Shunting-yard algorithm

The **shunting-yard algorithm** is a method for parsing mathematical expressions specified in infix notation. It can be used to produce output in Reverse Polish notation (RPN) or as an abstract syntax tree (AST). The algorithm was invented by Edsger Dijkstra and named the "shunting yard" algorithm because its operation resembles that of a railroad shunting yard. Dijkstra first described the Shunting Yard Algorithm in Mathematisch Centrum report MR-35.

Like the evaluation of RPN, the shunting yard algorithm is stack-based. Infix expressions are the form of mathematical notation most people are used to, for instance 3+4 or 3+4*(2−1). For the conversion there are two text variables (strings), the input and the output. There is also a stack that holds operators not yet added to the output queue. To convert, the program reads each symbol in order and does something based on that symbol.

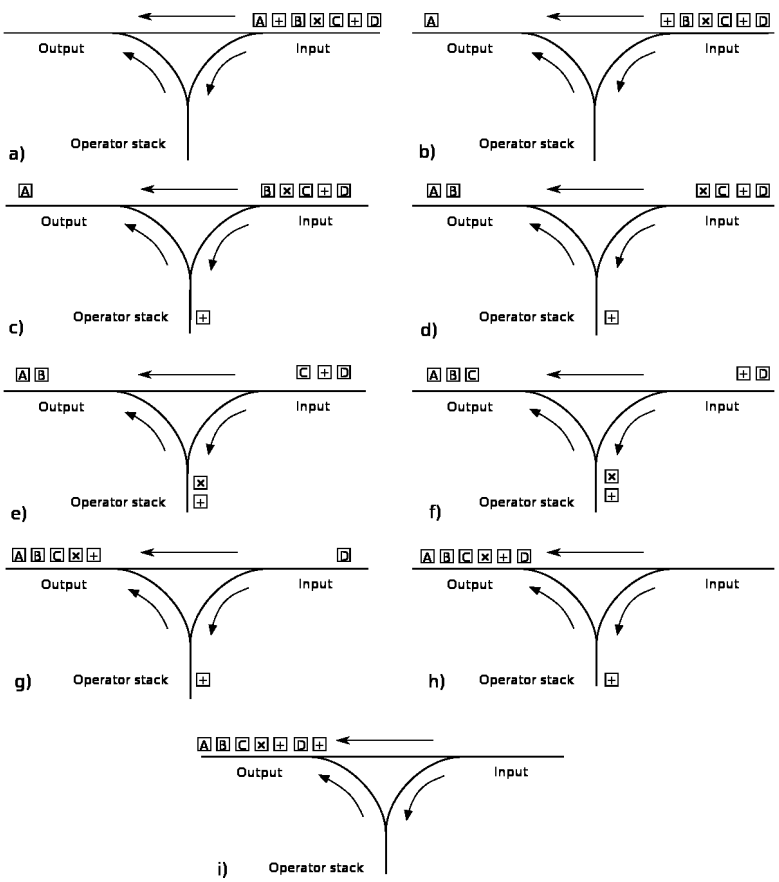
A simple conversion

Input: 3+4

1. Add 3 to the output queue (whenever a number is read it is added to the output)
2. Push + (or its ID) onto the operator stack
3. Add 4 to the output queue
4. After reading the expression pop the operators off the stack and add them to the output.
5. In this case there is only one, "+".
6. Output 3 4 +

This already shows a couple of rules:

- All numbers are added to the output when they are read.
- At the end of reading the expression, pop all operators off the stack and onto the output.



Graphical illustration of algorithm, using a three way railroad junction. The input is processed one symbol at a time, if a variable or number is found it is copied direct to the output b), d), f), h). If the symbol is an operator it is pushed onto the operator stack c), e), however, if its precedence is less than that of the operator at the top of the stack or the precedences are equal and the operator is left associative then that operator is popped off the stack and added to the output g). Finally remaining operators are popped off the stack and added to the output.

The algorithm in detail

- While there are tokens to be read:
  - Read a token.
  - If the token is a number, then add it to the output queue.
  - If the token is a function token, then push it onto the stack.
  - If the token is a function argument separator (e.g., a comma):
    - Until the token at the top of the stack is a left parenthesis, pop operators off the stack onto the output queue. If no left parentheses are encountered, either the separator was misplaced or parentheses were mismatched.

