Ramaiah Institute of Technology
(Autonomous Institute, Affiliated to VTU)
Department of Computer Science and Engineering

Course Name: Data Structures

Course Code: CS32

Credits: 3:1:0

Term: September – December 2020

Faculty:
Mamatha Jadhav V
Vandana S Sardar

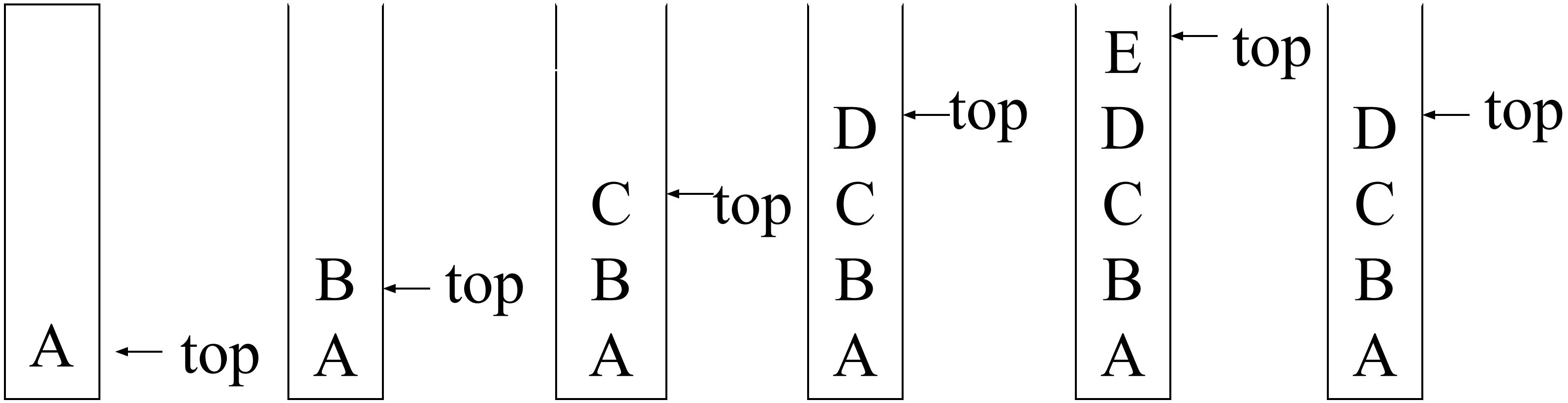
CHAPTER 3

STACKS AND QUEUES

All the programs in this file are selected from

Ellis Horowitz, Sartaj Sahni, and Susan Anderson-Freed
“Fundamentals of Data Structures in C”,
Computer Science Press, 1992.

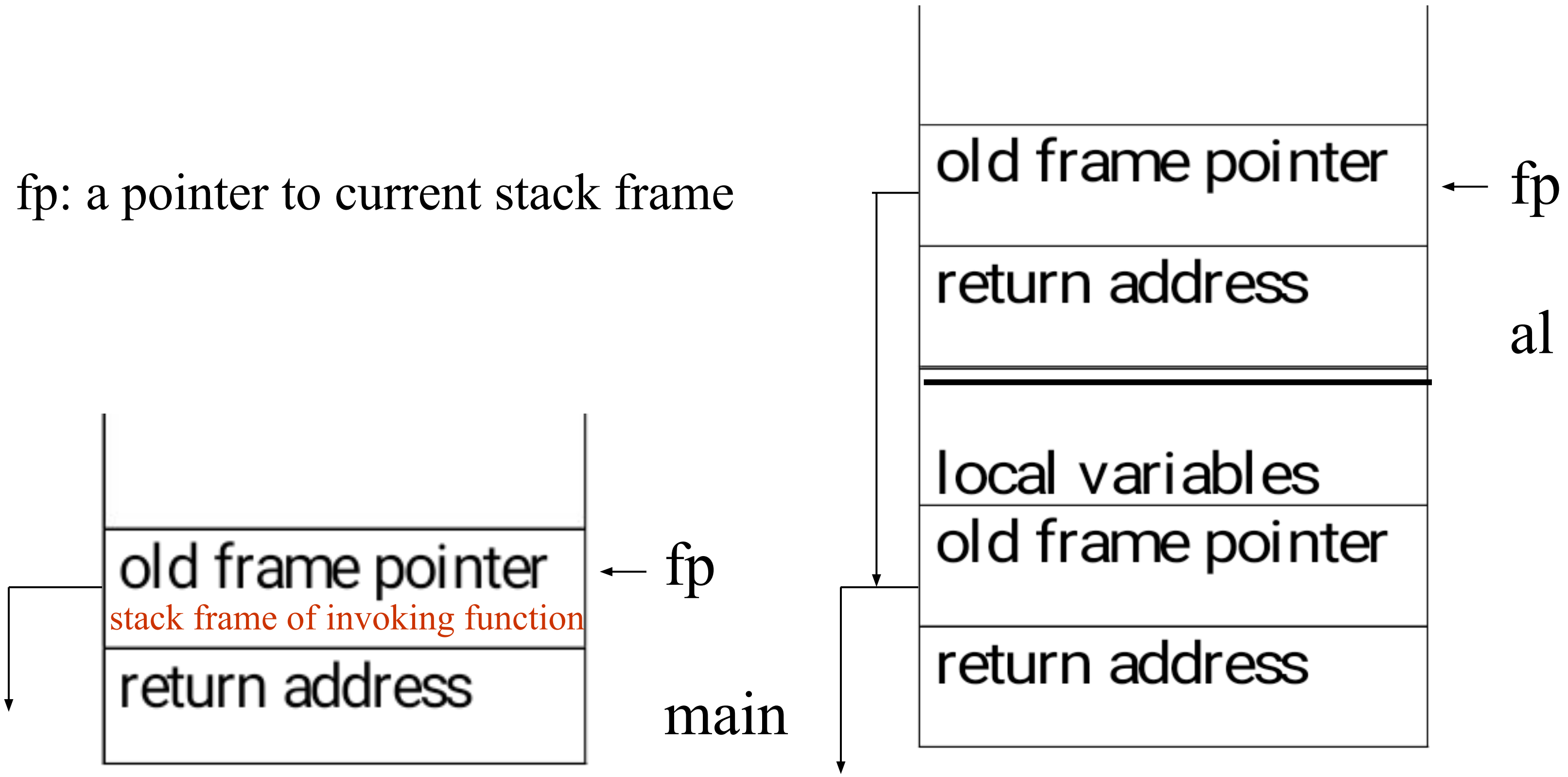
Stack: a Last-In-First-Out (LIFO) list



Inserting and deleting elements in a stack

An application of Stack: stack frame of function call (activation record)

