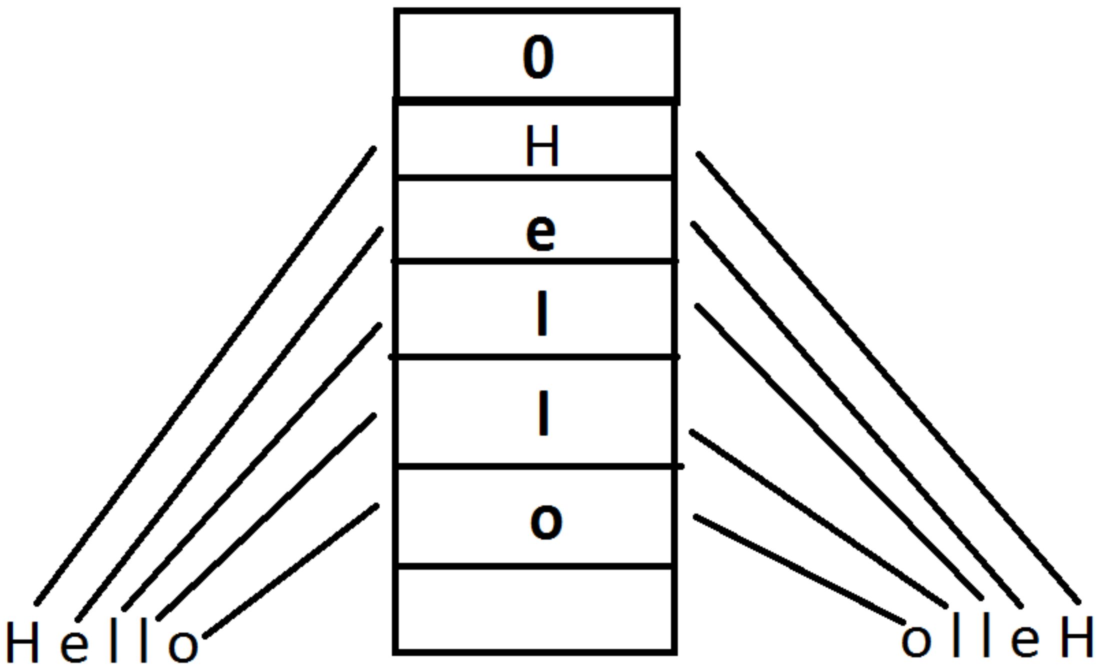
Scott Hofman s0943941

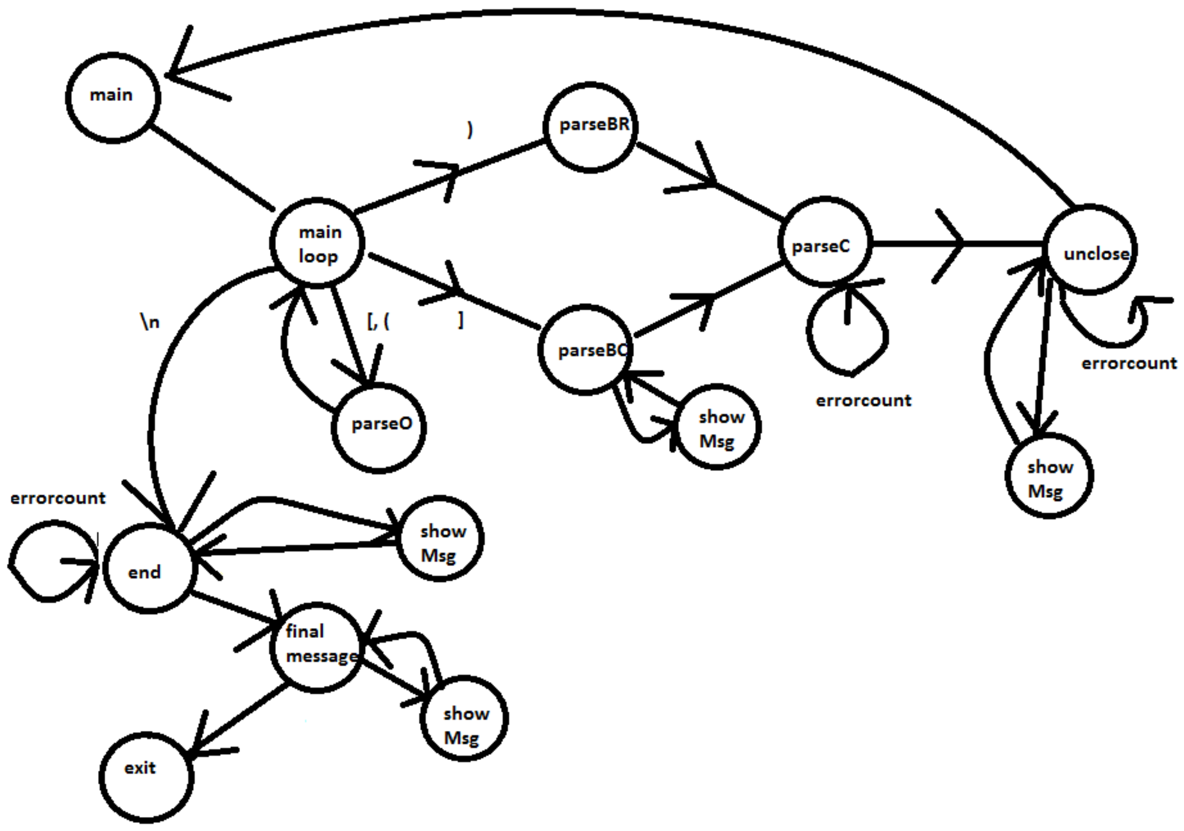
Informatics 2CS – Coursework Task 1

1) For the first, after receiving the code from the user, you implement a stack created by your own variables (since we weren't allowed to use the stack pointer). The program first places a zero onto the stack, so as to know that you have reached the end of the stack when you are breaking the function down. You then push each individual char onto the stack, moving the pointer up for each character. During this push loop, you check to see if all the characters have been parsed (by seeing if the array is zero), and if it has, then the string has successfully been passed onto the stack. Next, you reset the counters, and begin popping the characters off the stack, in the reverse order as you pushed them on. You check to see if the character is zero (or if the string is at an end), and, if not, you pop it back onto the character array. Upon reaching zero, the program realizes it has reached the end of the string, and thus goes to the end, where it prints the array, followed by a newline, before exiting.
Included in this explanation is a diagram of the process described above.
Of the registers chosen, most hold little significance. Using $v0, $a0 for syscall is simply a convention I have chosen to follow. The array is held in $s0 because I want it to be saved over multiple procedural calls, whereas the t1 and t0 (used as placeholders to pass things around, increment values, or check arguments) aren't needed over these calls.



