IF2224 TEORI BAHASA FORMAL DAN OTOMATA

IMPLEMENTASI SEMANTIC ANALYSIS PADA PASCAL-S COMPILER BERBASIS ABSTRACT SYNTAX TREE

Laporan Milestone 3

Disusun untuk memenuhi tugas mata kuliah Teori Bahasa Formal dan Otomata pada Semester 5 (lima)

Tahun Akademik 2025/2026

Disusun Oleh:

Brian Ricardo Tamin	13523126
Jovandra Otniel P. S.	13523141
Andrew Tedjapratama	13523148
Theo Kurniady	13523154

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2025

DAFTAR ISI

BAB I
DESKRIPSI PERSOALAN

1.1 Pendahuluan

Bahasa pemrograman Pascal-S merupakan sebuah subset dari bahasa Pascal yang dirancang dengan aturan sintaks yang lebih terbatas. Pada tahap ketiga pembangunan compiler Pascal-S, setelah mengembangkan tahap Lexical Analysis dan Syntax Analysis, compiler yang dibangun sudah bisa berhasil memecah kode sumber menjadi token dan menyusunnya menjadi struktur pohon parsing (Parse Tree) berdasarkan aturan tata bahasa (*grammar*) yang ditentukan. Parser dalam milestone 2 memastikan bahwa kode program ditulis dengan struktur kalimat yang benar.

Namun, kebenaran secara sintaks atau tata bahasa tidak menjamin kebenaran secara makna (*semantic*). Sebuah program bisa saja lolos valid dari parser meskipun melakukan operasi yang tidak masuk akal, contohnya seperti menjumlahkan tipe data boolean dengan integer, atau menggunakan variabel yang belum dideklarasikan. Kesalahan-kesalahan ini berkaitan dengan konsistensi tipe dan lingkup (*scope & type checking*) variabel yang tidak dapat dideteksi hanya dengan *Context-Free Grammar* (CFG).

Oleh karena itu, dalam Milestone 3, fokus akan berpindah pada implementasi Semantic Analysis. Tahap ini bertujuan untuk memvalidasi makna dari struktur program yang telah dibentuk. Proses ini melibatkan penggunaan *Attributed Grammar* dan pengelolaan Symbol Table untuk melacak deklarasi identifier, tipe data, dan scope, serta adanya *Visitor Functions* untuk menelusuri *Abstract Syntax Tree* (AST). Pada akhir, proses ini akan menghasilkan suatu hasil akhir berupa *Decorated Abstract Syntax Tree* (AST) yang telah dianotasi dengan informasi tipe ekspresi, referensi ke symbol table, dan informasi scope, serta jaminan bahwa program bebas dari kesalahan semantik sebelum masuk ke tahap berikutnya, yaitu tahap *Intermediate Code Generation*.

1.2 Rumusan Masalah

Berdasarkan latar belakang di atas, permasalahan utama yang harus diselesaikan dalam milestone ini adalah:

1. Pengelolaan Symbol Table: Bagaimana merancang struktur data Symbol Table yang efisien (menggunakan tabel tab, btab, dan atab) untuk menyimpan informasi identifier, menangani *nested scope* (misalnya prosedur dalam prosedur), dan membedakan antara *pass-by-value* dengan *pass-by-reference*?

2. Validasi Tipe (*Type Checking*): Bagaimana mekanisme pengecekan kompatibilitas tipe data pada operasi assignment, ekspresi aritmatika, dan pemanggilan fungsi agar sesuai dengan aturan ketat Pascal-S?

3. Konstruksi Decorated AST: Bagaimana mengubah Parse Tree mentah menjadi Abstract Syntax Tree (AST) yang lebih bersih dengan informasi yang lengkap, lalu melakukan traversals (menggunakan pola Visitor) untuk menyematkan atribut semantik (seperti tipe data dan referensi tabel) pada setiap node berbeda?

4. Penanganan Error Semantik: Bagaimana mendeteksi dan melaporkan kesalahan semantik seperti Undeclared Identifier, Type Mismatch, atau Redeclared Identifier secara akurat tanpa menghentikan atau melakukan *crash* kasar pada compiler?

1.3 Tujuan

Milestone ini bertujuan untuk:

1. Mengimplementasikan Semantic Analyzer yang mampu menelusuri AST untuk memverifikasi kebenaran logika program.

2. Membangun dan mengelola Symbol Table yang terdiri dari tabel identifier (tab), tabel blok (btab), dan tabel array (atab) sesuai spesifikasi Pascal-S untuk manajemen memori dan scope.

3. Menghasilkan Decorated AST, yaitu pohon sintaks abstrak yang setiap node-nya telah dilengkapi dengan informasi tipe data (type) dan referensi tabel simbol (tab_index).

4. Memastikan aturan strong typing pada Pascal-S terpenuhi, termasuk validasi operasi matematika, logika, dan akses array/record.

5. Menangani deklarasi tipe kompleks seperti Array dan Record beserta akses elemennya (seperti arr[i] atau rec.field).

1.4 Batasan Masalah

Lingkup pengerjaan pada tahap ini dibatasi pada:

1. Analisis semantik dilakukan secara statis (compile-time), tidak menangani runtime error.
2. Implementasi mengacu pada spesifikasi bahasa Pascal-S, termasuk aturan *predefined procedures* (writeln, readln), tipe data dasar (integer, real, boolean, char), dan daftar *reserved words*.

BAB II
LANDASAN TEORI

2.1 Semantic Analysis and Attributed Grammar

2.1.1 Semantic Analysis

Semantic analysis adalah salah satu tahapan krusial dari compiler yang memeriksa struktur semantik (makna) dari kode pemrograman yang sudah diubah menjadi *syntax tree* oleh parser. Sebelumnya, *Syntax Analysis* (Parser) hanya peduli pada "apakah struktur kalimatnya benar secara tata bahasa?", namun pada tahap ini, Semantic Analysis berfokus pada "apakah kalimat tersebut masuk akal?". Hal ini diperlukan karena parser berbasis Context-Free Grammar (CFG) memiliki keterbatasan dalam mendeteksi kesalahan yang bergantung pada konteks (context-sensitive), seperti penggunaan variabel yang belum dideklarasikan.

Dalam implementasinya, Semantic Analyzer terdiri dari tiga komponen fundamental: **Abstract Syntax Tree (AST)** sebagai representasi struktur program, **Symbol Table** sebagai memori persisten untuk melacak entitas, dan **Type Checking** untuk memvalidasi kompatibilitas data. Ketiganya bekerja secara sinergis melalui mekanisme tree traversal (sering menggunakan pola Visitor) untuk menelusuri setiap node dan melakukan validasi.

Selain validasi, tahap ini juga mencakup mekanisme penanganan kesalahan (error handling) untuk isu-isu semantik, seperti Type Mismatch (ketidaksesuaian tipe data pada assignment atau operasi), Argument Mismatch (jumlah/tipe parameter fungsi tidak sesuai), serta Undeclared Identifier. Validasi ini sangat penting terutama bagi Pascal-S yang menganut sistem ***Strong Static Typing***, di mana setiap variabel wajib dideklarasikan secara eksplisit dengan tipe data tertentu sebelum digunakan.

Keluaran atau output akhir dari fase ini adalah Decorated AST, yaitu pohon sintaks yang telah dianotasi dengan atribut semantik (seperti tipe data hasil evaluasi) dan referensi ke Symbol Table. Symbol Table itu sendiri adalah struktur data yang menyimpan informasi komprehensif mengenai identifier (nama, tipe, scope) dan diorganisasikan menggunakan struktur stack terpisah (tab, btab, atab) untuk menangani nested scope secara efisien.

2.1.2 Attributed Grammar

Dalam implementasi milestone ini, digunakan algoritma L-Attributed Grammar, yang secara definisi umum merupakan grammar yang dilengkapi dengan atribut dan aturan semantik. Attributed Grammar adalah perluasan dari Context-Free Grammar (CFG) yang memungkinkan setiap simbol (terminal maupun non-terminal) memiliki atribut. Setiap aturan produksi dalam attributed grammar dilengkapi dengan aturan semantik yang menentukan bagaimana atribut dihitung.

Dalam teori klasik, atribut dibagi menjadi dua jenis:

- **Synthesized Attribute**
  Mengalir dari *child* ke *parent*, digunakan untuk menghitung nilai ekspresi atau menentukan tipe hasil operasi.
- **Inherited Attribute**
  Mengalir dari *parent* atau *sibling* ke *child*. Berguna untuk meneruskan informasi konteks, seperti scope atau tipe deklarasi.

Penggunaan attributed grammar sangat penting dalam *semantic analysis* karena grammar CFG saja tidak cukup untuk memvalidasi konteks dan makna program. Misalnya, pada aturan produksi untuk assignment:

$$< assignment > \rightarrow < variable > := \ < expression >$$

Semantic rule yang terkait dapat berupa:

- variable.type = lookup(variable.name)
- expression.type = evaluate(expression)
- check(variable.type == expression.type)

Melalui skema ini, compiler dapat memastikan bahwa operasi yang dilakukan konsisten secara semantik. Dalam compiler Pascal-S, Attributed Grammar menjadi dasar Syntax Directed Translation (SDT) yang berperan besar pada proses pembentukan AST dan type checking, sebagaimana dijelaskan pada bagian lain dalam laporan. Dengan demikian, attributed grammar berfungsi sebagai jembatan antara struktur sintaks (hasil parsing) dan struktur makna (data semantik) yang diperlukan pada tahap selanjutnya.

2.2 Abstract Syntax Tree

Abstract Syntax Tree adalah representasi hierarkis terstruktur dari kode program yang lebih ringkas daripada *syntax tree* pada umumnya. AST merepresentasikan suatu tree yang telah diabstraksi dari Parse Tree yang dibentuk pada milestone 2, berisi detail derivasi tata bahasa (termasuk elemen sintaks dekoratif seperti titik koma, tanda kurung, dan keyword begin/end), maka AST hanya menyisakan atau mempertahankan elemen-elemen yang dianggap penting dan esensial secara semantik.

Konstruksi AST dalam tugas ini didasarkan pada prinsip *Syntax-Directed Translation Scheme*, mengidentifikasi setiap aturan produksi (production rule) dan aksi semantik (semantic action) yang terkait. Saat aksi semantik tersebut dijalankan pada node parse tree, sistem akan secara dinamis membentuk objek node baru yang lebih ringkas dan terstruktur, yang kemudian disusun menjadi Abstract Syntax Tree (AST). Pendekatan ini memastikan bahwa transformasi dari struktur sintaksis mentah menjadi struktur logis berjalan secara deterministik dan akurat.

Setelah AST sudah terbentuk, proses Semantic Analysis kemudian dilakukan dengan menelusuri setiap node menggunakan *semantic visitor*, yaitu serangkaian fungsi visit yang pada spesifikasi milestone ini, dilakukan secara top-down mengikuti struktur tree. Pendekatan ini memungkinkan informasi konteks seperti scope, level blok, dan tipe deklarasi diwariskan dari node induk ke node anak. Selama traversal, visitor mencatat deklarasi baru ke dalam symbol table, memvalidasi penggunaan identifier, mengecek konsistensi tipe pada ekspresi serta assignment, dan memberi anotasi semantik (seperti tipe hasil evaluasi atau referensi tabel simbol) langsung ke node AST.

2.3 Symbol Table

Symbol table adalah struktur data sentral dalam proses kompilasi di tahap *Semantic Analysis*, yang berfungsi seolah sebagai “database” untuk menyimpan informasi mengenai semua identifier yang dideklarasikan dalam program (variabel, konstanta, tipe, prosedur, fungsi). Perlu dipahami bahwa fase sebelumnya, yaitu *Syntax Analysis* berbasis CFG memiliki keterbatasan fundamental, ia bersifat *stateless* atau tidak memiliki memori tentang apa yang telah dideklarasikan sebelumnya. Parser hanya mampu memvalidasi urutan token (subjek-predikat-objek), tetapi buta terhadap konteks. Oleh karena itu, Symbol Table hadir sebagai struktur data persisten yang bertugas merekam, memelihara, dan melacak status serta

informasi dari setiap entitas sepanjang proses kompilasi, memastikan bahwa setiap penggunaan variabel konsisten secara makna, dengan deklarasi awalnya.

Sesuai dengan spesifikasi Pascal-S compiler, implementasi symbol table dibagi menjadi tiga jenis symbol table:

2.3.1 Identifier Table (tab)

Tabel tab menyimpan entri untuk setiap identifier yang dideklarasikan. Struktur data ini diorganisasikan sebagai stack global, namun secara logis dikelola menggunakan Linked List per scope untuk resolusi identifier yang efisien. Setiap entri pada tab memiliki atribut yang dapat dilihat pada tabel berikut:

Tabel 2.3.1 List atribut tabel tab dan deskripsinya

atribut	deskripsi
identifiers	Nama identifier (misalnya nama variabel, konstanta, tipe, prosedur, fungsi). Indeks dimulai dari 29 karena ada reserved words.
link	Pointer/indeks ke identifier sebelumnya dalam scope yang sama. Digunakan untuk manajemen scope (linked list per blok).
obj	Kelas objek yang dienumerasi: konstanta, variabel, tipe, prosedur, fungsi, dll.
type	Tipe dasar dari identifier, misalnya: integer, boolean, char, real, array, record, dll. Biasanya berupa kode numerik.
ref	Pointer/indeks ke tabel lain jika tipe adalah komposit (array/record). Mengarah ke atab (array table) atau btab (record/procedure block).
nrm	Menandai apakah identifier adalah variabel normal (=1) atau parameter by-reference (var parameter) (=0).
lev	Tingkat *lexical level* tempat identifier dideklarasikan (0 = global, 1 = dalam prosedur, 2 = dalam prosedur di dalam prosedur, dst).

adr	Makna tergantung jenis objek: offset variabel di stack frame, nilai konstanta, offset field record, alamat prosedur, atau ukuran/penanda lain.

2.3.2 Block Table (btab)

Tabel btab digunakan untuk menyimpan informasi mengenai struktur blok kode, yaitu program utama, prosedur, fungsi, dan definisi tipe record. Setiap entri pada btab memiliki atribut yang dapat dilihat pada tabel berikut:

Tabel 2.3.2 List atribut tabel btab dan deskripsinya

atribut	deskripsi
blocks	Indeks entri block (setiap block mewakili prosedur, fungsi, atau record type definition).
last	Pointer/indeks ke identifier terakhir yang dideklarasikan di dalam block tersebut (menghubungkan field record, parameter, atau variabel lokal).
lpar	Pointer/indeks ke parameter terakhir dari prosedur/fungsi pada block tersebut. Jika block adalah record, nilainya 0.
psze	Total ukuran parameter block (dalam byte/unit memori).
vsze	Total ukuran variabel lokal block (dalam byte/unit memori)

2.3.3 Array Table (atab)

Karena array memiliki struktur yang kompleks (tipe elemen, tipe indeks, dan rentang), informasinya dipisahkan ke dalam tabel atab. Setiap entri array atab didefinisikan pada tabel berikut:

Tabel 2.3.3 List atribut tabel atab dan deskripsinya

atribut	deskripsi
arrays	Indeks entri array

xtyp	Tipe indeks array (misalnya integer). Berupa kode tipe dari tabel tab.
etyp	Tipe elemen array (misalnya integer). Berupa kode tipe dari tabel tab.
eref	Pointer/indeks ke detail tipe elemen jika elemen adalah tipe komposit (misalnya array dalam array, atau record). Mengarah ke atab atau btab.
low	Batas bawah indeks array (misalnya 1 pada array[1..10] atau 0 pada array[0..15]).
high	Batas atas indeks array.
elsz	Ukuran satu elemen array (dalam byte/unit memori).
size	Total ukuran array

2.4 Type Check

Type checking adalah proses yang penting dalam menentukan kesesuaian tipe pada statement, deklarasi, atau ekspresi pada kode program. Type check merupakan tahap selanjutnya dari Symbol Table. Hal ini karena proses tersebut mengambil informasi dari symbol table untuk mengecek kesesuaian tipe data yang dipakai. Output dari Type Check adalah decorated abstract syntax tree dimana setiap simpul (node) dilabeli tipe data yang sesuai. Jika terdapat semantic error, type checker akan mengeluarkan Error (*raise error*) yang merupakan bagian dari *compile error*.

2.4.1 Representasi Tipe

Berdasarkan implementasi type_check pada aplikasi semantic analyzer, tipe data dibagi menjadi 7 macam, yaitu integer, real, boolean, char, array, record, dan notype. Integer merupakan bilangan bulat. Real merupakan bilangan floating-point (bilangan desimal). Boolean merupakan tipe data yang berisi nilai kebenaran (true/false). Char merupakan tipe data yang menyimpan satu karakter yang direpresentasikan oleh ASCII. Array merupakan tipe data yang menyimpan beberapa nilai dengan tipe sama pada suatu blok memori kontigu. Record merupakan tipe data yang menyimpan

beberapa nilai dengan tipe yang berbeda pada suatu blok memori kontigu. NoType merupakan label tipe data yang tidak dikenali di Pascal-S.

2.4.2 Assignment Checking

Pada bahasa pemrograman Pascal-S, assignment wajib mengikuti urutan ekspresi seperti berikut:

$$< target > := \ < expression >$$

Aturan kompatibilitas tipe data terdiri dari:

- Tipe target harus sama dengan nilai
- Integer boleh diassign ke real
- Target harus berupa i-value (identifier yang bisa diubah)
- Tidak boleh melakukan assignment ke const.

2.4.3 Pemeriksaan Operasi Aritmatika, Relasional, dan Logika

Pada Pascal-S, terdapat tiga operasi yang sangat penting, yaitu operasi aritmatika, relasional, dan logika. Operasi aritmatika melibatkan penjumlahan, pengurangan, perkalian, pembagian (real dan integer), dan modulo. Operasi relasional merupakan operasi yang membandingkan nilai dari kedua operand (lebih dari, kurang dari, sama dengan). Sementara itu, operasi logika menggunakan operasi logika dasar (and, or, not) untuk mencari nilai true/false dari operand yang terlibat.

Pada Pascal-S, penjumlahan, pengurangan, dan perkalian menghasilkan integer jika kedua operand integer atau real jika paling sedikit salah satu operand berupa real. Pembagian integer (div) menghasilkan nilai integer yang dibulatkan ke bawah, sedangkan pembagian real (/) menghasilkan nilai real. Modulo selalu menghasilkan integer.

Pada operasi relasional, hasil selalu berupa boolean. Operasi kesetaraan (= dan <>) harus melibatkan 2 tipe data yang sama. Sementara itu, operasi perbandingan (seperti >, <, >=, <=) harus melibatkan 2 tipe data ordinal seperti integer dan char.

