Stack Applications

- Parentheses Matching
- Maze Router
- Infix to Postfix

Parentheses Matching

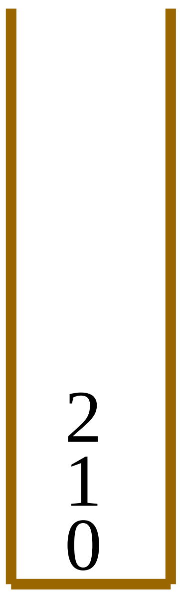
- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)
  - 출력 값 (i, j) 는 i 번째 여는 괄호는 j 번째 닫는 괄호와 쌍이라는 것을 의미하고 있다, 그러므로 위의 수식에서는 다음과 같은 괄호 매칭 쌍이 출력된다
    - (2,6) (1,13) (15,19) (21,25) (27,31) (0,32) (34,38)
- (a+b))*((c+d)
  - (0,4)
  - right parenthesis at 5 has no matching left parenthesis
  - (8,12)
  - left parenthesis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



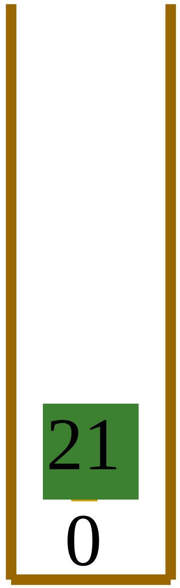
- (((a+b)*c+d-e)/(f+g)-(h+j)*(k-l))/(m-n)

15

0

(2,6) (1,13)

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



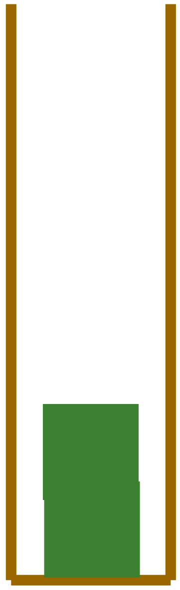
(2,6) (1,13) (15,19)

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

27
0

(2,6) (1,13) (15,19) (21,25)

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



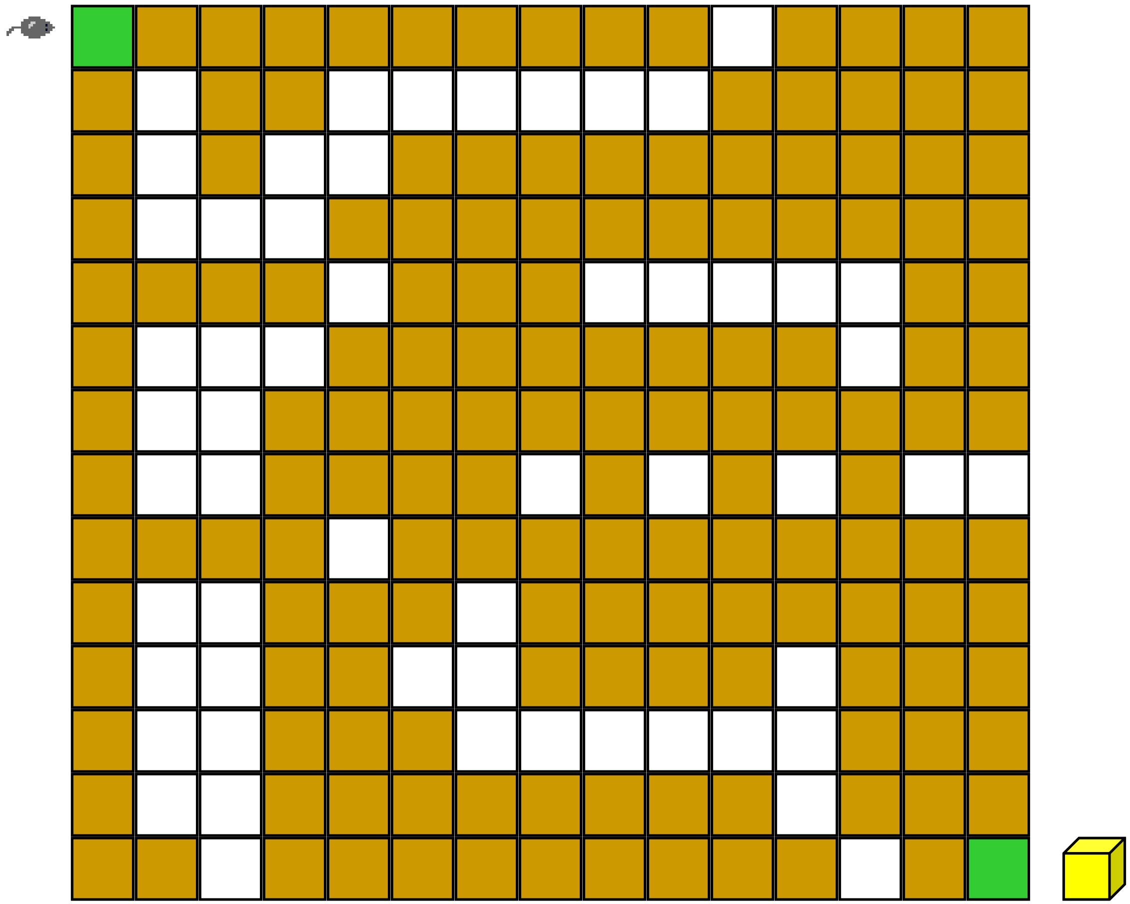
(2,6) (1,13) (15,19) (21,25)(27,31) (0,32)

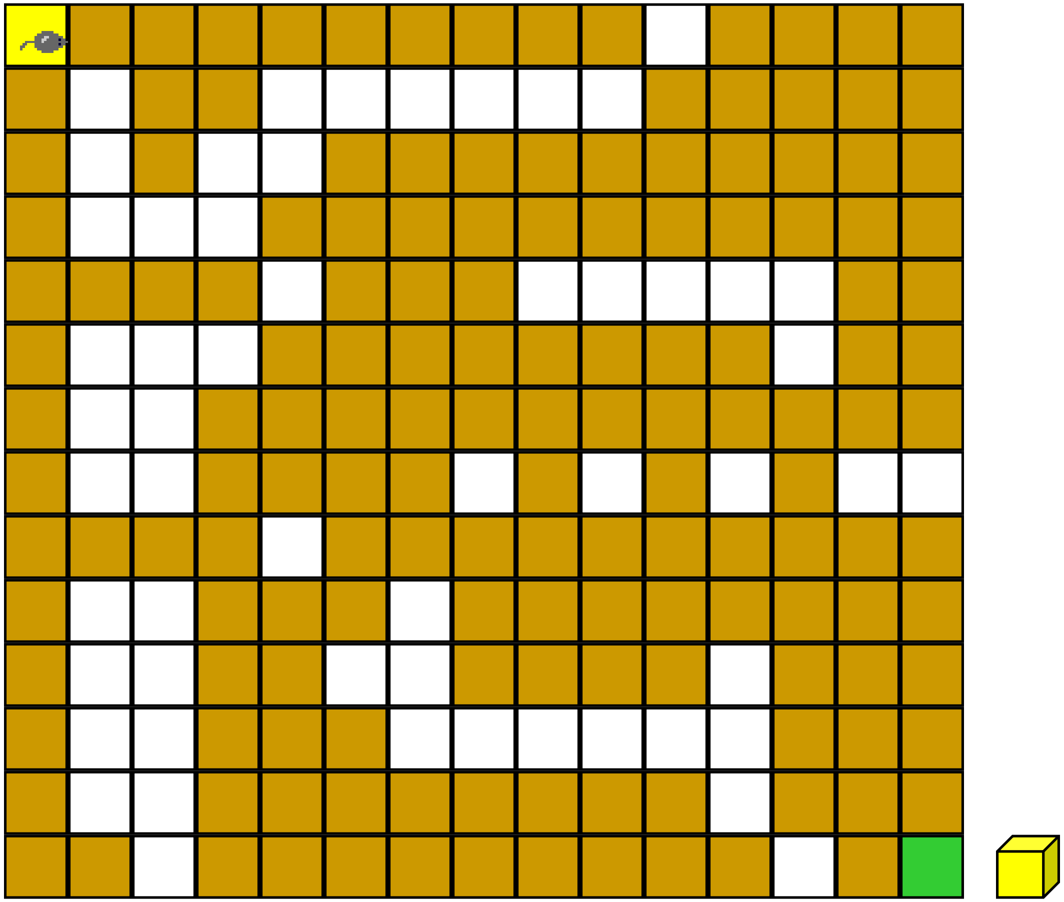
- and so on

More Stack Applications

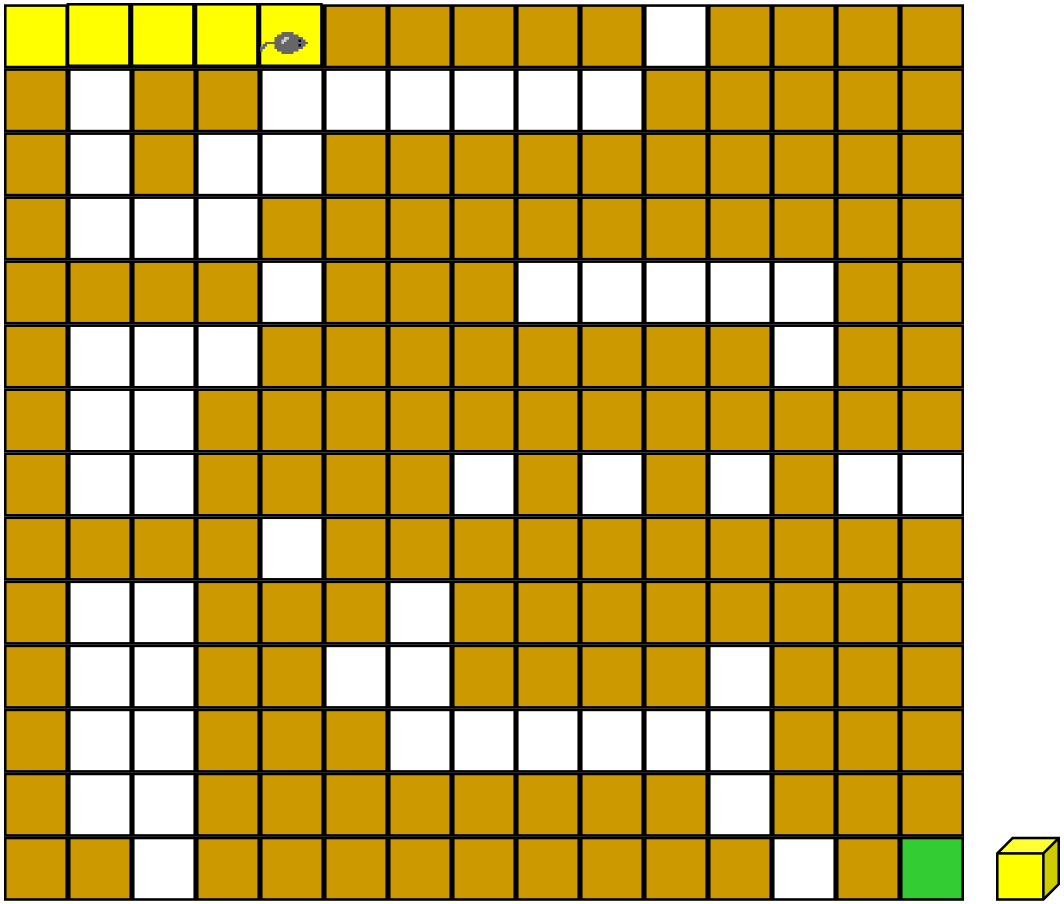
- Parentheses Matching
- **Maze Router**
- Infix to Postfix

