

# Bab 1

## Abstract Syntax Tree (AST) dan Struktur Data

### 1.1 Tujuan Pembelajaran

Setelah mempelajari bab ini, mahasiswa diharapkan mampu:

1. Memahami konsep Abstract Syntax Tree (AST) dan perbedaannya dengan parse tree
2. Merancang struktur data AST yang sesuai untuk berbagai konstruk bahasa
3. Mengimplementasikan AST node classes/structs dalam C/C++
4. Menerapkan berbagai metode tree traversal (pre-order, post-order, in-order)
5. Memodifikasi parser untuk membangun AST selama proses parsing
6. Membuat AST visualizer untuk debugging dan pembelajaran

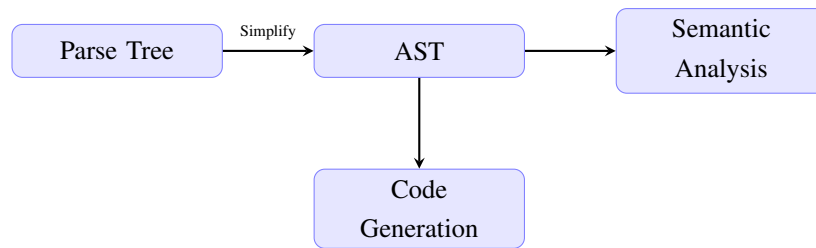
### 1.2 Pendahuluan

Setelah parser berhasil memverifikasi bahwa token-token membentuk struktur yang valid menurut grammar, parser perlu membangun representasi internal program. Representasi ini dapat berupa parse tree (concrete syntax tree) atau Abstract Syntax Tree (AST).

Menurut sumber terbuka:

“Abstract Syntax Trees represent the nested structure of language constructs (expressions, statements, declarations). AST representation: whether nodes represent every symbol from the grammar or a simplified form; whether annotations (types, scopes) are added later.”?

Gambar [1.1](#) menunjukkan posisi AST dalam pipeline kompilator.



Gambar 1.1: Posisi AST dalam pipeline kompilator

### 1.2.1 Perbedaan Parse Tree dan AST

Perbedaan utama antara parse tree dan AST:

- **Parse Tree (Concrete Syntax Tree):**

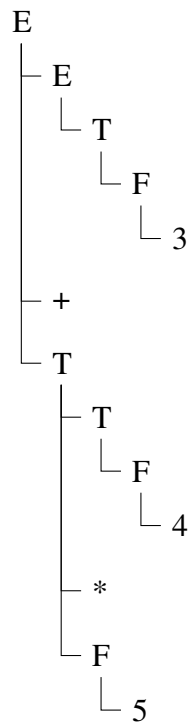
- Mencakup semua detail sintaksis, termasuk semua non-terminal dari grammar
- Setiap node sesuai dengan aturan produksi grammar
- Lebih verbose dan mencakup informasi yang tidak diperlukan untuk fase selanjutnya
- Contoh: Node untuk operator precedence yang sudah jelas dari struktur

- **Abstract Syntax Tree (AST):**

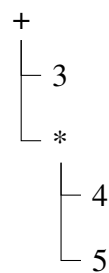
- Menghilangkan detail sintaksis yang tidak relevan
- Fokus pada struktur semantik program
- Lebih kompak dan efisien untuk analisis semantik dan code generation
- Hanya menyertakan informasi yang diperlukan untuk fase-fase selanjutnya

### 1.2.2 Contoh Perbandingan

Untuk ekspresi  $3 + 4 * 5$ , parse tree-nya akan mencakup semua non-terminal:

