

LAPORAN MILESTONE 3
IF2224 TEORI BAHASA FORMAL DAN OTOMATA
SEMANTIC ANALYSIS



Kelompok 6 DFantastic (DFC)

Mayla Yaffa Ludmilla 13523050

Anella Utari Gunadi 13523078

Muhammad Edo Raduputu Aprima 13523096

Athian Nugraha Muarajuang 13523106

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA
INSTITUT TEKNOLOGI BANDUNG
2025

DAFTAR ISI

DAFTAR ISI	1
1. DESKRIPSI TUGAS	2
2. LANDASAN TEORI	5
2.1. Semantic Analyzer	5
2.2. Attributed Grammar	5
2.3. Symbol Table	5
2.4. Visit Functions	7
3. PERANCANGAN & IMPLEMENTASI	8
3.1. Perancangan Program	8
3.1.1. Parser Output menjadi AST	8
3.1.2. Struktur Tabel Simbol	8
3.1.3. Mekanisme Scope dan Static Chain	9
3.1.4.. Alur Kerja Semantic Analysis	9
3.2. Implementasi	10
3.2.1. Implementasi Symbol Table (symbol_table.py)	10
3.2.2. Implementasi AST (ast.py)	11
3.2.3. Implementasi AST Builder (ast_builder.py)	13
3.2.4. Implementasi Semantic Analyzer (semantic_analyzer.py)	16
4. PENGUJIAN	19
4.1.	19
5. KESIMPULAN DAN SARAN	20
5.1. Kesimpulan	20
5.2. Saran	20
LAMPIRAN	21
REFERENSI	22

1. DESKRIPSI TUGAS

Pada milestone ini, kami diminta untuk membuat tahapan ketiga dari compiler, yaitu semantic analysis. Tahapan ini berfungsi melakukan analisis makna (semantic analysis) dengan menggunakan Attributed Grammar.

Parse tree yang sudah dihasilkan akan ditelusuri secara top-down menggunakan fungsi visit (semantic visitor) pada setiap node-nya yang dibuat berdasarkan grammar yang sebelumnya didefinisikan. Selama proses ini, symbol table digunakan untuk menyimpan dan mengambil informasi deklarasi simbol/identifier.

Pada tahap ini dilakukan hal sebagai berikut:

1. Membentuk Abstract Syntax Tree, termasuk:
 - Memanfaatkan Syntax-Directed Translation Scheme.
 - Identifikasi production rule dan semantic action yang terkait.
 - Menjalankan semantic action pada node parser tree, akan terbentuk node baru untuk AST.
2. Type & Scope Checking, meliputi:
 - Mengunjungi setiap node AST secara Depth First.
 - Push scope baru pada stack symbol table apabila memasuki block kode baru.
 - Memasukan variabel/fungsi pada symbol table untuk setiap deklarasi.
 - Setiap bertemu dengan identifier, lookup pada symbol table.
 - Kalkulasi type data dari node yang dikunjungi
 - Validasi apakah setiap type sesuai semantik, lalu “dekorasi” AST.
 - Hasil dari tahap ini adalah program yang sudah tervalidasi secara semantik serta Decorated AST, dimana setiap node minimal memiliki informasi:
 - a. Tipe ekspresi (lihat pada “Semantic Analysis (ULiège)” bagaimana tipe ekspresi diperoleh)
 - b. Referensi ke symbol table
 - c. Informasi scope

Sebelum memulai proses, inisiasi symbol table terlebih dahulu. Pada Pascal-S terdapat 3 jenis symbol table:

1. **tab**: digunakan untuk menyimpan informasi mengenai identifier termasuk konstanta, variabel, prosedur, atau fungsi.
2. **btab**: digunakan untuk menyimpan informasi blok prosedur dan definisi tipe rekord.
3. **atab**: digunakan untuk menyimpan informasi array seperti tipe index, batasan index, tipe elemen, dan ukuran array.

Dengan memiliki informasi pada symbol table, dapat dipastikan konsistensi tipe data serta memastikan variabel tidak bisa dioperasikan jika belum diinisialisasi dengan nilai.

Input Command (contoh yang digunakan menggunakan Python) Format: python [Compiler] [Kode]
python compiler.py program.pas
Isi program.pas
program Hello; variabel a, b: integer; mulai a := 5; b := a + 10; writeln('Result = ', b); selesai.
Keluaran (output)
tab (hanya sebagian yang relevan): idx id obj type ref nrm lev adr link ----- ... (reserved words 0-28) 29 Hello program 0 0 1 0 0 0 30 a variable 1 0 1 0 0 31 31 b variable 1 0 1 0 0 0 32 writeln procedure ... (predefined) btab: idx last lpar psze vsze ----- 0 31 0 0 2 ← global block (program), vsze=2 (a dan b) 1 0 0 0 0 ← main block (kosong variabel lokal) atab: (kosong karena tidak ada array) Decorated AST (contoh anotasi minimal): ProgramNode(name: 'Hello') ├─ Declarations │ ├─ VarDecl('a') → tab_index:30, type:integer, lev:0 │ └─ VarDecl('b') → tab_index:31, type:integer, lev:0 └─ Block → block_index:1, lev:1

```
└─ Assign('a' := 5) → type:void
  └─ target 'a' → tab_index:30, type:integer
    └─ value 5 → type:integer
└─ Assign('b' := a+10)→ type:void
  └─ target 'b' → tab_index:31, type:integer
    └─ BinOp '+' → type:integer
      └─ 'a' → tab_index:30, type:integer
        └─ 10 → type:integer
└─ writeln(...) → predefined, tab_index:32
```

2. LANDASAN TEORI

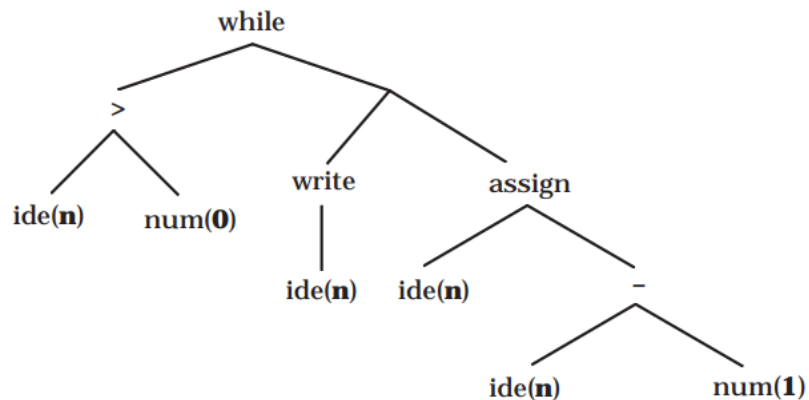
2.1. Semantic Analyzer

Semantic Analyzer merupakan fase ketiga dalam proses kompilasi yang bertujuan untuk memeriksa apakah struktur program yang telah valid secara sintaks juga memiliki makna yang benar. Meskipun sebuah *parser* telah memastikan kode mematuhi aturan produksi *grammar*, *parser* tidak dapat mendeteksi ketidakkonsistenan logika, seperti operasi matematika pada tipe data yang tidak kompatibel, deklarasi variabel berulang kali, dan sebagainya.

