Recursion and Recursive Functions

Prepared by: RNC Recario
rncrecario@gmail.com
Institute of Computer Science UPLB
Nov 2011

OBJECTIVES

At the end of the laboratory session, the student is expected to

- Define what a recursion is.
- Identify the parts of a recursive function.
- Manually trace and solve recursive functions.
- Implement a working recursive function in C.

RECURSION

Loosely speaking, recursion can be defined as a process of repeating items in a similar way. It can also be defined as a process of repeating items in terms of itself. Recursion is a very important concept as it is also discussed in the mathematics, linguistics and logic. (In the BSCS program, recursion is emphasized in CMSC 11, 21, 123 and 142). While recursion is the general term to describe the nature of repeating items in a similar manner, recursive function is the term used when recursion is applied in creating functions or modules (although *function* can also mean a mathematical function, but we will stick in the CS definition).

The C programming language supports the use of recursion through recursive functions. Basically, a recursive function is a function that calls itself. A recursive function has two portions: the base case and the recursive case.

- **Base Case**
  - The trivial case of a recursive function
  - It is usually the minimum value that can be accepted by the function
  - The base case is the one that terminates the recursive function; otherwise the function call will not end (a case to similar to infinite loops although the two cases are different).

- **Recursive Case**
  - The non trivial portion of the recursive function.
  - Performs an operation or set of operation to arrive at a certain state or value.

USES OF RECURSIVE FUNCTIONS

Recursive functions are used for mathematical and computer science concepts that are recursive in nature. Examples would be traversing a BST, Ackermann function and fractals (geometric shapes in which parts or portion resemble the whole.

We will now focus on some recursive functions. Consider the function below that computes the sum of two numbers *a* and *b*.

```
int computeSum(int a, int b){
      return(a + b);
}
```

Given the function shown above, how are we going to create a function that does the same task, that is, to add two numbers? First thing to remember is that a recursive function is a function that calls itself thus, we must translate the given function as something like

```
int computeSum(int a, int b){
      return(a + b);
       computerSum(a,b);
}
```

However, a problem will arise in the above setup. Can you think what it is?

To make the recursive function useful, we need to transform it by creating a base case and a recursive case, thus further transform the above function to something like this

```
int computeSum(int a, int b){
     if(){
       /*some nerd codes here*/
     }/*this will serve as our base condition*/
     else{ /*this is our recursive case*/
      return(a + b);
       computerSum(a,b);
     }
}/*end of function*/
```

The above code is a bit correct in a sense that we have shown the two parts of a recursive function: the base case and the recursive case. The two cases are usually done using an if-condition, if-else condition or a ladderized if-else condition.

Again, I want to emphasize that the above code is a *"bit correct"* because it will still not capture the essence of the problem. What do you think is/are the remaining problem(s) of the above code?

At this point, let's analyze the problem. We want to create a recursive function that adds to numbers similar to the original `computeSum()` function (see the first code shown on this handout). In order to think *recursively* (or to make my statement clear, to think in a recursive manner), we must first identify what is being done recursively or in other words, in the addition of two numbers, what is repeatedly done?

The answer is very trivial in case you still do not get the solution: given $a+b$, we will visualize the addition of a and b as a repeated process of adding 1 to a b times. For example, suppose we have $a=4$ and $b=3$, then the process will be 4+1+1+1.

With that in mind, we identify that the base case is when b is already zero (remember, b here serves as a counter how many times 1 has been added to a). The recursive case the one that continually adds 1 to a.

