

# **Milestone 3**

## **IF2224 Teori Bahasa Formal dan Otomata**

### **Semantic Analysis**



### **Kelompok MOE**

#### **Disusun oleh:**

Ivant Samuel Silaban 13523129  
Rafa Abdussalam Danadyaksa 13523133  
Muhamad Nazih Najmudin 13523144  
Anas Ghazi Al Gifari 13523159  
Muhammad Rizain Firdaus 13523164

**PROGRAM STUDI TEKNIK INFORMATIKA**  
**SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA**  
**INSTITUT TEKNOLOGI BANDUNG**  
**2025**

## DAFTAR ISI

<b>DAFTAR ISI.....</b>	<b>1</b>
<b>DAFTAR GAMBAR.....</b>	<b>3</b>
<b>DAFTAR TABEL.....</b>	<b>4</b>
<b>BAB I</b>	
<b>LANDASAN TEORI.....</b>	<b>5</b>
1.1 Semantic Analysis.....	5
1.2 Attributed Grammar.....	6
1.2.1 Jenis-Jenis Atribut.....	6
1.2.2 Aturan Semantik pada Attributed Grammar.....	8
1.2.3 L-Attributed Grammar.....	9
1.3 Symbol Table.....	9
1.3.1 Struktur Symbol Table.....	10
1.3.2 Contoh Proses Lookup dan Insert.....	13
1.4 Semantic Checking.....	14
1.4.1 Duplication Checking.....	14
1.4.2 Defined Checking.....	14
1.4.3 Type Checking.....	15
1.5 Syntax-Directed Translation.....	15
1.6 Abstract Syntax Tree (AST).....	16
1.6.1 Struktur dan Karakteristik AST.....	16
1.6.2 Perbedaan Parse Tree dan AST.....	16
1.6.3 Decorated AST.....	17
1.7 Error dalam Semantic Analysis.....	17
1.8 Visit Functions.....	18
<b>BAB II</b>	
<b>PERANCANGAN DAN IMPLEMENTASI.....</b>	<b>20</b>
2.1 Arsitektur Sistem.....	20
2.2 Desain Struktur Data.....	21
2.2.1 Abstract Syntax Tree Nodes.....	21
2.2.2 Symbol Table Structure.....	23
2.2.3 Error Representation.....	26
2.3 Algoritma dan Mekanisme Implementasi.....	27
2.3.1 Algoritma Pembangunan AST.....	27
2.3.2 Algoritma Type Checking.....	27
2.3.3 Algoritma Scope Resolution.....	27
2.3.4 Algoritma Symbol Table Insertion.....	27
2.3.5 Algoritma AST Decoration.....	28

2.4 Mekanisme Error Handling.....	28
2.5 Implementasi Algoritma.....	29
2.5.1 Struktur File.....	29
2.5.2 Kelas dan Fungsi.....	31
<b>BAB III</b>	
<b>PENGUJIAN.....</b>	<b>63</b>
3.1 Rencana Pengujian.....	63
3.2 Test Case.....	63
3.2.1 Test Case 1: Program.....	63
3.2.2 Test Case 2: Constant Modification.....	65
3.2.3 Test Case 3: Redeclaration.....	66
3.2.4 Test Case 4: Type Mismatch.....	66
3.2.5 Test Case 5: Undeclared Variable.....	67
3.3 Ringkasan Hasil Pengujian.....	67
<b>BAB IV</b>	
<b>KESIMPULAN DAN SARAN.....</b>	<b>69</b>
4.1 Kesimpulan.....	69
4.2 Saran.....	70
<b>LAMPIRAN.....</b>	<b>71</b>
<b>REFERENSI.....</b>	<b>73</b>

## DAFTAR GAMBAR

Gambar 1.1. Penggunaan Synthesized Attributes untuk mengevaluasi ekspresi.....	7
Gambar 1.2. Penggunaan Inherited Attributes untuk mengevaluasi ekspresi.....	8
Gambar 1.3. AST untuk ekspresi $2 * 7 + 3$ dengan penjelasannya.....	16
Gambar 1.4. Decorated AST untuk pernyataan $b := a + 10$ .....	17
Gambar 2.1. Arsitektur Sistem.....	20

**DAFTAR TABEL**

Tabel 1.1. Contoh tabel identifier.....	11
Tabel 1.2. Contoh tabel blok.....	12
Tabel 1.3. Contoh tabel array.....	12
Tabel 2.1. Hierarki kelas AST Node.....	23
Tabel 2.2. Struktur entri tab.....	24
Tabel 2.3. Struktur entri btab.....	25
Tabel 2.4. Struktur entri atab.....	26
Tabel 2.5. Jenis-jenis Semantic Error.....	27
Tabel 2.6. Kategori Semantic Error.....	29
Tabel 3.1. Hasil Pengujian.....	68

## BAB I

### LANDASAN TEORI

#### 1.1 Semantic Analysis

Sebelumnya, *parser* telah memastikan bahwa kode Pascal-S sudah benar secara sintaks berdasarkan aturan produksi grammar. Namun, perlu disadari bahwa sebuah *grammar* dapat menerima suatu program yang tidak masuk akal walaupun sintaksnya sudah benar. Akibatnya, perlu dilakukan pemeriksaan lebih lanjut apakah struktur program tersebut juga benar secara makna (*semantic*).

Maka dari itu, pada tahap berikutnya akan dilakukan analisis semantik untuk memeriksa konsistensi tipe data, deklarasi variabel dan fungsi, lingkup (*scope*) program, serta control flow yang sesuai dengan aturan bahasa.

Analisis semantik (Semantic Analysis) merupakan tahap ketiga dalam proses kompilasi yang bertujuan untuk memvalidasi makna (*semantics*) dari suatu program setelah konstruksi sintaksnya dinyatakan valid oleh *parser*. Meskipun pemeriksaan sintaksis memastikan bahwa suatu program telah mematuhi aturan produksi dalam Context-Free Grammar (CFG), kepatuhan tersebut tidak menjamin bahwa program memiliki makna yang dapat dieksekusi dengan benar.

Berdasarkan teori kompilasi, terdapat perbedaan mendasar antara sintaks dan semantik. Sintaks mengacu pada aturan-aturan yang menyatakan apakah suatu rangkaian simbol valid secara struktur tata bahasa atau tidak. Sebaliknya, semantik mengacu pada aturan-aturan yang menyatakan makna dari program tersebut. Dengan kata lain, analisis sintaksis memastikan bahwa struktur program sesuai dengan aturan tata bahasa (*grammar*), sementara analisis semantik memastikan bahwa setiap konstruksi dalam program memiliki makna yang benar dan konsisten berdasarkan aturan semantik bahasa pemrograman.

Dalam penerapannya, dilakukan anotasi terhadap parse tree yang telah dihasilkan pada tahap sebelumnya dengan menggunakan pendekatan *top-down traversal*. Anotasi berupa (dan tak terbatas kepada) penambahan informasi seperti tipe data dan referensi simbol (*identifier*) ke Symbol Table.

Sebagai contoh, perhatikan ekspresi berikut.

```
A := (A + B) * (C + D)
```

Secara sintaksis, ekspresi tersebut valid dan *parser* dapat mengenali token-token seperti *identifier A*, operator `:=`, `+`, dan `*`. Namun, *parser* tidak mengetahui apakah simbol-simbol tersebut valid secara semantik. Kompilator harus memanggil *semantic routines* untuk mengenali maknanya, misalnya untuk memastikan bahwa variabel *A* dan *B* benar-benar ada dan tipenya dapat dijumlahkan.

Tujuan utama dari fase ini adalah memanfaatkan Syntax Tree yang dihasilkan parser untuk memeriksa konsistensi program, meliputi:

1. Pemeriksaan Variabel: Apakah variabel telah dideklarasikan sebelum digunakan.
2. Pemeriksaan Tipe: Apakah tipe data variabel konsisten (sama atau kompatibel)
3. Pemeriksaan Operand: Apakah operand yang dioperasikan memiliki nilai yang valid.

## 1.2 Attributed Grammar

Attributed Grammar merupakan perluasan dari Context-Free Grammar (CFG) dengan menambahkan seperangkat atribut dan aturan semantik yang membawa dan memproses informasi tambahan selama analisis semantik. Dengan adanya atribut, setiap simbol dalam *grammar*, baik simbol terminal maupun nonterminal, dapat memuat informasi yang merepresentasikan berbagai aspek semantik.

### 1.2.1 Jenis-Jenis Atribut

Dalam Attributed Grammar, atribut dikategorikan menjadi dua jenis utama berdasarkan cara perolehan dan aliran informasinya pada Parse Tree.

#### 1. Synthesized Attributes

Atribut tersintesis (Synthesized Attributes) adalah nilai atribut yang dimiliki oleh suatu nonterminal pada sisi kiri (LHS) aturan produksi, yang diturunkan dari nilai atribut simbol-simbol pada sisi kanan (RHS) aturan tersebut.

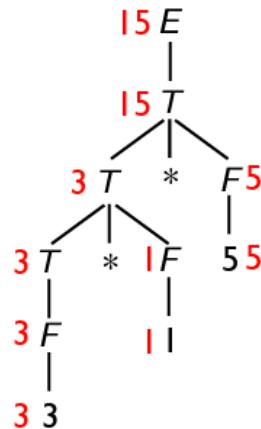
#### 2. Inherited Attributes

Atribut terwariskan (Inherited Attributes) adalah nilai atribut milik suatu nonterminal pada sisi kanan aturan produksi yang dihitung berdasarkan atribut nonterminal pada sisi kiri aturan. Dalam beberapa kasus, atribut ini juga dapat dihitung berdasarkan atribut dari nonterminal lain yang juga berada pada sisi kanan.

Terminal hanya dapat memiliki Synthesized Attributes yang diberikan oleh *lexer*, misalnya id.lexeme yang merepresentasikan nilai leksikal suatu *identifier*. Terminal tidak dapat mempunyai Inherited Attributes karena tidak berperan dalam membawa informasi dari struktur sintaksis yang lebih tinggi.

Berikut ini adalah contoh penggunaan Synthesized Attributes dalam proses evaluasi ekspresi aritmatika berdasarkan suatu *grammar* yang bersifat rekursif kiri. *Grammar* tersebut terdiri atas sejumlah produksi yang masing-masing dilengkapi dengan aturan semantik untuk menghitung nilai ekspresi.

Production	Semantic rules
$L \rightarrow E$	$L.val = E.val$
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow \text{num}$	$F.val = \text{num}.lexval$



Gambar 1.1. Penggunaan Synthesized Attributes untuk mengevaluasi ekspresi  
(Sumber: P. Geurts, Semantic Analysis)

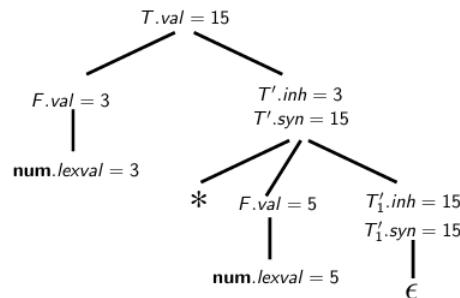
Pada produksi pertama, nonterminal  $L$  memperoleh nilai atribut secara langsung dari nonterminal  $E$ , sehingga  $L.val = E.val$ . Selanjutnya, produksi  $E \rightarrow E_1 + T$  mendefinisikan bahwa nilai ekspresi merupakan hasil penjumlahan antara nilai  $E_1$  dan nilai  $T$ , yakni  $E.val = E_1.val + T.val$ . Untuk produksi  $E \rightarrow T$ , nilai  $E$  sama dengan nilai  $T$ , dan aturan yang serupa berlaku untuk produksi  $T \rightarrow F$ , di mana  $T.val = F.val$ .

Produksi  $T \rightarrow T_1 * F$  menggambarkan operasi perkalian dalam ekspresi, sehingga nilai atribut  $T$  dihitung sebagai hasil perkalian nilai dari  $T_1$  dan nilai dari  $F$ , yakni  $T.val = T_1.val \times F.val$ . Pada tingkat faktor, produksi  $F \rightarrow (E)$  menyatakan bahwa nilai faktor diperoleh dari nilai ekspresi yang berada di dalam tanda kurung, yaitu  $F.val = E.val$ . Sementara itu, produksi  $F \rightarrow \text{num}$  menyatakan bahwa nilai faktor berasal dari nilai leksikal token num, dengan aturan semantik  $F.val = \text{num}.lexval$ . Dengan demikian, keseluruhan grammar memperlihatkan aliran informasi dari *leaf* menuju *root* dalam Syntax Tree melalui Synthesized Attributes.

Selanjutnya adalah contoh penggunaan Inherited Attributes untuk mengevaluasi ekspresi aritmatika dalam suatu grammar LL. Tidak seperti Synthesized Attributes yang mengalir dari *child* ke *parent*, Inherited Attributes memungkinkan informasi mengalir dari *parent* ke *child*, sehingga sangat cocok untuk grammar LL yang diproses secara *top-down*. Grammar yang ditampilkan terdiri atas produksi-produksi yang memecah ekspresi berdasarkan struktur faktor dan rangkaian operasi perkalian.

## LL expression grammar

Production	Semantic rules
$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow *FT'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow \text{num}$	$F.val = \text{num.lexval}$



Gambar 1.2. Penggunaan Inherited Attributes untuk mengevaluasi ekspresi  
(Sumber: P. Geurts, Semantic Analysis)

Produksi pertama,  $T \rightarrow FT'$ , menginisialisasi atribut terwarisi milik nonterminal  $T'$  dengan nilai atribut tersintesis dari  $F$ , yakni  $T'.inh = F.val$ . Setelah itu, nilai akhir dari  $T$  ditentukan oleh atribut tersintesis milik  $T'$ , sehingga  $T.val = T'.syn$ . Produksi kedua,  $T' \rightarrow *FT'_1$ , menangani operasi perkalian. Pada tahap ini, nilai atribut terwarisi untuk  $T'_1$  dihitung dengan mengalikan nilai terwarisi pada  $T'$  dengan nilai faktor  $F$ , yaitu  $T'_1.inh = T'.inh \times F.val$ . Nilai tersintesis  $T'$  kemudian diteruskan dari  $T'_1$ , sehingga  $T'.syn = T'_1.syn$ .

Produksi  $T' \rightarrow \epsilon$  berfungsi sebagai basis rekursi. Ketika tidak ada lagi operator perkalian yang harus diproses, atribut tersintesis dari  $T'$  ditetapkan sama dengan nilai atribut terwarisi yang dimilikinya, yakni  $T'.syn = T'.inh$ . Dengan demikian, nilai parsial yang telah dihitung melalui propagasi atribut terwarisi dapat dikembalikan sebagai hasil akumulasi. Pada tingkat faktor, produksi  $F \rightarrow \text{num}$  mendefinisikan bahwa nilai faktor berasal dari nilai leksikal token numerik, dituliskan sebagai  $F.val = \text{num.lexval}$ .

### 1.2.2 Aturan Semantik pada Attributed Grammar

Setiap aturan produksi dalam Attributed Grammar dilengkapi dengan aturan semantik yang mendefinisikan bagaimana atribut dihitung. Aturan semantik dapat berupa ekspresi atau prosedur kecil yang menentukan nilai atribut berdasarkan atribut lain yang terkait. Sebagai contoh, aturan produksi

`<AssignmentStatement> → IDENTIFIER := <Expression>`

mendeskripsikan bentuk pernyataan penugasan. IDENTIFIER memberikan nama variabel yang menjadi target dan operator `:=` menghubungkannya dengan nilai hasil evaluasi `<Expression>`. Aturan produksi ini hanya menjelaskan struktur sintaksis atau pola pembentukan kalimat dalam bahasa. Namun, *grammar* saja tidak cukup untuk memenuhi makna dari pernyataan tersebut. Untuk itu, diperlukan aturan semantik yang pada kasus ini dapat didefinisikan sebagai

```
AssignmentStatement.node = new AssignNode(
    new VarNode(IDENTIFIER.lexeme),
    Expression.node
)
```

Secara formal dalam Attributed Grammar, setiap simbol dalam *grammar* dapat memiliki satu atau lebih atribut. Pada contoh ini, atribut yang digunakan adalah atribut sintetis, yaitu atribut *.node* yang menyimpan AST node yang dihasilkan.

Aturan semantik ini memastikan bahwa analisis semantik dapat melakukan pelacakan yang konsisten terhadap variabel yang terlibat dalam operasi penugasan, sekaligus menyediakan representasi terstruktur yang memungkinkan pemeriksaan tipe, pemeriksaan keberadaan deklarasi variabel dalam tabel simbol, dan validasi aturan-aturan semantik lainnya. Dengan membangun node AST yang tepat pada tahap ini, kompilator dapat menjamin bahwa setiap pernyataan penugasan tidak hanya benar secara sintaktis, tetapi juga bermakna secara semantik sesuai dengan spesifikasi bahasa.

### 1.2.3 L-Attributed Grammar

L-Attributed Grammar merupakan subset dari Attributed Grammar yang memiliki sifat memungkinkan evaluasi atribut dilakukan dalam satu kali traversal *left-to-right, depth-first* terhadap Parse Tree. Dalam *grammar* jenis ini, setiap *inherited attribute* hanya boleh bergantung pada atribut dari simbol *parent*, atribut dari simbol *left siblings*, atau *synthesized attribute* dari simbol itu sendiri atau *children*-nya.

Sifat ini menjadikan L-Attributed Grammar cocok untuk diimplementasikan dalam kompilator karena mendukung *one-pass attribute evaluation* dan dapat memanfaatkan Syntax-Directed Translation Scheme.

## 1.3 Symbol Table

Symbol Table adalah struktur data yang digunakan *semantik analyzer* untuk sebagai pusat penyimpanan informasi mengenai seluruh *identifier* yang terdapat dalam suatu program. *Identifier* tersebut dapat berupa variabel, konstanta, prosedur, fungsi, parameter, tipe data buatan, *record field*, hingga informasi struktural lain yang diperlukan selama proses analisis semantik. Symbol table umumnya menyediakan *interface* untuk tiga operasi utama.