2.4.4 Pemeriksaan Index Array, Ekspresi Kondisi, dan Rentang Perulangan For

Pengaksesan array arr[index] diperiksa oleh semantic analyzer, dimana arr harus berupa variabel dengan tipe data array, dan index merupakan nilai integer. Pada blok if-then, bagian yang terdapat dalam blok kondisional (if) harus menghasilkan nilai boolean, baik itu dari operasi relasional maupun logika. Pada blok perulangan seperti *for i := angka_start to angka_end do …*, variabel loop (i) harus berupa ordinal (integer, boolean, char). Selain itu, tipe data *angka_start* dan *angka_end* harus sama.

2.5 Error Handling

Jika pada analisis semantik ditemukan kesalahan, compiler Pascal-S akan melempar error sesuai kesalahan pada kode program tersebut. Kesalahan tersebut mencakup variabel yang tidak dideklarasikan, ketidakcocokan tipe data, dan pemanggilan fungsi/prosedur yang argumennya tidak sesuai. Kesalahan tersebut menjadi rincian kesalahan yang terdapat pada kode yang sedang dianalisis secara semantik.

BAB III
PERANCANGAN & ALUR PROGRAM

Penjelasan ini merinci rancangan program *semantic analyzer* untuk *compiler* Pascal-S, mencakup teknologi, struktur, alur kerja, dan justifikasi atas pilihan desain yang diambil.

3.1 Arsitektur dan Teknologi yang Digunakan

Program ini dibangun murni menggunakan bahasa pemrograman Python 3. Tidak ada *library* eksternal yang digunakan untuk logika inti. Pemilihan Python didasarkan pada keterbacaan kode, kemudahan pengelolaan struktur data seperti stack dan dictionary yang digunakan pada symbol table, serta dukungan Python terhadap pendekatan rekursif yang sesuai dengan pola kerja visitor functions dan tree traversal, menghasilkan implementasi yang sederhana namun tetap terstruktur.

3.2 Struktur Program dan Kelas Utama

```
BBC-TUBES-IF2224
├── main.py
├── README.md
├── doc/
├── src/
│   ├── compiler.py
│   ├── utils.py
│   ├── config/
│   │   ├── states.json
│   │   ├── token_maps.json
│   │   └── transitions.json
│   ├── dfa/
│   ├── lexer/
│   ├── parser/
│   └── semantic/
│       ├── semantic_analyzer.py
│       ├── string_handler.py
│       ├── errors.py
│       ├── AST/
│       │   ├── ast_builder.py
│       │   ├── ast_node.py
│       │   └── nodes.py
│       ├── symbol/
│       │   ├── symbol_table.py
│       │   └── constants.py
│       └── type_checker/
│           └── type_checker.py
└── test/
    ├── milestone-1/
    │   ├── input/
    │   ├── input-indo/
    │   └── output/
    ├── milestone-2/
    │   ├── input/
    │   └── output/
    └── milestone-3/
        ├── input/
        └── output/
```

3.3 Alur Kerja Program

Alur kerja program dimulai dari pembacaan parse tree hingga pembuatan AST, tabel simbol, dan pemeriksaan semantik. Proses ini dapat dijelaskan secara bertahap untuk memberikan pemahaman menyeluruh tentang mekanisme analisis semantik yang diterapkan.

1. Pembentukan AST (*Abstract Syntax Tree*)
   Program memulai analisis dengan menerima parse tree yang dihasilkan oleh parser pada milestone-2. Parse tree ini merepresentasikan struktur sintaksis program secara hierarkis berdasarkan grammar Pascal.

2. Inisialisasi tabel simbol dan built-in
   Setelah AST terbentuk, program menginisialisasi Symbol Table yang menyimpan semua identifier, tipe, variabel, prosedur/fungsi, serta array. Reserved words dan prosedur/fungsi bawaan (built-in) dimasukkan pada level global, sehingga dapat dikenali sepanjang analisis. Setiap blok baru atau scope lokal akan otomatis dicatat untuk mendukung scoping dan lookup identifier.

3. Traversal AST untuk mengisi tabel simbol
   Program melakukan traversal AST secara rekursif menggunakan berbagai metode visit. Setiap node dikunjungi, dan deklarasi variabel, konstanta, tipe, prosedur, dan fungsi dicatat ke tabel simbol lengkap dengan level leksikal, alamat, dan referensi blok. Record dan array ditangani secara khusus agar akses field atau elemen array sesuai dengan tipe yang benar.

4. Pemeriksaan tipe dan konsistensi
   Selama traversal, tipe ekspresi, assignment, operasi biner/unary, serta pemanggilan fungsi dan prosedur diperiksa. Program memastikan kompatibilitas tipe operand, tipe return fungsi, serta jumlah dan tipe argumen pada prosedur/fungsi. Pemeriksaan ini mencegah kesalahan semantik, seperti penggunaan variabel yang belum dideklarasikan atau tipe yang tidak sesuai.

5. Dekorasi AST
   Semantic analyzer kemudian melakukan traversal parse tree, mengunjungi setiap node untuk mengekstrak informasi deklarasi, tipe data, ekspresi, dan pernyataan. Setiap node dihias (decorated) dengan atribut tambahan seperti tipe dan indeks tabel simbol, sehingga memudahkan proses pemeriksaan semantik pada tahap berikutnya.

6. Penanganan error semantik
   Bila ada pelanggaran semantik, akan ada error handling dari kelas-kelas error pada file Errors.py. Setiap pelanggaran semantik dicatat ke daftar errors tanpa menghentikan analisis. Hal ini memungkinkan program untuk mendeteksi dan melaporkan semua kesalahan sekaligus.

3.4 Justifikasi Pilihan Implementasi

1. Penggunaan AST (*Abstract Syntax Tree*)
   Pemilihan AST sebagai dasar analisis semantik karena AST sudah menyederhanakan struktur kode dibandingkan parse tree lengkap. Dengan AST, node-node yang tidak relevan (seperti tanda kurung atau token terminal yang sederhana) dihilangkan, sehingga traversal menjadi lebih efisien dan fokus pada elemen semantik penting. Hal ini memudahkan anotasi tipe, referensi simbol, dan deteksi kesalahan semantik secara lebih terstruktur.

2. Penggunaan Symbol Table untuk Scope dan Tipe
   Implementasi tabel simbol mendukung pencatatan entri untuk variabel, konstanta, fungsi, prosedur, dan tipe kustom seperti record maupun array. Selain itu, struktur ini memfasilitasi manajemen nested scope, sehingga deklarasi lokal dan global dapat dibedakan dengan jelas. Pendekatan ini juga memungkinkan akses cepat ke informasi tipe saat melakukan pemeriksaan semantik.

3. Pemisahan TypeChecker dari SemanticAnalyzer
   Implementasi checker tipe terpisah dari traversal semantic analyzer meningkatkan modularitas dan keterbacaan kode. TypeChecker bertanggung jawab pada logika kompatibilitas tipe, sementara SemanticAnalyzer fokus pada traversal AST, anotasi node, dan pencatatan error. Pemisahan ini memudahkan debugging, pengujian, dan kemungkinan reuse untuk analisis semantik bahasa lain di masa depan.

4. Gabungan Analisis Semantik dan Error Handling
   Dengan mengintegrasikan analisis tipe dan penanganan kesalahan dalam satu proses traversal, semantic analyzer dapat memberikan umpan balik kesalahan semantik secara real-time. Pendekatan ini mengurangi kebutuhan pass tambahan dan membantu mendeteksi error lebih awal, meningkatkan keandalan pemeriksaan semantik.

BAB IV
IMPLEMENTASI PROGRAM

4.1 AST Node

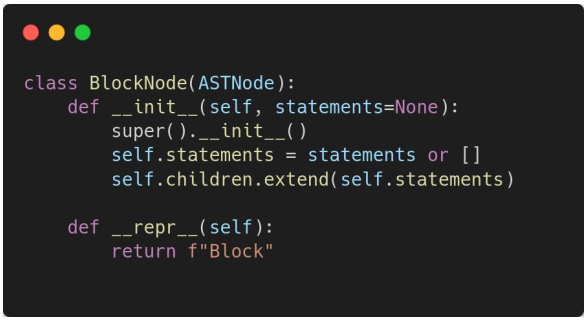
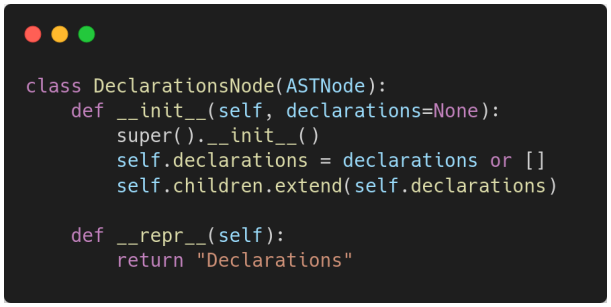
Kelas ini menjadi struktur utama dari AST dalam membentuk pohon.

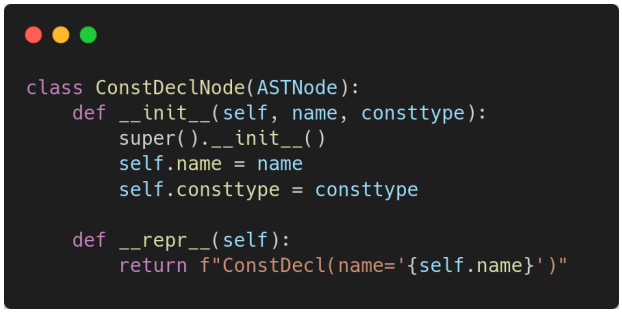
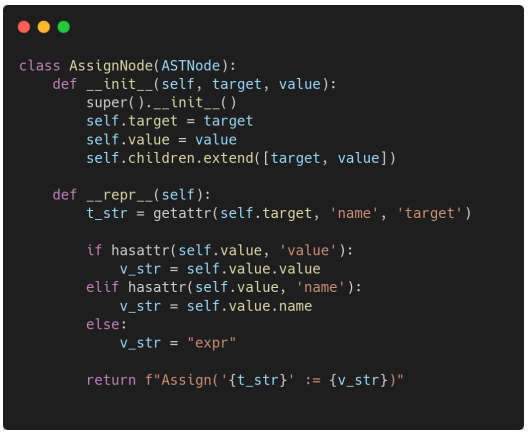
Class	Penjelasan Singkat
	**Class ASTNode** Kelas ASTNode adalah representasi dasar dari sebuah node dalam Abstract Syntax Tree (AST), struktur data yang digunakan untuk merepresentasikan hierarki sintaks program. Setiap node menyimpan daftar children, yaitu node anak yang merepresentasikan substruktur dari program, serta dictionary attr yang menyimpan atribut tambahan seperti tipe data, indeks di symbol table, level blok, atau informasi lain yang relevan untuk analisis semantik. Node ini bersifat fleksibel, sehingga dapat menampung node lain sebagai anaknya atau data sederhana seperti string atau angka. Kelas ini juga menyediakan mekanisme untuk mencetak AST secara visual menggunakan metode __str__ dan _format_tree, menghasilkan tampilan tree ASCII yang rapi dengan indentasi dan connector untuk menunjukkan hubungan hierarki antar node. Setiap node dapat menampilkan anotasi tambahan melalui _get_annotation_str, yang mengekstrak informasi dari atribut seperti tipe data (type), indeks tabel simbol (tab_index), level blok (block_index), atau level lexical (lev). Dengan cara ini, ASTNode tidak hanya menyimpan struktur program, tetapi juga metadata penting yang memudahkan debugging dan analisis semantik.

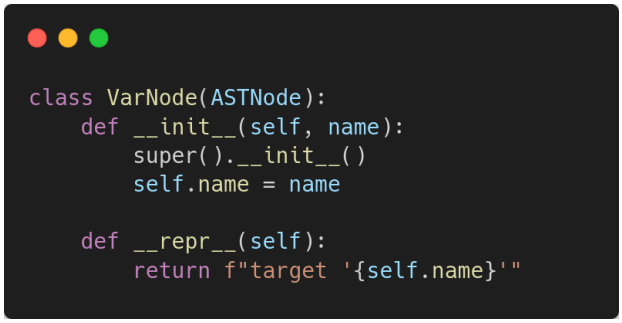
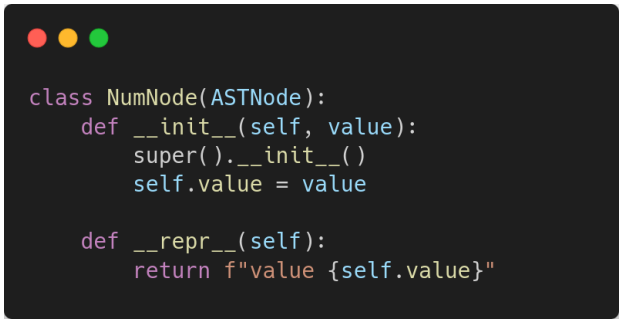
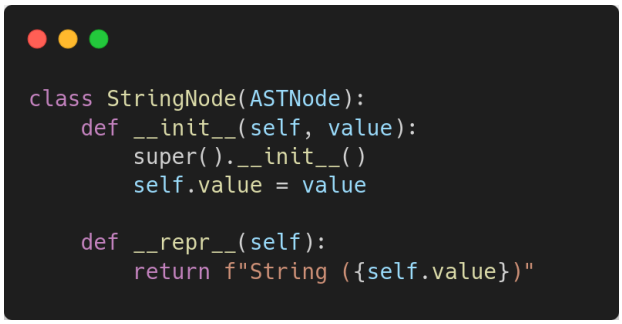
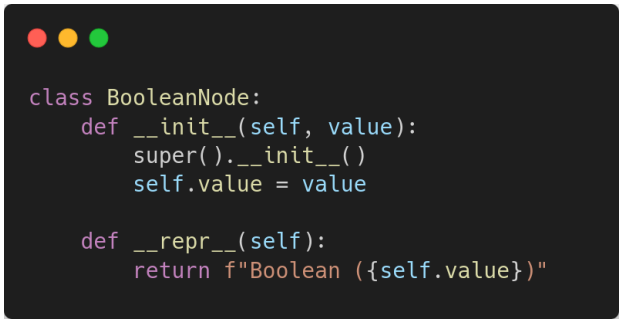
```python
from ..symbol.constants import TypeKind
class ASTNode:
    def __init__(self):
        self.children = []
        self.attr = {}

    def add_child(self, child):
        self.children.append(child)

    def __str__(self):
        return self._format_tree("", True)

    def _format_tree(self, prefix, is_last):
        connector = "└─ " if is_last else "├─ "
        annotation = self._get_annotation_str()

        # Bersihkan repr dari newline
        node_repr = repr(self).strip()
        line = f"{prefix}{connector}{node_repr}{annotation}\n"

        child_prefix = prefix + ("    " if is_last else "│   ")
        valid_children = [c for c in self.children if c is not None]
        count = len(valid_children)

        for i, child in enumerate(valid_children):
            if hasattr(child, '_format_tree'):
                line += child._format_tree(child_prefix, i == count - 1)
            else:
                line += f"{child_prefix}{'└─ ' if i == count-1 else '├─ '}{str(child)}\n"
        return line

    def _get_annotation_str(self):
        parts = []
        type_map = {
            TypeKind.NOTYPE: "void", TypeKind.INTEGER: "integer",
            TypeKind.REAL: "real", TypeKind.BOOLEAN: "boolean",
            TypeKind.CHAR: "char", "predefined": "predefined"
        }

        if self.attr.get('type') == 'predefined': parts.append("predefined")

        if 'tab_index' in self.attr and self.attr['tab_index'] is not None:
            parts.append(f"tab_index:{self.attr['tab_index']}")

        if 'block_index' in self.attr:
            parts.append(f"block_index:{self.attr['block_index']}")

        if 'type' in self.attr and self.attr['type'] != 'predefined':
            val = self.attr['type']
            t_str = type_map.get(val, str(val)) if isinstance(val, int) else str(val)
            parts.append(f"type:{t_str}")

        if 'lev' in self.attr: parts.append(f"lev:{self.attr['lev']}")

        return " → " + ", ".join(parts) if parts else ""
```

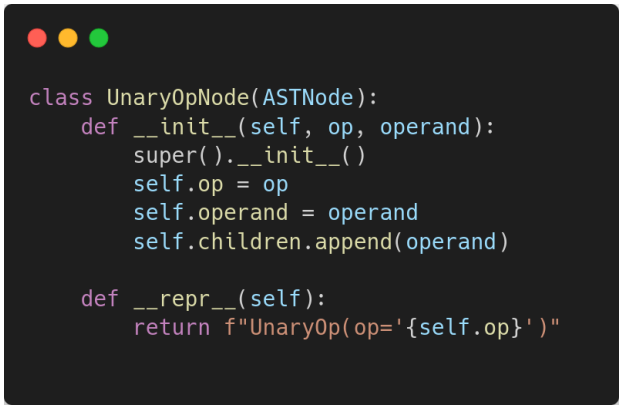
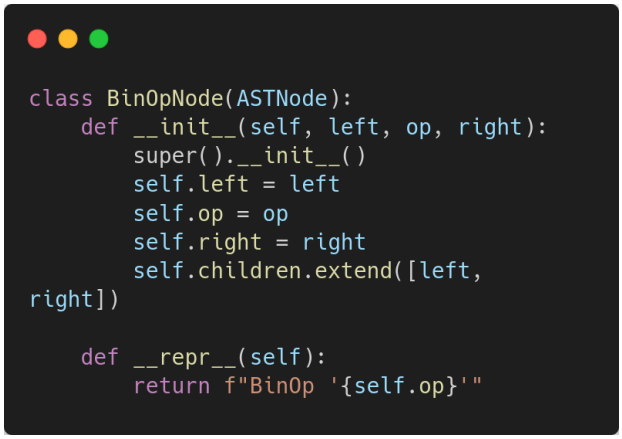
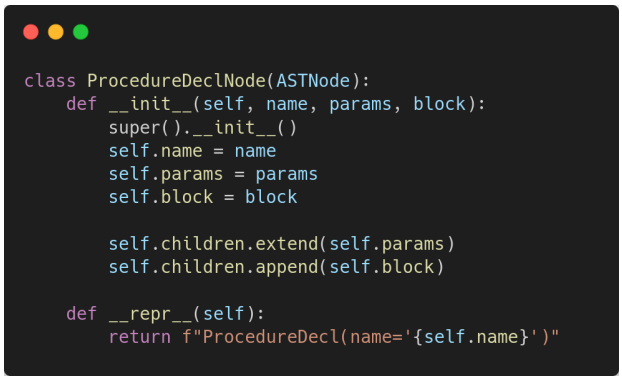
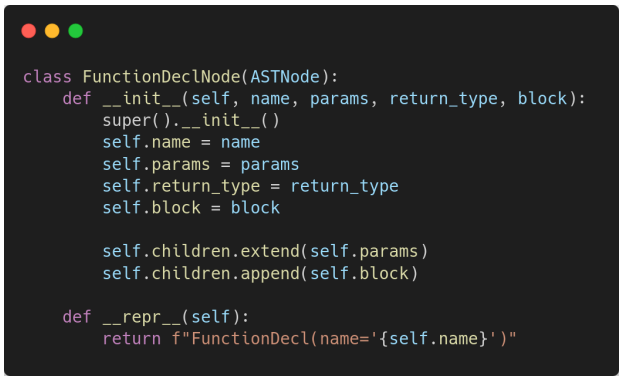
4.2 Nodes