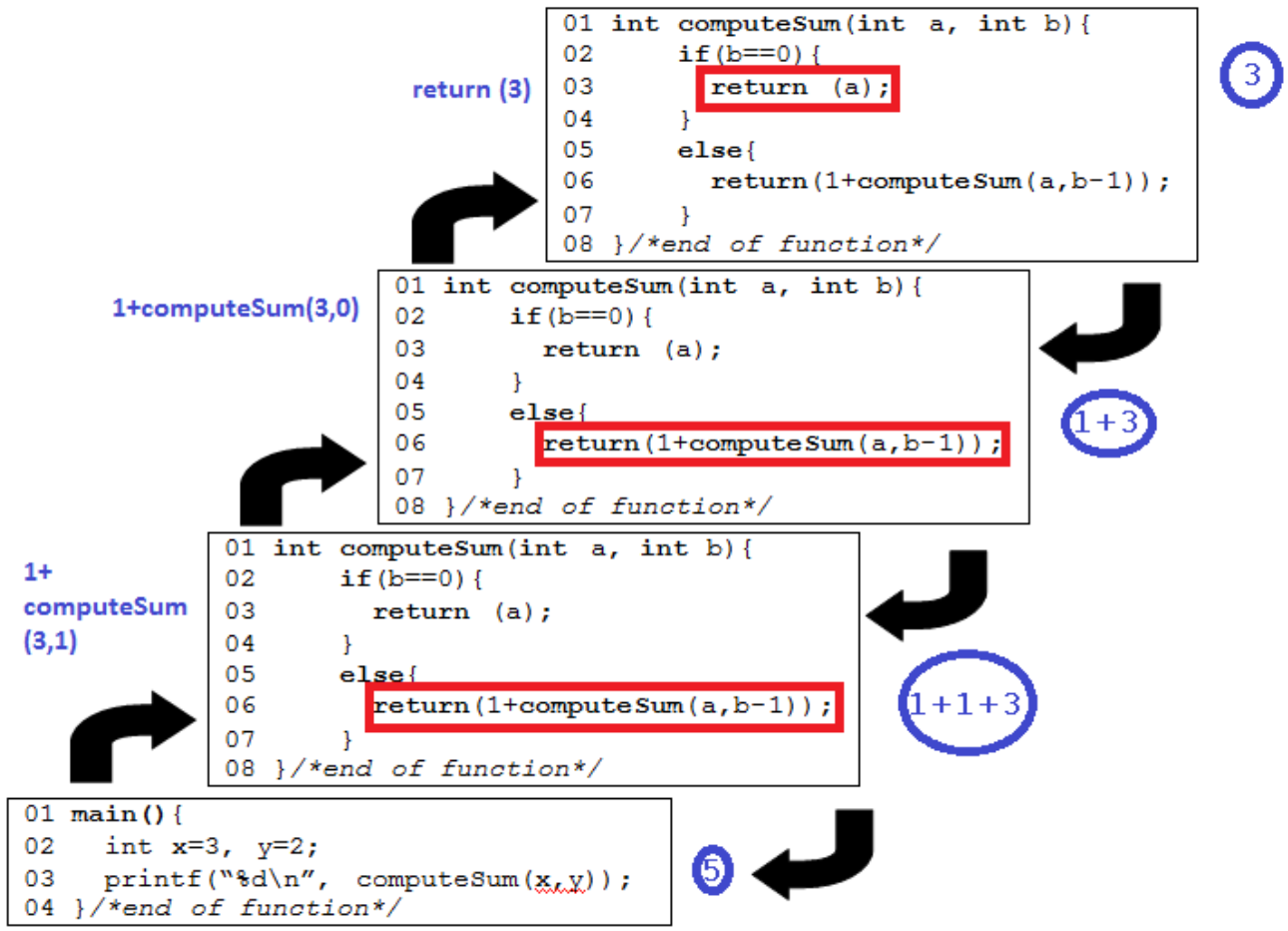
Try to analyze the code below which is the recursive version of our original `computeSum()` function. Identify the base and recursive case and examine how it works.

```
01 int computeSum(int a, int b){
02      if(b==0){
03        return (a);
04      }
05      else{
06        return(1+computeSum(a,b-1));
07      }
08 }/*end of function*/
```

Notice that we did not change the function head. The function body was changed into an if-else condition. The if-condition shows the trivial case which is when *b* is zero (0) (Note: At this point you should be able to notice that our recursive function has a weakness, that is, when *b* is a negative value. In this case, an additional process should be done to handle such case). The else part shows the recursive part, that is, what is done repeatedly.