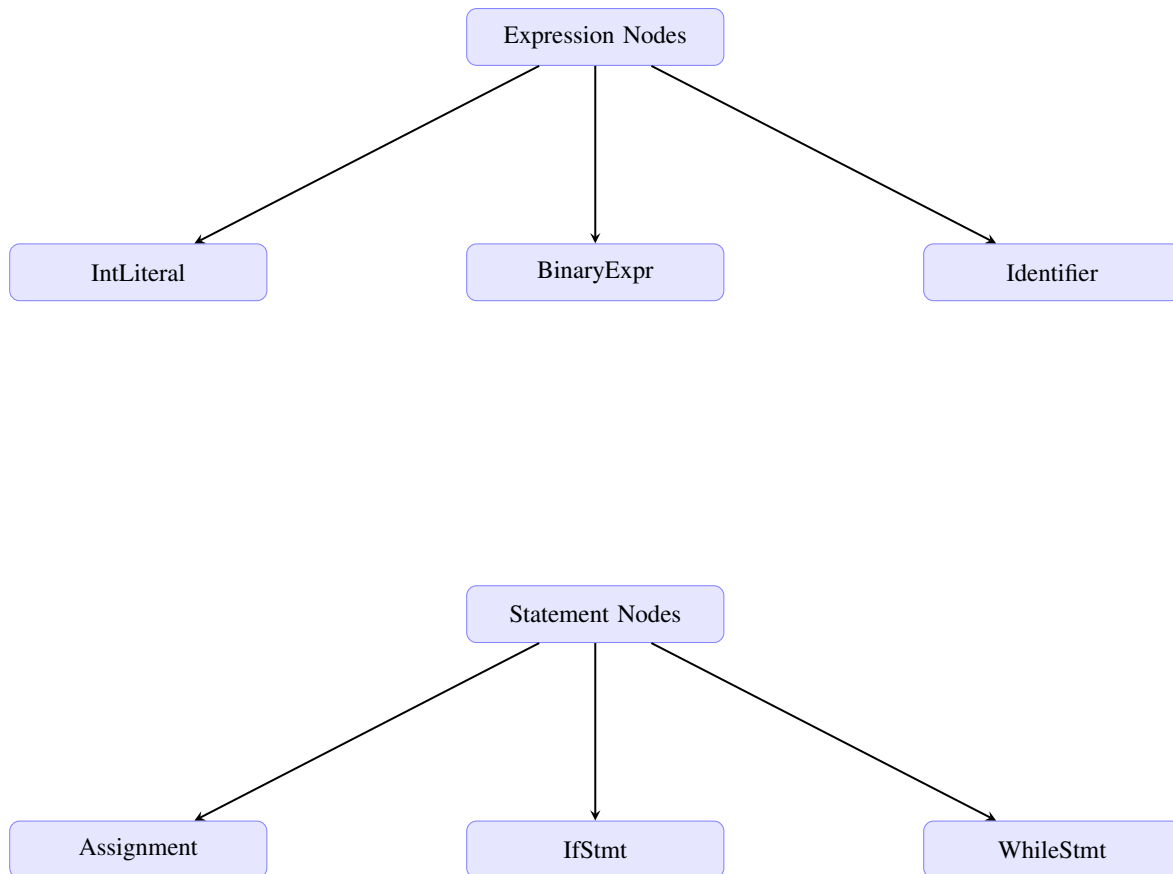
Gambar 1.2: Parse tree untuk ekspresi  $3 + 4 * 5$ 

Sedangkan AST-nya lebih sederhana dan langsung mencerminkan semantik:

Gambar 1.3: AST untuk ekspresi  $3 + 4 * 5$ 

AST menghilangkan node-node intermediate seperti  $E$ ,  $T$ ,  $F$  yang tidak diperlukan untuk pemahaman semantik.

Gambar 1.4 menunjukkan berbagai jenis node dalam AST.



Gambar 1.4: Klasifikasi jenis node dalam Abstract Syntax Tree (AST)

Precedence operator sudah tercermin dalam struktur tree.

### 1.3 Struktur Data AST

AST terdiri dari node-node yang merepresentasikan berbagai konstruk bahasa. Setiap node memiliki:

- **Node Type:** Jenis node (expression, statement, declaration, dll.)
- **Children:** Node-node anak (untuk operasi biner, blok statement, dll.)
- **Attributes:** Informasi tambahan (nilai literal, nama identifier, operator, dll.)
- **Location:** Informasi posisi dalam source code (untuk error reporting)

#### 1.3.1 Node Types dalam AST

Node types yang umum digunakan dalam AST:

**Expression Nodes**

- **Literal Nodes:** Integer, float, string, boolean, null
- **Identifier Nodes:** Nama variabel, fungsi, tipe
- **Binary Expression Nodes:** Operasi biner (+, -, \*, /, ==, !=, <, >, dll.)
- **Unary Expression Nodes:** Operasi unary (negation, logical NOT, address-of, dereference)
- **Function Call Nodes:** Pemanggilan fungsi dengan daftar argumen
- **Array Access Nodes:** Akses elemen array dengan index
- **Member Access Nodes:** Akses member struct/class (dot notation)

