EECS 211 Program 8
Due: Monday, June 4, 2012

In this last assignment we will handle expressions that involve test-and-branch (**if** and **if-else** statements) and looping (**while** statements). We will also handle left and right brackets ({ and }).

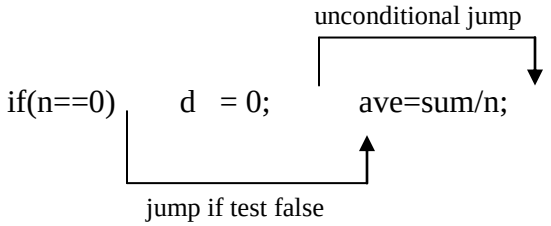
Background:

Representing branch and loop in postfix:

In basic postfix expressions evaluation proceeds one symbol after another. The next symbol to be processed is always the next symbol in the expression. The test-and-branch and looping operations require a new type of constant, an address value, and the use of conditional and unconditional jumps in the execution of a program. That is, under certain conditions, the next symbol to be processed during evaluation is NOT the next symbol in the expression. Consider the following if-else statement:

```
if(n==0)   d = 0;
else       ave = sum/n;
```

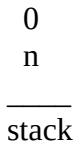
If n did have the value 0, then after setting d to 0 the execution has to skip around the else part. On the other hand, if n was different than 0 the execution has to skip around the d=0 statement. We have already seen that code is linear despite the fact that we type it into text files using lines and indentation. If we write the above statement on a single line and annotate it with arrows to indicate the "skip" feature, it would look like



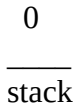
We introduce three new operators – if, while and jump. The two conditional operators – if and while – actually perform identically; they both examine the result of a test and jump to a new position in the postfix expression if the result indicates false. Thus, if and while are binary operators, taking as operands an integer (0 for false, non-zero for true) and a number (which tells the position in the postfix to jump to if the first argument is 0). The jump operator takes a single argument – a number that tells the position in the postfix; the jump operator always causes evaluation to continue at the position indicated, i.e., is an unconditional jump. Assuming the postfix for the above if-else statement started at postfix position 50, the postfix would be:

n	0	==	60	if	d	0	=	65	jump	ave	sum	n	/	=	
50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65

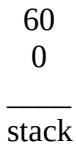
The numbers below show the position in the postfix expression of the corresponding symbol above. Now consider how the above postfix would be evaluated, and suppose n has the value 5. The first two symbols will simply be pushed onto the evaluation stack:



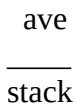
The compare-equal operator is applied to the top two elements. In this case the result is false (0), so after evaluation the stack looks like:



The next element of the postfix, 60, is a constant, much like any other constant, so it is simply pushed onto the stack:



Next, the if operator is performed. The second level of the stack represents the result of the test, true or false; in this case it is false. The top level of the stack gives the position of the next postfix symbol to be evaluated in case the test is false. The if operator removes these two elements. In this case the test was false, so the next symbol to be processed is the one at position 60 – the variable ave.

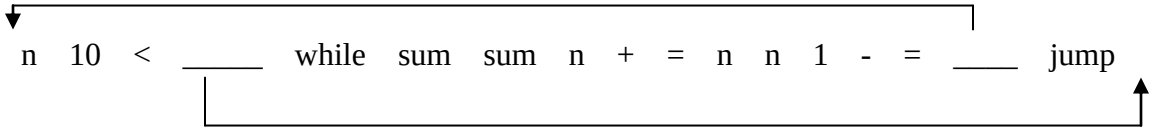


From this point on, the execution proceeds normally.

A while statement has a similar format except that at the end of the body of the loop there is an unconditional jump back to the beginning of the test. For example, the while statement

```
while(n<10) {
        sum = sum + n;
        n = n-1;
}
```

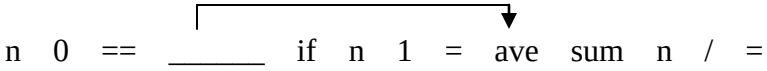
is represented in postfix notation as



Finally, an if-statement with no else part has a form similar to the full if-else statement except that there is no unconditional jump and the false target is the next position after the true part. For example, the following infix code

```
if(n==0)  n = 1;
ave = sum/n;
```

is represented in postfix notation as



Translating branch and loop from infix to postfix:

First, we make a number of remarks.