1) This program creates a stack, which then parses each character by moving up and down the stack. For every character, the system will check it against an open bracket '[', an open round '(', a closed bracket ']' or a closed round ')'. If it is not any of these, it will check to see if its finished by comparing the current char to a newline. If not any of these, it continues to loop until it finds one of these things. If open, the program makes a note out of it, increments the stack pointer by 1 and then moves onto the next character. If closed, the program checks the stack and then goes into a bunch of error type matching. If closed accidently, it goes to the function unclose, which shows the corresponding error message. If the brackets are matching, the function is popped, and the stack is shifted. If they are not matching, then it shows an error message. The program keeps track of how many errors in

$s4, incrementing it by 1 everytime. This continues looping, until we reach the newline, at which point the program towards the end. This will display the amount of errors/the success, before exiting.
Included in this explination is a flowchart of how the program works.
Of the registers chosen, I again have followed the standard conventions of MIPS. $v0 and $a0 are used as part of the syscall function.$s0 is used to store the brackets, $s1 is where the current position is, $s2 is the stack top, $s3 is the stack bottom, $s4 is the number of errors, $s5 is the input and $s6 checks the close. These need to be saved over multiple procedural calls, so that is why s is used.


main
main loop
)
parseBR
]
parseBC
[, (
parseO
\n
parseC
unclose
errorcount
errorcount
show Msg
show Msg
errorcount
end
show Msg
final message
show Msg
exit