Lab 5: Table Lookups and Recursion

CS/CoE 0447(B) Spring 2015 — Dr Kosiyatrakul (Tan)

Released: Friday 20 February 2015, 06:00 EST
Due: Thursday 26 February 2015, 23:59 EST

Submission timestamps will be checked and enforced strictly by CourseWeb; **late submissions will not be accepted.** Remember that, per the course syllabus, if you are not marked by your recitation instructor as having attended a recitation, your score will be cut in half. **Follow all instructions.**

For this lab, you will continue practicing functions by implementing a simple data structure, and you will also get your first taste of implementing recursion. **As always, all functions MUST follow the proper MIPS calling conventions discussed in class and detailed in Lab 4.** In fact, by the end of this assignment, you will hopefully have gained a fuller appreciation for just why adherence to calling conventions is so helpful to us as budding assembly writers, especially when we encounter recursion.

A. Implementing table lookups

In much of programming, one very useful data structure is a "dictionary" or "associative array". A dictionary is a table-like data structure which, generally speaking, maps input "keys" to output "values". One way to implement a rudimentary dictionary or table is simply by defining two arrays in parallel:

```
.data
   names:  .asciiz  "steve", "john", "chelsea", "julia", "ryan"
   ages:   .byte    20, 25, 22, 21, 23
```

This dictionary lists the names of five people in the `names` array and their corresponding ages in the `ages` array. Conceptually, we can think of the data defined above as having a **logical** form of one-to-one correspondence, like this:

Name:	steve	john	chelsea	julia	ryan
Age:	20	25	22	21	23

…even though the data is **physically** laid out in a much more linear fashion, like this:

s	t	e	v	e	/	j	o	h	n	/	c	h	e	l	s	e	a	/	j	u	l	i	a	/	r	y	a	n	/	20	25	22	21	23

Your task is to write a program that takes a user-inputted name and prints out the age of the person, if there is an entry for them in the table. Here are two sample outputs:

```
Please enter a name: ryan
Age is: 23
```

```
Please enter a name: jim
Not found!
```

To give you a head-start on your implementation, we will outline a basic algorithm for you by specifying two **required** functions: `_StrEqual` and `_LookUpAge`.

`_StrEqual` should take, as arguments, the addresses of two strings, and return 1 if those two strings match or 0 otherwise. Your main program will want to know when two strings match, so it will use this function to help it step through the table, checking the string the user entered against each entry in the `names` array.

As your main program checks for a name match, using `_StrEqual`, it should be keeping track of the current index in the `names` array. (For example, "`steve`" is at index 0, "`chelsea`" is at index 2, etc.) If and when you get a match, `_StrEqual` will return 1, and you should record the index where the match took place. Then you should find out the corresponding age from that index of the `ages` array using the second required function, `_LookUpAge`.

`_LookUpAge` should take the address of a byte array and an index, and return the value from that array at that index. For example, given the address of the `ages` array and the index 3, `_LookUpAge` should return 21, since that corresponds to Julia's age which given in position 3.

If your main program reaches the end of the `names` array without matching, then the input name does not match any name in the table. In that case your program should print "`Not found!`" and exit.

You may assume that there are always five names in the array; however, your code may NOT assume that they will be the same names as shown in the example above. You should submit your program with the example data given above.

Here is one possible pseudo-code outline of the main program's flow:

```
input = (user input)
index = 0

while index < 5 {
    match = _StrEqual(input, names[index])          // compare two strings
    if match == 1 {                                 // a match was found
        age = _LookUpAge(ages, index)               // look up corresponding age
        print "Age is: ", age                       // print result
        exit
    }
    else {                                          // no match found
        index = index + 1                           // try the next one
    }
} // end while

print "Not found!"          // if we ran out, give up
exit
```

▶ **Save your program as `lab05part1.asm`. Please match the output format exactly.**

B. Simple recursion: Factorial

Consider the factorial function, which is defined for any nonnegative integer n as follows[1]:

$$\text{Fac}(n) = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

n	1	2	3	4	5	6	7	…
Fac(n)	1	2	6	24	120	720	5040	…

Since each successive multiplication can simply be expressed as the previous product multiplied by the new integer, we can also write this mathematical function with a recursive definition:

$$\text{Fac}(n) = \begin{cases} \text{Fac}(n-1) \times n & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$

This recursion can then be expressed as the following pseudo-code:

```
int _Fac(int n) {
    if n <= 0 {
        return 1            // This base case stops the recursion
    }
    else {
        return _Fac (n–1) * n
    }
}
```

Your task is to write a simple factorial program in MIPS assembly based on the pseudo-code above. Your code **MUST** call itself recursively, as the grader will explicitly check for `jal` instructions at the proper places inside the function.

Remember that **your `_Fac` function MUST adhere to the proper calling conventions** for receiving arguments and placing the return values in the proper registers. Since a recursive function is non-leaf by definition, your function **MUST** save and restore any $s registers that it uses as well the $ra register and stack pointer using a prologue and epilogue. The final value Fac(n) for the integer provided must be printed by the main program.

Here are two sample outputs:

```
Enter a nonnegative integer: 5
5! = 120
```

```
Enter a nonnegative integer: -2
Invalid integer; try again.
Enter a nonnegative integer: 4
4! = 24
```

- **Save your program as `lab05part2.asm`. Please match the output format exactly.**
- **Zip your two programs for this assignment into a single zip file called `lab05.zip` and submit it via CourseWeb.**

[1] Fac(0) = 1, by convention, representing an "empty product" of no integers. Math is weird. But also cool.