1. The two tokens, "if" and "while" correspond to operators. As such, they will be assigned precedences.
2. In a correctly formed program, all of the other operators occurring in an if- or while-test will occur inside parentheses. Therefore, we can assign the precedence for these two operators in any way we like as long as it is less than the parentheses boost. However, we will assign the precedences to be 1.
3. In a correctly formed program, the token following "if" or "while" is "(", and the end of the if- or while-test occurs exactly at the first right parenthesis at which the precedence boost returns to 0. At that point, because our algorithm from Program 6 has the right parenthesis process all the operators on the operator stack that occurred inside the parentheses, after that processing there should be only one operator left – the "if" or "while".
4. if- and while-statements can be nested in arbitrary combinations.
5. Left and right brackets group statements and effectively isolate structures inside the brackets from those outside the brackets. Thus, a semicolon occurring inside a pair of brackets terminates a statement inside the brackets but does not terminate a structure or statement outside those brackets. In order to accomplish this during the infix-to-postfix translation, "{" tokens will be pushed onto the operator stack and will act like barriers when popping operators. In a correctly formed program, when a "}" token is encountered during translation, the top element of the operator stack should be the matching "{" token, which will be popped.
6. The exact format and details for an if expression in postfix form depends on whether that if statement has an else part or not. Therefore, when moving the if-operator from the operator stack to the postfix expression it will be necessary to look at the next token in the infix expression. Moreover, the outer loop of the translation will need to remember whether or not the next token has already been read.
7. The branch address for an if, while, or unconditional jump around an else part is unknown at the time the translator inserts the if/while/jump operator into the postfix. Therefore, the space for the address will be provided when the operator is inserted, but the value will be set at a later stage. Therefore, the location of the

empty address space needs to be saved, and, in fact, because of the possibility of nesting of if and while statements these empty spaces need to be remembered on a stack. We will use the operator stack for this purpose and introduce three new tokens – ifjump, elsejump, and whilejump. Note these tokens are used internally only; they do not occur in the infix expressions. They will be used to remember where gaps were left in the postfix translation that need to be filled in.

We can now summarize the additions and changes to the existing algorithm to handle if, while and brackets.

1. A left bracket ({) is pushed onto the operator stack.
2. For a close parenthesis, after popping the operators that were inside the parentheses, check if the new precedence boost is 0. If so, check if the top level of the operator stack is "if" or "while". If yes, leave a space in the postfix expression for the jump address (to be filled in later), move the operator from the operator stack to the postfix, and push an ifjump or whilejump with the appropriate reference to the space in the postfix expression onto the operator stack.
3. For a right bracket (}), the matching left bracket is the top element of the operator stack. It should be popped. After that, the processing depends on the token following the bracket.
    a. If the next token in the input stream is "else", pop to first ifjmp, leave a space in the postfix, add the operator jmp to postfix, fill in ifjmp hole to point after the new jmp, pop the ifjmp from the operator stack, and push an elsejump with appropriate reference to the new empty space onto the operator stack.
    b. If the next token in the input stream is not "else", just continue popping the operator stack until the next "{" or the stack is empty.
4. For a semicolon, the operation is the same as for right bracket except that the top of the operator stack will not be left bracket.
5. For an "else" token in the input stream, there is no need to do anything because all the required processing is handled at the semicolon or right bracket token.
6. For a "while" operator in the input stream, it is necessary to remember the current position in the postfix expression so that the backward jump at the end of the while loop can be filled in correctly.
7. In items 3 and 4, processing an ifjump or elsejump means just filling in the hole with the next postfix position. Processing a whilejump means first adding the jump-back then filling in the hole.

The Appendix has three examples to illustrate these modifications.

Assignment:

Add the following definitions to definitions.h:

STOPATFIRSTIFJMP	1
DONTSTOPATFIRSTIFJMP	0
DONTPROCESSJUMPS	0
PROCESSJUMPS	1
JUMP	243

The first four are operational constants, that is, constants used to control how some of the functions described below will operate. The last constant will be the label on a new case in the postfix evaluation switch.

Add a new class, JMP, publicly derived from the SYMBOL class with the following:

New data members: forwardReference and backwardReference, both of type **int**.

Function members:

Constructor: three arguments are the **char*** for the name and two **int** arguments for the forward and backward reference.

Print function: no arguments.

A function that returns the forward reference data member.

A function that returns the backward reference data member.

A function that sets the forward reference data member.

This class will be used to implement the ifjump, whilejump, and elsejump elements that need to be pushed onto the operator stack. The symbol name will be one of "ifjump", "whilejump" or "elsejump" so that other functions that process the **JMP** objects will know how to handle the object.

