Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
5/12/21

Computer Architecture ECE-251: Project 3

Requirements:

A basic calculator that considers the standard "PEMDAS" rule. The calculator will be invoked by the following:
$> ./calc <operation>
where operation is defined as:
operation = <operand> <arithmetic operation> <operand> , ... up to 4 times.

Exponents are supported via the ^ character (i.e. 3^2 would be 3*3). Floating point numbers are also supported.

Overall Architecture of Program:

The program begins by converting the input into post-fix or reverse polish notation, which can be more easily computed, as it handles PEMDAS well. For each character in the input, the program loops through, if it is a symbol all the previous characters (which are numbers) are added to a queue (but kept as strings, so it is possible to tell the difference between a number and a symbol), and the symbol itself it loaded onto the stack. When the whole expression has been processed, the conversion to post fix takes place, by incrementally popping from the stack and pushing to the queue, which leaves the queue with an operatable post-fix expression. The strings are then converted into floats if needed, and the operations determine which section of the code to jump too, (add, sub, multiply, divide, or exponents). Finally, the final answer is printed.

Overall Algorithm Design:

Post Fix Notation

Reverse Polish Notation, or Post Fix Notation is a mathematical notation in which operators follow their operands. These removes the need for parentheses (unless each operator takes a variable number of operands). Post fix notation has been found to lead to faster calculations because they do not need to process parenthesized, so fewer overall operations need to be considered.

Shunting Yard Algorithm

In order to convert from the standard infix notation to a post-fix or reverse polish notation the Shunting Yard Algorithm was used. For example, the following expressions converted to post-fix notation, would be.

Standard Infix Notation	Post Fix Notation
3 + 4	3 4 +
3 + 4 × (2 − 1)	3 4 2 1 − × +

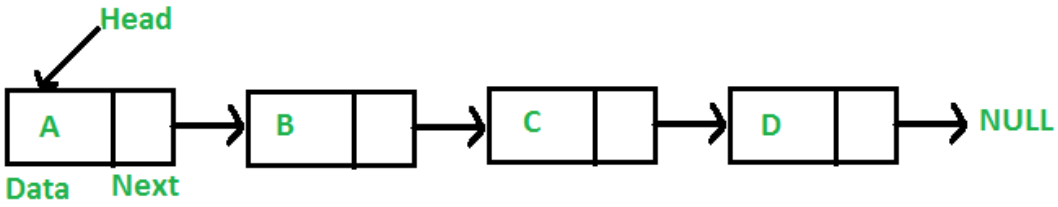
Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
5/12/21

3 + 4 × 2 ÷ (1 − 5) ^ 2 ^ 3	3 4 2 × 1 5 − 2 3 ^ ^ ÷ +

This notation makes it easy to process the order of operation. The Algorithm utilizes a stack and queue to do this conversion. First, the input is read, and the numbers are added to the output queue, and operator are pushed onto the stack. After the expression is read operators of the same or higher precedence are popped off the stack, such that the order of operations is correct.

Stack and Queues

Stacks and Queues are specific (as implemented in this project) implementations of the broader data structure Linked Lists. Traditionally, lists of data stored in an array are stored consecutively in memory, such that given the specific index of an element, the corresponding element can be found in constant time by multiplying the size of the data by the index number and adding it to the memory location of the array (this is why arrays start and index 0) However, this leads to linear insertion times, because in order to add something to the begin or middle of a list, all the elements following must be moved over. The linked list solves by creating links between nodes.



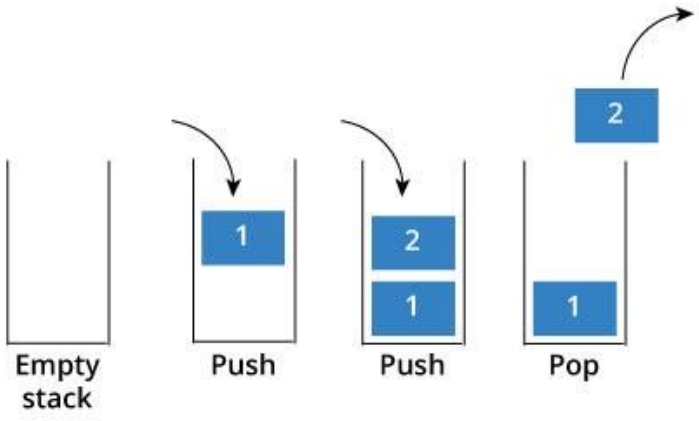
Nodes are created in the heap, which contain pointer to the next element, this allows for constant time insertion to either end of the list, by simply adjusting the location of the pointers.

