Laporan Milestone 2
Tugas Besar
IF2224-Teori Bahasa Formal dan Otomata

Disusun oleh:

Muhammad Alfansya	13523005
Orvin Andika Ikhsan Abhista	13523017
Dzaky Aurelia Fawwaz	13523065
Ferdin Arsenarendra Purtadi	13523117

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2025

1. Landasan Teori

Bahasa pemrograman adalah bahasa yang selalu digunakan untuk melakukan operasi-operasi yang melibatkan komputasi. Terkait bahasa pemrograman ini, proses untuk mengeksekusi bahasa pemrograman tersebut tidaklah mudah. Terdapat banyak proses yang perlu dilakukan oleh *compiler* supaya kode yang ditulis benar dan bisa memberikan output yang sesuai dengan bahasa pemrograman yang ditulis.

1.1. Analisis Sintaks

Analisis Sintaks atau parsing adalah tahap kedua dalam proses kompilasi setelah analisis leksikal. Pada tahap ini compiler memeriksa susunan token-token, yang sudah dibuat di tahap sebelumnya, untuk memastikan bahwa program mengikuti aturan tata bahasa (grannar) yang sudah didefinisikan. Grammar dituliskan dalam bentuk *context free grammar* (CFG) yang mendefinisikan bagaimana token dapat disusun menjadi struktur yang lebih besar seperti ekspresi, pernyataan, atau deklarasi

Tujuan utama dari analisis sintaks adalah untuk membangun representasi struktural dari program seperti parse tree yang merepresentasikan struktur hierarkis program sumber berdasarkan grammar yang telah didefinisikan, Apabila terdapat susunan token yang tidak sesuai dengan grammar, parses akan mengirim pesan kesalahan (*syntax error*). Tahap ini juga bertujuan untuk mempersiapkan representasi program untuk tahap berikutnya.

1.2. Parse Tree

Parse Tree adalah representasi grafis hierarkis yang menunjukkan bagaimana sebuah program dapat diturunkan dari grammar bahasa pemrograman. Parse tree memvisualisasikan proses derivasi dari start symbol hingga mencapai string program yang lengkap, dengan setiap node dalam tree merepresentasikan aplikasi dari satu production rule. Dalam parse tree, root node selalu merepresentasikan start symbol dari grammar, yang dalam kasus Pascal-S adalah <program>. Internal nodes merepresentasikan non-terminal symbols seperti <expression>, <statement>, dan <declaration-part>, sementara leaf nodes adalah terminal symbols berupa token-token aktual dari lexer seperti IDENTIFIER(x), NUMBER(5), dan OPERATOR(+). Struktur hierarkis parse tree mencerminkan urutan aplikasi production rules, di mana node untuk non-terminal di sisi kiri rule menjadi parent, dan simbol-simbol di sisi kanan rule menjadi children dari node tersebut. Proses ini berlanjut secara rekursif hingga semua leaf nodes adalah terminal symbols.

Parse tree berbeda dengan Abstract Syntax Tree (AST) dalam hal level of detail yang direpresentasikan. Parse tree merepresentasikan semua detail derivasi grammar, termasuk simbol-simbol yang diperlukan untuk parsing tetapi tidak penting untuk semantic analysis atau code generation, seperti parentheses, semicolons, dan keywords. Sebagai contoh, dalam ekspresi (a + b), parse tree akan menyimpan node untuk parentheses kiri dan kanan, sedangkan AST hanya akan menyimpan struktur esensial

yaitu operator plus dengan operand a dan b. Parse tree bersifat lebih verbose dan detail, sementara AST lebih kompak dan focused pada struktur semantik program. Parse tree sangat berguna untuk memvisualisasikan dan memverifikasi bahwa parser memahami struktur program dengan benar, serta menjadi input untuk tahap semantic analysis dalam proses kompilasi selanjutnya.

Dalam implementasi parser Pascal-S, setiap node dalam parse tree memiliki struktur yang konsisten. Tabel berikut menjelaskan berbagai jenis node yang mungkin muncul dalam parse tree beserta deskripsi, aturan produksi, dan contoh penggunaannya. Tabel node parse tree mendokumentasikan seluruh konstruksi bahasa Pascal-S yang dapat muncul sebagai node dalam parse tree. Setiap baris dalam tabel merepresentasikan satu jenis node yang memiliki karakteristik dan struktur tersendiri. Kolom Nama Node menunjukkan identifier yang digunakan untuk node tersebut dalam implementasi parser, ditulis dalam format <nama-node> untuk non-terminal. Kolom Deskripsi menjelaskan fungsi dan makna semantik dari node tersebut dalam konteks program Pascal-S. Kolom Aturan Produksi menunjukkan grammar rule yang mendefinisikan bagaimana node tersebut dapat diturunkan menjadi komponen-komponennya, memberikan pemahaman tentang struktur sintaks yang valid. Kolom Contoh memberikan ilustrasi konkret dari source code Pascal-S yang akan menghasilkan node tersebut. Tabel ini berfungsi sebagai referensi lengkap untuk memahami struktur parse tree yang dihasilkan parser dan membantu dalam proses debugging dengan memungkinkan verifikasi bahwa setiap konstruksi bahasa di-parse dengan benar sesuai grammar yang telah didefinisikan.

1.3. Recursive Descent

Recursive Descent adalah teknik parsing top-down yang mengimplementasikan setiap non-terminal dalam grammar sebagai fungsi atau prosedur terpisah. Teknik ini disebut "recursive" karena fungsi-fungsi parsing dapat memanggil satu sama lain secara rekursif sesuai dengan struktur grammar, dan disebut "descent" karena parsing dimulai dari start symbol (top) dan turun menuju terminal symbols (bottom) melalui aplikasi production rules. Prinsip dasar Recursive Descent adalah adanya korespondensi langsung antara struktur grammar dan struktur kode parser, di mana untuk setiap non-terminal dalam grammar dibuat sebuah fungsi parsing tersendiri yang bertanggung jawab untuk mengenali konstruksi bahasa tersebut.
Karakteristik utama Recursive Descent meliputi:

- Korespondensi Grammar-Kode: Setiap non-terminal dalam grammar direpresentasikan sebagai satu fungsi parsing
- Pemanggilan Rekursif: Fungsi parsing memanggil fungsi lain secara rekursif sesuai dengan production rules
- Validasi Token: Setiap fungsi membaca dan memvalidasi token dari input stream
- Pembentukan Parse Tree: Struktur call stack secara alami membentuk parse tree

Parsing dengan Recursive Descent dimulai dari fungsi untuk start symbol yang kemudian memproses komponen-komponennya secara berurutan sesuai production rule. Ketika menemui non-terminal, fungsi akan memanggil fungsi parsing untuk non-terminal

tersebut, yang kemudian memverifikasi keberadaan setiap terminal yang diharapkan. Proses ini berlanjut secara rekursif hingga seluruh input selesai diparse.
Mekanisme parsing melibatkan:

- Lookahead: Parser melihat satu token ke depan untuk menentukan production rule yang akan diterapkan
- Token Matching: Verifikasi bahwa token saat ini sesuai dengan terminal yang diharapkan
- Recursive Calls: Pemanggilan fungsi parsing untuk non-terminal yang muncul dalam production rule
- Error Detection: Pelaporan syntax error ketika token tidak sesuai dengan yang diharapkan

Untuk grammar LL(1), Recursive Descent dapat bekerja efisien tanpa backtracking dengan hanya melihat satu token ke depan untuk menentukan production rule mana yang harus diterapkan. Lookahead ini memungkinkan parser membuat keputusan deterministik tentang jalur parsing yang harus diambil, misalnya untuk membedakan antara assignment statement dan procedure call ketika menemui identifier.

1.4. Grammar dan Produksi

Grammar dalam konteks syntax analysis adalah seperangkat aturan formal yang mendefinisikan struktur valid dari bahasa pemrograman. Grammar berfungsi sebagai spesifikasi lengkap yang mendeskripsikan bagaimana token-token dari lexer dapat dikombinasikan untuk membentuk program yang sintaktiknya benar. Sebuah grammar formal terdiri dari empat komponen utama yang saling berinteraksi untuk mendefinisikan bahasa secara lengkap dan presisi.
Komponen-komponen grammar:

- Terminal Symbols: Simbol-simbol yang merupakan token dari lexer dan tidak dapat diturunkan lebih lanjut (contoh: KEYWORD, IDENTIFIER, NUMBER, SEMICOLON)
- Non-Terminal Symbols: Simbol-simbol yang merepresentasikan konstruksi bahasa dan dapat diturunkan menjadi kombinasi simbol lain (contoh: <program>, <expression>, <statement>)
- Production Rules: Aturan yang mendefinisikan bagaimana non-terminal dapat diturunkan menjadi urutan terminal dan non-terminal
- Start Symbol: Non-terminal khusus yang menjadi titik awal derivasi, untuk Pascal-S adalah <program>

Production rules adalah jantung dari grammar yang mendefinisikan bagaimana sebuah non-terminal dapat diturunkan menjadi kombinasi terminal dan non-terminal lainnya. Aturan produksi ditulis dalam format non-terminal → urutan simbol, di mana panah (→) menunjukkan bahwa non-terminal di sebelah kiri dapat digantikan dengan urutan simbol di sebelah kanan. Sebagai contoh, production rule <program> → <program-header> <declaration-part> <compound-statement> DOT menyatakan bahwa

sebuah program Pascal-S terdiri dari header program, bagian deklarasi, compound statement, dan diakhiri dengan titik.

