Programming 2 (SS 2023)
Saarland University
Faculty MI
Compiler Design Lab

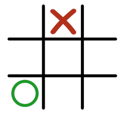
Sample Solution 3
MIPS and C

Prof. Dr. Sebastian Hack
Pascal Lauer, M. Sc.
Marcel Ullrich, B. Sc.

The calendar indicates which script chapters you should study in conjuction with each lecture. The exercises are designed to enhance your understanding of the lecture material and prepare you for the mini-tests and the final exam. Additional exercises can be found at the end of each chapter in the script.
The difficulty of an exercise on the sheet is determined by the number of annotated 'X' and 'O' marks in the tic-tac-toe field, with four levels (1-4) increasing by one mark per level.

1 MIPS

Exercise 3.1:



Write two assembler programs which both compute *powers of three*. The n-th power of three is defined by:

$$3^n = \begin{cases} 1 & \text{if } n = 0 \\ 3 \cdot 3^{(n-1)} & \text{if } n > 0 \end{cases}$$

Example:

$$3^4 = 3 \cdot 3^3 = ... = 3 \cdot (3 \cdot (3 \cdot (3 \cdot 1))) = 81$$

The first program `pow3_rec` shall compute the n-th power of three recursively as in the recursive definition above. To this end, the program calls itself. The second variant `pow3_iter` must compute the n-th power of three without calling other subroutines by using a loop. Consider the additional precautions (setting up the call frame, saving registers, etc.) which the recursive definition requires. The parameter n is passed via register `$a0`, the result must be passed back to the caller via register `$v0`.
Hint: Remember the example program from the lecture which calculates the factorial.

Solution

Recursive version:

```
.text
    .globl pow3_rec

pow3_rec:
    addi $sp, $sp, -8           # allocate some space in the stack
    sw $ra, 4($sp)              # store the address where we need to jump back
    sw $a0, 0($sp)              # store n

    beq $a0, $zero, base_case   # base case if $a0 is zero
    addi $a0, $a0, -1           # we call pow3_(n-1)
    jal pow3_rec

    mul $v0, $v0, 3             # multiply current result by 3

    lw $a0, 0($sp)              # load n stored before
    lw $ra, 4($sp)              # load return address stored before
    addi $sp, $sp, 8            # reset stack pointer
    jr $ra                      # return to caller

base_case:                      # (could be avoided and code shortened)
    li $v0, 1                   # base case
    lw $a0, 0($sp)              # load n stored before
    lw $ra, 4($sp)              # load return address stored before
    addi $sp, $sp, 8            # reset stack pointer
    jr $ra                      # return to caller
```

Iterative version:

```
.text
    .globl pow3_iter

pow3_iter:
    li $v0, 1                # start at 1
    li $t0, 0                # initialize loop counter

loop:
    beq $t0, $a0, end        # end if loop counter $t0 equals $a0
    mul $v0, $v0, 3          # multiply by 3
    addi $t0, $t0, 1         # increase loop counter
    j loop                   # loop

end:
    jr $ra                   # return to return address
```

Exercise 3.2:

Konrad Klug has been assigned the task to carry out a complex mathematical calculation. Since he will often need the first *n* Fibonacci numbers for this task, he wants to store as many of the as possible in an array so that he does not need to recompute them from scratch each time. Unfortunately, he does not know how to implement this yet. The Fibonacci numbers are calculated as follows:

$$
\begin{aligned}
F_0 &:= 0 \\
F_1 &:= 1 \\
F_n &:= F_{n-1} + F_{n-2} \qquad\qquad \text{for } n \in \mathbb{N} \land n \geq 2
\end{aligned}
$$

Help Konrad Klug by implementing the storage of the Fibonacci numbers in a *word* array. You are given the base address of the array in register `$a0` and the address one word behind the last element of the array in register `$a1`.