**Statement Nodes**

- **Expression Statement:** Statement yang merupakan ekspresi (dengan semicolon)
- **Variable Declaration:** Deklarasi variabel dengan tipe dan optional initializer
- **Assignment Statement:** Assignment ke variabel atau l-value lainnya
- **If Statement:** Conditional statement dengan condition, then-branch, optional else-branch
- **While Statement:** Loop dengan condition dan body
- **For Statement:** Loop dengan init, condition, increment, dan body
- **Return Statement:** Return dengan optional expression
- **Block Statement:** Blok statement yang berisi daftar statement
- **Break/Continue Statement:** Control flow untuk loop

**Declaration Nodes**

- **Function Declaration:** Deklarasi fungsi dengan parameter, return type, dan body
- **Type Declaration:** Deklarasi tipe baru (struct, enum, typedef)
- **Variable Declaration:** Deklarasi variabel global atau local

## Program Node

- **Program/Module Node:** Root node yang berisi semua deklarasi dan statement dalam program

## 1.4 Implementasi AST dalam C++

Dalam C++, AST biasanya diimplementasikan menggunakan inheritance dan virtual functions. Berikut adalah contoh implementasi dasar:

### 1.4.1 Base Node Class

Listing 1.1: Base AST Node Class

```
1 // ASTNode.hpp
2 #ifndef ASTNODE_HPP
3 #define ASTNODE_HPP
4
5 #include <string>
6 #include <memory>
7 #include <vector>
8
9 // Forward declaration
10 class ASTVisitor;
11
12 // Base class untuk semua AST nodes
13 class ASTNode {
14 public:
15     virtual ~ASTNode() = default;
16
17     // Visitor pattern untuk traversal
18     virtual void accept(ASTVisitor& visitor) = 0;
19
20     // Location information untuk error reporting
21     struct Location {
22         int line;
23         int column;
24         std::string filename;
25     };
26
27     Location location;
28 };
29
30 #endif
```

### 1.4.2 Expression Nodes

Listing 1.2: Expression Node Classes

```
1 // Expression.hpp
```

```

2 #ifndef EXPRESSION_HPP
3 #define EXPRESSION_HPP
4
5 #include "ASTNode.hpp"
6 #include <string>
7 #include <memory>
8
9 // Base class untuk semua expressions
10 class Expr : public ASTNode {
11 public:
12     // Type information (akan diisi oleh semantic analyzer)
13     std::string type;
14 };
15
16 // Integer literal node
17 class IntLiteral : public Expr {
18 public:
19     int value;
20
21     IntLiteral(int val) : value(val) {}
22     void accept(ASTVisitor& visitor) override;
23 };
24
25 // Identifier node (variable name, function name, etc.)
26 class Identifier : public Expr {
27 public:
28     std::string name;
29
30     Identifier(const std::string& n) : name(n) {}
31     void accept(ASTVisitor& visitor) override;
32 };
33
34 // Binary expression node (+, -, *, /, ==, etc.)
35 class BinaryExpr : public Expr {
36 public:
37     std::string op; // Operator: "+", "-", "*", "/", "==", etc.
38     std::unique_ptr<Expr> left;
39     std::unique_ptr<Expr> right;
40
41     BinaryExpr(const std::string& op,
42               std::unique_ptr<Expr> l,
43               std::unique_ptr<Expr> r)
44         : op(op), left(std::move(l)), right(std::move(r)) {}
45
46     void accept(ASTVisitor& visitor) override;
47 };
48
49 // Unary expression node (-, !, etc.)
50 class UnaryExpr : public Expr {
51 public:
52     std::string op;
53     std::unique_ptr<Expr> operand;
54
55     UnaryExpr(const std::string& op, std::unique_ptr<Expr> opnd)

```

```

56         : op(op), operand(std::move(opnd)) {}
57
58     void accept(ASTVisitor& visitor) override;
59 };
60
61 // Function call node
62 class FunctionCall : public Expr {
63 public:
64     std::unique_ptr<Identifier> functionName;
65     std::vector<std::unique_ptr<Expr>> arguments;
66
67     FunctionCall(std::unique_ptr<Identifier> name)
68         : functionName(std::move(name)) {}
69
70     void accept(ASTVisitor& visitor) override;
71 };
72
73 #endif