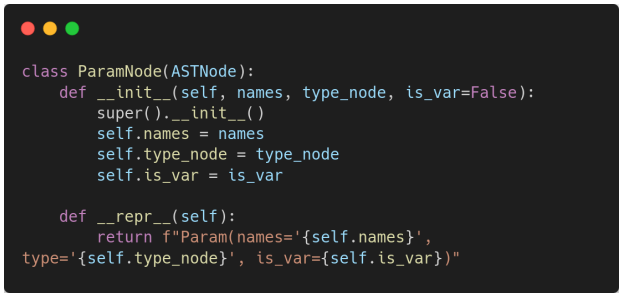
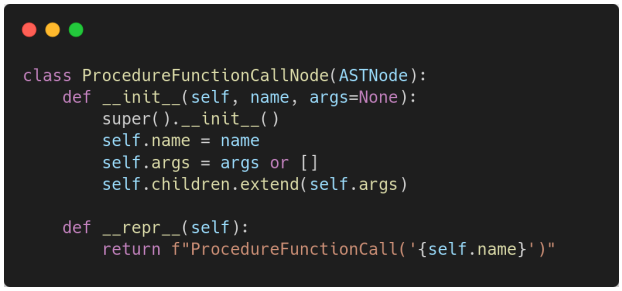
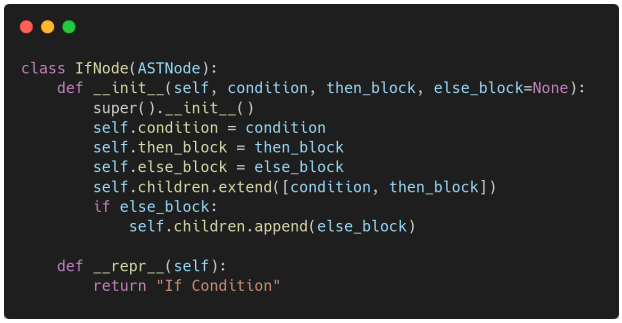
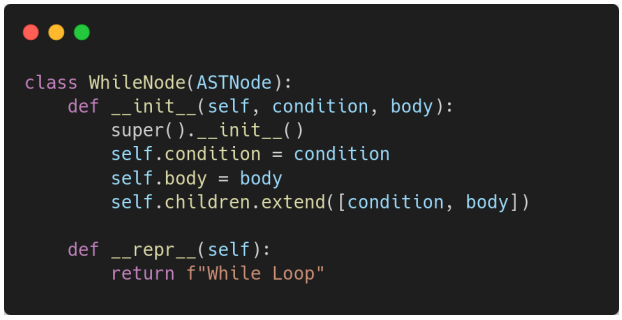
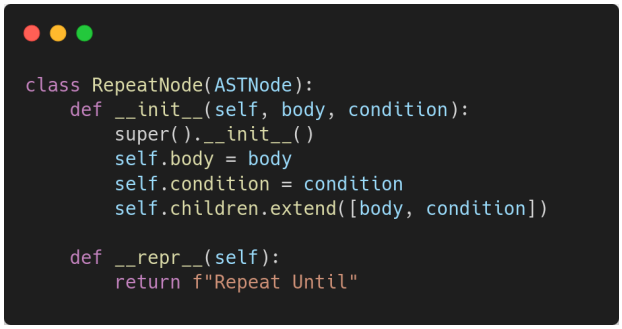
Kumpulan kelas yang merepresentasikan setiap node dalam AST.

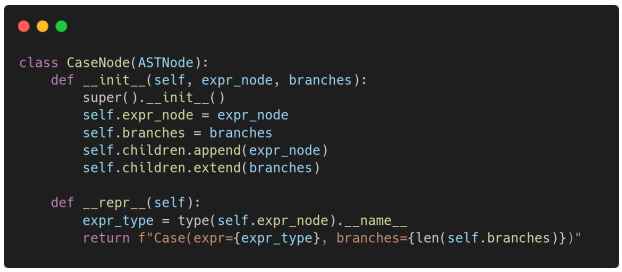
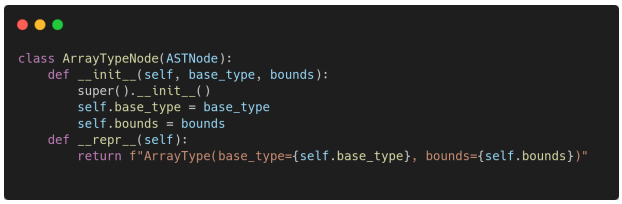
Class	Penjelasan Singkat
	**Kelas ProgramNode** Mewakili program Pascal, berisi nama program, deklarasi, dan block utama.
	**Kelas BlockNode** Mewakili sekumpulan statement yang dieksekusi sebagai satu blok.
	**Kelas DeclarationsNode** Menyimpan daftar deklarasi (const, var, type).

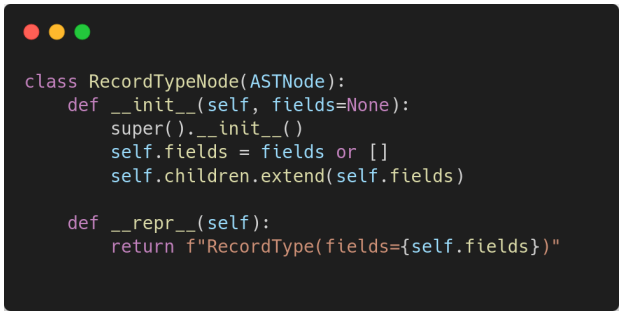
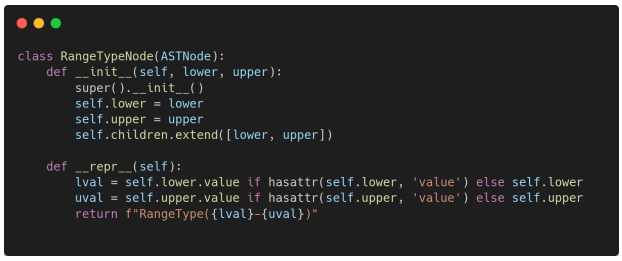
	**Kelas ConstDeclNode** Deklarasi konstanta dengan nama dan nilai bertipe tertentu.
	**Kelas VarDeclNode** Deklarasi variabel dengan nama dan tipe variabel.
	**Kelas TypeDeclarationsNode** Deklarasi tipe baru dengan nama tipe dan node tipe terkait.
	**Kelas AssignNode** Mewakili operasi assignment target dengan value.

	**Kelas VarNode** Mewakili referensi variabel berdasarkan nama.
	**Kelas NumNode** Literal angka.
	**Kelas StringNode** Literal string.
	**Kelas BooleanNode** Literal boolean.

	**Kelas UnaryOpNode** Operasi unary seperti negasi dan kebalikan.
	Operasi biner seperti aritmetika atau perbandingan.
	Deklarasi prosedur dengan nama, parameter, dan isi block.
	Deklarasi fungsi dengan parameter, tipe kembali, dan block.

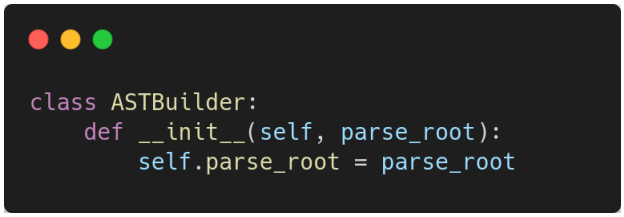
	Parameter pada fungsi/prosedur, bisa bernilai `var`-parameter.
	Pemanggilan fungsi atau prosedur dengan argumen.
	Struktur if condition berisi kondisi, then-block, dan optional else-block.
	Struktur while loop berisi kondisi dan body loop.
	Struktur repeat until loop berisi body dan kondisi penghentian.

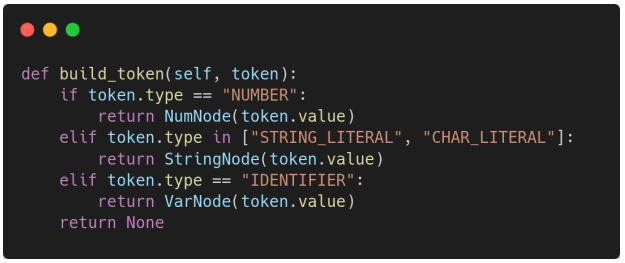
	**Kelas ForNode** Struktur for loop dengan variabel iterasi, batas awal/akhir, arah, dan body.
	**Kelas CaseBranchNode** Satu cabang kasus, berisi daftar konstanta dan statement yang dijalankan.
	**Kelas CaseNode** Struktur kasus dengan ekspresi kondisi dan kumpulan cabang.
	**Kelas ArrayTypeNode** Tipe array berisi tipe dasar dan rentang indeks.
	**Kelas ArrayAccessNode** Node akses elemen array dan index

	**Kelas RecordFieldNode** Satu field dalam tipe record dengan nama dan tipe.
	**Kelas UnaryOpNode** Tipe record berisi daftar field.
	**Kelas RangeTypeNode** Tipe rentang.

4.3 AST Builder

Kelas ini mengandung semua fungsi yang digunakan dalam membangun AST.

Functions	Penjelasan Singkat
	**Kelas ASTBuilder** Konstruktor ASTBuilder yang menerima akar parse tree (parse_root). Fungsi ini menyimpan parse root sebagai atribut kelas agar dapat digunakan saat membangun AST.
	**Fungsi build** Fungsi utama untuk memulai proses pembangunan AST. Ia memanggil build_node pada parse root dan mengembalikan node AST yang

	merepresentasikan keseluruhan program.
	**Fungsi build_node** Fungsi pengendali (controller) untuk Syntax Directed Translation (SDT). Memeriksa tipe node parse tree, memanggil builder spesifik jika node merupakan non-terminal, atau build_token jika terminal. Jika tidak ada builder, ia membuat BlockNode dari child nodes sebagai fallback.
	**Fungsi build_token** Membuat AST node dari token terminal. Token bertipe NUMBER menjadi NumNode, STRING_LITERAL/CHAR_LITERAL menjadi StringNode, dan IDENTIFIER menjadi VarNode. Token lain diabaikan.
	**Fungsi build_number_node** Mengambil token NUMBER dari child node dan membungkusnya menjadi NumNode. Digunakan untuk membangun AST dari ekspresi numerik.
	**Fungsi build_string_node** Mengambil token literal string dari child node dan membungkusnya menjadi StringNode. Berguna untuk literal string di program.

```
def build_variable_node(self, node):
    base_token = next((t for t in node.child if isinstance(t, Token) and t.type == "IDENTIFIER"), None)
    if not base_token:
        return None

    current_node = VarNode(base_token.value)

    # Traversal child untuk field atau array index
    i = 0
    while i < len(node.child):
        c = node.child[i]

        # Skip identifier pertama yang sudah diambil
        if isinstance(c, Token) and c.value == base_token.value and i == 0:
            pass

        # DOT -> akses field
        elif isinstance(c, Token) and c.type == "DOT":
            next_child = node.child[i] if i < len(node.child) else None
            if next_child and isinstance(next_child, Token) and next_child.type == "IDENTIFIER":
                current_node = RecordFieldNode(next_child.value, current_node)
                i += 1


        # <variable-index> -> akses array
        elif getattr(c, "type", None) == "<variable-index>":
            expr_node = next((self.build_node(x) for x in c.child if getattr(x, "type", None) == "<expression>"), None)
            if expr_node:
                current_node = ArrayAccessNode(current_node, expr_node)

        i += 1

    return current_node
```

Fungsi build_variable_node

Membuat node variabel yang dapat menangani identifier dasar, field record (dengan DOT), dan akses array (<variable-index>). Traversal dilakukan untuk menggabungkan akses bertingkat menjadi AST yang lengkap.

```
def build_statement_list(self, nodes):
    statements = []
    for n in nodes:
        node = self.build_node(n)
        if node:
            if isinstance(node, BlockNode):
                statements.extend(node.children)
            else:
                statements.append(node)
    return BlockNode(statements) if len(statements) > 1 else (statements[0] if statements else None)
```

Fungsi build_statement_list

Membuat BlockNode dari list child nodes, memastikan jika terdapat nested block, child-nya di-flatten agar menghasilkan daftar statement linear.

```
def build_program_node(self, node):
    prog_name = None
    decls_attr = []
    block_attr = None

    for c in node.child:
        if c.type == "<program-header>":
            token = next((t for t in c.child if isinstance(t, Token) and t.type == "IDENTIFIER"), None)
            if token:
                prog_name = token.value
        elif c.type == "<declaration-part>":
            decl_node = self.build_node(c)
            decls_attr = [decl_node] if decl_node else []
        elif c.type == "<compound-statement>":
            block_attr = self.build_node(c)

    return ProgramNode(prog_name, decls_attr, block_attr)
```

Fungsi build_program_node

Membuat ProgramNode dari parse node <program>. Mengekstrak nama program dari <program-header>, deklarasi dari <declaration-part>, dan body dari <compound-statement>.

```
def build_declarations_node(self, node):
    decls = []
    for c in node.child:
        child_decl = self.build_node(c)
        if child_decl:
            if isinstance(child_decl, list):
                decls.extend(child_decl)
            else:
                decls.append(child_decl)
    return DeclarationsNode(decls)
```

Fungsi build_declarations_node

Menggabungkan berbagai deklarasi (const, var, type) menjadi DeclarationsNode. Menangani child yang bisa berupa list atau single node.

```python
def build_const_decl_node(self, node):
    consts = []
    i = 0
    children = node.child

    while i < len(children):
        c = children[i]
        if isinstance(c, Token) and c.type == "IDENTIFIER":
            name = c.value
            j = i + 1
            while j < len(children) and children[j].type != "<expression>":
                j += 1
            expr_node = None
            if j < len(children) and children[j].type == "<expression>":
                expr_node = self.build_node(children[j])
            else:
                expr_node = None

            const_node = ConstDeclNode(name, expr_node)
            consts.append(const_node)

            k = j + 1
            while k < len(children) and not (isinstance(children[k], Token) and
children[k].type == "SEMICOLON"):
                k += 1
            i = k + 1
            continue

        i += 1

    return consts
```

Fungsi build_const_decl_node

Membuat daftar ConstDeclNode dari parse node <const-declaration>. Mengambil identifier, mengekstrak ekspresi terkait, dan mengabaikan semicolon.

```python
def build_var_decl_node(self, node):
    declarations = []
    identifiers_nodes = [c for c in node.child if c.type == "<identifier-list>"]
    type_nodes = [c for c in node.child if c.type == "<type>"]

    for id_node, type_node in zip(identifiers_nodes, type_nodes):
        vartype = self.build_type_definition_node(type_node)

        for item in id_node.child:
            if item.type == "IDENTIFIER":
                declarations.append(VarDeclNode(item.value, vartype))
    return declarations
```

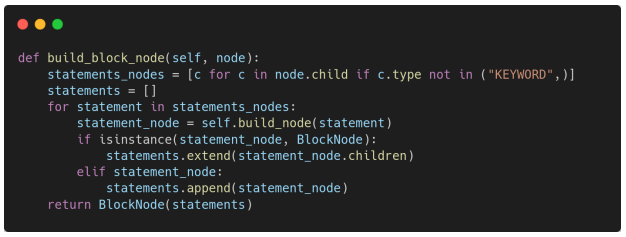
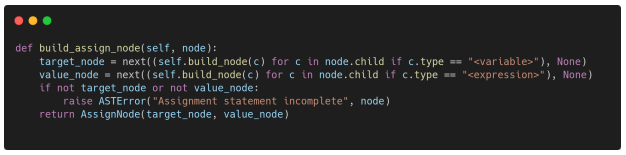
Fungsi build_var_decl_node

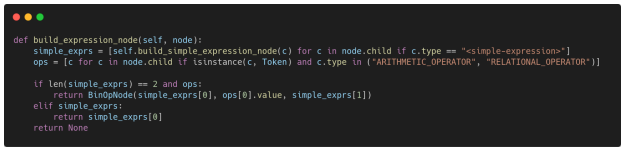
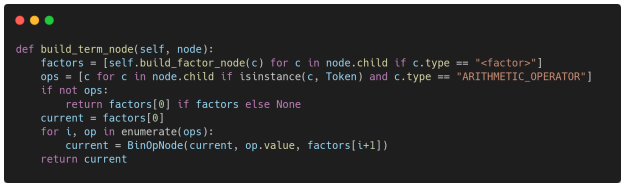
Membuat daftar VarDeclNode dari <var-declaration>. Mengambil <identifier-list> dan tipe, kemudian menggabungkannya untuk tiap variabel yang dideklarasikan.