A stack and a queue, both utilize this because they only insert from one end of the list.

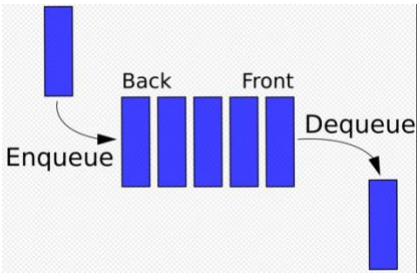
Stacks:

Stacks are LIFO (Last In First Out) meaning the first thing inserted into a stack is the first thing that will be returned when popping from the stack (See the accompanying diagram). Similar to a stack of plates, adding or removing is only possible from the top. Stacks support two operations, pushing which adds and element, and popping, which removes the most recently added elements.

Queues:



Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
5/12/21
Queues, however, are FIFO (First In First Out), just like a line or queue at a store. In our implementation of a singly linked list, for queues items are inserted at the end and removed from the beginning (in order to maintain a constant time, but the functionality remains the same). Queues support two operations, enqueue which adds an element to the back of the queue, and dequeue which removes an element from the queue.



Design Decisions:

Floating Point Numbers

Originally, we had several options as to how to deal with floating point numbers. One of the notable contenders was to convert all the numbers to integers prior to operating on them as if they were integers and then convert back. We would count the number of decimal points after the decimal point for all numbers, take the greatest one and multiply all numbers by that. This could be done by moving the decimal place and adding the required number of zeros. After doing all the operations, the number would be divided by removing zeros and/or adding a decimal place in the appropriate area.

Although this could theoretically work, it seemed too complicated to be implemented well, and a simpler solution using the floating-point registers worked effectively. This was accomplished using the flag "`-mfpu=vfp`". The floating-point registers and floating-point instructions work in the same manner as the typical registers but have a slightly different syntax.

C Functions Utilized

I. `printf`
   a. We chose printf because it is the most versatile function that can print a string of character to standard output. Furthermore, it does not append the string with a new line character, and thus allows the input of the string to be on the same line as the output.

II. `sscanf`
   a. Takes an input of the type string, and converts either the entire string or elements in the string to the destination format of choice
   b. This is valuable because in order to use floating point numbers, the program must first parse through the string and rewrite the string in postfix notation after which the operations are performed. Operations, however, cannot be performed on variables of type string, they must be performed on data of the type integer or float.

Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
5/12/21

III. `malloc`
   a. Allocates a block of *size* bytes of memory, returning a pointer to the beginning of the block.
   b. The content of the newly allocated block of memory is not initialized, remaining with indeterminate values.
   c. Used in the creation nodes in for stacks and queues

IV. `memcpy`
   a. Copies the values of *num* bytes from the location pointed to by *source* directly to the memory block pointed to by *destination*.
   b. The underlying type of the objects pointed to by both the *source* and *destination* pointers are irrelevant for this function; The result is a binary copy of the data.
   c. The function does not check for any terminating null character in *source* - it always copies exactly *num* bytes.
   d. Used to copy data from different locations in memory at various points

V. `pow`
   a. Returns *base* raised to the power *exponent*:
   b. Used for the ^ operator

Limitations:

We do not support negative numbers; if negative numbers are used, the output is not defined. If you would like to use negative numbers, subtract from zero in parentheses. For example: the value -3 could be written as (0-3)

We do not support whitespace in the middle of operation. For example, 3 + 3 / 3 ^3, would not be supported. The operation should be rewritten as 3+3/3^3. Additionally, if no operator the output is not defined, furthermore it may produce a segmentation fault.