Solution

```
.text
    .globl fib
    # a0 Base address of the array
    # a1 One past-the-end address

fib:
    # replace first element
    li $t0 0
    sw $t0 0($a0)
    # replace second element
    li $t0 1
    sw $t0 4($a0)
    # increase address
    addiu $a0 $a0 8

loop:
    # loop condition
    bge $a0 $a1 end
    # load previous elements
    lw $t0 -4($a0)
    lw $t1 -8($a0)
    # calculate next element
    addu $t0 $t0 $t1
```

```
    # store new element
    sw $t0 0($a0)
    # increase address
    addiu $a0 $a0 4
    j loop

end:
    jr $ra
```

Exercise 3.3:

Look at the following code implementing insertion sort.

1. Fill in the gaps such that `swap` swaps two elements in an array. The addresses of the elements to be swapped are passed via the registers `$a0` and `$a1`.

```
.text
swap:
  # $a0, $a1 adresses of the elements to be swapped
  lw ____ ____
  lw ____ ____
  sw ____ ____
  sw ____ ____
  jr $ra
```

2. Now fill in the gaps sucht that `insertion_sort` follows the calling convention.

```
insertion_sort:
  # $a0 base adress of the array
  # $a1 last adress of the array
  beq $a0 $a1 end
  move $t0 $a0

  loop:
    addiu $t0 $t0 4
    bgt $t0 $a1 end
    lw $t5 ($t0)
    move $t1 $t0

  loop_2:
    beq $t1 $a0 loop

    addiu $t2 $t1 -4
    lw $t6 ($t2)
    ble $t6 $t5 loop

    addiu $sp $sp -32
    sw ____ ____
    sw ____ ____
    sw ____ ____
    sw ____ ____
    sw ____ ____
    sw ____ ____
    sw ____ ____
    sw ____ ____

    move $a0 $t1
    move $a1 $t2

```

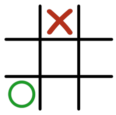
```
    jal swap

    lw ____ ____
    lw ____ ____
    lw ____ ____
    lw ____ ____
    lw ____ ____
    lw ____ ____
    lw ____ ____
    lw ____ ____
    addiu $sp $sp 32

    addiu $t1 $t1 -4
    b loop_2

  end:
    jr $ra
```

Solution

```
# a)
.text
swap:
  # $a0, $a1 adresses of elements to be swapped
  lw $t0 ($a0)
  lw $t1 ($a1)
  sw $t0 ($a1)
  sw $t1 ($a0)
  jr $ra

# b)
insertion_sort:
  # $a0 base adress of the array
  # $a1 last adress of the array
  beq $a0 $a1 end
  move $t0 $a0

  loop:
    addiu $t0 $t0 4
    bgt $t0 $a1 end
    lw $t5 ($t0)
    move $t1 $t0

  loop_2:
    beq $t1 $a0 loop

    addiu $t2 $t1 -4
    lw $t6 ($t2)
    ble $t6 $t5 loop

    addiu $sp $sp -32
    sw $a0 0($sp)
    sw $a1 4($sp)
    sw $t0 8($sp)
    sw $t1 12($sp)
    sw $t2 16($sp)
    sw $t5 20($sp)
    sw $t6 24($sp)
    sw $ra 28($sp)
```

```
    move $a0 $t1
    move $a1 $t2

    jal swap

    lw $a0 0($sp)
    lw $a1 4($sp)
    lw $t0 8($sp)
    lw $t1 12($sp)
    lw $t2 16($sp)
    lw $t5 20($sp)
    lw $t6 24($sp)
    lw $ra 28($sp)
    addiu $sp $sp 32

    addiu $t1 $t1 -4
    b loop_2

  end:
    jr $ra
```

Exercise 3.4: Nested Function calls



Take a look at the following MIPS program.

```
    .text
    .globl confusion
confusion:

    ...

    and     $t0 $t0 $zero
    or      $t0 $t0 $a0
    add     $t1 $a1 $t0
    or      $s1 $t0 $t1

    move    $s0 $t0

    ...

    jal     mystery

    ...


    add     $t0 $t1 $v0
    add     $v0 $a0 $t0

    ...

    jr      $ra
```

Figure 1: A program disregarding calling conventions.