Semantic Analyzer bekerja dengan melakukan penelusuran pada *parse tree* untuk memvalidasi konsistensi tipe data, deklarasi variabel, serta *scope* program. Proses ini akan menghasilkan anotasi pada *parse tree*, yaitu *Decorated Abstract Syntax Tree (AST)*, yang berisi informasi tambahan seperti tipe ekspresi dan referensi ke *symbol table*.

2.2. Abstract Syntax Tree (AST)

Abstract Syntax Tree (AST) adalah representasi pohon dari struktur sintaksis kode sumber yang telah disederhanakan. Berbeda dengan *Parse Tree* yang memetakan setiap detail aturan tata bahasa secara literal, termasuk tanda baca seperti titik koma, kurung, dan kata kunci, AST hanya menyimpan elemen-elemen yang penting untuk analisis makna dan eksekusi program.



Gambar 2.1 Ilustrasi *Abstract Syntax Tree*

Sumber: [5]

Perannya dalam Pascal-S, pembentukan AST dilakukan dengan memanfaatkan *Syntax-Directed Translation Scheme (SDTS)*. Saat *parser* mengenali aturan semantik tertentu, *Semantic actions* dijalankan untuk membuat objek *node* AST yang sesuai.

Diantaranya:

- a. Aturan $\text{factor} \rightarrow \text{NUMBER}$ akan menghasilkan `NumberNode` yang menyimpan nilai angka tersebut.
- b. Aturan $\text{expression} \rightarrow \text{term} + \text{term}$ akan menghasilkan `BinOpNode` yang memiliki dua anak (operand kiri dan kanan) serta operator `+`.

AST inilah yang kemudian menjadi subjek utama untuk proses *type checking* dan *annotation*, yang pada akhirnya menghasilkan *Decorated AST*.

2.3. Attributed Grammar

Attributed Grammar adalah perluasan dari *Context-Free Grammar* yang dilengkapi dengan atribut dan aturan semantik untuk setiap simbol *grammar*-nya. Konsep ini memungkinkan *compiler* untuk tidak hanya mengenali struktur bahasa, tetapi juga menghitung dan menyimpan nilai.

Pendekatan *L-Attributed Grammar* memungkinkan informasi semantik diproses secara efisien dalam satu lintasan dari kiri ke kanan. Implementasi teknisnya memanfaatkan *Syntax-Directed Translation Scheme* (SDTS), teknik yang menyisipkan *semantic actions* atau potongan kode langsung di antara simbol-simbol dalam aturan produksi. Ketika parser mengenali suatu pola dan mengunjungi node terkait, aksi ini dieksekusi seketika untuk memvalidasi logika dan membentuk *node* baru pada AST yang lebih kaya informasi.

2.4. Symbol Table

Symbol Table adalah struktur data yang digunakan untuk menyimpan informasi mengenai *identifier*, seperti nama, tipe data, dan scope-nya selama proses analisis semantik. Secara umum, *Symbol Table* menggunakan struktur data *stack* untuk menangani manajemen *scope* yang bersarang.

Identifiers:		link	obj	typ	ref	nrm	lev	adr
29	ten	0	0	1	0	0	1	10
30	plus	29	0	4	0	0	1	37
31	row	30	2	5	1	0	1	10
32	complex	31	2	6	3	0	1	2
33	re	0	1	2	0	1	2	0
34	im	33	1	2	0	1	2	1
35	i	32	1	1	0	1	1	5
36	j	35	1	1	0	1	1	6
37	p	36	1	3	0	1	1	7
38	z	37	1	6	3	1	1	8
39	matrix	38	1	5	2	1	1	10
40	pattern	39	1	5	3	1	1	80
41	dummy	40	3	0	4	1	1	16
42	i	0	1	1	0	0	2	5
43	z	42	1	6	3	0	2	6
44	u	43	1	5	1	1	2	7
45	v	44	1	5	1	1	2	17
46	h1	45	1	6	5	1	2	27
47	h2	46	1	6	5	1	2	39
48	c	0	1	6	3	1	3	0
49	r	48	1	5	1	1	3	2
50	null	47	4	3	6	1	2	0
51	x	0	1	2	0	1	3	5
52	Y	51	1	2	0	1	3	6
53	z	52	1	6	3	1	3	7
54	a	53	1	5	5	1	3	9
55	u	54	1	4	0	1	3	61

Blocks:		last	1par	psze	vsze
1	28	1	0	0	
2	41	28	5	105	
3	34	0	0	2	
4	50	43	7	51	
5	49	0	0	12	
6	55	53	9	62	

Arrays:		xtyp	etyp	eref	low	high	elsz	size
1	1	2	0	1	10	1	10	
2	1	5	1	-3	3	10	70	
3	1	5	4	1	5	5	25	
4	1	4	0	1	5	1	5	
5	4	6	3	1	26	2	52	

Gambar 2.2 Contoh *Symbol Table*

Sumber: [2]

Khusus pada Pascal-S, tabel simbol dibagi menjadi tiga struktur terpisah:

- Tabel Identifier (*tab*): Menyimpan seluruh *identifier* seperti konstanta, variabel, prosedur, dan fungsi. Setiap entri memiliki atribut detail seperti:
 - link*: *Pointer* ke *identifier* sebelumnya dalam *scope* yang sama (membentuk linked list).
 - obj*: Jenis objek (misal: variabel, prosedur).
 - typ*: Tipe data dasar.
 - lev*: Level leksikal. 0 untuk global, 1 untuk lokal, dst.
 - adr*: Alamat relatif atau *offset* memori.

- b. Tabel Blok (*btab*): Menyimpan informasi struktural untuk prosedur, fungsi, atau definisi rekaman. Atribut kuncinya meliputi *last* (*pointer* ke *identifier* terakhir di blok tersebut) dan *vsze* (ukuran total variabel lokal dalam byte).
- c. Tabel Array (*atab*): Menyimpan spesifikasi tipe data array. Atribut kuncinya meliputi *xtyp* (tipe indeks), *etyp* (tipe elemen), *low* (batas bawah indeks), serta *high* (batas atas indeks).

2.5. Visit Functions

Visit Functions adalah implementasi dari pola desain Visitor yang digunakan untuk menelusuri AST. Teknik ini memisahkan logika analisis dari struktur data node. Setiap tipe node dalam AST memiliki fungsi penanganan spesifik, misalnya *visit_VarDecl* untuk deklarasi variabel atau *visit_Assign* untuk operasi assignment.