Rat In A Maze

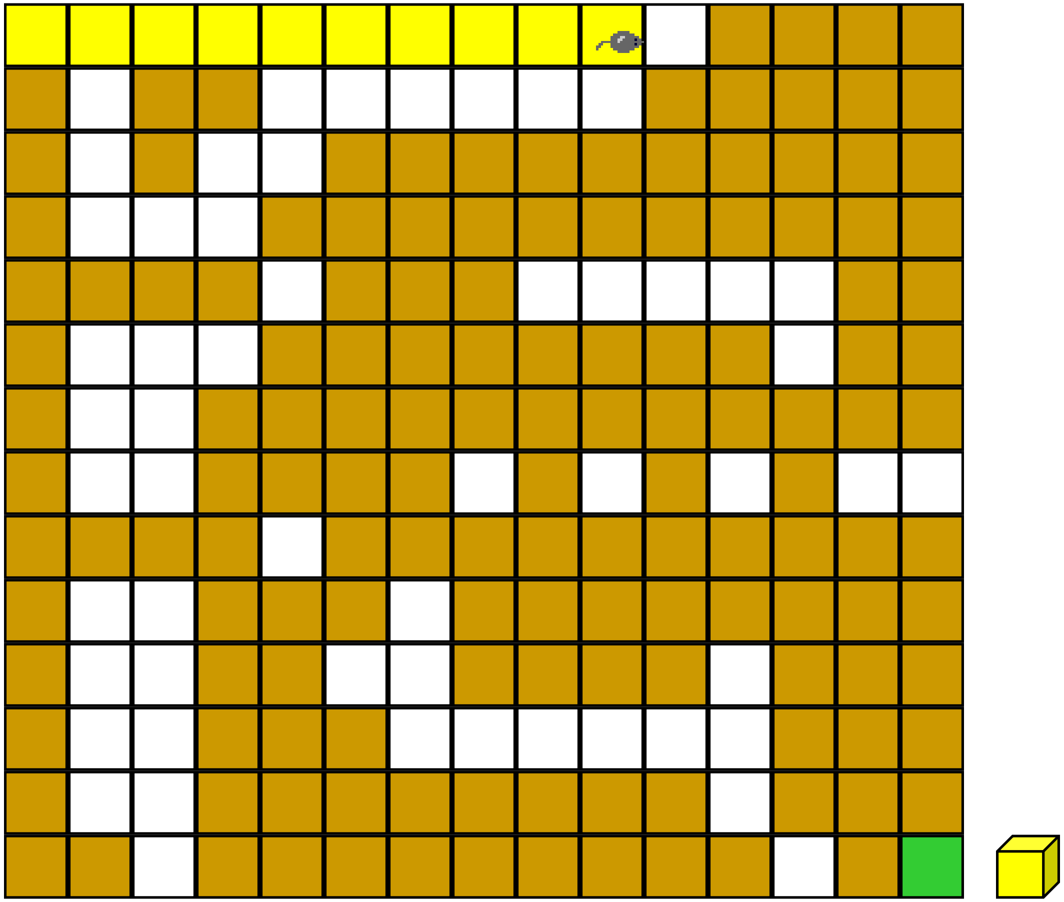




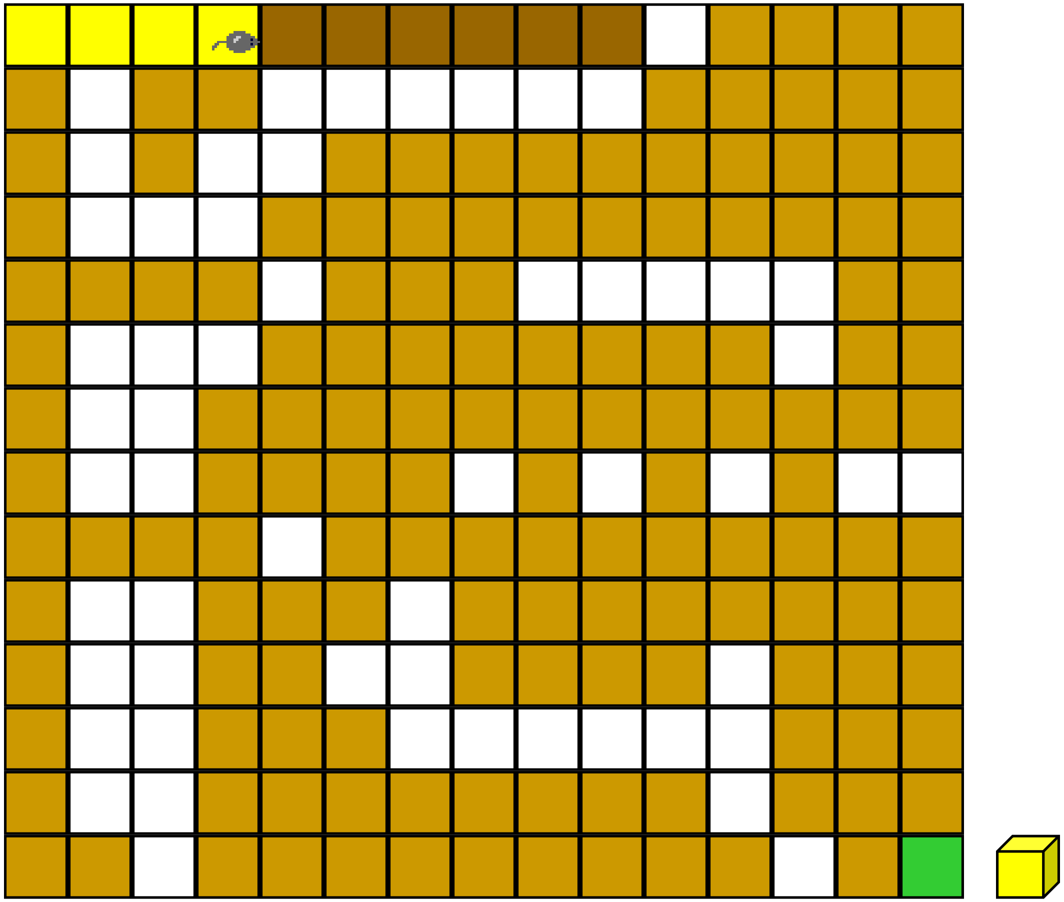
- Move order is: right, down, left, up
- Block positions to avoid revisit.



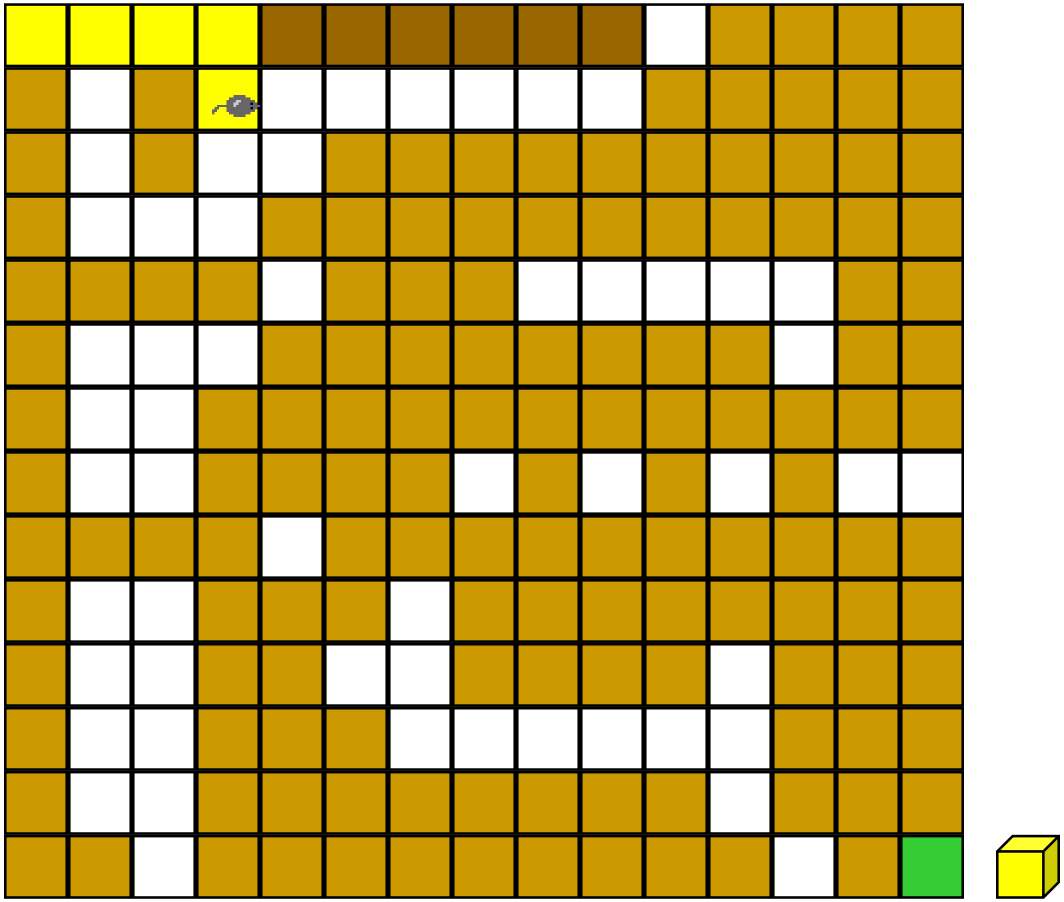
- Move order is: right, down, left, up
- Block positions to avoid revisit.



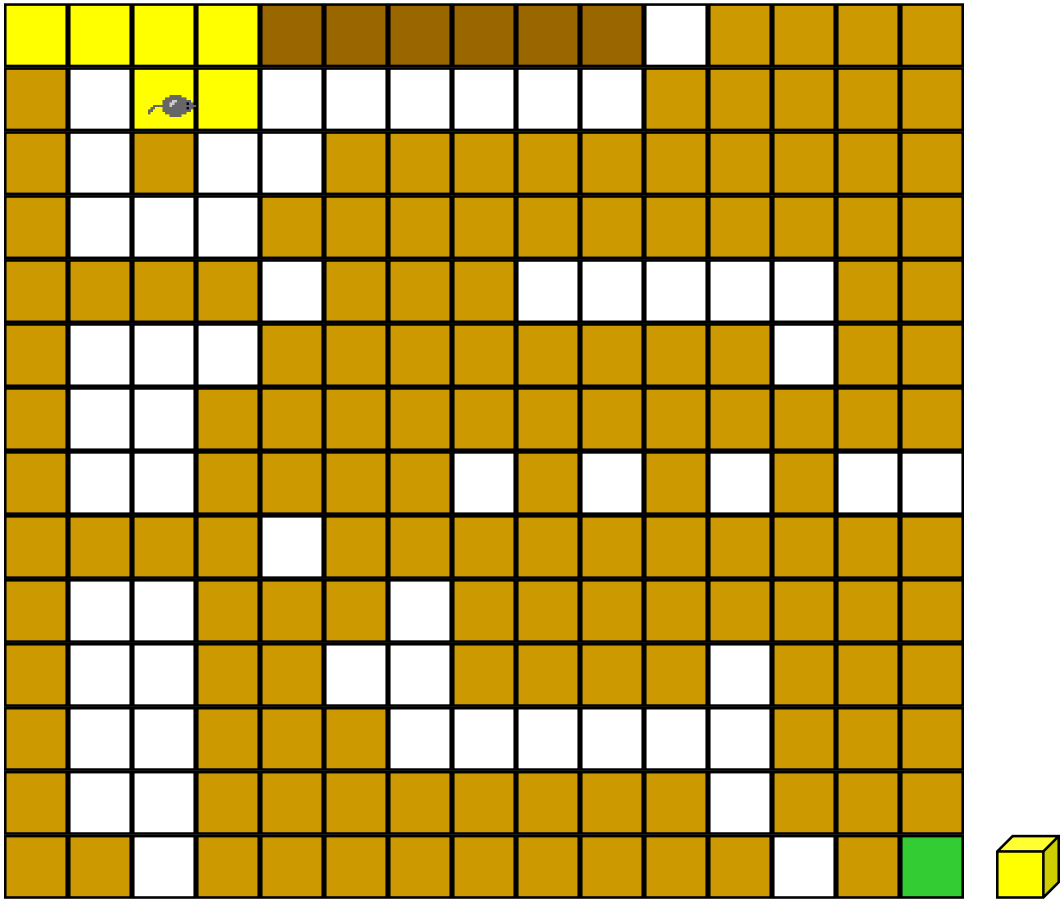
- Move backward until we reach a square from which a forward move is possible.



