# Chapter 13

# CASE STUDY: THE POLISH NOTATION

1. The Problem
2. The Idea
   (a) Expression Trees
   (b) Polish Notation
3. Evaluation of Polish Expressions
4. Translation from Infix Form to Polish Form
5. Application: An Interactive Expression Evaluator

# Priorities of Operators

## The quadratic formula:

$$x = (-b + (b \uparrow 2 - (4 \times a) \times c) \uparrow \tfrac{1}{2})/(2 \times a)$$

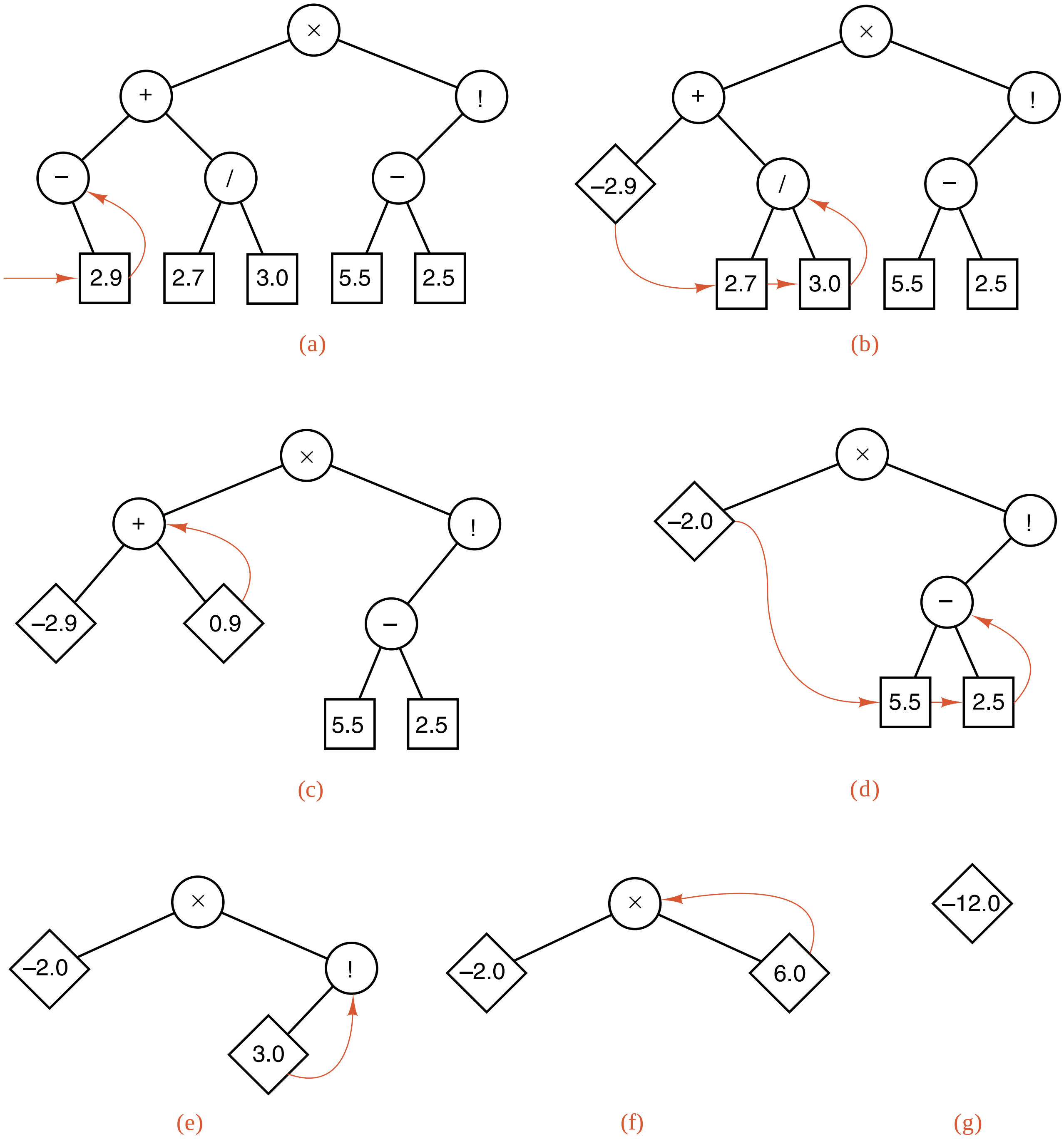
## Order of doing operators:

$$\begin{array}{ccccccccccccccccccccc}
x & = & ( & - & b & + & (b & \uparrow & 2 & - & (4 & \times & a) & \times & c) & \uparrow & \tfrac{1}{2}) & / & (2 & \times & a) \\
 & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & & \uparrow & \\
 & 10 & & 1 & & 7 & & 2 & & 5 & & 3 & & 4 & & 6 & & 9 & & 8 &
\end{array}$$

## Priorities of operators:

| *Operators* | *Priority* |
|---|---|
| ↑, all unary operators | 6 |
| * / % | 5 |
| + − (binary) | 4 |
| == != < > <= >= | 3 |
| not | 2 |
| && \|\| | 1 |
| = | 0 |

# Evaluation of an Expression Tree



# Polish Notation

The Polish mathematician Jan Łukasiewicz made the observation that, by writing all operators either before their operands or after them, it is not necessary to keep track of priorities, of parentheses bracketing expressions, or to make repeated scans to evaluate an expression.

## Polish forms:

- When the operators are written before their operands, it is called the ***prefix form***.
- When the operators come after their operands, it is called the ***postfix form***, or, sometimes, ***reverse Polish form*** or ***suffix form***.
- The ***infix form*** means writing binary operators between their operands.

## Example, Quadratic Formula:

$$\begin{array}{ll} \textit{Prefix}: & = \ x \ / \ + \ \sim \ b \ \uparrow \ - \ \uparrow \ b \ 2 \ \times \ \times \ 4 \ a \ c \ \tfrac{1}{2} \ \times \ 2 \ a \\ \textit{Postfix}: & x \ b \ \sim \ b \ 2 \ \uparrow \ 4 \ a \ \times \ c \ \times \ - \ \tfrac{1}{2} \ \uparrow \ + \ 2 \ a \ \times \ / \ = \end{array}$$

In these expressions, ‘$-$’ denotes binary subtraction and ‘$\sim$’ denotes unary negation.

