

# Expression Evaluation

Joseph Chuang-Chieh Lin (林莊傑)

Department of Computer Science & Engineering,  
National Taiwan Ocean University

Fall 2024



# Outline

1 Expressions

2 Infix to Postfix

# Outline

① Expressions

② Infix to Postfix

# Expressions

- Example:  $a = (3 * (5 - 2))$  ;
  - Operators (運算子):  $=$ ,  $*$ ,  $-$
  - Operands (運算元):  $a$ , 3, 5, 2
  - Parenthesis (括號):  $(, )$

# Expressions

- Example:

```
((rear+1 == front) || ((rear == MAX_QUEUE_SIZE-1) && !front))
```

- Operators (運算子): ==, +, -, ||, &&, !
- Operands (運算元): rear, front, MAX\_QUEUE\_SIZE
- Parenthesis (括號): (, )

# Why expression evaluation is important?

- $9 + 3 * 5 = ?$ 
  - $9 + 3 * 5 = 24$
  - $9 + 3 * 5 = 60$

# Why expression evaluation is important?

- $9 + 3 * 5 = ?$ 
  - $9 + 3 * 5 = 24$
  - $9 + 3 * 5 = 60$
  
- $9 - 3 - 2 = ?$ 
  - $9 - 3 - 2 = 4$
  - $9 - 3 - 2 = 8$

# Why expression evaluation is important?

- $9 + 3 * 5 = ?$ 
  - $9 + 3 * 5 = 24$
  - $9 + 3 * 5 = 60$
  - Key: precedence rule (優先權)
  
- $9 - 3 - 2 = ?$ 
  - $9 - 3 - 2 = 4$
  - $9 - 3 - 2 = 8$

# Why expression evaluation is important?

- $9 + 3 * 5 = ?$ 
  - $9 + 3 * 5 = 24$
  - $9 + 3 * 5 = 60$
  - Key: precedence rule (優先權)
  
- $9 - 3 - 2 = ?$ 
  - $9 - 3 - 2 = 4$
  - $9 - 3 - 2 = 8$
  - Key: associative rule (關聯性)

# Why expression evaluation is important?

- $9 + 3 * 5 = ?$ 
  - $9 + 3 * 5 = 24$
  - $9 + 3 * 5 = 60$
  - Key: precedence rule (優先權)
  
- $9 - 3 - 2 = ?$ 
  - $9 - 3 - 2 = 4$
  - $9 - 3 - 2 = 8$
  - Key: associative rule (關聯性)

Within any programming language, there is a precedence hierarchy that determines the order in which we evaluate operators.



# Precedence Hierarchy in C

Token	Operator	Precedence <sup>1</sup>	Associativity
0	function call	17	left-to-right
[]	array element		
-> .	struct or union member		
--- ++	increment, decrement <sup>2</sup>	16	left-to-right
--- --	decrement, increment <sup>3</sup>	15	right-to-left
!	logical not		
-	one's complement		
- +	unary minus or plus		
& *	address or indirection		
sizeof	size (in bytes)		
(type)	type cast	14	right-to-left
* / %	multiplicative	13	left-to-right
+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >=	relational	10	left-to-right
< <=			
== !=	equality	9	left-to-right
&	bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
	bitwise or	6	left-to-right
&&	logical and	5	left-to-right
	logical or	4	left-to-right
? :	conditional	3	right-to-left
= += -= /= *= %=	assignment	2	right-to-left
<=> >=& ^=  =			
,	comma	1	left-to-right

- The associativity column indicates how we evaluate operators with the same precedence.

1. The precedence column is taken from Harbison and Steele.

2. Postfix form

3. Prefix form

# Infix & Postfix

Infix	Postfix
$2 + 3 * 4$	$234*+$
$a * b + 5$	$ab*5+$
$(1 + 2 ) * 7$	$12+7*$
$a * b / c$	$ab*c/$
$((a / (b - c + d)) * (e - a) * c$	$abc-d+/ea-*c*$
$a / b - c + d * e - a * c$	$ab/c-de*+ac*-$

# Infix & Postfix

Infix	Postfix
$2 + 3 * 4$	$234*+$
$a * b + 5$	$ab*5+$
$(1 + 2 ) * 7$	$12+7*$
$a * b / c$	$ab*c/$
$((a / (b - c + d)) * (e - a) * c$	$abc-d+/ea-*c*$
$a / b - c + d * e - a * c$	$ab/c-de*+ac*-$

- Infix: the standard way we are used to.