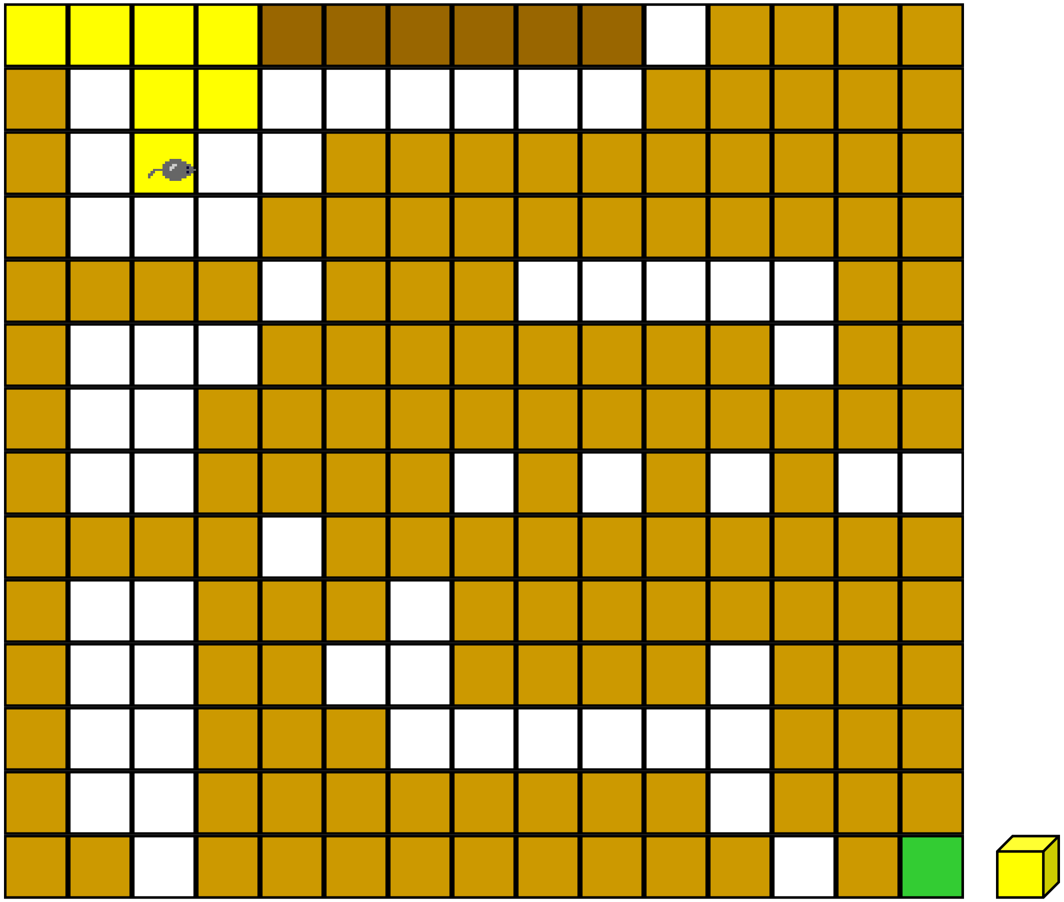
- Move down.



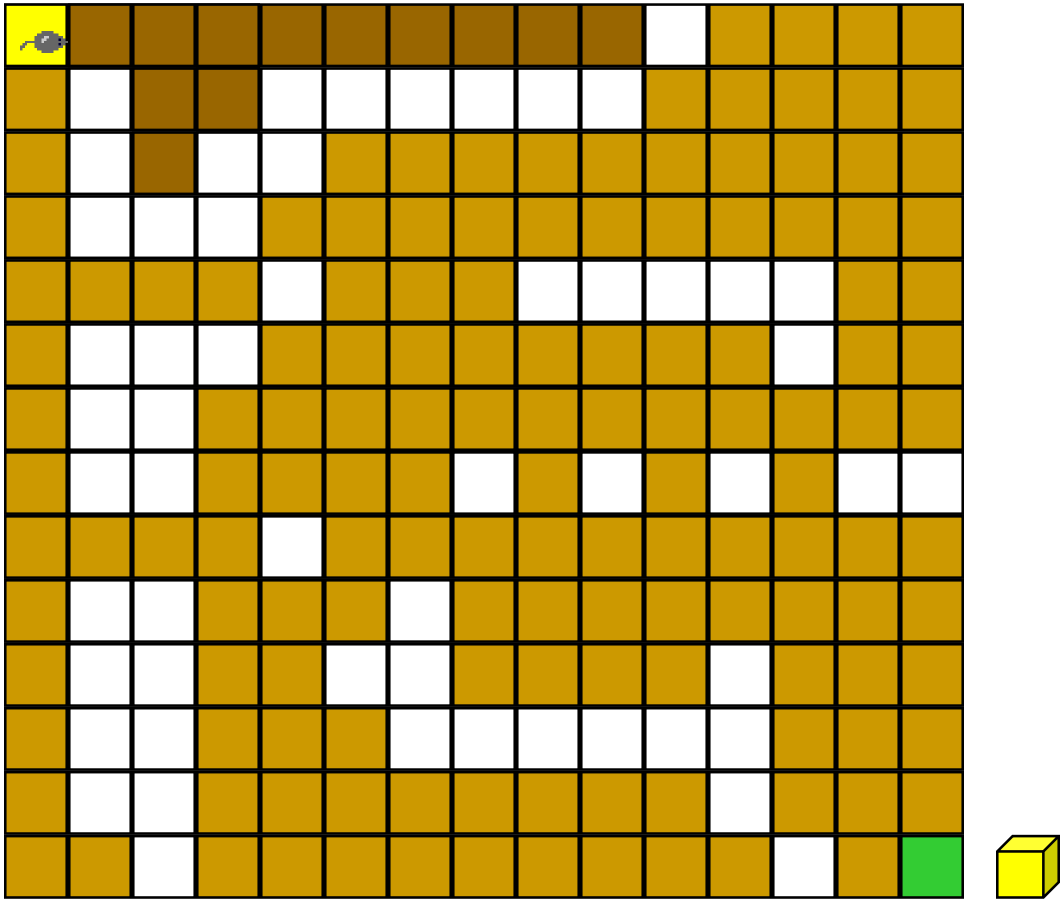
- Move left.



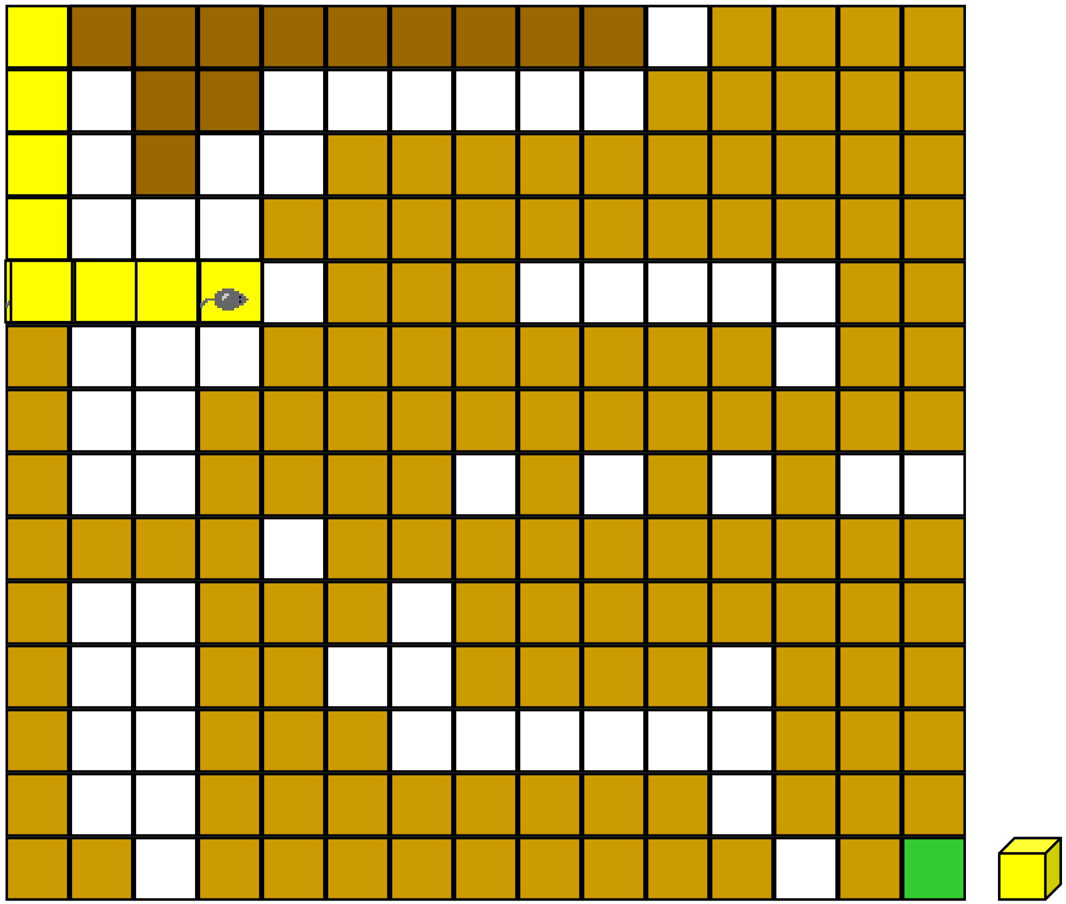
- Move down.



