Process of transforming infix string of arithmetic expression to prefix and postfix string using application of Object-Oriented Programming and Linked List

From Assignment Evaluating Expression - HCMUT - Faculty of Computer Science and Engineering - Course: Discrete Structure for Computing, Exercise 1a and 1b.

Academic year: 2021-2022 *(semester 212)*

Instructors: Dr. Nguyen Van Minh Man
Dr. Nguyen Tien Thinh

Student:
Nguyen Phan Tri Duc - 2152528

I. Idea:

1. Block class:

To transform an Infix string to a Prefix string, we change the order of [THIS][OP][THAT]*(left value-operator-right value)* to [OP][THIS][THAT]. Also, if a string has many operators, it's important knowing which operator has more priority in order to to change that first.

An Infix string can be detached into *Blocks* of values and operators, which will make a significant advantage for the process because working on a raw Infix string should be very horrible. So a *Block* will have a property `string content` which will save its name like "12" or "a" or "+".

Back to the problem of priority, a *Block* of an operator will have a property called `int operator_value`, the bigger that property is, the more priority that operator has. *A non-operator Block will have 0 operator value*.

Because *Blocks* are connected together to form a string, using Linked List is a powerful way to handle this so we have a pointer points to next *Block* `Block* next`.

To simplify the algorithm, all properties are public, and below is the `class Block` with a simple *constructor*.

```
class Block{
   public:
   int operator_value;
   string content;
   Block* next;

   Block(int o_v,string c){
       operator_value=o_v;
       content=c;
       next=NULL;
   }
};
```

2. List class:

As a normal Linked List, *List* class has `Block* head` and `Block* tail` to help create and process on the *List*. To make it less complicated, when creating a *List*, I will initialize one Block *that has no content* to be the head so the below algorithms is less complicated.

A push function is also added to the class which will receive data, create a Block relative to the data and and to the back of the List, this is what an initial *List* class looks like.

```
class List{
   public:
   Block* head;
   Block* tail;

   List(){
        Block* B=new Block(0,"");
        head=B;
        tail=B;
   }
   void push(int o_v, string c){
        Block* B=new Block(o_v,c);

        tail->next=B;
        tail=B;
   }
};
```

3. Additional functions:

`bool isAlpha(char c)`

(returns true if the character is in the alphabet)

`bool isDigit(char c)`

(returns true if the character is from 0 to 9)

`bool isOp(char c)`

(returns true if the character is an operator)

II. Process

1. Extracting Blocks from the input string:

Problem needs to be concerned:

- Extract the right content(name) of the Blocks.
- Store the operator value(the value that stands for the priority of an operator mentioned above) right in this process.

Back to the priority of an operator, it has more priority when it is:

- Inside the most pairs of brackets.
- An exponent operator.
- A multiply operator or a divide operator.
- A plus operator or a minus operator.
- More on the left side of a string.

Following the basic rules above, we will start the for loop from the starting position of the input string running back to the and, and also create some parameters to help indicate an operator's priority value:

- `int brackets_value=0`

  Increase by 10000 when meet an open bracket, and decrease by 10000 when meet a close bracket.

- `int lefter_value=INPUT.size()`

  Decrease by 1 when finish one loop.

- `int counter=0`

  To count how many Blocks.

- `int NumberOfOperators=0`

  To count how many operators.

When comes to an operator, it will have additional value: 3000 if '^', 2000 if '*' or '/', 1000 if '+' or '-'.

Example: *INPUT="30+A*((9/B)^7)-8"* (length 16 so initial `lefter_value` is 16)

Operator value of:
+ : 1014 *(1000+(16-2) because it's INPUT[2])*
* : 2012 *(2000+(16-4) because it's INPUT[4])*
/ : 22008 *(20000+2000+(16-8) because it's inside 2 pairs of brackets and it's INPUT[8])*
^ : 13005 *(10000+3000+(16-11) because it's inside a pair of brackets and it's INPUT[11])*
- : 1002 *(1000+(16-14) because it's INPUT[14])*