```

### 1.4.3 Statement Nodes

Listing 1.3: Statement Node Classes

```

1 // Statement.hpp
2 #ifndef STATEMENT_HPP
3 #define STATEMENT_HPP
4
5 #include "ASTNode.hpp"
6 #include "Expression.hpp"
7 #include <memory>
8 #include <vector>
9
10 // Base class untuk semua statements
11 class Stmt : public ASTNode {
12 };
13
14 // Expression statement (expression followed by semicolon)
15 class ExprStmt : public Stmt {
16 public:
17     std::unique_ptr<Expr> expression;
18
19     ExprStmt(std::unique_ptr<Expr> expr)
20         : expression(std::move(expr)) {}
21
22     void accept(ASTVisitor& visitor) override;
23 };
24
25 // Variable declaration statement
26 class VarDecl : public Stmt {
27 public:
28     std::string typeName;
29     std::string varName;
30     std::unique_ptr<Expr> initializer; // Optional

```



```

31
32     VarDecl(const std::string& type, const std::string& name)
33         : typeName(type), varName(name), initializer(nullptr) {}
34
35     VarDecl(const std::string& type, const std::string& name,
36             std::unique_ptr<Expr> init)
37         : typeName(type), varName(name), initializer(std::move(init)) {}
38
39     void accept(ASTVisitor& visitor) override;
40 };
41
42 // Assignment statement
43 class AssignStmt : public Stmt {
44 public:
45     std::unique_ptr<Identifier> left;
46     std::unique_ptr<Expr> right;
47
48     AssignStmt(std::unique_ptr<Identifier> l, std::unique_ptr<Expr> r)
49         : left(std::move(l)), right(std::move(r)) {}
50
51     void accept(ASTVisitor& visitor) override;
52 };
53
54 // If statement
55 class IfStmt : public Stmt {
56 public:
57     std::unique_ptr<Expr> condition;
58     std::unique_ptr<Stmt> thenBranch;
59     std::unique_ptr<Stmt> elseBranch; // Optional
60
61     IfStmt(std::unique_ptr<Expr> cond, std::unique_ptr<Stmt> thenStmt)
62         : condition(std::move(cond)),
63           thenBranch(std::move(thenStmt)),
64           elseBranch(nullptr) {}
65
66     IfStmt(std::unique_ptr<Expr> cond,
67            std::unique_ptr<Stmt> thenStmt,
68            std::unique_ptr<Stmt> elseStmt)
69         : condition(std::move(cond)),
70           thenBranch(std::move(thenStmt)),
71           elseBranch(std::move(elseStmt)) {}
72
73     void accept(ASTVisitor& visitor) override;
74 };
75
76 // While statement
77 class WhileStmt : public Stmt {
78 public:
79     std::unique_ptr<Expr> condition;
80     std::unique_ptr<Stmt> body;
81
82     WhileStmt(std::unique_ptr<Expr> cond, std::unique_ptr<Stmt> b)
83         : condition(std::move(cond)), body(std::move(b)) {}
84

```

```

85     void accept(ASTVisitor& visitor) override;
86 };
87
88 // Block statement (compound statement)
89 class BlockStmt : public Stmt {
90 public:
91     std::vector<std::unique_ptr<Stmt>> statements;
92
93     void addStatement(std::unique_ptr<Stmt> stmt) {
94         statements.push_back(std::move(stmt));
95     }
96
97     void accept(ASTVisitor& visitor) override;
98 };
99
100 // Return statement
101 class ReturnStmt : public Stmt {
102 public:
103     std::unique_ptr<Expr> expression; // Optional
104
105     ReturnStmt() : expression(nullptr) {}
106     ReturnStmt(std::unique_ptr<Expr> expr) : expression(std::move(expr))
107     → {}
108
109     void accept(ASTVisitor& visitor) override;
110 };
111 #endif

