1. Introduction. Mathematical expressions are usually written in *infix notation*, e.g,

$$a + b \times c - d,$$

in which operators are placed between operands.

We can intuitively *parse* and evaluate the above expression in our mind:

Add a with the result of multiplying b with c, and then subtract d from the sum.

Such parsing of an expression relies on the *precedence* and *associativity* of mathematical operators. In the above example, $\times$ has higher precedence than $+$ or $-$; $+$ and $-$ have the same precedence. Hence c is multiplied to b, not $a + b$.

Although not immediately evident in the presented example, associativity plays an important role while parsing expressions when operators such as ˆ are used—the exponentiation operator ˆ is right-associative, whereas all other aforementioned operators are left-associative. For example, 2ˆ3ˆ4 is 2ˆ(3ˆ4), not (2ˆ3)ˆ4.

Note that the associativity rule needs to be considered even without an introduction of right-associative operators—consider an expression $a - b + c$. It is ambiguous whether it should be parsed as $a - (b + c)$ or $(a - b) + c$ without an associativity rule. In general, associativity should be considered to properly deal with operators that have the same precedence in the same nested level of an expression.

While a human reader parses mathematical expressions and evaluate them in nonlinear fashion via intuition, in order to systematically parse expressions so that a machine can evaluate them, infix notation is not very convenient. *Reverse Polish notation (RPN)* was invented to remedy this situation. RPN, as its name hints, puts operators *after* the operands:

$$a\, b\, c \times + \, d\ - .$$

It is very straightforward to evaluate the above expression with a computer—one just puts operands to a stack in left-to-right order and pops them and evaluate accordingly when an operator is met.

Then the question is: How can we transform infix notation to RPN? Edsger Dijkstra invented the algorithm, and it was named *shunting-yard algorithm* because of the resemblance of its operations to that of a shunting yard.

Here is an imagery of the algorithm:

- Draw a T-shaped railroad in your mind.
- A "train" of operands and operators in infix notation are entering from the right "arm" of the T-shape to the left "arm".
- When an operator is met, however, it occasionally enters the "stem" part of the T-shape following the rule described by the shunting-yard algorithm we are going to demonstrate.
- Note that the "stem" part is a LIFO (last in, first out) structure, which is along the line with our railroad imagery.

Keep these images in mind as they will help you with a solid understanding of the algorithm.

```
#include <ctype.h>    /* isspace, isdigit */
#include <stdbool.h>
#include <stdio.h>
  ⟨ Preprocessor definitions ⟩
  ⟨ Token Implementation 2 ⟩
  ⟨ Stack Implementation 10 ⟩
  ⟨ Shunting-Yard Algorithm 12 ⟩
  int main(void)
  {    /* Driver for the shunting-yard algorithm. */
    int c;
    while (true) {
      printf("> ");    /* Input prompt */
      if ((c = getchar()) ≡ EOF) {
        putchar('\n');
        return 0;
      }
      ungetc(c, stdin);
      if (¬shunting_yard()) {      /* shunting_yard returns false when an EOF is met. */
        putchar('\n');
        return 0;
      }
    }
  }
```

2. Token. For the scope of this program, we consider arithmetic expressions that consist of non-negative integers; arithmetic operators $+$, $-$, $\times$, $\div$, and ˆ; and parentheses.

Each lexical token is represented by **Token**, and the type of an operator is represented by **Sym** (which can also represent a parenthesis, for simplicity) in our program. Note that **Token**s represent the aforementioned numbers, operators, parentheses, and additionally, EOF for convenience of I/O handling.

```
⟨ Token Implementation 2 ⟩ ≡
  typedef enum Sym {
    ADD,    /* + */
    SUB,    /* − */
    MUL,    /* × */
    DIV,    /* ÷ */
    POW,    /* ˆ */
    LPR,    /* ( */
    RPR    /* ) */
  } Sym;
  ⟨ Symbol Global Variables 3 ⟩
  typedef struct Token {
    enum {
      EOF_T, SYM_T, NUM_T
    } type;
    union {
      Sym sym;    /* SYM_T */
      long num;    /* NUM_T */
    } u;
  } Token;
  ⟨ Token Subroutines 6 ⟩
```

This code is used in section 1.

3. For printing to *stdout*, SYM_NAMES stores string representations of symbols.

```
⟨ Symbol Global Variables 3 ⟩ ≡
  static const char SYM_NAMES[ ][4] = {"ADD", "SUB", "MUL", "DIV", "POW", "LPR", "RPR"};
```

See also sections 4, 5, and 7.

This code is used in section 2.

