
Stack – LiFO

Part 2: Applications

Stack Applications

- Parentheses Matching
 - Infix to Postfix
-

Parentheses Matching(괄호 매칭)

- $((a+b)^*c+d-e)/(f+g)-(h+j)^*(k-l))/(m-n)$
 - 출력 값 (i, j) 는 i 번째 여는 괄호는 j 번째 닫는 괄호와 쌍이라는 것을 의미하고 있다, 그러므로 위의 수식에서는 다음과 같은 괄호 매칭 쌍이 출력된다
 - $(2,6) (1,13) (15,19) (21,25) (27,31) (0,32) (34,38)$
- $(a+b))^*[(c+d)$
 - $(0,4)$
 - right parenthesis at 5 has no matching left parenthesis
 - $(8,12)$
 - left parenthesis at 7 has no matching right parenthesis

Parentheses Matching

■ 알고리즘

- 수식을 왼쪽부터 오른쪽으로 scan 하면서 ...
 - 여는 괄호를 만나면 stack에 PUSH한다
 - 닫는 괄호를 만나면 stack으로부터 POP을 한다
 - 이때 stack이 empty이거나
 - Scan 완료 후에도 stack에 괄호 위치가 남아 있으면 괄호의 mis-matching
-

Example

- $((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)$

2
1
0

15
0

21
0

(2,6)
(1,13)

(15,19)

(21,25)
(27,31)
(0,32)

(34,38)

More Stack Applications

- Parentheses Matching
- Infix to Postfix

Evaluation of Expressions

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

$$a = 4, b = c = 2, d = e = 3$$

Interpretation 1:

$$((4/2)-2)+(3*3)-(4*2)=0 + 8+9=1$$

Interpretation 2:

$$(4/(2-2+3))*(3-4)*2=(4/3)*(-1)*2=-2.66666\dots$$

How to generate the machine instructions corresponding to a given expression?

precedence rule + associative rule

Mathematical Expression

- Infix notation
 - $3 + 4 * 5$
 - reverse Polish notation (RPN) : postfix notation
 - $3\ 4\ 5\ *\ +$
 - Polish notation : prefix notation
 - $+ 3\ * 4\ 5$
 - Infix 연산을 RPN으로 변환하면 stack을 이용하여 연산을 매우 효율적으로 수행할 수 있다
 - Shunting-yard algorithm
 - Stack machine
-

user

compiler

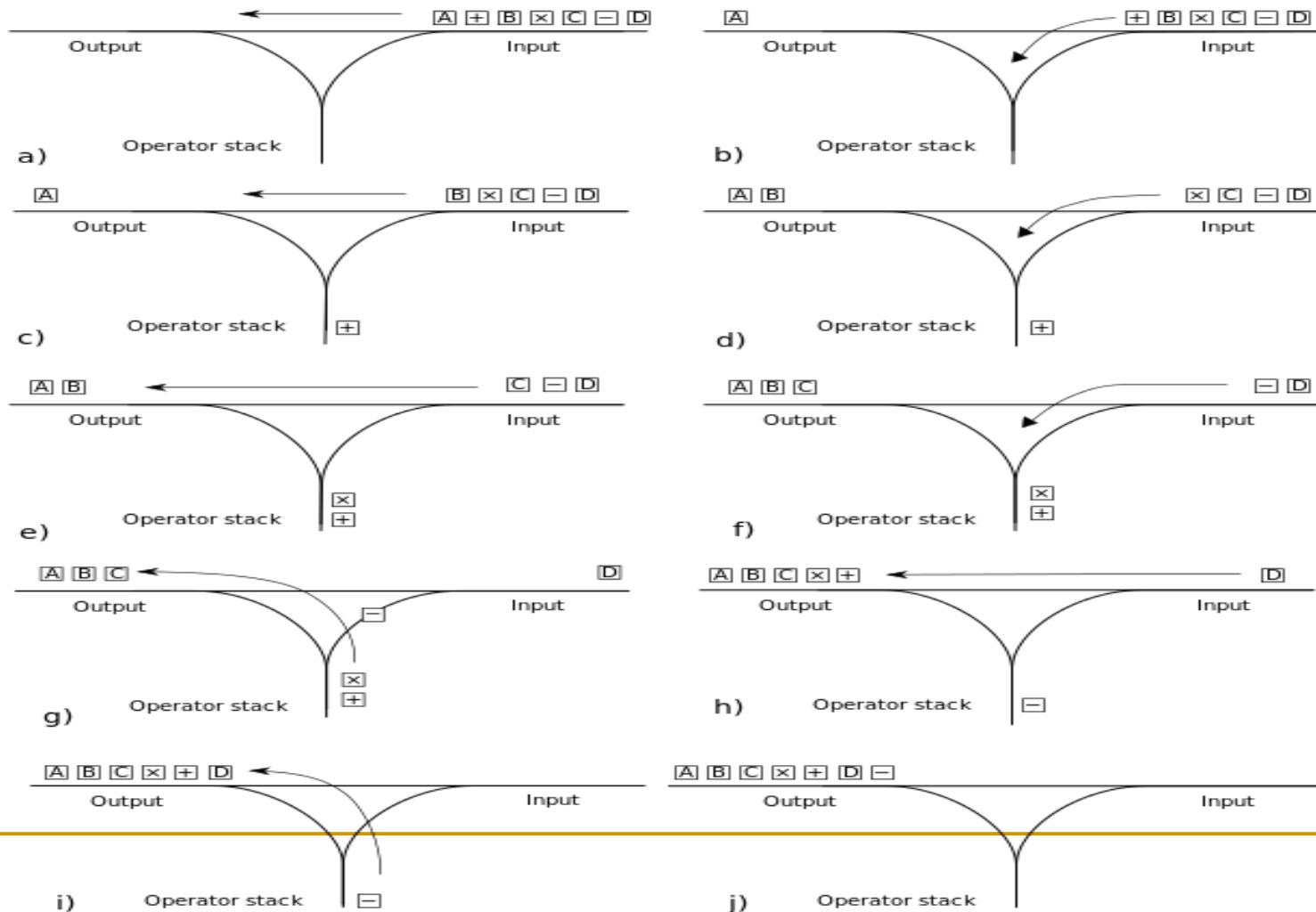
Infix	Postfix
$2 + 3 * 4$	$234 * +$
$a * b + 5$	$ab * 5 +$
$(1 + 2) * 7$	$12 + 7 *$
$a * b / c$	$ab * c /$
$(a / (b - c + d)) * (e - a) * c$	$abc - d + / ea - * c *$
$a / b - c + d * e - a * c$	$ab / c - de * ac * -$

Postfix: no parentheses, no precedence

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

Shunting-Yard Algorithm

- Developed by E. Dijkstra





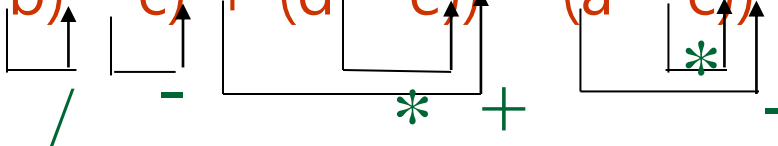
Infix to Postfix Conversion

(Intuitive Algorithm)

- (1) Fully parenthesized expression

$a / b - c + d * e - a * c \rightarrow$
 $((((a / b) - c) + (d * e)) - (a * c))$

- (2) All operators replace their corresponding right parentheses.

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


- (3) Delete all parentheses.

$ab/c-de^*+ac^*-$

two passes

The orders of operands in infix and postfix are the same.

$a + b * c, * > +$

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

$$a * _1 (b + c) * _2 d$$

Token	Stack			Top	Output
	[0]	[1]	[2]		
a				-1	a
* ₁	* ₁			0	a
(* ₁	(1	a
b	* ₁	(1	ab
+	* ₁	(+	2	ab
c	* ₁	(+	2	abc
)	* ₁			0	abc+
* ₂	* ₂			0	abc+* ₁
d	* ₂			0	abc+* ₁ d
eos	* ₂			0	abc+* ₁ d* ₂

Rules

-
- (1) Operators are taken out of the stack as long as their in-stack precedence is higher than or equal to the incoming precedence of the new operator.
- (2) (has low in-stack precedence, and high incoming precedence.

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

Infix to Postfix

Data structure

Assumptions:

operators: +, -, *, /, %

operands: single digit integer

```
#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 */
```

```
static int isp[ ] = {0, 19, 12, 12, 13, 13, 13, 0};    // in-stack precedence
```

```
static int icp[ ] = {20, 19, 12, 12, 13, 13, 13, 0};  // incoming precedence
```

함수 Infix2Postfix

```
void infix2postfix(void)
{
    // 입력 : infix 스트링
    // 출력 : postfix 형식(화면)

    char symbol;
    precedence token;
    int n = 0;
    int sp = 0;          // stack pointer
    stack[sp] = eos;     // 스트링 마지막에 NULL 스트링 삽입

    for (token = get_token(&symbol, &n); token != eos; token = get_token(&symbol, &n))
    {
        if (token == operand)
            printf ("%c", symbol);    // token이 operance이면 화면으로 출력
        else if (token == rparen ) { // token이 ')' 인 경우
```

```
// '('가 나올 때까지 스택의 내용을 화면으로 출력
while (stack[sp] != lparen)
    print_token(delete(&top));
pop(&top);      // '(' 를 스택으로부터 제거
}
else {          // ')' 이 아닌 경우
    // 현재 token의 precedence(icp) 이상의 precedenc를 가지는 symbol(isp)들을
    // 스택으로부터 읽어 화면으로 출력하고 완료후 자신은 스택에 저장
    while(isp[stack[sp]] >= icp[token] )
        print_token(delete(&top));
    push(&top, token);          // 자신을 스택에 저장
}
} // end of for 문
while ((token = pop(&top)) != eos) // 스트링 입력 완료후 스택에 남은 모든
    print_token(token);          // 모든 symbol들을 화면으로 출력
print("\n");
}
```

```
precedence get_token(char *symbol, int *p)
{
```

```
// 스트링으로부터 읽어서 해당 token에 대한 precedence 값을 반환
// precedence는 enum 타입으로 주어진다
```

```
    *symbol =expr[(*p)++];
```

```
    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;
```

```
        default : return operand;
```

```
    }
```

```
}
```

Postfix 표현식의 연산

```
int evalPostfix(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 token type 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 sp = -1;

token = get_token(&symbol, &n);
while (token != eos) {
    if (token == operand)
        push(&sp, symbol-'0'); /* stack insert */
```

```
else { /* remove two operands, perform operation, and return result to the stack */
    op2 = pop(&sp);          /* stack delete */
    op1 = pop(&sp);
    switch(token) {
        case plus: push(&sp, op1+op2); break;
        case minus: push(&sp, op1-op2); break;
        case times: push(&sp, op1*op2); break;
        case divide: push(&sp, op1/op2); break;
        case mod: push(&sp, op1%op2);
    }
}
token = get_token (&symbol, &n);
return pop(&sp); /* return result */
}
```