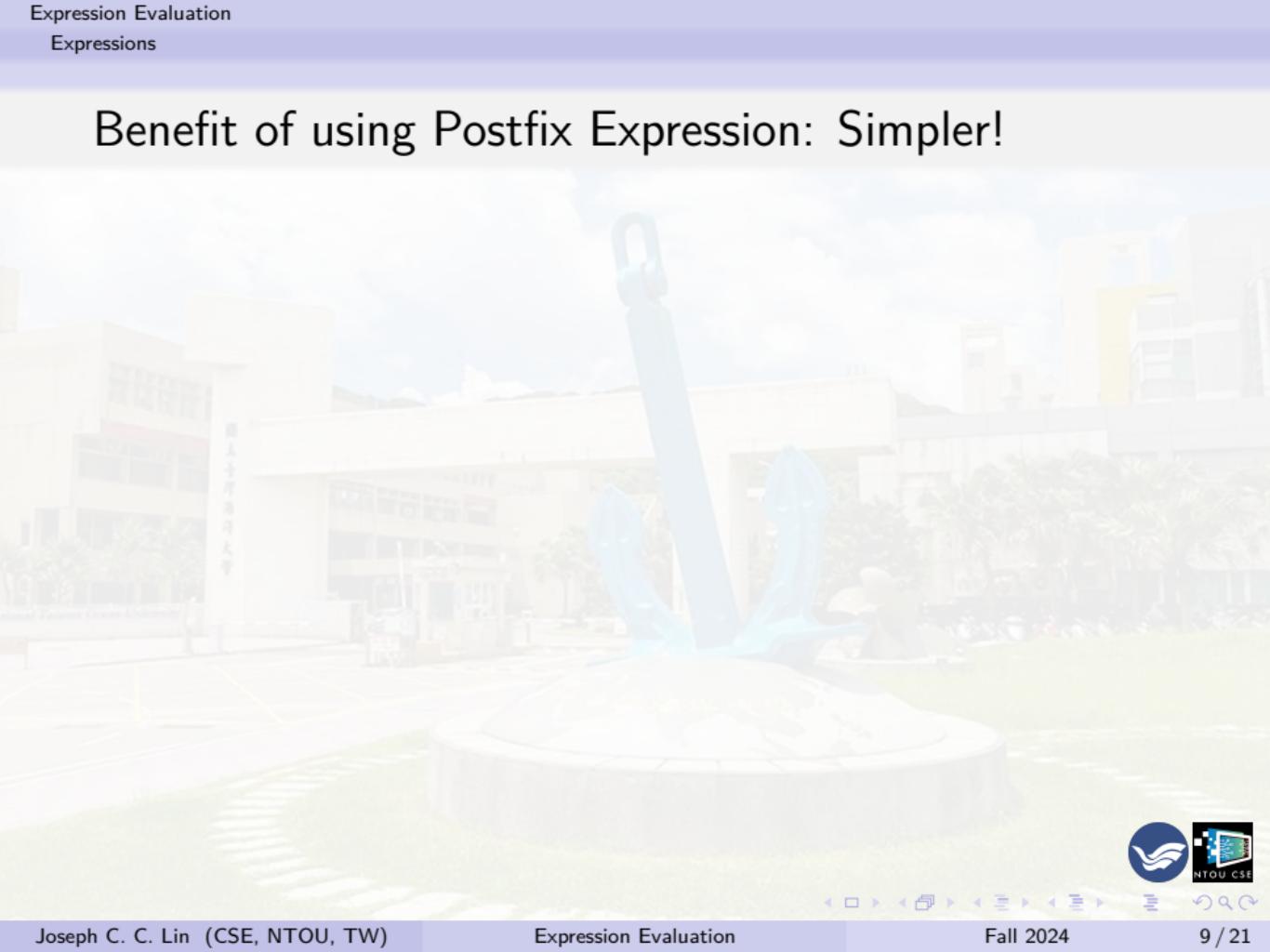


# Infix & Postfix

Infix	Postfix
$2 + 3 * 4$	$234*+$
$a * b + 5$	$ab*5+$
$(1 + 2) * 7$	$12+7*$
$a * b / c$	$ab*c/$
$((a / (b - c + d)) * (e - a) * c$	$abc-d+/ea-*c*$
$a / b - c + d * e - a * c$	$ab/c-de*+ac*-$

- Infix: the standard way we are used to.
- The compilers typically use **postfix**!

# Benefit of using Postfix Expression: Simpler!



# Benefit of using Postfix Expression: Simpler!

- No parenthesis and precedence to consider.



# Benefit of using Postfix Expression: Simpler!

- **No parenthesis and precedence to consider.**
- A single **left-to-right scan** of the expression suffices!