2. Perancangan & Implementasi

2.1. Implementasi

2.1.1. Struktur Kelas Parser

Implementasi parser Pascal-S menggunakan class Parser yang memiliki struktur state management untuk navigasi token stream. Class ini diinisialisasi dengan menerima list of tokens dari lexer dan mempersiapkan state awal untuk proses parsing.

```
class Parser:
    def __init__(self, tokens):
        if not tokens:
            raise ValueError("Token list is empty!
Cannot initialize parser.")
        self.tokens = tokens
        self.pos = 0
        self.current_token = self.tokens[self.pos]
        self.skip_comments()
```

Inisialisasi parser langsung memanggil skip_comments() untuk memastikan token pertama yang diproses bukan komentar, sehingga parsing dapat dimulai dari token yang meaningful.

2.1.2. Utility Methods

2.1.2.1. Comment handling

Method ini menggunakan loop untuk melewati semua token komentar berturut-turut. Setiap kali menemukan COMMENT_START atau COMMENT_END, parser akan maju ke token berikutnya hingga menemukan token non-komentar atau mencapai end of file.

```
def skip_comments(self):
        # Melewatkan token COMMENT_START dan
COMMENT_END
        while self.current_token and
self.current_token[0] in ('COMMENT_START',
'COMMENT_END'):
            self.pos += 1
            if self.pos < len(self.tokens):
                self.current_token =
self.tokens[self.pos]
            else:
                self.current_token = None
                break
```

2.1.2.2. Token Navigation

Method advance() memindahkan pointer ke token berikutnya dan secara otomatis melewati komentar yang mungkin ada. Jika sudah mencapai akhir token list, current_token di-set menjadi None.

```
def advance(self):
        # Maju ke token berikutnya
        self.pos += 1
        if self.pos < len(self.tokens):
            self.current_token =
self.tokens[self.pos]
            self.skip_comments()
        else:
            self.current_token = None
```

Method peek() memungkinkan parser melihat token berikutnya tanpa mengubah state. Method ini penting untuk lookahead, misalnya membedakan assignment statement dari procedure call. Method ini juga mengabaikan komentar dan mengembalikan token meaningful pertama yang ditemukan.

```
def peek(self):
        # Mengintip token berikutnya
        peek_pos = self.pos + 1
        while peek_pos < len(self.tokens):
            token = self.tokens[peek_pos]
            if token[0] not in
('COMMENT_START', 'COMMENT_END'):
                return token # Kembalikan
token non-komentar pertama
            peek_pos += 1
        return None
```

2.1.2.3. Token Matching and validation

Method eat() adalah core utility untuk validasi token dan pembangunan parse tree:

```
def eat(self, token_type, token_value=None):
        # Memeriksa token, membuat node
terminal, dan maju.
        if (self.current_token and
            self.current_token[0] ==
token_type and
            (token_value is None or
self.current_token[1].lower() ==
```

```
token_value)):

            token = self.current_token
            self.advance()

            node_name =
f"{token[0]}({token[1]!r})"
            return Node(node_name, token)
        else:
            expected = f"{token_type}
('{token_value}')" if token_value else
token_type
            found = f"{self.current_token[0]}
('{self.current_token[1]}')" if
self.current_token else "EOF"
            raise SyntaxError(f"Syntax Error:
Expected {expected} but got {found}")
```

Method ini melakukan beberapa tugas penting, yaitu memeriksa apakah current_token sesuai dengan token_type dan optional token_value, membuat node terminal untuk parse tree jika validasi berhasil, memindahkan pointer ke token berikutnya, dan melaporkan syntax error yang informatif jika validasi gagal

2.1.3. Implementasi Grammar Rules

2.1.3.1. Program Structure

Method parse_program() mengimplementasikan production rule <program> → <program-header> <declaration-part> <compound-statement> DOT. Method ini membuat node "program" dan menambahkan empat children sesuai urutan dalam grammar.

```
def parse_program(self):
    node = Node("program")

node.add_child(self.parse_program_header())

node.add_child(self.parse_declaration_part())

node.add_child(self.parse_compound_statement()
)
    node.add_child(self.eat("DOT"))
    return node
```

Method parse_program_header() mengimplementasikan rule <program-header> → KEYWORD(program) IDENTIFIER SEMICOLON. Setiap komponen divalidasi menggunakan eat() dan ditambahkan sebagai child node.

```
def parse_program_header(self):
    node = Node("program-header")
    node.add_child(self.eat("KEYWORD",
"program"))
    node.add_child(self.eat("IDENTIFIER"))
    node.add_child(self.eat("SEMICOLON"))
    return node
```

2.1.3.2. Deklarasi Variabel

Implementasi ini menggunakan while loop untuk menangani repetisi dalam production rule (<identifier-list> COLON <type> SEMICOLON)+. Loop terus berjalan selama token saat ini adalah identifier, memungkinkan multiple variable declarations.

```
def parse_var_declaration(self):
    node = Node("var-declaration")
    node.add_child(self.eat("KEYWORD",
"variabel"))

    while self.current_token and
self.current_token[0] == 'IDENTIFIER':

node.add_child(self.parse_identifier_list())
        node.add_child(self.eat("COLON"))
        node.add_child(self.parse_type())
        node.add_child(self.eat("SEMICOLON"))
    return node
```

Method ini menangani list identifier yang dipisahkan koma, seperti a, b, c. Loop terus memproses selama menemukan koma, menambahkan pasangan koma dan identifier ke parse tree.

```
def parse_identifier_list(self):
    node = Node("identifier-list")
    node.add_child(self.eat("IDENTIFIER"))
    while self.current_token and
self.current_token[0] == 'COMMA':
        node.add_child(self.eat("COMMA"))
        node.add_child(self.eat("IDENTIFIER"))
    return node
```

2.1.3.3. Spesifikasi Tipe

Method ini mengimplementasikan pilihan alternatif dalam grammar menggunakan conditional statements. Jika token adalah tipe primitif,

langsung di-eat. Jika keyword "larik", memanggil parse_array_type() untuk memproses array type definition.

```
def parse_type(self):
    node = Node("type")
    if self.current_token[0] == 'KEYWORD' and
self.current_token[1] in ['integer', 'real',
'boolean', 'char']:
        node.add_child(self.eat("KEYWORD",
self.current_token[1]))
    elif self.current_token[0] == 'KEYWORD'
and self.current_token[1] == 'larik':

node.add_child(self.parse_array_type())
    else:
        token_info = f"{self.current_token[0]}
('{self.current_token[1]}')" if
self.current_token else "EOF"
        raise SyntaxError(f"Syntax Error:
Expected type specification (integer, real,
larik, ...) but got {token_info}")
    return node
```

Array type parsing mengikuti struktur larik[range] dari type, dengan recursive call ke parse_type() untuk element type

```
def parse_array_type(self):
    node = Node("array-type")
    node.add_child(self.eat("KEYWORD",
"larik"))
    node.add_child(self.eat("LBRACKET"))
    node.add_child(self.parse_range())
    node.add_child(self.eat("RBRACKET"))
    node.add_child(self.eat("KEYWORD",
"dari"))
    node.add_child(self.parse_type())
    return node
```

2.1.3.4. Statement Parsing

Method parse_statement() menggunakan lookahead untuk disambiguation. Ketika menemui identifier, parser menggunakan peek() untuk melihat token berikutnya:

- Jika :=, maka ini assignment statement
- Jika bukan, maka ini procedure call

Untuk keyword, parser menggunakan conditional branching berdasarkan nilai keyword.

```
def parse_statement(self):
    if not self.current_token or
(self.current_token[0] == 'KEYWORD' and
self.current_token[1] == 'selesai'):
        return Node("empty-statement")

    if self.current_token[0] == 'IDENTIFIER':
        next_token = self.peek()

        if next_token and next_token[0] ==
'ASSIGN_OPERATOR':
            return
self.parse_assignment_statement()
        else:
            return self.parse_procedure_call()

    elif self.current_token[0] == 'KEYWORD':
        val = self.current_token[1]
        if val == 'jika':
            return self.parse_if_statement()
        elif val == 'selama':
            return
self.parse_while_statement()
        elif val == 'untuk':
            return self.parse_for_statement()
        elif val == 'mulai':
            return
self.parse_compound_statement()
        elif val == 'writeln':
            return self.parse_procedure_call()
        else:
            return Node("empty-statement")
    else:
        return Node("empty-statement")
```

2.1.3.5. Assignment Statement

Implementasi straightforward mengikuti rule <assignment-statement> → IDENTIFIER ASSIGN_OPERATOR <expression>.

```
def parse_assignment_statement(self):
    node = Node("assignment-statement")
    node.add_child(self.eat("IDENTIFIER"))

node.add_child(self.eat("ASSIGN_OPERATOR"))
    node.add_child(self.parse_expression())
    return node
```

2.1.3.6. Control Flow Statements

If statement mengimplementasikan optional else clause menggunakan conditional check. Jika keyword "selainitu" ditemukan, maka else branch diproses.

```
def parse_if_statement(self):
    node = Node("if-statement")
    node.add_child(self.eat("KEYWORD",
"jika"))
    node.add_child(self.parse_expression())
    node.add_child(self.eat("KEYWORD",
"maka"))
    node.add_child(self.parse_statement())
    if self.current_token and
self.current_token[0] == 'KEYWORD' and
self.current_token[1] == 'selainitu':
        node.add_child(self.eat("KEYWORD",
"selainitu"))
        node.add_child(self.parse_statement())
    return node
```