fp: a pointer to current stack frame



system stack before a1 is invoked

(a)

system stack after a1 is invoked

(b)

System stack after function call a1

Abstract data type for stack

```
structure Stack is
  objects: a finite ordered list with zero or more elements.
  functions:
    for all stack ∈ Stack, item ∈ element, max_stack_size  ∈ positive integer
   Stack CreateS(max_stack_size) ::=
             create an empty stack whose maximum size is  max_stack_size
   Boolean IsFull(stack, max_stack_size) ::=
             if (number of elements in stack == max_stack_size)
             return TRUE
             else return FALSE
   Stack Add(stack, item) ::=
             if (IsFull(stack)) stack_full
             else insert item into top of stack and return

    Boolean IsEmpty(stack) ::=
                        if(stack == CreateS(max_stack_size))
                        return TRUE
                        else return FALSE
    Element Delete(stack) ::=
                        if(IsEmpty(stack)) return
                        else remove and return the item on the top of the stack.
```

Implementation: using array

```
Stack CreateS(max_stack_size) ::=
    #define MAX_STACK_SIZE 100 /* maximum stack size */
    typedef struct {
                int key;
                /* other fields */
                } element;
    element stack[MAX_STACK_SIZE];
    int top = -1;

    Boolean IsEmpty(Stack) ::= top< 0;

    Boolean IsFull(Stack) ::= top >= MAX_STACK_SIZE-1;
```

Add to a stack

```
void add(int *top, element item)
{
 /* add an item to the global stack */
     if (*top >= MAX_STACK_SIZE-1)  {
           stack_full( );
           return;
     }
     stack[++*top] = item;
}
```

***program 3.1:** Add to a stack (p.104)

Delete from a stack

```
element delete(int *top)
{
 /* return the top element from the stack */
     if (*top == -1)
         return stack_empty( );  /* returns and error key */
     return stack[(*top)--];
 }
```

Delete from a stack

```
#define MALLOC(x,size,type)(x=(type*)malloc(size*sizeof(type)))
```

Stacks using Dynamic Arrays

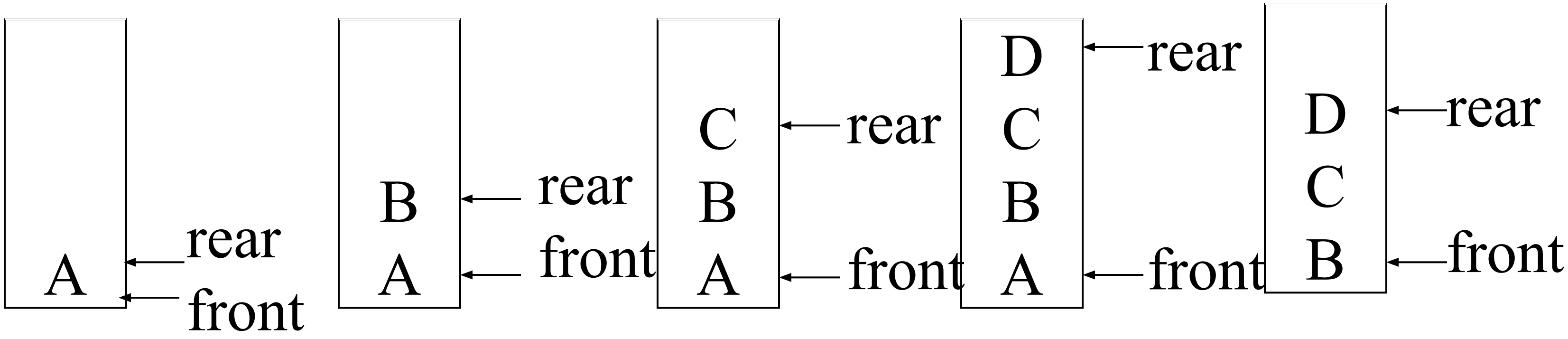
```
typedef struct {
                        int key;
                       }element;

element *stack;
 int capacity=1;
 int top = -1;
void main()
{
    MALLOC(stack, sizeof(*stack), element)
    --------------------
    --------------------
}
```

Note: Alter the code for push function, to test for a full stack (replace max_stack_size with capacity)

```
 void stackfull()
  {
     stack=realloc(stack, 2*capacity*sizeof(element))
     capacity=capacity*2;
}
```

