```
/*
ARM Development Programming Assignment - EE306H
Student Name: Maaz Ahmed
Student UTEID: ma69299
Task1:      Convert a given fully-parenthesized Infix expression
          to Postfix
Task2:      Evaluate the expression for specific values of the variables
Assume:     The infix expression is a string with variables that
        are single lower-case alphabets [a-z]. operators allowed
            are +,-,* and /
            Variable values are stored in a value table - VTab
                  an array of records where each record has two
                  attributes: symbol and value symbol is:
                  one Character (8-bit) and Value
                  a signed 32-bit number.
*/
      .syntax unified
      .data
// These are the outcomes of your two tasks
Result:           .space 4  // This is the final evaluated result
PostFix:    .space 20 // Here goes the postfix expression
      .text
      .global main
      .type main, function
// These are the inputs to your two tasks
InFix:      .string "(a+(b*((c-d)/f)))" // The InFix expression
                              // PostFix: abcd-f/ *+ for this case
                    //                  ^no blank
      .align 2
// A Value Table of values for the variables
VTab: .byte 'a'
            .long 2
            .byte 'b'
            .long -3
            .byte 'c'
            .long 4
            .byte 'd'
            .long 6
            .byte 'f'
            .long 2
            .byte 0           // Expression result is -1 for this case
      .align 2
main:
      bl Task1    // Should check R0 after Task1 is done
                        // to see if it was a success or failure
      bl Task2
      b .

/* ++++++++++++++++++Task 1 subroutine ++++++++++ */
/*
            Algorithm: Convert Infix to Postfix

            (1) Read next character cc from Infix
            a. If cc is \0, goto Step 3
            b. If cc is '(' (left-brace), push cc on Stack
            c. If cc is an operator, push cc on Stack
            d. If cc is a variable, write it to Postfix
            e. If cc is a ')' (right-brace)
                        - Pop from Stack write to Postfix
                    until a left brace (Note: do not print brace)

            (2) Goto Step 1

            (3) Write NULL (\0) to Postfix – Done

      Output: R0 has 1 for success; 0 for failure
*/

Task1:
      push {r4, r5, lr} // saving the registers we will be using
      ldr r4, =InFix // r4 now points to the infix string
      ldr r5, =PostFix // r5 now points to the postfix buffer

T1_Loop:
      ldrb r0, [r4] // (1) read the next character conditian code
      adds r4, r4, #1 // increments the inFix pointer

      cmp r0, #0 // compares to see if the cc is \0
      beq T1_Done // if so go to step 3

      cmp r0, #'(' // b. if cc is '('
      beq T1_Push // push the cc on the stack

      cmp r0, #')' // e. if the cc is ')'
      beq T1_PopUntil

      // now we want to check if variable

      push {r0} // save the cc before the call
      bl IsAlpha
      cmp r1, #1
      pop {r0} // restoring the cc
      beq T1_Write // d. if its a variable, write to post InFix

      // now we want to check if operator

      push {r0} // saves the cc
      bl IsOp
      cmp r1, #1
      pop {r0}
      beq T1_Push // if its an operator, push the cc onto the stack

      b T1_Loop // (2) go to step 1 (ignore the other chars)

T1_Push:
      push {r0} // push the cc to system stack
      b T1_Loop

T1_Write:
      strb r0, [r5] // write the cc to post InFix
      adds r5, r5, #1 // incremnets the post fix counter
      b T1_Loop

T1_PopUntil:
      pop {r1} // pop from the stack
      cmp r1, #'(' // if it is a '(' , we wanrt to stop popping
      beq T1_Loop // back to the main loop (we dont print brace)
      strb r1, [r5] // write the popped character to postfix
      adds r5, r5, #1 // increment the post fix counter
      b T1_PopUntil // keep popping

T1_Done:
      movs r0, #0
      strb r0, [r5] // (3) write NULL to the post fix
      movs r0, #1 // returns scucess (1)

      pop {r4, r5, pc}


/*--------------End of Task1 subroutine -----------*/

/* ++++++++++++++++++Task 2 subroutine ++++++++++ */
/*
      Algorithm: Evaluate to a Postfix Expression

            (1) Read next char from Postfix into cc
            If cc is '\0' then goto 5