In files symbolist.h and symbolist.cpp:

- Change LISTLEN from 100 to 200.
- Add the following functions to the **symboList** class:
  - A function that returns the integer indicating the next available position in the list. This will be used to identify and record the jump targets.
  - A function that takes a pointer to a symbol, s, and an integer, k, and replaces the symbol currently at position k in the list with s. This will be used to fill in the jump targets when the destinations are finally known.
- Add the following function to the **symbolStack** class:
  - A function that takes an **int** as argument representing the forward reference target of a whilejump object and updates the forwardReference data member.

In files system_utilities.h and system_utilities.cpp make the following additions and modifications:

- In the popOperatorStack function add two new arguments, both of type **int**. One argument tells whether or not this function should stop processing after handling the first ifjump object (i.e., if the translation has reached the end of an if statement for which there is NO else). The second argument tells whether or not this function should terminate when it encounters a **JMP** object. This function and the ones that call this function should use the constants mentioned in item 1 of the assignment. See the comments section for more discussion of this function.
- Add a new function – void checkForIfWhile(symbolStack *stk, symbolList *postfix). This function checks if the top of the operator stack is an "if" or "while" operator. If so, pop that symbol off the operator stack, then add a constant and the appropriate operator to the postfix stack. If the operator was an if, create a new **JMP** object and push onto the operator stack. If the operator was a while, the corresponding whilejump object will already be on the operator stack (see below), and this function will update the forewardReference data member of that **JMP** object.

In the infix-to-postfix translation, make the following additions and modifications:

- Add a mechanism that will tell if the next token has already been read or not.
- In the case for operators, check if the operator is a while. If so, push a whilejump object onto the operator stack. (Recall, you must do this now because you need to remember where the test starts.) Then pop the operator stack as usual but with parameter set for not processing jumps. (Note, if popOperatorStack is not processing jump objects at all, then the argument that tells whether to stop at the first ifjump or not is irrelevant.)
- In the case for semicolon, read the next token from the input and check if it is "else". This determines whether popOperatorStack will stop at the first ifjump or not. The other new argument should be set to process jumps.
- In the case for right parenthesis, popOperatorStack should be set not to process jumps. Also, after popping the operator stack, if the precedence boost is 0 call checkForIfWhile.
- Left brackets should be pushed onto the operator stack.
- Implement the case for right bracket.

In the postfix evaluation function: Implement the if, while, and jump operators.

Comments, suggestions, and hints:

As always, implement these a few at a time. For example, you could implement the new **JMP** class and then test it with throw away code (see below). You could then implement

the new functions in symbolList and again test with throw away code. Then you could tackle the changes to popOperatorStack and the remaining code for the translation. You may want to disable the evaluation of expressions until you get the translation right. (For example, you could just return from main after printing the postfix expression.) Finally, implement the evaluation of if, while, and jump.

It is often useful to do a quick test of a new section of code without using the machinery (i.e., code, in our case the big switch statement). On the other hand, during such quick testing you may not want to modify or hurt or otherwise mess with the existing code. Here's a trick I often use. In the main function after the declarations and before the existing code starts, add the following:

```
int k=0;
if(k!=0) {

        … place your throw-away test code here

        return 0:
}
```

The test on k will be true, and your program will execute the code inside the brackets, at the end of which your program will quit. For testing, say, the JMP class, the statements inside the brackets could declare some variables, call the various member functions and print out the results. You could use the debugger to step into these function calls to verify that they are working the way you intended. When you are satisfied, simply delete all of the above lines, and your main.cpp file is back to normal.

The modified popOperatorStack function needs to make the following checks:

- Check for the top stack element being left bracket. If so, do not process the symbol but simply exit the function. The bracket itself should be popped by the main switch case for right bracket after calling popOperatorStack.
- If the top element is one of the three jump objects and the function is not supposed to process jumps, return immediately.
- If the function is supposed to process jumps and the top of the stack is a jump object, branch to an appropriate case to handle it. Note, each of the three types of jump gets handled in a slightly different way. For example, for an ifjump object you need to check the new argument that tells whether or not to stop at the first ifjump. For a while, you need to create a backward jump.

If none of the above cases hold, then it is a regular operator and should be processed like in the previous assignments.

Test file p8input.txt:

```
VAR x;  VAR y;  VAR a;  VAR b;
x=1;  y=7;  a=0;    b=0;
while(x<y){
  if(x/2*2 == x) { a = a+x; cout a;}
  else           { b = b + x*x;  cout b; }
  x = x+1;
}
if(b>a) while(b>0) {cout b; b = b-1;}
a=5;
while((b=1)(a=a-1)a>0)while(b<a) { cout a; cout b; b=b+1;}
```

Appendix.

