Bab 9

Abstract Syntax Tree (AST) dan Struktur Data

9.1 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

9.2 Pendahuluan

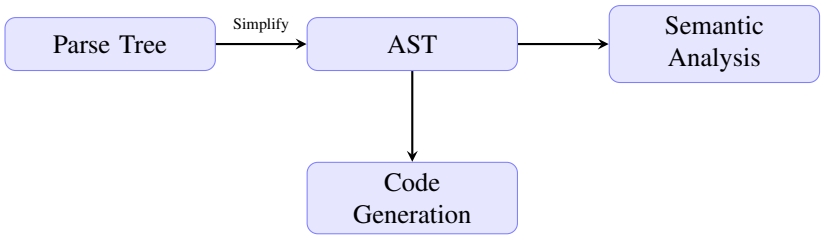
Dalam proyek compiler subset C, parser (Bab 8, `simplec.y`) membangun AST sebagai representasi internal program. Definisinya untuk subset C disajikan di Bagian 9.3; contoh berikut mengacu ke node types tersebut.

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."[1]

Gambar 9.1 menunjukkan posisi AST dalam pipeline kompilator.



Gambar 9.1: Posisi AST dalam pipeline kompilator

9.2.1 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

9.2.2 Contoh Perbandingan

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

Sedangkan AST-nya lebih sederhana dan langsung mencerminkan semantik:

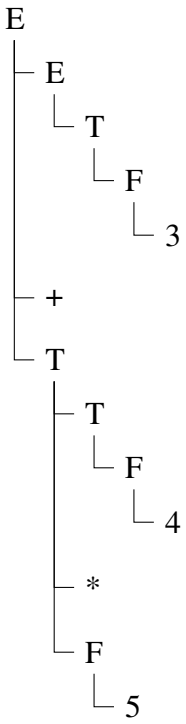
AST menghilangkan node-node intermediate seperti E, T, F yang tidak diperlukan untuk pemahaman semantik.

Gambar 9.4 menunjukkan berbagai jenis node dalam AST.

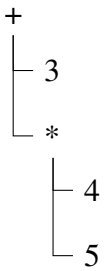
Precedence operator sudah tercermin dalam struktur tree.