Complete the program following the calling conventions. Be memory efficient, thus only store necessary registers.

Solution

```
    .text
    .globl confusion
confusion:
    # prologue
    add      $sp $sp -8
    sw       $s0 ($sp)
    sw       $s1 4($sp)

    and      $t0 $t0 $zero
    or       $t0 $t0 $a0
    add      $t1 $a1 $t0
    or       $s1 $t0 $t1

    move     $s0 $t0

    # store the registers
    add      $sp $sp -12
    sw       $t1 ($sp)
    sw       $a0 4($sp)
    sw       $ra 8($sp)

    jal      mystery

    # load back the registers
    lw       $t1 ($sp)
    lw       $a0 4($sp)
    lw       $ra 8($sp)
    add      $sp $sp 12


    add      $t0 $t1 $v0
    add      $v0 $a0 $t0

    # epilogue
    lw       $s0 ($sp)
    lw       $s1 4($sp)
    add      $sp $sp 8

    jr       $ra
```

Exercise 3.5:

In this exercise you must find the errors in a MIPS program (see *buggy.zip*). Find the errors and correct them. There are two errors per exercise. First, try to solve the exercise without using MARS.

1. In `buggyProgram1.asm`, all numbers in an array are supposed to be added and the results should be printed to the console.
2. A fellow student asks for your help. He tried to write down the faculty program from the lecture by himself, but for some reason it does not work. His attempt can be found in `buggyProgram2a.asm` and `buggyProgram2b.asm`.

Remark: To test these programs in MARS, copy them into the same directory and select the options *Assemble all files in directory* and *Initialize Program Counter to global 'main' if defined* in the settings of MARS.

Solution

1. The end address of the array is incorrect. A word is four bytes, to access the last element of the array we need to point to the address behind the fourth word, i.e. address of the first word plus $4 * 4 = 16$.

   ```
   addiu $a2 $a1 4      # wrong
   addiu $a2 $a1 16     # correct
   ```

   The data is stored as words, but is loaded as bytes. To load words, one must use `lw` instead of `lb`.

   ```
   lb $t0 0($a1)        # wrong
   lw $t0 0($a1)        # correct
   ```

2. The subroutine `fac` did not make its definition visible to external programs, therefore it cannot be used by `main`.

   ```
   .globl fac           # insert below .text
   ```

   The loop counts from n to 0, which causes the result to be always zero.

   ```
   bgez $a0 loop       # wrong
   bgtz $a0 loop       # correct
   ```

2 Introduction to C

Exercise 3.6:

1. Write a C program which declares two variables of type `int`, assigns arbitrary values to these variables, and then prints both variable values to the console. To this end, create a file `main.c` and implement the `main` function.

   Create an executable file `prog` and execute the program.

2. Extend the file `main.c` from part a) with a function `max` which receives two `int` arguments and returns the maximum of both values.

3. Extract the function written in b) into its own translation unit `max.c`. Create a header file `max.h` to make the signature of the function available to `main`.

   Create an executable `prog` and execute the program.

Solution

1. Program:

   ```c
   #include <stdio.h>

   int main() {
       int a = 10;
       int b = 20;

       printf("a: %d\n", a);
       printf("b: %d\n", b);

       return 0;
   }
   ```

   Create an executable and run it:

```
$ cc -o main.o -c main.c
$ cc -o prog main.o
$ ./prog
```

In short:

```
$ cc main.c -o prog
$ ./prog
```

2. Function max:

```c
#include <stdio.h>

int max(int x, int y) {
    if(x > y) {
        return x;
    } else {
        return y;
    }
}

int main() {
    int a = 10;
    int b = 20;

    int res = max(a, b);

    printf("Maximum of a and b: %d\n", res);

    return 0;
}
```

Create executable: Same as above.