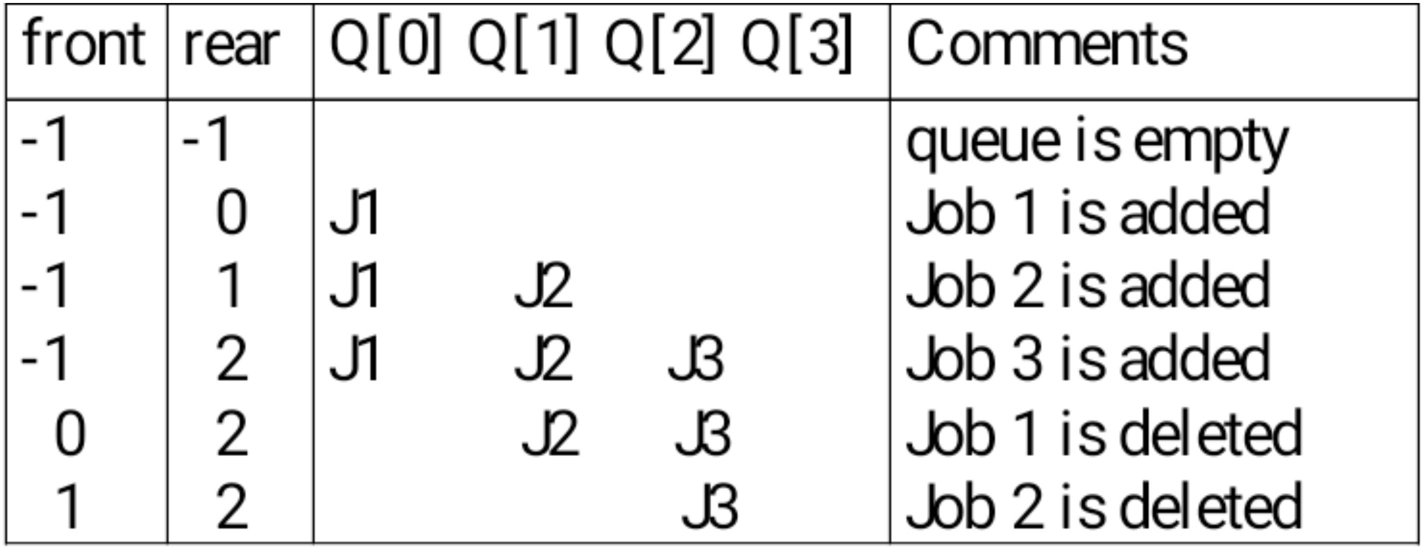
Note: capacity initially =1, then it goes on increasing in terms of $\mathbf{2^n}$

Queue: a First-In-First-Out (FIFO) list



Inserting and deleting elements in a queue

Application: Job scheduling



front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

Insertion and deletion from a sequential queue

Abstract data type of queue

```
structure Queue is
  objects: a finite ordered list with zero or more elements.
  functions:
      for all queue ∈ Queue, item ∈ element, max_ queue_ size ∈ positive integer
    Queue CreateQ(max_queue_size) ::=
          create an empty queue whose maximum size is   max_queue_size
    Boolean IsFullQ(queue, max_queue_size) ::=
          if(number of elements in queue == max_queue_size)
          return TRUE
          else return FALSE
    Queue AddQ(queue, item) ::=
          if (IsFullQ(queue)) queue_full
         else insert item at rear of queue and return queue
Boolean IsEmptyQ(queue) ::=
          if (queue ==CreateQ(max_queue_size))
          return TRUE
          else return FALSE
    Element DeleteQ(queue) ::=
          if (IsEmptyQ(queue)) return
          else remove and return the item at front of queue.
```

Implementation 1: using array

```
Queue CreateQ(max_queue_size) ::=
# define MAX_QUEUE_SIZE 100/* Maximum queue size */
typedef struct {
                int key;
                /* other fields */
                } element;
element queue[MAX_QUEUE_SIZE];
int rear = -1;
int front = -1;
Boolean IsEmpty(queue) ::= front == rear
Boolean IsFullQ(queue) ::= rear == MAX_QUEUE_SIZE-1

void addq(int *rear, element item)
{
/* add an item to the queue */
    if (*rear == MAX_QUEUE_SIZE - 1) {
       queue_full( );
       return;
   }
   queue [++*rear] = item;
}
```

Delete from a queue

```
element deleteq(int *front, int rear)
{
/* remove element at the front of the queue */
    if ( *front == rear)
        return queue_empty( );      /* return an error key */
    return queue [++ *front];
}
```

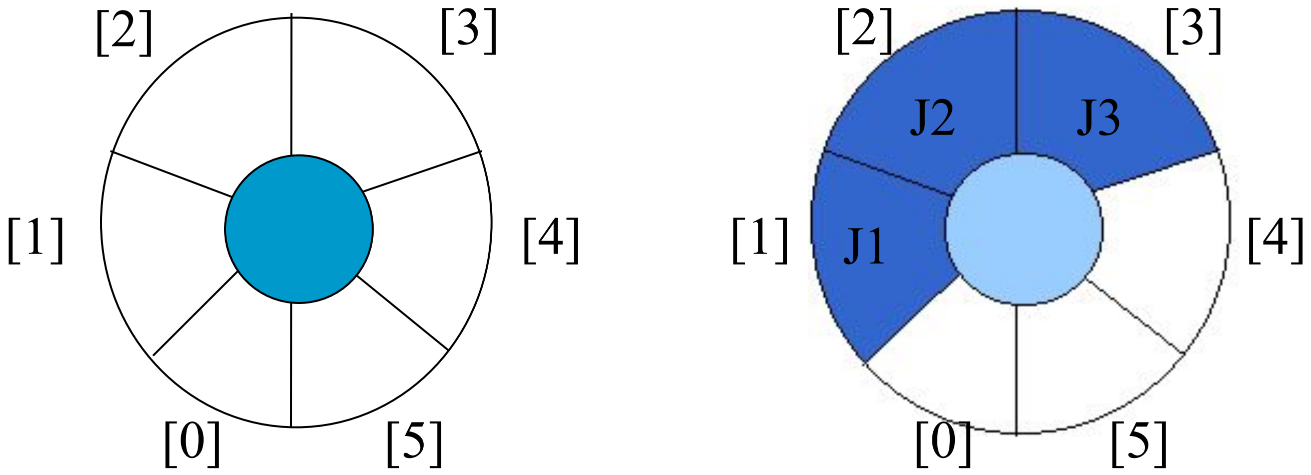
Delete from a queue

problem: there may be available space when IsFullQ is true
movement is required.

