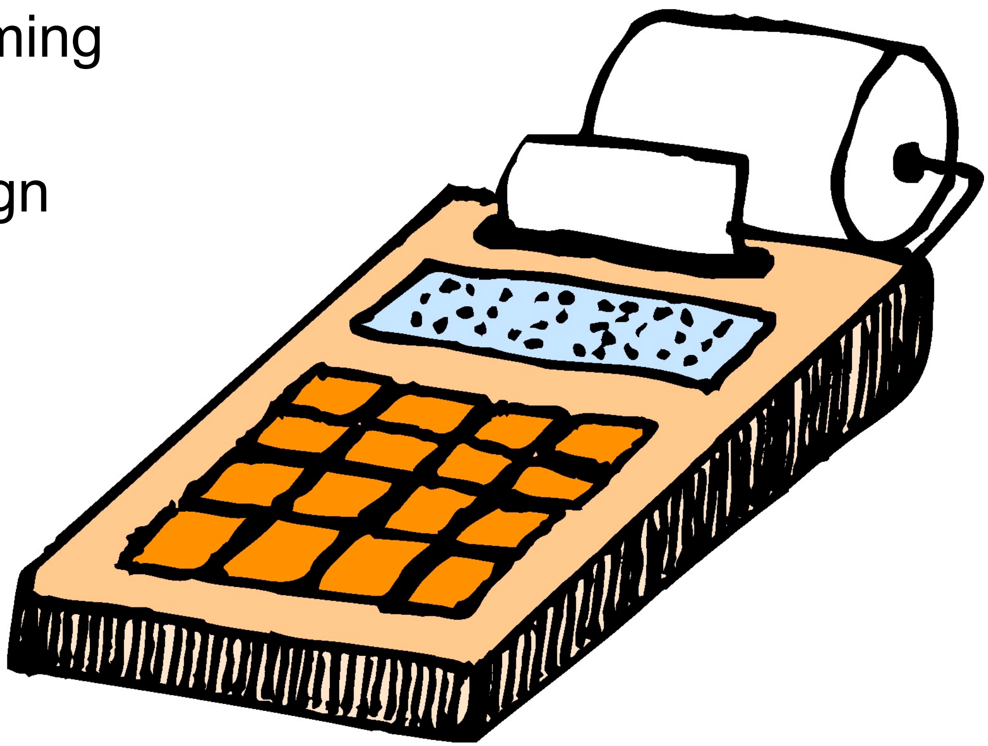
Case Study – Algebraic Expression Evaluator

Goals

- Develop an object-oriented basic algebraic expression evalutor using *patterns* & *frameworks*
- Demonstrate commonality/variability analysis in the context of a concrete application example
- Illustrate how OO frameworks can be combined with the generic programming features of C++ & STL
- Compare/contrast different OO design approaches

