A Case Study of “Gang of Four” (GoF) Patterns: Part 1

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

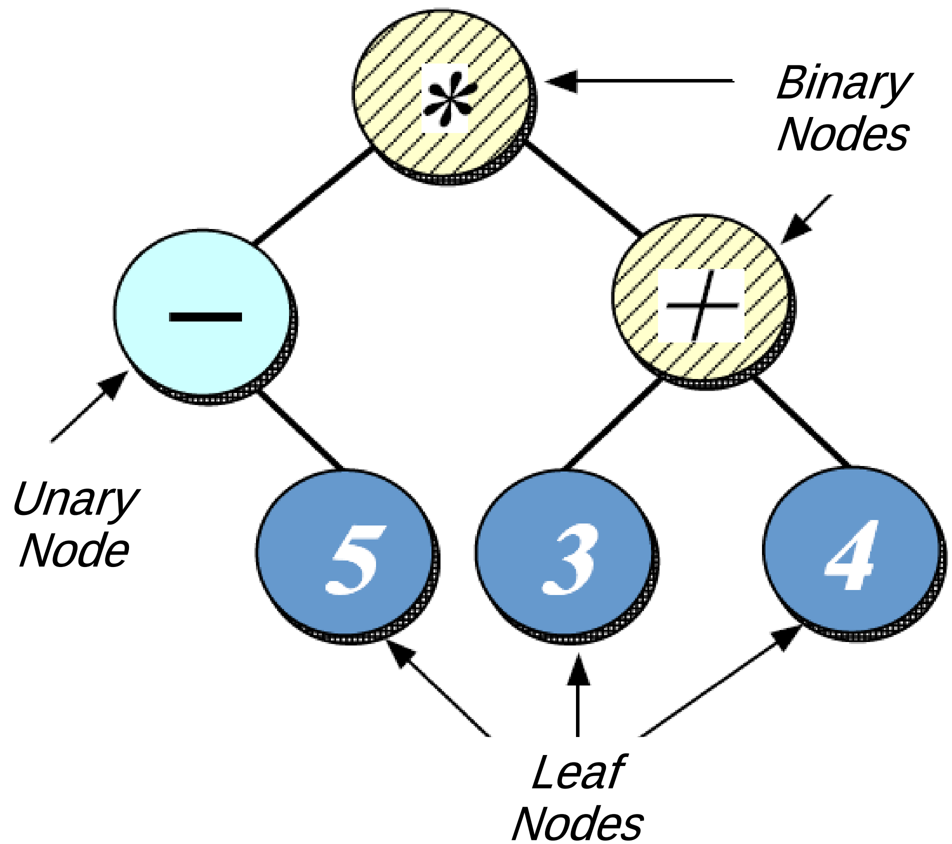
Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

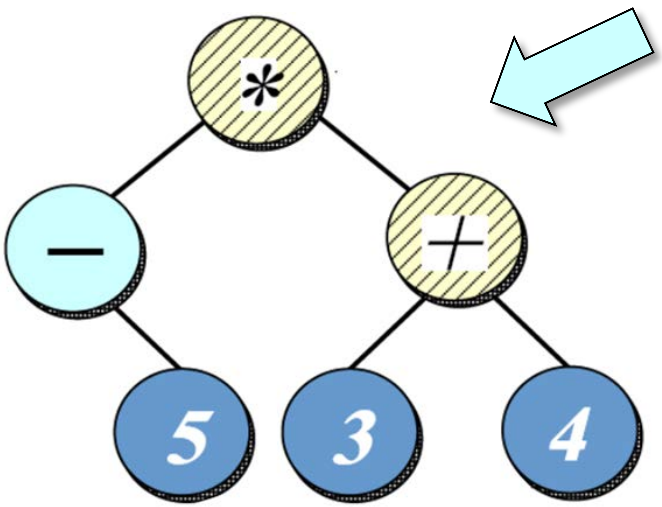
- Describe the object-oriented (OO) expression tree case study



Case Study: Expression Tree Processing App

Goals

- Develop an OO expression tree processing app using *patterns & frameworks*



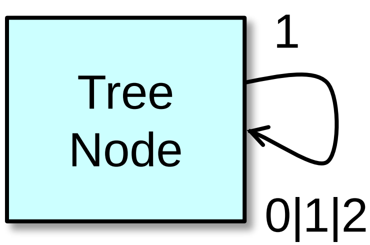
Design Problem	Pattern(s)
Extensible expression tree structure	Composite
Encapsulating variability & simplifying memory management	Bridge
Parsing expressions & creating expression tree	Interpreter & Builder
Extensible expression tree operations	Iterator & Visitor
Implementing STL iterator semantics	Prototype
Consolidating user operations	Command
Consolidating creation of variabilities for commands, iterators, etc.	Abstract Factory & Factory Method
Ensuring correct protocol for commands	State
Structuring application event flow	Reactor
Supporting multiple operation modes	Template Method & Strategy
Centralizing access to global resources	Singleton
Eliminating loops via the STL **std::for_each()** algorithm	Adapter

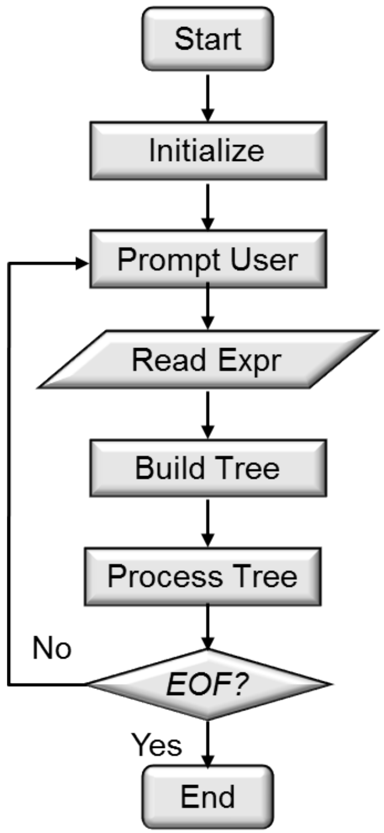
Expression trees are used to remove ambiguity in algebraic expressions