For loop mengimplementasikan pilihan antara "ke" (upward) dan "turunke" (downward) dengan conditional check, throwing error jika keduanya tidak ditemukan.

```
def parse_for_statement(self):
    node = Node("for-statement")
    node.add_child(self.eat("KEYWORD",
"untuk"))
    node.add_child(self.eat("IDENTIFIER"))

node.add_child(self.eat("ASSIGN_OPERATOR"))
    node.add_child(self.parse_expression())

    if self.current_token and
self.current_token[0] == 'KEYWORD' and
self.current_token[1] == 'ke':
        node.add_child(self.eat("KEYWORD",
"ke"))
    elif self.current_token and
self.current_token[0] == 'KEYWORD' and
self.current_token[1] == 'turunke':
        node.add_child(self.eat("KEYWORD",
"turunke"))
    else:
        raise SyntaxError("Syntax Error:
Expected 'ke' or 'turunke' in for loop")
```

```
    node.add_child(self.parse_expression())
    node.add_child(self.eat("KEYWORD",
"lakukan"))
    node.add_child(self.parse_statement())
    return node
```

2.1.3.7. Expression Parsing dengan Precedence

Expression parsing mengimplementasikan operator precedence melalui hierarchical method calls:
Level tertinggi: relational operators (=, <>, <, >, <=, >=)

```
def parse_expression(self):
    node = Node("expression")

node.add_child(self.parse_simple_expression())
    if self.current_token and
self.current_token[0] ==
'RELATIONAL_OPERATOR':
        rel_op_node =
self.eat("RELATIONAL_OPERATOR")
        node.add_child(rel_op_node)

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

Level menengah: additive operators (+, -, atau). Method ini juga menangani unary plus/minus di awal expression.

```
def parse_simple_expression(self):
    node = Node("simple-expression")

    if self.current_token and
self.current_token[0] == 'ARITHMETIC_OPERATOR'
and self.current_token[1] in ['+', '-']:

node.add_child(self.eat("ARITHMETIC_OPERATOR",
self.current_token[1]))

    node.add_child(self.parse_term())

    while (self.current_token and
           ( (self.current_token[0] ==
'ARITHMETIC_OPERATOR' and
self.current_token[1] in ['+', '-']) or
             (self.current_token[0] ==
'LOGICAL_OPERATOR' and self.current_token[1]
== 'atau') ) ):
```

```
        if self.current_token[1] == 'atau':

node.add_child(self.eat("LOGICAL_OPERATOR",
"atau"))
        else:

node.add_child(self.eat("ARITHMETIC_OPERATOR",
self.current_token[1]))
        node.add_child(self.parse_term())
    return node
```

Level terendah: multiplicative operators (*, /, bagi, mod, dan). Precedence lebih tinggi dari additive.

```
def parse_term(self):
    node = Node("term")
    node.add_child(self.parse_factor())

    while (self.current_token and
           ( (self.current_token[0] ==
'ARITHMETIC_OPERATOR' and
self.current_token[1] in ['*', '/', 'bagi',
'mod']) or
             (self.current_token[0] ==
'LOGICAL_OPERATOR' and self.current_token[1]
== 'dan') ) ):

        if self.current_token[1] in ['*',
'/']:

node.add_child(self.eat("ARITHMETIC_OPERATOR",
self.current_token[1]))
        elif self.current_token[1] in ['bagi',
'mod']:

node.add_child(self.eat("ARITHMETIC_OPERATOR",
self.current_token[1]))
        elif self.current_token[1] == 'dan':

node.add_child(self.eat("LOGICAL_OPERATOR",
"dan"))

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

Factor adalah unit terkecil expression. Method ini menggunakan lookahead untuk membedakan identifier biasa dari function call. Untuk

parenthesized expression, recursive call ke parse_expression() memungkinkan nesting expression arbitrary.

```
def parse_factor(self):
    node = Node("factor")

    if not self.current_token:
        raise SyntaxError("Syntax Error:
Unexpected EOF, expected a factor")

    if self.current_token[0] == 'IDENTIFIER':
        next_token = self.peek()

        if next_token and next_token[0] ==
'LPARENTHESIS':
            return self.parse_function_call()
        else:

node.add_child(self.eat("IDENTIFIER"))
    elif self.current_token[0] == 'NUMBER':
        node.add_child(self.eat("NUMBER"))
    elif self.current_token[0] ==
'CHAR_LITERAL':

node.add_child(self.eat("CHAR_LITERAL"))
    elif self.current_token[0] ==
'STRING_LITERAL':

node.add_child(self.eat("STRING_LITERAL"))
    elif self.current_token[0] ==
'LPARENTHESIS':

node.add_child(self.eat("LPARENTHESIS"))

node.add_child(self.parse_expression())

node.add_child(self.eat("RPARENTHESIS"))
    elif self.current_token[0] ==
'LOGICAL_OPERATOR' and self.current_token[1]
== 'tidak':

node.add_child(self.eat("LOGICAL_OPERATOR",
"tidak"))
        node.add_child(self.parse_factor())
    else:
        token_info = f"{self.current_token[0]}
('{self.current_token[1]}')" if
self.current_token else "EOF"
        raise SyntaxError(f"Syntax Error:
Expected factor (ID, Number, '(', 'tidak',
```

```
...) but got {token_info}")
    return node
```

2.1.4. Error Handling

Setiap error message mencakup:

- Apa yang diharapkan (expected token type/value)
- Apa yang ditemukan (actual token)
- Konteks berdasarkan posisi parsing

Error handling terintegrasi dalam:

- Method eat() untuk token mismatch
- Method parsing individual untuk konstruksi yang invalid
- Main parsing flow untuk unexpected EOF

```
raise SyntaxError(f"Syntax Error: Expected
{expected} but got {found}")
```

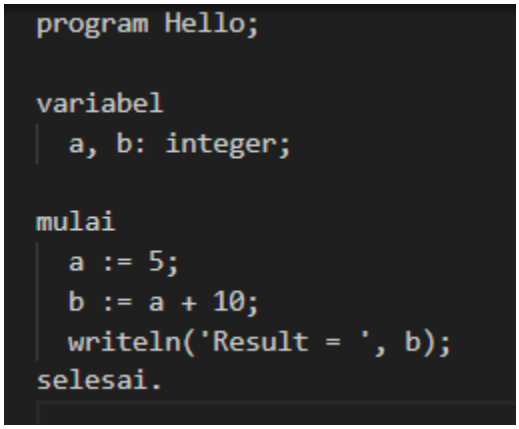
2.1.5. Integrasi

Main program melakukan inisialisasi parser dengan token list dari lexer, memanggil parse() yang memulai dari parse_program(), menangkap dan melaporkan SyntaxError jika terjadi, dan print parse tree jika parsing berhasil.

```
try:
    parser = Parser(token_list)
    print("\n----------- PARSE TREE -----------")
    parser.parse()

print("----------------------------------------\n
")

except SyntaxError as e:
    print(f"\n[SYNTAX ERROR]: Berhenti.")
    print(f"Pesan: {e}")
    sys.exit(1)
```