# Evaluation of an Expression in Prefix Form

```
Error_code Expression :: evaluate_prefix(Value &result)
/* Outline of a method to perform prefix evaluation of an Expression. The details
   depend on further decisions about the implementation of expressions and values. */
{
   if (the Expression is empty) return fail;
   else {
      remove the first symbol from the Expression, and
         store the value of the symbol as t;
      if (t is a unary operation) {
        Value the_argument;
        if (evaluate_prefix(the_argument) == fail) return fail;
        else result = the value of operation t applied to the_argument;
      }
      else if (t is a binary operation) {
        Value first_argument, second_argument;
        if (evaluate_prefix(first_argument) == fail) return fail;
        if (evaluate_prefix(second_argument) == fail) return fail;
        result = the value of operation t
                         applied to first_argument and second_argument;
      }
      else                                   //   t is a numerical operand.
        result = the value of t;
   }
   return success;
}
```

# C++ Conventions and Class Declaration

```
class Expression {
public:
   Error_code evaluate_prefix(Value &result);
   Error_code get_token(Token &result);   //   Add other methods.
private:                                  //   Add data members to store an expression
};
```

- We define a ***token*** to be a single operator or operand from the expression. In our programs, we shall represent tokens as objects of a **struct** Token.
- We employ a method Token Expression :: get_token( ); that will move through an expression removing and returning one token at a time.
- We assume the existence of a Token method kind() that will return one of the values of the following enumerated type:

```
enum Token_type {operand, unaryop, binaryop};
                 //   Add any other legitimate token types.
```

- We assume that all the operands and the results of evaluating the operators are of the same type, which we call Value.
- We assume the existence of three auxiliary functions that return a result of type Value:

```
Value do_unary(const Token &operation, const Value &the_argument);
Value do_binary(const Token &operation,
                const Value &first_argument,
                const Value &second_argument);
Value get_value(const Token &operand);
```

# C++ Function for Prefix Evaluation

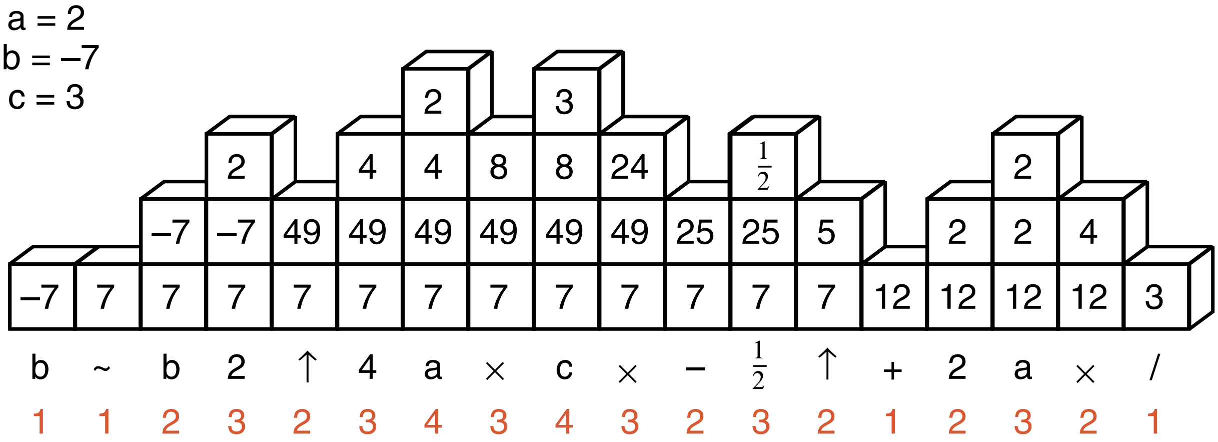
```
Error_code Expression :: evaluate_prefix(Value &result)
/* Post: If the Expression does not begin with a legal prefix expression, a code of fail
         is returned. Otherwise a code of success is returned, and the Expression
         is evaluated, giving the Value result. The initial tokens that are evaluated are
         removed from the Expression. */

{
   Token t;
   Value the_argument, first_argument, second_argument;
   if (get_token(t) == fail) return fail;

   switch (t.kind( )) {
   case unaryop:
      if (evaluate_prefix(the_argument) == fail) return fail;
      else result = do_unary(t, the_argument);
      break;

   case binaryop:
      if (evaluate_prefix(first_argument) == fail) return fail;
      if (evaluate_prefix(second_argument) == fail) return fail;
      else result = do_binary(t, first_argument, second_argument);
      break;

   case operand:
      result = get_value(t);
      break;
   }
   return success;
}
```

# Evaluation of Postfix Expressions

```
typedef Value Stack_entry;          //   Set the type of entry to use in stacks.
Error_code Expression::evaluate_postfix(Value &result)
/* Post: The tokens in Expression up to the first end_expression symbol are re-
         moved. If these tokens do not represent a legal postfix expression, a code of
         fail is returned. Otherwise a code of success is returned, and the removed
         sequence of tokens is evaluated to give Value result. */

{
   Token t;                         //   Current operator or operand
   Stack operands;                  //   Holds values until operators are seen
   Value the_argument, first_argument, second_argument;

   do {
      if (get_token(t) == fail) return fail;   //   No end_expression token
      switch (t.kind( )) {
      case unaryop:
         if (operands.empty( )) return fail;
         operands.top(the_argument);
         operands.pop( );
         operands.push(do_unary(t, the_argument));
         break;

      case binaryop:
         if (operands.empty( )) return fail;
         operands.top(second_argument);
         operands.pop( );
         if (operands.empty( )) return fail;
         operands.top(first_argument);
         operands.pop( );
         operands.push(do_binary(t, first_argument, second_argument));
         break;
      case operand:
         operands.push(get_value(t));
         break;

      case end_expression:
         break;
      }
   } while (t.kind( ) != end_expression);

   if (operands.empty( )) return fail;
   operands.top(result);
   operands.pop( );
   if (!operands.empty( )) return fail;   //   surplus operands detected
   return success;
}
```

# Proof of the Program: Counting Stack Entries

RUNNING SUM CONDITION. For a sequence $E$ of operands, unary operators, and binary operators, form a running sum by starting at the left end of $E$ and counting $+1$ for each operand, 0 for each unary operator, and $-1$ for each binary operator. $E$ satisfies the ***running-sum condition*** provided that this running sum never falls below 1 and is exactly 1 at the right-hand end of $E$.