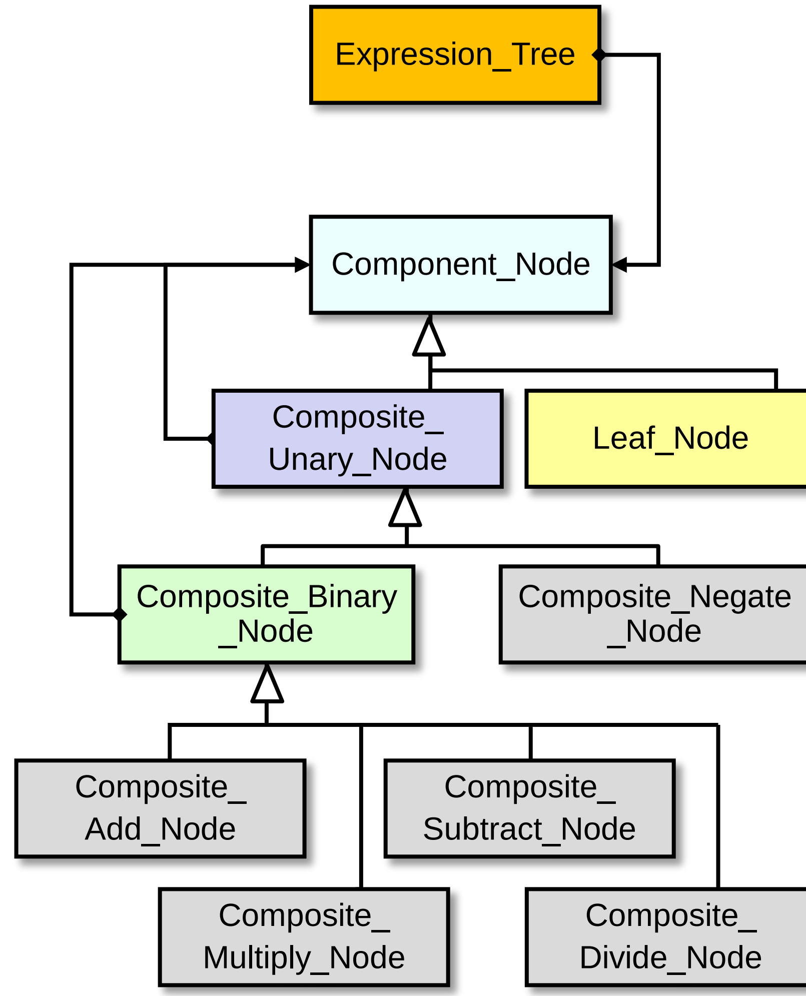
Case Study: Expression Tree Processing App

Goals

- Develop an OO expression tree processing app using *patterns & frameworks*
- Compare/contrast non-object-oriented & object-oriented approaches





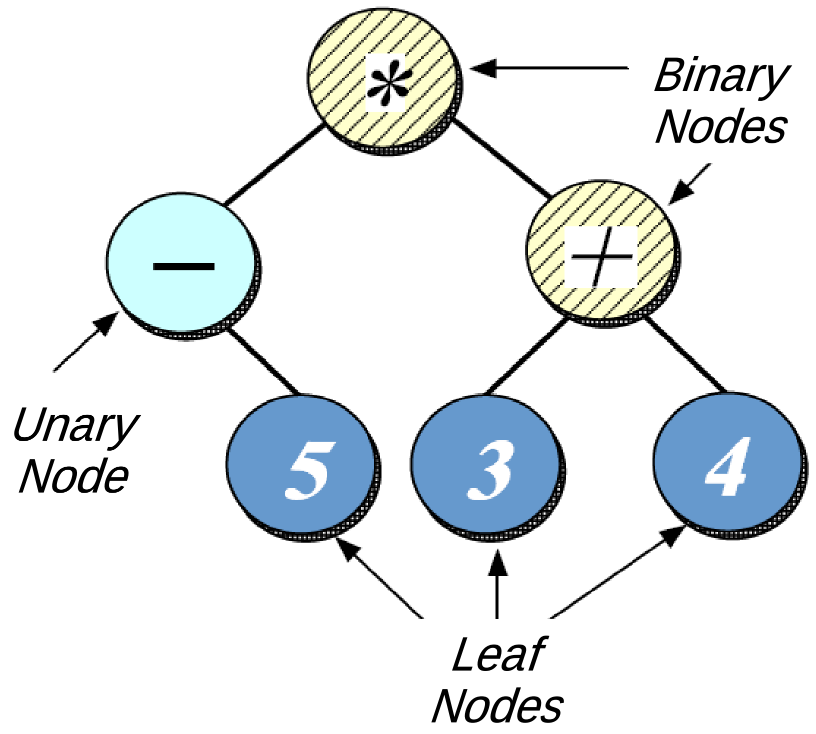


Despite decades of OO emphasis, algorithmic decomposition is still common

Case Study: Expression Tree Processing App

Goals

- Develop an OO expression tree processing app using *patterns & frameworks*
- Compare/contrast non-object-oriented & object-oriented approaches
- Demonstrate *Scope, Commonality, & Variability* (SCV) analysis in the context of a concrete example
  - SCV is a systematic software reuse method



www.cs.iastate.edu/~cs309/references/CoplienHoffmanWeiss_CommonalityVariability.pdf

Case Study: Expression Tree Processing App

Goals

- Develop an OO expression tree processing app using *patterns & frameworks*
- Compare/contrast non-object-oriented & object-oriented approaches
- Demonstrate *Scope, Commonality, & Variability* (SCV) analysis in the context of a concrete example
- Illustrate how pattern-oriented OO frameworks can be implemented in C++ & Java

```
Expression_Tree expr_tree = …;
Print_Visitor print_visitor;
```

C++11 range-based for loop

```
for (auto &iter : expr_tree)
   iter.accept(print_visitor);
```