# Benefit of using Postfix Expression: Simpler!

- **No parenthesis and precedence to consider.**
- A single **left-to-right scan** of the expression suffices!
- Evaluation of an expression can be done by using a **stack**.

# Benefit of using Postfix Expression: Simpler!

- No parenthesis and precedence to consider.
- A single left-to-right scan of the expression suffices!
- Evaluation of an expression can be done by using a stack.

Token	Stack			Top
	[0]	[1]	[2]	
6	6			0
2	6	2		1
/	6/2			0
3	6/2	3		1
-	6/2-3			0
4	6/2-3	4		1
2	6/2-3	4	2	2
*	6/2-3	4*2		1
+	6/2-3+4*2			0

# Outline

1 Expressions

2 Infix to Postfix

# Postfix Evaluation

- Expression is represented as a character array.
  - Operators: +, -, \*, / and %.
  - Operands: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.
  - The operands are stored on a stack of type int.
  - The stack is represented by a global array accessed only through top.
- The declarations are:

```
#define MAX_STACK_SIZE 100 // maximum stack size
#define MAX_EXPR_SIZE 100 // max size of expression
typedef enum {
    lparen, rparen, plus, minus, times,
    divide, mod, eos, operand
} precedence;
int stack[MAX_STACK_SIZE]; // global stack
char expr[MAX_EXPR_SIZE]; // input string
```

# To Get Tokens

```
precedence get_token(char *symbol, int *n) { // get the next token,
    *symbol = expr[(*n)++];
    switch (*symbol) {
        case '(':    return lparen;
        case ')':   return rparen;
        case '+':   return plus;
        case '-':   return minus;
        case '/':   return divide;
        case '*':   return times;
        case '%':   return mod;
        case '\0':  return eos; // end of string
        default:    return operand; /* no error checking,
                                         default: operand */
    }
}
```

```

int eval(void)
{
    /* evaluate a postfix expression, expr, maintained as a
    global variable. '\0' is the end of the expression.
    The stack and top of the stack are global variables.
    get_token is used to return the tokentype and
    the character symbol. Operands are assumed to be single
    character digits */
    precedence token;
    char symbol;
    int op1, op2;
    int n = 0; /* counter for the expression string */
    int top = -1;
    token = get_token(&symbol, &n);
    while (token != eos) {
        if (token == operand)
            push(symbol-'0');           /* stack insert */
        else {                         transform ASCII characters into numbers
            /* remove two operands, perform operation, and
            return result to the stack */
            op2 = pop();              /*stack delete */
            op1 = pop();
            switch(token) {
                case plus: push(op1+op2);
                break;
                case minus: push(op1-op2);
                break;
                case times: push(op1*op2);
                break;
                case divide: push(op1/op2);
                break;
                case mod: push(op1%op2);
            }
            token = get_token(&symbol, &n); → get next token
        }
        return pop();                  /* return result */
    }
}