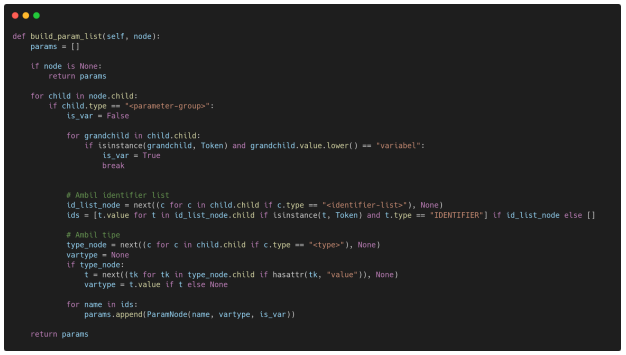
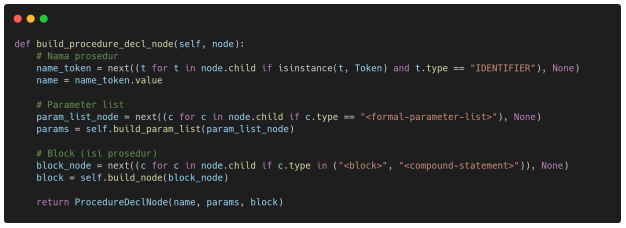
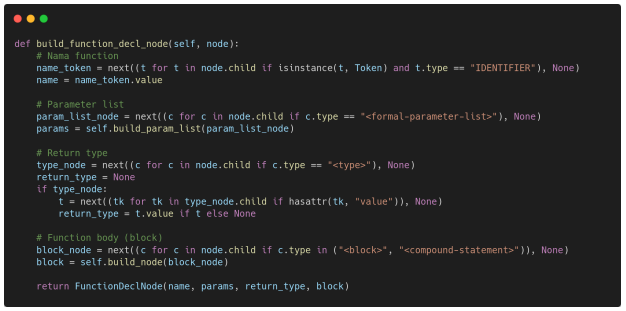
```python
def build_type_definition_node(self, node):
    if node is None:
        return None

    # handle token
    if isinstance(node, Token):
        if node.type in ("IDENTIFIER", "KEYWORD"):
            return node.value
        return None

    # handle range
    expr_nodes = [c for c in getattr(node, "child", []) if getattr(c, "type", None) == "<expression>"]
    range_op = next((c for c in getattr(node, "child", []) if isinstance(c, Token) and c.type == "RANGE_OPERATOR"), None)
    if len(expr_nodes) == 2 and range_op:
        lower = self.build_node(expr_nodes[0])
        upper = self.build_node(expr_nodes[1])
        return RangeTypeNode(lower, upper)

    # handle record / array / primitive
    for c in getattr(node, "child", []):
        if getattr(c, "type", None) == "<record-type>":
            return self.build_record_type_node(c)
        if getattr(c, "type", None) == "<array-type>":
            bounds = []
            range_node = next((x for x in c.child if getattr(x,"type",None)=="<range>"), None)
            if range_node:
                lower = self.build_node(range_node.child[0])
                upper = self.build_node(range_node.child[2])

                bounds.append((lower, upper))
            base_type_node = next((x for x in c.child if getattr(x,"type",None)=="<type>"), None)
            base_type = self.build_type_definition_node(base_type_node) if base_type_node else None
            return ArrayTypeNode(base_type, bounds)
        if isinstance(c, Token) and c.type in ("IDENTIFIER","KEYWORD"):
            return c.value
        res = self.build_type_definition_node(c)
        if res:
            return res

    return None
```

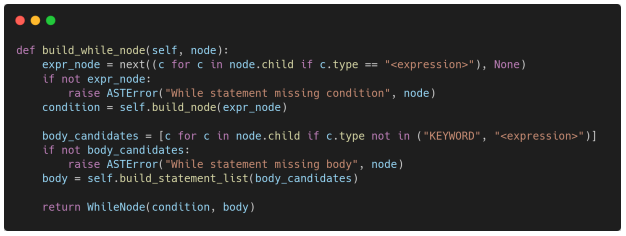
Fungsi build_type_definition_node

Mengonversi node tipe parse tree menjadi tipe AST. Menangani primitive, range, record, array, dan rekursi untuk nested type.

	**Fungsi build_type_declaration_node** Membuat daftar TypeDeclarationNode dari <type-declaration>. Menyimpan definisi tipe ke type_table untuk referensi internal.
	**Fungsi build_record_type_node** Membuat RecordTypeNode yang berisi field-field. Field diekstrak dari <parameter-group> dengan identifier dan tipe masing-masing, lalu dibungkus menjadi RecordFieldNode.
	**Fungsi build_block_node** Membuat BlockNode dari <compound-statement> dengan mengekstrak semua statement child. Nested block di-flatten agar menghasilkan daftar statement linear.
	**Fungsi build_variable_access** Membuat node akses variabel alternatif (mirip build_variable_node). Menangani identifier, akses array, dan field record.
	**Fungsi build_type_definition_node** Membuat AssignNode dari <assignment-statement>. Menentukan target (variabel) dan value (ekspresi). Jika salah satu tidak lengkap, menimbulkan ASTError.

	**Fungsi build_expression_node** Membuat node ekspresi dari <expression>. Mengambil simple expressions dan operator, membangun BinOpNode jika ada dua operand, atau mengembalikan single operand jika tunggal.
	**Fungsi build_simple_expression_node** Membuat node simple expression dari <simple-expression>. Menangani unary dan binary operator, looping manual untuk menggabungkan term menjadi BinOpNode atau UnaryOpNode.
	**Fungsi build_term_node** Membuat node term dari <term>. Menggabungkan faktor dengan operator aritmatika menjadi BinOpNode.
	**Fungsi build_factor_node** Membuat node faktor dari <factor>. Menangani unary (tidak), literal boolean, logical operator (dan/atau), parenthesis, dan fallback ke child pertama.

	**Fungsi build_param_list** Membuat daftar parameter (ParamNode) dari <formal-parameter-list>. Mengecek modifier variabel, identifier list, dan tipe parameter.
	**Fungsi build_procedure_decl_node** Membuat ProcedureDeclNode dari parse node procedure. Mengekstrak nama, parameter list, dan body block.
	**Fungsi build_function_decl_node** Membuat FunctionDeclNode dari parse node function. Mengekstrak nama, parameter list, return type, dan body block.
	**Fungsi build_procedure_function_call_node** Membuat ProcedureFunctionCallNode. Mengekstrak nama prosedur/function dan argumen dari <parameter-list>.
	**Fungsi build_if_node** Membuat IfNode dari <if-statement>. Menentukan kondisi dari ekspresi antara jika dan maka, lalu membuat then-block dan optional else-block.

	**Fungsi build_case_node** Membuat CaseNode dari <case-statement>. Mengambil ekspresi utama, lalu membangun daftar CaseBranchNode dengan value dan statement.
	**Fungsi build_while_node** Membuat WhileNode dari <while-statement>. Mengekstrak kondisi dan body statements. Menimbulkan ASTError jika salah satu tidak ada.
	**Fungsi build_repeat_node** Membuat RepeatNode dari <repeat-statement>. Mengekstrak body statements dan kondisi, memvalidasi kelengkapan.
	**Fungsi for_node** Membuat ForNode dari <for-statement>. Mengekstrak variabel loop, start/end expression, arah iterasi (ke/turun_ke), dan body.

4.4 Symbol Table

Kelas ini merupakan otak atau memori dari compiler itu sendiri, segala variabel yang dideklarasikan akan tersimpan pada tabel ini, dalam bentuk data tertentu mencakup berbagai informasi.

Class	Penjelasan Singkat
	**Kelas SymbolTable** memiliki 3 tabel yang terpisah : tab : tabel utama yang menyimpan seluruh deklarasi variabel dari nama , jenis, tipe, level, dan alamat memori relatif. tabel ini bentuknya list panjang yang cara kerjanya seperti linked list, menghubungkan variabel-variabel dalam satu fungsi yang sama btab : tabel ini mengelola informasi per blok atau fungsi , isinya berupa indeks variabel terakhir yang ada di tab milik fungsinya, psze yang merupakan jumlah parameter/argumen yang dimiliki fungsi, dan vsze yang merupakan jumlah memori yang dibutuhkan untuk variabel lokal di dalam fungsi ini atab: berisi informasi mengenai suatu array dari batas bawah (low), batas atas (high) , tipe elemen (etyp), dan total ukuran memori (size)
	enter_scope dipanggil saat masuk ke scope baru seperti deklarasi fungsi/prosedur baru, current_level akna increment naik 1 tingkat (basenya 0 di global), disini juga bakal bikin btab baru karena masuk ke block baru milik fungsi / preosedur. exist_scope dipanggil saat keluar ruangan , saat parser menemukan end, current_level decrement. btab disimpan dan tidak dihapus sebagai memori.

```python
# Add Entries
def add_variable(self, name: str, type_kind: int, size: int = 1):
    """Tambah variabel lokal"""
    current_btab_idx = self.display[self.current_level]
    current_block = self.btab[current_btab_idx]

    # Linkes List & adds
    link_idx = current_block["last"]
    adr = current_block["vsze"]

    new_entry = {
        "name": name,
        "link": link_idx,
        "obj": ObjKind.VARIABLE,
        "type": type_kind,
        "ref": 0,
        "nrm": 1,
        "lev": self.current_level,
        "adr": adr
    }
    self.tab.append(new_entry)

    # Update block
    new_id_idx = len(self.tab) - 1
    current_block["last"] = new_id_idx
    current_block["vsze"] += size
    return new_id_idx

def add_parameter(self, name: str, type_kind: int, is_var: bool = False):
    """Tambah parameter prosedur/fungsi"""
    current_btab_idx = self.display[self.current_level]
    current_block = self.btab[current_btab_idx]

    link_idx = current_block["last"]
    nrm_val = 0 if is_var else 1

    new_entry = {
        "name": name,
        "link": link_idx,
        "obj": ObjKind.VARIABLE,
        "type": type_kind,
        "ref": 0,
        "nrm": nrm_val,
        "lev": self.current_level,
        "adr": current_block["psze"]
    }
    self.tab.append(new_entry)

    # Update Blok (last & psze)
    new_id_idx = len(self.tab) - 1
    current_block["last"] = new_id_idx
    current_block["lsar"] = new_id_idx
    current_block["psze"] += 1
    return new_id_idx

def add_program_name(self, name: str):
    """
    Khusus untuk nama program: Masukkan ke tabel tapi JANGAN update 'last'.
    Agar variabel pertama (a) nanti link-nya ke 0, bukan ke Hello.
    """
    new_entry = {
        "name": name,
        "link": 0,
        "obj": ObjKind.PROGRAM,
        "type": TypeKind.NOTYPE,
        "ref": 0,
        "nrm": 1,
        "lev": 0,
        "adr": 0
    }
    self.tab.append(new_entry)
    return len(self.tab) - 1

def add_constant(self, name: str, type_kind: int, value: Any):
    """Add konstanta"""
    current_btab_idx = self.display[self.current_level]
    current_block = self.btab[current_btab_idx]
    link_idx = current_block["last"]

    new_entry = {
        "name": name,
        "link": link_idx,
        "obj": ObjKind.CONSTANT,
        "type": type_kind,
        "ref": 0,
        "nrm": 1,
        "lev": self.current_level,
        "adr": value
    }
    self.tab.append(new_entry)

    new_id_idx = len(self.tab) - 1
    current_block["last"] = new_id_idx
    return new_id_idx

def add_array_info(self, xtyp: int, etyp: int, low: int, high: int, elsz: int):
    """Menambah Info Array ke ATAB"""
    size = (high - low + 1) * elsz
    entry = {
        "index": len(self.atab) + 1,
        "xtyp": xtyp,
        "etyp": etyp,
        "eref": 0,
        "low": low,
        "high": high,
        "elsz": elsz,
        "size": size
    }
    self.atab.append(entry)
    return len(self.atab) - 1

def add_procedure(self, name: str, kind: str):
    """Tambah nama fungsi / prosedur di scope saat ini"""
    current_btab_idx = self.display[self.current_level]
    current_block = self.btab[current_btab_idx]

    link_idx = current_block["last"]

    new_entry = {
        "name": name,
        "link": link_idx,
        "obj": kind,
        "type": TypeKind.NOTYPE, # Nanti diupdate kala function (return type)
        "ref": 0,
        "nrm": 1,
        "lev": self.current_level,
        "adr": 0
    }
    self.tab.append(new_entry)

    new_id_idx = len(self.tab) - 1
    current_block["last"] = new_id_idx
    return new_id_idx

def add_builtin_procedure(self, name: str):
    """Tambah prosedur built-in (predefined) ke symbol table at global scope"""
    global_block = self.btab[0]
    link_idx = 0

    new_entry = {
        "name": name,
        "link": link_idx,
        "obj": ObjKind.PROCEDURE,
        "type": TypeKind.NOTYPE,
        "ref": 0,
        "nrm": 1,
        "lev": 0,
        "adr": 0
    }
    self.tab.append(new_entry)
    new_id_idx = len(self.tab) - 1

    return new_id_idx

def add_type(self, name: str, type_kind: int):
    current_btab_idx = self.display[self.current_level]
    current_block = self.btab[current_btab_idx]
    link_idx = current_block["last"]
    new_entry = {
        "name": name,
        "link": link_idx,
        "obj": ObjKind.TYPE,
        "type": type_kind,
        "ref": 0,
        "nrm": 1,
        "lev": self.current_level,
        "adr": 0
    }
    self.tab.append(new_entry)
    new_id_idx = len(self.tab) - 1
    current_block["last"] = new_id_idx
    return new_id_idx
```

add_variable & add_parameter:

fungsi untuk mendaftarkan variabel/parameter, otomatis menghitung alamat memori (adr) dan mengupdate ukuran memori yang dibutuhkan fungsi tersebut (vsze / psze) di btab

add_constant:

sama, bedanya adr tidak dipakai untuk alamat memori, tapi untuk menyimpan nilai konstanta nya langsung

add_array_info:

Khusus array, detailnya (batas bawah, atas, tipe elemen) disimpan terpisah di tabel atab, bukan tab utama

add_procedure & add_program_name:

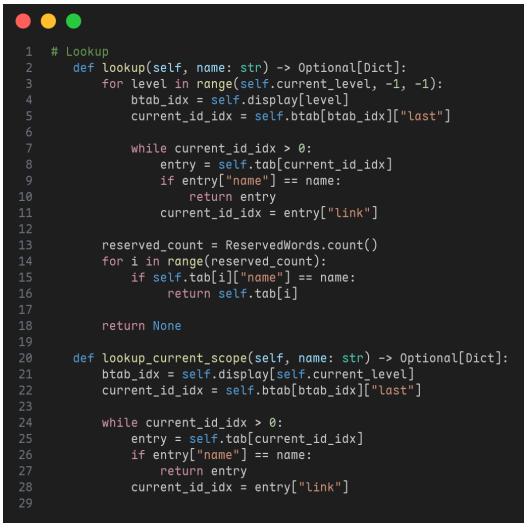
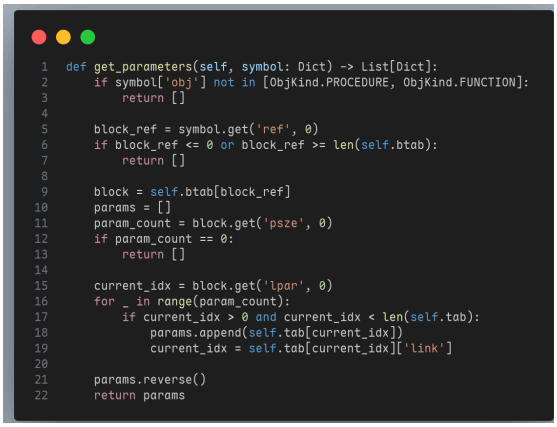
Mendaftarkan nama fungsi/program itu sendiri sebagai head dari sebuah blok kode

add_builtin_procedure:

mendaftarkan fungsi bawaan seperti write, read secara paksa di level 0 (global) agar bisa dipanggil dari mana saja.

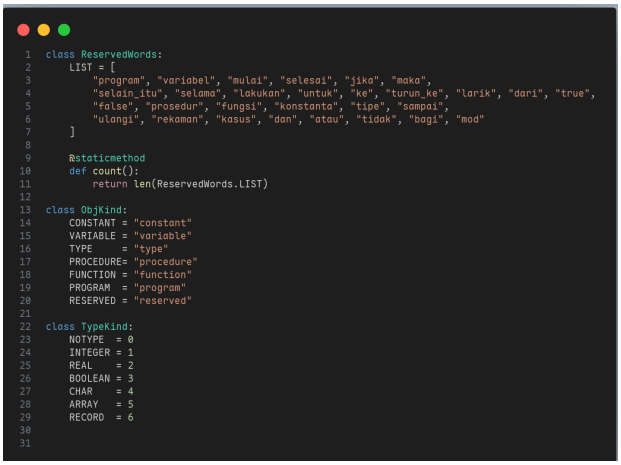
add_type:

Digunakan untuk menangani user-defined type, jadi declare type sendiri bukan built-in dari pascal-s nya.

	**lookup:** Fungsi ini digunakan saat kita menggunakan variabel di dalam kode (misal x := y + 1). compiler harus mencari definisi variabel y. mulai dari current_level, cek parent jika tidak ketemu, cek global jika tidak ketemu, cek reserved words jika masih tidak ketemu. intinya akan terus mundur ke level di atasnya. **lookup_current_scope**: Fungsi ini digunakan saat kita mendeklarasikan variabel baru. Jadi cara kerjanya ia melihat dan mengecek apakah sudah ada variabel yang sama dideklarasikan di scope yang sama.berbeda dengan lookup dia tidak loop mundur ke level induknya,
	**get_parameters:** Fungsi yang bertugas untuk mengambil daftar parameter (argumen) yang dimiliki oleh sebuah prosedur atau fungsi. Misal untuk suatu fungsi Hitung (a:int,b:int), fungsi inilah yang akan mereturn informasi mengenai parameter pada Hitung pada kasus ini adalah a dan b.

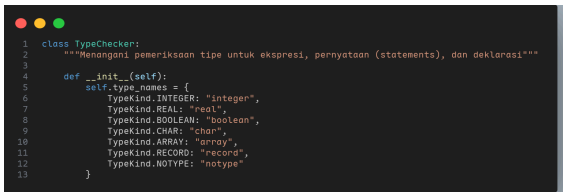
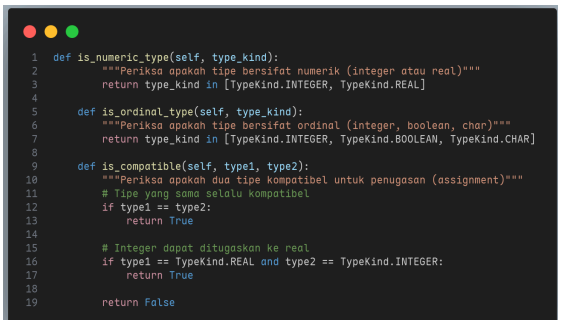
4.5 Constants

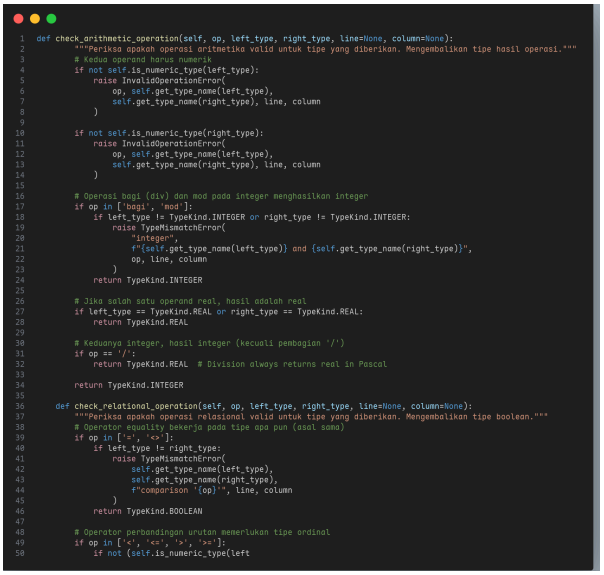
Terdapat 3 kelas yang masing2 bertugas untuk menyimpan suatu hardcoded constants yang akan digunakan nantinya pada symbol table untuk keperluan declaration dan type checking

Class	Penjelasan Singkat
	**Class ReservedWords:** merupakan daftar konstanta identifier list yang digunakan agar user tidak membuat variabel dengan nama yang sudah di reserved pada list **Class objKind:** merupakan kelas yang disimpan untuk menyimpan identifier yang berupa suatu jenis objek yang mencakup konstanta, variabel , prosedur/fungsi, tipe, dan program. **Class TypeKind** merupakan kelas yang digunakan untuk menyimpen tipe data builtin pada program yang mencakup tipe data primitif seperti integer, real, boolean, char, dan juga tipe data kompleks/struktur seperti record dan array.