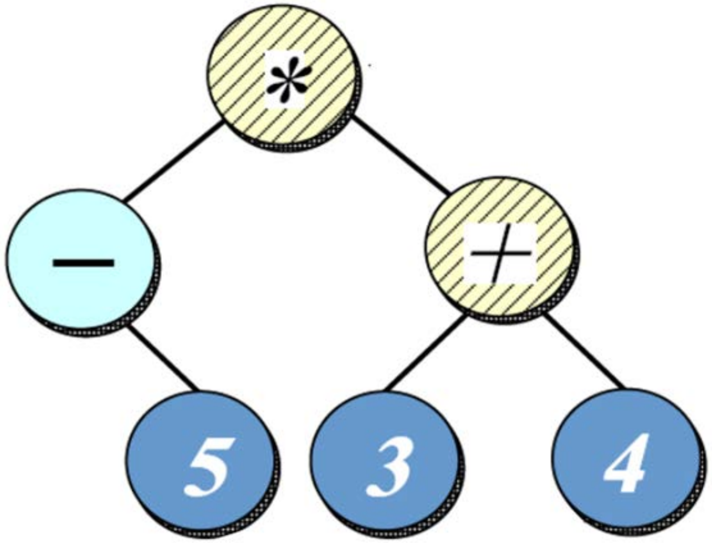
```
ExpressionTree exprTree = …;
ETVisitor printVisitor =
                 new PrintVisitor();
```

Java for-each loop

```
for (ComponentNode node : exprTree)
    node.accept(printVisitor);
```

Overview of Expression Tree Processing App

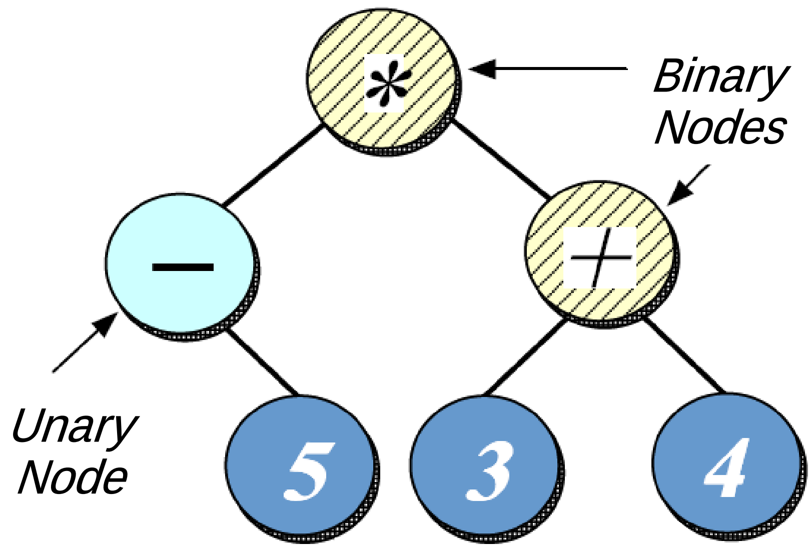
- Expression trees consist of nodes containing *operators* & *operands*



See en.wikipedia.org/wiki/Binary_expression_tree for expression tree info

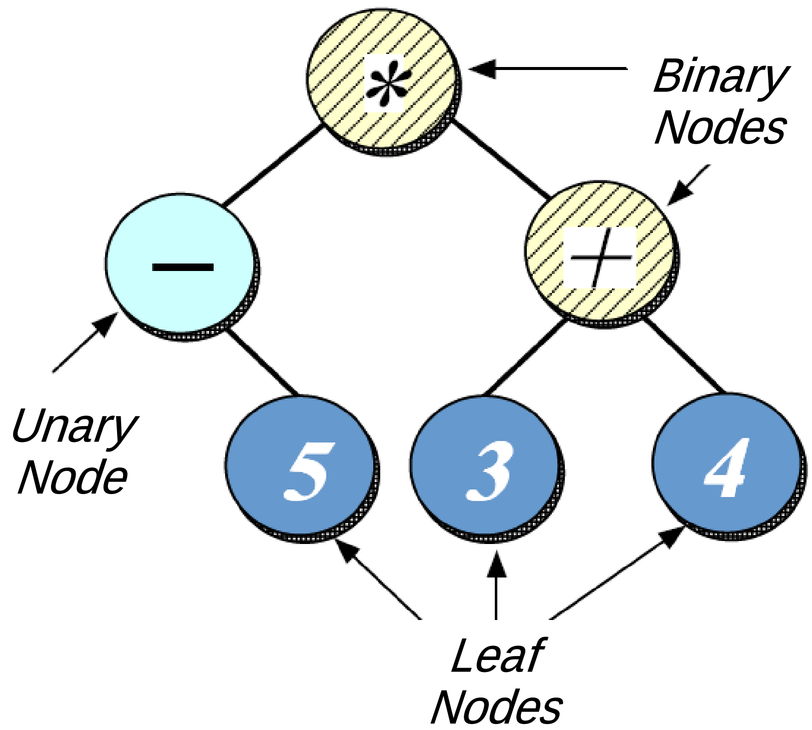
Overview of Expression Tree Processing App

- Expression trees consist of nodes containing *operators* & *operands*
  - Operators are *interior nodes* in the tree
    - i.e., *binary* & *unary nodes*



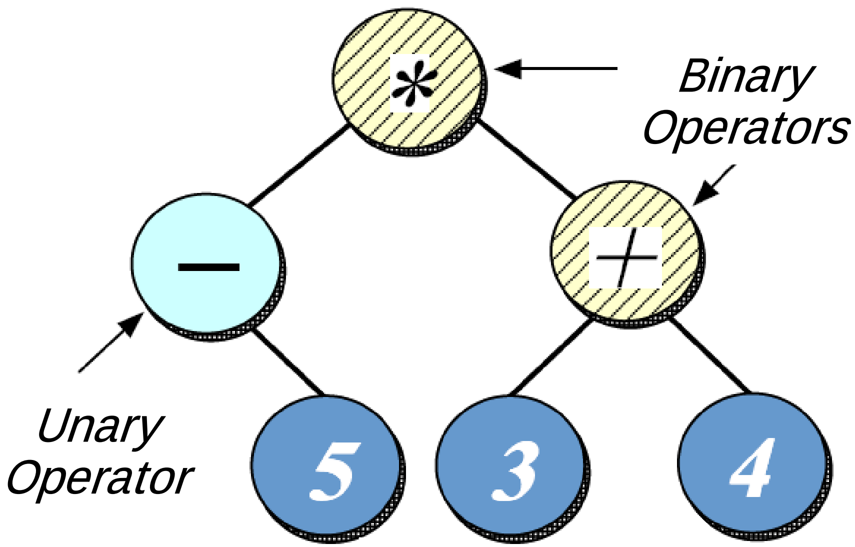
Overview of Expression Tree Processing App

