CSE2312-002, Fall 2014, Programming Assignment 3

Due Date via Blackboard: November 25, 2013 at 11:59pm Central

In the third programming assignment, we will create a 4 function calculator. The inputs to your program will come from the command line (stdin). The first input will either be a unary operation or a number. If the first input is a unary operation (negate), then there should only be one more input, which should be a number. If the next input is not a number, then `handleUndefined` should be called. If the first input is a number, then the next input must be a binary operation (+, -, *, /). If the next input is not a binary operation then `handleUndefined` should be called. Finally, there should only be a third input if the first input was a number and the second was a binary operation. Thus, the third input should also be a number. If the third input is not a number then `handleUndefined` should be called.

The encoding for the operations are as follows:

- `ADD: +: result = input_1 + input_3`
- `SUB: -: result = input_1 - input_3`
- `MUL: *: result = input_1 * input_3`
- `DIV: /: result = input_1 / input_3`
- `NEG: -: result = -input_2`

Based on the operation input, your program should output the result with the corresponding computation. For example, if the first input is 4, the second is +, and the third is 5, then 9 should be output to the console, since 9 = 4 + 5. After the result is output, your program should continue from the beginning by asking for the next inputs. This should continue until the user inputs a 'q' at which point your program should exit.

A partial solution is provided which takes of the major control flow. You will implement several functions with specific requirements. If implemented correctly, the calculator will function as described above. There is a label in the provided partial solution for each function you should implement. All functions will be called with bl, so you should end your functions by calling bx lr where lr is the value in the link register when the function was called. The names and requirements for the functions you will implement are below. Some of the functions will take the address of the first byte of a string as input. Note that these strings are null terminated.

`isNegate`: This function takes as input r0. R0 has the address of the first byte of a string. The function should return 1 if the string is '-' (negate symbol) and 0 if not. It should move the return value into r0.

`isOperation`: This function takes as input r0. R0 has the address of the first byte of a string. It should return 1 if the string is a value operation (+, -, x, /) and 0 if not. It should move the return value into r0.

`isNumber`: This function takes as input r0. R0 has the address of the first byte of a string. It should return 1 if the string is a valid integer (including negative integers) and 0 if not. It should move the return value into r0.

For the functions `executeAdd`, `executeSubtract`, and `executeMultiply` assume the first operand is in r0 and the second is in r1. The proper operation should be performed and the result returned in r0. You should check for overflow in all these functions, and branch to `handleOverflow` if it occurs.

You ***must*** write procedures / functions (e.g., called via `bl executeAdd` and returned via `bx lr`). Since you will be reusing your solution for programming assignments, having this additional structure is going to make your life much easier later on when we complicate the situation with floating point and input/output. You ***must*** use this naming scheme for the procedures.

The following pseudo-code clarifies the problem.

```
int input_1, int input_2, int input_3, int result;

void main() {
	for(;;)
	{
		input_1 = get_input_1();
		if( input_1 == 'q' )
		{
			quit();
		}
		else if( input_1 == '-' )
		{
			input_2 = toInt(get__input2());
			result = executeNegate(input_2);
		}
		else if( isInt(input_1) )
		{
			input_1 = toInt(input_1);
			input_2 = get_input_2();
			if( input_2 == '+' )
			{
				input_3 = get_input_3();
				if( !isInt(input_3) )
				{
					handleUndefined();
				}
				input_3 = toInt(input_3);
				result = executeAdd(input_1, input_3);
			}
			else if( input_2 == '-' )
			{
				input_3 = get_input_3();
				if( !isInt(input_3) )
				{
					handleUndefined();
				}
				input_3 = toInt(input_3);
				result = executeSubtract(input_1, input_3);
```

```
                }
                else if( input_2 == '*' )
                {
                        input_3 = get_input_3();
                        if( !isInt(input_3) )
                        {
                                handleUndefined();
                        }
                        input_3 = toInt(input_3);
                        result = executeMultiply(input_1, input_3);
                }
                else if( input_2 == '/' )
                {
                        input_3 = get_input_3();
                        if( !isInt(input_3) )
                        {
                                handleUndefined();
                        }
                        input_3 = toInt(input_3);
                        result = executeDivide(input_1, input_3);
                }
                else
                {
                        handleUndefined();
                }
        }
        else
        {
                handleUndefined();
        }
        printResult(result);
    }
}
```

Undefined Operations, Overflow Handling, and Special Instructions for Multiply and Divide

In the event that an operation is not defined, your program should execute a procedure called `handleUndefined`, which should output the string: "Operation Undefined." Your program should handle overflow and division by zero, by calling functions `handleOverflow` and `handleDivByZero`, which respectively output the strings: "Error Overflow" and "Error Divide by Zero."

We have provided a division operation you should use, but you must perform the error handling.

Directory Structure, Given Files, and Submission

As in the other programming assignments, we have provided files to help you begin in the archive pa03.zip. Included in these files are routines for converting an integer to a string and for converting a string into an integer. In addition, we have provided code for getting input from and writing output to the console. Write your solution in the file pa03/pa03.s. You will see we provide some example C functions in helper.c that are linked together with the

assembly file. This illustrates how to combine C and assembly, as well as how to call C functions from assembly.

Example Interaction and Output Files

In the subdirectory ‘tests’ you will find some example tests we have used on a correct solution. In the subdirectory ‘screenshots’ you will find some example interactions using the calculator, and is how your solution should work. Be sure you use EXACTLY the same string messages (all are provided in the given files). Be sure you use EXACTLY the same spacing, etc. Our grading scripts have some capability to detect minor spacing mismatches, but you might confuse it and lose points if you are too far off.