4.6 TypeChecker

Kelas ini digunakan untuk melakukan type check

Class	Penjelasan Singkat
	Class **type checker** bertugas memvalidasi keamanan tipe data (type safety) dalam compiler untuk mencegah illegal operation.didalam terdapat kamus dictionary bernama type_names yang berfungsi sebagai penerjemah kode numerik internal, menjadi teks yang mudah dibaca manusia seperti “integer” atau ‘real”.
	**get_field_type:** Fungsi dipanggil saat semantic analyzer menemukan syntax A.B, mengecek apakah A (record_type) benar-benar bertipe record yang lengkap. Jika A memang record, apakah diap unya field bernama B. **get_type_name:** bertugas menerjemahkkan code angka internal seperti 1, menjadi teks yang bisa dibaca manusia. (seperti “integer”)
	**is_numeric_type:** Mengecek apakah sebuah tipe bisa dipakai untuk numeric (tambah, kurang, kali bagi) **is_ordinal_type:** Mengecek apakah tipe data tersebut memiliki urutan yang pasti **is_compatible:** Mengecek apakah tipe2 boleh dimasukan ke dalam variabel bertipe tipe1


``python
def check_arithmetic_operation(self, op, left_type, right_type, line=None, column=None):
"""Periksa apakah operasi aritmetika valid untuk tipe yang diberikan. Mengembalikan tipe hasil operasi."""
# Kedua operand harus numerik 	
if not self.is_numeric_type(left_type):
raise InvalidOperationError(
op, self.get_type_name(left_type),
self.get_type_name(right_type), line, column
)
 	
if not self.is_numeric_type(right_type):
raise InvalidOperationError(
op, self.get_type_name(left_type),
self.get_type_name(right_type), line, column
)
 	
# Operasi bagi (div) dan mod pada integer menghasilkan integer 	
if op in ['bagi', 'mod']:
if left_type != TypeKind.INTEGER or right_type != TypeKind.INTEGER:
raise TypeMismatchError(
"integer",
f"{self.get_type_name(left_type)} and {self.get_type_name(right_type)}",
op, line, column
)
return TypeKind.INTEGER
 	
# Jika salah satu operand real, hasil adalah real 	
if left_type == TypeKind.REAL or right_type == TypeKind.REAL:
return TypeKind.REAL
 	
# Keduanya integer, hasil integer (kecuali pembagian '/') 	
if op == '/':
return TypeKind.REAL # Division always returns real in Pascal
 	
return TypeKind.INTEGER
 	
def check_relational_operation(self, op, left_type, right_type, line=None, column=None):
"""Periksa apakah operasi relasional valid untuk tipe yang diberikan. Mengembalikan tipe boolean."""
# Operator equality bekerja pada tipe apa pun (asal sama) 	
if op in ['=', '<>']:
if left_type != right_type:
raise TypeMismatchError(
self.get_type_name(left_type),
self.get_type_name(right_type),
f"comparison '{op}'", line, column
)
return TypeKind.BOOLEAN
 	
# Operator perbandingan urutan memerlukan tipe ordinal 	
if op in ['<', '<=', '>', '>=']:
if not (self.is_numeric_type(left
``
	check_arithmetic_operation:
	Fungsi dipanggil saat ada operasi tambah, kurang, kali , bagi atau mod
	
	check_relational_operation:
	Fungsi ini dipanggil saat ada operasi =, , , > , <>. Tujuannya selalu mengembalikan tipe boolean (benar / salah)


``python
def get_result_type(self, op, left_type, right_type, line=None, column=None):
"""Dapatkan tipe hasil dari operasi biner, menggabungkan seluruh pemeriksaan tipe operasi."""
# Operator aritmetika 	
if op in ['+', '-', '*', '/', 'bagi', 'mod']:
return self.check_arithmetic_operation(op, left_type, right_type, line, column)
 	
# Operator relasional 	
if op in ['=', '<>', '<', '<=', '>', '>=']:
return self.check_relational_operation(op, left_type, right_type, line, column)
 	
# Operator logika 	
if op in ['dan', 'and', 'atau', 'or']:
return self.check_logical_operation(op, left_type, right_type, line, column)
 	
raise InvalidOperationError(op, self.get_type_name(left_type),
self.get_type_name(right_type), line, column)
 	
``
	Bertugas sebagai dispatcher untuk menentukan tipe data hasil dari binary operation. Fungsi ini mendeteksi jenis operator yang digunakan (aritmatika?, relational? , ato logika), lalu dipper ke tugas validasi ke fungsi spesifik yang sesuai seperti check_arithmetic, check_relational, atau check_logical). Jika operator ga recognize, fungsi ini akan langsung throw error InvalidOperationError.

4.7 Errors

Kelas ini digunakan untuk mengecek error

Class	Penjelasan Singkat
	**SemanticError:** Base class semua error semantik **UndeclaredIdentifierError:** pakai variabel x tapi blom di deklarasi **RedeclaredIdentifierError:** deklarasi x , trus di bawah deklarasi lagi di scope yang sama **TypeMismatchError:** Operasi matematika tapi ga nyambung ama tipenya malah string **InvalidAssignmentError:** Mengubah nilai konstanta atau variabel yang gabole diubah manually **InvalidOperationError:** Operasi tidak masuk akal , (“theo” - “learning”) **NotAnArrayError:** coba meng-indeks variabel x, tapi x itu integer **InvalidArrayIndexError:** akses array pakai tipe data aneh, contoh : myarr[“baby”] **NotAFunctionError/NotAProcedureError:** variabel x itu integer , tapi panggil x() **ArgumentCountError:** Fungsi butuh 2 argumen, malah masukin 1 **ReturnTypeMismatchError:** Bilangnya return Integer, tapi didalam return “theo” **MissingReturnError:** Fungsi kelar tapi lupa return

4.8 Semantic Analyzer

Kelas ini babi haram

Class	Penjelasan Singkat
	Kelas SemanticAnalyzer ini bertujuan untuk mengemas (enkapsulasi) fungsi-fungsi yang terlibat dalam analisis semantik Pascal-S.
	**__init__()**: Inisialisasi analyzer dengan symbol table, type checker, list error, dan set builtin functions **analyze(ast_root)**: Entry point analisis semantik, menjalankan visitor dan mengembalikan status (sukses/gagal) + list error **report_error(error)**: Menambahkan error ke list errors **visit(node)**: Dispatcher pattern untuk memanggil visitor method sesuai tipe node (misal: visit_ProgramNode) **generic_visit(node)**: Fallback visitor yang mengunjungi semua children node secara rekursif

```python
def visit_ProgramNode(self, node):
    # Tambahkan nama program ke symbol table
    if node.name:
        prog_idx = self.symbol_table.add_program_name(node.name)
        # Dekorasi node
        node.attr['tab_index'] = prog_idx
        node.attr['type'] = TypeKind.NOTYPE
        node.attr['lev'] = 0

    # Kunjungi bagian deklarasi
    if node.declarations:
        for decl in node.declarations:
            self.visit(decl)

    # Kunjungi blok utama - enter scope untuk compound statement
    if node.block:
        block_idx = self.symbol_table.enter_scope()
        # Dekorasi block node
        node.block.attr['block_index'] = block_idx
        node.block.attr['lev'] = self.symbol_table.current_level

        self.visit(node.block)
        self.symbol_table.exit_scope()

def visit_BlockNode(self, node):
    """Kunjungi blok (compound statement)"""
    for statement in node.statements:
        if statement:
            self.visit(statement)

def visit_DeclarationsNode(self, node):
    """Kunjungi node deklarasi"""
    for decl in node.declarations:
        if decl:
            self.visit(decl)
```

visit_ProgramNode(node): Proses node program utama, tambah nama program ke symbol table, kunjungi deklarasi & blok utama

visit_BlockNode(node): Kunjungi compound statement (blok begin..end), iterasi semua statements di dalamnya

visit_DeclarationsNode(node): Kunjungi bagian deklarasi (konstanta, variabel, tipe, prosedur/fungsi)

```python
def visit_ConstDeclNode(self, node):
    """Kunjungi deklarasi konstanta"""
    # Periksa apakah sudah dideklarasikan di scope saat ini
    existing = self.symbol_table.lookup_current_scope(node.name)
    if existing:
        self.report_error(RedeclaredIdentifierError(node.name))
        return

    # Tentukan tipe dari ekspresi nilai konstanta
    type_kind = TypeKind.NOTYPE
    if node.consttype:
        # Kunjungi ekspresi untuk memperoleh tipenya
        type_kind = self.visit(node.consttype)

    # Tambahkan ke tabel simbol
    self.symbol_table.add_constant(node.name, type_kind, None)

def visit_VarDeclNode(self, node):
    existing = self.symbol_table.lookup_current_scope(node.name)
    if existing:
        self.report_error(RedeclaredIdentifierError(node.name))
        return

    # Handle variable of a user-defined RECORD type
    if isinstance(node.vartype, RecordTypeNode):
        # cek tipe record di symbol table jika ada type_name
        type_name = getattr(node.vartype, 'name', None)
        type_kind = TypeKind.RECORD

        var_idx = self.symbol_table.add_variable(node.name, type_kind)
        node.attr['tab_index'] = var_idx
        node.attr['type'] = type_kind
        node.attr['lev'] = self.symbol_table.current_level

        # isi fields
        fields = {}
        if hasattr(node.vartype, 'fields'):
            for f in node.vartype.fields:
                # gunakan _get_type_kind tapi jika f.type_ adalah user-defined record, ambil fields
                ft = self._get_type_kind(f.type_)
                # Jika user-defined record, ambil fields dari symbol table
                if ft == TypeKind.RECORD and isinstance(f.type_, str):
                    rec_symbol = self.symbol_table.lookup(f.type_)
                    if rec_symbol and 'fields' in rec_symbol:
                        fields[f.name] = rec_symbol['fields']
                    else:
                        fields[f.name] = TypeKind.NOTYPE
                else:
                    fields[f.name] = ft
        node.attr['fields'] = fields
        self.symbol_table.tab[var_idx]['fields'] = fields
        return

    # Handle array
    if getattr(node.vartype, '__class__', None).__name__ == 'ArrayTypeNode':
        base_type = self._get_type_kind(node.vartype.base_type)
        low, high = 1, 10
        if hasattr(node.vartype, 'bounds') and node.vartype.bounds:
            b = node.vartype.bounds[0]
            if isinstance(b, tuple) and len(b) == 2:
                low = int(getattr(b[0], 'value', 1))
                high = int(getattr(b[1], 'value', 10))

        var_idx = self.symbol_table.add_variable(node.name, TypeKind.ARRAY)
        atab_idx = self.symbol_table.add_array_info(
            xtyp=TypeKind.ARRAY, etyp=base_type, low=low, high=high, elsz=1
        )
        self.symbol_table.tab[var_idx]['ref'] = atab_idx
        node.attr['tab_index'] = var_idx
        node.attr['type'] = TypeKind.ARRAY
        node.attr['lev'] = self.symbol_table.current_level
        return

    # Tipe sederhana
    type_kind = self._get_type_kind(node.vartype)
    var_idx = self.symbol_table.add_variable(node.name, type_kind)
    node.attr['tab_index'] = var_idx
    node.attr['type'] = type_kind
    node.attr['lev'] = self.symbol_table.current_level

def visit_TypeDeclarationNode(self, node):
    """Kunjungi deklarasi tipe"""
    # Periksa apakah sudah dideklarasikan di scope saat ini
    existing = self.symbol_table.lookup_current_scope(node.name)
    if existing:
        self.report_error(RedeclaredIdentifierError(node.name))
        return

    # Ambil jenis tipe dari node tipe
    type_kind = self._resolve_type_node(node.type_node)

    # Tambahkan ke tabel simbol
    type_idx = self.symbol_table.add_type(node.name, type_kind)

    # Jika ini RecordTypeNode, simpan field info ke symbol table entry
    if isinstance(node.type_node, RecordTypeNode):
        # _resolve_type_node sudah mengisi node.type_node.attr['fields']
        field_info = node.type_node.attr.get('fields', {})
        if type_idx is not None and 0 <= type_idx < len(self.symbol_table.tab):
            self.symbol_table.tab[type_idx]['fields'] = field_info
```

visit_ConstDeclNode(node): Validasi deklarasi konstanta: cek redeclaration, tentukan tipe dari nilai, tambah ke symbol table

visit_VarDeclNode(node): Validasi deklarasi variabel: cek redeclaration, handle tipe sederhana/array/record, tambah ke symbol table & ATAB

visit_TypeDeclarationNode(node): Validasi deklarasi tipe user-defined: cek redeclaration, resolve tipe (record/array/range), tambah ke symbol table

```python
def visit_ProcedureDeclNode(self, node):
        """Kunjungi deklarasi prosedur"""
        # Periksa apakah sudah dideklarasikan di scope saat ini
        existing = self.symbol_table.lookup_current_scope(node.name)
        if existing:
            self.report_error(RedeclaredIdentifierError(node.name))
            return

        # Tambahkan prosedur ke tabel simbol
        proc_idx = self.symbol_table.add_procedure(node.name, ObjKind.PROCEDURE)

        # Masuk ke scope prosedur
        block_idx = self.symbol_table.enter_scope()

        # Kaitkan entri prosedur ke bloknya
        if proc_idx >= 0 and proc_idx < len(self.symbol_table.tab):
            self.symbol_table.tab[proc_idx]['ref'] = block_idx

        # Tambahkan parameter
        if node.params:
            for param in node.params:
                self.visit(param)

        # Kunjungi badan prosedur
        if node.block:
            self.visit(node.block)

        # Keluar dari scope prosedur
        self.symbol_table.exit_scope()

    def visit_FunctionDeclNode(self, node):
        """Kunjungi deklarasi fungsi"""
        # Periksa apakah sudah dideklarasikan di scope saat ini
        existing = self.symbol_table.lookup_current_scope(node.name)
        if existing:
            self.report_error(RedeclaredIdentifierError(node.name))
            return

        # Ambil tipe kembalian
        return_type = self._get_type_kind(node.return_type)

        # Tambahkan fungsi ke tabel simbol
        func_idx = self.symbol_table.add_procedure(node.name, ObjKind.FUNCTION)

        # Perbarui entri fungsi dengan tipe kembalian
        self.symbol_table.tab[func_idx]['type'] = return_type

        # Lacak fungsi saat ini untuk pemeriksaan return
        prev_function = self.current_function
        self.current_function = (node.name, return_type)

        # Masuk ke scope fungsi
        block_idx = self.symbol_table.enter_scope()

        # Kaitkan entri fungsi ke bloknya
        if func_idx >= 0 and func_idx < len(self.symbol_table.tab):
            self.symbol_table.tab[func_idx]['ref'] = block_idx

        # Tambahkan parameter
        if node.params:
            for param in node.params:
                self.visit(param)

        # Kunjungi badan fungsi
        if node.block:
            self.visit(node.block)

        # Keluar dari scope fungsi
        self.symbol_table.exit_scope()

        # Kembalikan fungsi sebelumnya
        self.current_function = prev_function

    def visit_ParamNode(self, node):
        """Kunjungi node parameter"""
        type_kind = self._get_type_kind(node.type_node)
        is_var_flag = getattr(node, 'is_var', False)

        for name in node.names:
            self.symbol_table.add_parameter(name, type_kind, is_var_flag)
```

visit_ProcedureDeclNode(node): Proses deklarasi prosedur: cek redeclaration, enter scope baru, tambah parameter, kunjungi body, exit scope

visit_FunctionDeclNode(node): Proses deklarasi fungsi: sama seperti prosedur tapi dengan return type, track current function untuk validasi return