Fungsi-fungsi ini bekerja secara rekursif dengan metode Depth First Search (DFS). Setiap pemanggilan fungsi *visit* akan melakukan empat tugas utama secara berurutan:

1. Traversal Rekursif: Fungsi pertama-tama mengunjungi *child nodes* untuk mengumpulkan informasi yang diperlukan, seperti tipe data operand, sebelum memproses node induk.
2. Manajemen Scope: Melakukan operasi manajemen tabel simbol, seperti *push scope* baru saat menemui deklarasi prosedur dan *pop* saat selesai, memastikan variabel lokal terjaga.
3. Validasi: Inti dari analisis semantik, di mana aturan logika divalidasi.
4. Anotasi Node: Menyimpan hasil kalkulasi dan referensi tabel simbol kembali ke dalam *node AST*. Hasil akhirnya adalah *Decorated AST* yang sepenuhnya tervalidasi dan siap diterjemahkan menjadi kode mesin.

3. PERANCANGAN & IMPLEMENTASI

3.1. Perancangan Program

Pada milestone ini, kami menambahkan lapisan semantic analysis di atas parser dan AST. Arsitektur program dirancang menjadi empat bagian utama, antara lain sebagai berikut.

3.1.1. Parser Output menjadi AST

Parser pada milestone 2 menghasilkan parse tree berbasis aturan grammar. Di tahap ini, class ASTBuilder digunakan untuk menyederhanakan parse tree menjadi AST yang lebih struktural dan memodelkan elemen bahasa menjadi node-node yang merepresentasikan deklarasi, ekspresi, statement, tipe, dan subprogram.

3.1.2. Struktur Tabel Simbol

Sistem tabel simbol yang digunakan terdiri dari tiga tabel utama.

a. TAB (Identifier Table)

Menyimpan seluruh identifier dengan atribut:

- ident: nama dari identifier
- obj: menyatakan kategori identifier
- typ: menyimpan tipe dasar identifier
- ref: menyimpan indeks ke ATAB untuk identifier array
- nrm: menandakan apakah parameter by-value atau by-reference
- lev: scope level dari identifier
- adr: pemakaian tergantung jenis identifier (constant, variable, atau function)

Berikut reserved words untuk identifier dalam TAB yang digunakan:

```
enter('      ', variable, notyp, 0); (* sentinel *)
enter('false  ', konstant, bools, 0);
enter('true   ', konstant, bools, 1);
enter('real   ', typel, reals, 1);
enter('char   ', typel, chars, 1);
enter('boolean', typel, bools, 1);
enter('integer', typel, ints, 1);
enter('abs    ', funktion, reals, 0);
enter('sqr    ', funktion, reals, 2);
enter('odd    ', funktion, bools, 4);
enter('chr    ', funktion, chars, 5);
enter('ord    ', funktion, ints, 6);
enter('succ   ', funktion, chars, 7);
enter('pred   ', funktion, chars, 8);
enter('round  ', funktion, ints, 9);
enter('trunc  ', funktion, ints, 10);
enter('sin    ', funktion, reals, 11);
enter('cos    ', funktion, reals, 12);
enter('exp    ', funktion, reals, 13);
enter('ln     ', funktion, reals, 14);
enter('sqrt   ', funktion, reals, 15);
enter('arctan ', funktion, reals, 16);
enter('eof    ', funktion, bools, 17);
enter('eoln   ', funktion, bools, 18);
enter('read   ', prozedure, notyp, 1);
enter('readln ', prozedure, notyp, 2);
enter('write  ', prozedure, notyp, 3);
enter('writeln', prozedure, notyp, 4);
enter('      ', prozedure, notyp, 0);
```

Gambar 3.1 *Reserved Words* yang dimasukkan ke dalam TAB

Sumber: [2]

b. BTAB (Block Table)

Menyimpan informasi tiap block (global, procedure, function):

- last: pointer ke identifier terakhir di scope ini
- lpar: last parameter index
- psze: parameter size
- vsze: variabel size

c. ATAB (Array Table)

Menyimpan definisi array:

- xtyp: selalu ARRAYS karena ATAB hanya menyimpan tipe array.
- etyp: tipe elemen array (real atau int)
- eref: jika elemen merupakan NamedType, kolom berisi reference index ke TAB
- low, high: range array
- elsz: ukuran tipe elemen
- size: total ukuran array

3.1.3. Mekanisme Scope dan Static Chain

Mekanisme scope dan static chain dikelola langsung melalui modul `symbol_table.py`, terutama melalui variabel `level`, array `display`, serta struktur entri pada TAB dan BTAB. Setiap kali analyzer memasuki sebuah block baru seperti procedure atau function, method `begin_block()` dipanggil sehingga level scope meningkat, BTAB baru dibuat, dan `display[level]` diperbarui untuk menyimpan indeks deklarasi terakhir pada level tersebut. Setiap deklarasi identifier yang dimasukkan ke TAB akan disimpan bersama nilai `lev` yang menunjukkan scope tempat deklarasi dibuat, dan kolom `link` akan menunjuk ke entri sebelumnya dalam scope yang sama (chaining).

3.1.4.. Alur Kerja Semantic Analysis

Alur kerja untuk program semantic analysis ini adalah sebagai berikut.

- Parser membaca file `.pas` dan menghasilkan parse tree sesuai grammar.
- ASTBuilder mengonversi parse tree menjadi AST yang telah disederhanakan ke bentuk node-node.
- SymbolTable diinisiasi dengan identifier bawaan (tipe dasar, konstanta, fungsi/prosedur)
- SemanticAnalyzer melalui traversal AST secara rekursif.
- Saat menemukan deklarasi, analyzer memasukkan entri baru ke TAB/BTAB/ATAB sesuai jenisnya.
- Saat memasuki prosedur atau fungsi, analyzer membuka scope baru (`begin_block()`), meningkatkan level scope, dan menyiapkan block baru dalam BTAB.
- Untuk fungsi, analyzer juga mendaftarkan implicit variable yang memiliki nama sama dengan fungsi sebagai tempat menyimpan nilai return.
- Pada VarDecl, analyzer mengevaluasi literalnya dan mengisi `typ` serta `adr` di TAB.
- Pada TypeDecl, analyzer mengisi entri pada ATAB dan memasang referensi tipe pada TAB.
- Saat emnjumpai VarRef, analyzer melakukan lookup identifier dengan static chain dimulai dari scope terdalam hingga global.
- Pada BinOp dan UnaryOp, analyzer melakukan pengecekan tipe operand, menentukan compatibility, dan menetapkan tipe hasil ekspresi.
- Pada AssignStmt, analyzer bahwa jenis operand kiri adalah variable/function result dan tipe ekspresi kanan cocok dengan tipe target.
- Pada pemanggilan prosedur atau fungsi, analyzer mengecek bahwa identifier yang dipanggil benar-benar prosedur/fungsi dan argumen memiliki tipe yang sesuai.
- Pada struktur kontrol (if, while, for), analyzer memastikan ekspresi syarat bertipe boolean, dan variabel loop bertipe integer.
- Setelah traversal block fungsi/prosedur selesai, scope ditutup (`end_block()`) sehingga semua deklarasi lokal tidak lagi dapat diakses.

