHESIS())
├─ ├─ ├─ ├─ KEYWORD(selain-itu)
├─ ├─ ├─ └─ <procedure-call>

```
├── ├── ├── └── ├── IDENTIFIER(writeln)
├── ├── ├── └── ├── LPARENTHESIS(()
├── ├── ├── └── ├── <parameter-list>
├── ├── ├── └── ├── └── <expression>
├── ├── ├── └── ├── └── └── <simple-expression>
├── ├── ├── └── ├── └── └── └── <term>
├── ├── ├── └── ├── └── └── └── └── <factor>
├── ├── ├── └── ├── └── └── └── └── └── STRING_LITERAL('Kondisi
tidak terpenuhi')
├── ├── ├── └── └── RPARENTHESIS())
├── ├── ├── SEMICOLON(;)
├── ├── └── <empty-statement>
├── └── KEYWORD(selesai)
└── DOT(.)
```

Pembahasan: Parser berhasil menangani ekspresi kompleks dengan multiple operators dan nested parentheses. Precedence operator terimplementasi dengan benar melalui hierarki expression-simple_expression-term-factor. Multiplicative operators (*, bagi, mod) berada di level term sehingga memiliki precedence lebih tinggi dari additive operators (+, -) yang ada di level simple_expression. Parentheses diparsing sebagai factor yang mengandung expression, memungkinkan override precedence. Logical operators "dan" dan "atau" juga mengikuti precedence yang benar dengan "dan" di level term dan "atau" di level simple_expression. Operator "tidak" sebagai unary operator diparsing di level factor.

3.2.6 Hasil Uji Error Missing Semicolon

```
Syntax Error: error: token tidak sesuai, ditemukan IDENTIFIER(y)
di posisi 14, diharapkan KEYWORD
```

Pembahasan: Parser berhasil mendeteksi missing semicolon dan melaporkan error dengan informasi yang akurat. Error terjadi pada posisi token ke-9 dalam stream. Parser mengharapkan SEMICOLON setelah assignment statement pertama tetapi menemukan IDENTIFIER(y) yang merupakan awal dari assignment statement berikutnya. Error message menunjukkan token yang ditemukan, posisinya, dan token yang diharapkan. Informasi ini cukup untuk developer mengetahui bahwa semicolon hilang di akhir statement sebelumnya.

3.2.7 Hasil Uji Error Missing Keyword

```
Syntax Error: error: token tidak sesuai, ditemukan IDENTIFIER(y)
di posisi 19, diharapkan KEYWORD
```

Pembahasan: Parser mendeteksi missing keyword "maka" dalam if-statement. Setelah parsing expression kondisi, parser mengharapkan keyword "maka" tetapi menemukan IDENTIFIER(writeln). Error terjadi karena statement setelah kondisi langsung dimulai tanpa keyword yang diperlukan. Error message menunjukkan posisi token dan tipe token yang diharapkan. Meskipun error message tidak menyebutkan secara spesifik keyword "maka", informasi posisi cukup untuk melokasi masalah.

Bab IV

Penutup

4.1 Kesimpulan

Berdasarkan perancangan, implementasi, dan pengujian yang telah dilakukan, dapat disimpulkan bahwa Parser untuk bahasa Pascal-S telah berhasil diimplementasikan menggunakan algoritma Recursive Descent. Parser menerima input berupa deretan token dari lexer dan memverifikasi kesesuaiannya dengan grammar Pascal-S. Setiap aturan produksi dalam grammar diimplementasikan sebagai method terpisah yang saling memanggil secara rekursif.

Grammar Pascal-S mencakup konstruksi lengkap mulai dari program structure, declarations (const, type, var, subprogram), statements (assignment, if, while, for, repeat, case, procedure call), hingga expressions dengan operator precedence. Total terdapat lebih dari 30 aturan produksi yang diimplementasikan dalam parser. Parser menghasilkan Parse Tree yang merepresentasikan struktur hierarkis program. Parse Tree memvisualisasikan bagaimana program diurai menurut grammar. Setiap node mewakili konstruksi sintaksis baik terminal maupun non-terminal. Tree dicetak dengan format ASCII art yang mudah dibaca.