visit_ParamNode(node): Tambahkan parameter (formal parameter) ke symbol table dengan tipe & flag var/value parameter

```python
def visit_AssignNode(self, node):
    """Kunjungi pernyataan assignment"""
    target_type = self.visit(node.target)
    value_type = self.visit(node.value)

    # cek kesesuaian tipe
    try:
        self.type_checker.check_assignment(target_type, value_type)
    except SemanticError as e:
        self.report_error(e)

    node.attr['type'] = TypeKind.NOTYPE


def visit_IfNode(self, node):
    """Kunjungi pernyataan if"""
    # Periksa kondisi bertipe boolean
    condition_type = self.visit(node.condition)
    try:
        self.type_checker.check_condition(condition_type, "if statement")
    except SemanticError as e:
        self.report_error(e)

    # Kunjungi blok then
    if node.then_block:
        self.visit(node.then_block)

    # Kunjungi blok else jika ada
    if node.else_block:
        self.visit(node.else_block)

def visit_WhileNode(self, node):
    """Kunjungi pernyataan while"""
    # Periksa kondisi bertipe boolean
    condition_type = self.visit(node.condition)
    try:
        self.type_checker.check_condition(condition_type, "while statement")
    except SemanticError as e:
        self.report_error(e)

    # Kunjungi badan
    if node.body:
        self.visit(node.body)

def visit_RepeatNode(self, node):
    """Kunjungi pernyataan repeat-until"""
    # Kunjungi badan terlebih dahulu
    if node.body:
        self.visit(node.body)

    # Periksa kondisi bertipe boolean
    condition_type = self.visit(node.condition)
    try:
        self.type_checker.check_condition(condition_type, "repeat-until statement")
    except SemanticError as e:
        self.report_error(e)

def visit_ForNode(self, node):
    """Kunjungi pernyataan for"""
    # Ambil tipe variabel loop
    var_symbol = self.symbol_table.lookup(node.var_node.name)
    if not var_symbol:
        self.report_error(UndeclaredIdentifierError(node.var_node.name))
        return

    var_type = var_symbol['type']

    # Ambil tipe ekspresi awal dan akhir
    start_type = self.visit(node.start_expr)
    end_type = self.visit(node.end_expr)

    # Periksa batas kompatibel
    try:
        self.type_checker.check_for_loop_bounds(var_type, start_type, end_type)
    except SemanticError as e:
        self.report_error(e)

    # Kunjungi badan
    if node.body:
        self.visit(node.body)

def visit_ProcedureFunctionCallNode(self, node):
    """Kunjungi pemanggilan prosedur/fungsi"""
    # Periksa apakah ini built-in
    call_name = node.name.lower() if isinstance(node.name, str) else str(node.name).lower()

    if call_name in self.builtins:
        # Built-in menerima argumen apa pun, lewati pemeriksaan rinci
        if node.args:
            for arg in node.args:
                self.visit(arg)

        # Cek apakah builtin sudah ada di symbol table
        symbol = self.symbol_table.lookup(call_name)
        if not symbol:
            # Tambahkan builtin ke symbol table saat pertama kali digunakan
            tab_idx = self.symbol_table.add_builtin_procedure(call_name)
        else:
            # Cari tab_index untuk built-in yang sudah ada
            tab_idx = None
            for idx, entry in enumerate(self.symbol_table.tab):
                if entry.get('name') == call_name:
                    tab_idx = idx
                    break

        node.attr['tab_index'] = tab_idx
        node.attr['type'] = 'predefined'
        return TypeKind.NOTYPE

    # Cari entri prosedur/fungsi
    symbol = self.symbol_table.lookup(node.name)
    if not symbol:
        self.report_error(UndeclaredIdentifierError(node.name))
        return TypeKind.NOTYPE

    # Periksa bahwa simbol dapat dipanggil
    if symbol['obj'] not in [ObjKind.PROCEDURE, ObjKind.FUNCTION]:
        self.report_error(NotAProcedureError(node.name))
        return TypeKind.NOTYPE

    # Ambil parameter yang diharapkan
    expected_params = self.symbol_table.get_parameters(symbol)

    # Periksa jumlah argumen
    actual_count = len(node.args) if node.args else 0
    expected_count = len(expected_params)

    if actual_count != expected_count:
        self.report_error(ArgumentCountError(expected_count, actual_count, node.name))

    # Periksa tipe argumen
    if node.args:
        for i, (arg, expected_param) in enumerate(zip(node.args, expected_params)):
            arg_type = self.visit(arg)
            expected_type = expected_param['type']

            if not self.type_checker.is_compatible(expected_type, arg_type):
                self.report_error(ArgumentTypeError(
                    f"argument {i+1}",
                    self.type_checker.get_type_name(expected_type),
                    self.type_checker.get_type_name(arg_type)
                ))

    # Kembalikan tipe untuk fungsi
    if symbol['obj'] == ObjKind.FUNCTION:
        return symbol['type']

    return TypeKind.NOTYPE
```

visit_AssignNode(node): Validasi assignment: kunjungi target & value, cek kompatibilitas tipe menggunakan type checker

visit_IfNode(node): Validasi if statement: cek kondisi boolean, kunjungi then block & else block (jika ada)

visit_WhileNode(node): Validasi while loop: cek kondisi boolean, kunjungi body

visit_RepeatNode(node): Validasi repeat-until loop: kunjungi body dulu, lalu cek kondisi boolean

visit_ForNode(node): Validasi for loop: cek tipe variabel loop ordinal, validasi bounds (start/end) kompatibel, kunjungi body

visit_ProcedureFunctionCallNode(node): Validasi pemanggilan: handle builtin secara khusus, cek identifier ada & callable, validasi jumlah & tipe argumen

```python
def visit_BinOpNode(self, node):
    """Kunjungi operasi biner"""
    left_type = self.visit(node.left)
    right_type = self.visit(node.right)

    try:
        result_type = self.type_checker.get_result_type(
            node.op, left_type, right_type
        )
        # Dekorasi node
        node.attr['type'] = result_type
        return result_type
    except SemanticError as e:
        self.report_error(e)
        return TypeKind.NOTYPE

def visit_UnaryOpNode(self, node):
    """Kunjungi operasi unary"""
    operand_type = self.visit(node.operand)

    try:
        result_type = self.type_checker.check_unary_operation(node.op, operand_type)
        return result_type
    except SemanticError as e:
        self.report_error(e)
        return TypeKind.NOTYPE

def visit_VarNode(self, node):
    symbol = self.symbol_table.lookup(node.name)
    if not symbol:
        self.report_error(UndeclaredIdentifierError(node.name))
        return TypeKind.NOTYPE

    # Dekorasi node dengan info dari symbol table
    tab_idx = None
    for idx, entry in enumerate(self.symbol_table.tas):
        if entry.get('name') == node.name:
            tab_idx = idx
            break

    node.attr['tab_index'] = tab_idx
    node.attr['type'] = symbol['type']
    node.attr['lev'] = symbol.get('lev', 0)

    # Copy fields dari symbol table (untuk record)
    if 'fields' in symbol:
        node.attr['fields'] = symbol['fields']  # <<< ini penting

    return symbol['type']

def visit_ArrayAccessNode(self, node):
    """Kunjungi akses array (misal: arr[i])"""
    # Ambil variabel array
    if hasattr(node.array, 'name'):
        array_name = node.array.name
        symbol = self.symbol_table.lookup(array_name)

        if not symbol:
            self.report_error(UndeclaredIdentifierError(array_name))
            return TypeKind.NOTYPE

        # Periksa apakah memang bertipe array
        if symbol['type'] != TypeKind.ARRAY:
            self.report_error(SemanticError(f"'{array_name}' is not an array"))
            return TypeKind.NOTYPE

        # Periksa tipe indeks
        index_type = self.visit(node.index)
        if index_type != TypeKind.INTEGER:
            try:
                self.type_checker.check_array_index(index_type)
            except SemanticError as e:
                self.report_error(e)

        # Ambil tipe elemen dari ATAB
        ref = symbol.get('ref', -1)
        if ref >= 0 and ref < len(self.symbol_table.atab):
            array_info = self.symbol_table.atab[ref]
            return array_info['etyp']

        # Fallback: kembalikan INTEGER jika info array tidak ditemukan
        return TypeKind.INTEGER

    return TypeKind.NOTYPE

def visit_RecordFieldNode(self, node):
    """Kunjungi field record, mendukung nested record & array"""
    parent_type = self.visit(node.parent)  # tipe dari record parent
    if parent_type == TypeKind.NOTYPE:
        return TypeKind.NOTYPE

    if isinstance(parent_type, dict) and parent_type.get('kind') == TypeKind.RECORD:
        fields = parent_type.get('fields', {})
    else:
        self.report_error(SemanticError(
            f"Cannot access field '{node.name}' of non-record type"
        ))
        return TypeKind.NOTYPE

    # cek apakah field ada
    field_type = fields.get(node.name)
    if field_type is None:
        self.report_error(UndeclaredIdentifierError(node.name))
        return TypeKind.NOTYPE

    # dekorasi node
    node.attr['type'] = field_type
    node.attr['parent_type'] = parent_type
    return field_type


def visit_ArrayTypeNode(self, node):
    """Resolusi info tipe array untuk VarDecl/TypeDecl"""

    base_type_kind = self._get_type_kind(node.base_type)

    # Ambil batasan
    low = 1
    high = 10
    if hasattr(node, 'bounds') and node.bounds:
        bound_info = node.bounds[0]
        if isinstance(bound_info, tuple) and len(bound_info) == 2:
            lower_node, upper_node = bound_info
            low = lower_node.value if hasattr(lower_node, 'value') else 1
            high = upper_node.value if hasattr(upper_node, 'value') else 10

    return TypeKind.ARRAY, base_type_kind, low, high


def visit_NumNode(self, node):
    """Kunjungi literal angka"""
    # Periksa apakah integer atau real
    if isinstance(node.value, int) or '.' not in str(node.value):
        node.attr['type'] = TypeKind.INTEGER
        return TypeKind.INTEGER
    node.attr['type'] = TypeKind.REAL
    return TypeKind.REAL

def visit_StringNode(self, node):
    """Kunjungi literal string"""
    # Cek literal char dari lexer/AST builder
    if getattr(node, 'is_char_literal', False):
        node.attr['type'] = TypeKind.CHAR
        return TypeKind.CHAR

    literal_value = node.value if isinstance(node.value, str) else ''
    stripped_value = literal_value

    if len(literal_value) >= 2 and literal_value[0] == "'" and literal_value[-1] == "'":
        stripped_value = literal_value[1:-1]

    if len(stripped_value) == 1:
        node.attr['type'] = TypeKind.CHAR
        return TypeKind.CHAR

    node.attr['type'] = TypeKind.STRING
    return TypeKind.STRING

def visit_BooleanNode(self, node):
    """Kunjungi literal boolean"""
    return TypeKind.BOOLEAN
```

visit_BinOpNode(node): Validasi operasi biner (+, -, *, /, and, or, dll): kunjungi operands, cek tipe valid untuk operator, return tipe hasil

visit_UnaryOpNode(node): Validasi operasi unary (+, -, not): kunjungi operand, cek tipe valid, return tipe hasil

visit_VarNode(node): Validasi referensi variabel: cek identifier dideklarasikan, dekorasi node dengan info dari symbol table, return tipe

visit_ArrayAccessNode(node): Validasi akses array arr[i]: cek variabel adalah array, validasi index integer, ambil tipe elemen dari ATAB

visit_RecordFieldNode(node): Validasi akses field record rec.field: kunjungi parent, cek parent adalah record, validasi field ada, return tipe field

visit_ArrayTypeNode(node): Resolve info tipe array (tipe elemen, bounds) untuk deklarasi variabel/tipe

visit_NumNode(node): Return tipe INTEGER atau REAL berdasarkan apakah ada titik desimal

visit_StringNode(node): Return tipe CHAR jika 1 karakter (dengan strip quote), atau STRING jika > 1 karakter

visit_BooleanNode(node): Return tipe BOOLEAN untuk literal true/false

```python
def _get_type_kind(self, type_str):
    """Konversikan string tipe menjadi konstanta TypeKind"""
    type_map = {
        'integer': TypeKind.INTEGER,
        'real': TypeKind.REAL,
        'boolean': TypeKind.BOOLEAN,
        'char': TypeKind.CHAR,
        'string': TypeKind.STRING,
        'array': TypeKind.ARRAY,
        'larik': TypeKind.ARRAY,
        'record': TypeKind.RECORD,
        'rekaman': TypeKind.RECORD
    }

    if isinstance(type_str, str):
        #Tipe Primitif/Keyword
        mapped_type = type_map.get(type_str.lower())
        if mapped_type is not None:
            return mapped_type

        #Tipe Didefinisikan Pengguna (Cari di Symbol Table)
        symbol = self.symbol_table.lookup(type_str)
        if symbol and symbol['obj'] == ObjKind.TYPE:
            type_kind = symbol['type']
            # ambil fields jika record
            if type_kind == TypeKind.RECORD and 'fields' in symbol:
                return {'kind': TypeKind.RECORD, 'fields': symbol['fields']}
            return type_kind

        # Tipe Record/Array (fallback)
        return TypeKind.NOTYPE

    return TypeKind.NOTYPE

def _resolve_type_node(self, type_node):

    # RangeTypeNode -> integer subrange
    if isinstance(type_node, RangeTypeNode):
        low = type_node.lower.value
        high = type_node.upper.value
        type_node.attr['range'] = (low, high)
        return TypeKind.INTEGER   # Pascal subrange dianggap integer

    # ArrayTypeNode
    if isinstance(type_node, ArrayTypeNode):
        # simpan batasan dan tipe elemen
        base_type = self._get_type_kind(type_node.base_type)
        bound = type_node.bounds[0]
        low = bound[0].value
        high = bound[1].value
        type_node.attr['array_info'] = (base_type, low, high)
        return TypeKind.ARRAY

    # RecordTypeNode
    if isinstance(type_node, RecordTypeNode):
        fields = {}
        for f in type_node.fields:
            fields[f.name] = self._get_type_kind(f.type_)
        type_node.attr['fields'] = fields
        return TypeKind.RECORD

    # primitive / identifier type
    if isinstance(type_node, str):
        return self._get_type_kind(type_node)

    return TypeKind.NOTYPE

```

_get_type_kind(type_str): Konversi string nama tipe ke konstanta TypeKind, support tipe primitif & user-defined (lookup di symbol table)

_resolve_type_node(type_node): Resolve AST type node (RangeTypeNode, ArrayTypeNode, RecordTypeNode) ke TypeKind dengan info tambahan (bounds, fields)

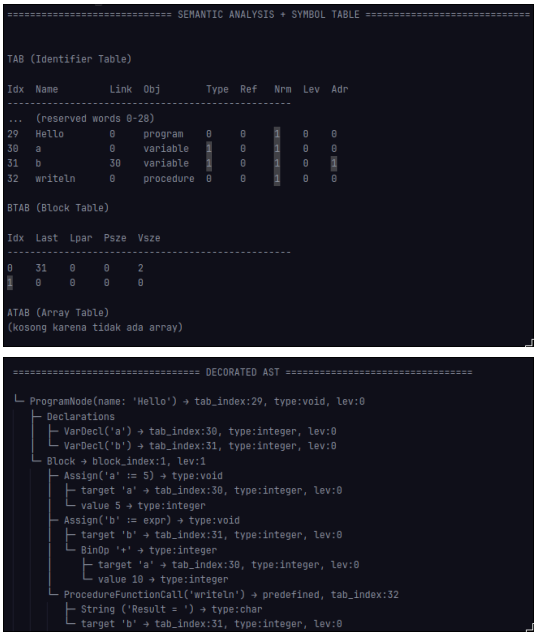
BAB V
PENGUJIAN TEST CASE

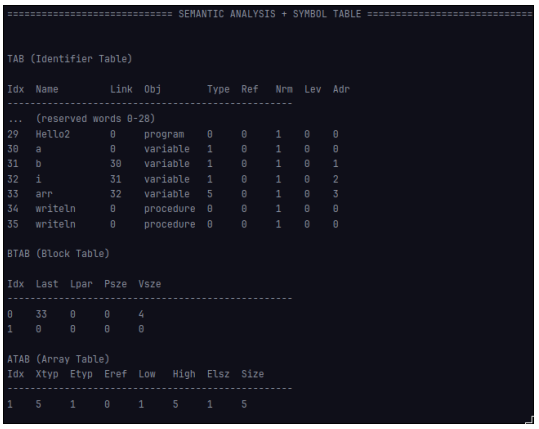
Untuk dokumentasi pengujian mandiri, dilakukan beberapa test case yang dapat dilihat pada tabel-tabel berikut:

(Dapat diakses pada repository, baik input maupun outputnya di “test/milestone-3/…”)

5.1 Test Case General

No.	Input	Output

1.	milestone-3/input/input-1.pas
	`program Hello;`
	`variabel`
	`  a, b: integer;`
	
	`mulai`
	`  a := 5;`
	`  b := a + 10;`
	`  writeln('Result = ', b);`
	`selesai.`
		

2.	milestone-3/input/input-2.pas
	`program Hello2;`
	`variabel`
	`  a, b, i: integer;`
	`  arr: larik[1 .. 5] dari integer;`
	
	`mulai`
	`  a := 5;`
	`  b := a + 10;`
	`  writeln('Result = ', b);`
	`  untuk i := 1 ke 5 lakukan`
	`  mulai`
	`  arr[i] := i + 1;`
	`  writeln(arr[i]);`
	`  selesai;`
	`selesai.`
		

<table>
<tr>
<td></td>
<td></td>
<td>

```
================================ DECORATED AST ================================
└─ ProgramNode(name: 'Hello2') → tab_index:29, type:void, lev:0
   ├─ Declarations
   │  ├─ VarDecl('a') → tab_index:30, type:integer, lev:0
   │  ├─ VarDecl('b') → tab_index:31, type:integer, lev:0
   │  ├─ VarDecl('i') → tab_index:32, type:integer, lev:0
   │  └─ VarDecl('arr') → tab_index:33, type:5, lev:0
   │     └─ ArrayType(base_type=integer, bounds=[(value 1, value 5)])
   └─ Block → block_index:1, lev:1
      ├─ Assign('a' := 5) → type:void
      │  ├─ target 'a' → tab_index:30, type:integer, lev:0
      │  └─ value 5 → type:integer
      ├─ Assign('b' := expr) → type:void
      │  ├─ target 'b' → tab_index:31, type:integer, lev:0
      │  └─ BinOp '+' → type:integer
      │     ├─ target 'a' → tab_index:30, type:integer, lev:0
      │     └─ value 10 → type:integer
      ├─ ProcedureFunctionCall('writeln') → predefined, tab_index:34
      │  ├─ String ('Result = ') → type:char
      │  └─ target 'b' → tab_index:31, type:integer, lev:0
      └─ For Loop
         ├─ target 'i'
         ├─ value 1 → type:integer
         ├─ value 5 → type:integer
         └─ Block
            ├─ Assign('target' := expr) → type:void
            │  ├─ Array Variable(array=arr, index=index)
            │  │  ├─ target 'arr'
            │  │  └─ target 'i' → tab_index:32, type:integer, lev:0
            │  └─ BinOp '+' → type:integer
            │     ├─ target 'i' → tab_index:32, type:integer, lev:0
            │     └─ value 1 → type:integer
            └─ ProcedureFunctionCall('writeln') → predefined, tab_index:35
               └─ Array Variable(array=arr, index=index)
                  ├─ target 'arr'
                  └─ target 'i' → tab_index:32, type:integer, lev:0
```

</td>
</tr>
<tr>
<td>3.</td>
<td>

milestone-3/input/input-3.pas

```
program RangesLoops;
variabel
  i: integer;
  a: larik[0 .. 3] dari integer;
mulai
  a[0] := 1;
  untuk i := 0 ke 3 lakukan
    a[i] := i;
  untuk i := 3 turun_ke 0 lakukan
    a[i] := a[i] + 1;
  jika a[1] < a[2] maka
    a[1] := a[2];
  selama i > 0 lakukan
    i := i - 1;
  writeln('ok', a[1]);
selesai.
```

</td>
<td>

```
========================== SEMANTIC ANALYSIS + SYMBOL TABLE ==========================


TAB (Identifier Table)

Idx  Name         Link  Obj        Type  Ref   Nrm  Lev  Adr
-----------------------------------------------------
...  (reserved words 0-28)
29   RangesLoops  0     program    0     0     1    0    0
30   i            0     variable   1     0     1    0    0
31   a            30    variable   5     0     1    0    1
32   writeln      0     procedure  0     0     1    0    0

BTAB (Block Table)

Idx  Last  Lpar  Psze  Vsze
-----------------------------------------------
0    31    0     0     2
1    0     0     0     0

ATAB (Array Table)
Idx  Xtyp  Etyp  Eref  Low   High  Elsz  Size
-----------------------------------------------
1    5     1     0     0     3     1     4
```

```
================================ DECORATED AST ================================
└─ ProgramNode(name: 'RangesLoops') → tab_index:29, type:void, lev:0
   ├─ Declarations
   │  ├─ VarDecl('i') → tab_index:30, type:integer, lev:0
   │  └─ VarDecl('a') → tab_index:31, type:5, lev:0
   │     └─ ArrayType(base_type=integer, bounds=[(value 0, value 3)])
   └─ Block → block_index:1, lev:1
      ├─ Assign('target' := 1) → type:void
      │  ├─ Array Variable(array=a, index=0)
      │  │  ├─ target 'a'
      │  │  └─ value 0 → type:integer
      │  └─ value 1 → type:integer
      ├─ For Loop
      │  ├─ target 'i'
      │  ├─ value 0 → type:integer
      │  ├─ value 3 → type:integer
      │  └─ Assign('target' := i) → type:void
      │     ├─ Array Variable(array=a, index=index)
      │     │  ├─ target 'a'
      │     │  └─ target 'i' → tab_index:30, type:integer, lev:0
      │     └─ target 'i' → tab_index:30, type:integer, lev:0
      ├─ For Loop
      │  ├─ target 'i'
      │  ├─ value 3 → type:integer
      │  ├─ value 0 → type:integer
      │  └─ Assign('target' := expr) → type:void
      │     ├─ Array Variable(array=a, index=index)
      │     │  ├─ target 'a'
      │     │  └─ target 'i' → tab_index:30, type:integer, lev:0
      │     └─ BinOp '+' → type:integer
      │        ├─ Array Variable(array=a, index=index)
      │        │  ├─ target 'a'
      │        │  └─ target 'i' → tab_index:30, type:integer, lev:0
      │        └─ value 1 → type:integer
```

</td>
</tr>
</table>

<table>
<tr>
<td></td>
<td></td>
<td>

```
    ├─ If Condition
    │  ├─ BinOp '<' → type:boolean
    │  │  ├─ Array Variable(array=a, index=1)
    │  │  │  ├─ target 'a'
    │  │  │  └─ value 1 → type:integer
    │  │  └─ Array Variable(array=a, index=2)
    │  │     ├─ target 'a'
    │  │     └─ value 2 → type:integer
    │  └─ Assign('target' := expr) → type:void
    │     ├─ Array Variable(array=a, index=1)
    │     │  ├─ target 'a'
    │     │  └─ value 1 → type:integer
    │     └─ Array Variable(array=a, index=2)
    │        ├─ target 'a'
    │        └─ value 2 → type:integer
    ├─ While Loop
    │  ├─ BinOp '>' → type:boolean
    │  │  ├─ target 'i' → tab_index:30, type:integer, lev:0
    │  │  └─ value 0 → type:integer
    │  └─ Assign('i' := expr) → type:void
    │     ├─ target 'i' → tab_index:30, type:integer, lev:0
    │     └─ BinOp '-' → type:integer
    │        ├─ target 'i' → tab_index:30, type:integer, lev:0
    │        └─ value 1 → type:integer
    └─ ProcedureFunctionCall('writeln') → predefined, tab_index:32
       ├─ String ('ok') → type:char
       └─ Array Variable(array=a, index=1)
          ├─ target 'a'
          └─ value 1 → type:integer
```

</td>
</tr>
<tr>
<td>4.</td>
<td>

milestone-3/input/input-4.pas

```
program Ops;
variabel
  x, y: integer;
  r: real;
  ch: char;
  ok: boolean;
mulai
  x := 10;
  y := 3;
  r := 12.5;
  ch := 'Z';
  ok := (x > y) dan tidak (x = y);
  x := x + y - 2 * (y bagi 2) / 1;
  x := x mod y;
  jika x <= y maka
    x := y
  selain_itu
    x := y + 1;
  writeln('done ', x);
selesai.
```

</td>
<td>

```
========================== SEMANTIC ANALYSIS + SYMBOL TABLE ==========================

TAB (Identifier Table)

Idx  Name        Link  Obj        Type  Ref  Nrm  Lev  Adr
-------------------------------------------------
...  (reserved words 0-28)
29   Ops         0     program    0     0    1    0    0
30   x           0     variable   1     0    1    0    0
31   y           30    variable   1     0    1    0    1
32   r           31    variable   2     0    1    0    2
33   ch          32    variable   4     0    1    0    3
34   ok          33    variable   3     0    1    0    4
35   writeln     0     procedure  0     0    1    0    0

BTAB (Block Table)

Idx  Last  Lpar  Psze  Vsze
-------------------------------------------------
0    34    0     0     5
1    0     0     0     0

ATAB (Array Table)
(kosong karena tidak ada array)
```

```
================================ DECORATED AST ================================

└─ ProgramNode(name: 'Ops') → tab_index:29, type:void, lev:0
   ├─ Declarations
   │  ├─ VarDecl('x') → tab_index:30, type:integer, lev:0
   │  ├─ VarDecl('y') → tab_index:31, type:integer, lev:0
   │  ├─ VarDecl('r') → tab_index:32, type:real, lev:0
   │  ├─ VarDecl('ch') → tab_index:33, type:char, lev:0
   │  └─ VarDecl('ok') → tab_index:34, type:boolean, lev:0
   └─ Block → block_index:1, lev:1
      ├─ Assign('x' := 10) → type:void
      │  ├─ target 'x' → tab_index:30, type:integer, lev:0
      │  └─ value 10 → type:integer
      ├─ Assign('y' := 3) → type:void
      │  ├─ target 'y' → tab_index:31, type:integer, lev:0
      │  └─ value 3 → type:integer
      ├─ Assign('r' := 12.5) → type:void
      │  ├─ target 'r' → tab_index:32, type:real, lev:0
      │  └─ value 12.5 → type:real
      ├─ Assign('ch' := 'Z') → type:void
      │  ├─ target 'ch' → tab_index:33, type:char, lev:0
      │  └─ String ('Z') → type:char
      ├─ Assign('ok' := expr) → type:void
      │  ├─ target 'ok' → tab_index:34, type:boolean, lev:0
      │  └─ BinOp '>' → type:boolean
      │     ├─ target 'x' → tab_index:30, type:integer, lev:0
      │     └─ target 'y' → tab_index:31, type:integer, lev:0
      ├─ Assign('x' := expr) → type:void
      │  ├─ target 'x' → tab_index:30, type:integer, lev:0
      │  └─ BinOp '-' → type:real
      │     ├─ BinOp '+' → type:integer
      │     │  ├─ target 'x' → tab_index:30, type:integer, lev:0
      │     │  └─ target 'y' → tab_index:31, type:integer, lev:0
      │     └─ BinOp '/' → type:real
      │        ├─ BinOp '*' → type:integer
      │        │  ├─ value 2 → type:integer
      │        │  └─ BinOp 'bagi' → type:integer
      │        │     ├─ target 'y' → tab_index:31, type:integer, lev:0
      │        │     └─ value 2 → type:integer
      │        └─ value 1 → type:integer
      ├─ Assign('x' := expr) → type:void
      │  ├─ target 'x' → tab_index:30, type:integer, lev:0
      │  └─ BinOp 'mod' → type:integer
      │     ├─ target 'x' → tab_index:30, type:integer, lev:0
      │     └─ target 'y' → tab_index:31, type:integer, lev:0
```

</td>
</tr>
</table>

<table>
<tr>
<td></td>
<td></td>
<td>

```
├─ If Condition
│  ├─ BinOp '<=' → type:boolean
│  │  ├─ target 'x' → tab_index:30, type:integer, lev:0
│  │  └─ target 'y' → tab_index:31, type:integer, lev:0
│  ├─ Assign('x' := y) → type:void
│  │  ├─ target 'x' → tab_index:30, type:integer, lev:0
│  │  └─ target 'y' → tab_index:31, type:integer, lev:0
│  └─ Assign('x' := expr) → type:void
│     ├─ target 'x' → tab_index:30, type:integer, lev:0
│     └─ BinOp '+' → type:integer
│        ├─ target 'y' → tab_index:31, type:integer, lev:0
│        └─ value 1 → type:integer
└─ ProcedureFunctionCall('writeln') → predefined, tab_index:35
   ├─ String ('done ') → type:char
   └─ target 'x' → tab_index:30, type:integer, lev:0
```

</td>
</tr>
<tr>
<td>5.</td>
<td>

milestone-3/input/input-5.pas

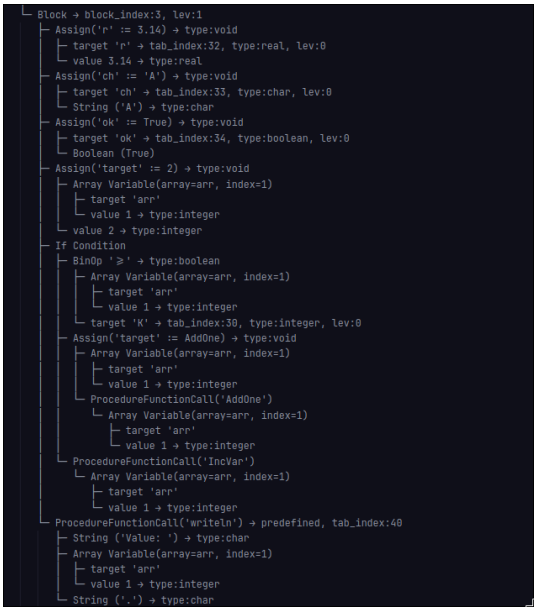
```
program DeclsProcFunc;
konstanta
  K = 10;
variabel
  i: integer;
  r: real;
  ch: char;
  ok: boolean;
  arr: larik[1 .. 10] dari integer;

prosedur IncVar(z: integer);
mulai
  z := z + 1;
selesai;

fungsi AddOne(t: integer): integer;
mulai
  AddOne := t + 1;
selesai;

mulai
  r := 3.14;
  ch := 'A';
  ok := true dan tidak false;
  arr[1] := 2;
  jika (arr[1] >= K) maka
    arr[1] := AddOne(arr[1])
  selain_itu
    IncVar(arr[1]);
  writeln('Value: ', arr[1], '.');
selesai.
```

</td>
<td>

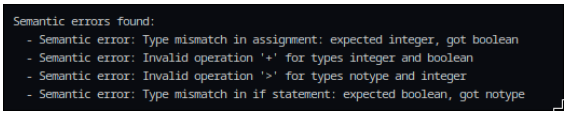
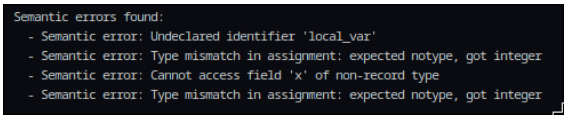
```
============================== SEMANTIC ANALYSIS + SYMBOL TABLE ==============================


TAB (Identifier Table)

Idx  Name         Link  Obj        Type  Ref   Nrm  Lev  Adr
-----------------------------------------------------------
...  (reserved words 0-28)
29   DeclsProcFunc 0    program    0     0     1    0    0
30   K            0     constant   1     0     1    0    None
31   i            30    variable   1     0     1    0    0
32   r            31    variable   2     0     1    0    1
33   ch           32    variable   4     0     1    0    2
34   ok           33    variable   3     0     1    0    3
35   arr          34    variable   5     0     1    0    4
36   IncVar       35    procedure  0     1     1    0    0
37   z            0     variable   1     0     1    1    0
38   AddOne       36    function   1     2     1    0    0
39   t            0     variable   1     0     1    1    0
40   writeln      0     procedure  0     0     1    0    0

BTAB (Block Table)

Idx  Last  Lpar  Psze  Vsze
-------------------------------------------------
0    38    0     0     5
1    37    37    1     0
2    39    39    1     0
3    0     0     0     0

ATAB (Array Table)
Idx  Xtyp  Etyp  Eref  Low   High  Elsz  Size
-------------------------------------------------
1    5     1     0     1     10    1     10
```

```
================================ DECORATED AST =================================

└─ ProgramNode(name: 'DeclsProcFunc') → tab_index:29, type:void, lev:0
   ├─ Declarations
   │  ├─ ConstDecl(name='K')
   │  ├─ VarDecl('i') → tab_index:31, type:integer, lev:0
   │  ├─ VarDecl('r') → tab_index:32, type:real, lev:0
   │  ├─ VarDecl('ch') → tab_index:33, type:char, lev:0
   │  ├─ VarDecl('ok') → tab_index:34, type:boolean, lev:0
   │  ├─ VarDecl('arr') → tab_index:35, type:5, lev:0
   │  │  └─ ArrayType(base_type=integer, bounds=[(value 1, value 10)])
   │  ├─ ProcedureDecl(name='IncVar')
   │  │  ├─ Param(names='z', type='integer', is_var=False)
   │  │  └─ Block
   │  │     ├─ Declarations
   │  │     └─ Block
   │  │        └─ Assign('z' := expr) → type:void
   │  │           ├─ target 'z' → tab_index:37, type:integer, lev:1
   │  │           └─ BinOp '+' → type:integer
   │  │              ├─ target 'z' → tab_index:37, type:integer, lev:1
   │  │              └─ value 1 → type:integer
   │  └─ FunctionDecl(name='AddOne')
   │     ├─ Param(names='t', type='integer', is_var=False)
   │     └─ Block
   │        ├─ Declarations
   │        └─ Block
   │           └─ Assign('AddOne' := expr) → type:void
   │              ├─ target 'AddOne' → tab_index:38, type:integer, lev:0
   │              └─ BinOp '+' → type:integer
   │                 ├─ target 't' → tab_index:39, type:integer, lev:1
   │                 └─ value 1 → type:integer
```

</td>
</tr>
</table>

5.2 Test Case untuk Milestone 3

No.	Input	Output
1.	milestone-3/input/input-fail-1.pas 	
2.	milestone-3/input/input-fail-2.pas	

```
program TestFailScope;

variabel
  global_var: integer;

prosedur localScope;
variabel
  local_var: integer;
mulai
  local_var := 10;
selesai;

mulai
  global_var := 20;

  local_var := 5;

  global_var.x := 10;
selesai.
```

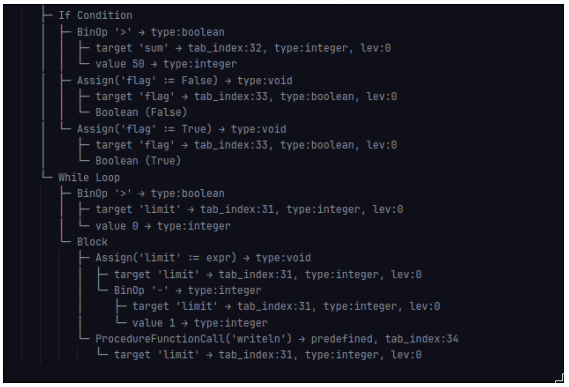
3.

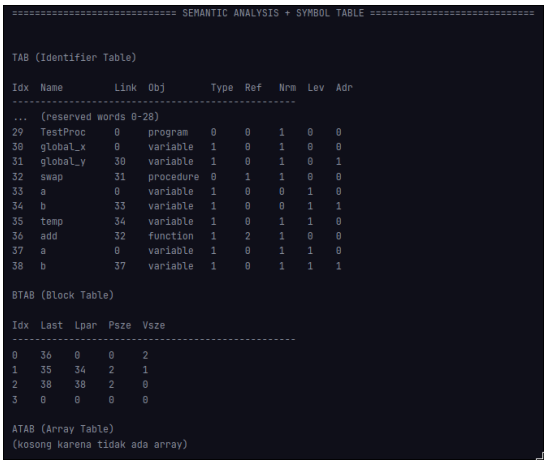
milestone-3/input/input-semantic-arrayrec.pas

```
program TestArrayRecord;

tipe
  Point = rekaman
    x, y: integer
  selesai;

variabel
  p: Point;
  matrix: larik[1 .. 5] dari integer;
  i, total: integer;

mulai
  p.x := 10;
  p.y := 20;
  total := p.x * p.y;

  untuk i := 1 ke 5 lakukan
    matrix[i] := i * 2;

  matrix[1] := matrix[1] + p.x;

  writeln(total);
selesai.
```

```
========================== SEMANTIC ANALYSIS + SYMBOL TABLE ==========================

TAB (Identifier Table)

Idx  Name         Link  Obj        Type  Ref   Nrm  Lev  Adr
-------------------------------------------------
...  (reserved words 0-28)
29   TestArrayRecord 0     program    0     0     1    0    0
30   Point        0     type       6     0     1    0    0
31   p            30    variable   6     0     1    0    0
32   matrix       31    variable   5     0     1    0    1
33   i            32    variable   1     0     1    0    2
34   total        33    variable   1     0     1    0    3
35   writeln      0     procedure  0     0     1    0    0

BTAB (Block Table)

Idx  Last  Lpar  Psze  Vsze
-------------------------------------------------
0    34    0     0     4
1    0     0     0     0

ATAB (Array Table)
Idx  Xtyp  Etyp  Eref  Low   High  Elsz  Size
-------------------------------------------------
1    5     1     0     1     5     1     5
```

```
============================== DECORATED AST ==============================

└─ ProgramNode(name: 'TestArrayRecord') → tab_index:29, type:void, lev:0
   ├─ Declarations
   │  ├─ TypeDecl(name='Point')
   │  │  └─ RecordType(fields=[RecordField('x'), RecordField('y')])
   │  │     ├─ RecordField('x')
   │  │     └─ RecordField('y')
   │  ├─ VarDecl('p') → tab_index:31, type:{'kind': 6, 'fields': {'x': 1, 'y': 1}}, lev:0
   │  ├─ VarDecl('matrix') → tab_index:32, type:5, lev:0
   │  │  └─ ArrayType(base_type=integer, bounds=[(value 1, value 5)])
   │  ├─ VarDecl('i') → tab_index:33, type:integer, lev:0
   │  └─ VarDecl('total') → tab_index:34, type:integer, lev:0
   └─ Block → block_index:1, lev:1
      ├─ Assign('x' := 10) → type:void
      │  ├─ RecordField('x') → type:integer
      │  │  └─ target 'p' → tab_index:31, type:{'kind': 6, 'fields': {'x': 1, 'y': 1}}, lev:0
      │  └─ value 10 → type:integer
      ├─ Assign('y' := 20) → type:void
      │  ├─ RecordField('y') → type:integer
      │  │  └─ target 'p' → tab_index:31, type:{'kind': 6, 'fields': {'x': 1, 'y': 1}}, lev:0
      │  └─ value 20 → type:integer
      ├─ Assign('total' := expr) → type:void
      │  ├─ target 'total' → tab_index:34, type:integer, lev:0
      │  └─ BinOp '*' → type:integer
      │     ├─ RecordField('x') → type:integer
      │     │  └─ target 'p' → tab_index:31, type:{'kind': 6, 'fields': {'x': 1, 'y': 1}}, lev:0
      │     └─ RecordField('y') → type:integer
      │        └─ target 'p' → tab_index:31, type:{'kind': 6, 'fields': {'x': 1, 'y': 1}}, lev:0
      ├─ For Loop
      │  ├─ target 'i'
      │  ├─ value 1 → type:integer
      │  ├─ value 5 → type:integer
      │  └─ Assign('target' := expr) → type:void
      │     ├─ Array Variable(array=matrix, index=index)
      │     │  ├─ target 'matrix'
      │     │  └─ target 'i' → tab_index:33, type:integer, lev:0
      │     └─ BinOp '*' → type:integer
      │        ├─ target 'i' → tab_index:33, type:integer, lev:0
      │        └─ value 2 → type:integer
      ├─ Assign('target' := expr) → type:void
      │  ├─ Array Variable(array=matrix, index=1)
      │  │  ├─ target 'matrix'
      │  │  └─ value 1 → type:integer
      │  └─ BinOp '+' → type:integer
      │     ├─ Array Variable(array=matrix, index=1)
      │     │  ├─ target 'matrix'
      │     │  └─ value 1 → type:integer
      │     └─ RecordField('x') → type:integer
      │        └─ target 'p' → tab_index:31, type:{'kind': 6, 'fields': {'x': 1, 'y': 1}}, lev:0
      └─ ProcedureFunctionCall('writeln') → predefined, tab_index:35
         └─ target 'total' → tab_index:34, type:integer, lev:0
```