- Expression trees consist of nodes containing *operators* & *operands*
  - Operators are *interior nodes* in the tree
    - i.e., *binary* & *unary nodes*
  - Operands are *exterior nodes* in the tree
    - i.e., *leaf nodes*



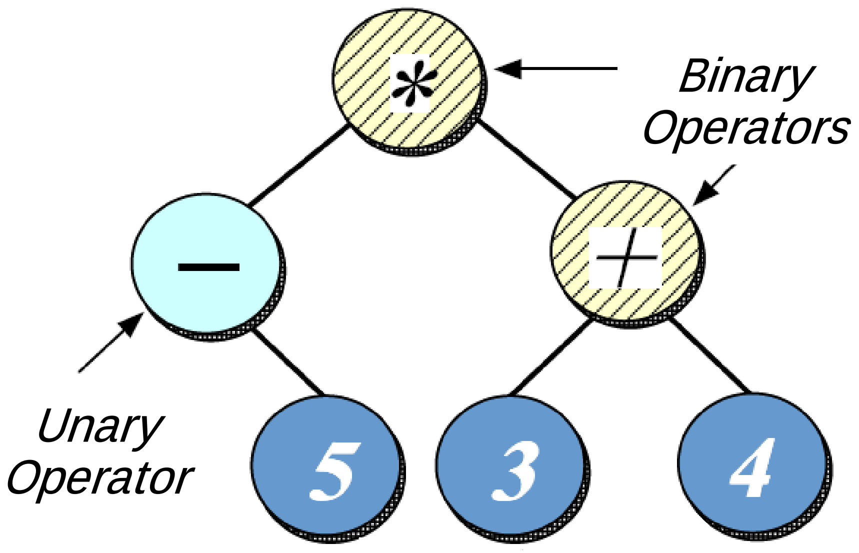
Overview of Expression Tree Processing App

- Expression trees consist of nodes containing *operators* & *operands*
- Operators have different precedence levels, different associativities, & different arities, e.g.:



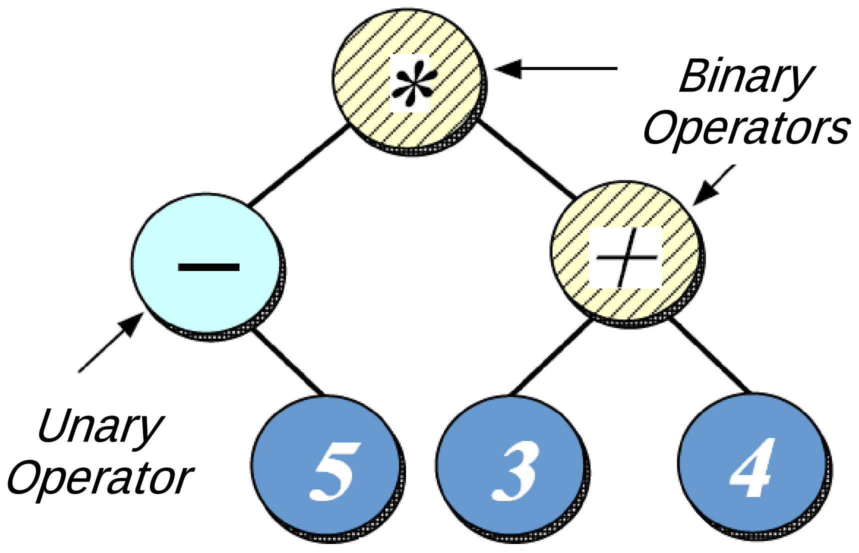
Overview of Expression Tree Processing App

- Expression trees consist of nodes containing *operators* & *operands*
- Operators have different precedence levels, different associativities, & different arities, e.g.:
  - The multiplication operator has two arguments, whereas unary minus operator has only one



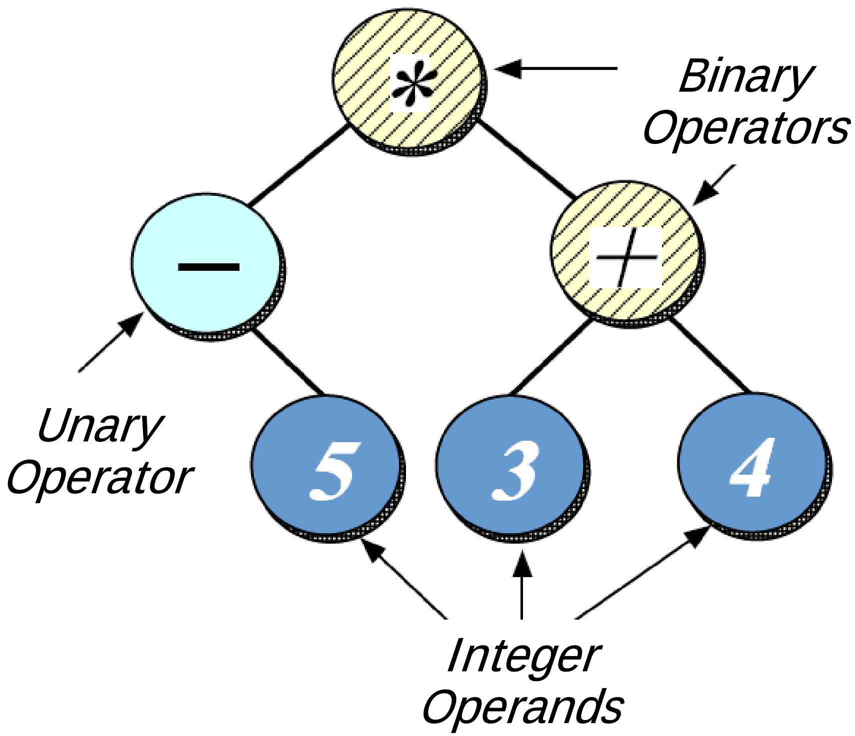
Overview of Expression Tree Processing App