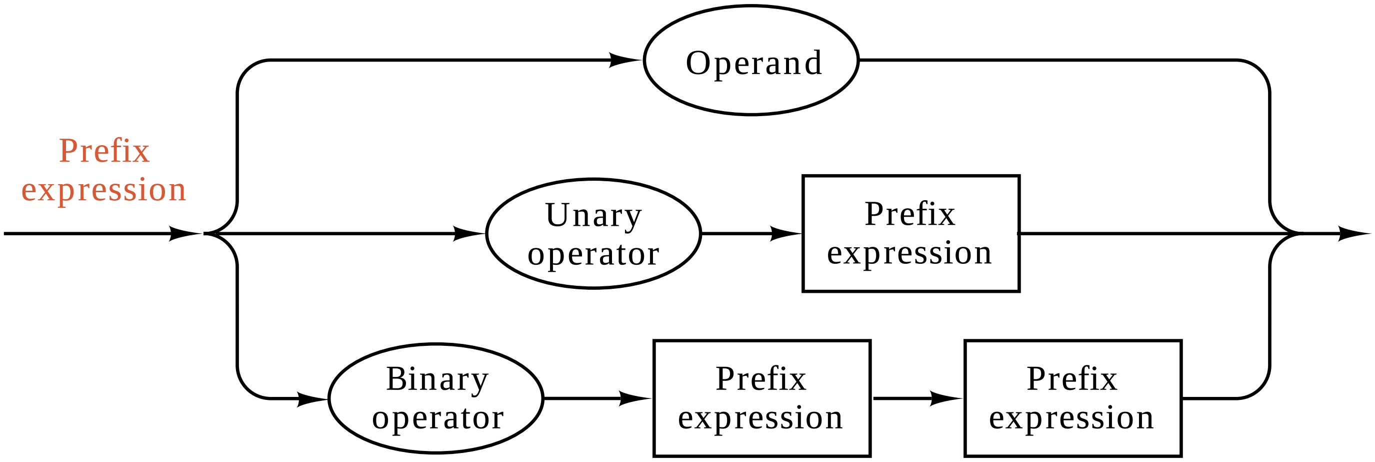
THEOREM 13.1 If $E$ is a properly formed expression in postfix form, then $E$ must satisfy the running sum condition.

THEOREM 13.2 A properly formed expression in postfix form will be correctly evaluated by method evaluate_postfix.

THEOREM 13.3 If $E$ is any sequence of operands and operators that satisfies the running-sum condition, then $E$ is a properly formed expression in postfix form.

THEOREM 13.4 An expression in postfix form that satisfies the running-sum condition corresponds to exactly one fully bracketed expression in infix form. Hence no parentheses are needed to achieve the unique representation of an expression in postfix form.