Implementation 2: regard an array as a circular queue

front: one position counterclockwise from the first element
rear: current end

EMPTY QUEUE



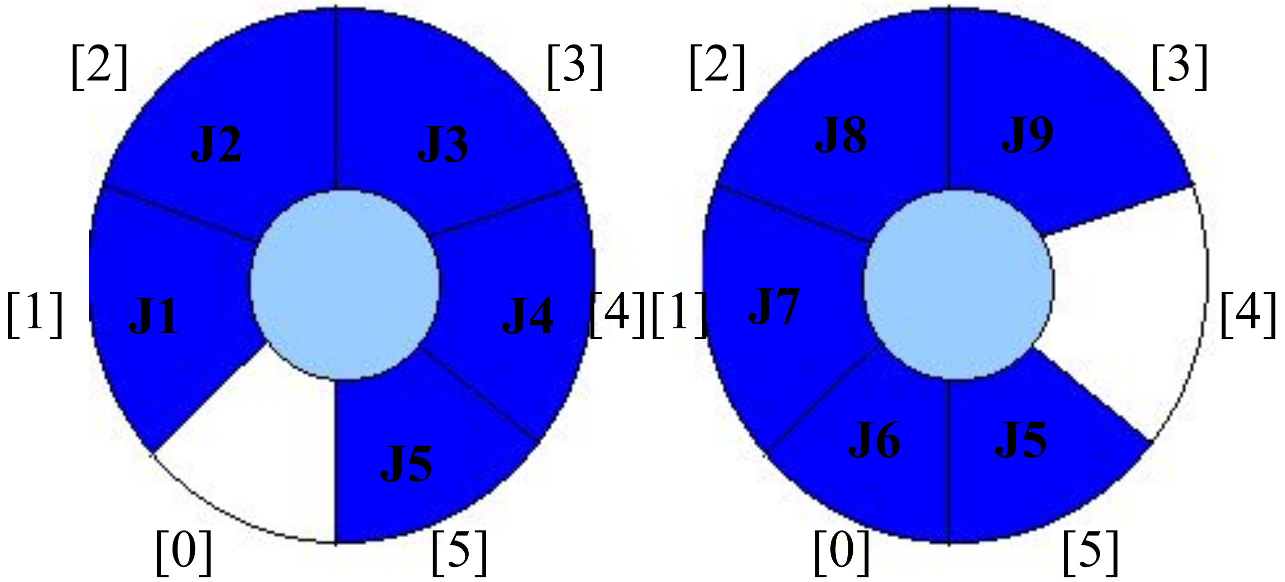
front = 0
rear = 0

front = 0
rear = 3

__Figure 3.6:__ Empty and nonempty circular queues (p.109)

Problem: one space is left when queue is full

FULL QUEUE FULL QUEUE



front =0
rear = 5

front =4
rear =3

<ins>***Figure 3.7:** Full circular queues and then we remove the item (p.110)</ins>

Add to a circular queue

```
void addq(int front, int *rear, element item)
{
/* add an item to the queue */
    *rear = (*rear +1) % MAX_QUEUE_SIZE;
     if (front == *rear) /* reset rear and print error */
     return;
   }
     queue[*rear] = item;
}
```

Add to a circular queue

Delete from a circular queue

```
element deleteq(int* front, int rear)
{
  element item;
  /* remove front element from the queue and put it in item */
     if (*front == rear)
        return queue_empty( );
                 /* queue_empty returns an error key */
     *front = (*front+1) % MAX_QUEUE_SIZE;
     return queue[*front];
}
```

Delete from a circular queue

Add to a dynamic circular queue

```
void addq( element item)
{
/* add an item to the queue */
    rear = (rear +1) %capacity
     if (front == rear) /* reset rear and print error */
    {
    queuefull();
     rear++;
   }
     queue[rear] = item;
}
```

Doubling queue capacity

```
      void queuefull()
      {
         element *newqueue;
         newqueue=(element *) malloc (2*capacity*sizeof(*queue));
         int start =(front+1) % capacity;
         if(start<2) /* no wrap around*/
          copy(queue+start, queue+start+capacity-1, newqueue);
         else
           {
             copy(queue+start,  queue+capacity,  newqueue);
             copy(queue, queue+rear+1, newqueue+capacity-start);
           }
        front = 2*capacity-1;
         rear = capacity – 2;
        capacity = capacity * 2;
         free(queue);
         queue = newqueue;
      }
void copy(element *start, element *end, element *newqueue)
{
    int I;
   element *j;
    j = start;
   for (i=0 ; j<end; j++, i++)
       *(newqueue+i) = *j;
}
```

Function for displaying elements in a circular queue

```
void display()
{
  int i;
  if(front==rear)
   {
    printf(“empty queue”);
    return;
    }
   for( i = front + 1 %  capacity ; i != rear +1 % capacity ; i = i + 1 % capacity )
     printf(“ %d”, queue[i].key);
}
```

Evaluation of Expressions

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

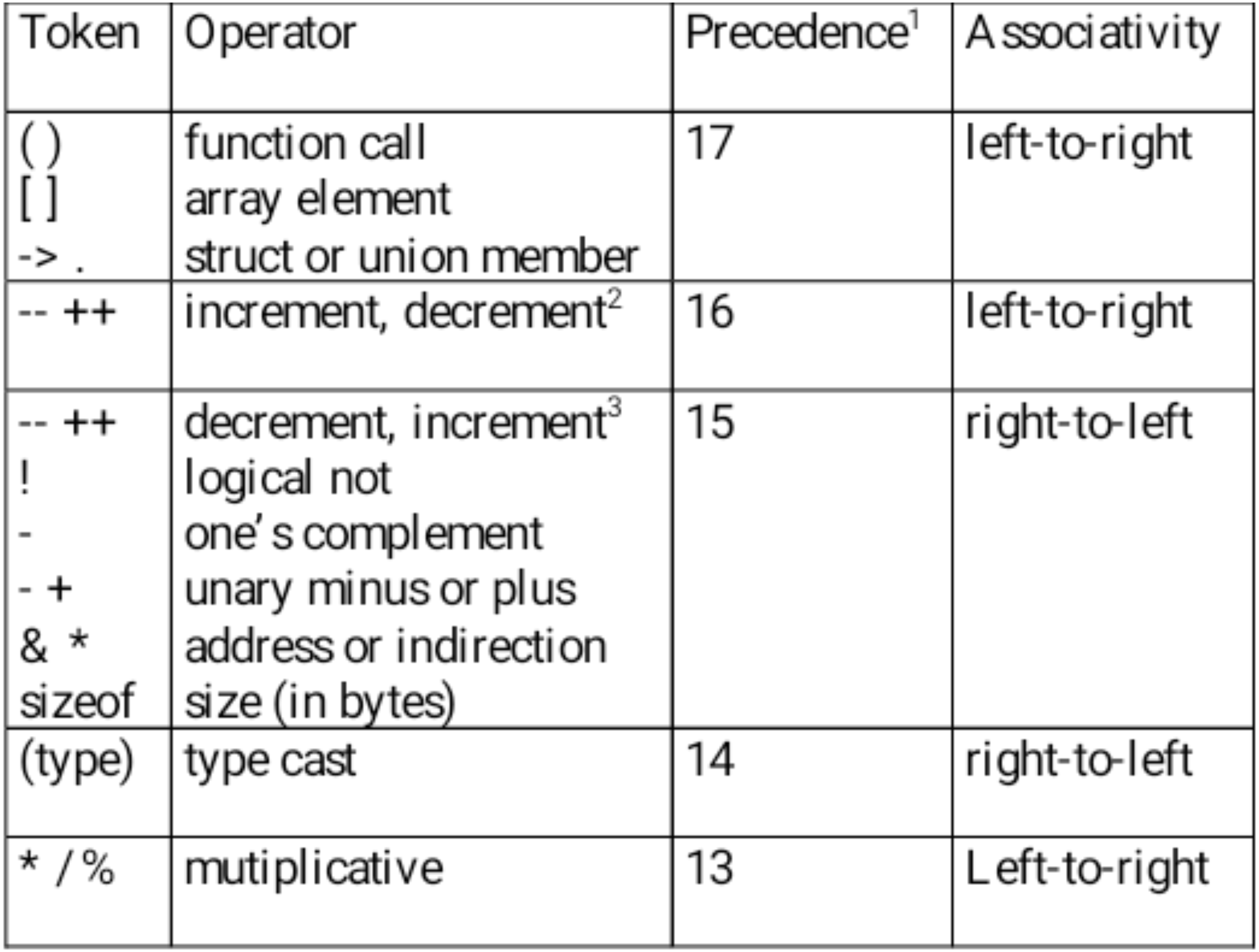
$a = 4, b = c = 2, d = e = 3$

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

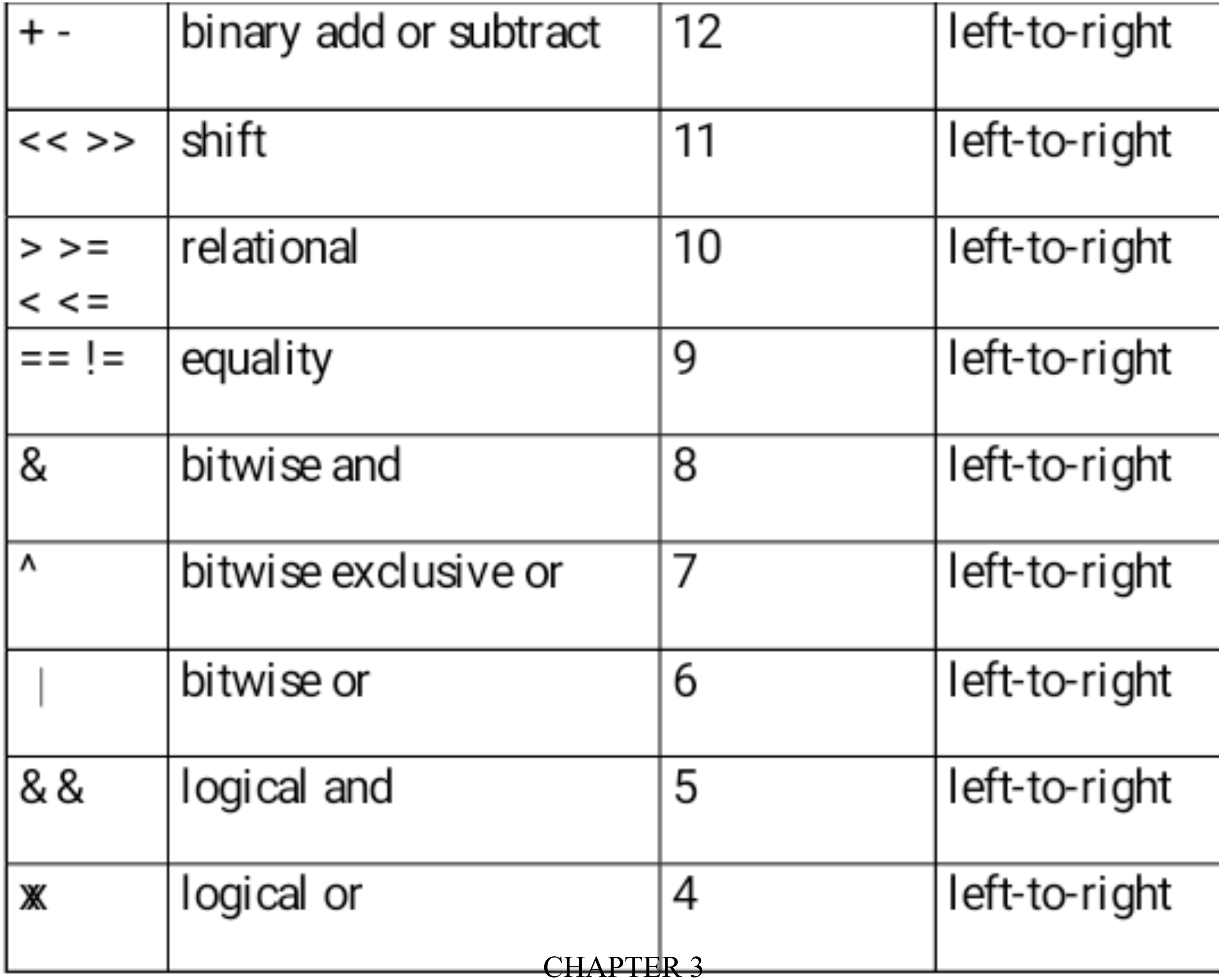
Interpretation 2:
$(4/(2-2+3))*(3-4)*2=(4/3)*(-1)*2=-2.66666\cdots$

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

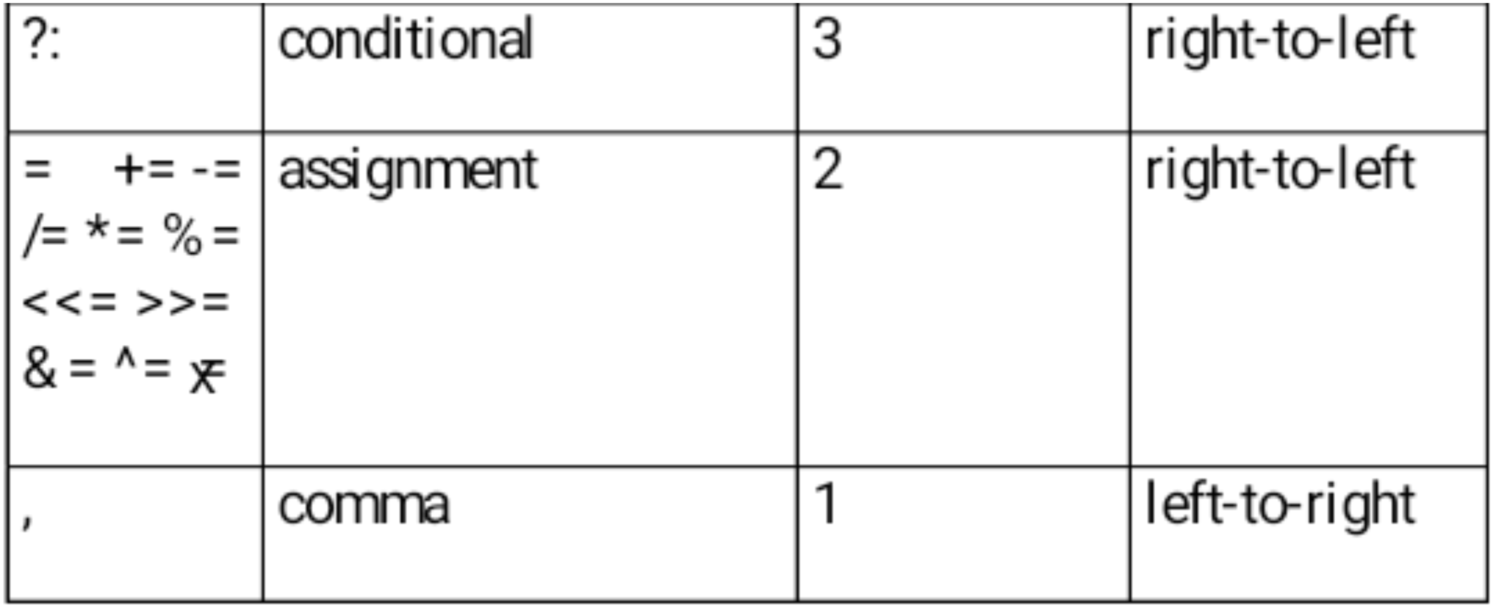
precedence rule + associative rule



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
* / %	mutiplicative	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
xx	logical or	4	left-to-right

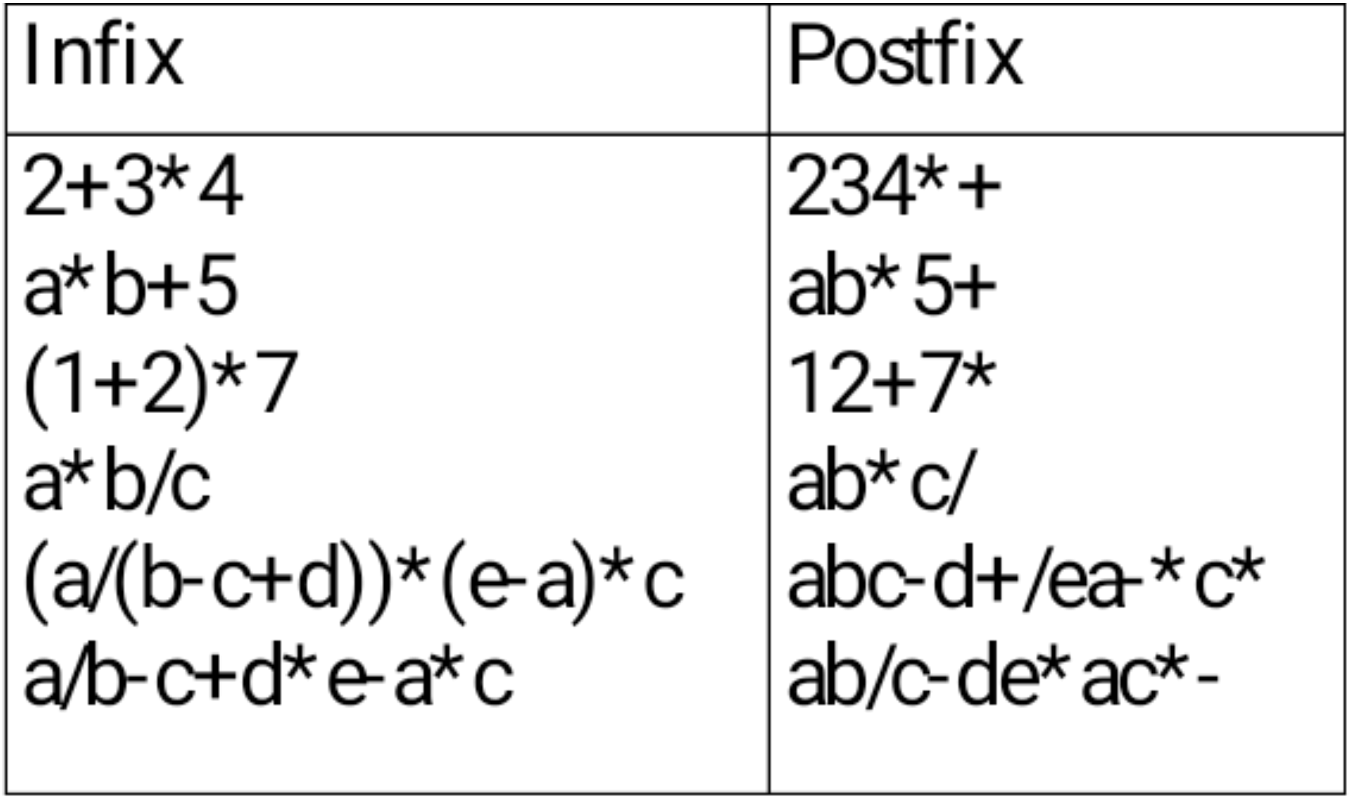


?:	conditional	3	right-to-left
= += -= /= *= %= <<= >>= &= ^= x=	assignment	2	right-to-left
,	comma	1	left-to-right

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

Precedence hierarchy for C

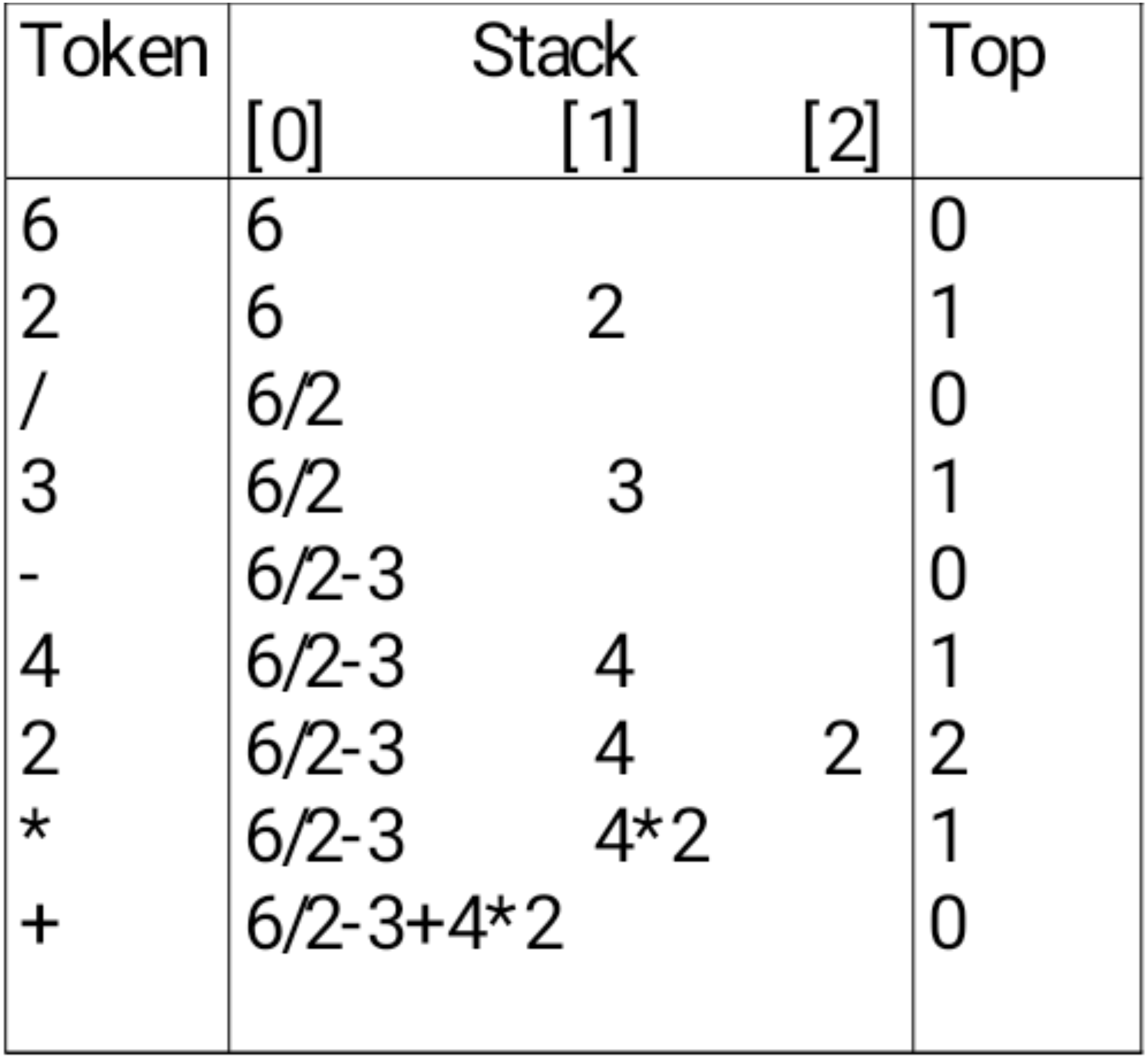
user compiler



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*-

Infix and postfix notation

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

Postfix evaluation

Goal: Evaluation of postfix expression

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{lparan, rparen, plus, minus, times, divide,
                     mod, eos, operand} precedence;
int stack[MAX_STACK_SIZE];      /* global stack */
char expr[MAX_EXPR_SIZE];      /* input string */
```