```



# ASCII Code Table

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0 000	000	<b>NUL</b> (null)	32	20 040	000	&#32;	<b>Space</b>	64	40 100	000	&#64;	<b>Ø</b>	96	60 140	000	&#96;	<b>'</b>
1	1 001	001	<b>SOH</b> (start of heading)	33	21 041	001	&#33;	<b>!</b>	65	41 101	001	&#65;	<b>A</b>	97	61 141	001	&#97;	<b>a</b>
2	2 002	002	<b>STX</b> (start of text)	34	22 042	002	&#34;	<b>"</b>	66	42 102	002	&#66;	<b>B</b>	98	62 142	002	&#98;	<b>b</b>
3	3 003	003	<b>ETX</b> (end of text)	35	23 043	003	&#35;	<b>#</b>	67	43 103	003	&#67;	<b>C</b>	99	63 143	003	&#99;	<b>c</b>
4	4 004	004	<b>EOT</b> (end of transmission)	36	24 044	004	&#36;	<b>\$</b>	68	44 104	004	&#68;	<b>D</b>	100	64 144	004	&#100;	<b>d</b>
5	5 005	005	<b>ENQ</b> (enquiry)	37	25 045	005	&#37;	<b>%</b>	69	45 105	005	&#69;	<b>E</b>	101	65 145	005	&#101;	<b>e</b>
6	6 006	006	<b>ACK</b> (acknowledge)	38	26 046	006	&#38;	<b>&amp;</b>	70	46 106	006	&#70;	<b>F</b>	102	66 146	006	&#102;	<b>f</b>
7	7 007	007	<b>BEL</b> (bell)	39	27 047	007	&#39;	<b>'</b>	71	47 107	007	&#71;	<b>G</b>	103	67 147	007	&#103;	<b>g</b>
8	8 010	010	<b>BS</b> (backspace)	40	28 050	010	&#40;	<b>{</b>	72	48 110	010	&#72;	<b>H</b>	104	68 150	010	&#104;	<b>h</b>
9	9 011	011	<b>TAB</b> (horizontal tab)	41	29 051	011	&#41;	<b>}</b>	73	49 111	011	&#73;	<b>I</b>	105	69 151	011	&#105;	<b>i</b>
10	A 012	012	<b>LF</b> (NL line feed, new line)	42	2A 052	012	&#42;	<b>*</b>	74	4A 112	012	&#74;	<b>J</b>	106	6A 152	012	&#106;	<b>j</b>
11	B 013	013	<b>VT</b> (vertical tab)	43	2B 053	013	&#43;	<b>+</b>	75	4B 113	013	&#75;	<b>K</b>	107	6B 153	013	&#107;	<b>k</b>
12	C 014	014	<b>FF</b> (NP form feed, new page)	44	2C 054	014	&#44;	<b>,</b>	76	4C 114	014	&#76;	<b>L</b>	108	6C 154	014	&#108;	<b>l</b>
13	D 015	015	<b>CR</b> (carriage return)	45	2D 055	015	&#45;	<b>-</b>	77	4D 115	015	&#77;	<b>M</b>	109	6D 155	015	&#109;	<b>m</b>
14	E 016	016	<b>SO</b> (shift out)	46	2E 056	016	&#46;	<b>.</b>	78	4E 116	016	&#78;	<b>N</b>	110	6E 156	016	&#110;	<b>n</b>
15	F 017	017	<b>SI</b> (shift in)	47	2F 057	017	&#47;	<b>/</b>	79	4F 117	017	&#79;	<b>O</b>	111	6F 157	017	&#111;	<b>o</b>
16	10 020	020	<b>DLE</b> (data link escape)	48	30 060	020	&#48;	<b>0</b>	80	50 120	020	&#80;	<b>P</b>	112	70 160	020	&#112;	<b>p</b>
17	11 021	021	<b>DC1</b> (device control 1)	49	31 061	021	&#49;	<b>1</b>	81	51 121	021	&#81;	<b>Q</b>	113	71 161	021	&#113;	<b>q</b>
18	12 022	022	<b>DC2</b> (device control 2)	50	32 062	022	&#50;	<b>2</b>	82	52 122	022	&#82;	<b>R</b>	114	72 162	022	&#114;	<b>r</b>
19	13 023	023	<b>DC3</b> (device control 3)	51	33 063	023	&#51;	<b>3</b>	83	53 123	023	&#83;	<b>S</b>	115	73 163	023	&#115;	<b>s</b>
20	14 024	024	<b>DC4</b> (device control 4)	52	34 064	024	&#52;	<b>4</b>	84	54 124	024	&#84;	<b>T</b>	116	74 164	024	&#116;	<b>t</b>
21	15 025	025	<b>NAK</b> (negative acknowledge)	53	35 065	025	&#53;	<b>5</b>	85	55 125	025	&#85;	<b>U</b>	117	75 165	025	&#117;	<b>u</b>
22	16 026	026	<b>SYN</b> (synchronous idle)	54	36 066	026	&#54;	<b>6</b>	86	56 126	026	&#86;	<b>V</b>	118	76 166	026	&#118;	<b>v</b>
23	17 027	027	<b>ETB</b> (end of trans. block)	55	37 067	027	&#55;	<b>7</b>	87	57 127	027	&#87;	<b>W</b>	119	77 167	027	&#119;	<b>w</b>
24	18 030	030	<b>CAN</b> (cancel)	56	38 070	030	&#56;	<b>8</b>	88	58 130	030	&#88;	<b>X</b>	120	78 170	030	&#120;	<b>x</b>
25	19 031	031	<b>EM</b> (end of medium)	57	39 071	031	&#57;	<b>9</b>	89	59 131	031	&#89;	<b>Y</b>	121	79 171	031	&#121;	<b>y</b>
26	1A 032	032	<b>SUB</b> (substitute)	58	3A 072	032	&#58;	<b>:</b>	90	5A 132	032	&#90;	<b>Z</b>	122	7A 172	032	&#122;	<b>z</b>
27	1B 033	033	<b>ESC</b> (escape)	59	3B 073	033	&#59;	<b>:</b>	91	5B 133	033	&#91;	<b>[</b>	123	7B 173	033	&#123;	<b>(</b>
28	1C 034	034	<b>FS</b> (file separator)	60	3C 074	034	&#60;	<b>&lt;</b>	92	5C 134	034	&#92;	<b>\</b>	124	7C 174	034	&#124;	<b> </b>
29	1D 035	035	<b>GS</b> (group separator)	61	3D 075	035	&#61;	<b>=</b>	93	5D 135	035	&#93;	<b>]</b>	125	7D 175	035	&#125;	<b>)</b>
30	1E 036	036	<b>RS</b> (record separator)	62	3E 076	036	&#62;	<b>&gt;</b>	94	5E 136	036	&#94;	<b>^</b>	126	7E 176	036	&#126;	<b>~</b>
31	1F 037	037	<b>US</b> (unit separator)	63	3F 077	037	&#63;	<b>?</b>	95	5F 137	037	&#95;	<b>_</b>	127	7F 177	037	&#127;	<b>DEL</b>