# Syntax Diagrams for Polish Expressions


Prefix expression

Operand

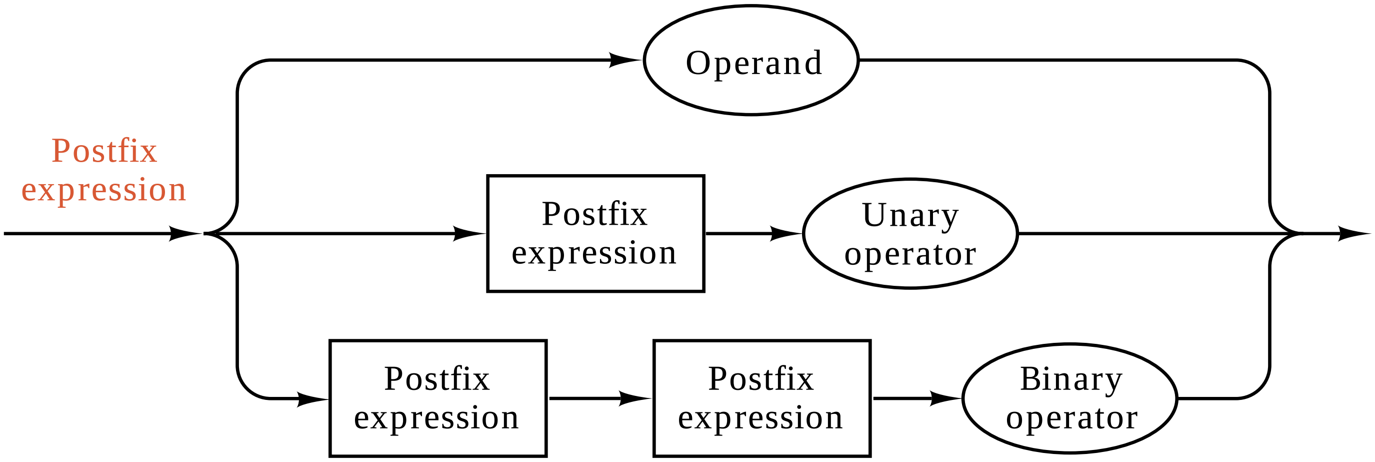
Unary operator

Prefix expression

Binary operator

Prefix expression

Prefix expression


Postfix expression

Operand

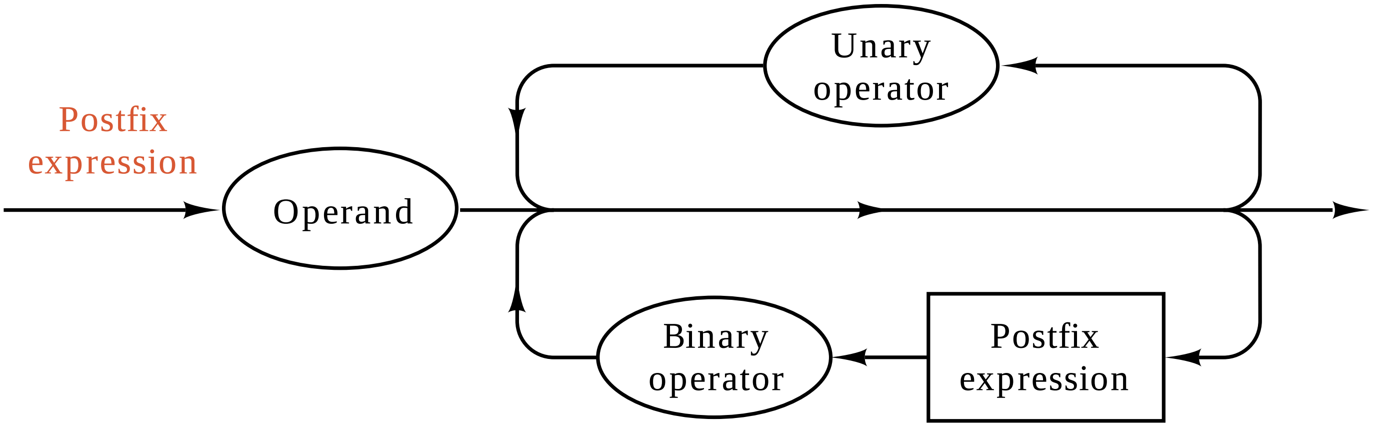
Postfix expression

Unary operator

Postfix expression

Postfix expression

Binary operator


Postfix expression

Operand

Unary operator

Binary operator

Postfix expression

# Recursive Evaluation of Postfix Expressions

```
Error_code Expression :: evaluate_postfix(Value &result)
/* Post: The tokens in Expression up to the first end_expression symbol are re-
         moved. If these tokens do not represent a legal postfix expression, a code of
         fail is returned. Otherwise a code of success is returned, and the removed
         sequence of tokens is evaluated to give Value result. */

{
   Token first_token, final_token;
   Error_code outcome;
   if (get_token(first_token) == fail || first_token.kind( ) != operand)
      outcome = fail;
   else {
      outcome = recursive_evaluate(first_token, result, final_token);
      if (outcome == success && final_token.kind( ) != end_expression)
         outcome = fail;
   }
   return outcome;
}
```

# Recursive Function for Postfix Evaluation

```
Error_code Expression :: recursive_evaluate(const Token &first_token,
                                            Value &result,
                                            Token &final_token)
/* Pre:  Token first_token is an operand.
   Post: If the first_token can be combined with initial tokens of the Expression
         to yield a legal postfix expression followed by either an end_expression
         symbol or a binary operator, a code of success is returned, the legal postfix
         subexpression is evaluated, recorded in result, and the terminating Token
         is recorded as final_token.  Otherwise a code of fail is returned.  The initial
         tokens of Expression are removed.
   Uses: Methods of classes Token and Expression, including recursive_evaluate
         and functions do_unary, do_binary, and get_value.  */

{
   Value first_segment = get_value(first_token),
         next_segment;
   Error_code outcome;
   Token current_token;
   Token_type current_type;

   do {
      outcome = get_token(current_token);
      if (outcome != fail) {
         switch (current_type = current_token.kind( )) {
         case binaryop:          //   Binary operations terminate subexpressions.
         case end_expression:    //   Treat subexpression terminators together.
            result = first_segment;
            final_token = current_token;
            break;
```

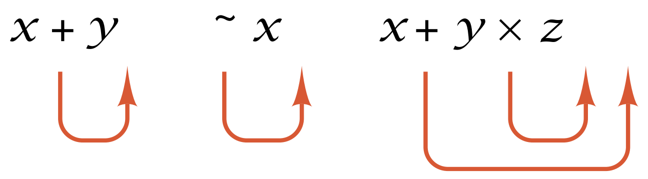
# Recursive Postfix Evaluation, Continued

```
      case unaryop:
        first_segment = do_unary(current_token, first_segment);
        break;

      case operand:
        outcome = recursive_evaluate(current_token,
                                    next_segment, final_token);
        if (outcome ==  success && final_token.kind( )  != binaryop)
          outcome = fail;
        else
          first_segment = do_binary(final_token, first_segment,
                                  next_segment);
        break;
      }
    }
  } while (outcome ==  success && current_type  != end_expression &&
                                 current_type  != binaryop);
  return outcome;
}
```

# Translation from Infix Form to Postfix Form

Delay each operator until its right-hand operand has been translated. A simple operand goes to output with no delay.

Infix form:

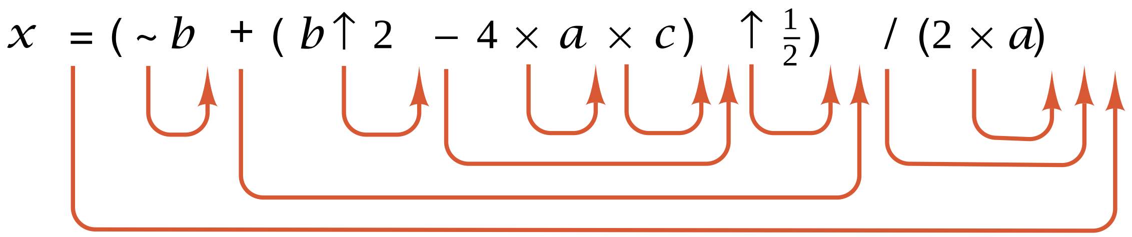


$x + y$ $\sim x$ $x + y \times z$

Postfix form:

$x\ y\ +$ $x\ \sim$ $x\ y\ z\ \times\ +$

Infix form:



$x = (\sim b + (b \uparrow 2 - 4 \times a \times c) \uparrow \frac{1}{2}) / (2 \times a)$

Postfix form:

$x\ b \sim b\ 2 \uparrow 4\ a \times c \times - \frac{1}{2} \uparrow + 2\ a \times / =$

Problem: What token will terminate the right-hand operand of a given operator and thereby mark the place at which that operator should be placed?

To find this token, we must take both parentheses and priorities of operators into account.

# Tokens Terminating a Right-Hand Operand

If *op* is an operator in an infix expression, then its right-hand operand contains all tokens on its right until one of the following is encountered:

1. the end of the expression;
2. an unmatched right parenthesis ')';
3. an operator of priority less than or equal to that of *op*, not within a bracketed sub-expression, if *op* has priority less than 6; or
4. an operator of priority strictly less than that of *op*, not within a bracketed subexpression, if *op* has priority 6.

# Stack Processing

1. At the end of the expression, all operators are output.
2. A right parenthesis causes all operators found since the corresponding left parenthesis to be output.
3. An operator of priority not 6 causes all other operators of greater or equal priority to be output.
4. An operator of priority 6 causes all other operators of strictly greater priority to be output, if such operators exist.