            (2) If cc is a variable, push its value on the Stack

            (3) If cc is an operator X
            - Pop 2 elements off the Stack
            - Perform operation X
            - Push result on the Stack

            (4) Goto Step 1

            (5) Pop value from Stack and write to Result
*/
Task2:
      push {r4, lr}
      ldr r4, =PostFix // R4 points to the PostFix string

T2_Loop:
      ldrb r0, [r4] // (1) reads next char
      adds r4, r4, #1

      cmp r0, #0 // if its '\0' , we want to go to 5, which is next line
      beq T2_Done

// this will be check IsAlpha
push {r0}
bl IsAlpha
cmp r1, #1
pop {r0}
beq T2_Var // (2) if its a variable

// this will check IsOp

push {r0}
bl IsOp
cmp r1, #1
pop {r0}
beq T2_Op // (3) if its an operator

b T2_Loop
```

```
T2_Var:
      bl Value // gets the value of the varialbe into r0
      push {r0}
      b T2_Loop
T2_Op:
// we want to pop 2 elemtns. and the stack is lifo
      pop {r2} // this will get the second operand the one on the right side
      pop {r1} // this will get the first operand, the one on the left side

      cmp r0, #'+' // just checks the symbol for plus
      beq DoAdd
      cmp r0, #'-' // just checks the symbol for sub
      beq DoSub
      cmp r0, #'*' // just checks the symbol for mult
      beq DoMul
      cmp r0, #'/' // just checks the symbol for div
      beq CallDiv
      b T2_Loop

DoAdd:
      adds r1, r1, r2
      push {r1}
      b T2_Loop
DoSub:
      subs r1, r1, r2
      push {r1}
      b T2_Loop
DoMul:
      muls r1, r2
      push {r1}
      b T2_Loop
CallDiv:
      bl Divide // divide r1 by r2, r1 will have the quotient
      push {r1}
      b T2_Loop

T2_Done:
      pop {r0}
      ldr r1, =Result // (5) pop the final result
      str r0, [r1] // write to the result
      pop {r4, pc}
/*--------------End of Task2 subroutine -----------*/

/*  Subroutine IsAlpha:
      Purpose: Checks if the given input is a variable
      Input: R0 has character to check
      Output: R1 has 1 if R0 is a variable: [a-z] 0 otherwise
*/

IsAlpha:
    push {lr}
      movs r1, #0 // default result will be zero
      cmp r0, #'a'
      blt IA_End // if its less than a, then its not alpha
      cmp r0, #'z'
      bgt IA_End // if its greater than z, its not alpha
      movs r1, #1 // in this case it is alpha
IA_End:
      pop {pc}


/*  Subroutine IsOp:
      Purpose: Checks if the given input is an operator
      Input: R0 has character to check
      Output: R1 has 1 if R0 is an operator (+,-,*,/) 0 otherwise
*/

IsOp:
    push {lr}
      movs r1, #0 // default is just gonna be zero
      cmp r0, #'+'
      beq IO_Yes
      cmp r0, #'-'
      beq IO_Yes
      cmp r0, #'*'
      beq IO_Yes
      cmp r0, #'/'
      beq IO_Yes
      b IO_End
IO_Yes:
      movs r1, #1
IO_End:
    pop {pc}


/*  Subroutine Divide
      Purpose: Divide R1 by R1
      Inputs: R1 an R2
      Output: R1 has the quotient
*/
Divide:
    push {r4,r5,lr}
    movs r5, #0     // keep quotient here
    movs r4, #0     // to flip result or not
    cmp  r1, #0
    blt  NrNeg
    cmp  r2, #0
    bgt  DoDiv
    // here means NrPos and DrNeg
    subs r2, r5, r2  // flip Dr
    movs r4, #1
    b    DoDiv
NrNeg:
    cmp  r2, #0
    blt  NrDrNeg
    // Here means NrNeg and DrPos
    subs r1, r5, r1  // flip Nr
    movs r4, #1
    b    DoDiv
NrDrNeg:
    subs r1, r5, r1  // flip Nr
    subs r2, r5, r2  // flip Dr
DoDiv:
    subs r1, r2
    bmi  DivDone
    adds r5, #1
    b    DoDiv
```

```
DivDone:
    movs r1, r5
    cmp r4, #0
    beq DivDoneDone
    movs r4, #0
    subs r1, r4, r5
DivDoneDone:
    pop {r4,r5,pc}

/*  Subroutine Value:
      Purpose: Finds the value of a variable
      Input: R0 has the variable [a-z]
      Output: R0 has the value or 1000 if not found
*/

Value:
    push {lr}
      ldr r1, =VTab // load the address of VTab
Val_Loop:
      ldrb r2, [r1] // loads the symbol character
      cmp r2, #0 // check for the end of the table (this will be a null byte bc
null term)
      beq Val_NotFound

      cmp r2, r0 // check if the symbol matches r0
      beq Val_Found

      adds r1, r1, #5 // this is to move to the next record, theres 1 byte
character, + 4 byte-value,  so 5 total
      b Val_Loop

Val_Found:
      adds r1, r1, #1
      ldr r0, [r1]
      b Val_Exit

Val_NotFound:
      ldr r0, =1000 // return 1000 if not found

Val_Exit:
    pop {pc}

      .end
```