Hello students. Welcome to this session

of semester two data structures.

In this session we would be doing utility

and conversion of these expressions from

one to another that is infix to prefix.

This is module #5. In this module.

We would be covering converting

an infix expression to prefix.

Then algorithm to convert an

infix expression to prefix.

That is, using stacks.

We'll see some examples.

Along with the stack contents.

And some advantages of prefix expressions.

So at the end of this module you would

be able to know the usage of stacks for

converting an infix expression to prefix.

You would be able to view the

status of the stack during the

infix to prefix conversion.

You would be able to list the

advantages of prefix expression.

And know the importance of

infix to prefix conversion.

Let me give you a brief

introduction of prefix notation.

Prefix expressions notation requires

that all the operators precede the two

operations that make that they work on.

For example.

If you have an expression like A+B,

this is the infix expression.

In prefix notation,

it would be returned as plus A B,

so you can see that the plus operator

is preceding the operands A&B.

Similarly, for a into B.

The prefix expression would be.

Multiplication followed by A&B.

The operators over here that is

the plus and the multiplication

precede the operands A&B,

so this is the prefix notation.

Now let's begin with the

algorithm to convert our infix

expression to a prefix.

1st. We need to reverse the

given X infix expression,

so the input to this algorithm

is an infix expression.

That we would be reversing it.

After reversing, the expression

will scan the characters 1 by 1.

If the character is an operand,

we copy it to the prefix notation output.

If the character is a closing parenthesis,

then we push it into the stack.

If the character is the opening parenthesis,

then we pop the elements of

the stack until we find the

corresponding closing parenthesis.

Next, if the character scan is an operator,

then if the operator has a higher precedence.

Then or equal to the top of the stack.

Then we push the operator into the stack.

That is, if the operator has a precedence

lesser than the top of the stack,

then we pop the operators.

From the stack and we output it or we.

Append to the prefix notation.

And then check the above condition again

with the new top value of the stack.

This particular step will be continued.

Until either we have reached

the end of the stack,

or we have an operator which

has got a higher precedence.

After all, the characters have

been scanned, then

Whatever prefix notation we have got so far,

we will reverse the prefix notation

to get the actual prefix expression.

Let's see how this particular algorithm

works with the help of an example.

You would be able to see the stack contents.

Basically, the stack contains 3 columns.

The table which you can see.

Will be showing the character

which is scanned the prefix

string and the stack contents.

The first example,

which I would be doing is the infix

expression A+B into C.

So the first step would be reversing

this string after reversing

what we get is C into B + A.

This particular string would

be scan character by character.

So the first character which is scanned is

C which is an operand. So what do we do?

Is we appended to the prefix string.

The next character which is read

is a multiplication operator.

Since the stack is empty right now.

This operator would be

simply added to the stack.

The next character is an operand,

so we added to the prefix string.

So now the prefix string contains C&B.

Where as the stack contains

the multiplication operator.

The next operator is the plus sign.

Now we will have to check.

With the stack contents,

that is the top of the stack

which contains multiplication.

Since the plus operator has a lesser

precedence over the top of the stack.

What will happen is the top

of the stack would be popped.

And added to the prefix string.

And then, since the stack becomes empty,

the plus operator would

be added to the stack.

The next character is A which would

be added to the prefix string.

We have reached the end of the string,

to be scanned.

Now we check for the stack contents

and we pop and the append

to the prefix string.

So at the end we have CB multiplication

operator A and then the plus operator.

Now, this particular prefix

string would be reversed.

And we would get the final output

that is the prefix string plus a.

followed by the

multiplication operator and then BC.

Which is the final prefix string.

Let's take another example. Now.

This example is along with the parenthesis.

So as you can see,

the infix expression a + B is.

Enclosed within the brackets,

followed by the multiplication

sign and then C..

So we begin with the algorithm

with the first step that is

reversing of the infix string.

So what result we get is C multiplication.

Closing brackets B + A and

then the opening brackets.

This particular string would be

scanned so the first character

scan is a operant so it would

be added to the prefix string.

The next operator is.

Multiplication. The stack is empty,

so it would be added to the stack.

The next operator is a closing

brackets. Now what happens is,

since this is a closing bracket,

we simply added to the stack.

Next, the next character is.

And operating so we added

to the prefix string.

Next we have plus.

Now this plus operator would be

checked with the top of the stack,

which is a closing bracket.

Since plus has got

A higher precedence over

the closing brackets.

It would be added to the stack.

The next operand is A would

be added to the prefix string.

Next is the opening brackets.

Now since opening brackets has got a higher

precedence over the other operators,

So what will happen is we're going to

pop plus add it to the prefix string.

The next operator that would be

checked would be the closing brackets.

The closing bracket would be

Popped and simply discarded.

The next operator is multiplication,

which would be popped an

added to the prefix string.

So what happens is the brackets

are simply popped and ignored.

They are not added to the prefix string.

Now, this particular prefix

string would be reversed to

get the final prefix string,

which is multiplication plus ABC.

Let's see the importance of

infix to prefix conversion.

Why do we require this conversion?

Normally an infix expression

like a + B -- C for example.

Needs to be checked for the

precedence in this example,

which you can see both operators

have got the same precedence,

so the associativity would be considered.

And the leftmost plus would be done first.

Secondly, if you can see the

next example which is shown,

it is A+B enclosed within

the brackets into C,

which would force the addition

of A+B to be done first.

And then the multiplication since

the parenthesis have got a higher

precedence over multiplication.

So these type of expressions are ambiguous.

Computers need to know exactly

what operators to be performed

and in what order.

That's why the prefix notation is

used because it eliminates this

above confusion of which operators

to be performed 1st and which

operator to be performed next.

For example, will take the same

example A + B into C would be

returned as plus A into BC in prefix.

here what happens is the

multiplication operator comes before.

The operand B and C denoting

that multiplication.

Or has presidents over plus.

The addition operator then

appears before the A and the

result of the multiplication.

So the advantages of prefix notation.

They are suitable for direct execution.

Prefix form can be trivially be turned

into a tree for further processing.

And prefix notations are

entirely unambiguous.

Infix notations are said to be

ambiguous and requires precidence and

associativity rules to be applied,

to make it unambiguous.

Prefix notations is entirely unambiguous.

That is expression which is returned

in a prefix form like multiplication

plus 563 OK cannot be interpreted

as the multiplication has to be

done 1st and then the addition.

OK, but in case of infix expression then

the parenthesis have to be considered.

1st and then the Multiplication or other operators.

So in this module, what we have

learned is what is a prefix expression?

The algorithm which is used to convert

an infix expression to prefix.

We saw the usage of stacks to convert

or infix expression to prefix.

What is the importance of this conversion?

An advantage is of prefix expressions.

These are the references which were

used for this particular module.

Thank you.