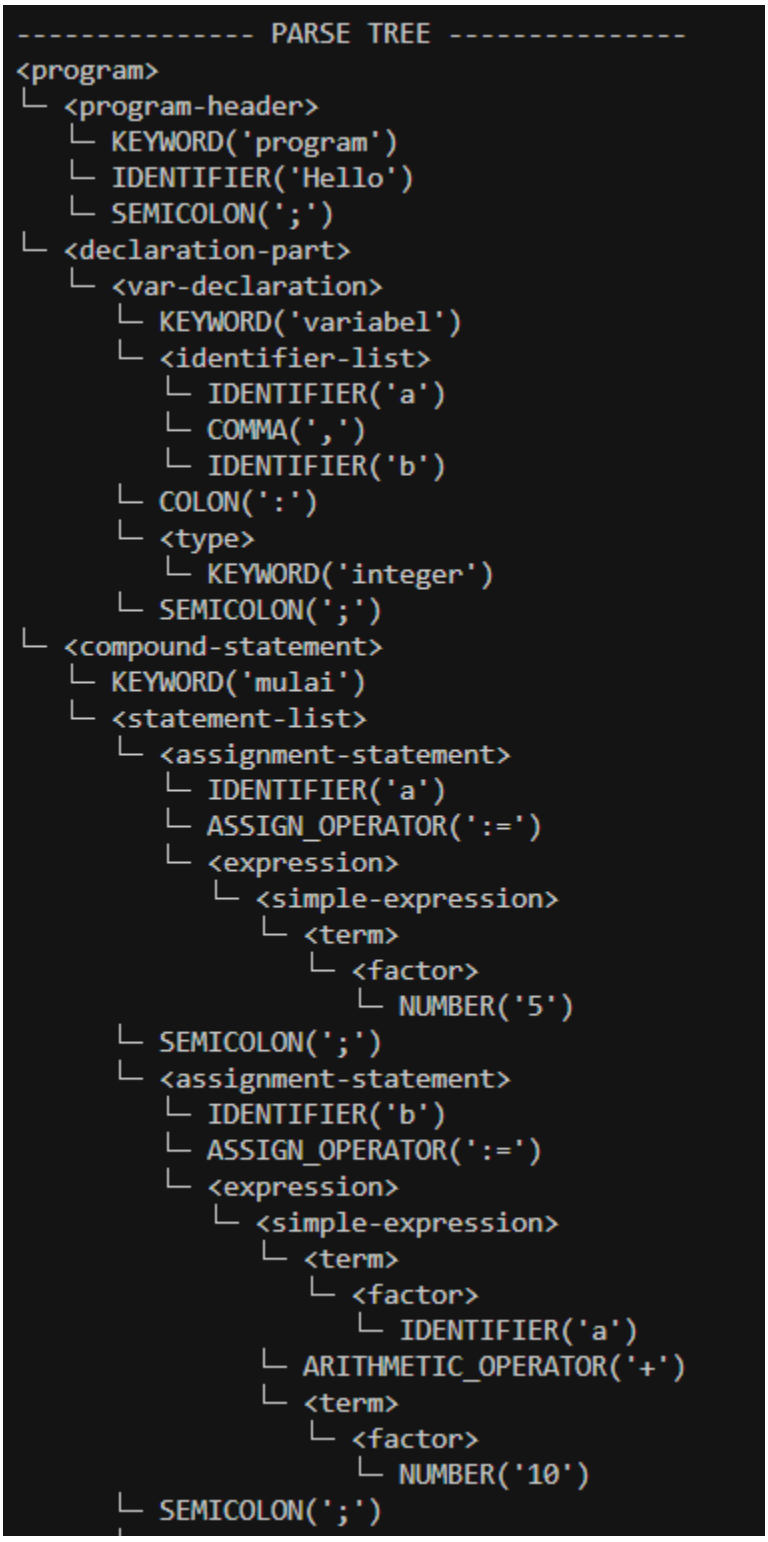
3. Pengujian

3.1. Pengujian 1 : Program dengan Struktur Dasar

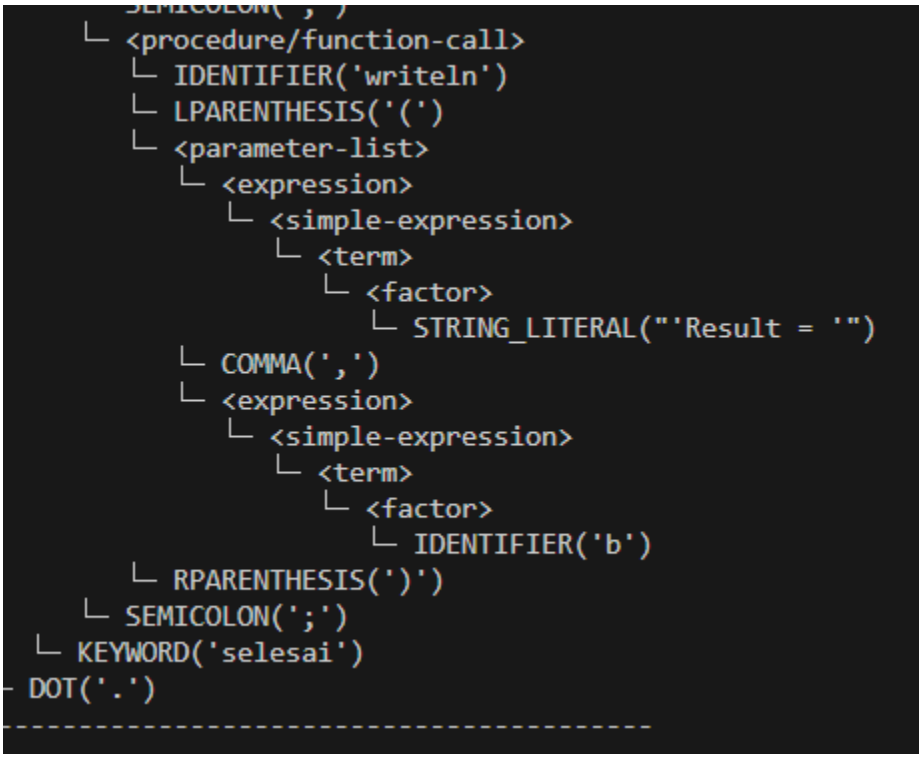
Input

Output

```
program Hello;

variabel
  a, b: integer;

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

Input


--------------- 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(';')

Input



Penjelasan

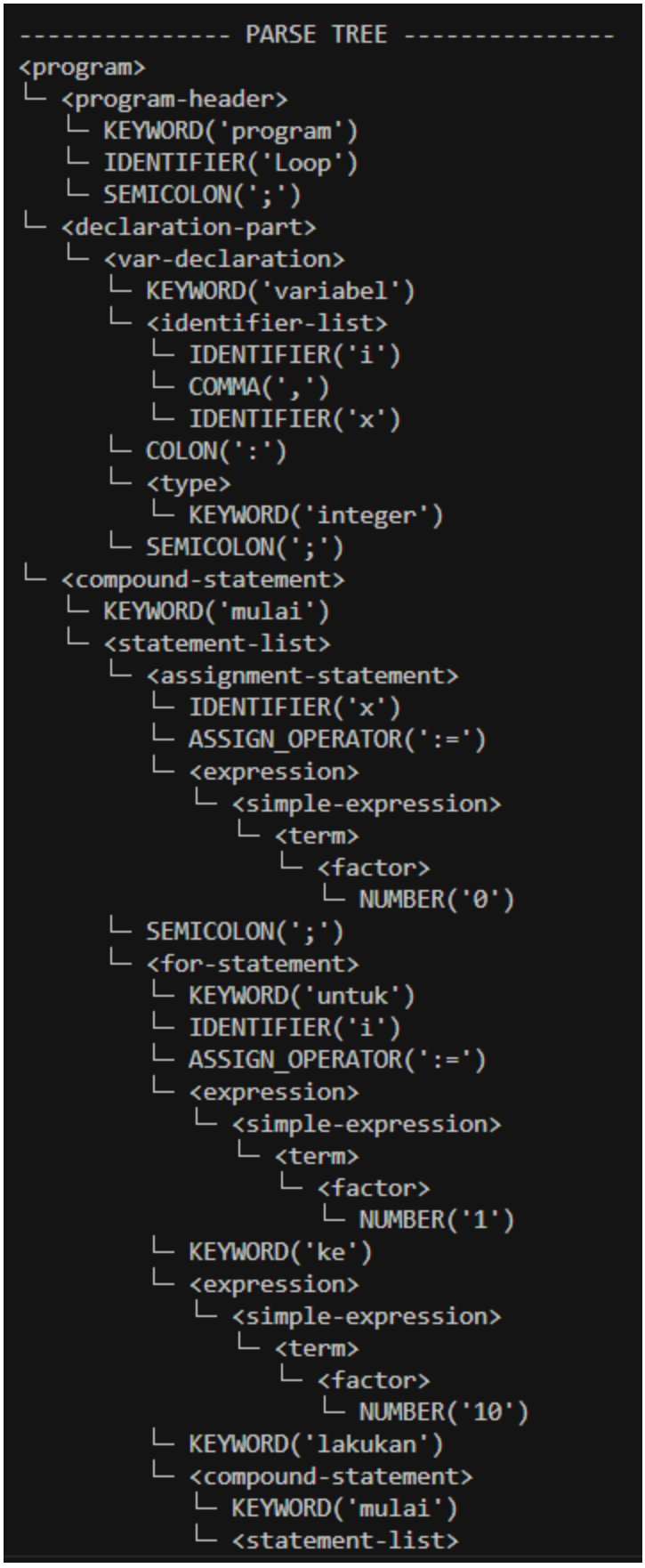
Parser berhasil mengenali struktur program lengkap (<program>, <declaration-part>, dan <compound-statement>. Parse Tree yang dihasilkan secara akurat memecah blok mulai-selesai menjadi sebuah <statement_list> yang berisi tiga statement. Ekspresi dan paramater diurai dengan benar ke dalam node <expression>, <term>, dan <factor>.

3.2. Pengujian 2 : Program dengan Loop dan Kondisional

Input

```
program Loop;
variabel
  i, x : integer;
mulai
  x := 0;

  untuk i := 1 ke 10 lakukan
    mulai
      x := x + i;
      jika (i = 5) maka
        writeln('Setengah jalan!')
      selain_itu
        writeln(i);
    selesai;

  selama x > 100 lakukan
    x := x - 1;