3. With a header file:

max.h

```c
#ifndef MAX_H
#define MAX_H

int max(int x, int y);

#endif
```

max.c

```c
#include "max.h"

int max(int x, int y) {
    if(x > y) {
        return x;
    } else {
        return y;
    }
}
```

main.c

```c
#include <stdio.h>
#include "max.h"

int main() {
    int a = 10;
```

```
    int b = 20;

    int res = max(a, b);

    printf("Maximum␣of␣a␣and␣b:␣%d\n", res);

    return 0;
}
```

Create executable and run it:

```
$ cc -o main.o -c main.c
$ cc -o max.o -c max.c
$ cc -o prog main.o max.o
$ ./prog
```

In short:

```
$ cc main.c max.c -o prog
$ ./prog
```

Exercise 3.7:

Analyze and understand the following C program:

```
#include <stdio.h>

int main() {
    for (int i = 1; i < 10; i++) {
        if ((i % 2) == 0) {
            printf("The␣number␣%i␣is␣...\n", i);
        } else {
            printf("The␣number␣%i␣is␣...\n", i);
        }
    }
    return 0;
}
```

1. What is the program output? How can you reasonably replace the ellipses (...) in the `printf` statements?
2. Modify the program such that it prints the numbers in range 0 to 20 (inclusive).
3. Can you modify the loop in such a way that the numbers from subtask (2.) are printed in reverse order?

Solution

1. The first ellipsis should be replaced with “even” and the second ellipsis with “odd”. The program then prints the following:

```
The number 1 is odd.
The number 2 is even.
The number 3 is odd.
The number 4 is even.
The number 5 is odd.
The number 6 is even.
The number 7 is odd.
The number 8 is even.
The number 9 is odd.
```

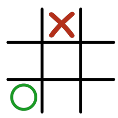
2. Modified program:

```c
#include <stdio.h>

int main() {
    for (int i = 0; i <= 20; i++) {
        if ((i % 2) == 0) {
            printf("The number %i is even\n", i);
        } else {
            printf("The number %i is odd\n", i);
        }
    }
    return 0;
}
```

3. With print order reversed:

```c
#include <stdio.h>

int main() {
    for (int i = 20; i >= 0; i--) {
        if ((i % 2) == 0) {
            printf("The number %i is even\n", i);
        } else {
            printf("The number %i is odd\n", i);
        }
    }
    return 0;
}
```

Exercise 3.8:



Konrad Klug wants to compute some integer calculations, but he lost his calculator.
He has already found some code that reads the inputs from the user and outputs the result, but he doesn't know how to implement the calculation.
Help him, by implementing the `calculate` function. The function receives two integers (`x` and `y`) and a character `op` representing the operation.
Your calculator should support addition (+), substraction (-), multiplication (∗) and division (/).

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int calculate(int x, char op, int y) {
    // Add your code here
    return 0;
}

int main(int argc, char* argv[]) {
    int x = 0;
    int y = 0;
    char op = 0;
    char* s = calloc(64, sizeof(char));

    printf("Please enter the expression to be calculated: ");
    scanf("%d %c %d", &x, &op, &y);

    sprintf (s,"%d %c %d = ", x, op, y );
```

```
    int slen = strlen(s);
    s += slen;

    if (!((op == '+') || (op == '-') || (op == '*') || (op == '/'))) {
        sprintf(s - slen, "Invalid input!");
    } else {
        sprintf (s, "%d", calculate(x, op, y));
    }

    s -= slen;
    printf( "%s \n", s);

