Expression Evaluation

Joseph Chuang-Chieh Lin (林莊傑)

Department of Computer Science & Engineering,
National Taiwan Ocean University

Fall 2024

Outline

Outline

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[1]	Associativity
() [] -> .	function call array element struct or union member	17	left-to-right
-- ++	increment, decrement[2]	16	left-to-right
-- ++ ! ~ - + & * sizeof	decrement, increment[3] logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(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

1. The precedence column is taken from Harbison and Steele.
2. Postfix form
3. Prefix form

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

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 [0]	[1]	[2]	Top
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 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 */
}
```

36

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

Source: www.LookupTables.com

The First Algorithm

- Consider the following example:

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

The First Algorithm

- Consider the following example:

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

1. 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:

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

1. fully parenthesize the expression. (將運算式加上括號).
   - `((((a / b) - c) + (d * e)) - (a * c))`
2. 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:

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

1. fully parenthesize the expression. (將運算式加上括號).
   - `((((a / b) - c) + (d * e)) - (a * c))`
2. 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 * -`
3. 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 [0]	[1]	[2]	top	output
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 [0]	[1]	[2]	top	output
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);
      else if (token == rparen) {
         /* unstack tokens until left parenthesis */
         while (stack[top] != lparen)
            print_token(delete(&top));
         delete(&top);  /* discard 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");
}
```

directly print out the operand

If it's the right parenthesis ')'

delete the left parenthesis '('

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