- Seluruh node AST diperkaya dengan annotation sehingga AST menjadi Decorated AST.
- Tabel-tabel simbol dicetak sebagai output akhir.

3.2. Implementasi

Berikut merupakan dokumentasi hasil implementasi semantic analysis. Implementasi dibagi menjadi 4 modul utama.

3.2.1. Implementasi Symbol Table (symbol_table.py)

Modul ini mengimplementasikan tiga struktur utama: TAB, BTAB, dan ATAB.

```

symbol_table.py

# ===== TAB (identifier table) =====
@dataclass
class TabEntry:
    ident: str
    link: int
    obj: ObjectKind
    typ: TypeKind
    ref: int
    nrm: bool
    lev: int
    adr: int

# ===== BTAB (block table) =====
@dataclass
class BTabEntry:
    last: int = 0    # last identifier
    lpar: int = 0    # last parameter index
    psze: int = 0    # parameter Size
    vsze: int = 0    # variable Size

# ===== ATAB (array type table) =====
@dataclass
class ATabEntry:
    xtyp: TypeKind
    etyp: TypeKind
    eref: int
    low: int
    high: int
    elsz: int
    size: int

```

3.2.2. Implementasi AST (ast.py)

Modul ini mendefinisikan struktur AST yang digunakan untuk merepresentasikan program Pascal-S dalam bentuk pohon abstrak. Setiap elemen sintaks seperti deklarasi, ekspresi, dan statement, diubah menjadi node AST sesuai jenisnya. Node-node ini nanti akan diperkaya oleh semantic analyzer dengan informasi tipe, scope level, serta indeks tabel simbol. Desain AST dibuat modular melalui kelas terpisah untuk deklarasi, subprogram, statement, dan ekspresi.

ast.py (sebagian kode)

```
@dataclass
class VarDecl (ASTNode):
    names: list[str] = field(default_factory=list)
    type_expr: TypeExpr | None = None

@dataclass
class ProcedureDecl (SubprogramDecl):
    name: str = ""
    params: list[Param] = field(default_factory=list)
    block: Block | None = None

@dataclass
class FunctionDecl (SubprogramDecl):
    name: str = ""
    params: list[Param] = field(default_factory=list)
    return_type: TypeExpr | None = None
    block: Block | None = None

@dataclass
class CompoundStmt (Statement):
    statements: list[Statement] = field(default_factory=list)

@dataclass
class AssignStmt (Statement):
    target: "VarRef | ArrayAccess | None" = None
    value: Expression | None = None

@dataclass
class IfStmt (Statement):
    condition: Expression | None = None
    then_branch: Statement | None = None
    else_branch: Statement | None = None

@dataclass
class WhileStmt (Statement):
    condition: Expression | None = None
    body: Statement | None = None
```

```

@dataclass
class ForStmt(Statement):
    var: VarRef | None = None
    start: Expression | None = None
    end: Expression | None = None
    direction: ForDirection = ForDirection.TO
    body: Statement | None = None

@dataclass
class ProcCallStmt(Statement):
    name: str = ""
    args: list[Expression] = field(default_factory=list)

@dataclass
class CallExpr(Expression):
    name: str = ""
    args: list[Expression] = field(default_factory=list)

@dataclass
class BinOp(Expression):
    op: str = ""
    left: Expression | None = None
    right: Expression | None = None

@dataclass
class UnaryOp(Expression):
    op: str = ""
    operand: Expression | None = None

@dataclass
class VarRef(Expression):
    name: str = ""
    symbol: int = 0 # index di symbol table

@dataclass
class ArrayAccess(Expression):
    """Array element access: arr[index]"""
    array: VarRef | None = None
    index: Expression | None = None

@dataclass
class NumberLiteral(Expression):
    value: str = ""
    evaluated_value: int | float | None = None
    def __post_init__(self):
        try :
            if '.' in self.value:
                self.evaluated_value = float(self.value)
            else :

```

```

        self.evaluated_value = int(self.value)
    except ValueError :
        self.evaluated_value = None

```

3.2.3. Implementasi AST Builder (ast_builder.py)

Modul ini bertanggung jawab mengonversi parse tree menjadi AST yang lebih terstruktur. ASTBuilder membaca node parse tree dan memetakan setiap produksi grammar ke node AST yang sesuai. Modul ini juga menangani bentuk grammar yang ambigu atau tidak eksplisit, seperti penanganan procedure/function call. Hasil akhir dari ASTBuilder adalah AST yang siap diproses semantic analyzer.

ast_builder.py (sebagian kode)

```

def _build_program(self, node: Node) -> Program:
    program_name = ""
    program_token = None
    for child in node.children:
        if child.label == "<program-header>":
            for header_child in child.children:
                if header_child.label == "IDENTIFIER" and
header_child.token:
                    program_name =
header_child.token.value
                    program_token = header_child.token
                    break
            break
    block = self._build_block(node)
    return Program(name=program_name, block=block,
token=program_token)

def _build_declaration_part(self, node: Node, block: Block) ->
None:
    """Populate a Block with declarations found under
<declaration-part>."""
    for child in node.children:
        match child.label:
            case "<const-declaration>":
block.const_decls.extend(self._build_const_declaration(child))
            case "<type-declaration>":
block.type_decls.extend(self._build_type_declaration(child))
            case "<var-declaration>":
block.var_decls.extend(self._build_var_declaration(child))
            case "<procedure-declaration>":
block.subprogram_decls.append(self._build_procedure_declaration(child))
            case "<function-declaration>":

```

```

block.subprogram_decls.append(self._build_function_declaration(child))

def _build_statement(self, node: Node) -> Statement:
    if not node.children:
        raise NotImplementedError("Empty statement node")

    if node.label == "<assignment-statement>":
        return self._build_assign_statement(node)

    elif node.label == "<procedure-function-call>": # Sesuaikan
label dari parser M2
        return self._build_proc_call_stmt(node)

    elif node.label == "<if-statement>":
        return self._build_if_statement(node)

    elif node.label == "<while-statement>":
        return self._build_while_statement(node)

    elif node.label == "<for-statement>":
        return self._build_for_statement(node)

    elif node.label == "<compound-statement>":
        return self._build_compound_statement(node)

    first_child = node.children[0]
    if node.children:
        if first_child.label == "KEYWORD" and
first_child.token:
            kw = first_child.token.value.lower()
            match kw:
                case "mulai":
                    return
self._build_compound_statement(node)
                case "jika":
                    return
self._build_if_statement(node)
                case "selama":
                    return
self._build_while_statement(node)
                case "untuk":
                    return
self._build_for_statement(node)

            elif first_child.label == "<assignment-statement>":
                return
self._build_assign_statement(first_child)

            elif first_child.label ==
"<procedure-function-call>":
                return self._build_proc_call_stmt(first_child)
            raise NotImplementedError(f"Unknown statement type:
{first_child.label}")

    def _build_term(self, node: Node) -> Expression:
        children = node.children