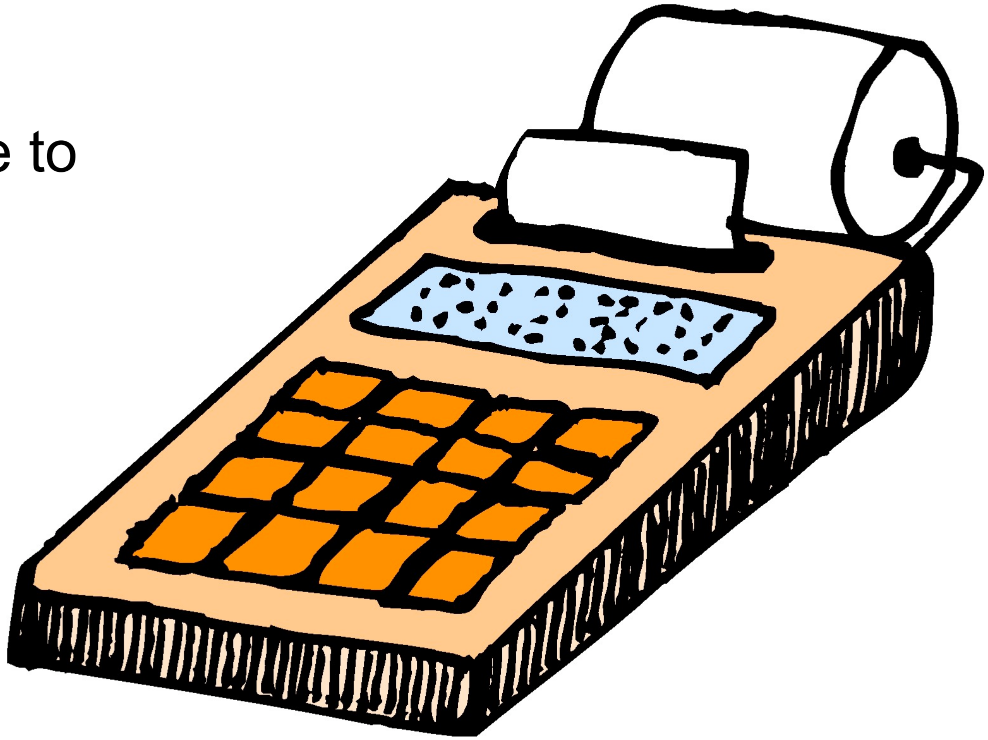


Case Study – Algebraic Expression Evaluator

Overview of the expression evaluator

- The evaluator is designed to handle basic algebraic expressions:
  - 2 + c
  - 6x – x / 8y * 9
  - (6 % 2) + 8abc / 78 – (923 + 8)

- The expression should be extensible to handle new math operators
  - Square root
  - Powers
  - etc.

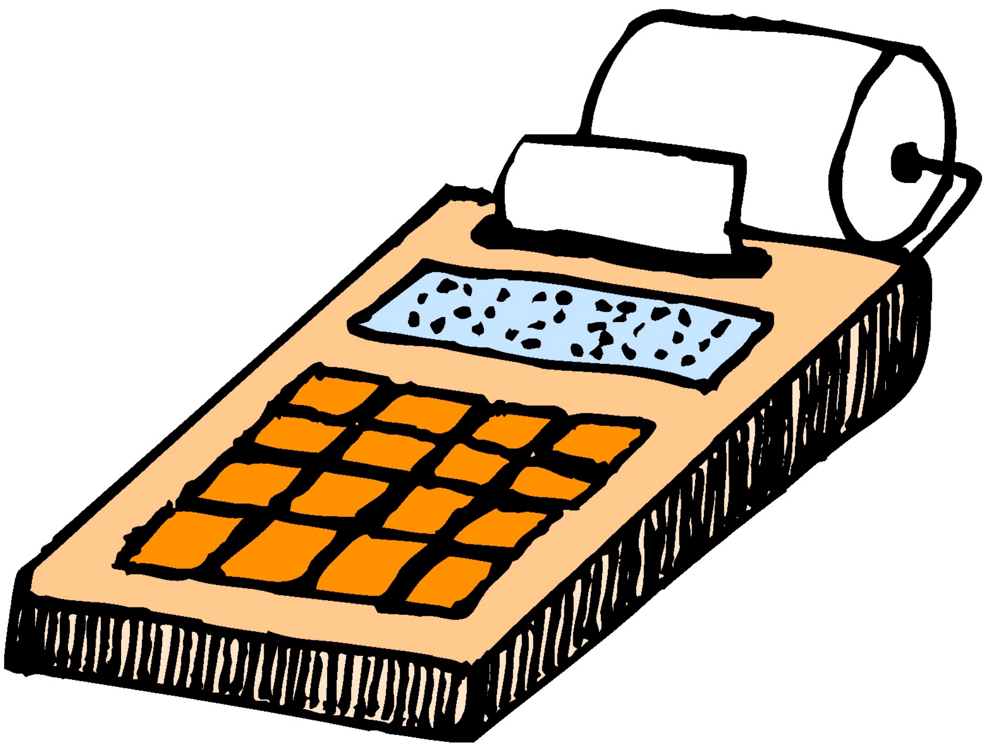


Case Study – Algebraic Expression Evaluator

Understanding Expressions

- It is hard for a computer to evaluate an expression in Infix format:
    - 2 + c
    - 6x – x / 8y * 9
    - (6 % 2) + 8abc / 78 – (923 + 8)