# Infix to Postfix Translation

```
Expression Expression :: infix_to_postfix( )
/* Pre:  The Expression stores a valid infix expression.
   Post: A postfix expression that translates the infix expression is returned.  */

{
   Expression answer;
   Token current, prior;
   Stack delayed_operations;

   while (get_token(current) != fail) {
      switch (current.kind( )) {
      case operand:
         answer.put_token(current);
         break;

      case leftparen:
         delayed_operations.push(current);
         break;

      case rightparen:
         delayed_operations.top(prior);
         while (prior.kind( ) != leftparen) {
            answer.put_token(prior);
            delayed_operations.pop( );
            delayed_operations.top(prior);
         }
         delayed_operations.pop( );
         break;
```

# Infix to Postfix Translation, Continued

```
      case unaryop:
      case binaryop:                 //  Treat all operators together.
         bool end_right = false;     //  End of right operand reached?

         do {
            if (delayed_operations.empty( )) end_right = true;
            else {
               delayed_operations.top(prior);
               if (prior.kind( ) ==  leftparen) end_right = true;
               else if (prior.priority( ) < current.priority( )) end_right = true;
               else if (current.priority( ) ==  6) end_right = true;
               else answer.put_token(prior);
               if (!end_right) delayed_operations.pop( );
            }
         } while (!end_right);

         delayed_operations.push(current);
         break;
      }
   }
   while (!delayed_operations.empty( )) {
      delayed_operations.top(prior);
      answer.put_token(prior);
      delayed_operations.pop( );
   }
   answer.put_token(";");
   return answer;
}
```

| *Input Token* | *Contents of Stack (rightmost token is on top)* | *Output Token(s)* |
|---|---|---|
| $x$ | | $x$ |
| $=$ | $=$ | |
| $($ | $=\ \ ($ | |
| $\sim$ | $=\ \ (\ \ \sim$ | |
| $b$ | $=\ \ (\ \ \sim$ | $b$ |
| $+$ | $=\ \ (\ \ +$ | $\sim$ |
| $($ | $=\ \ (\ \ +\ \ ($ | |
| $b$ | $=\ \ (\ \ +\ \ ($ | $b$ |
| $\uparrow$ | $=\ \ (\ \ +\ \ (\ \ \uparrow$ | |
| $2$ | $=\ \ (\ \ +\ \ (\ \ \uparrow$ | $2$ |
| $-$ | $=\ \ (\ \ +\ \ (\ \ -$ | $\uparrow$ |
| $4$ | $=\ \ (\ \ +\ \ (\ \ -$ | $4$ |
| $\times$ | $=\ \ (\ \ +\ \ (\ \ -\ \ \times$ | |
| $a$ | $=\ \ (\ \ +\ \ (\ \ -\ \ \times$ | $a$ |
| $\times$ | $=\ \ (\ \ +\ \ (\ \ -\ \ \times$ | $\times$ |
| $c$ | $=\ \ (\ \ +\ \ (\ \ -\ \ \times$ | $c$ |
| $)$ | $=\ \ (\ \ +$ | $\times\ \ -$ |
| $\uparrow$ | $=\ \ (\ \ +\ \ \uparrow$ | |
| $\frac{1}{2}$ | $=\ \ (\ \ +\ \ \uparrow$ | $\frac{1}{2}$ |
| $)$ | $=$ | $\uparrow\ +$ |
| $/$ | $=\ \ /$ | |
| $($ | $=\ \ /\ \ ($ | |
| $2$ | $=\ \ /\ \ ($ | $2$ |
| $\times$ | $=\ \ /\ \ (\ \ \times$ | |
| $a$ | $=\ \ /\ \ (\ \ \times$ | $a$ |
| $)$ | $=\ \ /$ | $\times$ |
| *end of expression* | | $/\ \ =$ |

Translation of quadratic formula into postfix

# An Interactive Expression Evaluator

## Goal:

Develop a program that can evaluate a function that is typed in interactively while the program is running. One such application is a program that will draw the graph of a mathematical function.

## First task:

Take as input an expression involving constants, variable(s), arithmetic operators, and standard functions, with bracketing allowed, as typed in from the terminal, and translate the expression into postfix form.

## Second task:

Evaluate this postfix expression for values of the variable(s) given as its calling parameter(s) and return the answer, which can then be graphed.