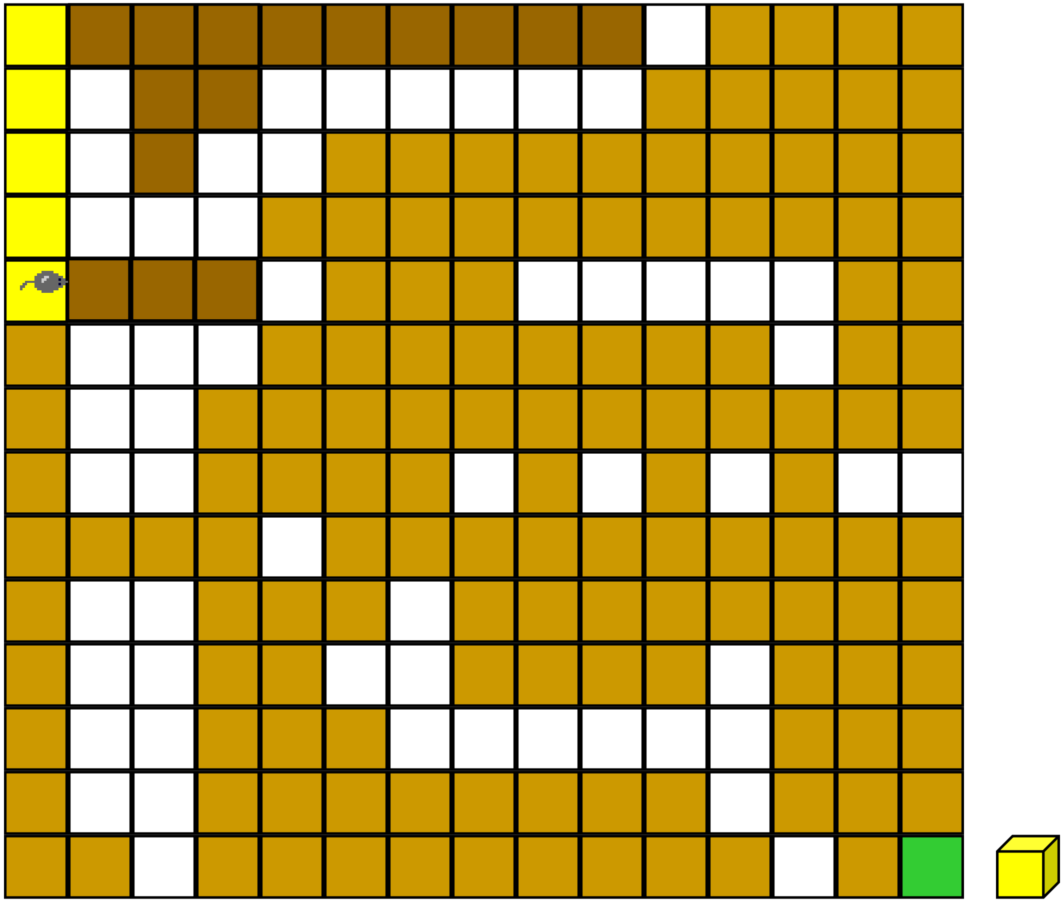
- Move backward until we reach a square from which a forward move is possible.



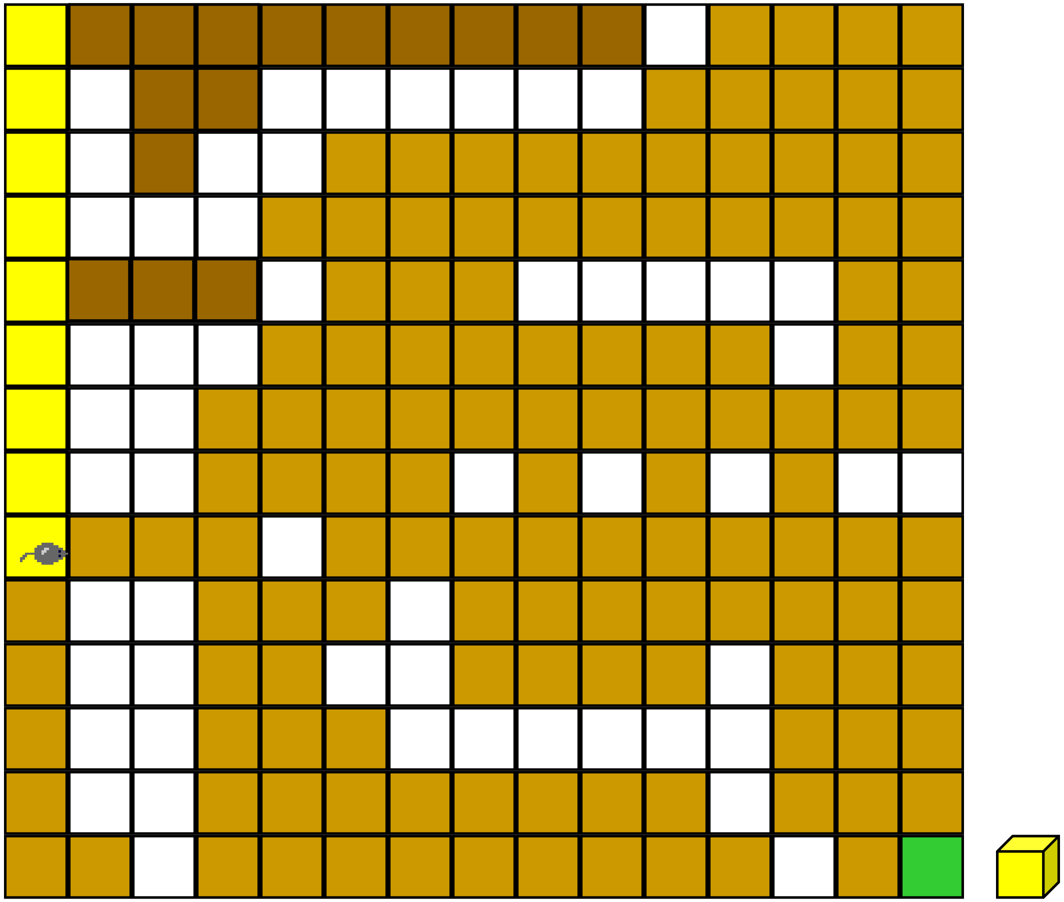
- Move backward until we reach a square from which a forward move is possible.
- Move downward



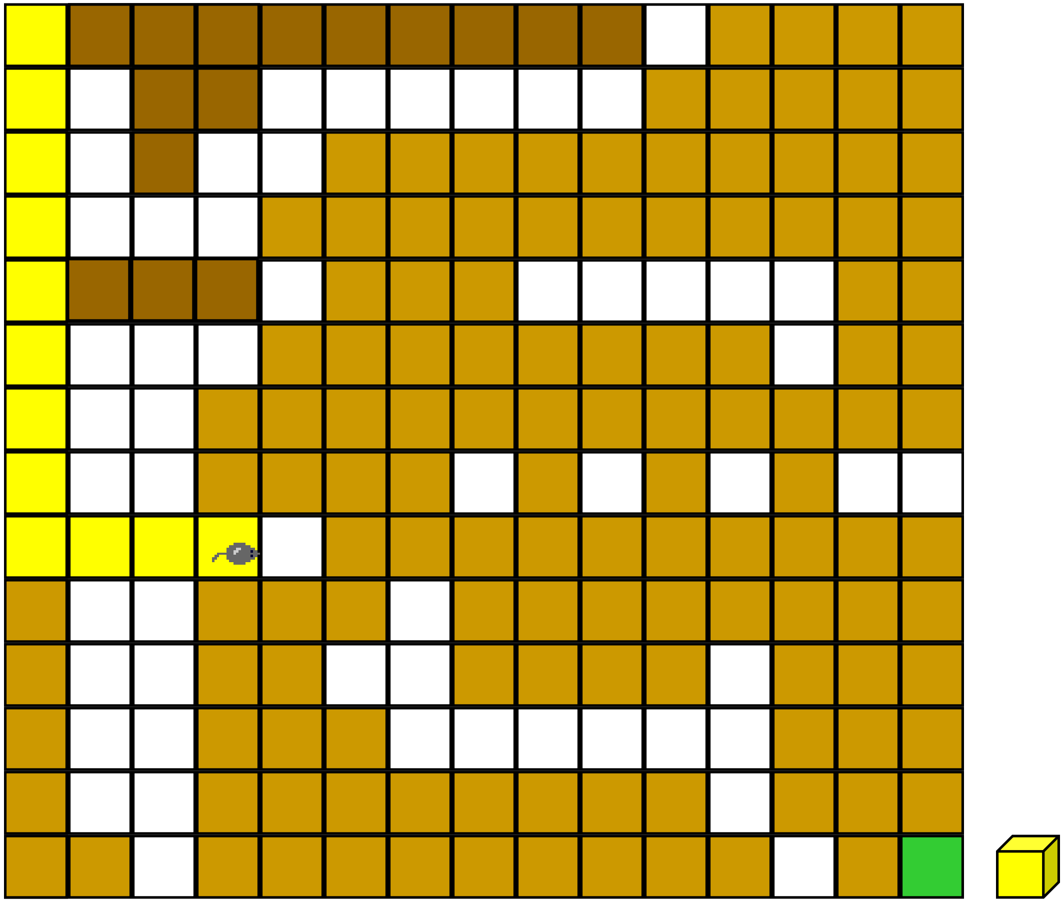
- Move right.
- Backtrack



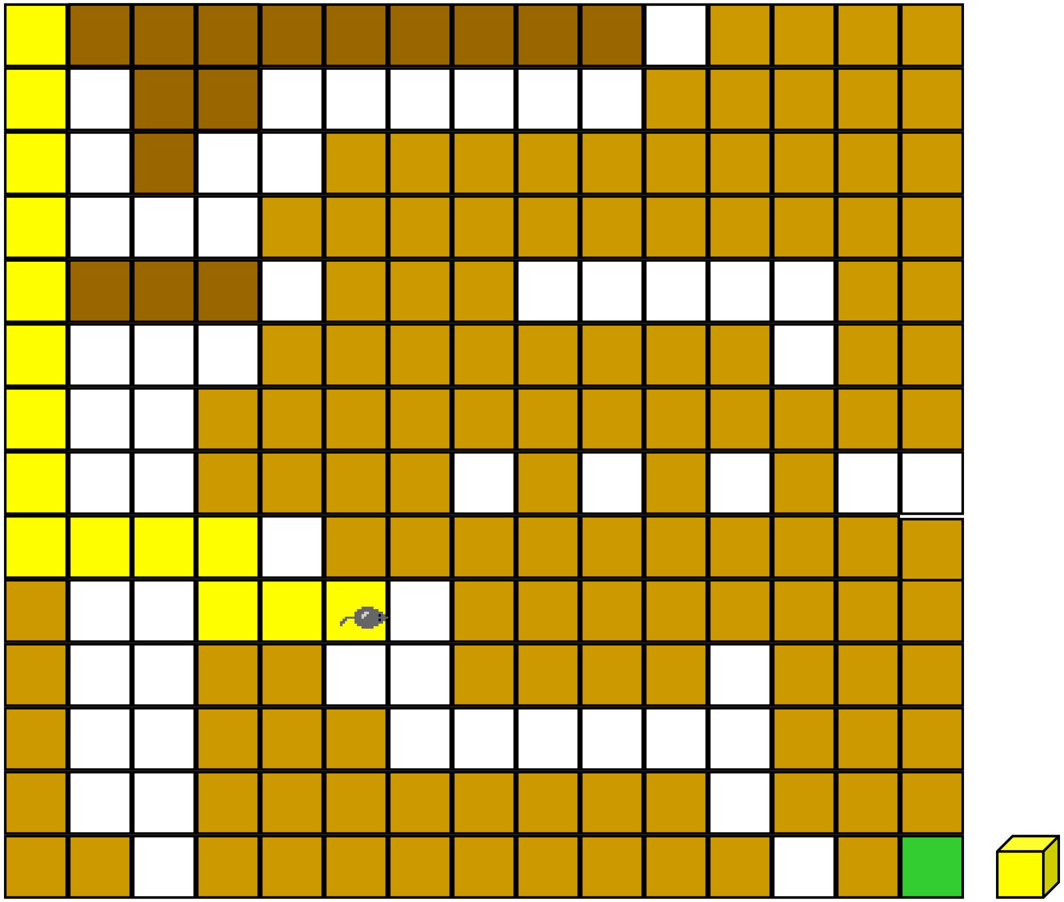
- Move downward.