    free(s);
    return 0;
}
```

Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int calculate(int x, char op, int y) {
    switch(op){
        case '+':
            return (x + y);
        case '-':
            return (x - y);
        case '*':
            return (x * y);
        case '/':
            return (x / y);
        default:
            return -1;
    }
}

int main(int argc, char* argv[]) {
    int x = 0;
    int y = 0;
    char op = 0;
    char* s = calloc(64, sizeof(char));

    printf("Please enter the expression to be calculated: ");
    scanf("%d %c %d", &x, &op, &y);

    sprintf (s,"%d %c %d = ", x, op, y );
    int slen = strlen(s);
    s += slen;

    if (!((op == '+') || (op == '-') || (op == '*') || (op == '/'))) {
        sprintf(s - slen, "Invalid input!");
    } else {
        sprintf (s, "%d", calculate(x, op, y));
    }

    s -= slen;
    printf( "%s \n", s);

    free(s);
```

```c
    return 0;
}
```

Exercise 3.9:

This exercise is all about dealing with a variety of pesky error messages one might encounter while programming in C.

1. Analyse the following error messages and describe the issue or what could have triggered them.

   (a)
   ```
   main.c:9:7: error: assignment of read-only variable ’a’
   ```

   (b)
   ```
   main.c:12:5: error: too few arguments to function ’g’
   ```

   (c)
   ```
   main.c:16:5: error: called object ’b’ is not a function
       or function pointer
   ```

   (d)
   ```
   main.c:19:5: warning: implicit declaration of function ’h’
       [-Wimplicit-function-declaration]
   /usr/bin/ld: /tmp/ccsVArgH.o: in function ’main’:
   main.c:(.text+0x3c): undefined reference to ’h’
   collect2: error: ld returned 1 exit status
   ```

   (e)
   ```
   main.c:23:8: error: lvalue required as left operand of assignment
   ```

   (f)
   ```
   main.c:26:6: error: #error "The cake is a lie!"
   ```

   (g)
   ```
   /usr/bin/ld: /tmp/cctUcBj6.o: in function ’foo’:
   foo.c:(.text+0x0): multiple definition of ’foo’;
       /tmp/cc01IbiT.o:main.c:(.text+0x0): first defined here
   /usr/bin/ld: /tmp/cckGDjLC.o: in function ’foo’:
   bar.c:(.text+0x0): multiple definition of ’foo’;
       /tmp/cc01IbiT.o:main.c:(.text+0x0): first defined here
   /usr/bin/ld: /tmp/cckGDjLC.o: in function ’bar’:
   bar.c:(.text+0xf): multiple definition of ’bar’;
       /tmp/cc01IbiT.o:main.c:(.text+0xf): first defined here
   collect2: error: ld returned 1 exit status
   ```

2. For each error, write a small program that generates such a compiler message.

Solution

1. (a) Constant expressions, for example constant variables or literals, can not be assigned new values.

   (b) Calling functions with the wrong number of arguments will result in an error. One exception to this are variadic functions, i.e. functions that allow for an arbitrary amount of arguments to be passed.

   (c) This error is generated when you try to call something as a function that is not callable, for example when the name of a variable is confused with that of a function.

   (d) An undefined reference means that the definition of an object can not be found. While the problem could simply be a misspelling, more commonly this occurs if files are not linked properly.

   (e) Some operators require (modifiable) lvalues as arguments, which are expressions that designate an object (e.g. a variables). Providing rvalues (e.g. literals or temporary return values) instead will result in this error.

   (f) It is possible to show user-defined diagnostics with the #error and #warning preprocessor directives.

(g) Multiple definitions are usually the consequence of definitions occurring in header as opposed to source files. If such a header file is included more than once, such definitions will exist more than once because the #include directive just pastes the contents of the supplied header file.

2.
```
#include <stdio.h>

int g(int arg) { return arg; }

int main()
{
    // 1
    const int a = 1;
    a = 2;

    // 2
    g();

    // 3
    int b = 42;
    b(42);

    // 4
    h();

    // 5
    int c = 1;
    42 = c;

    // 6
    #error "The␣cake␣is␣a␣lie!"

    return 0;
}
```

Exercise 3.10: FooBar Game

1. The rules of the game are as follows:

   (a) Before the game starts, two parameters are set.