selesai.
```

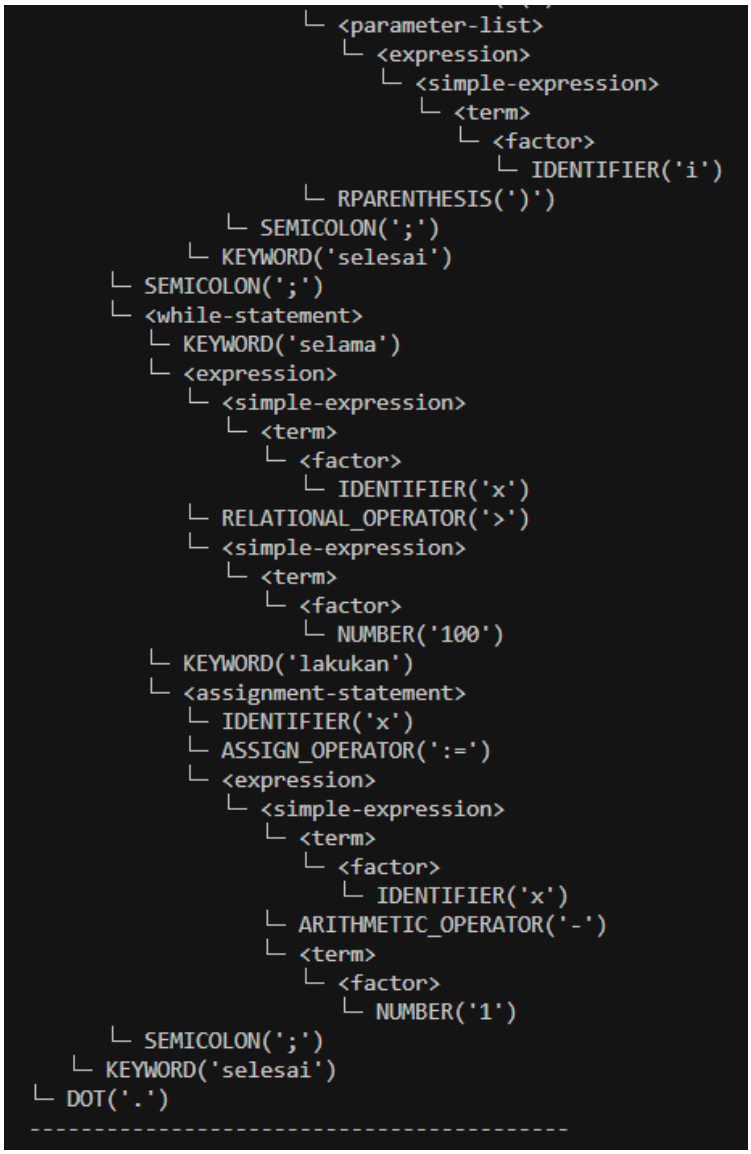
Output


--------------- PARSE TREE ---------------
<program>
└ <program-header>
└ KEYWORD('program')
└ IDENTIFIER('Loop')
└ SEMICOLON(';')
└ <declaration-part>
└ <var-declaration>
└ KEYWORD('variabel')
└ <identifier-list>
└ IDENTIFIER('i')
└ COMMA(',')
└ IDENTIFIER('x')
└ COLON(':')
└ <type>
└ KEYWORD('integer')
└ SEMICOLON(';')
└ <compound-statement>
└ KEYWORD('mulai')
└ <statement-list>
└ <assignment-statement>
└ IDENTIFIER('x')
└ 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')
└ <compound-statement>
└ KEYWORD('mulai')
└ <statement-list>

Input


└ <assignment-statement>
└ IDENTIFIER('x')
└ ASSIGN_OPERATOR(':=')
└ <expression>
└ <simple-expression>
└ <term>
└ <factor>
└ IDENTIFIER('x')
└ ARITHMETIC_OPERATOR('+')
└ <term>
└ <factor>
└ IDENTIFIER('i')
└ SEMICOLON(';')
└ <if-statement>
└ KEYWORD('jika')
└ <expression>
└ <simple-expression>
└ <term>
└ <factor>
└ LPARENTHESIS('(')
└ <expression>
└ <simple-expression>
└ <term>
└ <factor>
└ IDENTIFIER('i')
└ RELATIONAL_OPERATOR('=')
└ <simple-expression>
└ <term>
└ <factor>
└ NUMBER('5')
└ RPARENTHESIS(')')
└ KEYWORD('maka')
└ <procedure/function-call>
└ IDENTIFIER('writeln')
└ LPARENTHESIS('(')
└ <parameter-list>
└ <expression>
└ <simple-expression>
└ <term>
└ <factor>
└ STRING_LITERAL("'Setengah jalan!'")
└ RPARENTHESIS(')')
└ KEYWORD('selain_itu')
└ <procedure/function-call>
└ IDENTIFIER('writeln')
└ LPARENTHESIS('(')

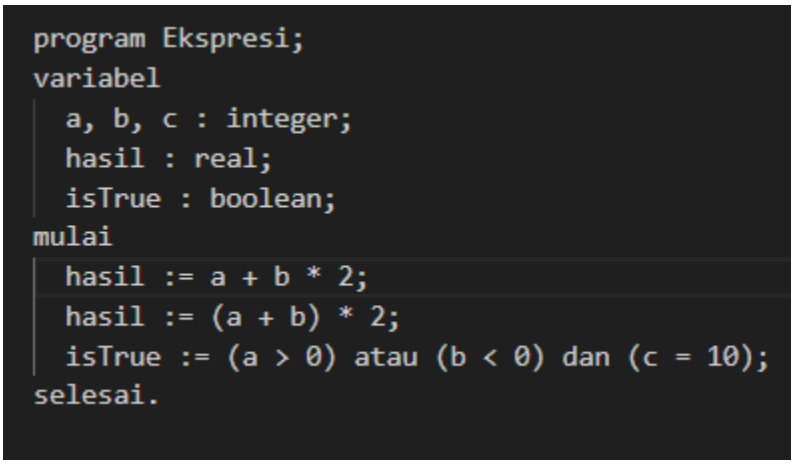
Input



Penjelasan

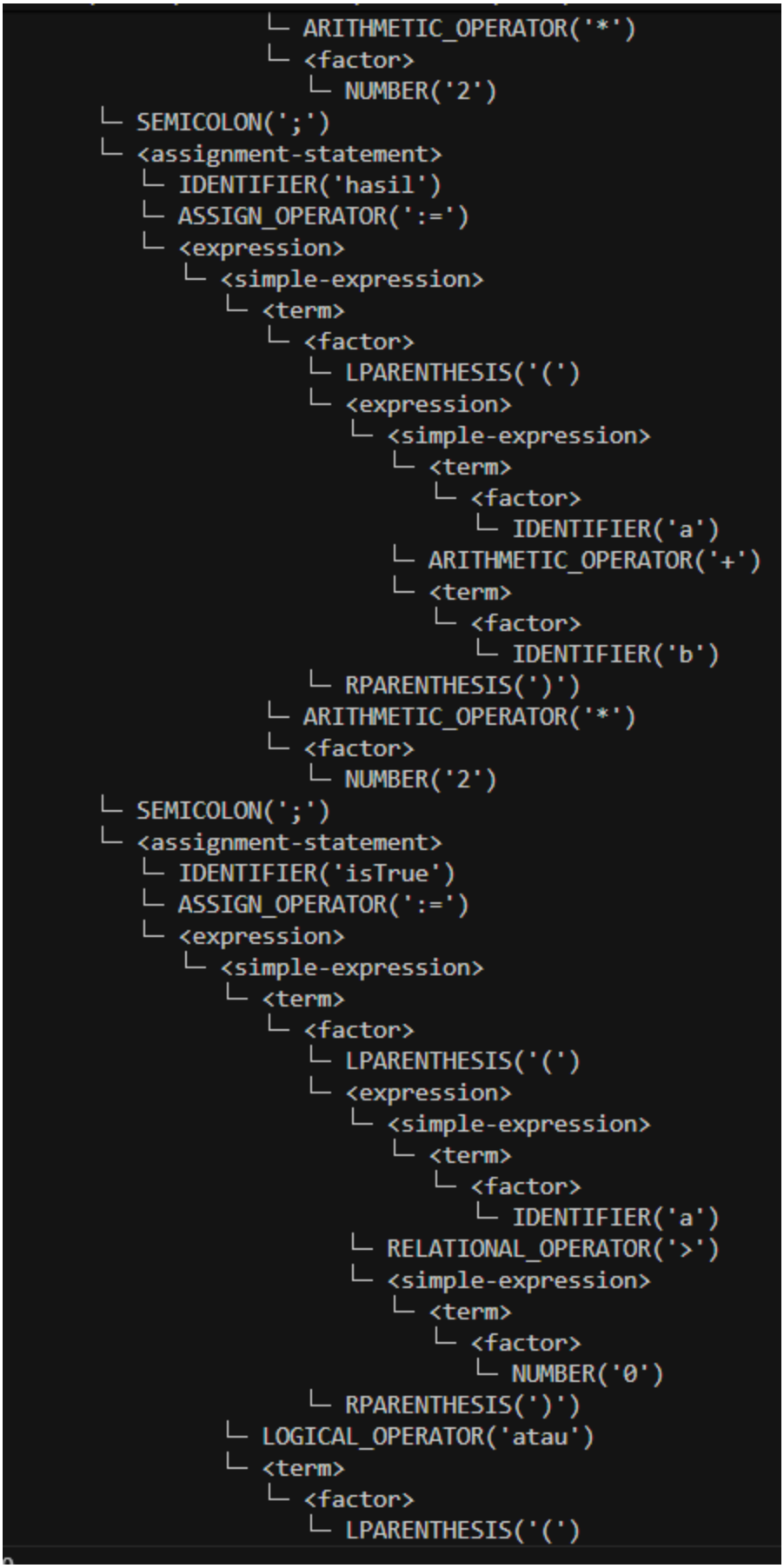
Parser sukses membangun Parse Tree bersarang. Node <for-statement> terlihat berisi <compound-statement>. Node <if-statement> juga berhasil mengidentifikasi dan membuat child untut keyword selain_itu, membuktikan bahwa parser mennagani cabang kondisional secara penuh

3.3. Pengujian 3: Program dengan Ekspresi yang Membutuhkan Urutan Operasi

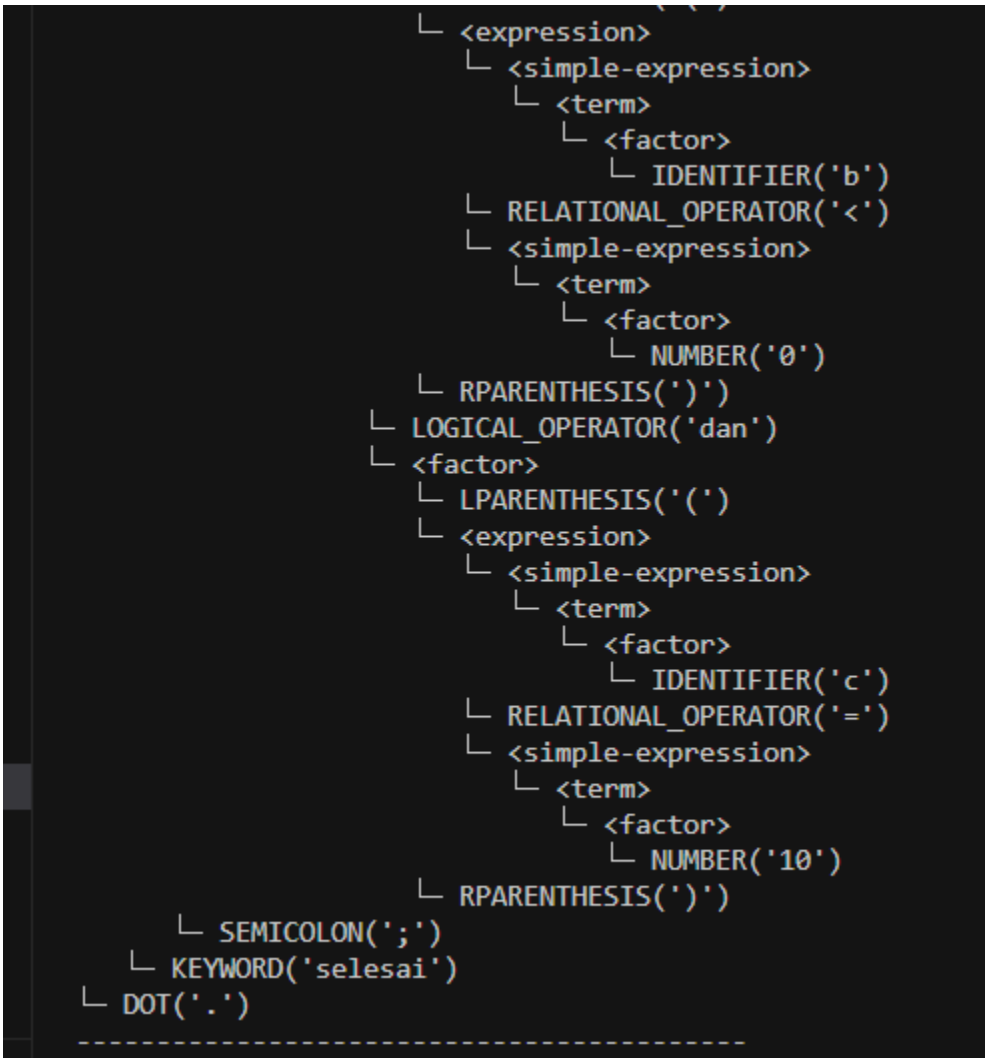
Input

Output

```
program Ekspresi;
variabel
  a, b, c : integer;
  hasil : real;
  isTrue : boolean;
mulai
  hasil := a + b * 2;
  hasil := (a + b) * 2;
  isTrue := (a > 0) atau (b < 0) dan (c = 10);
selesai.
```

```
--------------- PARSE TREE ---------------
<program>
└─ <program-header>
   └─ KEYWORD('program')
   └─ IDENTIFIER('Ekspresi')
   └─ 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('hasil')
      └─ COLON(':')
      └─ <type>
         └─ KEYWORD('real')
      └─ SEMICOLON(';')
      └─ <identifier-list>
         └─ IDENTIFIER('isTrue')
      └─ COLON(':')
      └─ <type>
         └─ KEYWORD('boolean')
      └─ SEMICOLON(';')
└─ <compound-statement>
   └─ KEYWORD('mulai')
   └─ <statement-list>
      └─ <assignment-statement>
         └─ IDENTIFIER('hasil')
         └─ ASSIGN_OPERATOR(':=')
         └─ <expression>
            └─ <simple-expression>
               └─ <term>
                  └─ <factor>
                     └─ IDENTIFIER('a')
               └─ ARITHMETIC_OPERATOR('+')
               └─ <term>
                  └─ <factor>
                     └─ IDENTIFIER('b')
```


└ ARITHMETIC_OPERATOR('*')
└ <factor>
└ NUMBER('2')
└ SEMICOLON(';')
└ <assignment-statement>
└ IDENTIFIER('hasil')
└ 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>
└ NUMBER('2')
└ SEMICOLON(';')
└ <assignment-statement>
└ IDENTIFIER('isTrue')
└ ASSIGN_OPERATOR(':=')
└ <expression>
└ <simple-expression>
└ <term>
└ <factor>
└ LPARENTHESIS('(')
└ <expression>
└ <simple-expression>
└ <term>
└ <factor>
└ IDENTIFIER('a')
└ RELATIONAL_OPERATOR('>')
└ <simple-expression>
└ <term>
└ <factor>
└ NUMBER('0')
└ RPARENTHESIS(')')
└ LOGICAL_OPERATOR('atau')
└ <term>
└ <factor>
└ LPARENTHESIS('(')

Input



Penjelasan

Parser berhasil mengimplementasikan urutan operasi dengan benar. Hal ini terlihat dari hierarki tree. Untuk a + b * 2, node <term> (level prioritas tinggi) mengelompokkan b * 2, yang kemudian digabungkan dengan a oleh <simple-expression> (level prioritas rendah). Ekspresi (a > 0) atau (b < 0) dan (c = 10) diurai dengan benar, di mana dan (multiplikatif) dikelompokkan di bawah atau (aditif), sesuai dengan aturan presedensi.

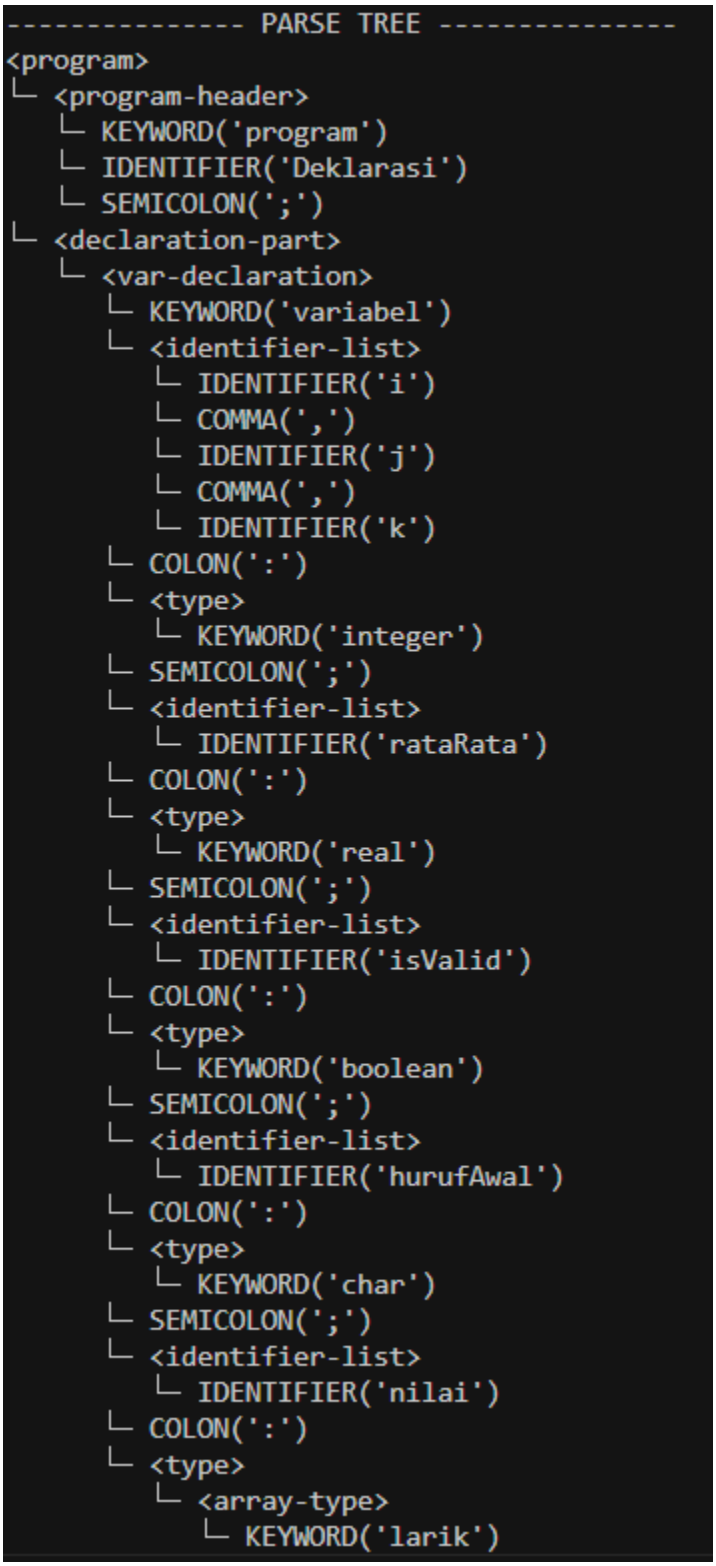
3.4. Pengujian 4: Program dengan Deklarasi Kompleks

Input