```

```

        if not children:
            raise NotImplementedError("empty term")

        if children[0].label == "<factor>":
            left = self._build_factor(children[0])
            i = 1

            while i < len(children):
                if children[i].label !=
"<multiplicative-operator>":
                    break
                op_node = children[i]
                op = op_node.children[0].token.value if
op_node.children else "*"
                i += 1
                right = self._build_factor(children[i])
                left = BinOp(op=op, left=left, right=right,
token=op_node.token)
                i += 1
            return left

        first = children[0]

        if first.label in ["NUMBER", "STRING_LITERAL",
"CHAR_LITERAL", "BOOLEAN_LITERAL", "IDENTIFIER",
"<procedure-function-call>"]:
            # treat whole <term> as a factor
            fake_factor = Node("<factor>")
            fake_factor.children = [first]
            return self._build_factor(fake_factor)

        for c in children:
            if c.label == "<factor>":
                return self._build_factor(c)

        raise NotImplementedError("term without factor (parser
shape not matched)")

    def _build_call_expr(self, node: Node) -> CallExpr:
        """
        Build CallExpr from <procedure-function-call>
        """
        name = ""
        args: list[Expression] = []
        ident_token = None

        for child in node.children:
            if child.label == "IDENTIFIER" and child.token:
                name = child.token.value
                ident_token = child.token
            elif child.label == "<parameter-list>":
                for param_child in child.children:
                    if param_child.label == "<expression>":
                        try:
                            arg_expr =
self._build_expression(param_child)

```

```

        args.append(arg_expr)
    except NotImplementedError:
        pass

    return CallExpr(name=name, args=args, token=ident_token)

```

3.2.4. Implementasi Semantic Analyzer (semantic_analyzer.py)

SemanticAnalyzer melakukan traversal rekursif terhadap AST dan memperkaya node-node dengan simbol, tipe, dan informasi scope. Setiap deklarasi akan dimasukkan ke TAB/ATAB/BTAB sesuai kategori identifier. Modul ini membuka dan menutup scope ketika memasuki atau keluar dari procedure dan function. Pengecekan tipe dilakukan pada setiap ekspresi, operator, assignment, dan struktur kontrol. Untuk fungsi, analyzer menangani implicit variable untuk menyimpan nilai return. Analyzer juga memastikan pemanggilan prosedur/fungsi valid serta parameter memiliki tipe yang sesuai. Hasil dari modul ini adalah decorated AST dan tabel simbol lengkap.

Semantic_analyzer.py (sebagian kode)

```

# ===== PROGRAM =====
def visit_Program(self, node: Program):
    if self._program_visited:
        return
    self._program_visited = True

    node.scope_level = self.symtab.level

    self.visit(node.block)

# ===== BLOCK =====
def visit_Block(self, node: Block):
    # Constants
    for c in node.const_decls:
        self.visit(c)

    # Types
    for t in node.type_decls:
        self.visit(t)

    # Variables
    for v in node.var_decls:
        self.visit(v)

    # Subprograms
    for s in node.subprogram_decls:
        self.visit(s)

    # Body
    if node.body:
        self.visit(node.body)

# ===== DECLARATIONS =====

```

```

def visit_VarDecl(self, node: VarDecl):
    var_type: TypeKind = TypeKind.NOTYP

    if isinstance(node.type_expr, PrimitiveType):
        nm = node.type_expr.name.lower()
        if nm == "integer":
            var_type = TypeKind.INTS
        elif nm == "real":
            var_type = TypeKind.REALS
        elif nm == "boolean":
            var_type = TypeKind.BOOLS
        elif nm == "char":
            var_type = TypeKind.CHARS
    elif isinstance(node.type_expr, NamedType):
        ref_idx = self.symtab.lookup(node.type_expr.name)
        var_type = TypeKind.NOTYP
    else:
        var_type = TypeKind.NOTYP

    for name in node.names:
        idx = self.symtab.insert(name, "variable", 0)
        entry = self.symtab.tab[idx]
        entry.adr = self.symtab.dx

        if isinstance(node.type_expr, ArrayType):
            aref = self._build_array_type(node.type_expr)
            entry.typ = TypeKind.ARRAYS
            entry.ref = aref
            self.symtab.dx +=
self.symtab.get_variable_size(TypeKind.ARRAYS, aref)
        else:
            entry.typ = var_type
            self.symtab.dx +=
self.symtab.get_variable_size(var_type)

        node.symbol = idx
        node.scope_level = self.symtab.level

# ===== STATEMENTS =====
def visit_CompoundStmt(self, node: CompoundStmt):
    for stmt in node.statements:
        self.visit(stmt)

def visit_AssignStmt(self, node: AssignStmt):
    var_name = node.target.name
    var_idx = self.symtab.lookup(var_name)

    if var_idx is None:
        raise SemanticError(f"Variable '{var_name}' not declared.")

    var_entry = self.symtab.tab[var_idx]

    if var_entry.obj not in (ObjectKind.VARIABLE,
ObjectKind.FUNCTION):
        raise SemanticError(
            f"Cannot assign to '{var_name}' because it is a

```

```

{var_entry.objj}.",
)

    expr_type = self.visit(node.value)

    if expr_type and var_entry.typ != expr_type:
        raise SemanticError(f"Type mismatch in assignment. Cannot
assign {expr_type} to variable '{var_name}' of type {var_entry.typ}.")

# ===== EXPRESSIONS =====
def visit_BinOp(self, node: BinOp):
    if node.left:
        left_type = self.visit(node.left)

    if node.right:
        right_type = self.visit(node.right)

    op = node.op

    if op in ['+', '-', '*', '/'] :
        is_real_op = (left_type == TypeKind.REALS or right_type ==
TypeKind.REALS or op == '/')

        if left_type not in (TypeKind.INTS, TypeKind.REALS) or
right_type not in (TypeKind.INTS, TypeKind.REALS):
            raise SemanticError(f"Operator '{op}' memerlukan operand
numerik")

        result = TypeKind.REALS if is_real_op else TypeKind.INTS
        node.type = result
        return result

    elif op in ['bagi', 'mod']:
        if left_type != TypeKind.INTS or right_type !=
TypeKind.INTS:
            raise SemanticError(f"Operator '{op}' hanya berlaku
untuk Integer")
        node.type = TypeKind.INTS
        return TypeKind.INTS

    elif op in ['dan', 'atau'] :
        if left_type != TypeKind.BOOLS or right_type !=
TypeKind.BOOLS:
            raise SemanticError(f"Operator '{op}' memerlukan operand
Boolean")
        node.type = TypeKind.BOOLS
        return TypeKind.BOOLS

    elif op in ['=', '<', '>', '<=', '>=', '<>', '!='] :
        if left_type != right_type:
            if {left_type, right_type} == {TypeKind.INTS,
TypeKind.REALS}:
                pass
            else:
                raise SemanticError(f"Tipe operand tidak cocok untuk
perbandingan '{op}'")

```

```
node.type = TypeKind.BOOLS
return TypeKind.BOOLS
```

4. PENGUJIAN

4.1. Pengujian 1

Tujuan pengujian: Untuk memvalidasi kemampuan Semantic Analyzer dalam memproses deklarasi variabel dan ekspresi aritmatika serta logika dasar. Pengujian ini memastikan bahwa setiap identifier (variabel a, b, c, d, e) dimasukkan dengan benar ke dalam Symbol Table dengan tipe data yang sesuai (integer atau boolean), dan memastikan bahwa operasi biner (+, -, *, bagi, mod, >, and, or) menghasilkan tipe data yang valid serta dianotasi dengan benar pada Abstract Syntax Tree (AST).