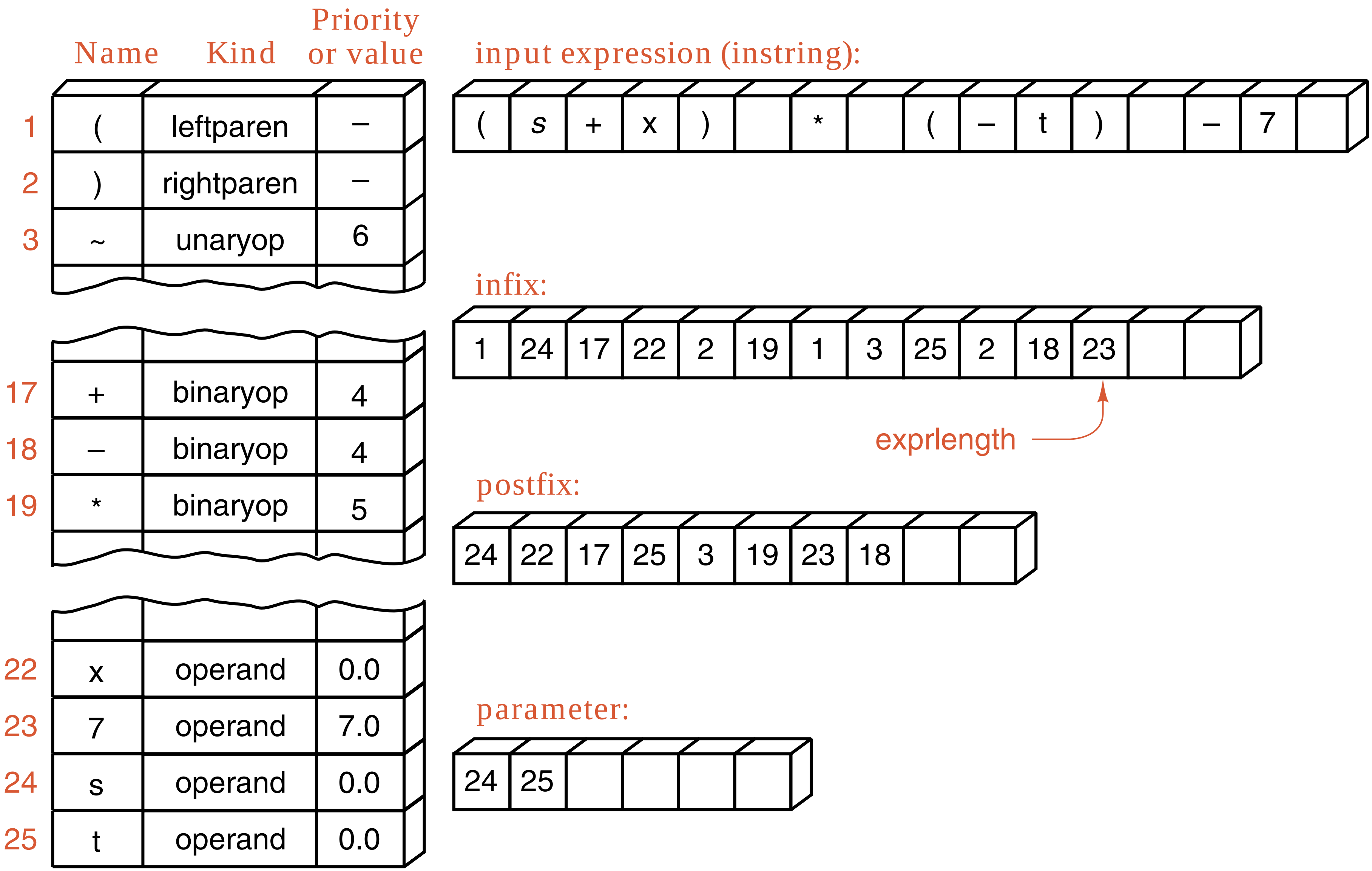
## Overall structure:

```
int main( )
/* Pre:  None
   Post: Acts as a menu-driven graphing program.
   Uses: Classes Expression and Plot, and functions introduction,
         get_command, and do_command.  */
{
   introduction( );
   Expression infix;          //   Infix expression from user
   Expression postfix;        //   Postfix translation
   Plot graph;
   char ch;
   while ((ch = get_command( )) != 'q')
      do_command(ch, infix, postfix, graph);
}
```

```
void do_command(char c, Expression &infix,
                            Expression &postfix, Plot &graph)
/* Pre:  None
   Post: Performs the user command represented by char c on the Expression infix,
         the Expression postfix, and the Plot graph.
   Uses: Classes Token, Expression and Plot. */
{ switch (c) {
  case ′r′:                          //  Read an infix expression from the user.
    infix.clear( );
    infix.read( );
    if (infix.valid_infix( ) ==  success) postfix = infix.infix_to_postfix( );
    else cout ≪ "Warning: Bad expression ignored. " ≪ endl; break;
  case ′w′:                          //  Write the current expression.
    infix.write( );
    postfix.write( ); break;
  case ′g′:                          //  Graph the current postfix expression.
    if (postfix.size( ) <= 0)
      cout ≪ "Enter a valid expression before graphing!" ≪ endl;
    else {
      graph.clear( );
      graph.find_points(postfix);
      graph.draw( ); } break;
  case ′l′:                          //  Set the graph limits.
    if (graph.set_limits( ) ! = success)
      cout ≪ "Warning: Invalid limits" ≪ endl; break;
  case ′p′:                          //  Print the graph parameters.
    Token :: print_parameters( ); break;
  case ′n′:                          //  Set new graph parameters.
    Token :: set_parameters( ); break;
  case ′h′:                          //  Give help to user.
    help( ); break;
  }
}
```

# Representation of the Data

- In addition to the (independent) variable x used for plotting, an expression may contain further variables that we call ***parameters*** for the graph.
- The program must establish the definitions of the predefined tokens (such as the operators +, −, and *), the operand x used to represent a coordinate in the graphing, and perhaps some constants.
- For each different token we must remember:
  - Its name (as a String), so that we can recognize it in an input expression;
  - Its kind, one of operand, unary operator, binary operator, and right unary operator (like factorial '!'), left parenthesis, or right parenthesis;
  - For operators, a priority;
  - For operands, a value.
- We associate an integer code with each token and place this code in the expression, rather than the full record. We shall thus set up a ***lexicon*** for the tokens, which will include an array indexed by the integer codes; this array will hold the full records for the tokens.
- Expressions are *lists* of token codes.
- Translation from infix to postfix uses a *stack* of token codes.
- Evaluation of the expression uses a *recursive* function.
- Given a token *code*, the *lexicon* provides information about the token.
- A *hash table* finds the code associated with a token name.



| | Name | Kind | Priority or value |
|---|---|---|---|
| 1 | ( | leftparen | – |
| 2 | ) | rightparen | – |
| 3 | ~ | unaryop | 6 |
| 17 | + | binaryop | 4 |
| 18 | – | binaryop | 4 |
| 19 | * | binaryop | 5 |
| 22 | x | operand | 0.0 |
| 23 | 7 | operand | 7.0 |
| 24 | s | operand | 0.0 |
| 25 | t | operand | 0.0 |

Lexicon

input expression (instring):

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ( | *s* | + | x | ) | | * | | ( | – | t | ) | | – | 7 | |

infix:

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 24 | 17 | 22 | 2 | 19 | 1 | 3 | 25 | 2 | 18 | 23 | | |

postfix:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 24 | 22 | 17 | 25 | 3 | 19 | 23 | 18 | | |

parameter:

| | | | | | |
|---|---|---|---|---|---|
| 24 | 25 | | | | |

# The Lexicon in C++ Implementation

## Token records:

```
struct Token_record {
   String name;
   double value;
   int priority;
   Token_type kind;
};
```

## Lexicon class:

```
struct Lexicon {
   Lexicon( );
   int hash(const String &x) const;
   void set_standard_tokens( );    //   Set up the predefined tokens.
   int count;                      //   Number of records in the Lexicon
   int index_code [hash_size];     //   Declare the hash table.
   Token_record token_data [hash_size];
};
```

## Token class:

```
class Token {
public:
//   Add methods here.
private:
   int code;
   static Lexicon symbol_table;
   static List<int> parameters;
};
List<int> Token :: parameters;    //   Allocate storage for static members.
Lexicon Token :: symbol_table;
```

# Token Class

```
class Token {
public:
   Token( ) { }
   Token (const String &x);
   Token_type kind( ) const;
   int priority( ) const;
   double value( ) const;
   String name( ) const;
   int code_number( ) const;
   static void set_parameters( );
   static void print_parameters( );
   static void set_x(double x_val);
private:
   int code;
   static Lexicon symbol_table;
   static List<int> parameters;
};
```

# Token Constructor

```
Token :: Token(const String &identifier)
/* Post: A Token corresponding to String identifier is constructed. It shares its code
         with any other Token object with this identifier.
   Uses: The class Lexicon. */

{
  int location = symbol_table.hash(identifier);
  if (symbol_table.index_code[location] == -1) {
                                                //  Create a new record.
    code = symbol_table.count++;
    symbol_table.index_code[location] = code;
    symbol_table.token_data[code] = attributes(identifier);
    if (is_parameter(symbol_table.token_data[code]))
      parameters.insert(0, code);
  }
  else code = symbol_table.index_code[location];
                                                //  Code of an old record
}
```

The auxiliary function attributes examines the name of a token and sets up an appropriate Token_record according to our conventions.