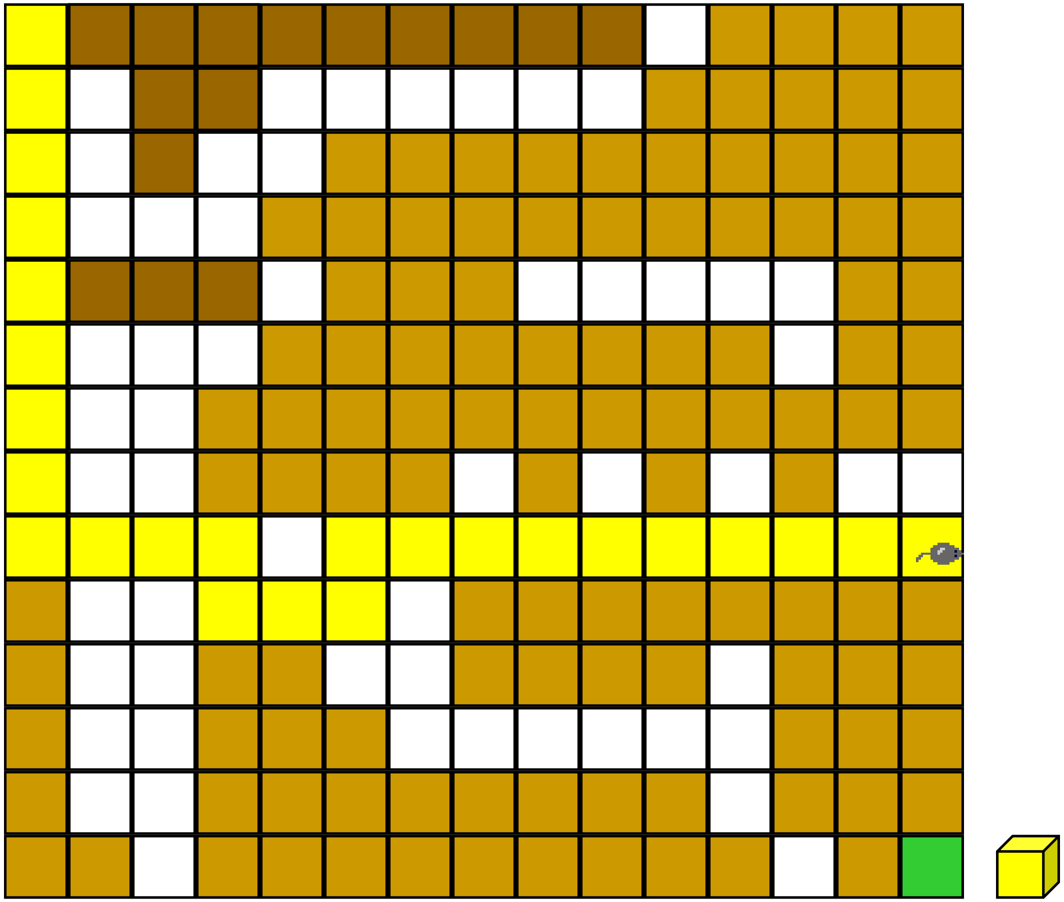
- Move right.



- Move one down and then right.



- Move one up and then right.



- Move down to exit and eat cheese.
- Path from maze entry to current position operates as a stack.

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE 100
#define MAZE_SIZE 5

typedef struct Pos{
short x;
short y;
}Pos;

typedef struct Stack
{
Pos data[MAX_SIZE];
int top;

}Stack;
```

```
char maze[MAZE_SIZE][MAZE_SIZE]={   {'1','1','1','1','1'},
                                    {'e','0','1','0','1'},
                                    {'1','0','0','0','1'},
                                    {'1','1','1','0','x'},
                                    {'1','1','1','1','1'}};
```

/* 미로 찾기의 핵심은 방문한 곳을 표기하고 다음 방문할 곳을 탐색 한 후 스택에 가능한 곳 전부를 Push하고, 다시 Pop하면서 현재 경로로 변경하는 것을 반복하는 것이다. 이동이 가능한 곳은 길 또는 방문하지 않은 곳이다 */

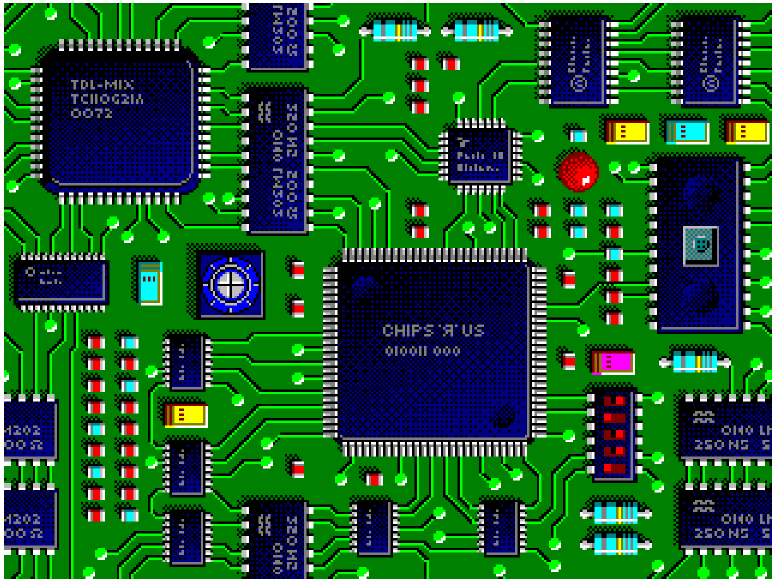
```
void Init(Stack *p) {

		p->top=-1;
}
int  Is_full(Stack *p) {
		return ( p->top == MAX_SIZE-1);
}
int  Is_empty(Stack *p) {
		return (p->top == -1);
}
void push(Stack *p,Pos data) {
   if(Is_full(p)) {
		printf("Stack Full !!₩n"); return ;
   }
   else {
		p->top++;
		p->data[p->top].x=data.x;
		p->data[p->top].y=data.y;
   }
}
```

```
Pos pop(Stack *p) {
   if(Is_empty(p)) {
            printf("스택이 비어있습니다₩n"); exit(1); }
   }
   return p->data[(p->top)--];
}

void Push_Loc(Stack *s,int x,int y) {
   if(x < 0 || y < 0 || x > MAZE_SIZE || y > MAZE_SIZE) return ;

   if(maze[x][y] != '1' && maze[x][y] != '.') {
            Pos tmp;
            tmp.x=x;
            tmp.y=y;
            Push(s,tmp);
   }
}
```

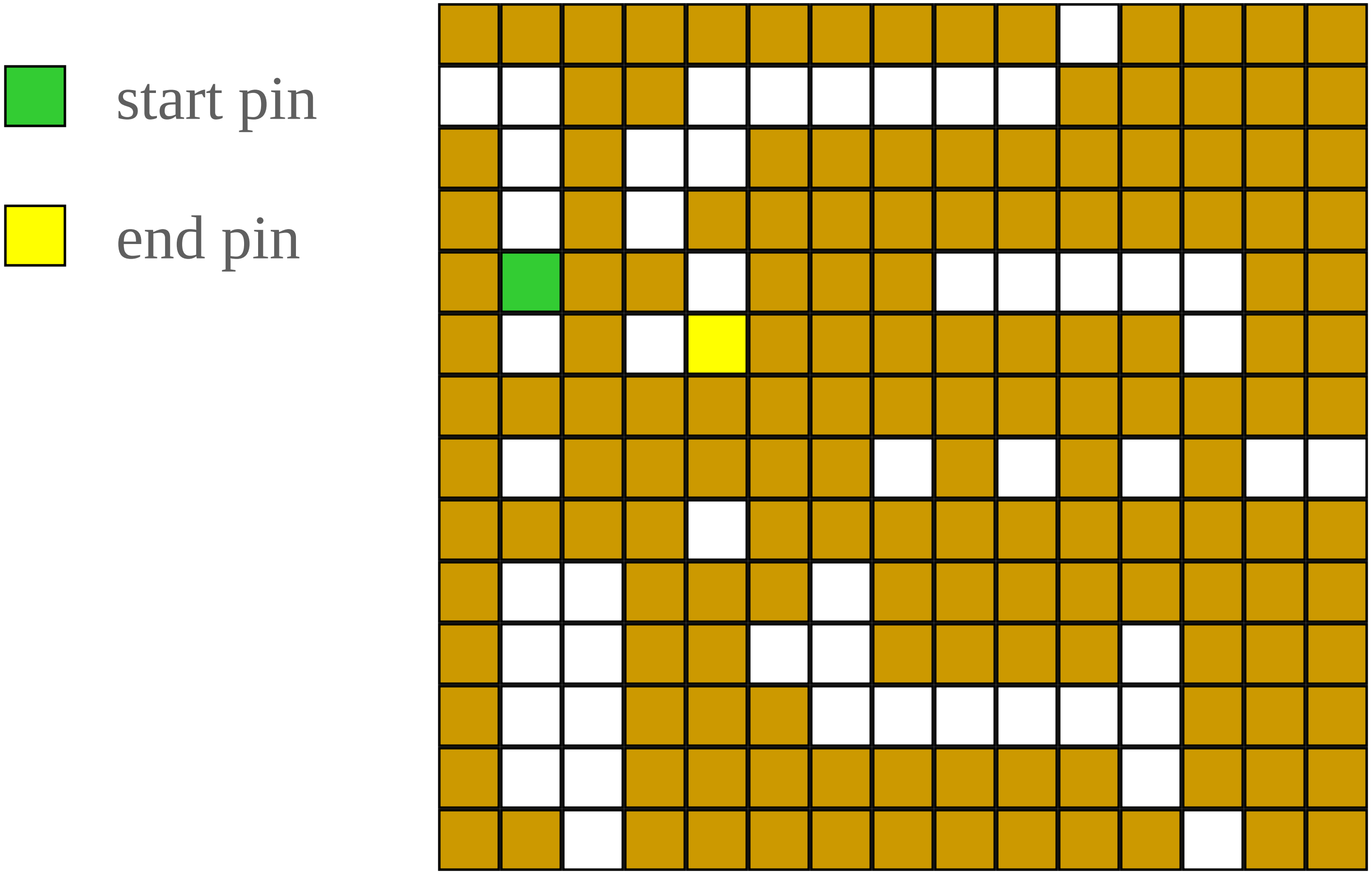
```
int main() {
    Stack s;
    Pos here;
    int i,j,x,y;

    Init(&s);