So the priority order is: / ^ * + -

Codes: (Notice that the codes haven't include the case where input string is more complex, which will be mentioned in another section)

```
int brackets_value=0;
int lefter_value=INPUT.size();
int counter=0;
int NumberOfOperators=0;
List MyList;

   for(int i=0;i<INPUT.size();i++)
   {
       if(INPUT[i]=='(')
           brackets_value+=10000;
       if(INPUT[i]==')')
           brackets_value-=10000;
       if(isAlpha(INPUT[i]))
       {
           string Alpha="";
           Alpha+=INPUT[i];
           counter++;
           MyList.push(0,Alpha);
       }
       if(isDigit(INPUT[i]))
       {
           string Number="";
           while(isDigit(INPUT[i]) || INPUT[i]=='.')
           {
               Number+=INPUT[i];
               i++;
           }
           i--;
           counter++;
           MyList.push(0,Number);
       }
       if(isOp(INPUT[i]))
       {
           int operator_value=brackets_value+lefter_value;
           switch(INPUT[i])
           {
               case '+':
               {
                   MyList.push(operator_value+1000,"+");
               } break;
               case '-':
```

```
                {
                    MyList.push(operator_value+1000,"-");
                } break;
                case '*':
                {
                    MyList.push(operator_value+2000,"*");
                } break;
                case '/':
                {
                    MyList.push(operator_value+2000,"/");
                } break;
                case '^':
                {
                    MyList.push(operator_value+3000,"^");
                } break;
            }
            NumberOfOperators++;
            counter++;
        }
        lefter_value--;
    } // end for loop
```

2. Switching position of group of Blocks to Prefix notation:

An operator that has the most priority(highest operator value) will be calculated first, and what is meant to be calculated is that you can think of that as another value.

Example: A*(B+C)
B+C will be calculated first, so we can think B+C is another value D, and the string will be simply A*D.

How will this help in the idea of switching position from Infix to Prefix?

Instead of thinking B+C is another value D in the example above, we first change B+C to Prefix notation, then remember it as D.
B+C → +BC (D is +BC)
So now A*(B+C) becomes A*D, and perform changing A*D to Prefix notation, we have:
A*D → *AD
or *A+BC

And that's exactly the Prefix notation of A(B+C).*

Explain:
What's really happening here is that when one part of the string is being switched to Prefix notation, that part will stay like that for the rest of the process. Just like you calculate that part first.

As a result, switching the positions of the operator from the highest priority to the lowest, and binding the relevant Blocks during the process will result in a Prefix notation of that List.

Observing the string 30+A*((9/B)^7)-(77+C)*8 using idea above to change from Infix to Prefix notation so we could have a better solution to create an algorithm:

$30\ +_1 A\ *_1((9/B)\text{^}7)\ -\ (77\ +_2 C)*_2 8$ $(length\ 23\ so\ initial$ `lefter_value` $is\ 23)$

Operator values(get from the idea of Extracting Blocks above):

$+_1$: 1021
$*_1$: 2019
/ : 22015
^ : 13012
$-$: 1009
$+_2$: 11005
$*_2$: 2002

Process:

$30\ +_1 A\ *_1([/\ 9\ B]\text{^}7)\ -\ (77\ +_2 C)\ *_2 8$ *(switching process on operator /)*
$30\ +_1 A\ *_1[\text{^}\ /\ 9\ B\ 7]\ -\ (77\ +_2 C)\ *_2 8$ *(switching process on operator ^)*
$30\ +_1 A\ *_1[\text{^}\ /\ 9\ B\ 7]\ -\ [+_2\ 77\ C]\ *_2 8$ *(switching process on operator $+_2$)*
$30\ +_1[*_1\ A\ \text{^}\ /\ 9\ B\ 7]\ -\ [+_2\ 77\ C]\ *_2 8$ *(switching process on operator $*_1$)*
$30\ +_1[*_1\ A\ \text{^}\ /\ 9\ B\ 7]\ -\ [*_2 +_2\ 77\ C\ 8]$ *(switching process on operator $*_2$)*
$[+_1\ 30\ *_1\ A\ \text{^}\ /\ 9\ B\ 7]\ -\ [*_2 +_2\ 77\ C\ 8]$ *(switching process on operator $+_1$)*
$[-\ +_1\ 30\ *_1\ A\ \text{^}\ /\ 9\ B\ 7\ *_2 +_2\ 77\ C\ 8]$ *(switching process on operator $-$)*

By using **square brackets** after switching position of higher priority operator, we can think that as a glue binding the parts inside together, so then when an another operator have left*(switching process on operator $*_1$)* or right*(switching process on operator $*_2$)* or both*(switching process on operator $-$)* value is a **square brackets**, a whole **square brackets** is being switched.

How to bind the codes in the program?