Error handling terimplementasi dengan baik melalui custom SyntaxError exception. Parser dapat mendeteksi berbagai jenis syntax error seperti missing token, unexpected token, atau invalid construct. Error message memberikan informasi posisi token dan token yang diharapkan untuk memudahkan debugging. Pengujian dengan berbagai test case menunjukkan parser bekerja dengan benar untuk program yang valid dan dapat mendeteksi error untuk program yang tidak valid. Test case mencakup struktur sederhana hingga kompleks seperti nested loops, procedure dengan parameter, dan ekspresi dengan multiple operators.

Parser dapat menangani ambiguitas seperti membedakan assignment statement dan procedure call dengan lookahead sederhana. Parser melihat token berikutnya setelah identifier untuk menentukan apakah ada assign operator atau parentheses. Integrasi antara lexer (Milestone 1) dan parser (Milestone 2) berjalan lancar. Output lexer berupa list of token tuples menjadi input parser. Compiler driver mengorkestrasi kedua komponen dengan baik. Program mendukung dua mode input: source code (.pas) dan tokenized file (.txt).

4.2 Saran

Meskipun parser telah berfungsi sesuai spesifikasi, beberapa aspek dapat ditingkatkan untuk development selanjutnya. Error recovery dapat ditambahkan agar parser dapat melanjutkan parsing setelah menemukan error. Saat ini parser berhenti pada error pertama. Dengan error recovery, parser dapat melaporkan multiple errors dalam satu run. Teknik seperti panic mode recovery atau phrase-level recovery dapat diimplementasikan. Error message dapat dibuat lebih spesifik dan user-friendly. Saat ini error message menunjukkan tipe token yang diharapkan. Pesan dapat ditingkatkan dengan menjelaskan konteks parsing dan memberikan suggestion. Parse Tree

dapat disimpan dalam format yang lebih structured seperti JSON atau XML. Format saat ini berupa string ASCII art cocok untuk debugging tetapi sulit untuk processing lebih lanjut. Representasi JSON memudahkan tahap selanjutnya seperti semantic analysis.

Performance dapat dioptimasi untuk source code yang sangat besar. Saat ini parser memuat seluruh token list ke memory. Untuk file besar, streaming approach dapat dipertimbangkan. Profiling dapat dilakukan untuk mengidentifikasi bottleneck. Documentation dapat ditingkatkan dengan menambahkan docstring untuk setiap method. Docstring menjelaskan grammar rule yang diimplementasikan, parameter, return value, dan possible exceptions.

Lampiran

Link Github Repository : https://github.com/fliegenhaan/JJK-Tubes-IF2224.git

Link Diagram :

Pembagian Tugas :

No	Tugas	NIM
1	Implementasi kode program (`compiler.py`, `lexer.py`, `parser.py`).	13523136, 13523142
2	Perancangan DFA rules.	13523136, 13523124
3	Dokumentasi laporan.	13523136, 13523155, 13523124, 13523142
4	Perancaangan diagram.	123523124, 13523136

Daftar Pustaka

Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung. Spesifikasi Tugas Besar IF2224 - Formal Languages and Automata Theory, Milestone 1 - Lexical Analysis. Diakses pada 17 Oktober 2025, dari https://docs.google.com/document/d/1w0GmHW5L0gKZQWbgmtJPFmOzlpSWBknNPdugucn4eII/edit?usp=sharing

GeeksforGeeks. "Introduction to Lexical Analysis." GeeksforGeeks. Diakses pada 14 Oktober 2025, dari https://www.geeksforgeeks.org/introduction-of-lexical-analysis/

GeeksforGeeks. "Deterministic Finite Automaton (DFA)." GeeksforGeeks. Diakses pada 15 Oktober 2025, dari https://www.geeksforgeeks.org/introduction-of-finite-automata/

TutorialsPoint. "Compiler Design - Lexical Analysis." TutorialsPoint. Diakses pada 15 Oktober 2025, dari https://www.tutorialspoint.com/compiler_design/compiler_design_lexical_analysis.htm

Free Pascal Documentation. "Pascal Language Reference." Free Pascal Wiki. Diakses pada 15 Oktober 2025, dari https://www.freepascal.org/docs.html

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. Compilers: Principles, Techniques, and Tools (2nd Edition). Pearson Education, 2006.