### 1. Insert

Operasi ini dilakukan ketika kompilator menemukan sebuah deklarasi baru. Informasi mengenai *identifier*, seperti nama, tipe data, kategori (variabel/konstanta/prosedur), dan atribut tambahan, dimasukkan ke dalam tabel.

### 2. Lookup

Operasi pencarian dilakukan ketika suatu *identifier* digunakan. Kompilator harus mengetahui apakah *identifier* tersebut telah dideklarasikan, apa tipenya, di *scope* mana ia berada, dan apakah penggunaannya konsisten dengan deklarasi sebelumnya.

### 3. Scope Management

Kompilator mengelola *visibility identifier* berdasarkan *lexical scope* untuk menentukan area keberlakuan suatu *identifier*.

Pengelolaan informasi yang efisien melalui Symbol Table merupakan inti dalam analisis semantik, terutama untuk pemeriksaan tipe, pemeriksaan deklarasi ganda, validasi parameter fungsi, serta pelacakan *offset address*.

#### 1.3.1 Struktur Symbol Table

Dalam implementasi Pascal-S, Symbol Table disusun ke dalam tiga tabel terpisah, yaitu tab, btab, dan atab. Masing-masing dari ketiga tabel ini dirancang untuk menangani jenis informasi yang berbeda dengan struktur yang lebih terfokus.

##### 1. tab (tabel *identifier*)

tab menyimpan atribut dari setiap *identifier*. Setiap entri pada tabel ini berisi

- identifiers: Nama *identifier* (misalnya nama variabel, konstanta, tipe, prosedur, fungsi). Indeks dimulai dari 29 karena ada *reserved words*.
- link: Pointer/index ke *identifier* sebelumnya dalam *scope* yang sama. Digunakan untuk *manajemen scope* (*linked list* per blok).
- obj: Kelas objek yang dienumerasi (0 = konstanta, 1 = variabel, 2 = type, 3 = prosedur, 4 = fungsi)
- type: Tipe data dari *identifier* (1 = integer, 2 = real, 3 = boolean, 4 = char, 5 = array, 6 = record)
- ref: Pointer/index 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, dan seterusnya).
- adr: Makna tergantung jenis objek, dapat berupa offset variabel di stack frame, nilai konstanta, offset field record, alamat prosedur, atau ukuran/penanda lain.

tab digunakan selama analisis semantik untuk memastikan apakah suatu variabel telah dideklarasikan sebelum digunakan, apakah pemanggilan prosedur/fungsi konsisten dengan deklarasinya, atau apakah tipe ekspresi sesuai dengan tipe variabel tujuan. Sebagai contoh, deklarasi berikut dalam Pascal-S

```
program TestIdentifierTable;
variabel x: integer;
konstanta N = 10;
mulai
```

```
// ...
selesai.
```

akan menghasilkan tab seperti berikut

identifiers	link	obj	typ	ref	nrm	lev	adr
29 x	0	1	1	0	1	0	1
30 N	29	0	1	0	1	0	10

Tabel 1.1. Contoh tabel identifier

tab menjadi pusat pencarian ketika kompilator membaca ekspresi seperti  $x + N$ .

## 2. btab (tabel blok)

btab menyimpan atribut dari blok program seperti prosedur, fungsi, atau *record type*. Setiap entri pada tabel ini berisi

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

Ketika kompilator masuk ke blok baru, sebuah entri btab dibuat. Sebagai contoh, deklarasi berikut dalam Pascal-S

```
program TestBlockTable;

prosedur compute(a, b: integer);
variabel temp: integer;
mulai
    temp := a + b;
selesai;

mulai
    // ...
selesai.
```

akan menghasilkan btab seperti berikut.

blocks	last	lpar	psze	vsze
1	28	1	0	0
2	29	31	3	1

Tabel 1.2. Contoh tabel blok

Informasi ini digunakan untuk mengatur penempatan parameter dan variabel lokal di memori.

### 3. atab (tabel array)

atab menyimpan atribut dari tipe data array. Setiap entri pada tabel ini berisi

- 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

Struktur ini diperlukan agar kompilator dapat melakukan pemeriksaan indeks, menghitung *offset* elemen array secara benar, dan mengonfirmasi konsistensi tipe pada ekspresi array. Sebagai contoh, deklarasi berikut dalam Pascal-S

```
program TestArrayTable;
variabel A: larik[1..10] dari integer;
mulai
  // ...
selesai.
```

akan menghasilkan atab seperti berikut

arrays	xtyp	etyp	eref	low	high	elsz	size
1	1	1	0	1	10	1	10

Tabel 1.3. Contoh tabel array

Ketika kompilator menemukan ekspresi  $A[i]$ , ia akan menggunakan atab untuk mengetahui informasi detail tentang array A agar dapat melakukan *type checking*, *range checking*, dan perhitungan alamat memori secara akurat untuk ekspresi array.

### 1.3.2 Contoh Proses Lookup dan Insert

Untuk memahami bagaimana proses lookup dan insert bekerja, perhatikan potongan kode Pascal-S berikut.

```
variabel x: integer;

prosedur test();
variabel y: integer;
mulai
    x := y + 1;
selesai;
```

Ketika kompilator mulai membaca kode ini, langkah pertama adalah memproses deklarasi variabel global x. Pada tahap ini, kompilator melakukan Insert ke dalam tab dengan parameter yang sesuai: nama *identifier* adalah x, objek (obj) ditetapkan sebagai variabel, tipe (typ) ditetapkan sebagai integer, referensi (ref) bernilai 0 karena ini tipe primitif, nrm diatur 1 karena ini variabel normal, level *lexical* (lev) adalah 0 karena berada di level global, dan *offset* (adr) diatur sesuai posisi variabel di memori global. Pada titik ini, entri untuk x dimasukkan ke dalam tabel sehingga semua informasi penting tentang variabel ini tersedia untuk lookup di tahap berikutnya.

Selanjutnya, kompilator menemukan deklarasi prosedur test(). Di sini, kompilator kembali melakukan Insert ke dalam tab untuk entri prosedur. Nama *identifier* adalah test, objek (obj) diatur sebagai prosedur, tipe (typ) biasanya 0 untuk prosedur, ref menunjuk ke btab yang akan dibuat untuk menyimpan informasi variabel lokal dan parameter, nrm diatur 1, level *lexical* tetap 0 karena prosedur dideklarasikan di level global, dan adr menyimpan alamat prosedur dalam memori. Bersamaan dengan ini, kompilator membuat entri baru di btab untuk prosedur test, di mana kolom last, lpar, psze, dan vsze akan *di-update* ketika variabel dan parameter ditambahkan.

Setelah masuk ke blok prosedur test(), kompilator membaca deklarasi variabel lokal y. Variabel ini kemudian di-Insert ke tab dengan parameter: nama y, objek sebagai variabel, tipe integer, ref 0, nrm 1, level *lexical* 1 karena berada di dalam prosedur, dan adr diatur sebagai *offset* pertama dalam *stack frame* lokal. Kolom vsze pada entri btab untuk blok test diperbarui sesuai ukuran variabel lokal ini. Pada tahap ini, tab dan btab sudah berisi semua informasi tentang variabel global, prosedur, dan variabel lokal.

Ketika kompilator melihat ekspresi  $x := y + 1$ , ia melakukan proses Lookup untuk menemukan setiap identifier. Pertama, Lookup x dilakukan di tab: kompilator menemukan x pada level 0, sehingga mengetahui tipe, *offset*, dan level *lexical* untuk

akses memori global. Kedua, Lookup y dilakukan: kompilator menemukan y pada level 1, di *stack frame* lokal prosedur test. Setelah kedua *identifier* ditemukan, kompilator melakukan *type checking* untuk memastikan operasi  $y + 1$  sesuai dengan tipe x. Selanjutnya, *offset* dan level *lexical* digunakan untuk menghasilkan instruksi *load* dan *store* yang benar, sehingga ekspresi dapat dieksekusi dengan akses memori yang tepat.

## 1.4 Semantic Checking

Kompilator memanfaatkan Symbol Table untuk melakukan berbagai bentuk pemeriksaan. Tujuannya adalah mendeteksi kesalahan yang tidak dapat ditemukan melalui analisis sintaksis saja, seperti penggunaan variabel yang belum didefinisikan, deklarasi ganda, atau operasi yang melibatkan tipe yang tidak kompatibel. Hal tersebut memastikan bahwa parse tree yang telah dihasilkan benar-benar merepresentasikan program yang valid, baik secara sintaks maupun semantik.

Pemeriksaan yang dilakukan oleh *semantic analyzer* dapat dikategorikan menjadi tiga proses utama.

### 1.4.1 Duplication Checking

Pemeriksaan ini memastikan tidak terjadi pendefinisian nama yang sama lebih dari satu kali dalam ruang lingkup (*scope*) yang sama. Aturannya adalah sebuah nama *identifier* tidak boleh dideklarasikan ulang di level blok yang sama. Saat *semantic routine* memproses deklarasi, ia akan memeriksa Symbol Table pada *scope* aktif saat ini. Jika nama sudah ada sebelumnya, kesalahan “Identifier Redefinition” dimunculkan. Sebagai contoh, perhatikan kesalahan berikut.

```
variabel x: integer;
variabel x: real; // Duplikasi deklarasi pada scope yang sama
```

Dalam proses ini, ketika kompilator memasukkan sebuah simbol baru ke dalam Symbol Table, ia akan memeriksa terlebih dahulu apakah nama yang sama telah ada dalam *scope* yang sama. Jika iya, kompilator melaporkan kesalahan semantik.

### 1.4.2 Defined Checking

Pemeriksaan ini memastikan bahwa setiap *identifier* yang digunakan dalam *statement* program telah dideklarasikan sebelumnya. Aturannya adalah variabel atau fungsi harus ada di Symbol Table sebelum bisa digunakan dalam ekspresi. Implementasi dilakukan pada semua tempat penggunaan (ekspresi, *assignment*) kecuali pada bagian deklarasi itu sendiri. Lookup dilakukan dari *scope* terdalam hingga terluar. Jika tidak ditemukan, kesalahan “Undeclared Identifier” dimunculkan. Sebagai contoh, perhatikan kesalahan berikut.

```
y := 10; // Kesalahan: y belum pernah dideklarasikan
```

Dalam proses ini, kompilator memeriksa Symbol Table setiap kali menjumpai referensi ke suatu *identifier*. Jika *identifier* tersebut tidak ditemukan, kompilator menyimpulkan bahwa penggunaannya tidak valid.

#### 1.4.3 Type Checking

Pemeriksaan ini memastikan kesesuaian tipe data dalam *statement* dan operasi. Untuk operasi aritmatika, jika terdapat operasi  $A + B$ , tipe  $A$  dan  $B$  harus kompatibel (misalnya integer atau real). Untuk kondisional, ekspresi yang mengikuti if while harus mengevaluasi ke tipe boolean. Dan untuk assignment, tipe variabel target harus kompatibel dengan tipe hasil ekspresi di ruas kanan. Sebagai contoh, perhatikan kesalahan berikut.

```

variabel x: integer;
variabel y: boolean;

begin
    x := y + 3; // Kesalahan: boolean tidak dapat
    ditambahkan dengan integer
end.
  
```

Dalam proses ini, masing-masing *node* pada Parse Tree diberi atribut tipe melalui aturan-aturan dalam Attributed Grammar. Type Checking kemudian menggabungkan informasi tersebut untuk menentukan apakah operasi yang dilakukan valid.

#### 1.5 Syntax-Directed Translation

Syntax-Directed Translation adalah metode untuk mengasosiasikan aturan semantik dengan aturan produksi di dalam *grammar*. Hal ini menyatakan bahwa penentuan makna suatu konstruksi bahasa dapat ditentukan secara langsung berdasarkan struktur sintaktisnya. Dengan kata lain, setiap *grammar* tidak hanya mendefinisikan bagaimana simbol-simbol disusun, tapi juga memicu serangkaian aksi untuk menghitung nilai atau membangun struktur semantik seperti Abstract Syntax Tree (AST).

Secara formal, sebuah Syntax-Directed Translation Scheme merupakan Context-Free Grammar yang di dalam produksinya disisipkan fragmen program, yang dikenal sebagai *semantic actions*. Fragmen program ini memungkinkan eksekusi instruksi tertentu selama proses analisis sintaksis berlangsung. Bentuk umum sebuah produksi dalam Syntax-Directed Translation dapat dituliskan sebagai berikut.

$$A \rightarrow \{R_0\}X_1\{R_1\}X_2 \dots X_k\{R_k\}$$

Di sini, setiap  $R_i$  merepresentasikan *semantic action* yang dikaitkan dengan posisi tertentu dalam deretan simbol produksi. Aksi-aksi tersebut dieksekusi secara berurutan dari kiri ke kanan seiring dengan terjadinya reduksi aturan dalam proses *parsing*.

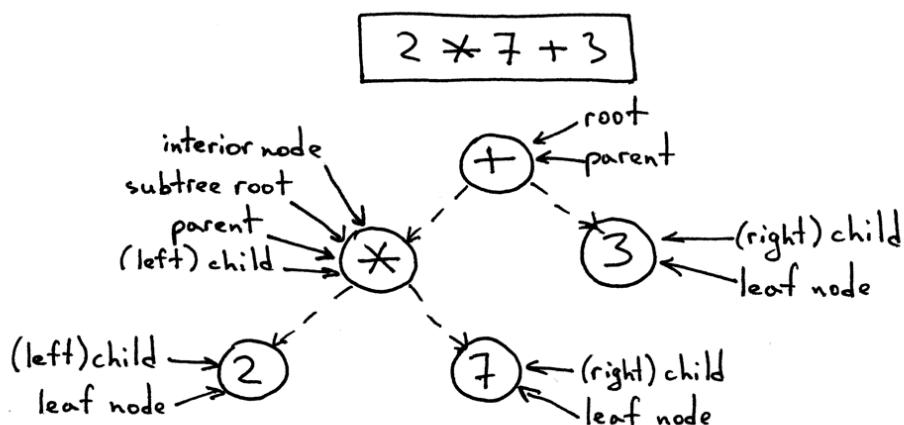
## 1.6 Abstract Syntax Tree (AST)

Abstract Syntax Tree (AST) adalah representasi *tree* dari struktur sintaks program yang telah diabstraksikan dari detail-detail *parsing*. Berbeda dengan Parse Tree yang secara ketat merepresentasikan struktur *grammar* formal, AST menghilangkan *node-node* yang tidak relevan sehingga hanya menyimpan informasi esensial yang diperlukan untuk tahap kompilasi selanjutnya.

### 1.6.1 Struktur dan Karakteristik AST

Dalam proses analisis sintaksis, *Syntax Analyzer* menghasilkan struktur pohon yang merepresentasikan struktur hierarkis program. Pada tahap analisis semantik, pohon ini digunakan dan ditransformasi menjadi bentuk yang lebih abstrak, yaitu *Abstract Syntax Tree* (AST).

AST terdiri atas *node-node* yang masing-masing merepresentasikan konstruksi semantik dari bahasa pemrograman. Setiap *node* umumnya tidak lagi merefleksikan seluruh aturan *grammar*, tetapi hanya menyimpan data esensial, seperti operator pada ekspresi biner, nilai literal, dan referensi ke *child nodes* yang merupakan *operand*. Sebagai contoh, perhatikan AST untuk ekspresi  $2 * 7 + 3$  berikut.



Gambar 1.3. AST untuk ekspresi  $2 * 7 + 3$  dengan penjelasannya  
(Sumber: R. Spivak, Ruslan's Blog, 2016)

### 1.6.2 Perbedaan Parse Tree dan AST

Parse Tree memuat seluruh detail derivasi grammar, termasuk simbol-simbol terminal yang tidak esensial secara semantik seperti (;), kata kunci *begin / end*, dan tanda kurung. Sedangkan AST merepresentasikan esensi instruksi program dengan membuang detail sintaks yang tidak perlu. *Node* dalam AST merepresentasikan konstruksi logika nyata seperti *AssignmentNode* atau *BinaryExpressionNode*. Transisi ke AST penting untuk efisiensi memori dan memudahkan traversasi saat melakukan validasi semantik.

### 1.6.3 Decorated AST

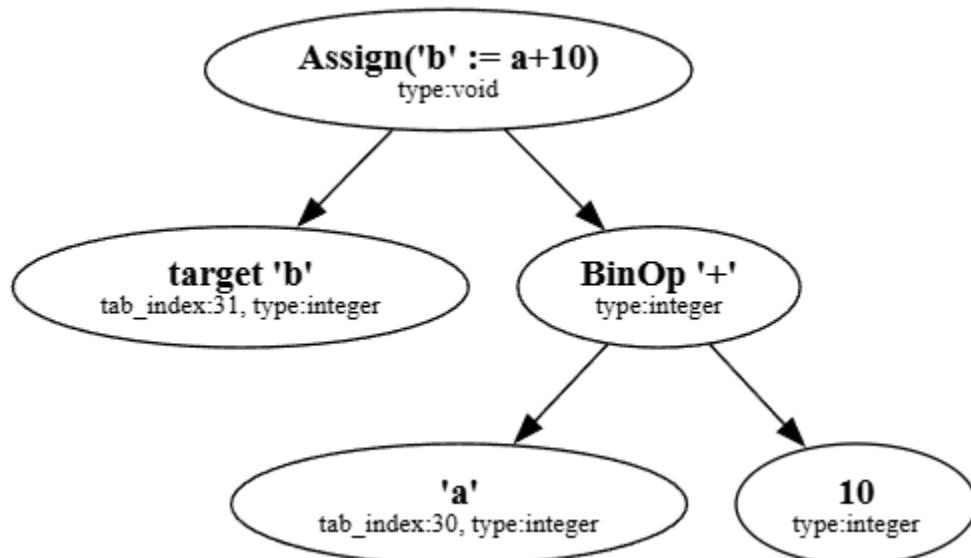
AST dapat diperkaya dengan berbagai atribut semantik. Versi yang telah diperluas ini disebut Decorated AST. Beberapa jenis informasi yang biasanya ditambahkan ke node AST antara lain adalah

- Tipe data untuk setiap ekspresi
- Referensi ke Symbol Table untuk *identifier*
- Informasi *scope level* untuk deklarasi

Sebagai contoh, untuk pernyataan

b := a + 10

Decorated AST yang dapat dihasilkan adalah



Gambar 1.4. Decorated AST untuk pernyataan  $b := a + 10$   
(Sumber: Dokumentasi kelompok)

### 1.7 Error dalam Semantic Analysis

Berbeda dengan kesalahan sintaksis yang muncul ketika struktur program tidak sesuai dengan *grammar*, kesalahan semantik terjadi ketika konstruksi program secara sintaktis benar, tapi tidak memiliki makna atau melanggar aturan semantik bahasa pemrograman. Beberapa jenis kesalahan semantik yang umum ditemui dalam tahap analisis semantik antara lain adalah

1. Undeclared Identifier

Kesalahan ini terjadi ketika sebuah variabel, fungsi, atau prosedur digunakan sebelum dideklarasikan. Contohnya

```
x := 10; // Kesalahan: x belum dideklarasikan
```

## 2. Redefinition

Terjadi ketika sebuah *identifier* dideklarasikan lebih dari satu kali dalam *scope* yang sama. Contohnya

```
var a: integer;
var a: real; // Kesalahan: a sudah dideklarasikan
sebelumnya
```

## 3. Type Mismatch

Muncul ketika sebuah operasi dilakukan pada operand dengan tipe yang tidak kompatibel dengan operatornya atau konteksnya. Contohnya

```
result := 'abc' + 10; // Kesalahan: operasi + tidak
didefinisikan untuk string dan integer
```

## 4. Scope Violation

Kesalahan ini muncul ketika sebuah *identifier* diakses di luar lingkup (*scope*) tempat ia dideklarasikan. Contohnya

```
begin
  var x := 5;
end;
y := x; // Kesalahan: x di luar scope
```

## 5. Constant Modification

Mencoba mengubah nilai konstanta atau objek yang dideklarasikan sebagai *immutable*. Contohnya

```
const pi = 3.14;
pi := 3.15; // Kesalahan: konstanta tidak boleh
dimodifikasi
```

## 1.8 Visit Functions

Untuk mengimplementasikan logika pemeriksaan di atas pada struktur AST, digunakan Visit Functions. Visitor (Semantic Analyzer) bertindak sebagai pengunjung yang menelusuri *node* AST. Visitable (AST Nodes) menerima kunjungan dan memberikan informasi yang dibutuhkan seperti nama variabel atau tipe data kepada visitor untuk diverifikasi terhadap Symbol Table.

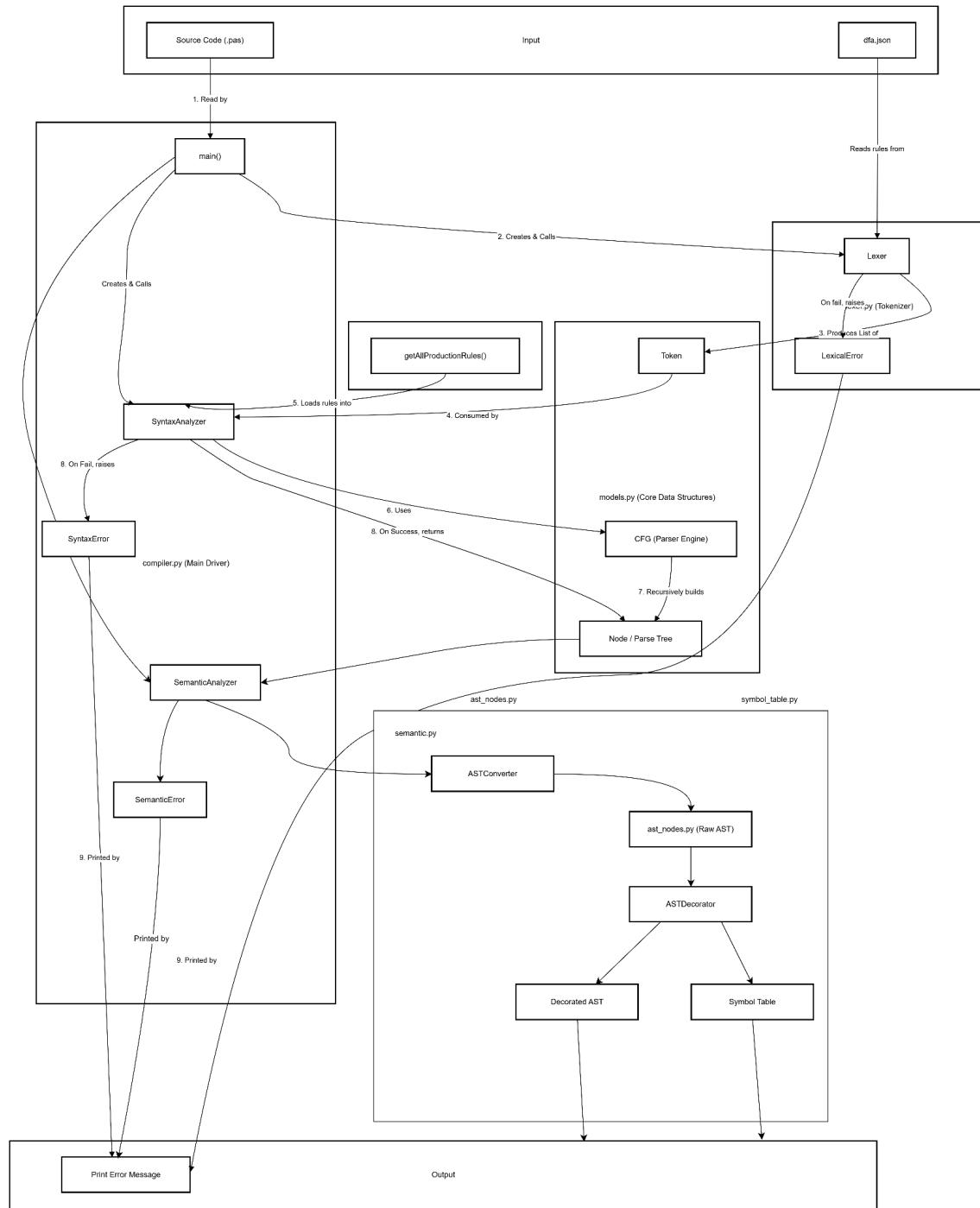
Setiap jenis *node* dalam AST menyediakan fungsi *visit* dengan penamaan *visit\_<node>* untuk menjalankan *semantic action* tertentu pada *node* tersebut. Dalam

prosesnya, fungsi *visit* umumnya menjalankan empat tugas utama. Pertama, fungsi tersebut memanggil operasi *visit* pada *child node* untuk memastikan semua substruktur diproses secara rekursif. Kedua, fungsi ini dapat mengakses atau memperbarui Symbol Table, misalnya untuk operasi deklarasi variabel. Ketiga, fungsi visitor menghitung atribut atau tipe semantik yang relevan seperti penentuan tipe ekspresi atau verifikasi kesesuaian tipe. Keempat, fungsi tersebut dapat menambahkan anotasi pada *node*.

## BAB II

# PERANCANGAN DAN IMPLEMENTASI

### 2.1 Arsitektur Sistem



Gambar 2.1. Arsitektur Sistem

Alur pemrosesan data berlangsung secara sekuensial melalui tahapan berikut:

1. Inisialisasi dan Parsing: Sistem dimulai dengan Lexer yang mengubah kode sumber menjadi aliran token, diikuti oleh SyntaxAnalyzer (menggunakan CFG dan recursive descent) yang menyusun token tersebut menjadi Parse Tree.
2. Konversi AST (Intermediate Representation): Parse Tree yang dihasilkan kemudian diserahkan kepada kelas ASTConverter. Modul ini menerapkan skema Syntax-Directed Translation untuk mentransformasi struktur pohon sintaksis yang kompleks menjadi Abstract Syntax Tree (AST). Node-node AST ini didefinisikan dalam `ast_nodes.py` sebagai objek data (dataclasses) yang merepresentasikan logika program secara hierarkis tanpa noise sintaksis.
3. Analisis Semantik: Inti dari tahap ini dijalankan oleh SemanticAnalyzer (atau ASTDecorator). Kelas ini mengimplementasikan Visitor Pattern untuk menelusuri setiap node dalam AST. Selama penelusuran, analyzer berinteraksi dengan SymbolTable untuk:
  - a. Menyimpan informasi deklarasi identifier (variabel, konstanta, fungsi) ke dalam tabel tab, btab, dan atab.
  - b. Memvalidasi aturan lingkup (scope checking) dan tipe data (type checking).
  - c. Melakukan "dekorasi" pada AST dengan menambahkan informasi tipe dan referensi tabel simbol ke dalam atribut node.
4. Visualisasi: Hasil akhir berupa Decorated AST yang telah tervalidasi.

## 2.2 Desain Struktur Data

Desain struktur data merupakan pondasi penting dalam implementasi analisis semantik yang efisien dan maintainable. Terdapat tiga struktur data utama yang dirancang untuk mendukung seluruh proses analisis semantik, yaitu AST nodes, symbol table, dan Parse Tree Nodes. Setiap struktur data dirancang dengan mempertimbangkan kebutuhan fungsional, efisiensi akses, serta kemudahan pemeliharaan kode.

### 2.2.1 Abstract Syntax Tree Nodes

Abstract Syntax Tree nodes merepresentasikan konstruksi-konstruksi dalam bahasa Pascal-S dalam bentuk struktur *tree* yang hierarkis. Desain hierarki AST node mengikuti prinsip object-oriented dengan *base class* ASTNode yang mendefinisikan atribut dan *behavior common* untuk semua jenis node, kemudian *derived classes* untuk setiap konstruksi spesifik seperti *declaration*, *statement*, dan *expression*.

Kategori	Node Class	Atribut	Fungsi
Base	ASTNode	type, symbol_entry	Base class untuk

Kategori	Node Class	Atribut	Fungsi
			semua node AST
Program	ProgramNode	name, declarations, block	Root dari AST
Declarations	ConstDeclNode	const_name, value	Deklarasi konstanta
	TypeDeclNode	type_name, value	Deklarasi tipe
	VarDeclNode	var_name, type_node	Deklarasi variabel
Statement	IfNode	condition, true_block, else_block	Representasi struktur percabangan (if-then-else)
	WhileNode	condition, body	Representasi perulangan while
	ForNode	variable, start_expr, direction, end_expr, body	Representasi perulangan for
	AssignNode	target, value	Representasi assignment
	ProcedureCallNode	proc_name, arguments	Representasi pemanggilan prosedur
	CompoundNode	children	Representasi blok kumpulan statement
Subprogram	ProcedureDeclNode	name, params, declarations, body	Deklarasi Prosedure
	FunctionDeclNode	name, return_type, params, declarations, body	Deklarasi Fungsi
	ParameterNode	names, type_node, is_ref	Definisi Parameter

Kategori	Node Class	Atribut	Fungsi
Expression	BinOpNode	Up, left, right	Operasi biner (+, -, *, /, >, <, dll)
	UnaryOpNode	Op, expr	Operasi unary (-, not)
	VarNode	name	Referensi penggunaan variabel
	NumNode	value	Nilai literal angka (integer/real)
	StringNode	value	Nilai literal string
	BoolNode	value	Nilai literal boolean (true/false)

Tabel 2.1. Hierarki kelas AST Node

Berikut ini adalah contoh struktur AST untuk program sederhana.

```
program Test;
var x: integer;
mulai
  x := 5 + 3;
selesai.
```

```
[DEBUG] Abstract Syntax Tree (AST)
Program('Test')
  +- Declarations
    |  \-- VarDecl(name: 'x', type: 'integer')
  \-- Block
    \-- Assign(target: var('x'), value:
      \-- BinOp('+')
        +- Num(5)
        \-- Num(3)
```

## 2.2.2 Symbol Table Structure

Symbol Table merupakan struktur data krusial yang menyimpan informasi tentang semua *identifier* yang dideklarasikan dalam program. Implementasi Symbol Table untuk Pascal-S menggunakan pendekatan *multi-table* yang terdiri dari tiga tabel

terpisah. Desain ini memfasilitasi berbagai jenis informasi yang perlu disimpan untuk berbagai jenis *identifier* dan *composite types*.

Tabel pertama adalah tab atau *identifier table* yang merupakan tabel utama untuk menyimpan informasi semua *identifier* dalam program. Setiap entri dalam tab merepresentasikan satu identifier dengan delapan atribut berikut.

Field	Deskripsi	Contoh Value
identifiers	Nama identifier	"x", "hitung", "MAX
link	Pointer/indeks ke identifier sebelumnya dalam scope yang sama.	30, 0
obj	Kelas objek yang dienumerasi: konstanta, variabel, tipe, prosedur, fungsi, dll.	variable, constant, procedure
typ	Tipe dasar dari identifier, misalnya: integer, boolean, char, real, array, record, dll.	1 (integer), 2 (boolean)
ref	Pointer/indeks ke tabel lain jika tipe adalah komposit (array/record). Mengarah ke atab (array table) atau btab (record/procedure block).	Indeks ke atab atau btab
nrm	Menandai apakah identifier adalah variabel normal (=1) atau parameter by-reference (var parameter) (=0).	1 (normal), 0 (ref)
lev	Tingkat lexical level tempat identifier dideklarasikan (0= global, 1= dalam prosedur, 2= dalam prosedur di dalam prosedur, dst).	0, 1, 2
adr	Makna tergantung jenis objek: offset variabel di stack frame, nilai konstanta, offset field record, alamat prosedur, atau ukuran/penanda lain.	0, 4, 100

Tabel 2.2. Struktur entri tab

Tabel kedua adalah btab atau *block table* yang menyimpan informasi tentang blok-blok dalam program, dimana setiap blok merepresentasikan *scope* baru seperti program utama, prosedur, fungsi, atau *record type definition*. Setiap entri btab memiliki lima atribut berikut.

Field	Deskripsi	Range
blocks	Indeks entri block	0...N
last	Pointer/indeks ke identifier terakhir yang dideklarasikan di dalam block tersebut.	Indeks ke tab
lpar	Pointer/indeks ke parameter terakhir dari prosedur/fungsi pada block tersebut. Jika block adalah record, nilainya 0.	Indeks ke tab
psze	Total ukuran parameter block (dalam byte/unit memori).	Integer $\geq 0$
vsze	Total ukuran variabel lokal block (dalam byte/unit memori).	Integer $\geq 0$

Tabel 2.3. Struktur entri btab

Tabel ketiga adalah atab atau *array table* yang khusus menyimpan informasi detail tentang array types. Setiap entri atab merepresentasikan satu array type dengan delapan atribut berikut.

Field	Deskripsi	Range
arrays	Indeks entri array	1...M
xtyp	Tipe indeks array (misalnya integer). Berupa kode tipe dari tabel tab	Kode tipe (misal 1)
etyp	Tipe elemen array (misalnya integer). Berupa kode tipe dari tabel tab.	Kode Tipe
eref	Pointer/indeks ke detail tipe elemen jika elemen adalah tipe komposit (misalnya array dalam array, atau record). Mengarah ke atab atau btab.	indeks ke atab/btab

Field	Deskripsi	Range
low	Batas bawah indeks array (misalnya 1 pada array[1..10] atau 0 pada array[0..15]).	Integer
high	Batas atas indeks array.	Integer
elsz	Ukuran satu elemen array (dalam byte/unit memori).	Integer $\geq 1$
size	Total ukuran array.	Integer $\geq 0$

Tabel 2.4. Struktur entri atab

### 2.2.3 Error Representation

Struktur data untuk *error representation* dirancang untuk melaporkan kesalahan dengan informasi yang cukup untuk *debugging*.

Tipe Error	Parameter	Contoh Message
Undeclared Identifier	name (Nama variabel/prosedur)	"Variable '{node.name}' not declared." atau "Identifier '{node.proc_name}' not declared."
Duplicate Identifier	name (Nama identifier)	"Duplicate identifier '{name}' in the same scope." atau "Failed to add variable '{node.var_name}'"
Type Mismatch (Assignment)	value_type, target_type	"Type mismatch in assignment. Cannot assign {value_type.name} to {target_type.name}"
Type Mismatch (Condition)	cond_type (Tipe ekspresi)	"IF condition must be BOOLEAN, got {cond_type.name}" atau "WHILE condition must be BOOLEAN..."
Invalid Operation	op, operand_type	"Operator '{op}' only for INTEGER." atau "Operator '{op}' requires BOOLEAN operands."
Array Index Type Error	index_type	"Array index must be INTEGER, got {index_type.name}"
Array Bounds Error	val, low, high	"Array index out of bounds: {val}. Valid: [{atab_entry.low}..{atab_entry.high}]"
Not An Array	-	"Variable is not an array."
Invalid Loop Limits	-	"FOR loop limits must be INTEGER."

Tipe Error	Parameter	Contoh Message
Invalid Array Definition	low_val, high_val	"Array lower bound ({low_val}) > upper bound ({high_val})"

Tabel 2.5. Jenis-jenis Semantic Error

## 2.3 Algoritma dan Mekanisme Implementasi

Implementasi analisis semantik menggunakan serangkaian algoritma yang saling terintegrasi untuk mencapai tujuan validasi semantik dan pembangunan Decorated AST.

### 2.3.1 Algoritma Pembangunan AST

Pembangunan AST dari Parse Tree memanfaatkan Syntax-Directed Translation Scheme yang mengimplementasikan *semantic actions* sesuai dengan aturan produksi *grammar*. Algoritma ini bekerja secara rekursif dengan melakukan *depth-first traversal* pada Parse Tree dengan setiap *node* diproses melalui pemanggilan *visitor method* yang sesuai dengan jenis nonterminal atau terminal yang direpresentasikan oleh *node* tersebut. *Visitor method* mengekstrak informasi yang relevan dari *children nodes*, menerapkan *semantic rule* yang sesuai, dan membentuk AST node baru yang merepresentasikan konstruksi tersebut secara abstrak.

### 2.3.2 Algoritma Type Checking

Type Checking memvalidasi bahwa semua operasi dalam program dilakukan pada tipe data yang kompatibel. Algoritma ini bekerja dengan melakukan bottom-up traversal pada AST dengan tipe setiap node dihitung berdasarkan tipe *children*-nya. Untuk *expression nodes*, tipe dihitung dengan mempertimbangkan operator dan tipe *operands*. Untuk *statement nodes*, pemeriksaan tipe memastikan bahwa *operands* memiliki tipe yang sesuai dengan *statement* tersebut.

### 2.3.3 Algoritma Scope Resolution

Scope Resolution menentukan *identifier* mana yang direferensi ketika ada *identifier* dengan nama yang sama di berbagai *scope*. Pascal-S menggunakan *lexical scoping* atau *static scoping* dengan *scope* ditentukan oleh struktur teks program dan tidak bergantung pada *execution flow*. Algoritma ini mengimplementasikan *most closely nested scope rule*, dimana *identifier* yang digunakan akan mereferensi deklarasi yang paling dekat dalam *structure nesting*, dicari dari *inner scope* ke *outer scope*.

### 2.3.4 Algoritma Symbol Table Insertion

Insertion *identifier* ke Symbol Table dilakukan ketika *semantic analyzer* mengunjungi *declaration nodes*. Algoritma ini memastikan bahwa setiap *identifier* dicatat dengan informasi yang lengkap dan konsisten dalam Symbol Table. Proses *insertion* berbeda tergantung jenis *identifier* yang dideklarasikan, namun mengikuti pola umum yang sama, yaitu memvalidasi bahwa tidak ada *redeclaration* dalam *current*

*scope*, menentukan tipe dan atribut lain dari *identifier*, membuat entri baru di tab, dan meng-update entri btab untuk *current block*.

### 2.3.5 Algoritma AST Decoration

Decoration menambahkan informasi semantik ke AST nodes setelah Semantic Checking selesai. Algoritma ini mengunjungi setiap node dalam AST dan mengisi atribut-atributnya dengan informasi yang telah dikumpulkan selama Semantic Checking. Decoration dilakukan secara *in-place* pada AST yang sama, mengubah raw AST menjadi Decorated AST yang kaya dengan informasi semantik.

## 2.4 Mekanisme Error Handling

Error Handling dalam Semantic Analysis dirancang untuk mendeteksi berbagai jenis kesalahan semantik dan melaporkannya dengan informasi yang jelas. Deteksi error dilakukan secara *distributed* dengan setiap *visitor method* dalam *semantic analyzer* bertanggung jawab untuk mendeteksi kesalahan yang relevan dengan konstruksi yang sedang diproses. Kesalahan semantik dikategorikan berdasarkan aspek semantik yang dilanggar

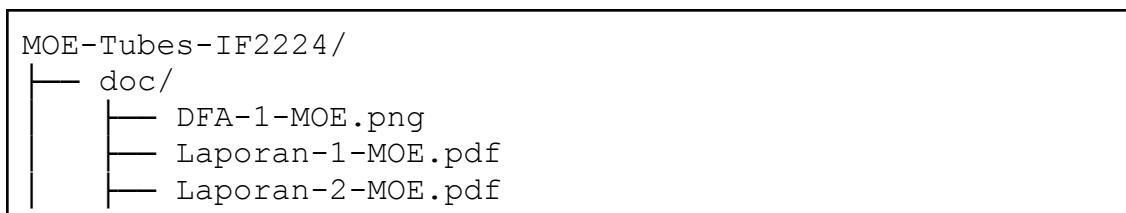
Kategori	Tipe Error	Contoh Kasus
Scope & Declaration	Undeclared Identifier	<pre>program TestUndeclaredVar; variabel   declared: integer; mulai   declared := 5;   undeclared := 10; {Error: Variabel 'undeclared' belum dideklarasikan}   writeln(declared);   writeln(undeclared); selesai.</pre>
	Duplicate Identifier	<pre>program TestRedeclaration; variabel   x: integer;   x: real; {Error: Identifier 'x' dideklarasikan ulang dalam scope yang sama}   y: integer; mulai   x := 67;   y := x + 2;   writeln(y); selesai.</pre>
	Constant Modification	<pre>program TestConstModification; konstanta   MAX = 100; variabel   a: integer; mulai   a := 10;   MAX := 200; {Error: Mencoba mengubah nilai konstanta 'MAX'} selesai.</pre>

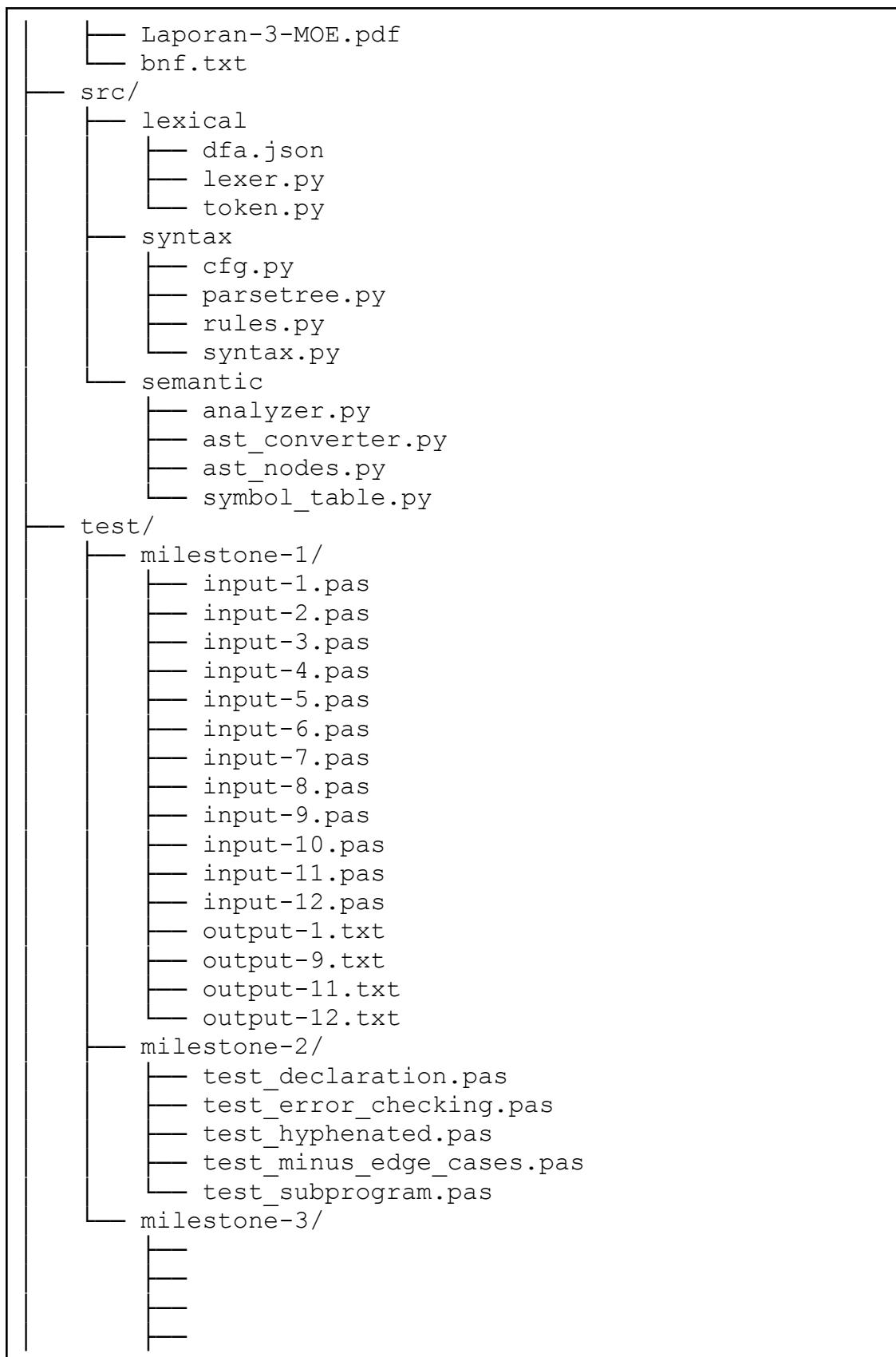
Kategori	Tipe Error	Contoh Kasus
Type Compatibility	Assignment Mismatch	<pre>program TestTypeMismatch; variabel   x: integer;   flag: boolean; mulai   x := 10;   flag := x; {Error: Tidak kompatibel assign Integer ke Boolean} selesai.</pre>
	Type Mismatch (Real to Int)	<pre>program TestTypeMismatch; variabel   x: integer;   y: real; mulai   y := 3.14;   x := y; {Error: Tidak bisa assign Real ke Integer secara implisit} selesai.</pre>
	Invalid Condition (Non-Boolean)	<pre>program TestInvalidCondition; variabel x: integer; mulai   jika x + 5 maka {Error: Kondisi IF harus tipe Boolean, bukan Integer}   x := x + 1 selesai.</pre>
Array Semantics	Not An Array	<pre>program TestNotArray; variabel x: integer; mulai   x[1] := 10; {Error: Variabel 'x' bukan tipe larik/array} Selesai</pre>
	Index Type Error	<pre>program TestIndexError; variabel data: larik [1..10] dari integer; mulai   data[3.5] := 10; {Error: Indeks array harus bertipe Integer, bukan Real} selesai.</pre>

Tabel 2.6. Kategori Semantic Error

## 2.5 Implementasi Algoritma

### 2.5.1 Struktur File







## 2.5.2 Kelas dan Fungsi

Kelas dan fungsi diimplementasikan dalam bahasa pemrograman python. Adapun kelas dan fungsi yang ditulis dalam laporan ini adalah kelas dan fungsi yang berkaitan dengan analisis semantic tanpa menyertakan kelas dan fungsi dari milestone sebelumnya.

### 1. ast\_nodes.py: AST Nodes

```

from dataclasses import dataclass, field, fields
from typing import List, Optional, Any, Union
from lexical.token import Token

@dataclass
class ASTNode:
    type: Optional[str] = field(default=None, init=False, repr=False)
    symbol_entry: Optional[dict] = field(default=None, init=False, repr=False)

    def __str__(self):
        # return self._print_tree()
        return self._print_ast_decorated()

@dataclass
class ProgramNode(ASTNode):
    name: str
    declarations: List[ASTNode] = field(repr=False)
    block: ASTNode = field(repr=False)

@dataclass
class VarDeclNode(ASTNode):
    var_name: str
    type_node: 'TypeNode'

@dataclass
class ConstDeclNode(ASTNode):
    const_name: str
    value: ASTNode

@dataclass
class TypeDeclNode(ASTNode):
    type_name: str
    value: ASTNode

@dataclass
class TypeNode(ASTNode):
    type_name: str
    def __repr__(self): return f'{self.type_name}'
    def __str__(self): return f'{self.type_name}'

@dataclass
class ArrayTypeNode(ASTNode):
  
```

```

        lower: ASTNode
        upper: ASTNode
        element_type: ASTNode

@dataclass
class RecordTypeNode(ASTNode):
    fields: List['VarDeclNode']

@dataclass
class CompoundNode(ASTNode):
    children: List[ASTNode]

@dataclass
class AssignNode(ASTNode):
    target: ASTNode
    value: ASTNode
    def _print_tree(self, prefix="", is_last=True):
        """Override untuk handle multi-line assignment"""
        return self._print_tree_multiline(prefix, is_last)

@dataclass
class ProcedureCallNode(ASTNode):
    proc_name: str
    arguments: List[ASTNode]

@dataclass
class IfNode(ASTNode):
    condition: ASTNode
    true_block: ASTNode
    else_block: Optional[ASTNode] = None

@dataclass
class WhileNode(ASTNode):
    condition: ASTNode
    body: ASTNode

@dataclass
class RepeatNode(ASTNode):
    body: List[ASTNode]
    condition: ASTNode

@dataclass
class ForNode(ASTNode):
    variable: str
    start_expr: ASTNode
    direction: str
    end_expr: ASTNode
    body: ASTNode

@dataclass
class CaseElementNode(ASTNode):
    value: ASTNode
    statement: ASTNode

@dataclass
class CaseNode(ASTNode):
    expr: ASTNode
    cases: List[CaseElementNode]

@dataclass

```

```
class BinOpNode(ASTNode):
    op: str
    left: ASTNode
    right: ASTNode

@dataclass
class UnaryOpNode(ASTNode):
    op: str
    expr: ASTNode

@dataclass
class VarNode(ASTNode):
    name: str
    def __repr__(self): return f"Var('{self.name}')"

@dataclass
class ArrayAccessNode(ASTNode):
    array: ASTNode
    index: ASTNode

@dataclass
class FieldAccessNode(ASTNode):
    record: ASTNode
    field_name: str

@dataclass
class NumNode(ASTNode):
    value: Union[int, float]
    def __repr__(self): return f"Num({self.value})"

@dataclass
class StringNode(ASTNode):
    value: str
    def __repr__(self): return f"String({self.value})"

@dataclass
class CharNode(ASTNode):
    value: str

@dataclass
class BoolNode(ASTNode):
    value: bool

@dataclass
class NoOpNode(ASTNode):
    pass

@dataclass
class ParameterNode(ASTNode):
    names: List[str]
    type_node: ASTNode
    is_ref: bool = False

@dataclass
class ProcedureDeclNode(ASTNode):
    name: str
    params: List[ParameterNode]
    local_vars: List[ASTNode]
    block: ASTNode
```

```
@dataclass
class FunctionDeclNode(ASTNode):
    name: str
    return_type: TypeNode
    params: List[ParameterNode]
    local_vars: List[ASTNode]
    block: ASTNode
```

## 2. Ast\_converter.py : AST Convert

```
class ASTConverter:
    def convert(self, parse_tree_root: Node) -> ASTNode:
        """Entry point untuk konversi Parse Tree ke AST"""
        if parse_tree_root.value == NonTerminal("<S>"):
            return self.convert(parse_tree_root.children[0])

        if parse_tree_root.value == NonTerminal("<Program>"):
            return self._convert_Program(parse_tree_root)

        raise ValueError(f"Root bukan <Program> atau <S>, tapi {parse_tree_root.value}")

    def visit(self, node):
        """Helper untuk menavigasi node secara dinamis."""
        if isinstance(node, Node):
            clean_name = str(node.value).replace('<', '').replace('>', '').replace('-', '_')

            # --- MAPPING MANUAL ---
            if clean_name == "VarDeclOpt": method_name = "_convert_VarDeclarationPart"
            elif clean_name == "ConstDeclOpt": method_name = "_convert_ConstDeclarationPart"
            elif clean_name == "TypeDeclOpt": method_name = "_convert_TypeDeclarationPart"
            elif clean_name == "SubprogDeclList": method_name = "_convert_SubprogDeclList"
            elif clean_name == "StatementPart": method_name = "_convert_StatementPart"
            else: method_name = '_convert_' + clean_name

            method = getattr(self, method_name, self._generic_visit)
            return method(node)
        return node

    def _generic_visit(self, node):
        return None

    # --- HELPERS ---
    def _get_lexeme(self, node_or_token) -> str:
        token = self._get_token(node_or_token)
        return token.lexeme if token else ""

    def _get_token(self, node_or_token) -> Optional[Token]:
        if isinstance(node_or_token, Token): return node_or_token
        if isinstance(node_or_token, Node):
            if isinstance(node_or_token.value, Token): return node_or_token.value
            if str(node_or_token.value) == "EPSILON": return None
```

```

        if node_or_token.children:
            for child in node_or_token.children:
                found = self._get_token(child)
                if found: return found
        return None

    def _get_operator_lexeme(self, node: Node) -> str:
        return self._get_lexeme(node)

    # ===== PROGRAM STRUCTURE =====
    def _convert_Program(self, node: Node) -> ProgramNode:
        header = node.children[0]
        prog_name = self._get_lexeme(header.children[1])

        child_1 = node.children[1]
        if str(child_1.value) == "<Block>":
            decl_node = child_1.children[0]
            stmt_part = child_1.children[1]
            declarations = self._convert_DeclarationPart(decl_node)
            compound = self.visit(stmt_part)
        else:
            declarations = self._convert_DeclarationPart(child_1)
            compound = self._convert_CompoundStatement(node.children[2])

        return ProgramNode(name=prog_name, declarations=declarations,
                           block=compound)

    def _convert_StatementPart(self, node: Node) -> CompoundNode:
        return self._convert_CompoundStatement(node.children[0])

    def _convert_DeclarationPart(self, node: Node) -> List[ASTNode]:
        decls = []
        if not node.children: return decls
        for child in node.children:
            if isinstance(child, Node) and str(child.value) == "EPSILON":
                continue
            res = self.visit(child)
            if isinstance(res, list):
                decls.extend(res)
            elif res and not isinstance(res, NoOpNode):
                decls.append(res)
        return decls

    # ===== DECLARATIONS =====
    def _convert_ConstDeclarationPart(self, node: Node) -> List[ConstDeclNode]:
        if not node.children or str(node.children[0].value) == "EPSILON": return []
        if len(node.children) > 1: return self.visit(node.children[1])
        return []

    def _convert_ConstList(self, node: Node) -> List[ConstDeclNode]:
        if not node.children or str(node.children[0].value) == "EPSILON": return []
        consts = []
        const_name = self._get_lexeme(node.children[0])
        val_node = self.visit(node.children[2])
        consts.append(ConstDeclNode(const_name=const_name, value=val_node))
        if len(node.children) > 4:
            rest = self.visit(node.children[4])
            if rest: consts.extend(rest)
        return consts

```

```

        return consts

    def _convert_TypeDeclarationPart(self, node: Node) -> List[TypeDeclNode]:
        if not node.children or str(node.children[0].value) == "EPSILON": return []
        return self.visit(node.children[1])

    def _convert_TypeList(self, node: Node) -> List[TypeDeclNode]:
        if not node.children or str(node.children[0].value) == "EPSILON": return []
        types = []
        type_name = self._get_lexeme(node.children[0])
        type_val = self.visit(node.children[2])
        types.append(TypeDeclNode(type_name=type_name, value=type_val))
        if len(node.children) > 4:
            rest = self.visit(node.children[4])
            if rest: types.extend(rest)
        return types

    def _convert_VarDeclarationPart(self, node: Node) -> List[VarDeclNode]:
        if not node.children or str(node.children[0].value) == "EPSILON": return []
        if len(node.children) > 1:
            if "List" in str(node.children[1].value): return
        self.visit(node.children[1])
        vars_list = []
        res = self.visit(node.children[1])
        if res: vars_list.extend(res)
        if len(node.children) > 2: self._collect_var_prime(node.children[2], vars_list)
        return vars_list
        return []

    def _convert_VarDeclList(self, node: Node) -> List[VarDeclNode]:
        if not node.children or str(node.children[0].value) == "EPSILON": return []
        vars_list = []
        curr = self.visit(node.children[0])
        if curr: vars_list.extend(curr)
        if len(node.children) > 2:
            vars_list.extend(self._convert_VarDeclList(node.children[2]))
        return vars_list

    def _collect_var_prime(self, node: Node, vars_list):
        if not node.children or str(node.children[0].value) == "EPSILON": return
        vars_list.extend(self.visit(node.children[0]))
        if len(node.children) > 1: self._collect_var_prime(node.children[1], vars_list)

    def _convert_VarDeclaration(self, node: Node) -> List[VarDeclNode]:
        ids = self.visit(node.children[0])
        type_node = self.visit(node.children[2])
        result = [VarDeclNode(var_name=name, type_node=type_node) for name in
ids]
        # print(f"[DEBUG Converter] Creating VarDecl: {[r.var_name for r in
result]}")
        return result

    def _convert_IdentifierList(self, node: Node) -> List[str]:
        ids = [self._get_lexeme(node.children[0])]


```

```

        if len(node.children) > 1: self._collect_ids(node.children[1], ids)
        return ids

    def _collect_ids(self, node:Node, ids):
        if not node.children or str(node.children[0].value) == "EPSILON": return
        ids.append(self._get_lexeme(node.children[1]))
        if len(node.children) > 2: self._collect_ids(node.children[2], ids)

    # ===== TYPES & SUBPROGRAMS =====
    def _convert_Type(self, node: Node) -> ASTNode:
        return self.visit(node.children[0])

    def _convert_SimpleType(self, node: Node) -> TypeNode:
        return TypeNode(type_name=self._get_lexeme(node))

    def _convert_ArrayType(self, node: Node) -> ArrayTypeNode:
        range_node = node.children[2]
        elem_type = self.visit(node.children[5])
        lower = self.visit(range_node.children[0])
        upper = self.visit(range_node.children[2])
        return ArrayTypeNode(lower=lower, upper=upper, element_type=elem_type)

    def _convert_RecordType(self, node: Node) -> RecordTypeNode:
        fields = []
        if len(node.children) > 1: fields = self.visit(node.children[1])
        return RecordTypeNode(fields=fields)

    def _convert_FieldList(self, node: Node) -> List[VarDeclNode]:
        fields = []
        self._collect_fields(node, fields)
        return fields

    def _collect_fields(self, node:Node, fields):
        if not node.children or str(node.children[0].value) == "EPSILON": return
        f = self.visit(node.children[0])
        if f: fields.extend(f)
        if len(node.children) > 1: self._collect_fields(node.children[1], fields)

    def _convert_SubprogDeclList(self, node: Node) -> List[ASTNode]:
        if not node.children or str(node.children[0].value) == "EPSILON": return
        subs = []
        sub = self.visit(node.children[0])
        if sub: subs.append(sub)
        if len(node.children) > 2: subs.extend(self.visit(node.children[2]))
        return subs

    def _convert_SubprogramDeclaration(self, node: Node):
        return self.visit(node.children[0])

    def _convert_ProcedureDeclaration(self, node: Node) -> ProcedureDeclNode:
        name = self._get_lexeme(node.children[1])
        params = []
        local_vars = []
        block = None
        for child in node.children:
            val = str(child.value)
            if "FormalParam" in val:
                res = self.visit(child)

```

```

        if res: params = res
    elif "Block" in val:
        # Unpack Block
        decl_part = child.children[0]
        stmt_part = child.children[1]
        local_decls = self._convert_DeclarationPart(decl_part)
        body = self.visit(stmt_part)
        block = CompoundNode(children=body.children)
    return ProcedureDeclNode(name=name, params=params, local_vars=
local_decls, block=block)

def _convert_FunctionDeclaration(self, node: Node) -> FunctionDeclNode:
    name = self._get_lexeme(node.children[1])
    params = []
    return_type = None
    block = None
    for child in node.children:
        val = str(child.value)
        if "FormalParam" in val:
            res = self.visit(child)
            if res: params = res
        elif "Type" in val and "Decl" not in val:
            return_type = self.visit(child)
        elif "Block" in val:
            decl_part = child.children[0]
            stmt_part = child.children[1]

            local_decls = self._convert_DeclarationPart(decl_part)
            body = self.visit(stmt_part)

            block = self.visit(child.children[1])
    return FunctionDeclNode(name=name, return_type=return_type,
params=params, local_vars = local_decls, block=block)

def _convert_FormalParamOpt(self, node: Node):
    if not node.children or str(node.children[0].value) == "EPSILON": return
[]
    return self.visit(node.children[0])

def _convert_FormalParameterList(self, node: Node):
    return self.visit(node.children[1])

def _convert_ParamSectionList(self, node: Node) -> List[ParameterNode]:
    params = []
    p = self.visit(node.children[0])
    if p: params.extend(p)
    if len(node.children) > 1:
        self._collect_param_section_prime(node.children[1], params)
    return params

def _collect_param_section_prime(self, node:Node, params):
    if not node.children or str(node.children[0].value) == "EPSILON": return
    p = self.visit(node.children[1])
    if p: params.extend(p)
    if len(node.children) > 2:
        self._collect_param_section_prime(node.children[2], params)

def _convert_ParamSection(self, node: Node) -> List[ParameterNode]:
    """Handle: VAR? <IdentifierList> : <Type>"""
    is_ref = False

```

```

        start_idx = 0

        # 1. Cek Keyword VAR/VARIABEL
        first_child = node.children[0]
        first_lex = self._get_lexeme(first_child).lower()

        if isinstance(first_child, Node) and "VarKeywordOpt" in
str(first_child.value):
            if first_child.children and str(first_child.children[0].value) != "EPSILON":
                is_ref = True

                start_idx = 1

        elif first_lex in ["var", "variabel"]:
            is_ref = True
            start_idx = 1

        # 2. Ambil Identifier List
        id_list_node = node.children[start_idx]
        names = self.visit(id_list_node)

        if names is None:
            print(f"[WARN] Identifier list parsing failed for node:
{id_list_node.value}")
            names = []

        # 3. Ambil Type
        type_idx = start_idx + 2
        type_node = None

        if len(node.children) > type_idx:
            type_node = self.visit(node.children[type_idx])

        # Buat ParameterNode
        return [ParameterNode(names=[n], type_node=type_node, is_ref=is_ref) for
n in names]

# ===== STATEMENTS =====
def _convert_CompoundStatement(self, node: Node) -> CompoundNode:
    if len(node.children) > 1:
        stmts = self.visit(node.children[1])
        return CompoundNode(children=stmts if stmts else [])
    return CompoundNode(children=[])

def _convert_StatementList(self, node: Node) -> List[ASTNode]:
    stmts = []
    if not node.children or str(node.children[0].value) == "EPSILON": return
stmts

    first = self.visit(node.children[0])
    if first and not isinstance(first, NoOpNode): stmts.append(first)

    if len(node.children) > 1: self._collect_stmt_prime(node.children[1],
stmts)
    return stmts

def _collect_stmt_prime(self, node: Node, stmts):
    if not node.children or str(node.children[0].value) == "EPSILON": return
stmts
    s = self.visit(node.children[1])
    if s:
        stmts.append(s)
    self._collect_stmt_prime(node.children[1], stmts)

```

```

        if s and not isinstance(s, NoOpNode): stmts.append(s)
        if len(node.children) > 2: self._collect_stmt_prime(node.children[2],
stmts)

    def _convert_Statement(self, node: Node):
        if not node.children or str(node.children[0].value) == "EPSILON": return
NoOpNode()
        return self.visit(node.children[0])

    def _convert_AssignmentStatement(self, node: Node) -> AssignNode:
        """Handle: IDENTIFIER <VariableTail>? := <Expression>"""
        target_name = self._get_lexeme(node.children[0])
        target = VarNode(name=target_name)

        expr = NoOpNode()

        assign_idx = -1
        for i, child in enumerate(node.children):
            lex = self._get_lexeme(child)
            if lex == "=":
                assign_idx = i
                break

        # Jika ketemu :=
        if assign_idx != -1:
            if assign_idx > 1:
                target = self._handle_variable_tail(target, node.children[1])
            if len(node.children) > assign_idx + 1:
                res = self.visit(node.children[assign_idx + 1])
                if res is not None:
                    expr = res
                else:
                    # print(f"[WARN] Expr conversion returned None for
{node.children[assign_idx+1].value}")
                    pass

        return AssignNode(target=target, value=expr)

    # --- PARAMETER LIST ---
    def _convert_ParameterListOpt(self, node: Node) -> List[ASTNode]:
        if not node.children or str(node.children[0].value) == "EPSILON": return
[]
        return self.visit(node.children[0])

    def _convert_ParameterList(self, node: Node) -> List[ASTNode]:
        # <ParameterList> -> <Expression> , <ParameterList> | <Expression>
        # Parse Tree structure:
        # Index 0: Expression
        # Index 2: ParameterList (Recursive) IF comma exists

        params = [self.visit(node.children[0])] # First Expression

        # if len(node.children) > 2:
        #     # Ada koma dan recursive list
        #     rest = self.visit(node.children[2])
        #     if rest: params.extend(rest)

        # return params
        if len(node.children) >= 2:

```

```

        return self.visit(node.children[1])
    return []

def _convert_ProcedureCall(self, node: Node) -> ProcedureCallNode:
    """Handle: IDENTIFIER (<ParameterList>?)"""
    name = self._get_lexeme(node.children[0])
    params = []

    # if len(node.children) >= 3:
    #     child_2 = node.children[2]
    #     val_2 = str(child_2.value) if isinstance(child_2, Node) else
str(child_2)
    #     if "RPAREN" not in val_2 and val_2 != ")":
    #         res = self.visit(child_2)
    #         if res: params = res

    # elif len(node.children) == 2:
    #     res = self.visit(node.children[1])
    #     if res: params = res

    if len(node.children) > 1:
        res = self.visit(node.children[1])
        if res:
            params = res
    return ProcedureCallNode(proc_name=name, arguments=params)

def _convert_ExpressionList(self, node: Node) -> List[ASTNode]:
    """Handle: <Expression> <ExpressionListPrime>"""
    exprs = []

    # children[0] = <Expression>
    first_expr = self.visit(node.children[0])
    if first_expr:
        exprs.append(first_expr)

    # children[1] = <ExpressionListPrime>
    if len(node.children) > 1:
        self._collect_expression_list_prime(node.children[1], exprs)

    return exprs

def _collect_expression_list_prime(self, node: Node, exprs: List[ASTNode]):
    """Handle: COMMA(,) <Expression> <ExpressionListPrime> | <Epsilon>"""
    if not node.children or str(node.children[0].value) == "EPSILON":
        return

    # children[0] = ','
    # children[1] = <Expression>
    # children[2] = <ExpressionListPrime>
    expr = self.visit(node.children[1])
    if expr:
        exprs.append(expr)

    if len(node.children) > 2:
        self._collect_expression_list_prime(node.children[2], exprs)

def _convert_IfStatement(self, node: Node) -> IfNode:
    cond = self.visit(node.children[1])
    true_blk = self.visit(node.children[3])
    else_blk = None

```

```

# Scan for ElsePart or SELAIN-ITU
if len(node.children) > 4:
    for child in node.children[4:]:
        if str(child.value) == "<ElsePart>":
            else_blk = self.visit(child)
            break
        if self._get_lexeme(child).lower() == "selain-itu":
            # Next child is Statement
            else_blk = self.visit(node.children[-1])
            break
    return IfNode(condition=cond, true_block=true_blk, else_block=else_blk)

def _convert_ElsePart(self, node: Node):
    if not node.children or str(node.children[0].value) == "EPSILON": return
None
    return self.visit(node.children[1])

def _convert_WhileStatement(self, node: Node) -> WhileNode:
    cond = self.visit(node.children[1])
    body = self.visit(node.children[3])
    return WhileNode(condition=cond, body=body)

def _convert_ForStatement(self, node: Node) -> ForNode:
    var = self._get_lexeme(node.children[1])
    start = self.visit(node.children[3])
    direct = self._get_lexeme(node.children[4])
    if not isinstance(direct, str): direct =
self._get_lexeme(node.children[4].children[0])
    end = self.visit(node.children[5])
    body = self.visit(node.children[7])
    return ForNode(variable=var, start_expr=start, direction=direct,
end_expr=end, body=body)

# ===== EXPRESSIONS =====
def _convert_Expression(self, node: Node):
    left = self.visit(node.children[0])
    if len(node.children) > 1: return
self._visit_expr_prime(node.children[1], left)
    return left

def _visit_expr_prime(self, node:Node, left):
    if not node.children or str(node.children[0].value) == "EPSILON": return
left
    op = self._get_operator_lexeme(node.children[0])
    right = self.visit(node.children[1])
    return BinOpNode(op=op, left=left, right=right)

def _convert_SimpleExpression(self, node: Node):
    left = self.visit(node.children[0])
    if len(node.children) > 1: return
self._visit_simple_prime(node.children[1], left)
    return left

def _visit_simple_prime(self, node:Node, left):
    if not node.children or str(node.children[0].value) == "EPSILON": return
left
    op = self._get_operator_lexeme(node.children[0])
    right = self.visit(node.children[1])
    new_left = BinOpNode(op=op, left=left, right=right)

```

```

        if len(node.children) > 2: return
self._visit_simple_prime(node.children[2], new_left)
return new_left

def _convert_SignedTerm(self, node: Node):
    sign_node = node.children[0]
    unary = None
    if sign_node.children and str(sign_node.children[0].value) != "EPSILON":
        unary = self._get_lexeme(sign_node.children[0].children[0])
    term = self.visit(node.children[1])
    if unary == "-": return UnaryOpNode(op="-", expr=term)
    return term

def _convert_Term(self, node: Node):
    left = self.visit(node.children[0])
    if len(node.children) > 1: return
self._visit_term_prime(node.children[1], left)
return left

def _visit_term_prime(self, node: Node, left):
    if not node.children or str(node.children[0].value) == "EPSILON": return
left
    op = self._get_operator_lexeme(node.children[0])
    right = self.visit(node.children[1])
    new_left = BinOpNode(op=op, left=left, right=right)
    if len(node.children) > 2: return
self._visit_term_prime(node.children[2], new_left)
return new_left

def _convert_Factor(self, node: Node):
    first = node.children[0]
    val_str = str(first.value)

    if val_str == "<Variable>": return self._convert_Variable(first)
    if val_str == "<Constant>": return self._convert_Constant(first)

    # Parenthesis ( Expr )
    if self._get_lexeme(first) == "(":
        return self.visit(node.children[1])

    token = self._get_token(first)
    if token:
        lex = token.lexeme
        if token.token_type == "LOGICAL_OPERATOR" and lex.lower() ==
"tidak":
            return UnaryOpNode(op="tidak",
expr=self.visit(node.children[1]))
        if token.token_type == "NUMBER":
            try: val = float(lex) if '.' in lex else int(lex)
            except: val = 0
            return NumNode(value=val)
        if token.token_type == "IDENTIFIER":
            if len(node.children) > 1 and
self._get_lexeme(node.children[1]) == "(":
                params = []
                if len(node.children) > 2:
                    res = self.visit(node.children[2])
                    if res: params = res
                return ProcedureCallNode(proc_name=lex, arguments=params)
            return VarNode(name=lex)

```

```

        return NoOpNode()

    def _convert_Variable(self, node: Node):
        name = self._get_lexeme(node.children[0])
        base = VarNode(name=name)
        if len(node.children) > 1: return self._handle_variable_tail(base,
node.children[1])
        return base

    def _handle_variable_tail(self, base, node:Node):
        if not node.children or str(node.children[0].value) == "EPSILON": return
base
        first = self._get_lexeme(node.children[0])
        if first == "[":
            idx = self.visit(node.children[1])
            new_base = ArrayAccessNode(array=base, index=idx)
            if len(node.children) > 3: return
        self._handle_variable_tail(new_base, node.children[3])
        return new_base
        if first == ".":
            field = self._get_lexeme(node.children[1])
            new_base = FieldAccessNode(record=base, field_name=field)
            if len(node.children) > 2: return
        self._handle_variable_tail(new_base, node.children[2])
        return new_base
        return base

    def _convert_Constant(self, node: Node):
        first = node.children[0]
        if str(first.value) == "<SignOpt>":
            sign = None
            if first.children and str(first.children[0].value) != "EPSILON":
                sign = self._get_lexeme(first.children[0].children[0])
            val = self.visit(node.children[1])
            if sign == "-" and isinstance(val, NumNode): val.value = -val.value
            return val
        token = self._get_token(first)
        if token:
            if token.token_type == "STRING_LITERAL": return
StringNode(value=token.lexeme)
            if token.token_type == "CHAR_LITERAL": return
CharNode(value=token.lexeme)
            if token.lexeme.lower() == "true": return BoolNode(value=True)
            if token.lexeme.lower() == "false": return BoolNode(value=False)
        return NoOpNode()

    def _convert_UnsignedConstant(self, node: Node):
        token = self._get_token(node.children[0])
        if token:
            if token.token_type == "NUMBER":
                try: val = float(token.lexeme) if '.' in token.lexeme else
int(token.lexeme)
                except: val = 0
                return NumNode(value=val)
            if token.token_type == "IDENTIFIER": return
VarNode(name=token.lexeme)
        return NoOpNode()

```

## 3. Ast\_analyzer.py : Analyzer

```

class ASTAnalyzer:
    def __init__(self):
        self.symbol_table = SymbolTable()

    def visit(self, node: ASTNode):
        """Dispatcher utama untuk mengunjungi node AST."""
        if node is None:
            return None

        # Panggil method visit_NamaNode
        method_name = 'visit_' + node.__class__.__name__
        visitor = getattr(self, method_name, self.generic_visit)
        return visitor(node)

    def generic_visit(self, node):
        """Fallback untuk node yang belum ada handler spesifiknya."""
        return None

# =====
# PROGRAM & BLOCKS
# =====

def visit_ProgramNode(self, node: ProgramNode):
    # 1. Masukkan nama program ke scope global
    self.symbol_table.enter(
        name=node.name,
        obj=ObjectKind.PROGRAM,
        type_kind=TypeKind.NOTYPE,
        ref=0, nrm=1, lev=0, adr=0
    )

    # 2. Visit Declarations
    for decl in node.declarations:
        self.visit(decl)

    # 3. Visit Main Block
    self.visit(node.block)

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

def visit_VarDeclNode(self, node: VarDeclNode):
    # Cek apakah tipe-nya adalah ArrayTypeNode (definisi array langsung)
    if isinstance(node.type_node, ArrayTypeNode):
        # Resolve struktur array dan dapatkan referensi ke atab
        type_kind, ref_idx = self._resolve_array_type(node.type_node)
    else:
        type_name = node.type_node.type_name if hasattr(node.type_node,
'type_name') else str(node.type_node)
        type_kind = self._resolve_type_str(type_name)
        ref_idx = 0
    if type_kind == TypeKind.NOTYPE:
        type_entry_idx = self.symbol_table.lookup(type_name)
        if type_entry_idx != 0:
            entry = self.symbol_table.get_entry(type_entry_idx)
            if entry.obj == ObjectKind.TYPE:
                type_kind = entry.type
                ref_idx = entry.ref

```

```

        try:
            idx = self.symbol_table.add_variable(node.var_name, type_kind,
ref=ref_idx)

            if idx is None:
                # print(f"[Semantic Error] Failed to add variable
'{node.var_name}'")
                # return
                raise ASTAnalyzerError(message=f"Failed to add variable
'{node.var_name}'")

            entry = self.symbol_table.get_entry(idx)
            node.type = entry.type.name
            node.symbol_entry = {'tab_index': idx, 'lev': entry.lev}

        except ASTAnalyzerError as e:
            raise ASTAnalyzerError(message=e)

        except ValueError as e:
            # print(f"[Semantic Error] {e}")
            raise ASTAnalyzerError(message=e)

    def _resolve_array_type(self, node: ArrayTypeNode):
        """
        Memproses ArrayTypeNode, mendaftarkan ke atab, dan mengembalikan
(TypeKind.ARRAY, ref_index)
        """

        # Asumsi bounds harus constant expression
        low_val = self._get_constant_value(node.lower)
        high_val = self._get_constant_value(node.upper)

        if low_val is None or high_val is None:
            print("[Semantic Error] Array bounds must be constant integers.")
            return TypeKind.NOTYPE, 0

        if low_val > high_val:
            # print(f"[Semantic Error] Array lower bound ({low_val}) > upper
bound ({high_val})")
            raise ASTAnalyzerError(message=f"Array lower bound ({low_val}) >
upper bound ({high_val})")

        if isinstance(node.element_type, ArrayTypeNode):
            etyp, eref = self._resolve_array_type(node.element_type)
        else:
            # Simple Type / Named Type
            type_name = getattr(node.element_type, 'type_name',
str(node.element_type))
            etyp = self._resolve_type_str(type_name)
            eref = 0

        # Handle Named Type untuk element
        if etyp == TypeKind.NOTYPE:
            type_entry_idx = self.symbol_table.lookup(type_name)
            if type_entry_idx != 0:
                entry = self.symbol_table.get_entry(type_entry_idx)
                if entry.obj == ObjectKind.TYPE:
                    etyp = entry.type
                    eref = entry.ref

# 3. Register to ATAB

```

```

        xtype = TypeKind.INTEGER

        atab_idx = self.symbol_table.add_array_type(xtyp, etyp, eref, low_val,
high_val)

        return TypeKind.ARRAY, atab_idx

    def _get_constant_value(self, node: ASTNode) -> Optional[int]:
        """Helper untuk mengevaluasi nilai konstan statis dari AST Node"""
        if isinstance(node, NumNode):
            return int(node.value)

        if isinstance(node, UnaryOpNode):
            val = self._get_constant_value(node.expr)
            if val is not None:
                if node.op == '-': return -val
                if node.op == '+': return val

        if isinstance(node, VarNode):
            # Lookup const identifier
            idx = self.symbol_table.lookup(node.name)
            if idx != 0:
                entry = self.symbol_table.get_entry(idx)
                if entry.obj == ObjectKind.CONSTANT:
                    return int(entry.adr) # Nilai konstanta disimpan di adr

        return None

    def visit_ConstDeclNode(self, node: ConstDeclNode):
        # 1. Evaluasi Tipe Nilai
        value_type = self.visit(node.value) # Mengembalikan TypeKind

        # 2. Ambil Nilai Raw (untuk disimpan di adr)
        raw_value = 0
        if hasattr(node.value, 'value'):
            raw_value = node.value.value

        try:
            idx = self.symbol_table.add_constant(node.const_name, value_type,
raw_value)

            # SAFETY CHECK
            if idx is None:
                # print(f"[Semantic Error] Failed to add constant
'{node.const_name}'")
                # return
                raise ASTAnalyzerError(message=f"Failed to add constant
'{node.const_name}'")

            # Dekorasi node (opsional)
            entry = self.symbol_table.get_entry(idx)
            if entry:
                node.type = entry.type.name
                node.symbol_entry = {'tab_index': idx, 'lev': entry.lev}

        except ASTAnalyzerError as e:
            raise ASTAnalyzerError(message=e)

        except ValueError as e:
            # print(f"[Semantic Error] {e}")

```

```

        raise ASTAnalyzerError(message=e)

def visit_TypeDeclNode(self, node: TypeDeclNode):
    # Untuk Milestone 3 dasar, kita catat namanya saja
    # Pengembangan lanjut: simpan struktur array/record di atab/btab
    type_kind = TypeKind.NOTYPE
    ref_idx = 0

    # 1. Cek apakah value-nya adalah Definisi Array
    if isinstance(node.value, ArrayTypeNode):
        type_kind, ref_idx = self._resolve_array_type(node.value)

    # 2. Cek apakah value-nya adalah Tipe Lain (Alias, misal: TYPE Angka = Integer)
    elif isinstance(node.value, TypeNode):
        type_kind = self._resolve_type_str(node.value.type_name)

    try:
        # Masukkan ke Symbol Table sebagai ObjectKind.TYPE
        self.symbol_table.enter(
            node.type_name,
            ObjectKind.TYPE,
            type_kind,
            ref_idx, # Simpan referensi ke atab
            1,
            self.symbol_table.current_level,
            0
        )
    except ASTAnalyzerError as e:
        raise ASTAnalyzerError(message=e)

    except Exception as e:
        # print(f"[Semantic Error] {e}")
        raise ASTAnalyzerError(message=e)

# =====
# SUBPROGRAMS (Procedure & Function)
# =====

def visit_ProcedureDeclNode(self, node: ProcedureDeclNode):
    proc_name = node.name

    # 1. Daftarkan Prosedur di Scope Parent
    self.symbol_table.enter(
        proc_name,
        ObjectKind.PROCEDURE,
        TypeKind.NOTYPE,
        0, 1, self.symbol_table.current_level, 0
    )

    # 2. Masuk Scope Baru
    self.symbol_table.enter_scope(proc_name)

    # 3. Proses Parameter
    for param in node.params:
        self.visit_ParameterNode(param)

    current_btab =
self.symbol_table.btab[self.symbol_table.display[self.symbol_table.current_level

```

```

]]>
    current_btab.lpar = self.symbol_table.tx

    if hasattr(node, 'local_vars'):
        for decl in node.local_vars:
            self.visit(decl)
    # 4. Proses Block (yang sudah include deklarasi lokal & statements)
    if node.block:
        self.visit(node.block)

    # 5. Keluar Scope
    self.symbol_table.exit_scope()

def visit_FunctionDeclNode(self, node: FunctionDeclNode):
    func_name = node.name
    return_type = self._resolve_type_str(node.return_type.type_name)

    # 1. Daftarkan Fungsi di Scope Parent (dengan tipe kembalian)
    self.symbol_table.enter(
        func_name,
        ObjectKind.FUNCTION,
        return_type,
        0, 1, self.symbol_table.current_level, 0
    )

    # 2. Masuk Scope Baru
    self.symbol_table.enter_scope(func_name)

    # 3. Parameter
    for param in node.params:
        self.visit_ParameterNode(param)

    if hasattr(node, 'local_vars'):
        for decl in node.local_vars:
            self.visit(decl)

    # 4. Proses Block (yang sudah include deklarasi & body)
    if node.block:
        self.visit(node.block)

    # 5. Keluar Scope
    self.symbol_table.exit_scope()

def visit_ParameterNode(self, node: ParameterNode):
    type_kind = self._resolve_type_str(node.type_node.type_name)
    for name in node.names:
        try:
            # Masukkan parameter sebagai variabel lokal
            nrm_val = 0 if node.is_ref else 1
            self.symbol_table.enter(
                name, ObjectKind.VARIABLE, type_kind, 0, nrm_val,
                self.symbol_table.current_level, 0
            )
        except ASTAnalyzerError as e:
            raise ASTAnalyzerError(message=e)
        except ValueError as e:
            # print(f"[Semantic Error] {e}")
            raise ASTAnalyzerError(message=e)

# =====

```

```

# STATEMENTS
# =====

def visit_CompoundNode(self, node: CompoundNode):
    for child in node.children:
        self.visit(child)

def visit_AssignNode(self, node: AssignNode):
    # 1. Cek Tipe Target (Variable)
    target_type = self.visit(node.target)

    # 2. Cek Tipe Value (Expression)
    value_type = self.visit(node.value)
    if target_type is None: target_type = TypeKind.NOTYPE
    if value_type is None: value_type = TypeKind.NOTYPE
    # 3. Validasi Kompatibilitas
    if target_type != value_type and target_type != TypeKind.NOTYPE and value_type != TypeKind.NOTYPE:
        if target_type == TypeKind.REAL and value_type == TypeKind.INTEGER:
            return

        # print(f"[Semantic Error] Type mismatch in assignment. "
        #       f"Cannot assign {value_type.name} to {target_type.name}")
        raise ASTAnalyzerError(message=f"Type mismatch in assignment. Cannot assign {value_type.name} to {target_type.name}")

def visit_IfNode(self, node: IfNode):
    cond_type = self.visit(node.condition)
    if cond_type != TypeKind.BOOLEAN and cond_type != TypeKind.NOTYPE:
        # print(f"[Semantic Error] IF condition must be BOOLEAN, got {cond_type.name}")
        raise ASTAnalyzerError(message=f"IF condition must be BOOLEAN, got {cond_type.name}")

    self.visit(node.true_block)
    if node.else_block:
        self.visit(node.else_block)

def visit_WhileNode(self, node: WhileNode):
    cond_type = self.visit(node.condition)
    if cond_type != TypeKind.BOOLEAN and cond_type != TypeKind.NOTYPE:
        # print(f"[Semantic Error] WHILE condition must be BOOLEAN, got {cond_type.name}")
        raise ASTAnalyzerError(message=f"WHILE condition must be BOOLEAN, got {cond_type.name}")
    self.visit(node.body)

def visit_ForNode(self, node: ForNode):
    # Cek variabel loop
    var_idx = self.symbol_table.lookup(node.variable)
    if var_idx == 0:
        # print(f"[Semantic Error] Loop variable '{node.variable}' not declared.")
        raise ASTAnalyzerError(message=f"Loop variable '{node.variable}' not declared.")

    start_type = self.visit(node.start_expr)
    end_type = self.visit(node.end_expr)

    if start_type != TypeKind.INTEGER or end_type != TypeKind.INTEGER:

```

```

        # print("[Semantic Error] FOR loop limits must be INTEGER.")
        raise ASTAnalyzerError(message=f"FOR loop limits must be INTEGER.")

    self.visit(node.body)

def visit_ProcedureCallNode(self, node: ProcedureCallNode):
    # Handle Built-in functions
    if node.proc_name in ['writeln', 'write', 'readln', 'read']:
        for arg in node.arguments:
            self.visit(arg)
        return

    # Lookup Prosedur/Fungsi
    idx = self.symbol_table.lookup(node.proc_name)
    if idx == 0:
        # print(f"[Semantic Error] Identifier '{node.proc_name}' not
declared.")
        # return TypeKind.NOTYPE
        raise ASTAnalyzerError(message=f"Identifier '{node.proc_name}' not
declared.")

    entry = self.symbol_table.get_entry(idx)

    # Validasi: Harus Prosedur atau Fungsi
    if entry.obj == ObjectKind.FUNCTION:
        return entry.type
    return TypeKind.NOTYPE

# =====
# EXPRESSIONS & FACTORS
# =====

def visit_ArrayAccessNode(self, node: ArrayAccessNode) -> TypeKind:
    # 1. Periksa Variabel Array
    array_type = self.visit(node.array)

    if array_type != TypeKind.ARRAY:
        # print(f"[Semantic Error] Variable is not an array.")
        # return TypeKind.NOTYPE
        raise ASTAnalyzerError(message=f"Variable is not an array.")

    # 2. Periksa Index
    index_type = self.visit(node.index)
    if index_type != TypeKind.INTEGER:
        # print(f"[Semantic Error] Array index must be INTEGER, got
{index_type.name}")
        raise ASTAnalyzerError(message=f"Array index must be INTEGER, got
{index_type.name}")

    # 3. Ambil Tipe Elemen dari Symbol Table (atab)
    if not hasattr(node.array, 'symbol_entry'):
        return TypeKind.NOTYPE

    # Ambil referensi ke atab
    tab_idx = node.array.symbol_entry.get('tab_index')
    entry = self.symbol_table.get_entry(tab_idx)

    if entry and entry.ref > 0:
        atab_entry = self.symbol_table.atab[entry.ref]

```

```

# Validasi Range Index
if isinstance(node.index, NumNode):
    val = int(node.index.value)
    if val < atab_entry.low or val > atab_entry.high:
        # print(f"[Semantic Error] Array index out of bounds: {val}.")
Valid: [{atab_entry.low}..{atab_entry.high}]")
        raise ASTAnalyzerError(message=f"Array index out of bounds: {val}. Valid: [{atab_entry.low}..{atab_entry.high}]")

# Kembalikan tipe elemen (etyp)
return atab_entry.etyp

return TypeKind.NOTYPE


def visit_BinOpNode(self, node: BinOpNode) -> TypeKind:
    left = self.visit(node.left)
    right = self.visit(node.right)

    if left == TypeKind.NOTYPE or right == TypeKind.NOTYPE:
        return TypeKind.NOTYPE

    op = node.op.lower()

    # Aritmatika: +, -, *, div, mod
    if op in ['+', '-', '*', 'div', 'mod', 'bagi']:
        # Integer operan
        if left == TypeKind.INTEGER and right == TypeKind.INTEGER:
            if op == '/' or op == 'bagi': return TypeKind.REAL
            return TypeKind.INTEGER

        # Real operan
        if left == TypeKind.REAL or right == TypeKind.REAL:
            if op == 'div' or op == 'mod':
                # print(f"[Semantic Error] Operator '{op}' only for
INTEGER.")
                # return TypeKind.NOTYPE
                raise ASTAnalyzerError(message=f"Operator '{op}' only for
INTEGER.")
            return TypeKind.REAL

    # Relasional: =, <>, <, >, <=, >=
    if op in ['=', '<>', '<', '>', '<=', '>=']:
        return TypeKind.BOOLEAN

    # Logika: and, or
    if op in ['and', 'or', 'dan', 'atau']:
        if left == TypeKind.BOOLEAN and right == TypeKind.BOOLEAN:
            return TypeKind.BOOLEAN
        else:
            # print(f"[Semantic Error] Operator '{op}' requires BOOLEAN
operands.")
            raise ASTAnalyzerError(message=f"Operator '{op}' requires
BOOLEAN operands.")

    return TypeKind.NOTYPE


def visit_UnaryOpNode(self, node: UnaryOpNode) -> TypeKind:
    expr_type = self.visit(node.expr)
    op = node.op.lower()

```

```

        if op == 'tidak' or op == 'not':
            if expr_type == TypeKind.BOOLEAN: return TypeKind.BOOLEAN
        elif op == '-':
            if expr_type in [TypeKind.INTEGER, TypeKind.REAL]: return expr_type

        # print(f"[Semantic Error] Invalid unary op '{op}' on {expr_type.name}")
        raise ASTAnalyzerError(message=f"Invalid unary op '{op}' on
{expr_type.name}")
        # return TypeKind.NOTYPE

    def visit_VarNode(self, node: VarNode) -> TypeKind:
        # 1. Lookup Identifier
        idx = self.symbol_table.lookup(node.name)

        if idx == 0:
            # print(f"[Semantic Error] Variable '{node.name}' not declared.")
            # return TypeKind.NOTYPE
            raise ASTAnalyzerError(message=f"Variable '{node.name}' not
declared.")

        # 2. Ambil Entry
        entry = self.symbol_table.get_entry(idx)

        node.type = entry.type.name
        # Simpan tab_index beserta info entry lainnya
        node.symbol_entry = {
            'tab_index': idx,
            'lev': entry.lev,
            'adr': entry.adr,
            'ref': entry.ref,
            'obj': entry.obj,
            'type': entry.type
        }

        return entry.type

    def visit_NumNode(self, node: NumNode) -> TypeKind:
        if isinstance(node.value, float): return TypeKind.REAL
        return TypeKind.INTEGER

    def visit_StringNode(self, node: StringNode) -> TypeKind:
        return TypeKind.STRING

    def visit_BoolNode(self, node: BoolNode) -> TypeKind:
        return TypeKind.BOOLEAN

    def visit_CharNode(self, node: CharNode) -> TypeKind:
        return TypeKind.CHAR

    def visit_NoOpNode(self, node: NoOpNode):
        return TypeKind.NOTYPE

    # --- HELPERS ---
    def _resolve_type_str(self, type_name: str) -> TypeKind:
        tn = type_name.upper()
        if tn == 'INTEGER': return TypeKind.INTEGER
        if tn == 'REAL': return TypeKind.REAL
        if tn == 'BOOLEAN': return TypeKind.BOOLEAN
        if tn == 'CHAR': return TypeKind.CHAR

```

```

if tn == 'STRING': return TypeKind.STRING
# Array/Record
if tn == 'ARRAY' or 'ARRAY' in tn: return TypeKind.ARRAY
return TypeKind.NOTYPE

```

## 4. semantic.py: Semantic Analyzer

```

class SemanticAnalyzer:
    converter: ASTConverter
    analyzer: ASTDecorator

    def __init__(self):
        self.converter = ASTConverter()
        self.analyzer = ASTDecorator()

    def analyze(self, parse_tree:Node, debug:bool=False) -> Tuple[ASTNode,
SymbolTable, ASTNode]:
        try:
            # Jalankan AST Converter
            converter = ASTConverter()
            ast = converter.convert(parse_tree)
            if debug :
                print("\n[DEBUG] Abstract Syntax Tree (AST)")
                ast_printer = ASTPrinter()
                print(ast_printer.print(ast))

            # Jalankan Analyzer
            analyzer = ASTDecorator()
            decorated_ast = analyzer.generate_decorated_ast(ast)

            return decorated_ast, analyzer.symbol_table, ast

        except ASTAnalyzerError as e:
            raise SemanticError(message=e)

```

## 5. semantic.py: Semantic Error

```

class SemanticError(Exception):
    """Custom exception untuk error semantik."""
    def __init__(self, message:str, line:int=None, column:int=None) -> None:
        if line and column:
            super().__init__(f"Semantic Error on line {line}, column {column}:
{message}")
        else:
            super().__init__(f"Semantic Error: {message}")
        self.message = message
        self.line = line
        self.column = column

```

## 6. Print\_tree.py : ASTPrinter

```

class ASTPrinter:
    def print(self, node):
        """Entry point untuk mencetak AST"""
        return self._print_node(node, "", True)

    def _print_node(self, node, prefix, is_last):
        # 1. Dapatkan Label Node (Text Header)

```

```

label = self._get_label(node)
annot = self._get_annotation(node)

connector = "\\\\"-- " if is_last else "+-- "
if prefix == "": result = f"{label}{annot}\n"
else: result = f"{prefix}{connector}{label}{annot}\n"

# 2. Dapatkan Children (Virtual atau Real)
children_map = self._get_children(node)

# 3. Print Children
child_prefix = prefix + ("      " if is_last else "|    ")
count = len(children_map)

for i, (tag, child) in enumerate(children_map):
    is_last_child = (i == count - 1)

    if tag: # Virtual Node (Declarations / Block)
        tag_conn = "\\\\"-- " if is_last_child else "+-- "
        result += f"{child_prefix}{tag_conn}{tag}\n"
        virtual_prefix = child_prefix + ("      " if is_last_child else "|")
    )

    # Logic Unwrap untuk Block -> CompoundNode
    items = []
    if (tag in ["Block", "Body"]) and isinstance(child,
CompoundNode):
        items = child.children
    elif isinstance(child, list):
        items = child
    elif isinstance(child, ASTNode):
        items = [child]

        for k, item in enumerate(items):
            is_last_item = (k == len(items) - 1)
            # Rekursi
            result += self._print_node(item, virtual_prefix,
is_last_item)
        else:
            # Direct Child
            result += self._print_node(child, child_prefix, is_last_child)

    return result

# --- HELPERS ---

def _get_annotation(self, node) -> str:
    parts = []
    # Cek keberadaan atribut sebelum akses
    if hasattr(node, 'symbol_entry') and node.symbol_entry and 'tab_index' in node.symbol_entry:
        parts.append(f"idx:{node.symbol_entry['tab_index']}\"")
    if hasattr(node, 'type') and node.type:
        parts.append(f"type:{node.type.lower()}\"")
    return " \t→ " + ", ".join(parts) if parts else ""

def _get_label(self, node) -> str:
    """Menentukan teks yang muncul di node"""
    if isinstance(node, ProgramNode): return f"Program(' {node.name} ')"

```

```

        if isinstance(node, VarDeclNode):
            t_name = getattr(node.type_node, 'type_name', 'unknown')
            return f"VarDecl(name: '{node.var_name}', type: '{t_name}')"

        if isinstance(node, ConstDeclNode): return
        f"ConstDecl('{node.const_name}')"
        if isinstance(node, TypeDeclNode): return
        f"TypeDecl('{node.type_name}')"
        if isinstance(node, TypeNode): return f"Type('{node.type_name}')"

        if isinstance(node, ProcedureCallNode):
            args = [self._compact(a) for a in node.arguments]
            return f"ProcedureCall(name: '{node.proc_name}', args: [{',
                '.join(filter(None, args))}]"

        if isinstance(node, AssignNode):
            target_str = self._compact(node.target) or "Target"
            value_str = self._compact(node.value)
            # Jika value kompleks (None), jangan tampilkan di label
            if value_str:
                return f"Assign(target: {target_str}, value: {value_str})"
            else:
                return f"Assign(target: {target_str}, value:"

        if isinstance(node, BinOpNode): return f"BinOp('{node.op}')"
        if isinstance(node, UnaryOpNode): return f"UnaryOp('{node.op}')"
        if isinstance(node, VarNode): return f"Var('{node.name}')"
        if isinstance(node, NumNode): return f"Num({node.value})"
        if isinstance(node, StringNode): return f"String('{node.value}')"

        if isinstance(node, ParameterNode):
            prefix = "VAR " if node.is_ref else ""
            t = getattr(node.type_node, 'type_name', '?')
            return f"Param({prefix}{','.join(node.names)}: {t})"

        if isinstance(node, ProcedureDeclNode): return
        f"ProcedureDecl('{node.name}')"
        if isinstance(node, FunctionDeclNode): return
        f"FunctionDecl('{node.name}')"
        if isinstance(node, ForNode): return f"For('{node.variable}')"

        return node.__class__.__name__.replace("Node", "")

def _compact(self, node):
    """Return compact string untuk simple nodes, None untuk complex nodes"""
    if node is None: return None
    if isinstance(node, VarNode): return f"Var('{node.name}')"
    if isinstance(node, NumNode): return f"Num({node.value})"
    if isinstance(node, StringNode): return f"String('{node.value}')"
    if isinstance(node, CharNode): return f"Char('{node.value}')"
    if isinstance(node, BoolNode): return f"Bool({node.value})"
    # Complex nodes return None agar ditampilkan sebagai child
    return None

def _get_children(self, node):
    """Menentukan anak mana yang akan dicetak dan grouping-nya"""
    children = []

    if isinstance(node, ProgramNode):
        if node.declarations: children.append(("Declarations",

```

```

node.declarations))
    children.append(("Block", node.block))
    return children

if isinstance(node, (ProcedureDeclNode, FunctionDeclNode)):
    params = getattr(node, 'params', [])
    decls = getattr(node, 'declarations', [])
    body = getattr(node, 'body', None) or getattr(node, 'block', None)

    if params: children.append(("Params", params))
    if decls: children.append(("Declarations", decls))
    if body: children.append(("Body", body))
    return children

if isinstance(node, AssignNode):
    # Jika value sudah di-compact di header, jangan tampilkan children
    if self._compact(node.value):
        return []
    # Jika value kompleks, tampilkan hanya value (target sudah di
header)
    children.append((None, node.value))
    return children

elif isinstance(node, BinOpNode):
    children.append((None, node.left))
    children.append((None, node.right))
    return children

elif isinstance(node, UnaryOpNode):
    children.append((None, node.expr))
    return children

elif isinstance(node, IfNode):
    children.append((None, node.condition))
    children.append((None, node.true_block))
    if node.else_block: children.append((None, node.else_block))
    return children

elif isinstance(node, WhileNode):
    children.append((None, node.condition))
    children.append((None, node.body))
    return children

elif isinstance(node, ForNode):
    children.append((None, node.start_expr))
    children.append((None, node.end_expr))
    children.append((None, node.body))
    return children

elif isinstance(node, CompoundNode):
    for c in node.children:
        children.append((None, c))
    return children

elif isinstance(node, ArrayAccessNode):
    children.append((None, node.array))
    children.append((None, node.index))
    return children

elif isinstance(node, ProcedureCallNode):

```

```

        # Jika semua args sudah di-compact di header, jangan tampilkan
children
        all_compact = all(self._compact(arg) for arg in node.arguments)
        if all_compact:
            return []
        # Jika ada arg kompleks, tampilkan semuanya
        for arg in node.arguments:
            children.append((None, arg))
        return children

    return children

```

## 7. Ast\_decorator : AST Decorated

```

class ASTDecorator(ASTAnalyzer):

    def __init__(self):
        super().__init__()
        # Gunakan tabel yang sudah dipatch
        self.symbol_table = PatchedSymbolTable()

    def _set_type(self, node: ASTNode, type_kind: TypeKind):
        if type_kind and type_kind != TypeKind.NOTYPE:
            node.type = type_kind.name
        else:
            node.type = "VOID"

    def _set_void(self, node: ASTNode):
        node.type = "VOID"

    # =====#
    # DECLARATIONS & PARAMETERS (NEW: Handle ParameterNode)
    # =====#

    def visit_VarDeclNode(self, node: VarDeclNode):
        super().visit_VarDeclNode(node)
        idx = self.symbol_table.lookup_local(node.var_name)
        if idx > 0:
            entry = self.symbol_table.get_entry(idx)
            node.type = entry.type.name
            node.symbol_entry = {
                'tab_index': idx, 'lev': entry.lev,
                'adr': entry.adr, 'ref': entry.ref
            }

    def visit_ParameterNode(self, node: ParameterNode):
        # 1. Jalankan logic asli (insert ke tabel simbol)
        super().visit_ParameterNode(node)

        # 2. Decorate
        # ASTConverter membuat ParameterNode per satu nama, jadi kita ambil
        names[0]
        if node.names:
            name = node.names[0]
            # Lookup local karena parameter ada di scope prosedur itu sendiri
            idx = self.symbol_table.lookup_local(name)
            if idx > 0:
                entry = self.symbol_table.get_entry(idx)
                node.type = entry.type.name

```

```

        node.symbol_entry = {
            'tab_index': idx,
            'lev': entry.lev,
            'adr': entry.adr,
            'ref': entry.ref
        }

# =====
# PROGRAM & BLOCKS
# =====

def visit_ProgramNode(self, node: ProgramNode):
    super().visit_ProgramNode(node)
    node.type = "PROGRAM"
    node.symbol_entry = {'lev': 0}

def visit_CompoundNode(self, node: CompoundNode):
    current_lev = self.symbol_table.current_level
    current_block_idx = self.symbol_table.bx
    node.type = "BLOCK"
    node.symbol_entry = {'block_index': current_block_idx, 'lev': current_lev}
    super().visit_CompoundNode(node)

# =====
# VARIABLES & OPERATIONS
# =====

def visit_VarNode(self, node: VarNode) -> TypeKind:
    result_type = super().visit_VarNode(node)
    idx = self.symbol_table.lookup(node.name)
    if idx > 0:
        entry = self.symbol_table.get_entry(idx)
        node.symbol_entry = {
            'tab_index': idx, 'lev': entry.lev,
            'adr': entry.adr, 'ref': entry.ref
        }
    return result_type

def visit_BinOpNode(self, node: BinOpNode) -> TypeKind:
    result_type = super().visit_BinOpNode(node)
    self._set_type(node, result_type)
    return result_type

def visit_UnaryOpNode(self, node: UnaryOpNode) -> TypeKind:
    result_type = super().visit_UnaryOpNode(node)
    self._set_type(node, result_type)
    return result_type

def visit_NumNode(self, node: NumNode) -> TypeKind:
    result_type = super().visit_NumNode(node)
    self._set_type(node, result_type)
    return result_type

# =====
# STATEMENTS (Updated: ProcedureCall visit arguments)
# =====

def visit_AssignNode(self, node: AssignNode):
    super().visit_AssignNode(node)

```

```

        self._set_void(node)

    def visit_IfNode(self, node: IfNode):
        super().visit_IfNode(node)
        node.type = "STATEMENT"

    def visit_ForNode(self, node: ForNode):
        super().visit_ForNode(node)
        node.type = "STATEMENT"

    def visit_WhileNode(self, node: WhileNode):
        super().visit_WhileNode(node)
        node.type = "STATEMENT"

    def visit_ProcedureCallNode(self, node: ProcedureCallNode):
        # Jalankan logic asli (lookup nama prosedur)
        super().visit_ProcedureCallNode(node)

        if node.proc_name in ['writeln', 'write', 'readln', 'read']:
            node.type = "PREDEFINED"
        else:
            # Handle User Defined Procedure
            idx = self.symbol_table.lookup(node.proc_name)
            if idx > 0:
                entry = self.symbol_table.get_entry(idx)
                node.symbol_entry = {'tab_index': idx, 'lev': entry.lev}
                if entry.obj == ObjectKind.FUNCTION:
                    node.type = entry.type.name
                else:
                    self._set_void(node)

        # [FIX] Force visit arguments agar VarNode di dalamnya ter-dekorasi
        # Analyzer asli kadang skip visit argumen user-defined procedure
        for arg in node.arguments:
            self.visit(arg)

    def generate_decorated_ast(self, root_node: ASTNode) -> ASTNode:
        self.visit(root_node)
        return root_node

```

## 8. compiler.py: Main Program

```

def main():
    """
    Driver utama untuk parser.
    Mengambil 1 argumen: path ke file source code .pas
    """

    # --- 1. Validasi Argumen Input ---
    if len(sys.argv) != 2:
        print("Usage: python syntax.py <source_file_path.pas>", file=sys.stderr)
        sys.exit(1)

    source_file_path = sys.argv[1]
    if not source_file_path.lower().endswith('.pas'):
        print(f"Input Error: Source file harus berekstensi .pas. Diberikan: '{source_file_path}'", file=sys.stderr)
        sys.exit(1)

```

```
# Mendapatkan path absolut ke dfa.json
BASE_DIR = os.path.dirname(os.path.abspath(__file__)) # src/syntax/
DFA_FILE_PATH = os.path.join(BASE_DIR, 'lexical', 'dfa.json')

# --- 2. Baca Source Code ---
try:
    with open(source_file_path, 'r') as f:
        source_code = f.read()
except FileNotFoundError:
    print(f"Error: Input file tidak ditemukan di '{source_file_path}'",
file=sys.stderr)
    sys.exit(1)

# --- 3. Jalankan Lexer ---
lexer = Lexer(DFA_FILE_PATH)
tokens = []
try:
    tokens = lexer.tokenize(source_code)
except LexicalError as e:
    print(str(e), file=sys.stderr)
    sys.exit(1) # Keluar jika ada error leksikal

# print("\n--- Daftar Token ---")
# for token in tokens:
#     print(token)
# print("-----\n")

# --- 4. Jalankan Parser ---
parser = SyntaxAnalyzer()
try:
    parse_tree = parser.parse(tokens=tokens)
    # print(parse_tree)
except SyntaxError as e:
    print(str(e), file=sys.stderr)
    sys.exit(1) # Keluar jika ada error sintaks
except Exception as e:
    print(f"\nFATAL PARSER ERROR: {e}", file=sys.stderr)
    import traceback
    traceback.print_exc()
    sys.exit(1)

# --- 4. Jalankan Semantic Analyzer (parse_tree -> AST -> [ASTDecorated,
SymbolTable]) ---
try:
    semantic_analyzer = SemanticAnalyzer()
    decorated_ast, symbol_table, ast = semantic_analyzer.analyze(parse_tree,
debug=True)
    # Print Output
    print(symbol_table)
    print(decorated_ast)

except SemanticError as e:
    print(str(e), file=sys.stderr)
    sys.exit(1) # Keluar jika ada error semantik
except Exception as e:
    print(f"\nFATAL SEMANTIC ERROR: {e}", file=sys.stderr)
    import traceback
    traceback.print_exc()
    return
```



## BAB III

### PENGUJIAN

#### **3.1 Rencana Pengujian**

Rencana pengujian ini bertujuan untuk memvalidasi fungsionalitas, ketahanan (robustness), dan kebenaran dari Semantic Analyzer yang telah diimplementasikan. Berbeda dengan tahap sebelumnya yang berfokus pada struktur kalimat, pengujian pada tahap ini berfokus pada validitas makna program, konsistensi tipe data, dan pengelolaan lingkup (scope) variabel.

Ruang lingkup pengujian mencakup:

1. Manajemen Symbol Table & Scope: Memvalidasi bahwa setiap identifier (variabel, konstanta, prosedur) dimasukkan ke dalam Symbol Table (tab) dengan atribut yang benar, serta memastikan hierarki Block Table (btab) dikelola dengan tepat saat memasuki dan keluar dari prosedur/fungsi (scope local vs global).
2. Validasi Deklarasi (Declaration Checking): Memastikan analyzer dapat mendeteksi kesalahan seperti penggunaan variabel yang belum dideklarasikan (undeclared identifier) atau pendefinisian ulang variabel dengan nama yang sama dalam lingkup yang sama (duplicate identifier).
3. Pemeriksaan Tipe (Type Checking): Menguji konsistensi tipe data dalam operasi assignment dan ekspresi. Memastikan bahwa operasi aritmatika, relasional, dan logika hanya dilakukan pada tipe data yang kompatibel (misalnya: mencegah penjumlahan integer dengan boolean).
4. Anotasi AST (Decorated AST): Memverifikasi bahwa pohon sintaksis abstrak (AST) berhasil "didekorasi" dengan informasi semantik yang tepat, seperti tipe data (type) dan referensi indeks tabel simbol (tab\_index), sesuai dengan spesifikasi output.
5. Error Handling: Memastikan *analyzer* dapat berhenti dengan benar ketika menemukan kesalahan semantik dan memberikan pesan yang informatif

#### **3.2 Test Case**

##### **3.2.1 Test Case 1: Program**

```
Program.pas
program Hello;
variabel
  a, b: integer;
mulai
```

```

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

```

## Output

```

[DEBBUG] Abstract Syntax Tree (AST)
Program("Hello")
  +-- Declarations
    |  +-- VarDecl(name: "a", type: "integer")
    |  +-- VarDecl(name: "b", type: "integer")
  \-- Block
    +-- Assign(target: Var("a"), value: Num(5))
    +-- Assign(target: Var("b"), value:
      |  \-- BinOp("+")
      |    +-- Var("a")
      |    \-- Num(10)
    \-- ProcedureCall(name: "writeln", args: [String("Result = "), Var("b")])

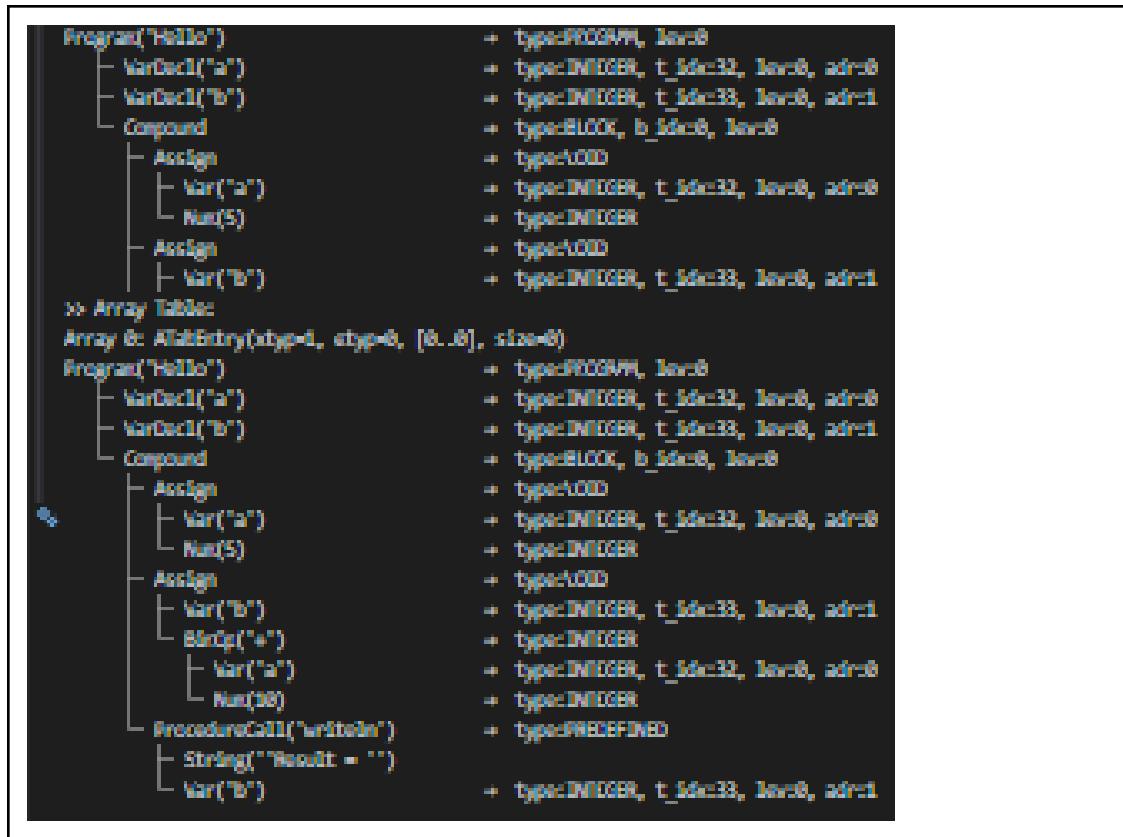
```

>> Symbol Table (Identifier Table):								
Idx	Id	Obj	Type	nm	lev	adr	link	
1	PROGRAM	constant	KOTYPE	1	0	0	0	
2	KONSTAN	constant	KOTYPE	1	0	0	0	
3	TIPE	constant	KOTYPE	1	0	0	0	
4	VARTABEL	constant	KOTYPE	1	0	0	0	
5	PROSEDUR	constant	KOTYPE	1	0	0	0	
6	RUNGSI	constant	KOTYPE	1	0	0	0	
7	MULAI	constant	KOTYPE	1	0	0	0	
8	SELESAI	constant	KOTYPE	1	0	0	0	
9	DERA	constant	KOTYPE	1	0	0	0	
10	MAKA	constant	KOTYPE	1	0	0	0	
11	SEJADIN-DIU	constant	KOTYPE	1	0	0	0	
12	UNTUK	constant	KOTYPE	1	0	0	0	
13	DARI	constant	KOTYPE	1	0	0	0	
14	KE	constant	KOTYPE	1	0	0	0	
15	TURUN-KE	constant	KOTYPE	1	0	0	0	
16	LAKUKAN	constant	KOTYPE	1	0	0	0	
17	SEJUWA	constant	KOTYPE	1	0	0	0	
18	LIANGSI	constant	KOTYPE	1	0	0	0	
19	SIMPATI	constant	KOTYPE	1	0	0	0	
20	DIV	constant	KOTYPE	1	0	0	0	
21	POD	constant	KOTYPE	1	0	0	0	
22	DAN	constant	KOTYPE	1	0	0	0	
23	ATAU	constant	KOTYPE	1	0	0	0	
24	TIDAK	constant	KOTYPE	1	0	0	0	
25	INTEGER	type	INTEGER	1	0	0	0	
26	ECOLEFAN	type	ECOLEFAN	1	0	0	25	
27	CHAR	type	CHAR	1	0	0	26	
28	REAL	type	REAL	1	0	0	27	
29	TRUE	constant	ECOLEFAN	1	0	1	28	
30	FALSE	constant	ECOLEFAN	1	0	0	29	
31	Hello	program	KOTYPE	1	0	0	30	
32	a	variable	INTEGER	1	0	0	31	
33	b	variable	INTEGER	1	0	1	32	

>> Block Table (Scope Info):

>> Array Table:

Array 0: ATabEntry(xtyp=1, ctyp=0, [0..0], size=0)



### 3.2.2 Test Case 2: Constant Modification

test\_constant\_modification.pas

```

program TestConstModification;

konstanta
  PI = 3.14;
  MAX = 100;

variabel
  a, b, c: integer;

mulai
  a := 10;
  b := 20;
  MAX := 200;
  c := (a * b) + (PI * MAX);
  writeln(c);
selesai.
  
```

Output

Semantic Error: Type mismatch in assignment. Cannot assign REAL to INT

### 3.2.3 Test Case 3: Redeclaration

test\_redeclaration.pas

```
program TestRedeclaration;

variabel
  x: integer;
  x: real;
  y: integer;

mulai
  x := 67;
  y := x + 2;
  writeln(y);
selesai.
```

Output

Semantic Error: Duplicate identifier 'x' in the same scope.

### 3.2.4 Test Case 4: Type Mismatch

test\_type\_mismatch.pas

```
program TestTypeMismatch;

variabel
  x: integer;
  y: real;
  flag: boolean;

mulai
  x := 10;
  y := 3.14;
  flag := x;
  x := y;
selesai.
```

Output

```
Semantic Error: Type mismatch in assignment. Cannot assign INTEGER to
```

### 3.2.5 Test Case 5: Undeclared Variable

test\_undeclared\_variable.pas

```
program TestUndeclaredVar;

variabel
    declared: integer;

mulai
    declared := 5;
    undeclared := 10;
    writeln(declared);
    writeln(undeclared);
selesai.
```

Output

```
Semantic Error: Variable 'undeclared' not declared.
```

## 3.3 Ringkasan Hasil Pengujian

Hasil dari pengujian dirangkum dalam tabel berikut :

Kasus Uji Coba	File Input	Tujuan Pengujian	Hasil
Test Case 1	program.pas	Memastikan program valid diproses tanpa error, Symbol Table terisi dengan benar, dan decorated AST.	Berhasil. Program dikompilasi sukses. Symbol Table memuat a dan b, dan Decorated AST terbentuk lengkap dengan tipe datanya.
Test Case 2	test_constant_modification.pas	Validasi Immutability Konstanta.	Berhasil. Program berhenti dan melaporkan Semantic Error. Analyzer mendeteksi upaya assignment ilegal terhadap sebuah konstanta.

Test Case 3	test_redeclaration.pas	Validasi Deklarasi Ganda.	Berhasil. Program berhenti dan melaporkan Semantic Error: "Duplicate identifier 'x' in the same scope".
Test Case 4	test_type_mismatch.pas	Pemeriksaan Tipe (Type Checking).	Berhasil. Program berhenti dan melaporkan Semantic Error: "Type Mismatch".
Test Case 5	test_undeclared_variable.pas	Validasi Identifier Tak Terdeklarasi.	Berhasil. Program berhenti dan melaporkan Semantic Error: "Variable 'undeclared' not declared".

Tabel 3.1. Hasil Pengujian

Secara keseluruhan, Semantic Analyzer yang diimplementasikan telah berhasil melewati semua pengujian. Modul ini terbukti mampu melakukan analisis konteks sensitif dengan benar: berhasil membangun hubungan antara deklarasi dan penggunaan variabel melalui Symbol Table, memvalidasi konsistensi tipe data (Type Checking), serta menegakkan aturan Scope. Selain itu, sistem pelaporan error berfungsi efektif dalam memberikan informasi spesifik mengenai jenis pelanggaran semantik yang terjadi.

## BAB IV

# KESIMPULAN DAN SARAN

### 4.1 Kesimpulan

Berdasarkan laporan yang sudah dibuat, laporan ini dibuat untuk menunjukkan bahwa kelompok MOE berhasil dalam mengembangkan sebuah komponen utama dalam semantic analysis yakni berdasarkan konteks dalam laporan kami yang telah menyinggung beberapa hal sebagai berikut.

#### 1. Sistem Semantic Analysis Berjalan Sesuai Teori

Implementasi mengikuti konsep-konsep klasik seperti

- Attributed Grammar (synthesized & inherited attributes)
- L-Attributed Grammar
- Syntax-Directed Translation
- Decorated AST
- Visitor Pattern

Semua konsep tersebut diintegrasikan dengan baik dalam modul ASTConverter dan SemanticAnalyzer.

#### 2. Desain Struktur Data Sudah Komprehensif

Tiga tabel simbol Pascal-S (tab, btab, atab) berhasil direplikasi dengan struktur yang lengkap:

- *tab* menyimpan detail identifier
- *btab* mengelola blok, scope, parameter, dan variabel lokal
- *atab* menyimpan definisi array

Struktur AST juga dibuat modular menggunakan dataclasses, sehingga mudah diperluas dan diproses kembali oleh analyzer.

#### 3. Mekanisme Checking Sudah Lengkap

Mekanisme checking yang dibuat juga meliputi sebagai berikut:

- Defined checking (undeclared identifier)
- Duplication checking
- Type checking
- Scope resolution
- Array checking
- Const-modification checking

Semua jenis error yang diuraikan pada Bab I.7 juga telah diimplementasi pada kode.

#### 4. Algoritma Implementasi Konsisten dan Terintegrasi

Setiap algoritma pembangunan AST, dekorasi AST, type checking, scope resolution, dan insertion dibahas secara rinci dan sinkron dengan kode Python yang disediakan.

#### 5. Pengujian Menunjukkan Sistem Bekerja Benar

Pengujian pada Bab III mendokumentasikan:

- Berbagai test case untuk assignment, constant modification, redeclaration, type mismatch, dan undeclared variable
- Semua hasil menunjukkan pemeriksaan semantik bekerja sesuai harapan

Hal ini mengindikasikan implementasi stabil untuk semua kasus dasar Pascal-S.

## 4.2 Saran

Berdasarkan capaian pengembangan dari laporan, kelompok MOE memiliki upaya pengembangan untuk meningkatkan hasil tugas kami sebagai berikut:

### 1. Menambahkan Error Recovery

Pada penanganan sekarang sifat error handling masih bersifat fail-fast (yakni langsung berhenti selagi eksekusi). Untuk compiler lengkap, sebaiknya menambahkan:

- Panic mode recovery
- Phrase level recovery

Tujuannya yakni agar error lain tetap dapat terdeteksi meskipun program memiliki banyak kesalahan

### 2. Perluas Type System

Rekomendasi pengembangan selanjutnya:

- Dukungan sebuah tipe record yang lebih kompleks
- Tipe enumerasi
- Tipe subrange
- Type inference (yang digunakan jika ingin meningkatkan sebuah fitur)

### 3. Tingkatkan Integrasi AST → IR (Intermediate Representation)

Setelah semantic analysis, biasanya AST diterjemahkan menjadi sebuah IR yang misalnya terdiri dari 3-address code.

### 4. Tambahkan Visualizer AST dan Symbol Table

Walaupun AST sudah bisa dicetak, sebaiknya dibuat:

- Diagram AST berbasis graph
- Dump tabel simbol seusai parsing

Hal ini dapat meningkatkan proses debugging dan verifikasi jauh lebih mudah.

### 5. Perbaiki Organisasi File untuk Skalabilitas

Struktur terkini sudah cukup baik namun dapat ditingkatkan misalnya:

- Pisahkan visitor untuk semantic rule per kategori
- Buat kelas khusus untuk tipe (misalnya TypeSystem)

## 6. Tambahkan Pengujian Lebih Luas

Pengujian terkini sudah cukup, namun dapat diperluas untuk meningkatkan dan memastikan robustnessnya. Seperti:

- Nested procedures dengan banyak level scope
- Array multidimensi
- Record kompleks
- Kombinasi operasi boolean, arithmetic, dan comparison

## LAMPIRAN

Link Repository: <https://github.com/ivant8k/MOE-Tubes-IF2224/tree/main>

Releases:

- v0.1.1-Milestone 1: Lexical Analysis (19 Oktober 2025)
- v0.2.1-Milestone 2: Syntax Analysis (16 November 2025)
- v0.3.1-Milestone 3 : Semantic Analysis (30 November 2025)

Pembagian Tugas

Nama	NIM	Pembagian Tugas	Percentase
Ivant Samuel Silaban	13523129	<ul style="list-style-type: none"> <li>● Mengimplementasi Symbol Table dan semantic checking.</li> <li>● Menulis Landasan Teori (BAB I).</li> </ul>	20%
Rafa Abdussalam Danadyaksa	13523133	<ul style="list-style-type: none"> <li>● Merancang AST Nodes.</li> <li>● Mengimplementasi pembangunan AST.</li> <li>● Semantic analyzer</li> <li>● Melakukan pengujian dan dokumentasi Hasil Pengujian (BAB III).</li> </ul>	23%
Muhamad Nazih Najmudin	13523144	<ul style="list-style-type: none"> <li>● Mengimplementasi pembentukan Decorated AST.</li> <li>● Merancang sistem semantic error handling.</li> <li>● Menulis Perancangan dan Implementasi (BAB II).</li> </ul>	21%
Anas Ghazi Al Gifari	13523159	<ul style="list-style-type: none"> <li>● Membuat test case untuk semantic analyzer.</li> <li>● Menulis Landasan Teori (BAB I), Lampiran, dan Referensi.</li> <li>● Menyusun kerangka konseptual laporan Milestone 3 secara menyeluruh.</li> </ul>	19%
Muhammad Rizain Firdaus	13523164	<ul style="list-style-type: none"> <li>● Menulis Kesimpulan dan Saran (BAB IV).</li> </ul>	17%



## REFERENSI

- [1] P. Geurts, “Semantic Analysis”, Course Notes, Université de Liège, 2015–2016. [Online]. Available:  
<https://people.montefiore.ulg.ac.be/~geurts/Cours/compil/2015/04-semantic-2015-2016.pdf>
- [2] R. Spivak, “Let’s Build A Simple Interpreter. Part 7: Abstract Syntax Trees,” Ruslan’s Blog, 2016. [Online]. Available: <https://ruslanspivak.com/lsbasi-part7>