We will use a variable called `int flag` for Blocks, if a bunch of Blocks have the same *flag*, they will be in one **square brackets**, let's initialize the Blocks with `flag=0`.

```
class Block{
   public:
   int operator_value;
   string content;
   Block* next;

   int flag;

   Block(int o_v,string c){
        operator_value=o_v;
        content=c;
        next=NULL;

        flag=0;
   }
};
```

Switching process of an operator and its two relevant values will bind those 3 together so it will update the flag of all.
Also after the switching process on an operator, we will have new **square brackets**, so for each switching process completed, the flag used for the next switching process should be updated too.
The switch function will be in List class, so this is what a List class looks like with the switch function.

```
class List{
   public:
   Block* head;
   Block* tail;

   int flag;

   List(){
        Block* B=new Block(0,"");
        head=B;
        tail=B;

        flag=1000;
   }
```

```
    void push(int o_v, string c){
        Block* B=new Block(o_v,c);

        tail->next=B;
        tail=B;
    }

    void SWITCH(Block* THIS, Block* OP, Block* THAT){
    //Switching [THIS] [OP] [THAT] to [OP] [THIS] [THAT]
    //Update the flag of [OP] and [THIS] and [THAT]
    //to be the current flag variable of the List
       flag+=1000;
    }
};
```

Notice that a flag variable is also added in the class, and initialize it as 1000. Whenever a Switch function is called, it will switch position of the Blocks, then update the flag of those Blocks to be the current flag of List class, then finally update the flag of List class for the next switching process.

Pseudo-code: (full switch function is in the source code)

```
void SWITCH(Block* THIS, Block* OP, Block* THAT){
    //UPDATE FLAG OF [OP] and [THIS] THEN SWITCH
    if(THIS->flag==0) //THIS is not in any square brackets
        Update flag of [OP]
        Update flag of [THIS]
        Switching position form [THIS][OP][THAT] to [OP][THIS][THAT]
    else    //THIS is in a square brackets
            //We must find the starting Block of the square brackets,
            //then let decoy be the previous Block of the starting Block.
        Update flag of [OP]
        Update flag of whole square brackets contain [THIS]
        Switching position form [THIS][OP][THAT] to [OP][THIS][THAT]

    //UPDATE FLAG OF [THAT]
    if(THAT->flag==0) //THAT is not in any square brackets
        Update flag of [THAT]
    else
        Update flag of whole square brackets contain [THAT]

    flag+=1000;
}
```

When to call the switch function?

Switch function will take an operator and its two relevant values to switch position, previously we had the order of operators so now we just called the switch function in order.

Pseudo-code: (full while loop is in the source code)

```
int max;
while(true)
{
   max=0;

   Update max is the highest operator_value out of the operators
   if(max==0) break; //If there's no more operator to switch

   Locate the operator [OP] have max as its operator_value
   Locate [THIS], [THAT]

   MyList.SWITCH([THIS],[OP],[THAT]);
   Update the operator_value of [OP] to -1
}
```

After this will give us the List of Blocks that are in order of Prefix notation.

3. From Prefix notation to Postfix notation:

Observing the string 30+A*((9/B)^7)-(77+C)*8:
Prefix: - + 30 * A ^ / 9 B 7 * + 77 C 8
Postfix: 30 A 9 B / 7 ^ * + 77 C + 8 * -

Even if the string is in Prefix or in Postfix notation, the order of values is unchanged! But it's important knowing where the operators should be from Prefix to Postfix notation.

From Prefix to Postfix, we have [OP][THIS][THAT] to [THIS][THAT][OP], so [OP] should be moved to the end of [THAT]. Check the '+' operator in the examples below:

+AB	[OP]	+
	[THIS]	A
	[THAT]	B
	End of [THAT]	B

+A*BD	[OP]	+
	[THIS]	A
	[THAT]	*BD
	End of [THAT]	D

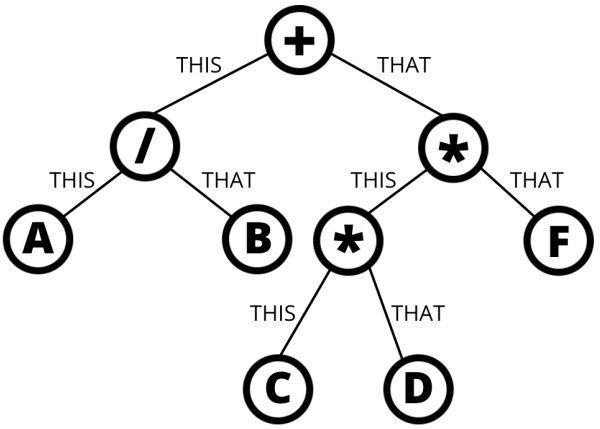
-+/AB**CDFG	[OP]	+
	[THIS]	/AB
	[THAT]	**CDF
	End of [THAT]	F

If the Postfix transformation process on those Prefix is functioning right, the '+' operator should be moved to the right of each 'End of [THAT]'.