```

### 1.4.4 Program Node

Listing 1.4: Program Node Class

```

1 // Program.hpp
2 #ifndef PROGRAM_HPP
3 #define PROGRAM_HPP
4
5 #include "ASTNode.hpp"
6 #include "Statement.hpp"
7 #include <vector>
8 #include <memory>
9
10 // Root node untuk seluruh program
11 class Program : public ASTNode {
12 public:
13     std::vector<std::unique_ptr<Stmt>> statements;
14
15     void addStatement(std::unique_ptr<Stmt> stmt) {
16         statements.push_back(std::move(stmt));
17     }
18
19     void accept(ASTVisitor& visitor) override;
20 };

```

```

21
22 #endif

```

## 1.5 Visitor Pattern untuk Tree Traversal

Visitor pattern memisahkan algoritma traversal dari struktur node. Ini memungkinkan kita menambahkan operasi baru tanpa memodifikasi kelas node.

### 1.5.1 Visitor Interface

Listing 1.5: AST Visitor Interface

```

1 // ASTVisitor.hpp
2 #ifndef ASTVISITOR_HPP
3 #define ASTVISITOR_HPP
4
5 // Forward declarations
6 class IntLiteral;
7 class Identifier;
8 class BinaryExpr;
9 class UnaryExpr;
10 class FunctionCall;
11 class ExprStmt;
12 class VarDecl;
13 class AssignStmt;
14 class IfStmt;
15 class WhileStmt;
16 class BlockStmt;
17 class ReturnStmt;
18 class Program;
19
20 class ASTVisitor {
21 public:
22     virtual ~ASTVisitor() = default;
23
24     // Expression visitors
25     virtual void visit(IntLiteral& node) = 0;
26     virtual void visit(Identifier& node) = 0;
27     virtual void visit(BinaryExpr& node) = 0;
28     virtual void visit(UnaryExpr& node) = 0;
29     virtual void visit(FunctionCall& node) = 0;
30
31     // Statement visitors
32     virtual void visit(ExprStmt& node) = 0;
33     virtual void visit(VarDecl& node) = 0;
34     virtual void visit(AssignStmt& node) = 0;
35     virtual void visit(IfStmt& node) = 0;
36     virtual void visit(WhileStmt& node) = 0;
37     virtual void visit(BlockStmt& node) = 0;
38     virtual void visit(ReturnStmt& node) = 0;
39

```

```
40 // Program visitor
41 virtual void visit(Program& node) = 0;
42 };
43
44 #endif
```

## 1.5.2 Implementasi Accept Methods

Setiap node class mengimplementasikan method `accept` yang memanggil method `visit` yang sesuai pada visitor:

Listing 1.6: Accept Methods Implementation

```
1 // Expression.cpp
2 #include "Expression.hpp"
3 #include "ASTVisitor.hpp"
4
5 void IntLiteral::accept (ASTVisitor& visitor) {
6     visitor.visit(*this);
7 }
8
9 void Identifier::accept (ASTVisitor& visitor) {
10    visitor.visit(*this);
11 }
12
13 void BinaryExpr::accept (ASTVisitor& visitor) {
14    visitor.visit(*this);
15    if (left) left->accept(visitor);
16    if (right) right->accept(visitor);
17 }
18
19 void UnaryExpr::accept (ASTVisitor& visitor) {
20    visitor.visit(*this);
21    if (operand) operand->accept(visitor);
22 }
23
24 void FunctionCall::accept (ASTVisitor& visitor) {
25    visitor.visit(*this);
26    if (functionName) functionName->accept(visitor);
27    for (auto& arg : arguments) {
28        if (arg) arg->accept(visitor);
29    }
30 }
```

## 1.6 Tree Traversal Methods

Ada tiga metode utama untuk traversing tree: pre-order, post-order, dan in-order.

### 1.6.1 Pre-Order Traversal

Pre-order traversal mengunjungi node sebelum children-nya. Urutan: Root  $\rightarrow$  Left  $\rightarrow$  Right.

Contoh untuk ekspresi  $3 + 4 * 5$ :

```
Visit: +  
Visit: 3  
Visit: *  
Visit: 4  
Visit: 5
```

Pre-order berguna untuk:

- Menyalin tree
- Prefix notation (Polish notation)
- Print struktur tree

### 1.6.2 Post-Order Traversal

Post-order traversal mengunjungi node setelah children-nya. Urutan: Left  $\rightarrow$  Right  $\rightarrow$  Root.