Program:

.data

In the .data directive the various “variables” used by the program are defined and aligned in memory.

main:

In the .text directive, the first section is called “main” which reads the user input by incrementing the stack pointer which points to the arguments passed in when executing the program. If the program is executed correctly, the first argument should be “./jag3.out” and the second argument should be the operation of numbers of which the program performs. At the end of main, r2 and r4 are initialized to zero which are used in the next section to load in the individual bytes of the string and find the size of each number.

Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
5/12/21

loop:

In loop, the program parses through the string until parenthesis or an operator are found. This is done using the ldrb instruction offset by r2, which is initially set to #0 in the main section and incremented with each loop. The program continuously returns to the loop section to continue parsing through the string after branching to the respective sections following the cmp instructions. When the program reaches the end of the string which is terminated by a null character, it will read the null byte and branch to the finalizePostfix.

ifIsSymbol:

The program branches to the ifIsSymbol section if the byte read is a symbol. The registers are pushed to the stack and the program immediately branches to the buildString section. If the string length is zero, which indicates that the program has reached the end of the string, the program branches back from buildString and confirms that it has reached the end of the string by comparing the contents in r3, which contains the last byte read and returns to the loop section to then branch to finalizePostfix. The program branches back here after pushing the string denoting a number to the queue to then move the symbol byte into r7 and branch to handlePushingSymbol to push the symbol read onto list1, which is a stack. After properly pushing the respective string/symbol to the correct list, #0 is moved into r4 to reset the counter of the string length for the next number. The program then branches back to loop to start the process over again moving on to the next number/symbol.

buildString

In the buildString section, the bytes between symbols are built into a string and pushed onto the queue. The string length, which is found in the loop section is first compared to #0 in which case the program returns immediately to ifIsSymbol. If the string length is not zero, indicating that a number is being read, malloc is called to allocate space in memory for the string which is used later to copy the number to a new point in memory. The amount of space passed down to malloc is the contents of r4, which contains the length of the string. In order to copy the string into a new memory space, the contents of r1, which contains the pointer to the first byte is added to r2, which contains the total amount of bytes read thus far and is placed in r1. Afterwards, r4, which contains the string length, is subtracted from r1 and placed in r1, and the string length (r4) is placed in r2. At this point, the parameters for the memcpy function are set up given that r0 contains the pointer to the space in memory allocated from malloc, r1 contains the pointer to the first byte of the number being read and r2 contains the string length. After memcpy is called, a null character is placed at the end of the string and the number (which is still in the form of the string) is pushed onto the queue using by branching to queuePush.

queuePush

The contents of r0 which contains the string are moved into r3 because when malloc is called in the next instruction, r0 gets overwritten. The string is pushed onto the queue by allocating 16 bytes of space from the heap and storing the value of r3 (i.e. the string representing the number) into the location in memory allocated by malloc. The contents of r5, which contains the tail pointer, is loaded into the

Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
5/12/21
address pointed to by r6 which is the address of the pointer to the tail of the queue (list 2). The new node is then stored next to the current tail pointer (i.e. at the bottom of the list which is what categorizes this list as a queue) and the program branches to queuePush2. However, if the tail pointer is null, the program skips the storing operation and branches to queuePush1.

queuePush1

In queuePush1, the new node is set as the head of list2 and the program continues linearly to queuePush2.

queuePush2

The tail is then set to be the new node and the program branches back to biuldString, after which it branches back to ifIsSymbol and calls handlePushingSymbol to push the symbol into the stack.

handlePushingSymbol:

The contents of r7, which is the byte of the symbol, is moved into r0 and the program branches to stackEmpty to check if the top of the stack is empty and returns to handlePushingSymbol with the comparison condition. If the top of the stack is empty, the program branches to handlePushingSymbol1 to create the first node in list1. If the symbol is a left parenthesis, the program branches to handlePushingSymbol1 and if the symbol is a right parenthesis, the program branches to handlePushingSymbol2. If the symbol is neither the top of the list, a left parenthesis, or a right parenthesis, the program continues linearly and branches to stackPeek to get the value at the top of the stack in r0. The symbol byte is then moved into r1 and then the program branches to precedence to compare the value at the top of the stack to the symbol that was just read. The condition is applied to the cmp instruction from the precedence section and if the value at the top of the stack is less than the current symbol being read, indicating lower precedence, the program branches to handlePushingSymbol1 to push the current symbol to the stack.** If not, the symbol at the top of the stack is popped off the stack (list1) and pushed onto the queue (list2) and the program branches back to the beginning of handlePushingSymbol.

handlePushingSymbol1

In handlePushingSymbol1, the symbol byte is moved into r0 and stackPush is called. See stackPush below. After the operator is pushed onto the stack (list1), the program branches back to ifIsSymbol.

handlePushingSymbol2

In handlePushingSymbol2, the top of the stack is popped by branching to stackPop. See stackPop below. After stackPop is called, r0 contains the top value in the stack which is compared to a left parenthesis symbol. If it is not a left parenthesis, the symbol is popped and then pushed to the queue, following the shunting yard algorithm. To push the symbol to the queue, queuePush is called, after which it branches back to handlePushingSymbol2 until the left parenthesis pair to the right parenthesis is found. Once the left parenthesis is found, the program branches back to ifIsSymbol.

Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
5/12/21

stackEmpty:

All the current data is pushed to the stack, and then if the `list1HeadPointer` is empty it leaves the cpsr to false, else true. It detects empty or not by comparing the value pointed to by `list1HeadPointer` to see if it is null.

stackPeek

The value at the top of the stack is returned without popping it (i.e. it peeks at the top value) by obtaining the value at the first node, placing that value in r0 and then branching back to handlePushingSymbol.

stackPush

The goal of this section is to insert the value at r0 into the start of list1. First the value at r0 is moved to r3, because malloc will overwrite the value. Then malloc is run to create 16 bytes of space for the new node. The current head pointer is obtained, and the data is stored in the new node and the next node is set to the current head pointer. The program then returns to where it last left off.

stackPop

stackPop pops the top value in the stack and returns it in r0. The head pointer is obtained from the =list1HeadPointer and it loads the into r0. Finally, the next node is placed as the = list1HeadPointer. The program then returns to where it last left off.

precedence

precedence returns a precedence comparison. In precedence, precedenceNum is called twice to obtain a value #0, #1, #2 or #3 to represent the precedence of the symbol that was the value at the top of list1 (stack) which is in r0 and the symbol that is currently being read which is in r1. See precedenceNum below. After branching back into precedence, the values assigned to the different symbols are placed in r3 (the number denoting the precedence of the symbol at the top of the stack) and r4 (the number denoting the precedence of the current symbol). r3 is then compared to r4 using the cmp instruction. The program branches back to handlePushingSymbol to continue with the conditions from the cmp instruction.

precedenceNum

In precedenceNum, the first time it is called, r0 contains the current value at the top of the stack. The value in r0 is then compared to every symbol and when this comparison is found to be equal to the respective symbol, a value 0-3 is placed in r0 and the program branches back to precedence. These values represent the precedence of the symbols which is used to compare the symbols to each other when the program returns to the precedence section. The second time precedenceNum is called, r0 contains the symbol that was just read and performs the same process of comparing and moving the respective values into r0.

Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
5/12/21

finalizePostfix:

The finalizePostfix section acts like a while loop in that at this point in the program, the finalizePostfix section ensures that while the stack is not empty, the remaining items are popped from the stack (list1) and pushed to the queue (list2). First, the program branches to stackEmpty to check if the stack is empty and the comparison condition in the finalizePostfix section branches to the solveExpression section, if it is indeed empty (i.e. the cspr register holds true). If not, the program branches back to stackPop and queuePush and repeats until the stack is empty and the queue contains the postfix expression in which case the program ultimately branches to solveExpression.

solveExpression:

Currently, the postfix expression is the queue. A loop is created, which stops when the queue is empty by branching to queueEmpty. queueEmpty returns a value in the cspr which, if it holds true, the program branches to finalPrint. If the queue is not empty, the program begins processing the post fix notation. queuePop is called to pop the top value from the queue. See queuePop below. It starts by determining if the next item in the queue is a pointer, and calls solveExpression1 if it is not, which means it is an operator. If the item in the queue (list2) is a pointer, the pointer is to a string which needs to be converted to a float. For debugging purposes, this string is printed out. It is then converted to a float by leveraging the libc function sscanf. Using the floating-point registers and instructions, the float is loaded into d0. The libc function malloc is then called to then store the float in memory. Afterwards, stackPush is called to push it onto the stack and then the program branches back to solveExpression to continue converting the subsequent numbers from strings into floats and evaluate the operators.

queueEmpty:

All the current data is pushed to the stack, and then if the `list2HeadPointer` is empty it leaves the cpsr to false, else true. It detects empty or not by comparing the value pointed to by `list2HeadPointer` to see if it is null.

queuePop:

In this section, the head pointer is obtained and loaded into r0. The pointer is then incremented to the next item in list2 (the queue) which is stored in the address of list2HeadPointer which means that it is set to be the next head pointer. This value is compared to #0. If r2 is not equal to #0, the program returns to solveExpression. If this value is true, meaning there are no items remaining in the queue, the tail pointer is set to the same value (i.e. null) and the program returns to solveExpression.

solveExpression1

At this point, the item popped off the queue (list2) is an operator. For debugging purposes, the symbol is printed out. The operator is then compared to the ascii values for the operators and branches to the respective sections to perform the proper operation. See the respective sections in detail below. If the value is #0, the program branches back to solveExpression.

Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
5/12/21

exponent:

If the operator is "^", the program pops the top two values from the stack which are now type-float. The pointer to the float struct (currently in r0) is moved into r8 for later use. The floats are moved into regular registers and the libc pow function is leveraged to perform the operation. The operation takes the value that was the second to the top of the stack and raises it to the power of the value that was on the top of the stack. The resultant value is then moved back into a float register and stored back into a float struct after which it is pushed to the top of the stack by calling stackPush. The program later returns to solveExpression to continue moving through the stack.

divide:

If the operator is "/", the program pops the top two values from the stack which are now type-float. Using the floating point instruction vdiv.f64, the float in d0 which holds the first item from the top of the stack is divided by the float in d1 which holds the second item from the top of the stack. The resultant value is pushed to the top of the stack by calling stackPush and then the program returns to solveExpression to continue moving through the stack.

multiply:

If the operator is "*", the program pops the top two values from the stack which are now type-float. Using the floating point instruction vmul.f64, the floats in d0 and d1 which hold the top two values on the stack are multiplied. The resultant value is pushed to the top of the stack by calling stackPush and then the program returns to solveExpression to continue moving through the stack.

subtract:

If the operator is "-", the program pops the top two values from the stack which are now type-float. Using the floating point instruction vsub.f64, the float in d1 which holds the second item from the top of the stack is subtracted from the float in d0 which holds the first item from the top of the stack. The resultant value is pushed to the top of the stack by calling stackPush and then the program returns to solveExpression to continue moving through the stack.

add:

If the operator is "+", the program pops the top two values from the stack which are now type-float. Using the floating point instruction vadd.f64, the floats in the two registers are added together and the resultant value is pushed to the top of the stack by calling stackPush and then the program returns to solveExpression to continue moving through the stack.

finalPrint:

The value at the top of the stack is the final answer, so to get the value at the top of the stack, stackPop is called. This value of type-float is then loaded into d0. Afterwards, it is moved into r2 and r3 for printing using printf. Printf is called with the %f parameter, which is located in the address of float, so that the value printed is a float. The program then proceeds to end.

Jacob Khalili
Gary Kim
Aliza Meller
ECE-251
Professor Billoo
5/12/21
end:

#0 is moved into r0, to achieve a 0-return code which indicates that the program ran successfully. #1 is moved into r7 to achieve the correct system call to exit the program. Then ‘swi 0’ is used to exit successfully.