File: 1-expression.pas

Input								
<pre> program CekEkspresi; variabel a, b, c: integer; d, e: boolean; mulai { Tes Aritmatika } a := 10 + 5 - 2 * 3; b := 100 bagi 10 mod 3; { Tes Relasional & Logika } jika (a > b) dan (b <= c) maka d := tidak e selain_itu e := (a <> c) atau (c = 10); selesai. </pre>								
Output								
<pre> Semantic Analysis Successful. ===== SYMBOL TABLES ===== TAB (identifier table): </pre>								
idx	id	obj	typ	ref	nrm	lev	adr	link
0		VARIABLE	NOTYP	0	1	0	0	0
1	false	CONSTANT	BOOLS	0	1	0	0	0
2	true	CONSTANT	BOOLS	0	1	0	1	1
3	real	TYPE	REALS	0	1	0	1	2
4	char	TYPE	CHARS	0	1	0	1	3
5	boolean	TYPE	BOOLS	0	1	0	1	4
6	integer	TYPE	INTS	0	1	0	1	5
7	abs	FUNCTION	REALS	0	1	0	0	6
8	sqr	FUNCTION	REALS	0	1	0	2	7
9	odd	FUNCTION	BOOLS	0	1	0	4	8
10	chr	FUNCTION	CHARS	0	1	0	5	9
11	ord	FUNCTION	INTS	0	1	0	6	10

12		succ		FUNCTION		CHARS		0		1		0		7		11
13		pred		FUNCTION		CHARS		0		1		0		8		12
14		round		FUNCTION		INTS		0		1		0		9		13
15		trunc		FUNCTION		INTS		0		1		0		10		14
16		sin		FUNCTION		REALS		0		1		0		11		15
17		cos		FUNCTION		REALS		0		1		0		12		16
18		exp		FUNCTION		REALS		0		1		0		13		17
19		ln		FUNCTION		REALS		0		1		0		14		18
20		sqrt		FUNCTION		REALS		0		1		0		15		19
21		arctan		FUNCTION		REALS		0		1		0		16		20
22		eof		FUNCTION		BOOLS		0		1		0		17		21
23		eoln		FUNCTION		BOOLS		0		1		0		18		22
24		read		PROCEDURE		NOTYP		0		1		0		1		23
25		readln		PROCEDURE		NOTYP		0		1		0		2		24
26		write		PROCEDURE		NOTYP		0		1		0		3		25
27		writeln		PROCEDURE		NOTYP		0		1		0		4		26
28				PROCEDURE		NOTYP		0		1		0		0		27
29		a		VARIABLE		INTS		0		1		0		0		28
30		b		VARIABLE		INTS		0		1		0		1		29
31		c		VARIABLE		INTS		0		1		0		2		30
32		d		VARIABLE		BOOLS		0		1		0		3		31
33		e		VARIABLE		BOOLS		0		1		0		4		32

BTAB (block table):

idx		last		lpar		psze		vsze

0		34		0		0		0

ATAB (array table):

idx		xtyp		etyp		eref		low		high		elsz		size

===== DECORATED AST =====

```

└─ Program [name=CekEkspres]
  └─ Block
    └─ VarDecl [symbol=32]
      └─ PrimitiveType [name=integer]
    └─ VarDecl [symbol=34]
      └─ PrimitiveType [name=boolean]
    └─ CompoundStmt
      └─ AssignStmt
        └─ VarRef [name=a]
      └─ AssignStmt
        └─ VarRef [name=b]
      └─ IfStmt
        └─ BinOp [type=bools]
          └─ BinOp [type=bools]
            └─ VarRef [name=a, type=ints, symbol=30]
            └─ VarRef [name=b, type=ints, symbol=31]
          └─ BinOp [type=bools]
            └─ VarRef [name=b, type=ints, symbol=31]
            └─ VarRef [name=c, type=ints, symbol=32]
        └─ AssignStmt
          └─ VarRef [name=d]
        └─ AssignStmt
          └─ VarRef [name=e]

```

4.2. Pengujian 2

Tujuan pengujian: Untuk memvalidasi mekanisme *Type Checking* pada assignment statement. Pengujian ini bertujuan memastikan bahwa Semantic Analyzer dapat mendeteksi dan mencegah pemberian nilai (assignment) dengan tipe data yang tidak kompatibel ke suatu variabel (contoh: variabel bertipe integer tidak boleh diisi dengan nilai bertipe boolean), sehingga menjamin konsistensi tipe data dalam program.

File: 2-type.pas

Input
<pre>program CekTipe; variabel a: integer; flag: boolean; mulai a := 10; flag := (a = 10); a := flag; { integer - boolean } selesai.</pre>
Output
<pre>===== COMPILATION FAILED: SEMANTIC ERROR ===== Message : [SemanticError] Type mismatch in assignment. Cannot assign bools to variable 'a' of type ints. =====</pre>

4.3. Pengujian 3

Tujuan pengujian: Untuk memvalidasi kemampuan Semantic Analyzer dalam menangani deklarasi dan pemanggilan subprogram (prosedur). Pengujian ini berfokus pada manajemen scope (lingkup), memastikan bahwa parameter formal (x, s) dan variabel lokal disimpan dalam Symbol Table pada level leksikal yang benar (lev=1) dan terpisah dari variabel global (lev=0). Selain itu, pengujian ini juga memverifikasi bahwa pemanggilan prosedur (ProcCallStmt) berhasil menghubungkan argumen aktual dengan definisi prosedur yang sesuai.