- Expression trees consist of nodes containing *operators* & *operands*
- Operators have different precedence levels, different associativities, & different arities, e.g.:
  - The multiplication operator has two arguments, whereas unary minus operator has only one
  - Operator locations in the tree unambiguously designate precedence



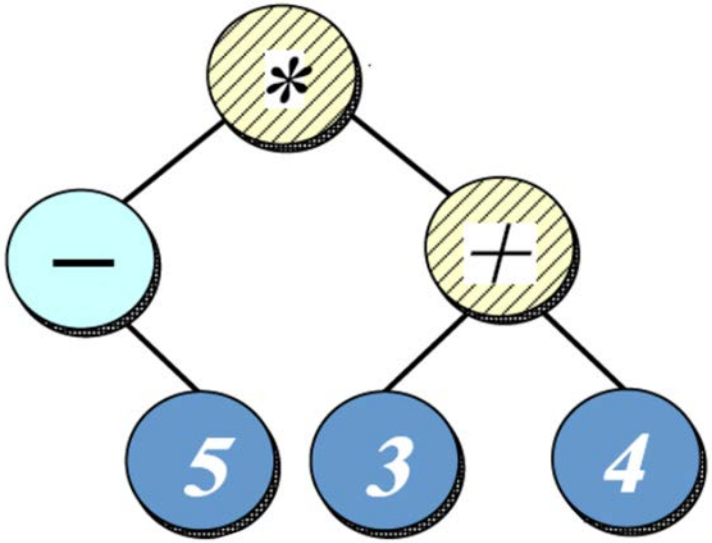
Overview of Expression Tree Processing App

- Expression trees consist of nodes containing *operators* & *operands*
- Operators have different precedence levels, different associativities, & different arities
- Operands can be integers, doubles, variables, etc.
  - We'll just handle integers in this example, though it can easily be extended



Overview of Expression Tree Processing App

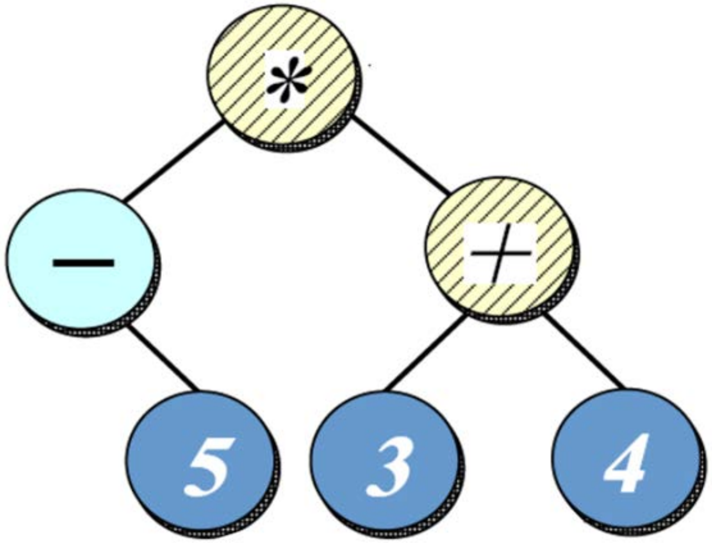
- Trees may be "evaluated" via different traversal orders, e.g.,
  - "in-order iterator" = **-5*(3+4)**
  - "pre-order iterator" = ***-5+34**
  - "post-order iterator" = **5-34+***
  - "level-order iterator" = ***-+534**



See en.wikipedia.org/wiki/Binary_expression_tree#Traversal for more info

Overview of Expression Tree Processing App

- Trees may be "evaluated" via different traversal orders, e.g.,
  - "in-order iterator" = **`-5*(3+4)`**
  - "pre-order iterator" = **`*-5+34`**
  - "post-order iterator" = **`5-34+*`**
  - "level-order iterator" = **`*-+534`**
- The evaluation step may perform various actions, e.g.:
  - Print contents of expression tree
  - Return the "value" of the expression tree
  - Perform semantic analysis & optimization
  - Generate code
  - etc.



1. S = [5] **`push(node.item())`**
2. S = [-5] **`push(-pop())`**
3. S = [-5, 3] **`push(node.item())`**
4. S = [-5, 3, 4] **`push(node.item())`**
5. S = [-5, 7] **`push(pop()+pop())`**
6. S = [-35] **`push(pop()*pop())`**

Summary

- The expression tree processing app can be run in multiple modes, e.g.:
  - “Succinct mode”

```
% tree-traversal
> 1+4*3/2
7
> (8/4) * 3 + 1
7
^D
```

- “Verbose mode”

```
% tree-traversal -v
format [in-order]
expr [expression]
print [in-order|pre-order|post-
       order|level-order]
eval [post-order]
quit
> format in-order
> expr 1+4*3/2
> print post-order
143*2/+
> eval post-order
7
> quit
```