# Parameter Values

```
void Token :: set_parameters( )
/* Post:  All parameter values are printed for the user, and any changes specified by the
          user are made to these values.
   Uses:  Classes List, Token_record, and String, and function read_num.  */

{
   int n = parameters.size( );
   int index_code;
   double x;

   for (int i = 0;  i < n;  i++) {
      parameters.retrieve(i, index_code);
      Token_record &r = symbol_table.token_data[index_code];

      cout  << "Give a new value for parameter " << (r.name).c_str( )
            << " with value " << r.value << endl;
      cout  << "Enter a value or a new line to keep the old value: "
            << flush;
      if (read_num(x) ==  success) r.value = x;
   }
}
```

*Data Structures and Program Design In C++*

# The Lexicon Constructor

```
Lexicon :: Lexicon( )
/* Post:  The Lexicon is initialized with the standard tokens.
   Uses: set_standard_tokens  */

{
   count = 0;
   for (int i = 0;  i < hash_size;  i++)
      index_code[i]  =  -1;          //    code for an empty hash slot
   set_standard_tokens( );
}
```

```
void Lexicon :: set_standard_tokens( )
{
   int i = 0;
   String symbols = (String)
"; ( ) ~ abs sqr sqrt exp ln lg sin cos arctan round trunc ! % + - * / ^ x pi e";
   String word;
   while (get_word(symbols, i++, word)  != fail) {
      Token t = word;
   }
   token_data[23].value = 3.14159;
   token_data[24].value = 2.71828;
}
```

# Predefined Tokens for Expression Evaluation

| *Token* | *Name* | *Kind* | *Priority / Value* | |
|---|---|---|---|---|
| 0 | ; | end_expression | | |
| 1 | ( | leftparen | | |
| 2 | ) | rightparen | | |
| 3 | ∼ | unaryop | 6 | *negation* |
| 4 | abs | unaryop | 6 | |
| 5 | sqr | unaryop | 6 | |
| 6 | sqrt | unaryop | 6 | |
| 7 | exp | unaryop | 6 | |
| 8 | ln | unaryop | 6 | *natural logarithm* |
| 9 | lg | unaryop | 6 | *base 2 logarithm* |
| 10 | sin | unaryop | 6 | |
| 11 | cos | unaryop | 6 | |
| 12 | arctan | unaryop | 6 | |
| 13 | round | unaryop | 6 | |
| 14 | trunc | unaryop | 6 | |
| 15 | ! | right unary | 6 | *factorial* |
| 16 | % | right unary | 6 | *percentage* |
| 17 | + | binaryop | 4 | |
| 18 | − | binaryop | 4 | |
| 19 | ∗ | binaryop | 5 | |
| 20 | / | binaryop | 5 | |
| 21 | ^ | binaryop | 6 | |
| 22 | x | operand | 0.00000 | |
| 23 | pi | operand | 3.14159 | |
| 24 | e | operand | 2.71828 | |

# Hash Table Processing

```
int Lexicon :: hash(const String &identifier) const
/* Post: Returns the location in table Lexicon :: index_code that corresponds to the
         String identifier. If the hash table is full and does not contain a record for
         identifier, the exit function is called to terminate the program.
   Uses: The class String, the function exit. */
{
   int location;
   const char *convert = identifier.c_str( );
   char first = convert[0], second;   //   First two characters of identifier
   if (strlen(convert) >= 2) second = convert[1];
   else second = first;

   location = first % hash_size;
   int probes = 0;
   while (index_code[location] >= 0 &&
            identifier != token_data[index_code[location]].name) {
      if (++probes >= hash_size) {
         cout << "Fatal Error: Hash Table overflow. Increase its size\n";
         exit(1);
      }
      location += second;
      location %= hash_size;
   }
   return location;
}
```

# Class Expression: Token Lists

```
class Expression {
public:
   Expression( );
   Expression(const Expression &original);
   Error_code get_token(Token &next);
   void put_token(const Token &next);
   Expression infix_to_postfix( );
   Error_code evaluate_postfix(Value &result);
   void read( );
   void clear( );
   void write( );
   Error_code valid_infix( );
   int size( );
   void rewind( );                    //    resets the Expression to its beginning
private:
   List<Token> terms;
   int current_term;
   Error_code recursive_evaluate(const Token &first_token,
                                         Value &result, Token &final_token);
};
```

```
Error_code Expression :: get_token(Token &next)
/* Post:  The Token next records the current_term of the Expression;
          current_term is incremented; and an error code of success is returned, or if
          there is no such term a code of fail is returned.
   Uses:  Class List.  */
{
   if (terms.retrieve(current_term, next) != success) return fail;
   current_term++;
   return success;
}
```

# Reading an Expression

```
void Expression :: read( )
/* Post: A line of text, entered by the user, is split up into tokens and stored in the
         Expression.
   Uses: Classes String, Token, and List. */

{
   String input, word;
   int term_count = 0;
   int x;
   input = read_in(cin, x);
   add_spaces(input);                 //   Tokens are now words of input.

   bool leading = true;
   for (int i = 0; get_word(input, i, word) != fail; i++) {
                                      //   Process next token
      if (leading)
         if (word == "+") continue;   //   unary +
         else if (word == "-") word = "~";  //   unary -

      Token current = word;           //   Cast word to Token.
      terms.insert(term_count++, current);
      Token_type type = current.kind( );
      if (type == leftparen || type == unaryop || type == binaryop)
         leading = true;
      else
         leading = false;
   }
}
```

# Leading and Non-Leading Positions

The value of Boolean variable leading detects whether an operator has a left argument. We shall show that if leading is true, then the current token can have no left argument and therefore cannot legally be a binary operator. In this way, our method is able to distinguish between unary and binary versions of the operators + and −.

