MIPS Assignment 2

ECE 4612

Leo Battalora

2021/10/05

Table of Contents

Objectives

This lab asked students to create an arithmetic syntax checker REPL (Read Evaluate Print Loop) to practice the fundamentals of programming using the MIPS32 Instruction Set. This program which checks the validity of user-inputted numerical expressions will use MATLAB conventions for its syntax rule. The program displays the prompt ">>> " to allow user input of expressions up to 64 symbols in length. Expressions should include only "()0123456789+-*/ ". If there is an error in an expression, "Invalid input" will be written to the I/O window; if there is no error, "Valid input" will be written instead. The program will loop so that users can enter new expressions without re-running the program.

Skills practiced include: moving data between registers, reading and writing to and from memory, issuing system calls, branching, writing complex conditionals, and calling functions and subroutines.

Tools/Equipment

Mips Assembler and Runtime Simulator (MARS) version 4.5 was used to write, assemble, execute, and debug MIPS32 based programs. Developed by Pete Sanderson and Kenneth Vollmar, MARS is a compiled .jar which was executed using JDK 11, the open-source reference implementation of version 11 of the Java SE Platform.

Analysis/Algorithm/Procedure

This heading is sectioned into a list of tasks. These tasks are executed in order for each iteration of the REPL. Each subheading will detail any specific logic or algorithms used.

Task 1: Reading input

First the user is prompted for an input. The read_string system call is set to read 65 characters, one more than the assignment specification. This is because the read_string system call will include the newline character in its read buffer.

After reading the input and storing it into memory, the newline character at the end of the read string is replaced with a null-terminator so that the remainder of the syntax checking does not need to account for newline characters.

Task 2: Check for empty input

As MATLAB will not produce any errors when no input is given, an empty input is considered a valid input. The function which removes the newline character from the input also returns the string length on return. This value is used to check if the input is empty.

Task 3: Check for disallowed symbols

In order to have simpler syntax checking in the following tasks, a simple loop will check each character in the input string against a list of allowed characters: `"0123456789+-*/() "`.

```
# if (*str == one from "0123456789+-*/() ") loop again
beq $t0, 43, check_symbols_loop0    # '+'
beq $t0, 45, check_symbols_loop0    # '-'
beq $t0, 42, check_symbols_loop0    # '*'
beq $t0, 47, check_symbols_loop0    # '/'
beq $t0, 40, check_symbols_loop0    # '('
beq $t0, 41, check_symbols_loop0    # ')'
beq $t0, 32, check_symbols_loop0    # ' '
blt $t0, 48, check_symbols_found # less than '0'
bgt $t0, 57, check_symbols_found # greater than '9'
b check_symbols_loop0
```

The algorithm for this task is a very straightforward loop which iterates over each character in the input string, checking it against the list of allowed symbols above. If the symbol matches any of the allowed symbols, it loops again, if the symbol falls outside the allowed ASCII values, it short-circuits to calling a subroutine which returns to the main function with a status code indicating the error.

Task 4: Check for space-separated numbers

The only case where spaces influence the meaning of an input is when they separate numbers. A function `check_ssn()` will look for characters `"0123456789"` and throw an error if a numeral is directly followed by space(s) then another numeral.

```
int check_ssn(char *str)
{
  // loop over each character
  while (*str != '\0')
  {
    // only for numerals followed by a space...
    if (*str >= '0' && *str <= '9' && *(str + 1) == ' ')
    {
      str++;

      // skip over consecutive spaces
      while (*str == ' ')
```

```
      {
        str++;
      }

      // if the first non-space character is another numeral, then a
space-separated-number has been found.
      if (*str >= '0' && *str <= '9')
      {
        return 1;
      }
    }
    str++;
  }

  return 0;
}
```

The C function above shows how to iterate over a string, checking for space-separated-numbers. Essentially, once a number followed by at least one space is found, it loops until the next non-space character is found, and checks if it is also a numeral. Once a single space-separated number is found, it returns a status code of 1.

After verifying that there are no space-separated number violations, all remaining spaces in the string are removed for simpler processing in the next steps.

```
void remove_spaces(char *str)
{
  char *scan = str;

  // loop over each character
  while (*scan != '\0')
  {
    // skip scanner pointer over spaces
    while (*scan == ' ')
    {
      scan++;
    }

    // copy each char from the scanner head
    *str = *scan;

    // advance scanner and result heads
    scan++;
    str++;
  }

  // null-terminate the result
  *str = '\0';
}
```

The C function above outlines the logic to remove all spaces in a null-terminated char array with minimal backtracking. It uses a read head which skips over white spaces and a write head which overwrites the current string only with non-space characters found by the read head.

Task 5: Check for unbalanced parentheses

In task 5, a simple algorithm is implemented to make sure all parentheses are balanced. Open parentheses must precede closing parenthesis, and the number of opening and closing parenthesis must match.

The algorithm uses a counter to monitor the parenthesis balance as it iterates over the input string:

1. Initialize counter to 0.
2. Loop until a parenthesis is found. If it is an open parentheses, increment counter. Else, decrement counter.
3. Check if the counter is negative. A negative counter would indicate a close parenthesis appeared before a corresponding opening parenthesis. In this case the function would immediately return an error status code.
4. Repeat steps 2 and 3 until the end of the string is reached.
5. Check if the counter is equal to zero. If not, there were open parentheses which were never closed. Return a corresponding status code.