- If the token is an operator, o_1, then:
  - while there is an operator token, o_2, at the top of the stack, and

    either o_1 is left-associative and its precedence is less than or equal to that of o_2,

    or o_1 is right-associative and its precedence is less than that of o_2,

    pop o_2 off the stack, onto the output queue;
  - push o_1 onto the stack.
- If the token is a left parenthesis, then push it onto the stack.
- If the token is a right parenthesis:
  - Until the token at the top of the stack is a left parenthesis, pop operators off the stack onto the output queue.
  - Pop the left parenthesis from the stack, but not onto the output queue.
  - If the token at the top of the stack is a function token, pop it onto the output queue.
  - If the stack runs out without finding a left parenthesis, then there are mismatched parentheses.
- When there are no more tokens to read:
  - While there are still operator tokens in the stack:
    - If the operator token on the top of the stack is a parenthesis, then there are mismatched parentheses.
    - Pop the operator onto the output queue.
- Exit.

To analyze the running time complexity of this algorithm, one has only to note that each token will be read once, each number, function, or operator will be printed once, and each function, operator, or parenthesis will be pushed onto the stack and popped off the stack once – therefore, there are at most a constant number of operations executed per token, and the running time is thus O(n) – linear in the size of the input.

The shunting yard algorithm can also be applied to produce prefix notation (also known as polish notation). To do this one would simply start from the beginning of a string of tokens to be parsed and work backwards, and then reversing the output queue (therefore making the output queue an output stack).

Detailed example

Input: 3 + 4 * 2 / (1 - 5) ^ 2 ^ 3

operator	precedence	associativity
^	4	Right
*	3	Left
/	3	Left
+	2	Left
-	2	Left

Token	Action	Output (in RPN)	Operator Stack	Notes
3	Add token to output	3		
+	Push token to stack	3	+	
4	Add token to output	3 4	+	
*	Push token to stack	3 4	* +	* has higher precedence than +
2	Add token to output	3 4 2	* +	
/	Pop stack to output	3 4 2 *	+	/ and * have same precedence
	Push token to stack	3 4 2 *	/ +	/ has higher precedence than +
(	Push token to stack	3 4 2 *	(/ +	
1	Add token to output	3 4 2 * 1	(/ +	
–	Push token to stack	3 4 2 * 1	– (/ +	
5	Add token to output	3 4 2 * 1 5	– (/ +	
)	Pop stack to output	3 4 2 * 1 5 –	(/ +	Repeated until "(" found
	Pop stack	3 4 2 * 1 5 –	/ +	Discard matching parenthesis
^	Push token to stack	3 4 2 * 1 5 –	^ / +	^ has higher precedence than /
2	Add token to output	3 4 2 * 1 5 – 2	^ / +	
^	Push token to stack	3 4 2 * 1 5 – 2	^ ^ / +	^ is evaluated right-to-left
3	Add token to output	3 4 2 * 1 5 – 2 3	^ ^ / +	
end	Pop entire stack to output	3 4 2 * 1 5 – 2 3 ^ ^ / +		

If you were writing an interpreter, this output would be tokenized and written to a compiled file to be later interpreted. Conversion from infix to RPN can also allow for easier simplification of expressions. To do this, act like you are solving the RPN expression, however, whenever you come to a variable its value is null, and whenever an operator has a null value, it and its parameters are written to the output (this is a simplification, problems arise when the parameters are operators). When an operator has no null parameters its value can simply be written to the output. This method obviously doesn't include all the simplifications possible: It's more of a constant folding optimization.