## Tokens Legal in Leading and Non-Leading Positions

| | *Previous token*<br>*any one of:* | *Legal tokens*<br>*any one of:* |
|---|---|---|
| *Leading position:* | | |
| | start of expression | operand |
| | binary operator | unary operator |
| | unary operator | left parenthesis |
| | left parenthesis | |
| *Nonleading position:* | | |
| | operand | binary operator |
| | right unary operator | right unary operator |
| | right parenthesis | right parenthesis |
| | | end of expression |

# Error Checking for Correct Syntax

```
Error_code Expression :: valid_infix( )
/* Post: A code of success or fail is returned according to whether the Expression
         is a valid or invalid infix sequence.
   Uses: Class Token. */
{
   Token current;
   bool leading = true;
   int paren_count = 0;
   while (get_token(current) != fail) {
      Token_type type = current.kind( );
      if (type == rightparen || type == binaryop
                             || type == rightunaryop) {
         if (leading) return fail;
      }
      else if (!leading) return fail;
      if (type == leftparen) paren_count++;
      else if (type == rightparen) {
         paren_count--;
         if (paren_count < 0) return fail;
      }
      if (type == binaryop || type == unaryop || type == leftparen)
         leading = true;
      else leading = false;
   }
   if (leading)
      return fail;                 //  An expected final operand is missing.
   if (paren_count > 0)
      return fail;                 //  Right parentheses are missing.
   rewind( );
   return success;
}
```

# Auxiliary Evaluation Functions

```
Value get_value(const Token &current)
/* Pre:   Token current is an operand.
   Post:  The Value of current is returned.
   Uses:  Methods of class Token.  */
{
   return current.value( );
}
```

```
Value do_binary(const Token &operation,
                const Value &first_argument,
                const Value &second_argument)
/* Pre:   Token operation is a binary operator.
   Post:  The Value of operation applied to the pair of Value parameters is returned.
   Uses:  Methods of class Token.  */
{
   switch (operation.code_number( )) {
   case 17:
      return first_argument + second_argument;
   case 18:
      return first_argument − second_argument;
   case 19:
      return first_argument * second_argument;
   case 20:
      return first_argument/second_argument;
   case 21:
      return exp(first_argument, second_argument);
   }
}
```

# Graphing the Expression: The Class Plot

```
class Plot {
public:
   Plot( );
   Error_code set_limits( );
   void find_points(Expression &postfix);
   void draw( );
   void clear( );
   int get_print_row(double y_value);
   int get_print_col(double x_value);
private:
   Sortable_list<Point> points;   //   records of points to be plotted
   double x_low, x_high;          //   x limits
   double y_low, y_high;          //   y limits
   double x_increment;            //   increment for plotting
   int max_row, max_col;          //   screen size
};
```

```
int Plot :: get_print_row(double y_value)
/* Post:  Returns the row of the screen at which a point with y-coordinate y_value should
          be plotted, or returns −1 if the point would fall off the edge of the screen.  */
{
   double interp_y =
      ((double) max_row) * (y_high − y_value)/(y_high − y_low) + 0.5;
   int answer = (int) interp_y;
   if (answer < 0 || answer > max_row) answer = −1;
   return answer;
}
```

# Points

```
struct Point {
   int row, col;
   Point( );
   Point(int a, int b);
   bool operator == (const Point &p);
   bool operator != (const Point &p);
   bool operator >= (const Point &p);
   bool operator <= (const Point &p);
   bool operator >  (const Point &p);
   bool operator <  (const Point &p);
};

bool Point :: operator < (const Point &p)
{return (row < p.row) || (col < p.col && row == p.row); }

void Plot :: find_points(Expression &postfix)
/* Post: The Expression postfix is evaluated for values of x that range from x_low
         to x_high in steps of x_increment. For each evaluation we add a Point to
         the Sortable_list points.
   Uses: Classes Token, Expression, Point, and List. */
{
   double x_val = x_low;
   double y_val;
   while (x_val <= x_high) {
      Token :: set_x(x_val);
      postfix.evaluate_postfix(y_val);
      postfix.rewind( );
      Point p(get_print_row(y_val), get_print_col(x_val));
      points.insert(0, p);
      x_val += x_increment;
   }
}
```

# Drawing the Graph

```
void Plot :: draw( )
/* Post: All screen locations represented in points have been marked with the character
         '#' to produce a picture of the stored graph.
   Uses: Classes Point and Sortable_list and its method merge_sort. */

{
   points.merge_sort( );
   int at_row = 0, at_col = 0;         //   cursor coordinates on screen

   for (int i = 0; i < points.size( ); i++) {
      Point q;
      points.retrieve(i, q);
      if (q.row < 0 || q.col < 0) continue;   //   off the scale of the graph
      if (q.row < at_row || (q.row == at_row && q.col < at_col)) continue;

      if (q.row > at_row) {                   //   Move cursor down the screen.
         at_col = 0;
         while (q.row > at_row) {
            cout ≪ endl;
            at_row++;
         }
      }

      if (q.col > at_col)                     //   Advance cursor horizontally.
         while (q.col > at_col) {
            cout ≪ " ";
            at_col++;
         }
      cout ≪ "#";
      at_col++;
   }
   while (at_row++ <= max_row) cout ≪ endl;
}
```

# A Graphics-Enhanced Plot Class

```
void Plot :: find_points(Expression &postfix)
/* Post: The Expression postfix is evaluated for values of x that range from x_low
         to x_high in steps of x_increment. For each evaluation we add a Point of
         the corresponding data point to the Sortable_list points.
   Uses: Classes Token, Expression, Point, and List and the library <graphics.h>.
         */

{
   int graphicdriver = DETECT, graphicmode;
   initgraph( &graphicdriver,  &graphicmode, "");   //   screen detection and
   graphresult( );                                  //   initialization
   max_col = getmaxx( );                            //   with graphics.h library
   max_row = getmaxy( );

   double x_val = x_low;
   double y_val;
   while (x_val <= x_high) {
      Token :: set_x(x_val);
      postfix.evaluate_postfix(y_val);
      postfix.rewind( );
      Point p(get_print_row(y_val), get_print_col(x_val));
      points.insert(0, p);
      x_val += x_increment;
   }
}
```

# Plotting the Points

```
void Plot :: draw( )
/* Post: All pixels represented in points have been marked to produce a picture of the
         stored graph.
   Uses: Classes Point and Sortable_list and the library <graphics.h> */

{
   for (int i = 0;  i < points.size( );  i++) {
      Point q;
      points.retrieve(i, q);
      if (q.row < 0 || q.col < 0) continue;    //   off the scale of the graph
      if (q.row > max_row || q.col > max_col) continue;
      putpixel(q.col, q.row, 3);               //   graphics.h library function
   }

   cout "Enter a character to clear screen and continue: " << flush;
   char wait_for;
   cin >> wait_for;
   restorecrtmode( );                          //   graphics.h library function
}
```