4.	milestone-3/input/input-semantic-basic.pas
	
	``
	program TestBasic;
	
	variabel
	  i, limit, sum: integer;
	  flag: boolean;
	
	mulai
	  limit := 10;
	  sum := 0;
	  flag := true;
	
	  untuk i := 1 ke limit lakukan
	  sum := sum + i;
	
	  jika sum > 50 maka
	  flag := false
	  selain_itu
	  flag := true;
	
	  selama limit > 0 lakukan
	  mulai
	  limit := limit - 1;
	  writeln(limit);
	  selesai;
	selesai.
	``
	
	``
	============================ SEMANTIC ANALYSIS + SYMBOL TABLE ============================
	
	TAB (Identifier Table)
	
	Idx Name Link Obj Type Ref Nrm Lev Adr
	-- 	
	... (reserved words 0-28)
	29 TestBasic 0 program 0 0 1 0 0
	30 i 0 variable 1 0 1 0 0
	31 limit 30 variable 1 0 1 0 1
	32 sum 31 variable 1 0 1 0 2
	33 flag 32 variable 3 0 1 0 3
	34 writeln 0 procedure 0 0 1 0 0
	
	BTAB (Block Table)
	
	Idx Last Lpar Psze Vsze
	--- 	
	0 33 0 0 4
	1 0 0 0 0
	
	ATAB (Array Table)
	(kosong karena tidak ada array)
	``
	
	``
	================================ DECORATED AST ================================
	
	└─ ProgramNode(name: 'TestBasic') → tab_index:29, type:void, lev:0
	  ├─ Declarations
	  │ ├─ VarDecl('i') → tab_index:30, type:integer, lev:0
	  │ ├─ VarDecl('limit') → tab_index:31, type:integer, lev:0
	  │ ├─ VarDecl('sum') → tab_index:32, type:integer, lev:0
	  │ └─ VarDecl('flag') → tab_index:33, type:boolean, lev:0
	  └─ Block → block_index:1, lev:1
	  ├─ Assign('limit' := 10) → type:void
	  │ ├─ target 'limit' → tab_index:31, type:integer, lev:0
	  │ └─ value 10 → type:integer
	  ├─ Assign('sum' := 0) → type:void
	  │ ├─ target 'sum' → tab_index:32, type:integer, lev:0
	  │ └─ value 0 → type:integer
	  ├─ Assign('flag' := True) → type:void
	  │ ├─ target 'flag' → tab_index:33, type:boolean, lev:0
	  │ └─ Boolean (True)
	  ├─ For Loop
	  │ ├─ target 'i'
	  │ ├─ value 1 → type:integer
	  │ ├─ target 'limit' → tab_index:31, type:integer, lev:0
	  │ └─ Assign('sum' := expr) → type:void
	  │ ├─ target 'sum' → tab_index:32, type:integer, lev:0
	  │ └─ BinOp '+' → type:integer
	  │ ├─ target 'sum' → tab_index:32, type:integer, lev:0
	  │ └─ target 'i' → tab_index:30, type:integer, lev:0
	  ├─ If Condition
	  │ ├─ BinOp '>' → type:boolean
	  │ │ ├─ target 'sum' → tab_index:32, type:integer, lev:0
	  │ │ └─ value 50 → type:integer
	  │ ├─ Assign('flag' := False) → type:void
	  │ │ ├─ target 'flag' → tab_index:33, type:boolean, lev:0
	  │ │ └─ Boolean (False)
	  │ └─ Assign('flag' := True) → type:void
	  │ ├─ target 'flag' → tab_index:33, type:boolean, lev:0
	  │ └─ Boolean (True)
	  └─ While Loop
	  ├─ BinOp '>' → type:boolean
	  │ ├─ target 'limit' → tab_index:31, type:integer, lev:0
	  │ └─ value 0 → type:integer
	  └─ Block
	  ├─ Assign('limit' := expr) → type:void
	  │ ├─ target 'limit' → tab_index:31, type:integer, lev:0
	  │ └─ BinOp '-' → type:integer
	  │ ├─ target 'limit' → tab_index:31, type:integer, lev:0
	  │ └─ value 1 → type:integer
	  └─ ProcedureFunctionCall('writeln') → predefined, tab_index:34
	  └─ target 'limit' → tab_index:31, type:integer, lev:0
	``
	
	 

5.	milestone-3/input/input-semantic-proc.pas	

```
program TestProc;

variabel
  global_x, global_y: integer;

prosedur swap(variabel a: integer; variabel b: integer);
variabel temp: integer;
mulai
  temp := a;
  a := b;
  b := temp;
selesai;

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

mulai
  global_x := 100;
  global_y := 200;

  swap(global_x, global_y);

  global_x := add(global_x, global_y);
selesai.
```

```
================================ DECORATED AST ================================

└─ ProgramNode(name: 'TestProc') → tab_index:29, type:void, lev:0
   ├─ Declarations
   │  ├─ VarDecl('global_x') → tab_index:30, type:integer, lev:0
   │  ├─ VarDecl('global_y') → tab_index:31, type:integer, lev:0
   │  ├─ ProcedureDecl(name='swap')
   │  │  ├─ Param(names='a', type='integer', is_var=True)
   │  │  ├─ Param(names='b', type='integer', is_var=True)
   │  │  └─ Block
   │  │     ├─ Declarations
   │  │     │  └─ VarDecl('temp') → tab_index:35, type:integer, lev:1
   │  │     └─ Block
   │  │        ├─ Assign('temp' := a) → type:void
   │  │        │  ├─ target 'temp' → tab_index:35, type:integer, lev:1
   │  │        │  └─ target 'a' → tab_index:33, type:integer, lev:1
   │  │        ├─ Assign('a' := b) → type:void
   │  │        │  ├─ target 'a' → tab_index:33, type:integer, lev:1
   │  │        │  └─ target 'b' → tab_index:34, type:integer, lev:1
   │  │        └─ Assign('b' := temp) → type:void
   │  │           ├─ target 'b' → tab_index:34, type:integer, lev:1
   │  │           └─ target 'temp' → tab_index:35, type:integer, lev:1
   │  └─ FunctionDecl(name='add')
   │     ├─ Param(names='a', type='integer', is_var=False)
   │     ├─ Param(names='b', type='integer', is_var=False)
   │     └─ Block
   │        ├─ Declarations
   │        └─ Block
   │           └─ Assign('add' := expr) → type:void
   │              ├─ target 'add' → tab_index:36, type:integer, lev:0
   │              └─ BinOp '+' → type:integer
   │                 ├─ target 'a' → tab_index:33, type:integer, lev:1
   │                 └─ target 'b' → tab_index:34, type:integer, lev:1
```

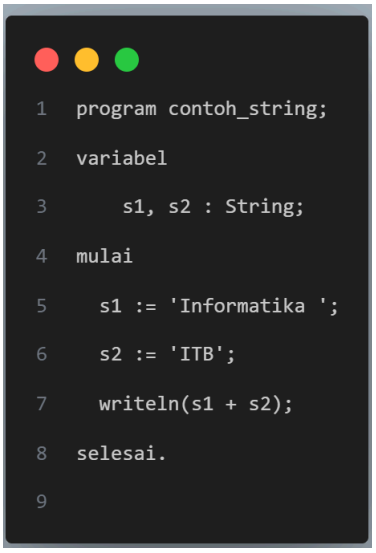
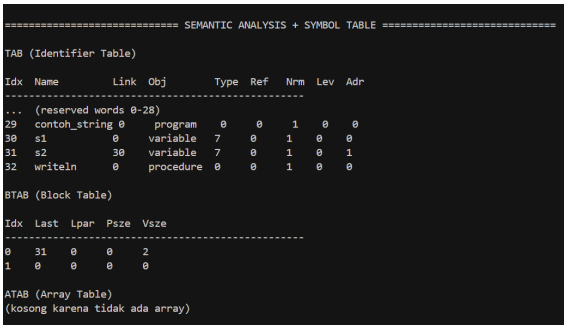
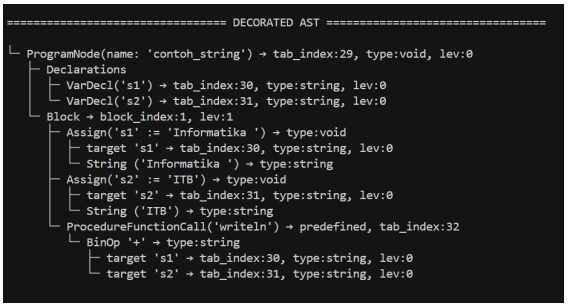
```
   └─ Block → block_index:3, lev:1
      ├─ Assign('global_x' := 100) → type:void
      │  ├─ target 'global_x' → tab_index:30, type:integer, lev:0
      │  └─ value 100 → type:integer
      ├─ Assign('global_y' := 200) → type:void
      │  ├─ target 'global_y' → tab_index:31, type:integer, lev:0
      │  └─ value 200 → type:integer
      ├─ ProcedureFunctionCall('swap')
      │  ├─ target 'global_x' → tab_index:30, type:integer, lev:0
      │  └─ target 'global_y' → tab_index:31, type:integer, lev:0
      └─ Assign('global_x' := add) → type:void
         ├─ target 'global_x' → tab_index:30, type:integer, lev:0
         └─ ProcedureFunctionCall('add')
            ├─ target 'global_x' → tab_index:30, type:integer, lev:0
            └─ target 'global_y' → tab_index:31, type:integer, lev:0
```

6. milestone-3/input/input-semantic-varreference.pas

```
program VarRef;
prosedur tukar(variabel x: integer; y: integer);
mulai
  x := y;
selesai;
mulai
  tukar(10, 20);
selesai.
```

```
============================ SEMANTIC ANALYSIS + SYMBOL TABLE ============================


TAB (Identifier Table)

Idx  Name         Link  Obj        Type  Ref   Nrm  Lev  Adr
------------------------------------------------------
...  (reserved words 0-28)
29   VarRef       0     program    0     0     1    0    0
30   tukar        0     procedure  0     1     1    0    0
31   x            0     variable   1     0     0    1    0
32   y            31    variable   1     0     1    1    1

BTAB (Block Table)

Idx  Last  Lpar  Psze  Vsze
--------------------------------------------------
0    30    0     0     0
1    32    32    2     0
2    0     0     0     0

ATAB (Array Table)
(kosong karena tidak ada array)
```

```
================================ DECORATED AST ================================

└─ ProgramNode(name: 'VarRef') → tab_index:29, type:void, lev:0
   ├─ Declarations
   │  └─ ProcedureDecl(name='tukar')
   │     ├─ Param(names='x', type='integer', is_var=True)
   │     ├─ Param(names='y', type='integer', is_var=False)
   │     └─ Block
   │        ├─ Declarations
   │        └─ Block
   │           └─ Assign('x' := y) → type:void
   │              ├─ target 'x' → tab_index:31, type:integer, lev:1
   │              └─ target 'y' → tab_index:32, type:integer, lev:1
   └─ Block → block_index:2, lev:1
      └─ ProcedureFunctionCall('tukar')
         ├─ value 10 → type:integer
         └─ value 20 → type:integer
```

7.	milestone-3/input/input-string.pas 	 

```
1  program contoh_string;
2  variabel
3      s1, s2 : String;
4  mulai
5    s1 := 'Informatika ';
6    s2 := 'ITB';
7    writeln(s1 + s2);
8  selesai.
9
```

```
========================== SEMANTIC ANALYSIS + SYMBOL TABLE ==========================

TAB (Identifier Table)

Idx  Name         Link  Obj        Type  Ref   Nrm  Lev  Adr
-------------------------------------------------------
...  (reserved words 0-28)
29   contoh_string 0     program    0     0     1    0    0
30   s1           0      variable   7     0     1    0    0
31   s2           30     variable   7     0     1    0    1
32   writeln      0      procedure  0     0     1    0    0

BTAB (Block Table)

Idx  Last  Lpar  Psze  Vsze
-------------------------------------------------------
0    31    0     0     2
1    0     0     0     0

ATAB (Array Table)
(kosong karena tidak ada array)
```

```
================================ DECORATED AST ================================

└─ ProgramNode(name: 'contoh_string') → tab_index:29, type:void, lev:0
   ├─ Declarations
   │  ├─ VarDecl('s1') → tab_index:30, type:string, lev:0
   │  └─ VarDecl('s2') → tab_index:31, type:string, lev:0
   └─ Block → block_index:1, lev:1
      ├─ Assign('s1' := 'Informatika ') → type:void
      │  ├─ target 's1' → tab_index:30, type:string, lev:0
      │  └─ String ('Informatika ') → type:string
      ├─ Assign('s2' := 'ITB') → type:void
      │  ├─ target 's2' → tab_index:31, type:string, lev:0
      │  └─ String ('ITB') → type:string
      └─ ProcedureFunctionCall('writeln') → predefined, tab_index:32
         └─ BinOp '+' → type:string
            ├─ target 's1' → tab_index:30, type:string, lev:0
            └─ target 's2' → tab_index:31, type:string, lev:0
```

BAB VI
KESIMPULAN & SARAN

6.1 Kesimpulan

Implementasi Semantic Analysis untuk compiler Pascal-S pada Milestone 3 telah berhasil diselesaikan. Pada tahap ini, struktur program yang sebelumnya direpresentasikan sebagai parse tree dan AST telah dianalisis lebih lanjut untuk memastikan konsistensi makna dan kepatuhan terhadap aturan bahasa Pascal-S yang bersifat *strong static typing*. Proses analisis dilakukan dengan menggabungkan mekanisme Attributed Grammar, penyusunan dan manajemen Symbol Table yang mencakup tab, btab, dan atab, serta traversal AST secara top-down menggunakan semantic visitor. Pendekatan ini memungkinkan pemeriksaan menyeluruh terhadap deklarasi identifier, tipe data, scope blok bersarang, operasi aritmatika maupun logika, pemanggilan prosedur/fungsi, dan akses struktur komposit (array serta record).

Selain melakukan validasi, semantic analyzer juga menghasilkan *Decorated AST*, yaitu AST yang telah dianotasi dengan atribut semantik seperti tipe hasil ekspresi, indeks tabel simbol, dan informasi blok. Dengan demikian, milestone ini berhasil menyediakan pondasi semantik yang solid dan konsisten untuk tahapan berikutnya, yaitu Intermediate Code Generation.

6.2 Saran

Meskipun semantic analyzer telah berfungsi dengan baik untuk menangani aturan semantik Pascal-S, masih terdapat beberapa aspek yang dapat ditingkatkan. Pertama, mekanisme error reporting dapat diperluas untuk menampilkan pesan kesalahan yang lebih kontekstual dan rapih, termasuk pelacakan lokasi kesalahan yang lebih presisi dengan informasi line yang menyebabkan error. Selain itu, optimisasi traversal AST dan pengelolaan symbol table memungkinkan peningkatan efisiensi, terutama terhadap *edge cases* seperti tipe rekursif, pengecekan kompatibilitas operator yang lebih lengkap dan akurat, serta verifikasi parameter prosedur/fungsi yang lebih ketat juga.

REFERENSI

[1] Edunex ITB. (n.d.). *Compiler Parsing dan Analisis Semantik.* ITB Lecture Module.

Accessed: November 24, 2025. [Online]. Available: http://edunex.itb.ac.id

[2] N. Wirth, PASCAL-S: A Subset and its implementation. Accessed: November 12, 2025. [Online]. Available: http://pascal.hansotten.com/uploads/pascals/PASCAL-S%20A%20subset%20and%20its%20Implementation%20012.pdf

[3] J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd ed. Boston, MA: Addison-Wesley, 2006.

[4] ULiège, *Semantic Analysis* Available: https://people.montefiore.uliege.be/geurts/Cours/compil/2015/04-semantic-2015-2016.pdf

[5]“Pascal for small machines” Available: http://pascal.hansotten.com/niklaus-wirth/pascal-s/

LAMPIRAN

Link Repository : Link Repository [GitHub]

Link Diagram DFA (Lexer) : Link Diagram [Draw.io]

Pembagian Tugas:

Nama	NIM	Tugas	Kontribusi
Brian Ricardo Tamin	13523126	Implementasi Symbol Table, Laporan	25%
Jovandra Otniel P. S.	13523141	Implementasi Type Checker, Laporan	25%
Andrew Tedjapratama	13523148	Implementasi Decorated Abstract Syntax Tree, Laporan	25%
Theo Kurniady	13523154	Implementasi AST Builder dan Struktur Data AST Node	25%