C example

```
#include <string.h>
#include <stdio.h>
#define bool int
#define false 0
#define true 1

// operators
// precedence   operators       associativity
// 1            !               right to left
// 2            * / %           left to right
// 3            + -             left to right
// 4            =                right to left
int op_preced(const char c)
{
    switch(c)     {
```

```
        case '!':
            return 4;
        case '*':  case '/': case '%':
            return 3;
        case '+': case '-':
            return 2;
        case '=':
            return 1;
    }
    return 0;
}

bool op_left_assoc(const char c)
{
    switch(c)    {
        // left to right
        case '*': case '/': case '%': case '+': case '-':
            return true;
        // right to left
        case '=': case '!':
            return false;
    }
    return false;
}

unsigned int op_arg_count(const char c)
{
    switch(c)  {
        case '*': case '/': case '%': case '+': case '-': case '=':
            return 2;
        case '!':
            return 1;
      default:
           return c - 'A';
    }
    return 0;
}

#define is_operator(c)  (c == '+' || c == '-' || c == '/' || c == '*'
|| c == '!' || c == '%' || c == '=')
#define is_function(c)  (c >= 'A' && c <= 'Z')
#define is_ident(c)     ((c >= '0' && c <= '9') || (c >= 'a' && c <= 'z'))

bool shunting_yard(const char *input, char *output)
{
    const char *strpos = input, *strend = input + strlen(input);
    char c, *outpos = output;
```

```
    char stack[32];       // operator stack
    unsigned int sl = 0;  // stack length
    char     sc;          // used for record stack element

    while(strpos < strend)   {
        // read one token from the input stream
        c = *strpos;
        if(c != ' ')    {
            // If the token is a number (identifier), then add it to
the output queue.
            if(is_ident(c))  {
                *outpos = c; ++outpos;
            }
            // If the token is a function token, then push it onto the
stack.
            else if(is_function(c))   {
                stack[sl] = c;
                ++sl;
            }
            // If the token is a function argument separator (e.g., a
comma):
            else if(c == ',')   {
                bool pe = false;
                while(sl > 0)   {
                    sc = stack[sl - 1];
                    if(sc == '(')  {
                        pe = true;
                        break;
                    }
                    else  {
                        // Until the token at the top of the stack is a
 left parenthesis,
                        // pop operators off the stack onto the output
queue.
                        *outpos = sc;
                        ++outpos;
                        sl--;
                    }
                }
                // If no left parentheses are encountered, either the
separator was misplaced
                // or parentheses were mismatched.
                if(!pe)   {
                    printf("Error: separator or parentheses
mismatched\n");
                    return false;
```

```
                }
            }
            // If the token is an operator, op1, then:
            else if(is_operator(c))  {
                while(sl > 0)    {
                    sc = stack[sl - 1];
                    // While there is an operator token, o2, at the top
 of the stack
                    // op1 is left-associative and its precedence is
less than or equal to that of op2,
                    // or op1 is right-associative and its precedence
is less than that of op2,
                    if(is_operator(sc) &&
                        ((op_left_assoc(c) && (op_preced(c) <= op_preced(sc))) ||
                           (!op_left_assoc(c) && (op_preced(c) < op_preced(sc)))))   {
                        // Pop o2 off the stack, onto the output queue;
                        *outpos = sc;
                        ++outpos;
                        sl--;
                    }
                    else   {
                        break;
                    }
                }
                // push op1 onto the stack.
                stack[sl] = c;
                ++sl;
            }
            // If the token is a left parenthesis, then push it onto
the stack.
            else if(c == '(')   {
                stack[sl] = c;
                ++sl;
            }
            // If the token is a right parenthesis:
            else if(c == ')')    {
                bool pe = false;
                // Until the token at the top of the stack is a left
parenthesis,
                // pop operators off the stack onto the output queue
                while(sl > 0)     {
                    sc = stack[sl - 1];
                    if(sc == '(')    {
                        pe = true;
                        break;
                    }
                    else  {
```

```
                        *outpos = sc;
                        ++outpos;
                        sl--;
                    }
                }
                // If the stack runs out without finding a left
parenthesis, then there are mismatched parentheses.
                if(!pe)  {
                    printf("Error: parentheses mismatched\n");
                    return false;
                }
                // Pop the left parenthesis from the stack, but not
onto the output queue.
                sl--;
                // If the token at the top of the stack is a function
token, pop it onto the output queue.
                if(sl > 0)   {
                    sc = stack[sl - 1];
                    if(is_function(sc))   {
                        *outpos = sc;
                        ++outpos;
                        sl--;
                    }
                }
            }
            else  {
                printf("Unknown token %c\n", c);
                return false; // Unknown token
            }
        }
        ++strpos;
    }
    // When there are no more tokens to read:
    // While there are still operator tokens in the stack:
    while(sl > 0)  {
        sc = stack[sl - 1];
        if(sc == '(' || sc == ')')   {
            printf("Error: parentheses mismatched\n");
            return false;
        }
        *outpos = sc;
        ++outpos;
        --sl;
    }
    *outpos = 0; // Null terminator
    return true;
}
```

```
bool execution_order(const char *input) {
    printf("order: (arguments in reverse order)\n");
    const char *strpos = input, *strend = input + strlen(input);
    char c, res[4];
    unsigned int sl = 0, sc, stack[32], rn = 0;
      // While there are input tokens left
    while(strpos < strend)  {
            // Read the next token from input.
        c = *strpos;
            // If the token is a value or identifier
        if(is_ident(c))    {
                  // Push it onto the stack.
            stack[sl] = c;
            ++sl;
        }
            // Otherwise, the token is an operator  (operator here
includes both operators, and functions).
        else if(is_operator(c) || is_function(c))    {
                  sprintf(res, "_%02d", rn);
                  printf("%s = ", res);
                  ++rn;
                  // It is known a priori that the operator takes n
arguments.
                  unsigned int nargs = op_arg_count(c);
                  // If there are fewer than n values on the stack
                  if(sl < nargs) {
                        // (Error) The user has not input sufficient
values in the expression.
                        return false;
                  }
                  // Else, Pop the top n values from the stack.
                  // Evaluate the operator, with the values as
arguments.
                  if(is_function(c)) {
                        printf("%c(", c);
                        while(nargs > 0)      {
                              sc = stack[sl - 1];
                              sl--;
                              if(nargs > 1)       {
                                    printf("%s, ", &sc);
                              }
                              else {
                                    printf("%s)\n", &sc);
                              }
                              --nargs;
                        }
```

```
                }
                else      {
                      if(nargs == 1) {
                            sc = stack[sl - 1];
                            sl--;
                            printf("%c %s;\n", c, &sc);
                      }
                      else      {
                            sc = stack[sl - 1];
                            sl--;
                            printf("%s %c ", &sc, c);
                            sc = stack[sl - 1];
                            sl--;
                            printf("%s;\n",&sc);
                      }
                }
                // Push the returned results, if any, back onto the
stack.
            stack[sl] = *(unsigned int*)res;
            ++sl;
        }
        ++strpos;
    }
      // If there is only one value in the stack
      // That value is the result of the calculation.
      if(sl == 1) {
            sc = stack[sl - 1];
            sl--;
            printf("%s is a result\n", &sc);
            return true;
      }
      // If there are more values in the stack
      // (Error) The user input has too many values.
      return false;
}

int main() {
    // functions: A() B(a) C(a, b), D(a, b, c) ...
    // identifiers: 0 1 2 3 ... and a b c d e ...
    // operators: = - + / * % !
    const char *input = "a = D(f - b * c + d, !e, g)";
    char output[128];
    printf("input: %s\n", input);
    if(shunting_yard(input, output))    {
        printf("output: %s\n", output);
        if(!execution_order(output))
            printf("\nInvalid input\n");
```

```
    }
    return 0;
}
```