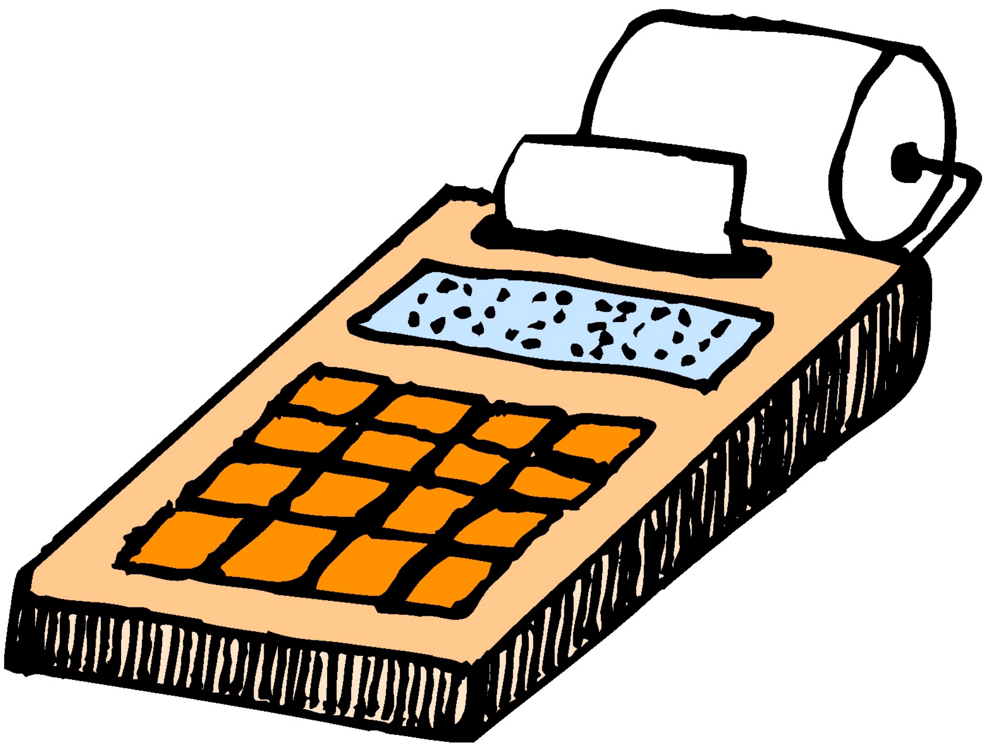
- Expressions are composed from:
    - Operands: 6, 7, & 678
    - Operators: +, -, %



Case Study – Algebraic Expression Evaluator

Understanding Expressions

- Instead, it is easier for a computer to evaluate an expression if it in postfix format:
  - 2 c +
  - ? ? ?
  - ? ? ?



Infix to Postfix Conversion Example

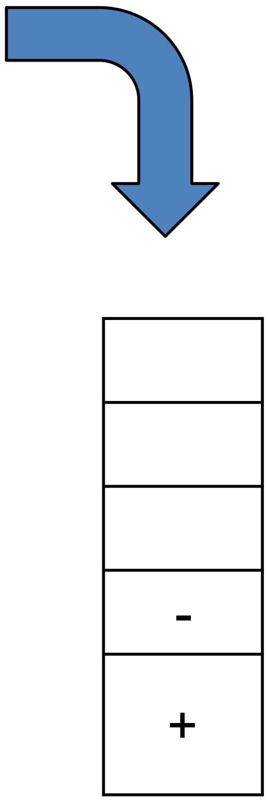
Algorithm for Converting Infix to Postfix

A stack can be used to perform the conversion as follows when parsing each element *e* in the expression:

1. If *e* is operand, then append to end of postfix expression
2. If *e* is operator, & (stack is empty or *e* of lower precedence than top of stack), then push *e* on stack.

    Else pop from stack & append to postfix until *e* has greater precedence than top of stack or empty, then push *e* on stack
3. If *e* is a parenthesis, pop elements from stack and append until found matching open parenthesis

2 + 7 – 8 + 8



C Example

A typical algorithmic-based solution for implementing the postfix expression is to use a C struct/union to represent the main data structure

```
typedef struct Expr_Node {
  /* type information for the node */
  enum { NUM, OPERATOR} tag_;

  /* either a operator, or a number */
  union {
    char op_;
    int num_;
  } o;
#define num_ o.num_
#define op_  o.op_
} Expr_Node;
```

C Example

- A typical algorithmic implementation uses a switch statement & a for loop to evaluate the postfix expression, e.g., 2 7 – 8 + 8 + :

```
void evaluate_postfix (Expr_Node * expr, size_t length) {
  Eval_Stack stack;

  for (size_t i = 0; i < length; ++ i) {
  switch (expr[i].tag_)
  case OPERATOR:
    int n1 = pop (stack), n2 = pop (stack);
    switch (expr[i].op_) {
    case '+': push (stack, n2 + n1); break;
    case '-': push (stack, n2 - n1); break;
    case '*': push (stack, n2 * n1); break;
    case '/': push (stack, n2 / n1); break;
    }
  case NUM:
    push (stack, expr[i].num_); break;
  default:
    printf ("error, unknown type ");
  }

  return top (stack);
}
```

Limitations of Algorithmic Approach

- Little or no use of encapsulation: implementation details available to clients
- Incomplete modeling of the application domain, which results in
  - Tight coupling between nodes/ edges in union representation
  - Complexity being in algorithms rather than the data structures, e.g., switch statements are used to select between various types of nodes in the expression trees
- Data structures are "passive" functions that do their work explicitly
- The program organization makes it hard to extend
  - e.g., Any small changes will ripple through entire design/ implementation
- Easy to make mistakes switching on type tags
- Wastes space by making worst-case assumptions with respect to structs & unions