```
program Deklarasi;
variabel
  i, j, k : integer;
  rataRata : real;
  isValid : boolean;
  hurufAwal : char;
  nilai : larik [1 .. 100] dari integer;
  matriks : larik [1 .. 100] dari real;

mulai
  i := 1;
  hurufAwal := 'a';
  isValid := tidak (i > 100);
selesai.
```

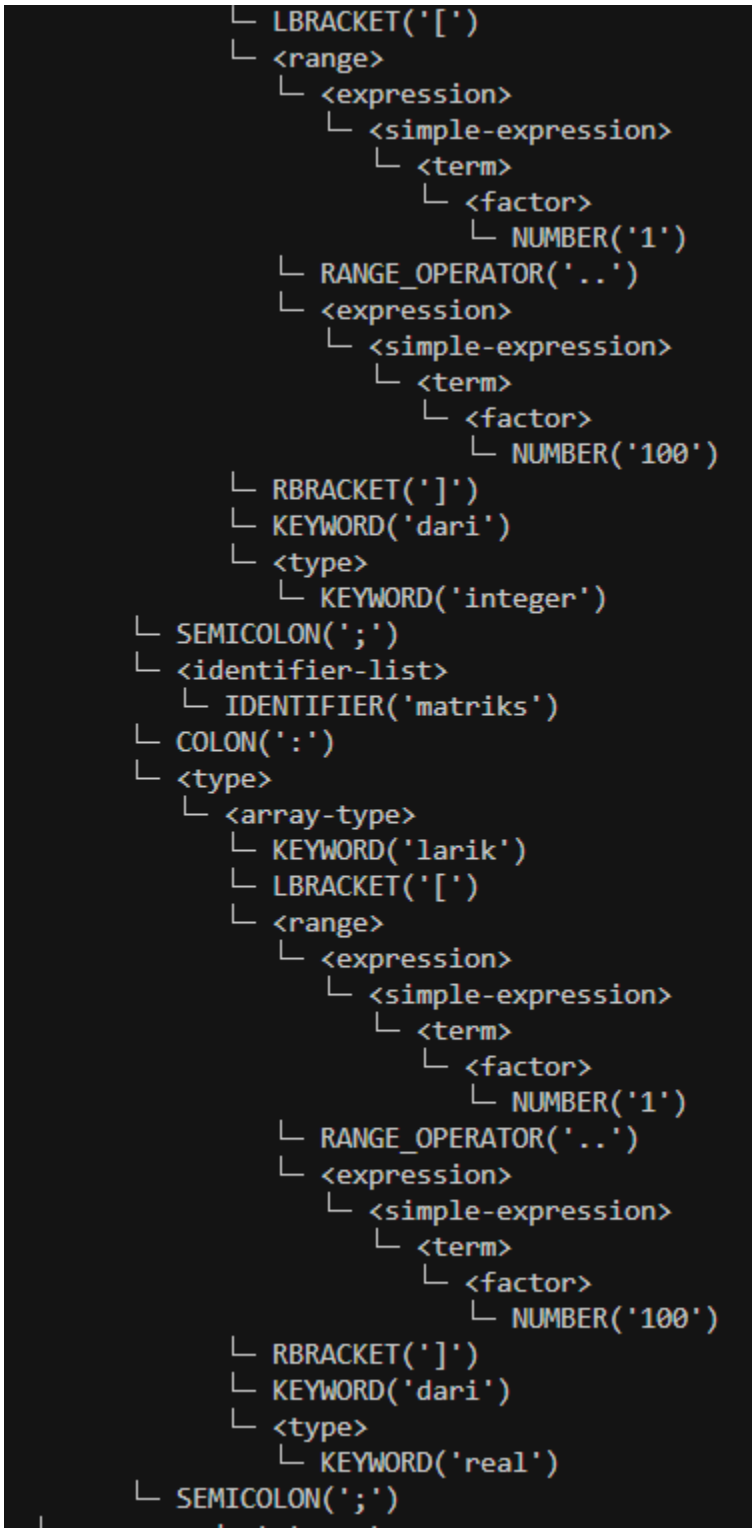
Output

Input