// 시작점 탐색
    for(i=0;i<MAZE_SIZE;i++) {
            for(j=0;j<MAZE_SIZE;j++) {
                if(maze[i][j]=='e') {
                        here.x=i;
                        here.y=j;
                }
            }
}

printf("시작 점 (%d,%d) ₩n",here.x,here.y);
```

```
while(maze[here.x][here.y] != 'x') {
        x=here.x;
        y=here.y;

        maze[x][y]='.'; // 방문한 곳을 표시

        // 좌,우,위,아래중 이동 가능한 곳을 탐색
        Push_Loc(&s,x+1,y);
        Push_Loc(&s,x-1,y);
        Push_Loc(&s,x,y+1);
        Push_Loc(&s,x,y-1);

        if(Is_empty(&s))  {
                printf("실패₩n");
                return 0;
        }
        else {
                here=Pop(&s); // 현재 좌표를 변경
                printf("(%d,%d)₩n",here.x,here.y);
        }
}
printf("도착 점 (%d,%d)₩n", here.x, here.y);
printf("탐색 성공₩n");
```

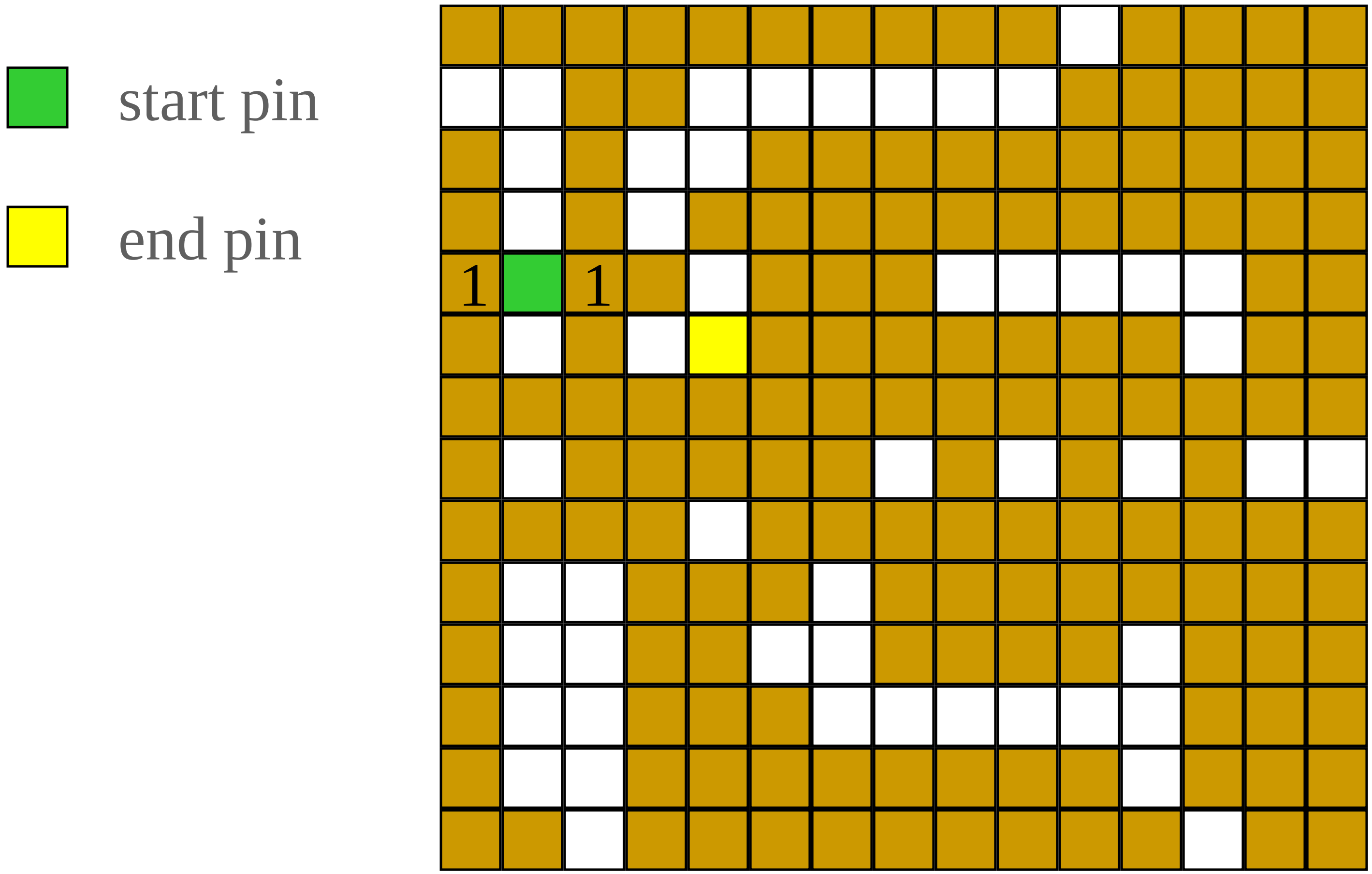
Wire Routing



Lee's Wire Router



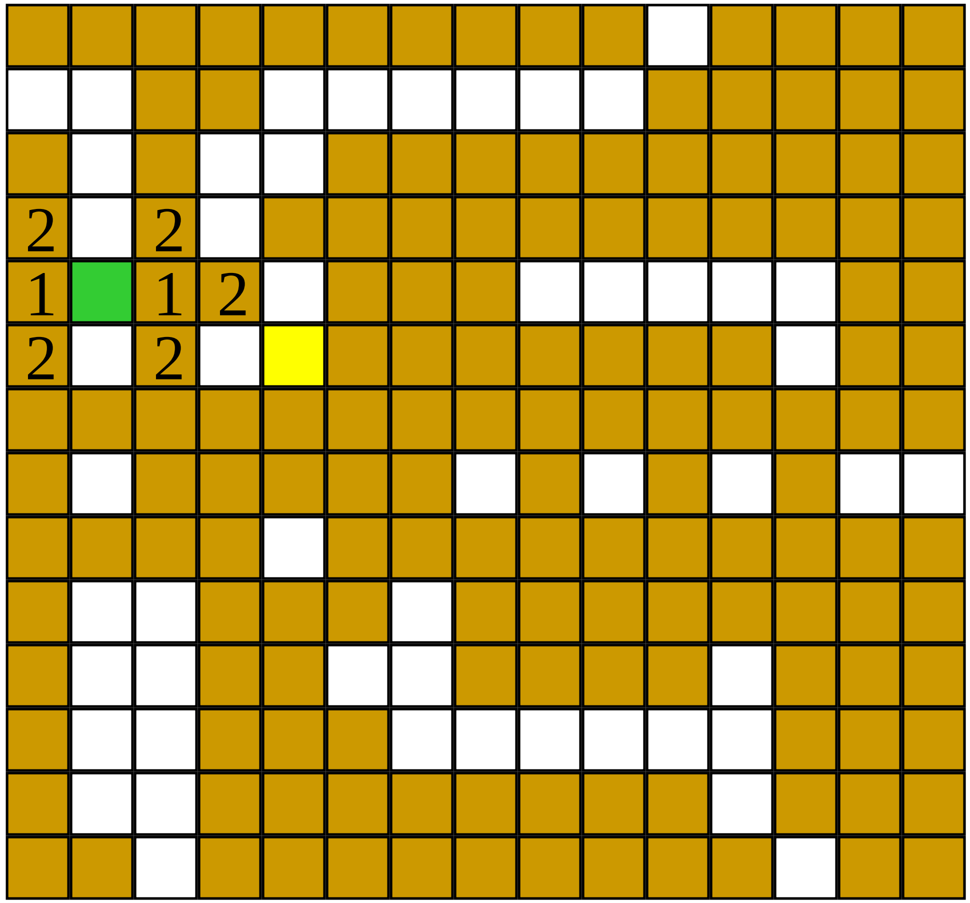
Label all reachable squares by 1 unit from start.



Label all reachable unlabeled squares by 2 units from start.

start pin

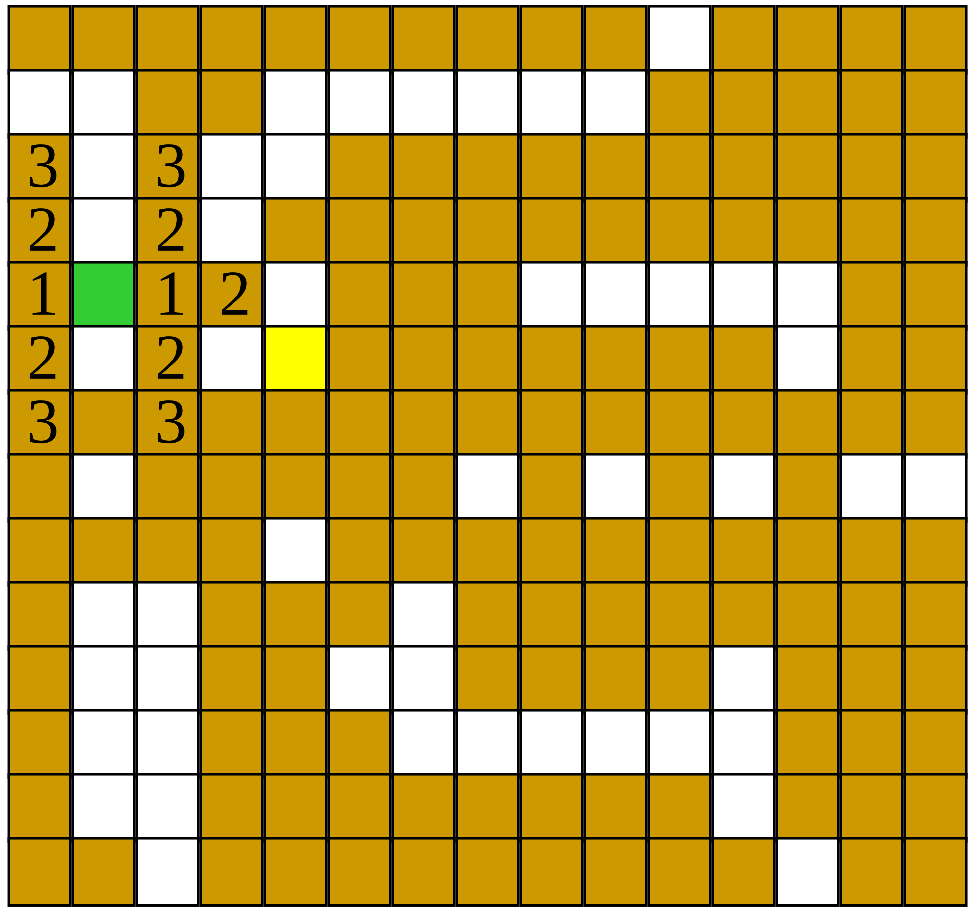
end pin



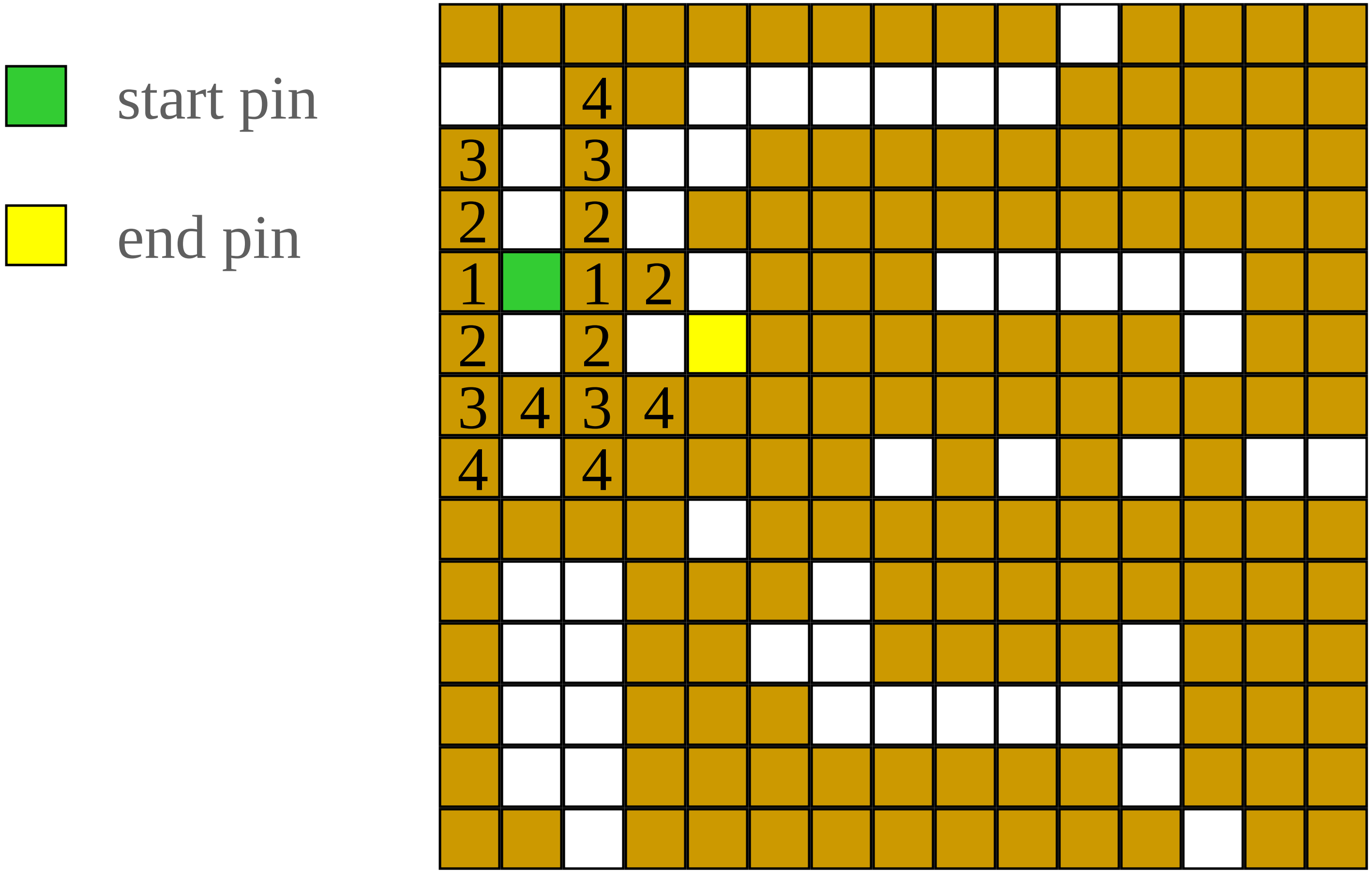
Label all reachable unlabeled squares by 3 units from start.

start pin