How to find End of [THAT]?

A simple operator has 2 relevant values/children [THIS] and [THAT], when [THIS] or [THAT] is another operator, it should have 2 children also. Total number of descendants as values of the original operator increases by 1 when one of its children/grandchildren/… is an operator*(because an operator type child takes 1 value space and contributes 2 more value spaces).*

Think of it as a tree:



Initially, every operator has 2 children, but '+' has 2 children is an operator and 1 grandchildren is an operator, so the total descendants as values of '+' is 2+2+1=5.

The number 5 means that if we go through 5 values at the right of the '+' operator in the Prefix string, we will find the **End of [THAT]**, or the position of '+' in the Postfix string.

In the program:

Using this idea, because we have got the Prefix List previously, we will loop from the start of the Prefix List, for each operator met, initialize `int desc=2`.

Then for each next operator met, `desc++`; for each next value met, `desc--`.

When `desc==0` we know that it has gone to the **End of [THAT]**. And now all we have to do is switch the position of the operator to the **End of [THAT]**.

Example:
```
Prefix: + - c 7 * 8 - ^ 7 P / 6 - 5 d
          ↑                         ↑
        [OP]                End of [THAT]
          ↑                         ↑
int desc=2 3 2 1 2 1 2 3 2 1 2 1 2 1 0
```

So the operator `+` will be relocated to the position right after `d`.
Pseudo-code: *(full function is in the source code)*

```
void PrefixToPostfix(List &MyList, int NumberOfOperators){
   while(NumberOfOperators>0)
   {
       Find the first operator from the left(head)
           Update something to the operator so it cannot be trigger again
           int desc=2;
           Go next to the right
               If meets an operator
                   desc++;
               Else
                   desc--;

               If desc==0
                   Switch the original operator to the current position

       NumberOfOperators--;
   }
}
```

The function above will change the Prefix List to Postfix List, and the final job is to return the string from the Blocks in MyList.

```
   string OUTPUT;
   for(Block* decoy=MyList.head->next;decoy!=NULL;decoy=decoy->next)
   {
       OUTPUT+=decoy->content+" ";
   }
   return OUTPUT;
```

If we just need Prefix, ignore the process changing from Prefix List to Postfix List, then we have completed the program that gets the input Infix string and turns it to Prefix or Postfix string.

4. The function build from ideas above:

Pseudo-code of Infixto(INPUT,mode) function: *(full function is in the source code)*

```
string InfixTo(string INPUT, string mode){
    if(mode!="prefix" && mode!="postfix")
    {
        cout<<"\n! Wrong mode !\n";
        return INPUT;
    }
    string OUTPUT;

    //Extracting Blocks from input string
    int brackets_value=0;
    int lefter_value=INPUT.size();
    int counter=0;
    int NumberOfOperators=0;
    List MyList;
    for(int i=0;i<INPUT.size();i++)
    {
        ...(code given above)
    }
    //Obtained MyList with Blocks of values and operators

    int max;
    while(true)
    {
        ... (pseudo-code given above)
        //Use SWITCH(THIS,OP,THAT) to transfer MyList to Prefix List
    }

    if(mode=="postfix") PrefixToPostfix(MyList,NumberOfOperators);

    for(Block* decoy=MyList.head->next;decoy!=NULL;decoy=decoy->next)
    {
        OUTPUT+=decoy->content+" ";
    }

    return OUTPUT;
}
```

III. Problem with input string

Before extracting Blocks from the input string, the input string will go through a Check(INPUT) function, which will return “?” if it has syntax errors.

1. Invalid characters, wrong brackets, multiple * / ^, multiple Alphas and Digits, final character is an operator:

Input string accept the character which is:

- From the Latin alphabet
- From 0 to 9
- An operator
- A dot(float numbers)
- Brackets

(and ‘@’, which will be mentioned later)

Condition of brackets:

- When going from left to right, the number of open brackets is always higher than or equal to the number of close brackets.
- At the end, the number of open brackets must equal to the number of close brackets.

Never have multiple * / ^ like “a**b”, “c//^d”.