```
int eval(void)
{
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)
          add(&top, symbol-'0');   /* stack insert */

        else {
        op2 = delete(&top);  /* stack delete */
              op1 = delete(&top);
              switch(token) {
                   case plus: add(&top, op1+op2); break;
                   case minus: add(&top, op1-op2); break;
                   case times: add(&top, op1*op2); break;
                   case divide: add(&top, op1/op2); break;
                   case mod: add(&top, op1%op2);
              }
           }
           token = get_token (&symbol, &n);
        }
        return delete(&top);
       }
```

```
precedence get_token(char *symbol, int *n)
{
/* get the next token, symbol is the character  representation, which is returned, the token is
   represented by its enumerated value, which is returned in the function name */

 *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;
    default  : return operand;
                   /* no error checking, default is operand */
    }
}
```

Infix to Postfix Conversion
(Intuitive Algorithm)

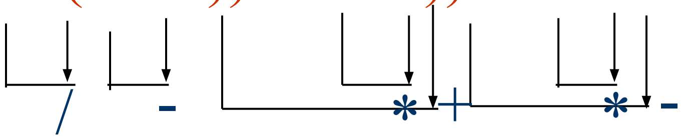
(1) Fully parenthesize expression

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

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

(2) All operators replace their corresponding right parentheses.

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



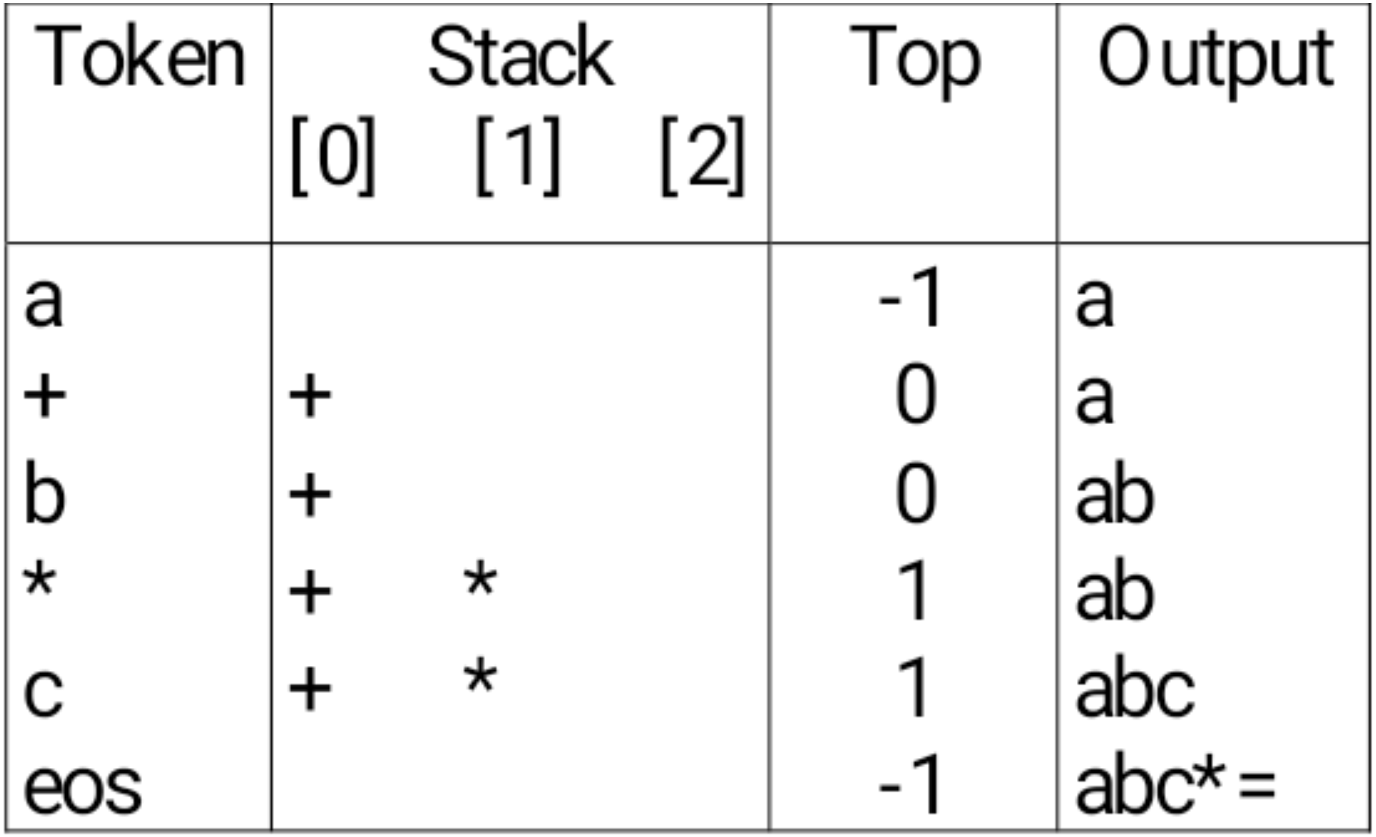
(3) Delete all parentheses.

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*=

***Figure 3.15:** Translation of a+b*c to postfix (p.124)*

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

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
)	$*_1$	match)		0	abc+
$*_2$	$*_2$	$*_1 = *_2$		0	$abc+*_1$
d	$*_2$			0	$abc+*_1d$
eos	$*_2$			0	$abc+*_1d*_2$

Translation of a*(b+c)*d to postfix

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

```
precedence stack[MAX_STACK_SIZE];
/* isp and icp arrays -- index is value of precedence
lparen, rparen, plus, minus, times, divide, mod, eos */
static int isp [ ] = {0, 19, 12, 12, 13, 13, 13, 0};
static int icp [ ] = {20, 19, 12, 12, 13, 13, 13, 0};
```

isp: in-stack precedence
icp: incoming precedence

```
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);
  print(“\n”);
}
```

Function to convert from infix to postfix