This code produces the following output:

```
input: a = D(f - b * c + d, !e, g)
output: afbc*-d+e!gD=
order: (arguments in reverse order)
_00 = c * b;
_01 = _00 - f;
_02 = d + _01;
_03 = ! e;
_04 = D(g, _03, _02)
_05 = _04 = a;
_05 is a result
```

External links

- Dijkstra's Original description of the Shunting yard algorithm [1]
- Literate Programs implementation in C [2]
- Java Applet demonstrating the Shunting yard algorithm [3]
- Silverlight widget demonstrating the Shunting yard algorithm and evaluation of arithmetic expressions [4]
- Parsing Expressions by Recursive Descent [5] Theodore Norvell © 1999–2001. Access date September 14, 2006.
- Extension to the 'Shunting Yard' algorithm to allow variable numbers of arguments to functions [6]
- A Python implementation of the Shunting yard algorithm [7]

References

[1] http://www.cs.utexas.edu/~EWD/MCReps/MR35.PDF
[2] http://en.literateprograms.org/Shunting_yard_algorithm_(C)
[3] http://www.chris-j.co.uk/parsing.php
[4] http://www.codeding.com/?article=11
[5] http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm
[6] http://www.kallisti.net.nz/blog/2008/02/extension-to-the-shunting-yard-algorithm-to-allow-variable-numbers-of-arguments-to-functions/
[7] http://github.com/ekg/shuntingyard/blob/master/shuntingyard.py

Article Sources and Contributors

Shunting-yard algorithm *Source*: http://en.wikipedia.org/w/index.php?oldid=450298906 *Contributors*: -

Image Sources, Licenses and Contributors

File:Shunting yard.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Shunting_yard.svg *License*: Creative Commons Attribution-Sharealike 3.0 *Contributors*: Salix alba

License