Never have multiple Alphas and Digits like “ab”, “a9-4b”.

Final character cannot be an operator.

These errors can be easily detected, the algorithm is in the source code.

2. Duplicate + or/and -, problem with negative numbers:

We will simplify multiple '+' and '-' to single + or - by using simple rules:

- ++ ⟶ +
- +- ⟶ -
- -+ ⟶ -
- -- ⟶ +

```
for(int i=0;i<INPUT.size()-1;i++)
{
    if(INPUT[i]=='+' && INPUT[i+1]=='+') {INPUT[i]=' ';INPUT[i+1]='+';}
    if(INPUT[i]=='-' && INPUT[i+1]=='-') {INPUT[i]=' ';INPUT[i+1]='+';}
    if(INPUT[i]=='-' && INPUT[i+1]=='+') {INPUT[i]=' ';INPUT[i+1]='-';}
    if(INPUT[i]=='+' && INPUT[i+1]=='-') {INPUT[i]=' ';INPUT[i+1]='-';}
}
//The process above will create blanks, so we
//should erase all blanks left from solving +-+-
string::iterator it=INPUT.begin();
for(int i=0;i<INPUT.size();i++)
{
    if(INPUT[i]==' ')
        INPUT.erase(it);
    it++;
}
```

Negative numbers:

In case of "a*-b", "-10/-7", we will let that minus character be a part of the content(name) of the value Blocks, not an operator anymore. The idea is to change that '-' to '@' so when extracting Blocks from string, it will not think of that as an operator.

```
if(INPUT[0]=='+') INPUT.erase(INPUT.begin());
if(INPUT[0]=='-') INPUT[0]='@';
for(int i=0;i<INPUT.size();i++)
{
   if((INPUT[i]=='*' || INPUT[i]=='/' || INPUT[i]=='^' || INPUT[i]==' ' ||
   INPUT[i]=='(') && (INPUT[i+1]=='-'))
      INPUT[i+1]='@';
   if((INPUT[i]=='*' || INPUT[i]=='/' || INPUT[i]=='^') && (INPUT[i+1]=='+'))
      INPUT[i+1]=' ';
}
```

```
   //Make A@B become A-B (but not .../@B... or ...*@B..., this will be solve in
the extracting Blocks process)
   for(int i=0;i<INPUT.size()-2;i++)
   {
       if( !(INPUT[i]=='*' || INPUT[i]=='/' || INPUT[i]=='^' || INPUT[i]=='(')
           &&
     INPUT[i+1]=='@' && !(INPUT[i+2]=='*' || INPUT[i+2]=='/' || INPUT[i+2]=='^'))
           INPUT[i+1]='-';
   }
```

There will be cases when a normal operator A-B can be changed to A@B and this will lead to wrong output, so we must make sure to change back if this happens like above.

3. Floating numbers:

Floating number can have just 1 dot inside, and the dot cannot be at the start or at the end of a number.

```
   bool Dot;
   if(INPUT[0]=='.') return "?";

   for(int i=0;i<INPUT.size()-1;i++)
   {
     if( (!isDigit(INPUT[i]) && INPUT[i+1]=='.')
         ||
         (INPUT[i]=='.' && !isDigit(INPUT[i+1])) )
          return "?";

     if(INPUT[i]=='.' && INPUT[i+1]=='.') return "?";
     //If this part is a number, check if there's many dot in there
     if(i>0 && !isDigit(INPUT[i-1]) && isDigit(INPUT[i]))
     {
        Dot=false;
        for(int j=i;j<INPUT.size() && (isDigit(INPUT[j]) || INPUT[j]=='.');j++)
           {
               if(Dot && INPUT[j]=='.') return "?";
               if(INPUT[j]=='.') Dot=true;
           }
      }
   }
```

4. Space between values:

In infix notation, there cannot be any two value Blocks are next to each other, strings like “A+B 10*C” or “12/7(C^D)” can pass through the checking process and cause wrong output.

We can solve that by simply thinking that any string that has n operator Blocks should have n+1 value Blocks. The string “12/7(C^D)” will produce 2 operator Blocks and 4 value Blocks.

This problem will be checked after the extracting Blocks process because it can pass through the Check(INPUT) function.

Note that we have 2 variables `counter` to count the total of Blocks and `NumberOfOperators` to count the total of Operators.

```
INPUT=Check(INPUT);
if(INPUT=="?") return "Syntax error";

...(Extracting Blocks process)

if(counter-2*NumberOfOperators!=1) return "Syntax error";
```