   (b) The goal of the game is to count upwards starting from 1 trying to reach as high a number as possible.

   (c) Each round the game asks for input and the player has to enter either the next number, `foo`, `bar` or `foobar`.

   (d) Which of these depends on the current number:

      i. If the current number is only divisible by the first parameter, the player has to enter `foo`

      ii. If it is only divisible by the second parameter, the player has to enter `bar`

      iii. If it is divisible by both, the player has to enter `foobar`

      iv. Otherwise, the current number must be entered.

   (e) The game ends as soon as the player makes a mistake.

   For example, if the parameters are 3 and 4 the first few steps would be

```
Please provide your input: 1
Please provide your input: 2
Please provide your input: foo
Please provide your input: bar
Please provide your input: 5
```

2. Implement the following function, where x and y serve as the parameters of the game.

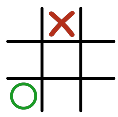
```c
#include <stdio.h>
#include <memory.h>

void foobar(int x, int y){
  /* TODO: Implement FooBarGame */
}
```

Solution

```c
#include <stdio.h>
#include <memory.h>

void foobar(int x, int y){
    int current = 1;
    char input[10];
    char expected[10];
    printf("FooBar␣-␣Game\n␣Parameters:␣%d␣and␣%d\n", x, y);
    while(1){
        printf("Please␣provide␣your␣input:␣");
        scanf("%s", input);
        if (current % x == 0 && current % y == 0){
            strcpy(expected, "foobar");
        }
        else if (current % x == 0){
            strcpy(expected, "foo");
        }
        else if (current % y == 0){
            strcpy(expected, "bar");
        }
        else {
            snprintf(expected, 10, "%d", current);
        }
        if (strcmp(input, expected)){
            printf("Failure␣-␣Expected:␣%s", expected);
            return;
        }
        current += 1;
    }
}
```

Exercise 3.11:

Dieter Schlau just saw this C program online and wants to find out, what exactly it does. Help him by creating an execution trace for the program and calculate what it outputs by hand. You can use a computer afterwards to verify your result



```c
#include <stdio.h>

char func1(int arg1, int arg2) {
  int res = 0;
  int i = arg1;
  while (i > 0) {
    res = res + arg2;
    i = i - 1;
  }
  return res;
}
```

```
12
13 int main(int argc, char* argv[]) {
14   int a;
15   int b;
16   a = 3;
17   b = 7;
18   // b = 8;
19   int c;
20   int d;
21   c = (5 - 3) + (a - 2);
22   d = b + b;
23   a = func1(c, d);
24   printf("The␣answer␣is␣%d!\n", a);
25   return 0;
26 }
```

Solution

Output of the program:

Step	**Line**	a	b	c	d	side effect
1	14	-	-	-	-	
2	15	?	-	-	-	
3	16	?	?	-	-	
4	17	3	?	-	-	
5	18	3	7	-	-	
6	19	3	7	-	-	
7	20	3	7	?	-	
8	21	3	7	?	?	
9	22	3	7	3	?	
10	23	3	7	3	14	call `func1` ->
11	24	42	7	3	14	
12	25	42	7	3	14	print a
13	26	42	7	3	14	termination

Table 1: Call to `main`

Step	**Line**	arg1	arg2	res	i	side effect
1	4	3	14	-	-	
2	5	3	14	0	-	
3	6	3	14	0	3	
4	7	3	14	0	3	
5	8	3	14	14	3	
6	9	3	14	14	2	
7	6	3	14	14	2	
8	7	3	14	14	2	
9	8	3	14	28	2	
10	9	3	14	28	1	
11	6	3	14	28	1	
12	7	3	14	28	1	
13	8	3	14	42	1	
14	9	3	14	42	0	
15	6	3	14	42	0	
16	10	3	14	42	0	return with 42

Table 2: Call to `func1`