File: 3-procedure-param.pas

Input
<pre>program CekParameterProsedur; prosedur printTwo(x: integer; s: string); mulai</pre>

<pre> writeln(s, x); selesai; variabel n: integer; mulai n := 7; printTwo('hello', n); { salah urutan/tipe parameter } selesai. </pre>
Output
<pre> ===== COMPILATION FAILED: SEMANTIC ERROR ===== Message : [SemanticError] Type mismatch in argument 1 of procedure 'printTwo'. Expected ints, but got strings. ===== </pre>

4.4. Pengujian 4

Tujuan pengujian: Untuk memvalidasi mekanisme *Scope Checking* dan deteksi duplikasi identifier. Pengujian ini bertujuan memastikan bahwa *Semantic Analyzer* dapat mendeteksi kesalahan jika terdapat identifier (dalam kasus ini PI) yang dideklarasikan ulang dengan nama yang sama dalam *scope* yang sama, yang melanggar aturan keunikan identifier dalam PASCAL-S.

File: 4-redeclare-constant.pas

Input
<pre> program CekRedeclarasiKonstanta; konstanta PI = 3; variabel PI: integer; mulai PI := 5; selesai. </pre>
Output
<pre> ===== COMPILATION FAILED: SEMANTIC ERROR ===== Message : [SemanticError] Identifier 'PI' already defined in this scope ===== </pre>

4.5. Pengujian 5

Tujuan pengujian:

File: 5-array.pas

Input								
<pre> program CekArray; variabel skor: larik [1 .. 3] dari integer; mulai skor[1] := 80; skor[2] := 65; skor[3] := 90; jika skor[3] > skor[1] maka writeln('skor[3] lebih besar') selain_itu writeln('skor[1] lebih besar atau sama'); selesai. </pre>								
Output								
<pre> ERROR:root:Syntax error: expected LPARENTHESIS(, but got LBRACKET([@ 5:7 ERROR:root:Syntax error: expected DOT(.), but got LBRACKET([@ 5:7 Semantic Analysis Successful. ===== SYMBOL TABLES ===== TAB (identifier table): idx id obj typ ref nrm lev adr link ----- 0 VARIABLE NOTYP 0 1 0 0 0 1 false CONSTANT BOOLS 0 1 0 0 0 2 true CONSTANT BOOLS 0 1 0 1 1 3 real TYPE REALS 0 1 0 1 2 4 char TYPE CHARS 0 1 0 1 3 5 boolean TYPE BOOLS 0 1 0 1 4 6 integer TYPE INTS 0 1 0 1 5 7 abs FUNCTION REALS 0 1 0 0 6 8 sqr FUNCTION REALS 0 1 0 2 7 9 odd FUNCTION BOOLS 0 1 0 4 8 10 chr FUNCTION CHARS 0 1 0 5 9 11 ord FUNCTION INTS 0 1 0 6 10 12 succ FUNCTION CHARS 0 1 0 7 11 13 pred FUNCTION CHARS 0 1 0 8 12 14 round FUNCTION INTS 0 1 0 9 13 15 trunc FUNCTION INTS 0 1 0 10 14 16 sin FUNCTION REALS 0 1 0 11 15 17 cos FUNCTION REALS 0 1 0 12 16 18 exp FUNCTION REALS 0 1 0 13 17 19 ln FUNCTION REALS 0 1 0 14 18 20 sqrt FUNCTION REALS 0 1 0 15 19 21 arctan FUNCTION REALS 0 1 0 16 20 22 eof FUNCTION BOOLS 0 1 0 17 21 23 eoln FUNCTION BOOLS 0 1 0 18 22 24 read PROCEDURE NOTYP 0 1 0 1 23 25 readln PROCEDURE NOTYP 0 1 0 2 24 26 write PROCEDURE NOTYP 0 1 0 3 25 27 writeln PROCEDURE NOTYP 0 1 0 4 26 28 PROCEDURE NOTYP 0 1 0 0 27 29 skor VARIABLE ARRAYS 0 1 0 0 28 </pre>								

```

BTAB (block table):
idx | last | lpar | psze | vsze
-----
0   | 30   | 0    | 0    | 0

ATAB (array table):
idx | xtyp | etyp | eref | low | high | elsz | size
-----
0   | INTS | INTS | 0    | 1   | 3    | 1    | 3

===== DECORATED AST =====
└─ Program [name=CekArray]
  └─ Block
    └─ VarDecl [symbol=29]
      └─ ArrayType
        └─ RangeExpr
          └─ NumberLiteral [type=ints]
            └─ NumberLiteral [type=ints]
              └─ PrimitiveType [name=integer]
        └─ CompoundStmt
          └─ AssignStmt
            └─ ArrayAccess
              └─ VarRef [name=skor]
                └─ NumberLiteral [type=ints]
          └─ AssignStmt
            └─ ArrayAccess
              └─ VarRef [name=skor]
                └─ NumberLiteral [type=ints]
          └─ AssignStmt
            └─ ArrayAccess
              └─ VarRef [name=skor]
                └─ NumberLiteral [type=ints]
          └─ IfStmt
            └─ BinOp [type=bools]
              └─ ArrayAccess [type=ints]
                └─ VarRef [name=skor]
                  └─ NumberLiteral [type=ints]
              └─ ArrayAccess [type=ints]
                └─ VarRef [name=skor]
                  └─ NumberLiteral [type=ints]
            └─ ProcCallStmt [name=writeln]
              └─ StringLiteral [type=strings]
            └─ ProcCallStmt [name=writeln]
              └─ StringLiteral [type=strings]

```

4.6. Pengujian 6

Tujuan pengujian:

File: 6-array-type.pas

Input
<pre> program CekArrayType; variabel arr: larik [1 .. 5] dari integer; i: integer; </pre>

<pre> idx: boolean; mulai arr[1] := 10; arr[2] := arr[1] + 5; arr[idx] := 3; { index harus integer } arr[3] := 'test'; { elemen integer tdk bs diberi string } selesai. </pre>
Output
<pre> ===== COMPILATION FAILED: SEMANTIC ERROR ===== Message : [SemanticError] Array index must be of integer type, got bools. ===== </pre>

4.7. Pengujian 7

Tujuan pengujian: Untuk memvalidasi *scope rules* pada bahasa Pascal-S. Pengujian ini bertujuan memastikan bahwa variabel yang dideklarasikan secara lokal di dalam sebuah prosedur (misalnya lokalY) bersifat privat dan tidak dapat diakses dari blok program utama (global scope). Sebaliknya, prosedur harus tetap dapat mengakses variabel global. Testcase ini diharapkan menghasilkan *Semantic Error* ketika program mencoba mengakses variabel lokal dari luar lingkungannya.

File: 7-scope-visibility.pas

Input
<pre> program CekScope; variabel globalX: integer; prosedur tesLokal; variabel lokalY: integer; mulai globalX := 10; { Valid: Prosedur bisa akses variabel global } lokalY := 5; { Valid: Akses variabel lokal sendiri } selesai; mulai globalX := 20; lokalY := 99; { Error: Variabel 'lokalY' tidak dikenali di scope global ini } selesai. </pre>
Output
<pre> ===== COMPILATION FAILED: SEMANTIC ERROR ===== Message : [SemanticError] Variable 'lokalY' not declared. ===== </pre>

4.8. Pengujian 8

Tujuan pengujian: Untuk memvalidasi mekanisme *Type Checking* pada ekspresi kondisi di dalam struktur kontrol (If, While, For). Pengujian ini memastikan bahwa ekspresi kondisi pada jika (if) dan selama (while) harus bertipe Boolean, sedangkan variabel iterator pada loop untuk (for) harus bertipe Integer. Testcase ini diharapkan menghasilkan *Semantic Error* karena terdapat ekspresi aritmatika (tipe Integer) yang digunakan sebagai kondisi If.