A Case Study of "Gang of Four" (GoF) Patterns: Part 2

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

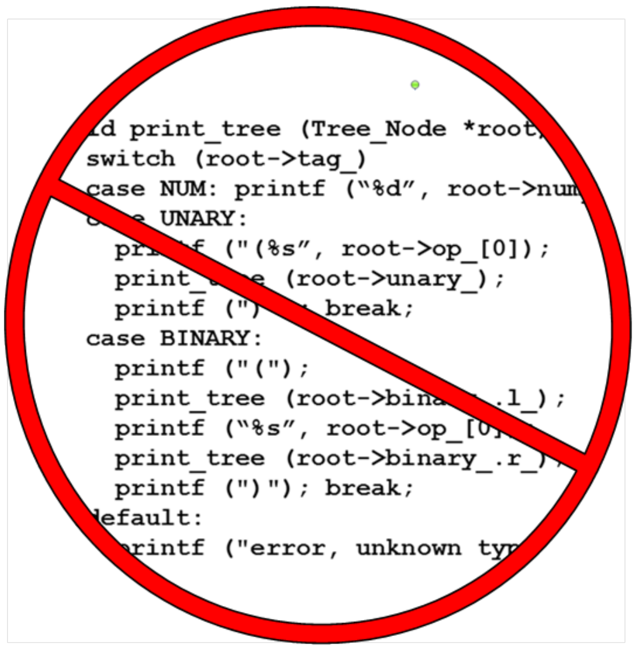
Professor of Computer Science

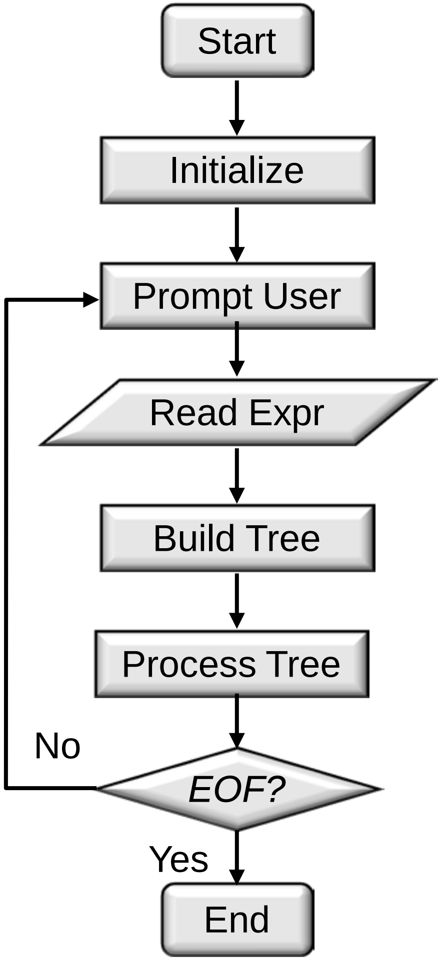
Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Topics Covered in this Part of the Module

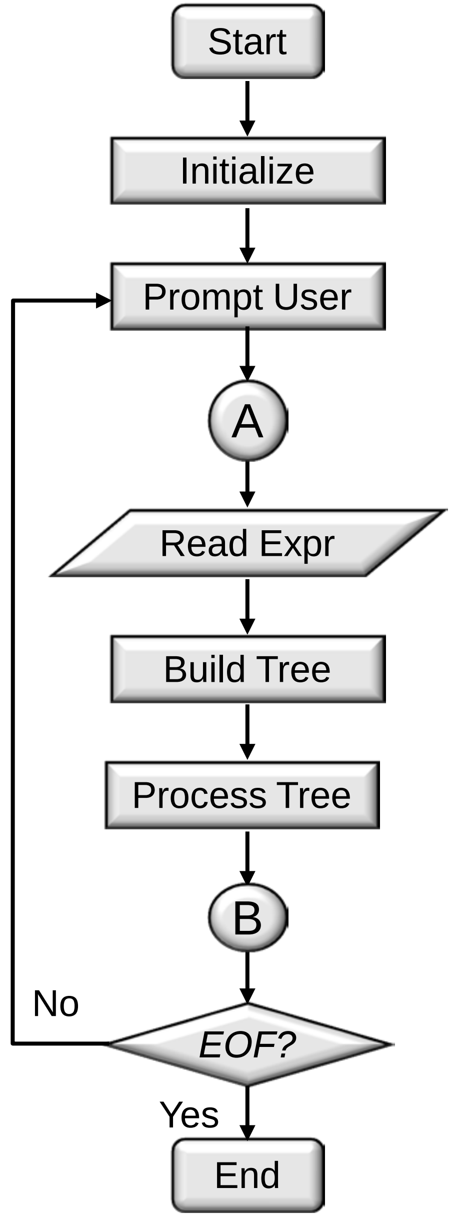
- Describe the object-oriented (OO) expression tree case study
- Evaluate the limitations with algorithmic design techniques