TRACING A RECURSIVE FUNCTION

Tracing a recursive function is similar to tracing normal functions. Let's use the case of the recursive `computeSum()` shown previously. Suppose a function call of `computeSum(x,y)` is invoked from `main()` with x=3 and y=2.



Now try tracing the recursive function computeSum() using the following values:

- x=6, y=3
- x=-5, y=3
- x=10, y=5
- x=3, y=0

Now, what additional changes can be done to this recursive code so that the case where the second value is negative can be handled? With that adjustment, will the function work for all possible cases of the input? Why? Why not?

ADDITIONAL RECURSIVE FUNCTIONS

Try to trace the following recursive functions and determine their output:

1. The *foo-foo* function. Assume that the initial input value k is equal to 5

```
void foofoo(int k){
      if(k<=0) printf("A");
       else{ printf("U"); foofoo(k-1); }
}
```

2. The *foo-foo-foo* function. Assume that the initial input value k is equal to 9

```
void foofoofoo(int k){
      if(k < 0) printf("A");
       else if (k==0){ printf("B"); foofoofoo(k-1); }
       else{
           printf("C");
           foofoofoo(k-2);
       }
}
```

3. The WilltimeBhigTym function. Assume that the input k is equal to 11.

```
void WilltimeBhigTym(int k){
      if(k < 0) printf("A");
      if(k > 3) printf("B");
      if(k > 1) {printf("C"); WilltimeBhigTym(k-2);}
      else printf("D");
}
```

TIPS ON TRACING AND CREATING A RECURSIVE FUNCTION

- Make sure that there is a base case. Otherwise, your function call will not stop (similar to infinite loops). Technically speaking, what will happen is that you will have a stack error (Stacks will be discussed later on the course).

- Make sure that you identify the trivial case. The trivial case is your stopping condition. The trivial case once translated to code is your base case.

- The recursive case should handle the case where a certain set of actions need to be done repeatedly. Think of loops (ala recusive function version ☺).

RECURSION ON POPULAR COMPUTER SCIENCE CULTURE

- **Recursive Acronyms**
  - GNU —GNU Not Unix
  - PHP —PHP Hyper Processor
  - CYGNUS — Cygnus, Your GNU Support

- **Quine Computing** —is a computer program which takes no input and produces a copy of its own source code as its only output (Wikipedia, 2011). (I suggest you see perform search over the Internet to appreciate the beauty of this! ☺).

Exercise 2: Recursive Functions

Prepared by: RNC Recario
rncrecario@gmail.com
Institute of Computer Science UPLB
Nov 2011

filename: `<surname>_<section>_exer<exerno>.c`
Example: `recario_u1l_exer2.c`

Approximate time to finish exercise: 20 minutes.

Using the same skeleton code from the first exercise, you will need to create transform your `subNum()` and `multNum()` to recursive functions. The only difference with this one is that it includes a USER-DEFINED LIBRARY.

Visit the google sites and download: `recarioexer2.zip`

At this point, you DO NOT need to consider the special cases yet (e.g, numbers are negative). Thus, only positive numbers are considered as input (together with zero). Show to your instructor your work once you are done.

ASSIGNMENT
Deadline: next lab meeting
Filename: `<surname>_<section>_assign<exerno>.c`
Example: `recario_u1l_assign1.c`
Email Subject: `CMSC 21 <section> Assign<no> <Surname>`
Example: `CMSC 21 U1L Assign1 Recario`
Send to: `exercisereceiver@gmail.com`

Your assignment or take home work is to complete the all the operations (add, subtract, multiplication and division) with the special cases under consideration (e.g., negative numbers, division by zero) from this exercise. You will upgrade your work that was checked by your lab instructor.

Be sure to be able to explain your work as your teacher will have random questions about your work.