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

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 */
    }
}
```


The Algorithm

- Consider the following example:

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

The 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 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 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 * -`

Discussions