```
--------------- PARSE TREE ---------------
<program>
└─ <program-header>
   └─ KEYWORD('program')
   └─ IDENTIFIER('Deklarasi')
   └─ SEMICOLON(';')
└─ <declaration-part>
   └─ <var-declaration>
      └─ KEYWORD('variabel')
      └─ <identifier-list>
         └─ IDENTIFIER('i')
         └─ COMMA(',')
         └─ IDENTIFIER('j')
         └─ COMMA(',')
         └─ IDENTIFIER('k')
      └─ COLON(':')
      └─ <type>
         └─ KEYWORD('integer')
      └─ SEMICOLON(';')
      └─ <identifier-list>
         └─ IDENTIFIER('rataRata')
      └─ COLON(':')
      └─ <type>
         └─ KEYWORD('real')
      └─ SEMICOLON(';')
      └─ <identifier-list>
         └─ IDENTIFIER('isValid')
      └─ COLON(':')
      └─ <type>
         └─ KEYWORD('boolean')
      └─ SEMICOLON(';')
      └─ <identifier-list>
         └─ IDENTIFIER('hurufAwal')
      └─ COLON(':')
      └─ <type>
         └─ KEYWORD('char')
      └─ SEMICOLON(';')
      └─ <identifier-list>
         └─ IDENTIFIER('nilai')
      └─ COLON(':')
      └─ <type>
         └─ <array-type>
            └─ KEYWORD('larik')
```

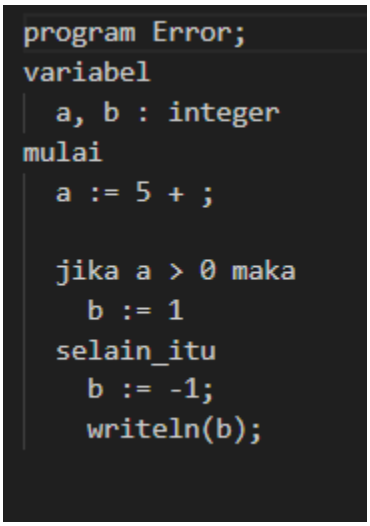
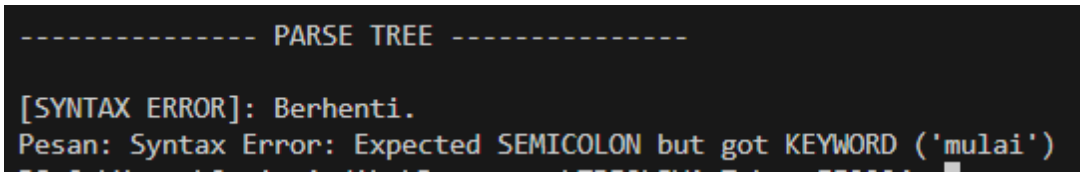
Input


└ LBRACKET('[')
└ <range>
└ <expression>
└ <simple-expression>
└ <term>
└ <factor>
└ NUMBER('1')
└ RANGE_OPERATOR('..')
└ <expression>
└ <simple-expression>
└ <term>
└ <factor>
└ NUMBER('100')
└ RBRACKET(']')
└ KEYWORD('dari')
└ <type>
└ KEYWORD('integer')
└ SEMICOLON(';')
└ <identifier-list>
└ IDENTIFIER('matriks')
└ COLON(':')
└ <type>
└ <array-type>
└ KEYWORD('larik')
└ LBRACKET('[')
└ <range>
└ <expression>
└ <simple-expression>
└ <term>
└ <factor>
└ NUMBER('1')
└ RANGE_OPERATOR('..')
└ <expression>
└ <simple-expression>
└ <term>
└ <factor>
└ NUMBER('100')
└ RBRACKET(']')
└ KEYWORD('dari')
└ <type>
└ KEYWORD('real')
└ SEMICOLON(';')


└ <compound-statement>
└ KEYWORD('mulai')
└ <statement-list>
└ <assignment-statement>
└ IDENTIFIER('i')
└ ASSIGN_OPERATOR(':=')
└ <expression>
└ <simple-expression>
└ <term>
└ <factor>
└ NUMBER('1')
└ SEMICOLON(';')
└ <assignment-statement>
└ IDENTIFIER('hurufAwal')
└ ASSIGN_OPERATOR(':=')
└ <expression>
└ <simple-expression>
└ <term>
└ <factor>
└ STRING_LITERAL("'a'")
└ SEMICOLON(';')
└ <assignment-statement>
└ IDENTIFIER('isValid')
└ ASSIGN_OPERATOR(':=')
└ <expression>
└ <simple-expression>
└ <term>
└ <factor>
└ LOGICAL_OPERATOR('tidak')
└ <factor>
└ LPARENTHESIS('(')
└ <expression>
└ <simple-expression>
└ <term>
└ <factor>
└ IDENTIFIER('i')
└ RELATIONAL_OPERATOR('>')
└ <simple-expression>
└ <term>
└ <factor>
└ NUMBER('100')
└ RPARENTHESIS(')')
└ SEMICOLON(';')
└ KEYWORD('selesai')
└ DOT('.')

Input
Penjelasan
Parser berhasil memvalidasi bagian <declaration-part> yang kompleks, termasuk beberapa deklarasi variabel dan tipe. Node <array-type> dan <range> diurai dengan benar, menunjukkan parser sukses menangani sintaks larik [...] dari .... Bagian <compound-statement> setelahnya juga diurai dengan benar, membuktikan parser dapat melanjutkan setelah blok deklarasi yang rumit.

3.5. Pengujian 5 : Program dengan Syntax Error

Input

Output

Penjelasan

Input
Output tidak menghasilkan Parse Tree, yang merupakan perilaku yang benar untuk input yang gagal. Parser berhasil mendeteksi sintaks yang tidak valid (SEMICOLON yang hilang) dan berhenti. Program mengeluarkan pesan Syntax Error yang informatif, membuktikan bahwa mekanisme error handling parser berfungsi sesuai spesifikasi.

4. Kesimpulan dan saran

4.1. Kesimpulan

Berdasarkan pelaksanaan Milestone 2 tugas besar mata kuliah IF2224 Formal Language and Automata Theory, implementasi syntax analysis untuk compiler Pascal-S telah berhasil dikembangkan. Implementasi ini memanfaatkan algoritma Recursive Descent untuk memvalidasi tata bahasa yang telah dimasukkan ke program.

Proses pengujian yang mencakup lima kasus uji unik (termasuk kasus dasar, alur kontrol, ekspresi kompleks, deklarasi, dan satu kasus error) menunjukkan bahwa parser dapat mengenali berbagai struktur gramatikal. Parser terbukti mampu mengurai struktur program dasar, alur kontrol kompleks, dan penanganan array.

Lebih lanjut, pengujian memvalidasi penanganan urutan operasi untuk ekspresi aritmatika dan logika, serta penggabungan node <procedure/function-call>. Parser juga menunjukkan kemampuan error handling yang baik dengan mendeteksi sintaks tidak valid (misalnya, SEMICOLON yang hilang) dan melaporkan Syntax Error secara informatif. Secara keseluruhan, implementasi parser telah memenuhi tujuan untuk mengubah daftar token dari lexer menjadi Parse Tree hierarkis yang valid, yang siap digunakan untuk tahap analisis semantik berikutnya.

4.2. Saran

Berdasarkan evaluasi implementasi dan hasil pengujian, beberapa saran dapat diajukan untuk perbaikan dan pengembangan lebih lanjut:

1. Optimalisasi Error Recovery: Saat ini, parser berhenti pada syntax error pertama yang ditemui. Disarankan untuk mengembangkan mekanisme error recovery yang lebih canggih, misalnya dengan melakukan sinkronisasi agar parser dapat menemukan dan melaporkan multiple syntax error dalam satu kali proses kompilasi.

5. Referensi

[1] T. Norvell, "Parsing Expressions by Recursive Descent," 1999. [Online]. Available: https://www.engr.mun.ca/~theo/Misc/exp_parsing.htm. [Accessed: November 15th, 2025].

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

[3] Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2007). Introduction to Automata Theory, Languages, and Computation (3rd ed.). Pearson Education.

[4] Institut Teknologi Bandung. Spesifikasi Tugas Besar IF2224 - Formal Language and Automata Theory, Milestone 2. https://docs.google.com/document/d/1G_pC2dltQ5Q-iQ3xOpmLIl3XWLDcLfodN8jdiaJBA0Q/edit?tab=t.0

Lampiran

- Link Release Repository Github.
  https://github.com/orvin14/JWA-Tubes-IF2224
- Aturan Grammar.

No	Nama Node	Deskripsi	Aturan Produksi	Contoh
1	<program>	Node root yang merepresentasikan keseluruhan program Pascal-S	<program> ::= <program-header> <declaration-part> <compound-statement> '.'	Seluruh struktur program dari awal hingga akhir
2	<program-header>	Bagian kepala program yang berisi nama program	<program-header> ::= 'program' IDENTIFIER ';'	program Hello;
3	<declaration-part>	Bagian deklarasi yang dapat berisi const, type, var, procedure, function	<declaration-part> ::= { <const-declaration> } { <type-declaration> } { <var-declaration> } { <subprogram-declaration> }	Semua deklarasi sebelum "mulai"
4	<const-declaration>	Deklarasi konstanta	<const-declaration> ::= 'konstanta' (IDENTIFIER '=' <expression> ';')+	konstanta MAX = 100;
5	<type-declaration>	Deklarasi tipe data baru	<type-declaration> ::= 'tipe' (IDENTIFIER '=' <type> ';')+	tipe Range = 1..10;
6	<var-declaration>	Deklarasi variabel	<var-declaration> ::= 'variabel' (<identifier-list> ':' <type> ';')+	variabel a, b: integer;
7	<identifier-list>	Daftar identifier yang dipisahkan koma	<identifier-list> ::= IDENTIFIER { ',' IDENTIFIER }	a, b, c
8	<type>	Tipe data (integer, real, boolean, char,	<type> ::= 'integer' | 'real' | 'boolean' |	integer, larik[1..10] dari integer

		array)	'char' | <array-type>	
9	<array-type>	Definisi tipe array	<array-type> ::= 'larik' '[' <range> ']' 'dari' <type>	larik[1..10] dari integer
10	<range>	Rentang nilai untuk array atau subrange	<range> ::= <expression> '..' <expression>	1..10, 'a'..'z'
11	<subprogram-declaration>	Deklarasi prosedur atau fungsi	<subprogram-declaration> ::= <procedure-declaration> | <function-declaration>	prosedur print(x: integer);
12	<procedure-declaration>	Deklarasi prosedur	<subprogram-declaration> ::= <procedure-declaration> | <function-declaration>	Prosedur dengan parameter opsional
13	<function-declaration>	Deklarasi fungsi	<function-declaration> ::= 'fungsi' IDENTIFIER [<formal-parameter-list>] ':' <type> ';' <block> ';'	Fungsi yang mengembalikan nilai
14	<formal-parameter-list>	Daftar parameter formal	<formal-parameter-list> ::= '(' <parameter-group> { ';' <parameter-group> } ')'	(x, y: integer; z: real)
15	<compound-statement>	Blok statement yang diawali begin dan diakhiri end	<compound-statement> ::= 'mulai' <statement-list> 'selesai'	mulai ... selesai
16	<statement-list>	Daftar statement yang dipisahkan semicolon	<statement-list> ::= [<statement> { ';' <statement> }]	Urutan statement dalam blok

17	<statement>	Unit instruksi yang melakukan sebuah aksi	<statement> ::= <assignment-statement> | <if-statement> | <while-statement> | <for-statement> | <procedure/function-call> | <compound-statement> | <empty-statement>	A := 5, writeln(‘Hello’)
18	<assignment-statement>	Statement penugasan nilai ke variabel	<assignment-statement> ::= IDENTIFIER ':=' <expression>	x := 5, y := x + 10
19	<if-statement>	Statement kondisional	<if-statement> ::= 'jika' <expression> 'maka' <statement> ['selain_itu' <statement>]	jika x > 0 maka y := 1 selain_itu y := 0
20	<while-statement>	Perulangan dengan kondisi di awal	<while-statement> ::= 'selama' <expression> 'lakukan' <statement>	selama x < 10 lakukan x := x + 1
21	<for-statement>	Perulangan dengan counter	<for-statement> ::= 'untuk' IDENTIFIER ':=' <expression> ('ke' | 'turun_ke') <expression> 'lakukan' <statement>	untuk i := 1 ke 10 lakukan ...
22	<procedure/function-call>	Pemanggilan prosedur atau fungsi	<procedure/function-call> ::= IDENTIFIER ['(' <parameter-list> ')']	writeln('Hello'), print(x, y)
23	<parameter-list>	Daftar parameter aktual saat pemanggilan	<parameter-list> ::= <expression> { ',' <expression> }	'Result = ', b, 100

24	<expression>	Ekspresi yang menghasilkan nilai	<expression> ::= <simple-expression> [<relational-operator> <simple-expression>]	x + y, a > b, 5 * (x + 1)
25	<simple-expression>	Ekspresi tanpa operator relasional	<simple-expression> ::= ['+' | '-'] <term> { <additive-operator> <term> }	x + y - z, 5 * 2
26	<term>	Bagian ekspresi dengan prioritas lebih tinggi	<term> ::= <factor> { <multiplicative-operator> <factor> }	x * y, a bagi b
27	<factor>	Unit terkecil dalam ekspresi	<factor> ::= IDENTIFIER | NUMBER | CHAR_LITERAL | STRING_LITERAL | '(' <expression> ')' | 'tidak' <factor> | <function-call>	x, 5, 'a', (x+y), flag "tidak"
28	<relational-operator>	Operator perbandingan	<relational-operator> ::= '=' | '<>' | '<' | '<=' | '>' | '>='	x = 5, y > 10
29	<additive-operator>	Operator penjumlahan/pengurangan	<additive-operator> ::= '+' | '-' | 'atau'	x + y, a atau b
30	<multiplicative-operator>	Operator perkalian/pembagian	<multiplicative-operator> ::= '*' | '/' | 'bagi' | 'mod' | 'dan'	x * y, a bagi b, p dan q

- Pembagian Tugas (menunjukan persentase kontribusi).

Anggota	Pembagian Tugas

Muhammad Alfansya (13523005)	DFA, laporan
Orvin Andika Ikhsan Abhista (13523017)	Parser, Laporan
Dzaky Aurelia Fawwaz (13523065)	DFA, Testing, Laporan
Ferdin Arsenarendra Purtadi (13523117)	DFA, laporan, implementasi kode