Source: [www.LookupTables.com](http://www.LookupTables.com)

# The First Algorithm

- Consider the following example:

$$a / b - c + d * e - a * c$$

# The First Algorithm

- Consider the following example:

$$a / b - c + d * e - a * c$$

- fully parenthesize the expression. (將運算式加上括號).
  - $((((a / b) - c) + (d * e)) - (a * c))$

# The First Algorithm

- Consider the following example:

$$a / b - c + d * e - a * c$$

- fully parenthesize the expression. (將運算式加上括號).
  - $((((a / b) - c) + (d * e)) - (a * c))$
- move all binary operators so that they replace their corresponding right parentheses. (將運算符號取代其相對應的右括號)
  - $((((a b /c - (d e * + a c * -$

# The First Algorithm

- Consider the following example:

$$a / b - c + d * e - a * c$$

- fully parenthesize the expression. (將運算式加上括號).
  - $((((a / b) - c) + (d * e)) - (a * c))$
- move all binary operators so that they replace their corresponding right parentheses. (將運算符號取代其相對應的右括號)
  - $((((a b / c - (d e * + a c * -$
- delete all parentheses.
  - $a b / c - d e * + a c * -$

# The Second Algorithm

- Scan string from left to right.
- Operands are taken out immediately.
- Operators are taken out of the stack as long as their in-stack precedence (isp) is  $\geq$  the incoming precedence (icp) of the new operator.
- If the token is the right parenthesis ')', unstack tokens until we reach the corresponding left parenthesis '('.

# Algorithm 2 (Example 1)

$a + b * c$

Token	Stack			top	output
	[0]	[1]	[2]		
a				-1	a
+	+			0	a
b	+			0	ab
*	+	*		1	ab
c	+	*		1	abc
eos				-1	abc*+

op	isp	icp
(	0	20
)	19	19
+	12	12
-	12	12
*	13	13
/	13	13
%	13	13
eos	0	0

# Algorithm 2 (Example 2)

$$a * (b + c) * d$$

token	Stack			top	output
	[0]	[1]	[2]		
a				-1	a
*	*			0	a
(	*	(		1	a
b	*	(		1	ab
+	*	(	+	2	ab
c	*	(	+	2	abc
)	*			0	abc+
*	*			0	abc+*
d	*			0	abc+*d
eos				-1	abc+*d*

op	isp	icp
(	0	20
)	19	19
+	12	12
-	12	12
*	13	13
/	13	13
%	13	13
eos	0	0

# The Postfix Algorithm

```
void postfix(void)
{
    /* output the postfix of the expression. The expression
       string, the stack, and top are global */
    char symbol;
    precedence token;
    int n = 0;
    int top = 0;    /* place eos on stack */
    stack[0] = eos;
    for (token = get_token(&symbol, &n); token != eos;
         token = get_token(&symbol,&n)) {
        if (token == operand)
            printf("%c",symbol); directly print out the operand
        else if (token == rparen) { If it's the right parenthesis '
            /* unstack tokens until left parenthesis */
            while (stack[top] != lparen)
                print_token(delete(&top));
            delete(&top); /* discard the left parenthesis */
        } delete the left parenthesis '('
        else {
            /* remove and print symbols whose isp is greater
               than or equal to the current token's icp */
            while(isp[stack[top]] >= icp[token])
                print_token(delete(&top));
            add(&top, token);
        }
    }
    while ( (token=delete(&top)) != eos)
        print_token(token);
    printf("\n"); Print out the token when the end of string is reached
}
```

# Time Complexity of the Postfix Algorithm

- Total time:  $\Theta(n)$ .

# Time Complexity of the Postfix Algorithm

- Total time:  $\Theta(n)$ .
  - The number of stacked tokens that get stacked:  $O(n)$
  - The total number of unstacked tokens:  $O(n)$ .
  - $\Omega(n)$ : at least scanning over the input once.

# Discussions