9.3 AST Proyek Subset C

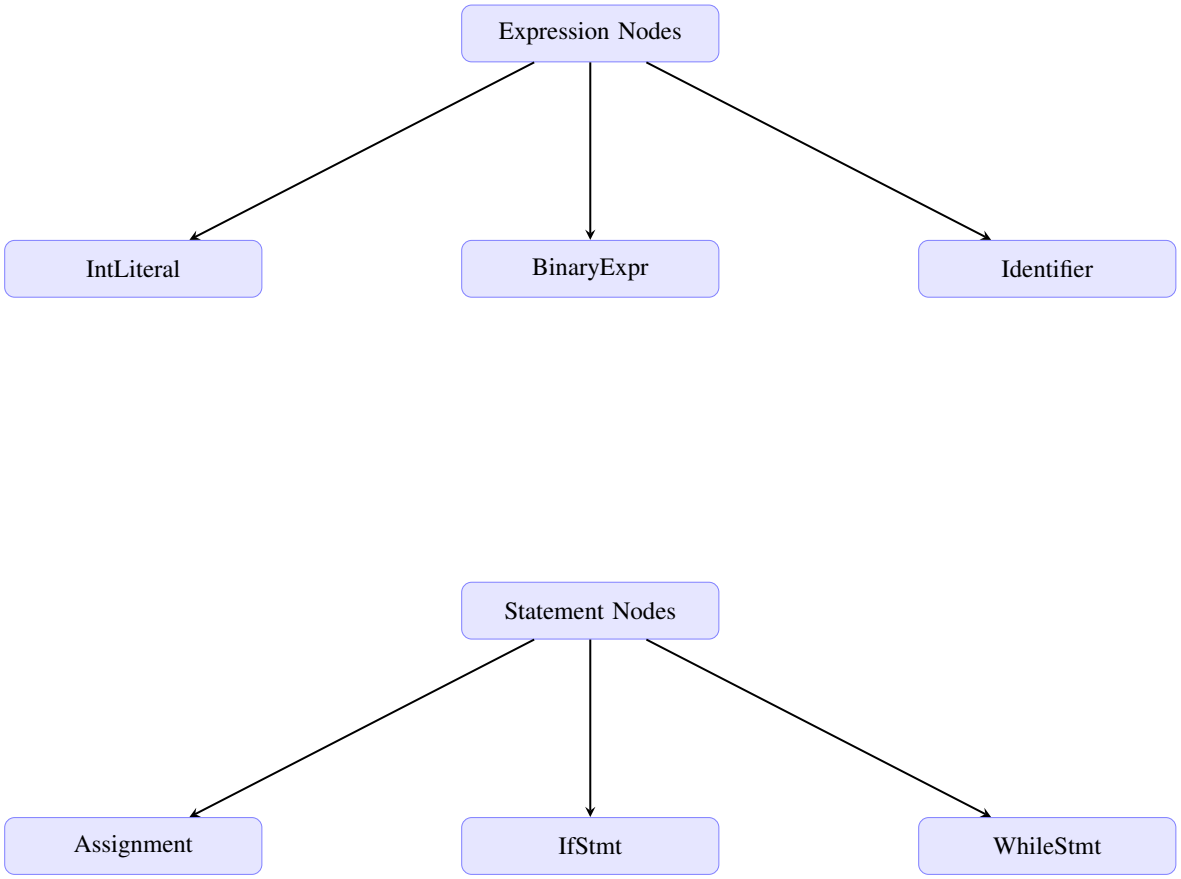
Untuk proyek compiler subset C (spesifikasi Bab 1, grammar Bab 5), AST didefinisikan sebagai berikut. Parser proyek di Bab 8 (`simplec.y`) membangun AST ini melalui semantic actions; definisi node mengacu ke grammar proyek (Bagian **??**).



Gambar 9.2: Parse tree untuk ekspresi `3 + 4 * 5`



Gambar 9.3: AST untuk ekspresi `3 + 4 * 5`



Gambar 9.4: Klasifikasi jenis node dalam Abstract Syntax Tree (AST)

9.3.1 Node Types AST Proyek

- **Program**: root; berisi daftar statement (sequence).
- **Declaration**: `int` identifier atau `float` identifier; menyimpan nama dan tipe.
- **Assignment**: identifier = expr; menyimpan nama variabel dan pointer ke node ekspresi.
- **PrintStmt**: `print ( arg )` ; arg berupa string-literal atau expr.
- **Expr**: ekspresi; subtipe: BinOp (op, left, right), Number (literal), FloatLiteral, Identifier (nama), StringLiteral (nilai).
- **BinOp**: operator biner (+, −, ⋆, /); anak kiri dan kanan bertipe expr.

Struktur data dapat diimplementasikan dalam `ast.h` / `ast.c` di folder proyek (`proyek-compiler-subset-c/`). Traversal (pre-order, post-order) untuk semantic analysis dan code generation mengacu ke node-node ini. Bab 10 (symbol table) dan Bab 11 (type checking) memakai AST ini sebagai input; Bab 12 (IR) menghasilkan three-address code atau quadruples dari AST proyek.

9.4 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)

9.4.1 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

9.5 Implementasi AST dalam C++

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

9.5.1 Base Node Class

Listing 9.1: Base AST Node Class

```cpp
// ASTNode.hpp
#ifndef ASTNODE_HPP
#define ASTNODE_HPP

#include <string>
#include <memory>
#include <vector>

// Forward declaration
class ASTVisitor;

// Base class untuk semua AST nodes
class ASTNode {
public:
    virtual ~ASTNode() = default;

    // Visitor pattern untuk traversal
    virtual void accept(ASTVisitor& visitor) = 0;

    // Location information untuk error reporting
    struct Location {
        int line;
        int column;
        std::string filename;
    };

    Location location;
};

#endif
```