4. SYM_ASSOC stores an associativity of each SYM. LASSOC, RASSOC, and NASSOC each indicates left-associativity, right-associativity, and non-associativity.

```
⟨ Symbol Global Variables 3 ⟩ +≡
  static const enum {
    LASSOC, RASSOC, NASSOC
  } SYM_ASSOC[ ] = {
  LASSOC,    /* ADD */
  LASSOC,    /* SUB */
  LASSOC,    /* MUL */
  LASSOC,    /* DIV */
  RASSOC,    /* POW */
  NASSOC,    /* LPR */
  NASSOC    /* RPR */
  };
```

5. SYM_PREC stores a precedence of each SYM. LPR and RPR have −1 assigned, which means they are not applicable to a precedence rule.

```
⟨ Symbol Global Variables 3 ⟩ +≡
  static const int SYM_PREC[ ] = {
  0,     /* ADD */
  0,     /* SUB */
  1,     /* MUL */
  1,     /* DIV */
  2,     /* POW */
  −1,     /* LPR */
  −1     /* RPR */
  };
```

6. *op_cmp* compares two **Sym**s and returns zero if a and b have the same precedence, a positive value if a has a higher precedence, a negative value if a has a lower precedence. Note that it is an error to pass LPR or RPR as an argument.

```
⟨ Token Subroutines 6 ⟩ ≡
  static inline int op_cmp(Sym a, Sym b)
  {
    return SYM_PREC[a] − SYM_PREC[b];
  }
```

See also sections 8 and 9.

This code is used in section 2.

7. To easily convert a **char** to a corresponding **Sym**, SYM_TBL provides a mapping between to two.

```
⟨ Symbol Global Variables 3 ⟩ +≡
  static const Sym SYM_TBL[ ] = {['+'] = ADD, ['-'] = SUB, ['*'] = MUL, ['/'] = DIV, ['^'] = POW,
      ['('] = LPR, [')'] = RPR};
```

8. *isop* and *ispar* check if an input character is an operator or a parenthesis, respectively.

```
⟨ Token Subroutines 6 ⟩ +≡
  static inline bool isop(int c)
  {
    return c ≡ '+' ∨ c ≡ '-' ∨ c ≡ '*' ∨ c ≡ '/' ∨ c ≡ '^';
  }
  static inline bool ispar(int c)
  {
    return c ≡ '(' ∨ c ≡ ')';
  }
```

9. *gettok* reads characters from *stdin* and returns a recognized token.

```
⟨ Token Subroutines 6 ⟩ +≡
  Token gettok( )
  {
    int ch;
    do {
      ch = getchar( );
    } while (isspace(ch));      /* Ignore whitespaces */
    if (isop(ch) ∨ ispar(ch)) {      /* an operator or a parenthesis read */
      return (struct Token){SYM_T, .u.sym = SYM_TBL[ch]};
    }
    if (ch ≡ EOF ∨ ¬isdigit(ch)) {      /* an EOF or an unknown character read */
      return (struct Token) {EOF_T};
    }      /* Read a number, including the already-read digit ch. */
    ungetc(ch, stdin);
    long num;
    if (scanf("%ld", &num) ≤ 0) {      /* Unknown case */
      return (struct Token) {EOF_T};
    }
    return (struct Token){NUM_T, .u.num = num};
  }
```

10. Stack. Operators need to be saved in a stack—the "stem" of the T-shape—and since operators are represented by **Sym**, our stack implementation only needs to store **Sym** type.

Our **Stack** is backed by array *container* that lives in the stack—as opposed to the heap—so the maximum size should be relatively small. It is set to 1000 with `STK_CAP`, which is probably more than enough for our purpose.

The top element is accessed through $*top$, and the size is tracked by *size*.

```
#define STK_CAP 1000     /* Maximum size of Stack */
⟨ Stack Implementation 10 ⟩ ≡
  typedef struct Stack {
    Sym *top;     /* Must be initialized to Λ. */
    Sym container[STK_CAP];
    size_t size;     /* Must be initialized to 0. */
  } Stack;
  ⟨ Stack Subroutines 11 ⟩
```

This code is used in section 1.

11. Both *push* and *pop* returns *true* if the operation was successful, *false* otherwise. In each case, failure indicates an overflow and an underflow, respectively.

```
⟨ Stack Subroutines 11 ⟩ ≡
  bool push(Sym val, Stack *s)
  {
    if (s→size ≡ STK_CAP) {     /* overflow */
      return false;
    }
    ++s→size;
    if (¬s→top) {     /* s is empty */
      s→top = s→container;
      *s→top = val;
    }
    else {
      *++s→top = val;
    }
    return true;
  }
  bool pop(Stack *s)
  {
    if (¬s→size) {     /* underflow */
      return false;
    }
    if (s→size ≡ 1) {
      s→top = Λ;
    }
    else {
      --s→top;
    }
    --s→size;
    return true;
  }
```

This code is used in section 10.