File: 8-control-flow-types.pas

Input
<pre>program CekTipeControlFlow; variabel i: integer; x: real; mulai i := 10; { Error 1: Kondisi IF harus boolean, bukan integer (hasil i + 5) } jika (i + 5) maka i := 0; { Error 2: Kondisi WHILE harus boolean } selama (i) lakukan i := i - 1; { Error 3: Iterator loop FOR harus integer, bukan real } untuk x := 1.5 ke 10.5 lakukan i := i + 1; selesai.</pre>
Output
<pre>===== COMPILATION FAILED: SEMANTIC ERROR ===== Message : [SemanticError] If condition must be of boolean expression. =====</pre>

4.9. Pengujian 9

Tujuan pengujian: Untuk memvalidasi sifat *immutability* dari identifier yang dideklarasikan sebagai konstanta. Pengujian ini bertujuan memastikan bahwa *Semantic Analyzer* mencegah operasi *assignment* kepada identifier yang berjenis konstanta setelah inisialisasi awal. Testcase ini diharapkan menghasilkan *Semantic Error* saat mencoba mengubah nilai konstanta BATAS.

File: 9-const-immutability.pas

Input
<pre> program CekKonstantaDanUndeclared; konstanta BATAS = 100; variabel angka: integer; mulai angka := 50; { Error 1: Mencoba mengubah nilai konstanta } BATAS := 200; { Error 2: Menggunakan variabel 'total' yang belum dideklarasikan } angka := total + 10; selesai. </pre>
Output
<pre> ===== COMPILATION FAILED: SEMANTIC ERROR ===== Message : [SemanticError] Cannot assign to 'BATAS' because it is a constant. ===== </pre>

4.10. Pengujian 10

Tujuan pengujian: Untuk memvalidasi konsistensi tipe data antara nilai yang dikembalikan oleh fungsi dengan tipe deklarasi fungsi tersebut. Pengujian ini memastikan bahwa nilai yang di-assign ke nama fungsi (sebagai *return value*) memiliki tipe data yang kompatibel dengan tipe yang didefinisikan pada header fungsi. Testcase ini diharapkan menghasilkan *Semantic Error* karena mencoba mengembalikan nilai boolean pada fungsi yang dideklarasikan bertipe integer.

File: 10-function-return.pas

Input
<pre> program CekReturnFungsi; variabel hasil: integer; fungsi hitungLuas(sisi: integer): integer; mulai { Valid: integer assigned to integer function } hitungLuas := sisi * sisi; jika (sisi < 0) maka { Error: Mencoba mengembalikan boolean pada fungsi bertipe integer } </pre>

```
    hitungLuas := false;
selesai;

mulai
    hasil := hitungLuas(5);
selesai.
```

Output

```
=====
COMPILATION FAILED: SEMANTIC ERROR
=====
Message : [SemanticError] Type mismatch in assignment. Cannot assign bools to
variable 'hitungLuas' of type ints.
=====
```

5. KESIMPULAN DAN SARAN

5.1. Kesimpulan

Proses *semantic analysis* berhasil diimplementasikan menggunakan attributed grammar, penyusunan Abstract Syntax Tree (AST), dan mekanisme type checking. Setiap node pada AST sekarang memiliki atribut semantik seperti tipe ekspresi, referensi simbol, dan informasi lingkup sehingga memungkinkan pengecekan terhadap operasi aritmatika, deklarasi variabel, parameter fungsi, serta struktur data komposit. Implementasi tiga jenis symbol table juga memastikan pengelolaan identifier, blok, dan tipe array dapat dilakukan secara konsisten dengan aturan Pascal-S. Setelah implementasi, compiler mampu memvalidasi makna program dan tidak hanya struktur sintaksnya. Hasil pengujian menunjukkan bahwa semantic analyzer berhasil membuat *decorated* AST dan mengecek aturan bahasa yang digunakan dan mengidentifikasi program yang valid secara makna dan memberikan pesan kesalahan yang jelas ketika ditemukan ketidaksesuaian.

5.2. Saran

Untuk pengerjaan selanjutnya, hasil semantic analysis yang telah dibangun pada *milestone* ini dapat dimanfaatkan sebagai dasar untuk proses *code generation* karena informasi tipe, scope, dan struktur AST sudah terbentuk.

LAMPIRAN

- Tautan *repository* GitHub : <https://github.com/anellautari/DFC-Tubes-IF2224>

PEMBAGIAN TUGAS

NAMA	NIM	TUGAS	Persentase (%)
Mayla Yaffa Ludmilla	13523050	Implementasi AST builder untuk Expression, Expression logic check. Pembuatan Test Cases. Laporan Kesimpulan & Saran.	25
Anella Utari Gunadi	13523078	Implementasi Symbol Table, Scope Logic, dan Program Structure. Laporan Deskripsi Tugas, Perancangan, dan Implementasi.	25
Muhammad Edo Raduputu Aprima	13523096	Implementasi AST builder dan analyzer untuk Statement, Laporan Pengujian.	25
Athian Nugraha Muarajuang	13523106	Implementasi Struktur Data AST, Setup ASTBuilder. Laporan Landasan Teori.	25

REFERENSI

- [1] Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. 3rd ed, 2007. Tersedia: https://cdn-edunex.itb.ac.id/29161-Formal-Language-Theory-and-Automata/1629640939613_Introduction-to-Automata-Theory,-Languages,-and-Computation-Edition-3.pdf. [Diakses: 25-November-2025].
- [2] Wirth, Niklaus. "PASCAL-S: A Subset and its implementation", Tersedia: <http://pascal.hansotten.com/uploads/pascals/PASCAL-S%20A%20subset%20and%20its%20Implementation%20012.pdf>. [Diakses: 25-November-2025].
- [3] tutorialspoint.com. *Compiler Design*. Tersedia: https://www.tutorialspoint.com/compiler_design/index.htm. [Diakses 26-November-2025]
- [4] tutorialspoint.com. *Compiler Design - Attributed Grammars*. Tersedia: https://www.tutorialspoint.com/compiler_design/compiler_design_attributed_grammars.htm. [Diakses 30-November-2025]
- [5] Slonneger, Ken. *Syntax and Semantics of Programming Languages*. Tersedia: <http://homepage.divms.uiowa.edu/%7Eslonnegr/plf/Book/Chapter1.pdf>. [Diakses 30-November-2025]