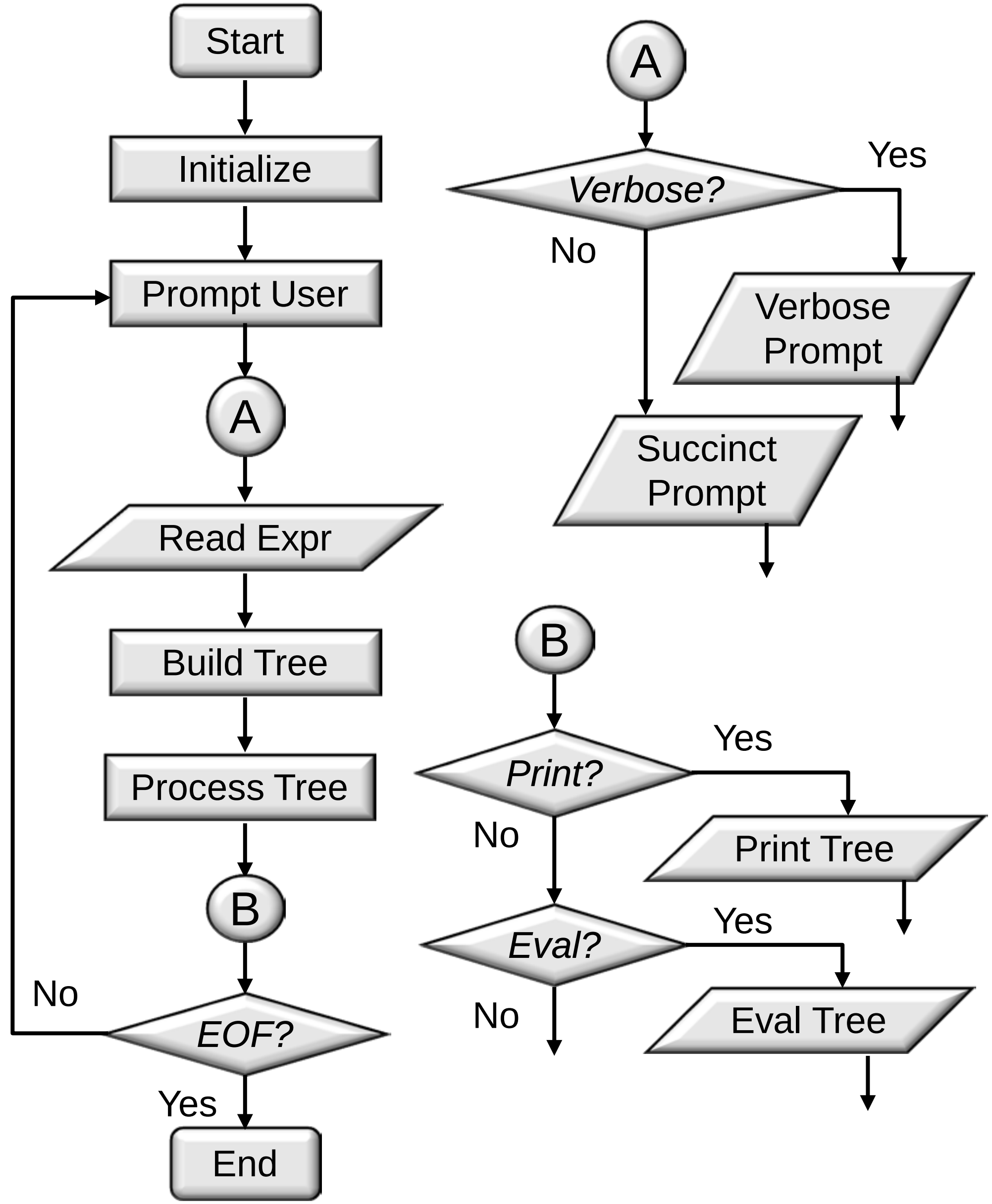
How *Not* to Design an Expression Tree Application

- Apply *algorithmic decomposition*
  - Top-down design based on the actions performed by the system



How *Not* to Design an Expression Tree Application

- Apply *algorithmic decomposition*
  - Top-down design based on the actions performed by the system
  - Generally follows a "divide & conquer" strategy based on the actions
    - i.e., general actions are iteratively/recursively decomposed into more specific ones



How *Not* to Design an Expression Tree Application

- Apply *algorithmic decomposition*
  - Top-down design based on the actions performed by the system
  - Generally follows a "divide & conquer" strategy based on the actions
    - i.e., general actions are iteratively/recursively decomposed into more specific ones
  - Primary design components correspond to processing steps in execution sequence
    - e.g., C functions

```
typedef struct Tree_Node {
   ...
} Tree_Node;
```

We'll explore this shortly

```
void prompt_user(int verbose);
char *read_expr(FILE *fp);
Tree_Node *build_tree
              (const char *expr);
void process_tree
     (Tree_Node *root, FILE *fp);
void eval_tree
     (Tree_Node *root, FILE *fp);
void print_tree
     (Tree_Node *root, FILE *fp);
...
```

We'll explore this shortly

Algorithmic Decomposition of Expression Tree

- A typical algorithmic decomposition for implementing expression trees would use a C struct/union to represent the main data structure

```
typedef struct Tree_Node {
  enum { NUM, UNARY, BINARY } tag_;
  short use_;
  union {
    char op_[2];
    int num_;
  } o;
#define num_ o.num_
#define op_  o.op_
  union {
    struct Tree_Node *unary_;
    struct { struct Tree_Node *l_,
             *r_;} binary_;
  } c;
#define unary_ c.unary_
#define binary_ c.binary_
} Tree_Node;
```

Type tag

Reference count

Node value

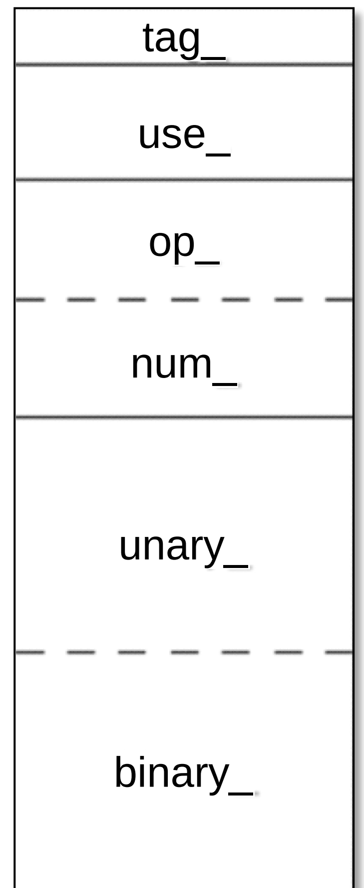
Node child(ren)

Algorithmic Decomposition of Expression Tree

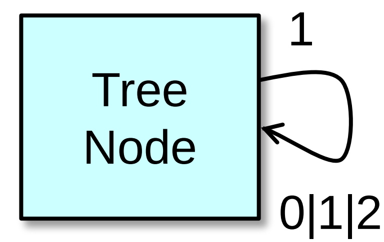
- A typical algorithmic decomposition for implementing expression trees would use a C struct/union to represent the main data structure

```
typedef struct Tree_Node {
  enum { NUM, UNARY, BINARY } tag_;
  short use_;
  union {
    char op_[2];
    int num_;
  } o;
#define num_ o.num_
#define op_  o.op_
  union {
    struct Tree_Node *unary_;
    struct { struct Tree_Node *l_,
                      *r_;} binary_;
  } c;
#define unary_ c.unary_
#define binary_ c.binary_
} Tree_Node;
```