Contoh untuk ekspresi  $3 + 4 * 5$ :

```
Visit: 3  
Visit: 4  
Visit: 5  
Visit: *  
Visit: +
```

Post-order berguna untuk:

- Menghapus tree (deallocate memory)
- Postfix notation (Reverse Polish notation)
- Evaluasi ekspresi
- Code generation (stack-based)

### 1.6.3 In-Order Traversal

In-order traversal mengunjungi left child, kemudian node, kemudian right child. Urutan: Left  $\rightarrow$  Root  $\rightarrow$  Right.

Contoh untuk ekspresi  $3 + 4 * 5$ :

```
Visit: 3
Visit: +
Visit: 4
Visit: *
Visit: 5
```

In-order berguna untuk:

- Infix notation (seperti yang ditulis dalam source code)
- Binary search tree operations

### 1.6.4 Implementasi Traversal dengan Visitor

Berikut adalah contoh implementasi PrettyPrinter visitor yang menggunakan pre-order traversal:

Listing 1.7: PrettyPrinter Visitor

```
1 // PrettyPrinter.hpp
2 #ifndef PRETTYPRINTER_HPP
3 #define PRETTYPRINTER_HPP
4
5 #include "ASTVisitor.hpp"
6 #include <iostream>
7 #include <string>
8
9 class PrettyPrinter : public ASTVisitor {
10 private:
11     int indentLevel;
12
13     std::string indent() const {
14         return std::string(indentLevel * 2, ' ');
15     }
16
17 public:
18     PrettyPrinter() : indentLevel(0) {}
19
20     void visit(IntLiteral& node) override {
21         std::cout << indent() << "IntLiteral: " << node.value << std::
↵ endl;
22     }
23
24     void visit(Identifier& node) override {
25         std::cout << indent() << "Identifier: " << node.name << std::endl
↵ ;
26     }
27
28     void visit(BinaryExpr& node) override {
29         std::cout << indent() << "BinaryExpr: " << node.op << std::endl;
30         indentLevel++;
31         if (node.left) node.left->accept(*this);
32         if (node.right) node.right->accept(*this);
```

```

33     indentLevel--;
34 }
35
36 void visit(UnaryExpr& node) override {
37     std::cout << indent() << "UnaryExpr: " << node.op << std::endl;
38     indentLevel++;
39     if (node.operand) node.operand->accept(*this);
40     indentLevel--;
41 }
42
43 void visit(FunctionCall& node) override {
44     std::cout << indent() << "FunctionCall" << std::endl;
45     indentLevel++;
46     if (node.functionName) node.functionName->accept(*this);
47     for (auto& arg : node.arguments) {
48         if (arg) arg->accept(*this);
49     }
50     indentLevel--;
51 }
52
53 void visit(ExprStmt& node) override {
54     std::cout << indent() << "ExprStmt" << std::endl;
55     indentLevel++;
56     if (node.expression) node.expression->accept(*this);
57     indentLevel--;
58 }
59
60 void visit(VarDecl& node) override {
61     std::cout << indent() << "VarDecl: " << node.typeName
62         << " " << node.varName;
63     if (node.initializer) {
64         std::cout << " = ";
65         indentLevel++;
66         node.initializer->accept(*this);
67         indentLevel--;
68     }
69     std::cout << std::endl;
70 }
71
72 void visit(AssignStmt& node) override {
73     std::cout << indent() << "AssignStmt" << std::endl;
74     indentLevel++;
75     if (node.left) node.left->accept(*this);
76     if (node.right) node.right->accept(*this);
77     indentLevel--;
78 }
79
80 void visit(IfStmt& node) override {
81     std::cout << indent() << "IfStmt" << std::endl;
82     indentLevel++;
83     std::cout << indent() << "Condition:" << std::endl;
84     indentLevel++;
85     if (node.condition) node.condition->accept(*this);
86     indentLevel--;

```

```
87     std::cout << indent() << "Then:" << std::endl;
88     indentLevel++;
89     if (node.thenBranch) node.thenBranch->accept(*this);
90     indentLevel--;
91     if (node.elseBranch) {
92         std::cout << indent() << "Else:" << std::endl;
93         indentLevel++;
94         node.elseBranch->accept(*this);
95         indentLevel--;
96     }
97     indentLevel--;
98 }
99
100 void visit(WhileStmt& node) override {
101     std::cout << indent() << "WhileStmt" << std::endl;
102     indentLevel++;
103     std::cout << indent() << "Condition:" << std::endl;
104     indentLevel++;
105     if (node.condition) node.condition->accept(*this);
106     indentLevel--;
107     std::cout << indent() << "Body:" << std::endl;
108     indentLevel++;
109     if (node.body) node.body->accept(*this);
110     indentLevel--;
111     indentLevel--;
112 }
113
114 void visit(BlockStmt& node) override {
115     std::cout << indent() << "BlockStmt" << std::endl;
116     indentLevel++;
117     for (auto& stmt : node.statements) {
118         if (stmt) stmt->accept(*this);
119     }
120     indentLevel--;
121 }
122
123 void visit(ReturnStmt& node) override {
124     std::cout << indent() << "ReturnStmt";
125     if (node.expression) {
126         std::cout << std::endl;
127         indentLevel++;
128         node.expression->accept(*this);
129         indentLevel--;
130     } else {
131         std::cout << std::endl;
132     }
133 }
134
135 void visit(Program& node) override {
136     std::cout << "Program" << std::endl;
137     indentLevel++;
138     for (auto& stmt : node.statements) {
139         if (stmt) stmt->accept(*this);
140     }
```



```

141         indentLevel--;
142     }
143 };
144
145 #endif

```

## 1.7 Integrasi AST dengan Parser

Parser perlu dimodifikasi untuk membangun AST selama proses parsing, bukan hanya memverifikasi grammar. Berikut adalah contoh integrasi dengan Bison parser generator.

### 1.7.1 Bison Grammar dengan AST Building

Listing 1.8: Bison Grammar dengan AST Building

```

1  %{
2  #include "ASTNode.hpp"
3  #include "Expression.hpp"
4  #include "Statement.hpp"
5  #include "Program.hpp"
6  %}
7
8  %union {
9      int intVal;
10     char* strVal;
11     Expr* expr;
12     Stmt* stmt;
13     Program* program;
14 }
15
16 %token <intVal> INT_LITERAL
17 %token <strVal> IDENTIFIER
18 %token PLUS MINUS MULTIPLY DIVIDE
19 %token ASSIGN SEMICOLON
20 %token IF ELSE WHILE RETURN
21 %token LBACE RBACE LPAREN RPAREN
22
23 %type <expr> expression
24 %type <stmt> statement block_statement
25 %type <program> program
26
27 %%
28
29 program:
30     statement_list {
31         $$ = new Program();
32         // Add statements to program
33     }
34     ;
35
36 statement_list:

```

```
37     statement_list statement {
38         // Add statement to list
39     }
40 | /* empty */
41 ;
42
43 statement:
44     expression SEMICOLON {
45         $$ = new ExprStmt($1);
46     }
47 | IDENTIFIER ASSIGN expression SEMICOLON {
48     auto id = new Identifier($1);
49     $$ = new AssignStmt(std::unique_ptr<Identifier>(id),
50                         std::unique_ptr<Expr>($3));
51 }
52 | block_statement
53 ;
54
55 block_statement:
56     LBRACE statement_list RBRACE {
57         $$ = new BlockStmt();
58         // Add statements to block
59     }
60 ;
61
62 expression:
63     INT_LITERAL {
64         $$ = new IntLiteral($1);
65     }
66 | IDENTIFIER {
67     $$ = new Identifier($1);
68 }
69 | expression PLUS expression {
70     $$ = new BinaryExpr("+",
71                         std::unique_ptr<Expr>($1),
72                         std::unique_ptr<Expr>($3));
73 }
74 | expression MINUS expression {
75     $$ = new BinaryExpr("-",
76                         std::unique_ptr<Expr>($1),
77                         std::unique_ptr<Expr>($3));
78 }
79 | expression MULTIPLY expression {
80     $$ = new BinaryExpr("*",
81                         std::unique_ptr<Expr>($1),
82                         std::unique_ptr<Expr>($3));
83 }
84 | expression DIVIDE expression {
85     $$ = new BinaryExpr("/",
86                         std::unique_ptr<Expr>($1),
87                         std::unique_ptr<Expr>($3));
88 }
89 | LPAREN expression RPAREN {
90     $$ = $2;
```

```

91     }
92     ;
93
94 %%

```

## 1.8 AST Visualizer

AST visualizer berguna untuk debugging dan memahami struktur program. Berikut adalah contoh sederhana menggunakan text-based visualization:

Listing 1.9: AST Text Visualizer

```

1  // ASTVisualizer.hpp
2  #ifndef ASTVISIZER_HPP
3  #define ASTVISIZER_HPP
4
5  #include "ASTVisitor.hpp"
6  #include <iostream>
7  #include <string>
8  #include <vector>
9
10 class ASTVisualizer : public ASTVisitor {
11 private:
12     std::vector<bool> isLastChild;
13
14     void printPrefix() {
15         for (size_t i = 0; i < isLastChild.size() - 1; i++) {
16             if (isLastChild[i]) {
17                 std::cout << "    ";
18             } else {
19                 std::cout << "|    ";
20             }
21         }
22         if (!isLastChild.empty()) {
23             std::cout << (isLastChild.back() ? "+-- " : "+-- ");
24         }
25     }
26
27 public:
28     void visit(BinaryExpr& node) override {
29         printPrefix();
30         std::cout << "BinaryExpr: " << node.op << std::endl;
31
32         isLastChild.push_back(false);
33         if (node.left) node.left->accept(*this);
34         isLastChild.back() = true;
35         if (node.right) node.right->accept(*this);
36         isLastChild.pop_back();
37     }
38
39     void visit(IntLiteral& node) override {
40         printPrefix();
41         std::cout << "IntLiteral: " << node.value << std::endl;

```

```
42     }
43
44     void visit(Identifier& node) override {
45         printPrefix();
46         std::cout << "Identifier: " << node.name << std::endl;
47     }
48
49     // Implement other visit methods...
50 };
51
52 #endif
```

## 1.9 Best Practices

Berikut adalah beberapa best practices dalam implementasi AST:

1. **Memory Management:** Gunakan smart pointers (`std::unique_ptr` atau `std::shared_ptr`) untuk menghindari memory leaks
2. **Immutable Nodes:** AST nodes biasanya immutable setelah dibuat. Transformasi menghasilkan tree baru
3. **Location Tracking:** Simpan informasi lokasi (line, column) di setiap node untuk error reporting yang lebih baik
4. **Separation of Concerns:** AST nodes hanya menyimpan struktur, bukan algoritma. Gunakan visitor pattern untuk operasi
5. **Type Safety:** Gunakan type system yang kuat (inheritance, enums) untuk mencegah kesalahan
6. **Modularity:** Pisahkan definisi node types ke file-file terpisah untuk kemudahan maintenance

## 1.10 Kesimpulan

Dalam bab ini, kita telah mempelajari:

1. Abstract Syntax Tree (AST) adalah representasi abstrak dari struktur program yang menghilangkan detail sintaksis yang tidak relevan
2. AST terdiri dari berbagai node types: expression nodes, statement nodes, declaration nodes, dan program node
3. Implementasi AST dalam C++ menggunakan inheritance dan virtual functions, dengan smart pointers untuk memory management

4. Visitor pattern memisahkan algoritma traversal dari struktur node, memungkinkan ekstensibilitas tanpa modifikasi node classes
5. Tree traversal dapat dilakukan dengan pre-order, post-order, atau in-order, masing-masing berguna untuk tujuan berbeda
6. Parser dapat dibangun untuk menghasilkan AST selama proses parsing menggunakan semantic actions

Pemahaman tentang AST sangat penting karena AST menjadi input untuk fase-fase selanjutnya: semantic analysis, optimization, dan code generation.

## 1.11 Referensi dan Bahan Bacaan Lanjutan

Untuk memperdalam pemahaman tentang AST, mahasiswa disarankan membaca:

- **Dragon Book:** Aho, Lam, Sethi, & Ullman (2006). *Compilers: Principles, Techniques, and Tools* ? - Bab 2: A Simple Syntax-Directed Translator, Bab 5: Syntax-Directed Translation
- **Engineering a Compiler:** Cooper & Torczon (2011) ? - Bab 4: Intermediate Representations
- **UC San Diego CSE 231:** Course materials tentang AST construction ?
- **Johns Hopkins University EN.601.428:** Course tentang AST dan syntax trees ?
- **GNU Bison Manual:** Dokumentasi tentang semantic actions dan AST building ?