12. Shunting-Yard Algorithm. *shunting_yard* implements the shunting-yard algorithm. It reads all tokens in the current input and prints them in RPN.

shunting_yard returns *false* when an `EOF` or an unknown symbol is met during parsing.

```
⟨ Shunting-Yard Algorithm 12 ⟩ ≡
  bool shunting_yard( )
  {
    Stack stk = {0};
    int c;
    ⟨ Read all tokens 13 ⟩
    ⟨ Handle symbols still left in stk 17 ⟩
    if (c ≡ '\n') {    /* The last character input was '\n'. */
      putchar('\n');
      return true;
    }
    else {
      return false;
    }
  }
```

This code is used in section 1.

13. The token-reading process is halted whenever a newline character, an `EOF`, or an unknown symbol is encountered. Otherwise, the input token, which is either a `SYM_T` type or a `NUM_T` type, is handled respectively.

```
⟨ Read all tokens 13 ⟩ ≡
  while (true) {
    c = getchar( );
    if (c ≡ EOF ∨ c ≡ '\n') {
      break;
    }
    ungetc(c, stdin);    /* prepare for a token */
    Token tok = gettok( );
    switch (tok.type) {
    case EOF_T:    /* unknown symbol */
      return false;
    case SYM_T:
      ⟨ Handle a SYM_T token 14 ⟩
      break;
    case NUM_T:
      printf("%ld␣", tok.u.num);
      break;
    }
  }
```

This code is used in section 12.

14. A left parenthesis is handled specially because an expression nested in parentheses must be handled first. Assuming that parentheses are properly balanced, a left parenthesis token pushed onto the stack will be handled when the corresponding right parenthesis token is read.

An operator token is handled according to the precedence rule.

```
⟨ Handle a SYM_T token 14 ⟩ ≡
  if (tok.u.sym ≡ LPR) {
    push(tok.u.sym, &stk);
  }
  else if (tok.u.sym ≡ RPR) {
    ⟨ Handle a right parenthesis 15 ⟩
  }
  else {
    ⟨ Handle an operator 16 ⟩
  }
```

This code is used in section 13.

15. When a right parenthesis is encountered, *stk* is popped until the balancing left parenthesis is found. If *stk* becomes empty before a left parenthesis is met, it means that the parentheses in the original expression were not balanced. In this case, an error message is printed, and *shunting_yard* exits.

```
⟨ Handle a right parenthesis 15 ⟩ ≡
  while (stk.size ∧ ∗stk.top ≠ LPR) {
    printf("%s␣", SYM_NAMES[∗stk.top]);
    pop(&stk);
  }
  if (¬stk.size ∨ ∗stk.top ≠ LPR) {      /∗ Could not find '('! ∗/
    printf("MALFORMED␣EQ\n");
    return true;
  }
  pop(&stk);      /∗ pops LPR ∗/
```

This code is used in section 14.

16. When an operator token is read, *stk* is popped until a parenthesis, an operator with a lower precedence, or a right-associative operator with the same precedence is found.

One might wonder why popping more than once is required. Indeed, the example expressions shown in the introduction can be parsed fine when we substitute the below **while** to an **if**. However, it will in general fail to convert some expressions correctly. Consider a hypothetical operator $\oplus$ which is left-associative but has a higher precedence than $\times$ or $\div$. Now try to convert $2+3\div4\oplus5\times2$ to RPN, without the **while** loop. You'll see why!

```
⟨ Handle an operator 16 ⟩ ≡
  while (stk.size ∧ ∗stk.top ≠ LPR ∧ ∗stk.top ≠ RPR) {
    Sym s = ∗stk.top;
    int cmp = op_cmp(tok.u.sym, s);
    if (cmp < 0) {
      printf("%s␣", SYM_NAMES[s]);
      pop(&stk);
    }
    else if (cmp ≡ 0 ∧ SYM_ASSOC[s] ≡ LASSOC) {
      printf("%s␣", SYM_NAMES[s]);
      pop(&stk);
    }
    else {
      break;
    }
  }
  push(tok.u.sym, &stk);    /∗ Push the current operator token. ∗/
```

This code is used in section 14.

17. When all tokens are read, pop all the tokens left in *stk*. A left parenthesis left in *stk* implies that there was a no matching right parenthesis. In this case, an error message is printed and *shunting_yard* exits.

```
⟨ Handle symbols still left in stk 17 ⟩ ≡
  while (stk.size) {
    Sym s = ∗stk.top;
    if (s ≡ LPR) {
      printf("MALFORMED␣EQ\n");
      return true;
    }
    printf("%s␣", SYM_NAMES[s]);
    pop(&stk);
  }
```

This code is used in section 12.

Shunting-Yard Algorithm