Task 6: Check remaining syntax rules

Task 6 catches the remaining syntax rules centered around operator and operand rules. The expression must begin and end with operands, operands may not be adjacent to one another without being separated by an operator, and every operand must be separated by only a single operator.

This task implements a set of conditionals that must be true for each character encountered.

1. First the expression much begin with an operand, so * and / are illegal first-characters. If they are found, return an error.
2. Then, for each encountered character applying the following conditionals, iterating until the end of the string is reached with no errors:

```
if str[i] is one of "+-(" then must be followed by "+-0123456789(" but not "*/)\0"

if str[i] is one of "*/" then must be followed by "01234567890(" but not "+-*/)\0"

  if str[i] is one of ")" then can be followed by "+-*/)\0" but not "0123456789("

if str[i] is one of "0123456789" then can only be followed by "+-*/0123456789)\0" but
                                    not "("
```

Task 7: Print result and loop

If any of the syntax-checking functions and subroutines above ever detect a syntax error, they immediately branch to the "Invalid input" output subroutine, then repeat the REPL over again.

If all the syntax-checking functions are passed without throwing any syntax errors, then the "Valid input" output subroutine will trigger.

Results/Observations

Because the syntax checker was implemented as 7 simple tasks abstracted into functions, each task was individually developed and iteratively tested.

After the entire procedure was implemented into the REPL syntax-checker, an array of tests were passed through the program and evaluated on the accuracy of the results.

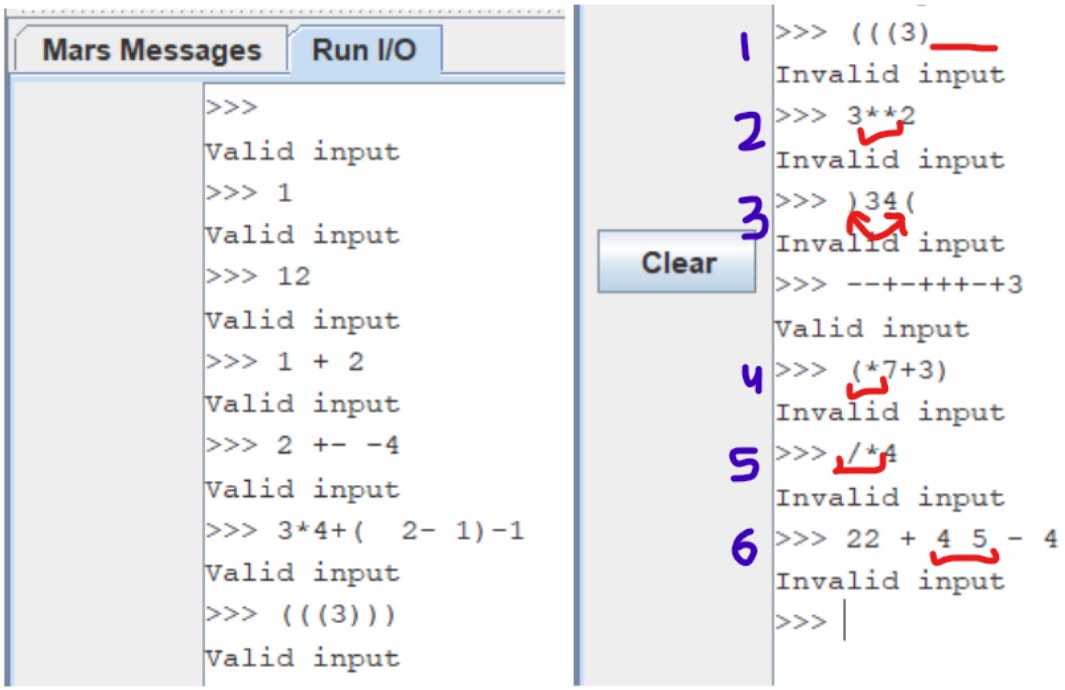

Figure 1: Syntax tests

In the figure above, six examples of invalid syntax are annotated in red.

1. This is an example of unbalanced parentheses.
2. This is an example of two adjacent operators which are not separated by an operand.
3. This is an example of unbalanced parenthesis.
4. This is an example of a sub-expression (expression within parentheses) beginning with an operator.

5. This is an example of an expression beginning with an operator and two adjacent operators.
6. This is an example of space-separated-numbers.

All syntax tests passed as expected.

Conclusions

This assignment served as a powerful exercise in learning how to think like a CPU. In developing some of the functions used in the program, they were first conceptualized in C, then translated into the MIPS32 instruction set. This process required me to be mindful of designing simple conditionals, minimizing backtracking in loops, and maintaining short lists of instance variables.

The program developed in this assignment can be used as the syntax checker before passing inputted expressions to an evaluator which could calculate the value of expressions. Because the output can be confirmed to be syntactically correct, the evaluation of expression can be performed without consideration for invalid syntaxes.