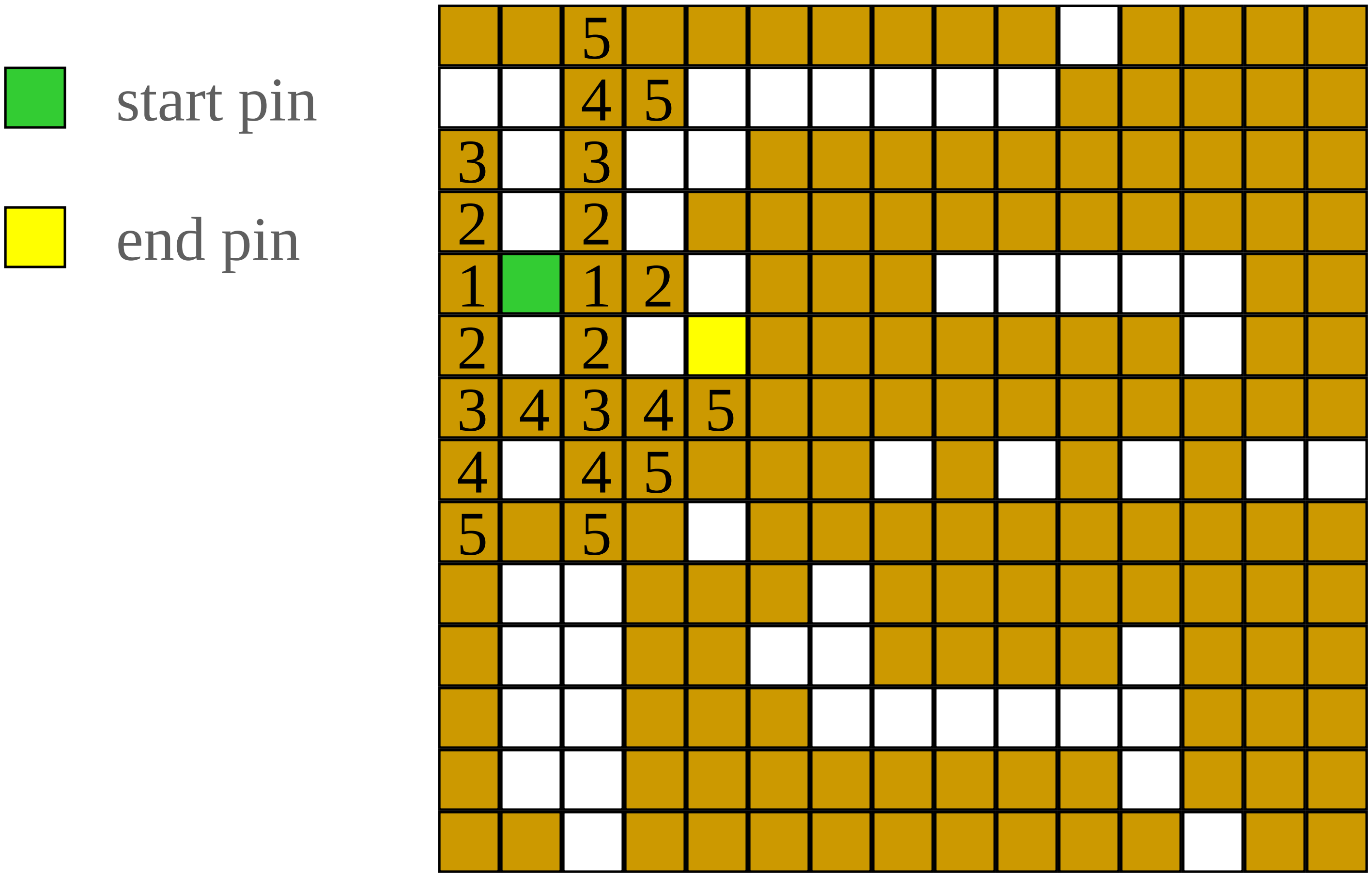
end pin



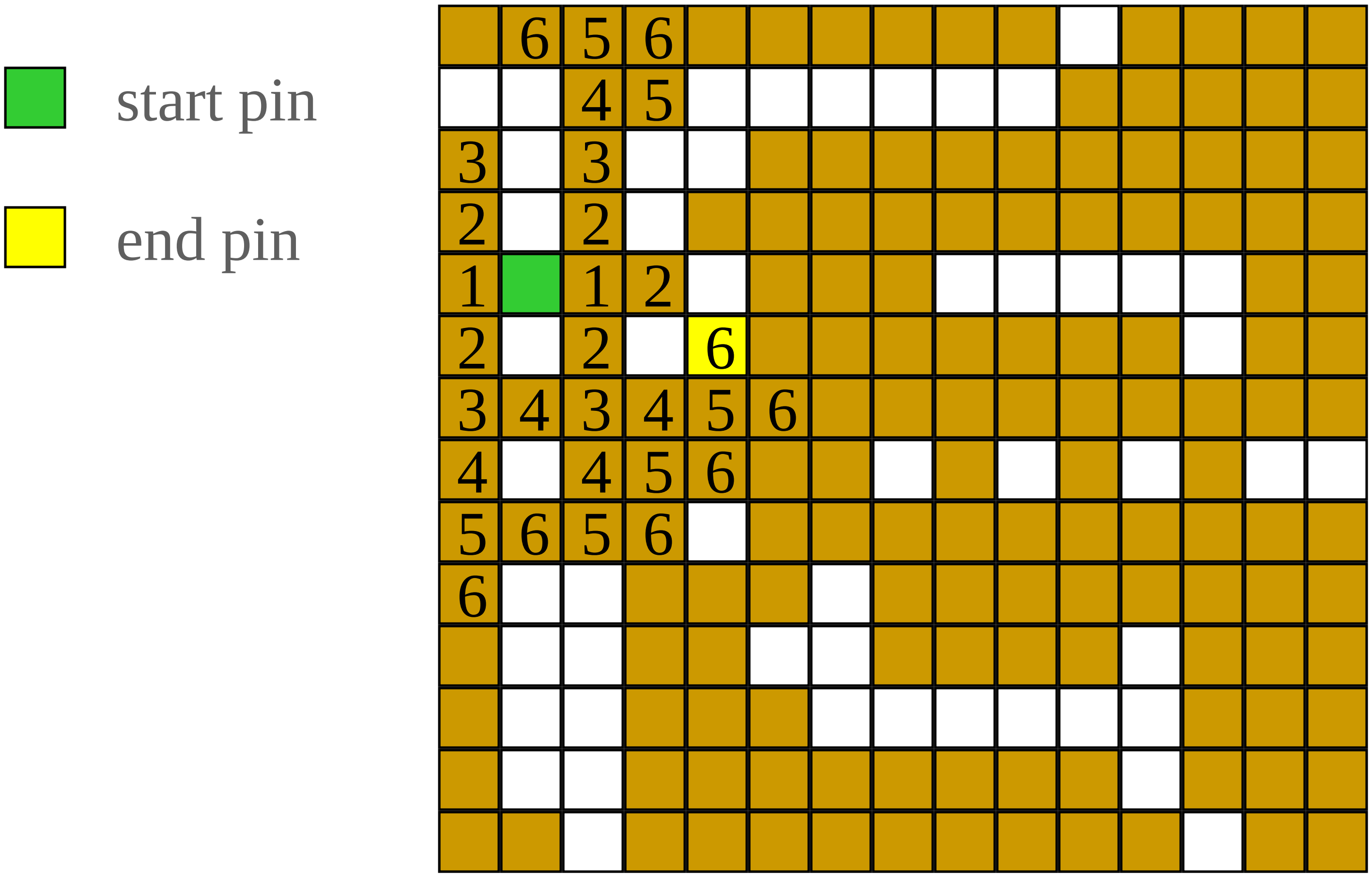
Label all reachable unlabeled squares by 4 units from start.



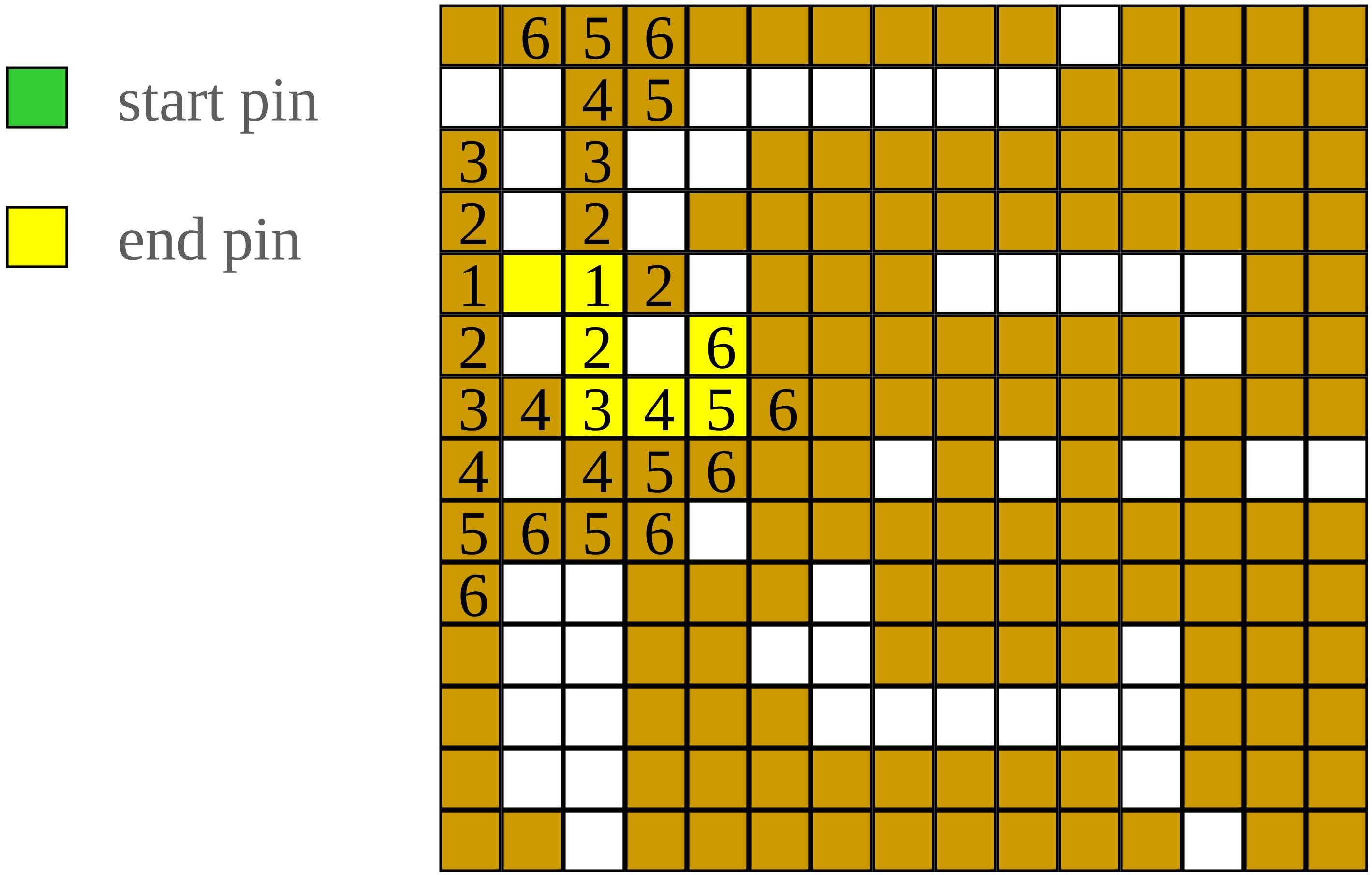
Label all reachable unlabeled squares by 5 units from start.



Label all reachable unlabeled squares by 6 units from start.



End pin reached. Traceback.



End pin reached. Traceback.

More Stack Applications

- Parentheses Matching
- Maze Router
- 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…

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

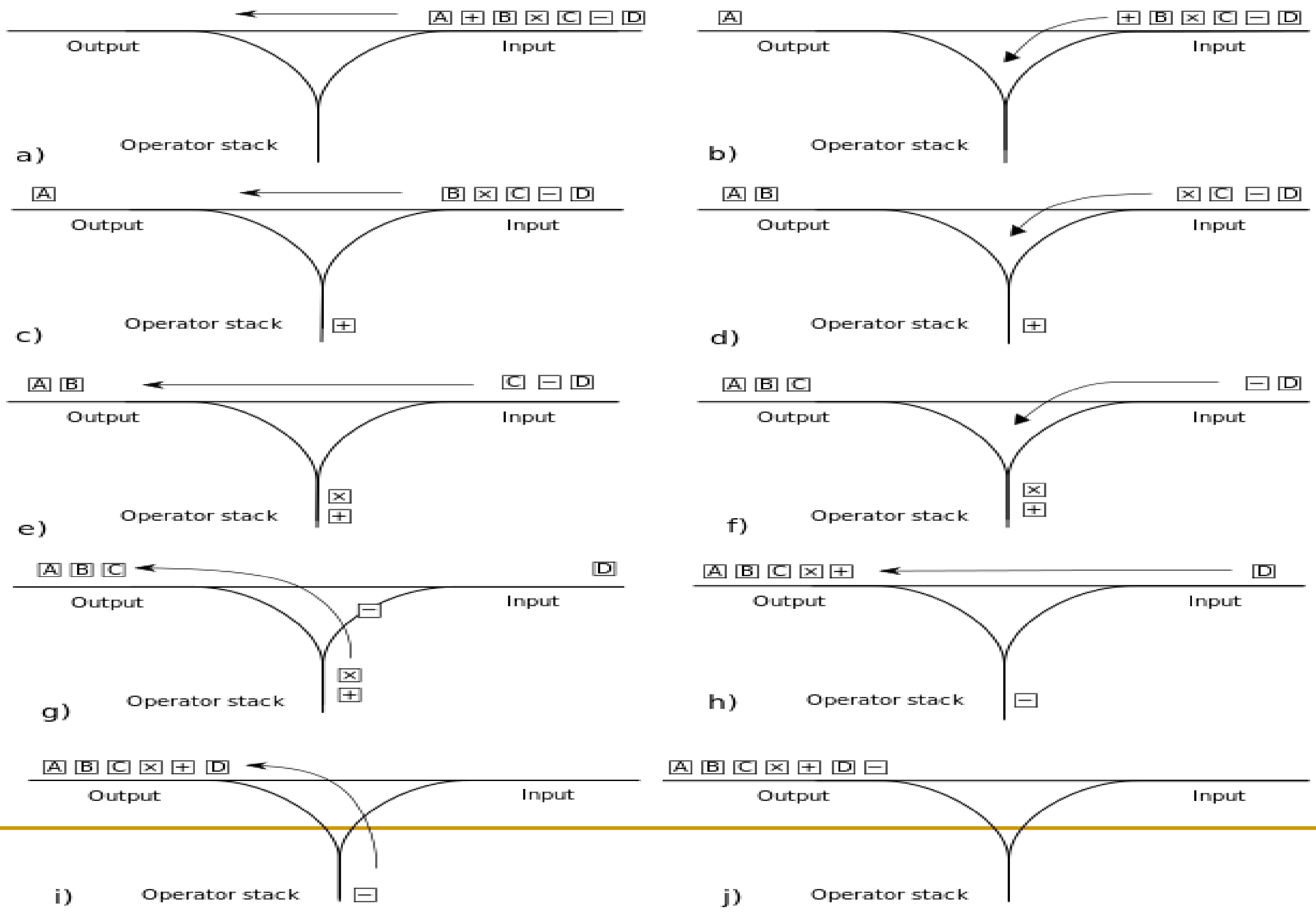
Infix	Postfix
2+3*4	234*+
a*b+5	ab*5+
(1+2)*7	12+7*
a*b/c	ab*c/
(a/(b-c+d))*(e-a)*c	abc-d+/ea-*c*
a/b-c+d*e-a*c	ab/c-de*ac*-

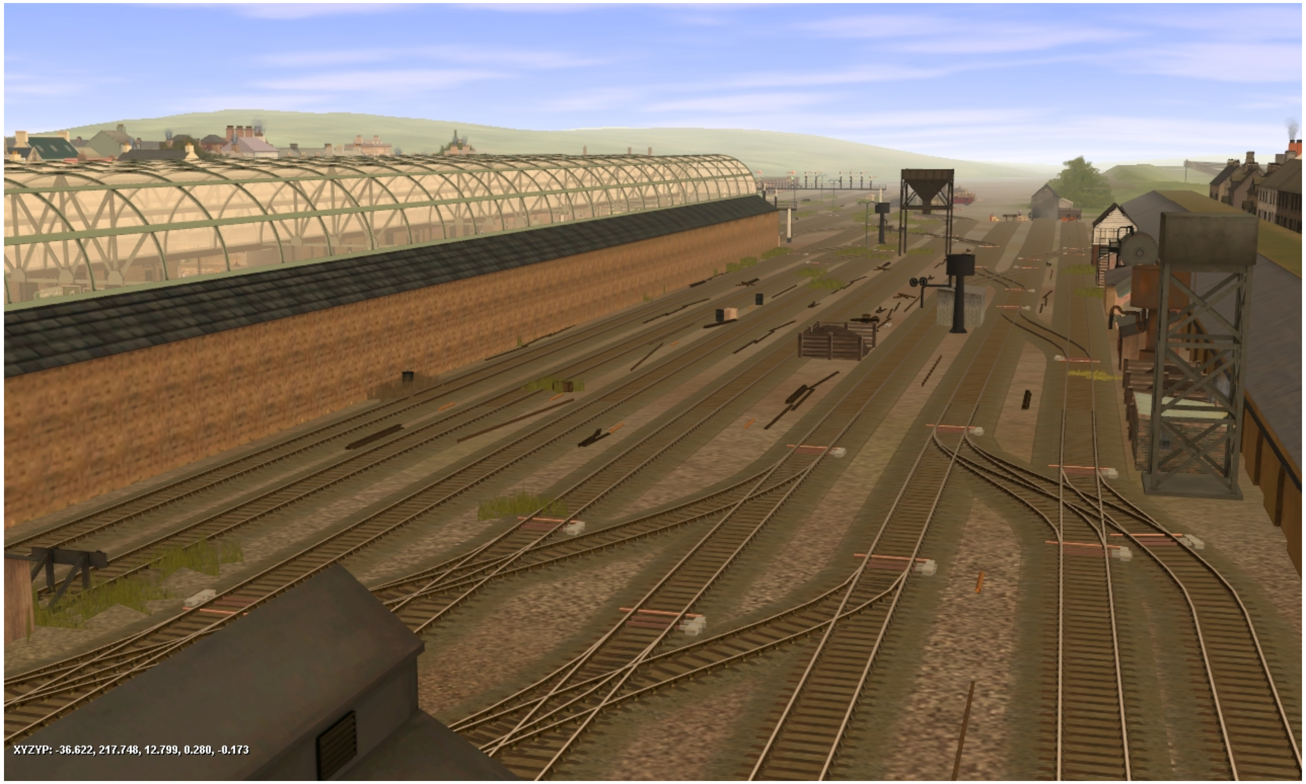
Postfix: no parentheses, no precedence

Token	Stack [0]	Stack [1]	Stack [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

Shunting-Yard Algorithm

- Developed by E. Dijkstra




XYZYP: -36.622, 217.748, 12.799, 0.280, -0.173

Infix to Postfix

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
{1paran, rparen, plus, minus, times, divide, mod, eos, operand} precedence;

int stack[MAX_STACK_SIZE]; /* global stack */

char expr[MAX_EXPR_SIZE]; /* input string */
```

Evaluation of Postfix Expressions

```
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 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 top = -1;
  token = get_token(&symbol, &n);
  while (token != eos)  {
     if (token == operand)
          push(&top, symbol-'0');   /* stack insert */
```

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

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 parenthesized 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*+

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

Token	Stack [0]	Stack [1]	Stack [2]	Top	Output
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	$abc+*_1d$
eos	$*_2$			0	$abc+*_1d*_2$

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

```
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

Infix to Postfix

```
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 ) {
```

Infix to Postfix (cont’d)

```
    /*unstack tokens until left parenthesis */
    while (stack[top] != lparen)
       print_token(delete(&top));
    pop(&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));
    push(&top, token);
   }
 }
 while ((token = pop(&top)) != eos)
    print_token(token);
 print(“\n”);
}
```