Memory Layout



“Class” Relationships



Algorithmic Decomposition of Expression Tree

- A typical algorithmic decomposition uses a switch statement & a recursive function to build & evaluate a tree, e.g.:

```
void print_tree(Tree_Node *root, FILE *fp) {
  switch(root->tag_) {
  case NUM: fprintf(fp, "%d", root->num_); break;
  case UNARY:
    fprintf(fp, "(%s", root->op_[0]);
    print_tree(root->unary_, fp);
    fprintf(fp, ")"); break;
  case BINARY:
    fprintf(fp, "(");
    print_tree(root->binary_.l_, fp);
    fprintf(fp, "%s", root->op_[0]);
    print_tree(root->binary_.r_, fp);
    fprintf(fp, ")"); break;
  ...
}
```

Switch on type tag

Algorithmic Decomposition of Expression Tree

- A typical algorithmic decomposition uses a switch statement & a recursive function to build & evaluate a tree, e.g.:

```
void print_tree(Tree_Node *root, FILE *fp) {
  switch(root->tag_) {
  case NUM: fprintf(fp, "%d", root->num_); break;
  case UNARY:
    fprintf(fp, "(%s", root->op_[0]);
    print_tree(root->unary_, fp);
    fprintf(fp, ")"); break;
  case BINARY:
    fprintf(fp, "(");
    print_tree(root->binary_.l_, fp);
    fprintf(fp, "%s", root->op_[0]);
    print_tree(root->binary_.r_, fp);
    fprintf(fp, ")"); break;
  ...
}
```

Algorithmic Decomposition of Expression Tree

- A typical algorithmic decomposition uses a switch statement & a recursive function to build & evaluate a tree, e.g.:

```
void print_tree(Tree_Node *root, FILE *fp) {
  switch(root->tag_) {
  case NUM: fprintf(fp, "%d", root->num_); break;
  case UNARY:
    fprintf(fp, "(%s", root->op_[0]);
    print_tree(root->unary_, fp);          Recursive call
    fprintf(fp, ")"); break;
  case BINARY:
    fprintf(fp, "(");
    print_tree(root->binary_.l_, fp);
    fprintf(fp, "%s", root->op_[0]);
    print_tree(root->binary_.r_, fp);
    fprintf(fp, ")"); break;
  ...
}
```

Algorithmic Decomposition of Expression Tree

- A typical algorithmic decomposition uses a switch statement & a recursive function to build & evaluate a tree, e.g.:

```
void print_tree(Tree_Node *root, FILE *fp) {
  switch(root->tag_) {
  case NUM: fprintf(fp, "%d", root->num_); break;
  case UNARY:
    fprintf(fp, "(%s", root->op_[0]);
    print_tree(root->unary_, fp);
    fprintf(fp, ")"); break;
  case BINARY:
    fprintf(fp, "(");
    print_tree(root->binary_.l_, fp);      Recursive call
    fprintf(fp, "%s", root->op_[0]);
    print_tree(root->binary_.r_, fp);      Recursive call
    fprintf(fp, ")"); break;
  ...
}
```

Summary

- Limitations with algorithmic decomposition
  - Little/no encapsulation

```
typedef struct Tree_Node {
  enum { NUM, UNARY, BINARY } tag_;
  short use_;
  union {
    char op_[2];
    int num_;
  } o;
#define num_ o.num_
#define op_  o.op_
  union {
    struct Tree_Node *unary_;
    struct { struct Tree_Node *l_,
                   *r_;} binary_;
  } c;
#define unary_ c.unary_
#define binary_ c.binary_
} Tree_Node;
```

Implementation details available to clients

Small changes ripple through entire program

Use of macros pollutes global namespace

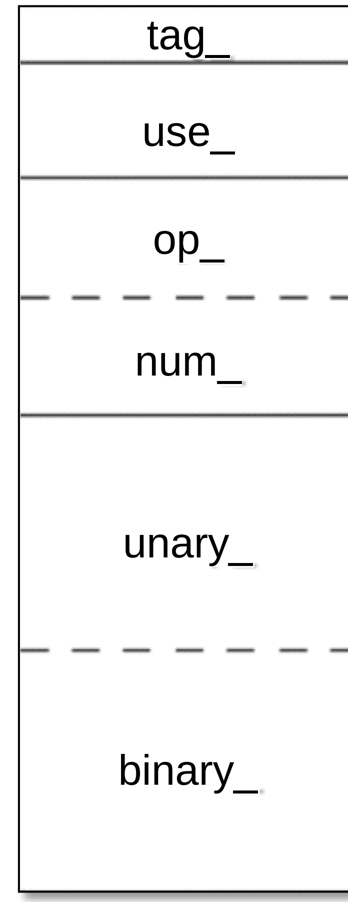
Summary

- Limitations with algorithmic decomposition
  - Incomplete modeling of application domain

```
typedef struct Tree_Node {
  enum { NUM, UNARY, BINARY } tag_;
  short use_;
  union {
    char op_[2];
    int num_;
  } o;
#define num_ o.num_
#define op_  o.op_
  union {
    struct Tree_Node *unary_;
    struct { struct Tree_Node *l_,
             *r_;} binary_;
  } c;
#define unary_ c.unary_
#define binary_ c.binary_
} Tree_Node;
```

Tight coupling between nodes/edges in union

Wastes space by making worst-case assumptions wrt structs & unions



Summary

- Limitations with algorithmic decomposition
  - Complexity in (variable) algorithms rather than (stable) structure

Tree_Node data structure is "passive" & functions do all the real work

```
void print_tree(Tree_Node *root, FILE *fp) {
  switch(root->tag_) {
  case NUM: fprintf(fp, "%d", root->num_); break;
  case UNARY:
    fprintf(fp, "(%s", root->op_[0]);
    print_tree(root->unary_, fp);
    fprintf(fp, ")"); break;
  case BINARY:
    fprintf(fp, "(");
    print_tree(root->binary_.l_, fp);
    fprintf(fp, "%s", root->op_[0]);
    print_tree(root->binary_.r_, fp);
    fprintf(fp, ")"); break;
  ...
}
```

Easy to make mistakes when switching on type tags

Tailoring the app for specific requirements & specifications impedes reuse & complicates software sustainment

Overcoming limitations requires rethinking modeling, design, & implementation