Infix, Prefix, and Postfix Expressions:

For simplicity, we will consider algebraic expressions with binary operators +, –, *, and / only.

Infix Notation:

Usual notation in constructing algebraic expression such that operator appears between two operands; it is ambiguous and requires knowledge of operator hierarchy for its evaluation. Parentheses can also be used to override operator hierarchy.

Prefix Notation:

A special notation in constructing algebraic expression such that operator appears before its two operands. It is unambiguous and does not require the use of parentheses or any knowledge of operator hierarchy for its evaluation.

Postfix Notation:

A special notation in constructing algebraic expression such that operator appears after its two operands. It is unambiguous and does not require the use of parentheses or any knowledge of operator hierarchy for its evaluation.

Example: The infix expression ((a–b)/c)*((d+e)–f) has the following postfix and prefix expression.

Postfix: a b – c / d e + f – *

Prefix: * / – a b c – + d e f

Evaluating Postfix Expression:

```
scan given postfix expression;
for each symbol in postfix
    if operand
        then push its value onto a stack S;
    if operator
        then {    pop operand2;
                  pop operand1;
                  apply operator to compute operand1 op operand2;
                  push result back onto stack S;
              }
return value at top of stack;
```

Evaluating Prefix Expression:

```
reverse given prefix expression;
scan the reversed prefix expression;
for each symbol in reversed prefix
    if operand
        then push its value onto a stack S;
    if operator
        then  {  pop operand1;
                 pop operand2;
                 apply operator to compute operand1 op operand2;
                 push result back onto stack S;
              }
return value at top of stack;
```

Infix to Postfix Conversion:

```
given a legal infix string;
create an initially empty postfix string;
create an initially empty operator stack S;
for each symbol ch in the infix string do
        if ch is an operand
            then
                append it to the output postfix string;
        else if ch == '('
            then
                push ch onto stack S;
        else if S == ')'
            then
                pop and append operators to output string until the matching '(' is encountered;
                        // discard the two parentheses
            else        // ch must be some other operator
                {   while operator stack not empty
                        and precedence(top(S)) ≥ precedence(ch)
                            and top(S) != '(' do
                        pop operator;
                        append it to the postfix string;
                    end while;
                    push S
                }
end for;
while operator stack is not empty do
        pop operator;
        append it to the postfix string;
endwhile;
```

Infix to Prefix Conversion:

```
reverse a given legal infix string;
create an initially empty reversed prefix string;
create an initially empty operator stack S;
for each symbol ch in the reversed infix string do
        if ch is an operand
            then
                append it to the output prefix string;
        else if ch == ')'
            then
                push ch onto stack S;
        else if S == '('
            then
                pop and append operators to output string until the matching ')' is encountered;
                        // discard the two parentheses
            else     // ch must be some other operator
                {    while operator stack not empty
                        and precedence(top(S)) > precedence(ch)
                            and top(S) != ')' do
                        pop operator;
                        append it to the reversed prefix string;
                    end while;
                    push S
                }
end for;
while operator stack is not empty do
        pop operator;
        append it to the reversed prefix string;
endwhile;
reverse the reversed output prefix string;
```

Example:

1. Given infix expression: ((a–b)/c)*((d+e)–f) and its equivalent postfix expression: a b – c / d e + f – *

Postfix evaluation:

ch	**action**	**operand stack**
a	push	a
b	push	a b
–	pop operand2	a
	pop operand1	
	compute and push	a–b
c	push	a–b c
/	pop operand2	a–b
	pop operand1	
	compute and push	(a–b)/c
d	push	(a–b)/c d
e	push	(a–b)/c d e
+	pop operand2	(a–b)/c d
	pop operand1	a–b)/c
	compute and push	(a–b)/c (d+e)
f	push	(a–b)/c (d+e) f
–	pop operand2	(a–b)/c (d+e)
	pop operand1	a–b)/c
	compute and push	(a–b)/c (d+e)–f
*	pop operand2	(a–b)/c
	pop operand1	
	compute and push	((a–b)/c) * ((d+e)–f)

2. Given infix expression: ((a–b)/c)*((d+e)–f) and its equivalent prefix expression:
   * / – a b c – + d e f

   Reverse the given prefix expression to get f e d + – c b a – / *

   Prefix evaluation:

ch	action	operand stack
f	push	f
e	push	f e
d	push	f e d
+	pop operand1	f e
	pop operand2	f
	compute and push	f (d+e)
–	pop operand1	f
	pop operand2	
	compute and push	(d+e)–f
c	push	(d+e)–f c
b	push	(d+e)–f c b
a	push	(d+e)–f c b a
–	pop operand1	(d+e)–f c b
	pop operand2	(d+e)–f c
	compute and push	(d+e)–f c (a–b)
/	pop operand1	(d+e)–f c
	pop operand2	(d+e)–f
	compute and push	(d+e)–f (a–b)/c
*	pop operand1	(d+e)–f
	pop operand2	
	compute and push	((a–b)/c)* ((d+e)–f)

3. Given infix expression: (a–b)/c*(d + e – f / g).

   Input: (a–b)/c*(d + e – f / g)

   Postfix conversion:

ch	**action**	**operator stack**	**Postfix**
(	push	(	
a	output	(	a
–	push	(–	a
b	output	(–	a b
)	pop until (		a b –
/	push	/	a b –
c	output	/	a b – c
*	pop		a b – c /
	push	*	
(	push	* (	a b – c /
d	output	* (	a b – c / d
+	push	* (+	a b – c / d
e	output	* (+	a b – c / d e
–	pop	* (	a b – c / d e +
	push	* (–	a b – c / d e +
f	output	* (–	a b – c / d e + f
/	push	* (– /	a b – c / d e + f
g	output	* (– /	a b – c / d e + f g
)	pop until (	*	a b – c / d e + f g / –
	pop until empty stack		a b – c / d e + f g / – *

   Postfix: a b – c / d e + f g / – *

4. Given infix expression: (a–b)/c*(d + e – f / g).

   Input:) g / f – e + d (* c /) b – a (

   Prefix conversion:

ch	**action**	**operator stack**	**Reversed Prefix**
)	push	)	
g	output	)	g
/	push	) /	g
f	output	) /	g f
–	pop	)	g f /
	push	) –	g f /
e	output	) –	g f / e
+	push	) – +	g f / e
d	output	) – +	g f / e d
(	pop until)		g f / e d + –
*	push	*	g f / e d + –
c	output	*	g f / e d + – c
/	push	* /	g f / e d + – c
)	push	* /)	g f / e d + – c
b	output	* /)	g f / e d + – c b
–	push	* /) –	g f / e d + – c b
a	output	* /) –	g f / e d + – c b a
(	pop until)	* /	g f / e d + – c b a –
	pop until empty stack		g f / e d + – c b a – / *

   Prefix: * / – a b c – + d e / f g