9.5.2 Expression Nodes

Listing 9.2: Expression Node Classes

```cpp
// Expression.hpp
#ifndef EXPRESSION_HPP
#define EXPRESSION_HPP

#include "ASTNode.hpp"
#include <string>
#include <memory>

// Base class untuk semua expressions
class Expr : public ASTNode {
public:
    // Type information (akan diisi oleh semantic analyzer)
    std::string type;
};

```

```
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
```

9.5.3 Statement Nodes

Listing 9.3: 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     void accept(ASTVisitor& visitor) override;
109 };
110
111 #endif
```

9.5.4 Program Node

Listing 9.4: 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
```

9.6 Visitor Pattern untuk Tree Traversal

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

9.6.1 Visitor Interface

Listing 9.5: AST Visitor Interface

```
// ASTVisitor.hpp
#ifndef ASTVISITOR_HPP
#define ASTVISITOR_HPP

// Forward declarations
class IntLiteral;
class Identifier;
class BinaryExpr;
class UnaryExpr;
class FunctionCall;
class ExprStmt;
class VarDecl;
class AssignStmt;
class IfStmt;
class WhileStmt;
class BlockStmt;
class ReturnStmt;
class Program;

class ASTVisitor {
public:
    virtual ~ASTVisitor() = default;

    // Expression visitors
    virtual void visit(IntLiteral& node) = 0;
    virtual void visit(Identifier& node) = 0;
    virtual void visit(BinaryExpr& node) = 0;
    virtual void visit(UnaryExpr& node) = 0;
    virtual void visit(FunctionCall& node) = 0;

    // Statement visitors
    virtual void visit(ExprStmt& node) = 0;
    virtual void visit(VarDecl& node) = 0;
    virtual void visit(AssignStmt& node) = 0;
    virtual void visit(IfStmt& node) = 0;
    virtual void visit(WhileStmt& node) = 0;
    virtual void visit(BlockStmt& node) = 0;
    virtual void visit(ReturnStmt& node) = 0;

    // Program visitor
    virtual void visit(Program& node) = 0;
};

#endif
```

9.6.2 Implementasi Accept Methods

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

Listing 9.6: Accept Methods Implementation

```cpp
// Expression.cpp
#include "Expression.hpp"
#include "ASTVisitor.hpp"

void IntLiteral::accept(ASTVisitor& visitor) {
    visitor.visit(*this);
}

void Identifier::accept(ASTVisitor& visitor) {
    visitor.visit(*this);
}

void BinaryExpr::accept(ASTVisitor& visitor) {
    visitor.visit(*this);
    if (left) left->accept(visitor);
    if (right) right->accept(visitor);
}

void UnaryExpr::accept(ASTVisitor& visitor) {
    visitor.visit(*this);
    if (operand) operand->accept(visitor);
}

void FunctionCall::accept(ASTVisitor& visitor) {
    visitor.visit(*this);
    if (functionName) functionName->accept(visitor);
    for (auto& arg : arguments) {
        if (arg) arg->accept(visitor);
    }
}
```

9.7 Tree Traversal Methods

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

9.7.1 Pre-Order Traversal

Pre-order traversal mengunjungi node sebelum children-nya. Urutan: Root → Left → 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

9.7.2 Post-Order Traversal

Post-order traversal mengunjungi node setelah children-nya. Urutan: Left → Right → 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)

9.7.3 In-Order Traversal

In-order traversal mengunjungi left child, kemudian node, kemudian right child. Urutan: Left → Root → 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

9.7.4 Implementasi Traversal dengan Visitor

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

Listing 9.7: PrettyPrinter Visitor

```
// PrettyPrinter.hpp
#ifndef PRETTYPRINTER_HPP
#define PRETTYPRINTER_HPP

#include "ASTVisitor.hpp"
#include <iostream>
#include <string>

class PrettyPrinter : public ASTVisitor {
private:
    int indentLevel;

    std::string indent() const {
        return std::string(indentLevel * 2, ' ');
    }

public:
    PrettyPrinter() : indentLevel(0) {}

    void visit(IntLiteral& node) override {
        std::cout << indent() << "IntLiteral: " << node.value << std::
↪ endl;
    }

    void visit(Identifier& node) override {
        std::cout << indent() << "Identifier: " << node.name << std::endl
↪ ;
    }

    void visit(BinaryExpr& node) override {
        std::cout << indent() << "BinaryExpr: " << node.op << std::endl;
        indentLevel++;
        if (node.left) node.left->accept(*this);
        if (node.right) node.right->accept(*this);
        indentLevel--;
    }

    void visit(UnaryExpr& node) override {
        std::cout << indent() << "UnaryExpr: " << node.op << std::endl;
        indentLevel++;
        if (node.operand) node.operand->accept(*this);
        indentLevel--;
    }

    void visit(FunctionCall& node) override {
        std::cout << indent() << "FunctionCall" << std::endl;
        indentLevel++;
        if (node.functionName) node.functionName->accept(*this);
        for (auto& arg : node.arguments) {
            if (arg) arg->accept(*this);
        }
        indentLevel--;
    }
```

```
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);
```

```
        indentLevel--;
        std::cout << indent() << "Body:" << std::endl;
        indentLevel++;
        if (node.body) node.body->accept(*this);
        indentLevel--;
        indentLevel--;
    }

    void visit(BlockStmt& node) override {
        std::cout << indent() << "BlockStmt" << std::endl;
        indentLevel++;
        for (auto& stmt : node.statements) {
            if (stmt) stmt->accept(*this);
        }
        indentLevel--;
    }

    void visit(ReturnStmt& node) override {
        std::cout << indent() << "ReturnStmt";
        if (node.expression) {
            std::cout << std::endl;
            indentLevel++;
            node.expression->accept(*this);
            indentLevel--;
        } else {
            std::cout << std::endl;
        }
    }

    void visit(Program& node) override {
        std::cout << "Program" << std::endl;
        indentLevel++;
        for (auto& stmt : node.statements) {
            if (stmt) stmt->accept(*this);
        }
        indentLevel--;
    }
};

#endif
```

9.8 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.

9.8.1 Bison Grammar dengan AST Building

Listing 9.8: Bison Grammar dengan AST Building

```
%{
```

```
#include "ASTNode.hpp"
#include "Expression.hpp"
#include "Statement.hpp"
#include "Program.hpp"
%}

%union {
    int intVal;
    char* strVal;
    Expr* expr;
    Stmt* stmt;
    Program* program;
}

%token <intVal> INT_LITERAL
%token <strVal> IDENTIFIER
%token PLUS MINUS MULTIPLY DIVIDE
%token ASSIGN SEMICOLON
%token IF ELSE WHILE RETURN
%token LBRACE RBRACE LPAREN RPAREN

%type <expr> expression
%type <stmt> statement block_statement
%type <program> program

%%

program:
    statement_list {
        $$ = new Program();
        // Add statements to program
    }
    ;

statement_list:
    statement_list statement {
        // Add statement to list
    }
  | /* empty */
    ;

statement:
    expression SEMICOLON {
        $$ = new ExprStmt($1);
    }
  | IDENTIFIER ASSIGN expression SEMICOLON {
        auto id = new Identifier($1);
        $$ = new AssignStmt(std::unique_ptr<Identifier>(id),
                            std::unique_ptr<Expr>($3));
    }
  | block_statement
    ;

block_statement:
```

```
    LBRACE statement_list RBRACE {
        $$ = new BlockStmt();
        // Add statements to block
    }
    ;

expression:
    INT_LITERAL {
        $$ = new IntLiteral($1);
    }
  | IDENTIFIER {
        $$ = new Identifier($1);
    }
  | expression PLUS expression {
        $$ = new BinaryExpr("+",
                            std::unique_ptr<Expr>($1),
                            std::unique_ptr<Expr>($3));
    }
  | expression MINUS expression {
        $$ = new BinaryExpr("-",
                            std::unique_ptr<Expr>($1),
                            std::unique_ptr<Expr>($3));
    }
  | expression MULTIPLY expression {
        $$ = new BinaryExpr("*",
                            std::unique_ptr<Expr>($1),
                            std::unique_ptr<Expr>($3));
    }
  | expression DIVIDE expression {
        $$ = new BinaryExpr("/",
                            std::unique_ptr<Expr>($1),
                            std::unique_ptr<Expr>($3));
    }
  | LPAREN expression RPAREN {
        $$ = $2;
    }
  ;

%%
```

9.9 AST Visualizer

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

Listing 9.9: AST Text Visualizer

```
// ASTVisualizer.hpp
#ifndef ASTVISIZER_HPP
#define ASTVISIZER_HPP

#include "ASTVisitor.hpp"
#include <iostream>
```

```
#include <string>
#include <vector>

class ASTVisualizer : public ASTVisitor {
private:
    std::vector<bool> isLastChild;

    void printPrefix() {
        for (size_t i = 0; i < isLastChild.size() - 1; i++) {
            if (isLastChild[i]) {
                std::cout << "    ";
            } else {
                std::cout << "|   ";
            }
        }
        if (!isLastChild.empty()) {
            std::cout << (isLastChild.back() ? "+-- " : "+-- ");
        }
    }

public:
    void visit(BinaryExpr& node) override {
        printPrefix();
        std::cout << "BinaryExpr: " << node.op << std::endl;

        isLastChild.push_back(false);
        if (node.left) node.left->accept(*this);
        isLastChild.back() = true;
        if (node.right) node.right->accept(*this);
        isLastChild.pop_back();
    }

    void visit(IntLiteral& node) override {
        printPrefix();
        std::cout << "IntLiteral: " << node.value << std::endl;
    }

    void visit(Identifier& node) override {
        printPrefix();
        std::cout << "Identifier: " << node.name << std::endl;
    }

    // Implement other visit methods...
};

#endif
```

9.10 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

9.11 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. AST proyek subset C (Bagian 9.3) dipakai oleh symbol table (Bab 10), type checking (Bab 11), dan IR generation (Bab 12) dalam pipeline compiler proyek.

9.12 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* [2] - Bab 2: A Simple Syntax-Directed Translator, Bab 5: Syntax-Directed Translation
- **Engineering a Compiler**: Cooper & Torczon (2011) [3] - Bab 4: Intermediate Representations
- **UC San Diego CSE 231**: Course materials tentang AST construction [4]
- **Johns Hopkins University EN.601.428**: Course tentang AST dan syntax trees [5]
- **GNU Bison Manual**: Dokumentasi tentang semantic actions dan AST building [6]

Daftar Pustaka

[1] Diznr. *Six Phases of Compiler*. Online educational resource. 2024. URL: https://diznr.com/six-phases-of-compiler-lexical-syntax-semantic-intermediate-code-generation-optimization-code/.

[2] Alfred V. Aho **andothers**. *Compilers: Principles, Techniques, and Tools*. 2nd. Dragon Book. Pearson Education, 2006.

[3] Keith D. Cooper **and** Linda Torczon. *Engineering a Compiler*. 2nd. Morgan Kaufmann, 2011.

[4] UC San Diego CSE Department. *CSE 231: Compiler Construction / Advanced Compiler Design*. Course materials and syllabus. 2024. URL: https://catalog.ucsd.edu/archive/2024-25/courses/CSE.html.

[5] Johns Hopkins University. *EN.601.428/628: Compilers and Interpreters*. Course syllabus and materials, taught by David Hovemeyer. 2024. URL: https://jhucompilers.github.io/fall2025/syllabus.html.

[6] GNU Project. *GNU Bison Manual*. Official Bison documentation. 2024. URL: https://www.gnu.org/software/bison/manual/.