In these examples, the position of the empty space for the various jumps is indicated by a number concatenated to the jump symbol in the operator stack. For example, ifjump3 means position 3 in the postfix expression is the space provided for an if conditional branch address. For a while jump, the notation provides two postfix references – the position where the test starts and the position where the space was left for the while jump. Thus, in the first example, whilejump(12,15) means that the first postfix symbol for the while expression is in position 12 and the empty space for the branch address is at position 15.

Example 1: if(a>0) if(b>x+y) while(c<0) if(d>a) x = y+z; a = 0;

At first):

postfix: a 0
opstack: if >

Close parenthesis causes > to be popped. Boost is zero and there is an if, so pop opstack, leave space for address and add if to postfix, then push ifjump reference

postfix: a 0 > ____ if
opstack: ifjump3

At second):

postfix: a 0 > ____ if b x y
opstack: ifjump3 if > +

Close parenthesis causes + and > to be popped into postfix. Boost is zero and there is an if, so pop opstack, leave space for address and add if to postfix, then push ifjump reference

postfix: a 0 > ___ if b x y + > ___ if
opstack: ifjump3 ifjump10

At third):

postfix: a 0 > ____ if b x y + > ___ if c 0
opstack: ifjump3 ifjump10 while(12) <

Close parenthesis causes < to be moved. Boost is zero and there is a while, so pop opstack, leave space for address and move while to postfix, then push whilejmp reference to opstack.

postfix: a 0 > ___ if b x y + > ___ if c 0 < ___ while
opstack: ifjump3 ifjump10 whilejump(12,15)

At fourth):

postfix: a 0 > ____ if b x y + > ___ if c 0 < ___ while d 0
opstack: ifjump3 ifjump10 whilejump(12,15) if >

Close parenthesis causes > to be moved to postfix. Boost is zero and there is an if on top of opstack, so pop opstack, leave space for address and add the if operator to the postfix, then push ifjump reference/

postfix: a 0 > ___ if b x y + > ___ if c 0 < ___ while d 0
> ___ if
opstack: ifjump3 ifjump10 whilejump(12,15) ifjump20

At the first semicolon:

postfix: a 0 > ___ if b x y + > ___ if c 0 < ___ while d 0
> ___ if x y z
opstack: ifjump3 ifjump10 whilejump(12,15) ifjump20 = +

Next token is not else, so this terminates all if/while structures. Pop the two operators, then pop all the jumps and fill in with next postfix position. For whilejump(12,15), add 12 and JMP in postfix before filling in position 15, and then later jumps go after the two new spaces.

postfix: a 0 > 29 if b x y + > 29 if c 0 < 29 while d 0
> 27 if x y z + = 12 jmp
opstack:

Example 2: if(a>0) if(b>x+y) while(c<0) if(d>a) {x = y+z; … } a = 0;

Same except after the fourth right parenthesis the opstack will be

opstack: ifjump3 ifjump10 whilejump(12,15) ifjump20 {

Any semicolons and right brackets inside will be blocked from going below this {, but the matching } will pop it and would continue processing if/else until the next { or the stack was empty.

Example 3: if(a>0) while(b>0) b = b-1; else x = x+y; a = 0;

After processing first):

postfix: a 0 > ___ if
opstack: ifjump3

After processing second):
postfix: a 0 > ___ if b 0 > ___ while
opstack: ifjump3 whilejump(5,8)

When processing reaches the first semicolon we have:
postfix: a 0 > ___ if b 0 > ___ while b b 1
opstack: ifjump3 whilejump(5,8) = -

The next token is else, so we need to pop opstack to if and replace. The while operator causes the jump back to be added to the postfix. Pop regular operator elements of opstack first and the whilejump operator, then make an unconditional JUMP that will go around the else part. Finally, when get to ifjump, fill in and replace by elsejump and reference.
postfix: a 0 > 19 if b 0 > 17 while b b 1 - = 5 jmp
___ jmp
opstack: elsejump17

Target for the while jump is 17, but target for the if jump is 19. Note, there could be several while jumps, and they all have same target. So pop while jumps until reach ifjump.

At the next semicolon: This is end of else statement.
postfix: a 0 > 19 if b 0 > 17 while b b 1 - = 5 jmp
___ jmp x x y
opstack: elsejump15 = +

Pop operators and process elsejump:
postfix: a 0 > 19 if b 0 > 17 while b b 1 - = 5 jmp
24 JMP x x y + =
opstack: