Computer Science 1FC3

Lab 7 – Recursion

Author – Paul Vrbik – vrbikp@mcmaster.ca

The purpose of this lab is to implement recursive functions into Maple. We will also investigate iteration invariants and bound values as a means of proving our code.

RECURSION

Recursion allows us to make very simple and natural definition of functions that would otherwise have very complicated explicit formula. The strategy to determine a recursive function will be very similar to induction; determine the simplest form (the "base case"), and relate the value at some step to the step immediately after it (the "recursive step"). We will demonstrate how recursion works through a series of examples.

Example 1 (Factorial)

By now we are all familiar with the factorial function that is defined as:

`n!=fact(n)=(n)(n-1)(n-2)…(2)(1)` where `0!=fact(0)=1`

To realize the recursive definition we can rewrite the above definition like:

`n!=fact(n)=(n)*(n-1)…(1)=n*fact(n-1)`
`fact(n)=n*fact(n-1)` where `fact(0)=1`

To implement this definition into maple we do the following:

```
(1)        fact := proc(n:integer)
(2)              if (n=0) then
(3)                    1;
(4)              else
(5)                    n*(fact(n-1));
(6)              end if;
(7)        end:
```

`(1)` Indicates to maple that we would like to define a function and limits `n` to an integer.
`(2)` Defines `fact(0)=1`.
`(4)` Defines `fact(n)=n*fact(n-1)`, that is, it relates the `nth` step to the `(n-1)th` step.
`(5)` Ends the procedure.

Question 1:
What happens if we take out line `(3)` from the above function?

Question 2:
What is `fact(-7)`? What is `fact` of any negative number? Why is this okay?

Example 2 (squaring a number)

We would like to square a number without using multiplication. That is we would like to define

`square(n)=n*n`

using addition.

The first thing we realize is that `square(n)=square(-n)` which means our function needs definition on the positives integers only. Since the smallest positive integer is `0`, `square(0)=0` would be our base case.

Now the difficult part is relating `square(n-1)` to `square(n)`. Well, with a little mathematical investigation we realize that

$$(n + 1)^2 = n^2 + 2n + 1 \quad \text{or}$$

`square(n+1)=square(n)+n+n+1` or

`square(n)=square(n-1)+(n-1)+(n-1)+1`

So breaking down this definition we have:

`square(0)=0`	
`square(n)=square(-n)`	when `n` negative
`square(n)=square(n-1)+n+n-1`	when `n` positive

The maple implementation would be:

```
(1)   square:=proc(n:integer)
(2)         if (n<0) then
(3)               sqaure(-n);
(4)         elif (n=0) then
(5)               0;
(6)         else
(7)               square(n-1)+n+n-1;
(8)         end if;
(9)   end:
```

First note that `elif` is a Maple abbreviation for `else if`. An English translation of `(2)-(8)` could be:

```
if n is negative then do square(-n)
otherwise if n is equal to zero then the answer is zero.
otherwise if n is positive then the answer is square(n-1)+n+n-1.
```

Question 3:

What happens if we take out line `(2)` from the above definition?

Question 4:

What happens if we take out line `(4)` from the above definition?

PROGRAM CORRECTNESS

To prove that a program is correct we must do two things: (1) show that the function will eventually stop and give you a result (2) show that this result is the correct result. When we do this we say that we have *verified* the code.

EXAMPLE 3 (verifying `square`)

We will now show that `square` always returns the correct answer. Well first we ask ourselves what it is that we want `square` to do, well simply put `square(n)=n*n`. So it is now only necessary to check this against all cases outline in the procedure.

```
Case (n<0)                    =square(-n)
              defintion       =square(n)

              check LHS       =(-n)*(-n)=n*n
              check RHS       =n*n

Case (n=0)                    =square(0)
              definition      =0

              check LHS       =0*0
              check RHS       =0

Case (n>0)                    =square(n)
              defintion       =square(n-1)+n+n-1

              check LHS       =n*n
              check RHS       =(n-1)(n-1)+n+n-1=n*n-2n+1+2n-1=n*n
```

When we say that `square(n)` should be equal to `n*n` we call `n*n` the `iteration invariant`.

To show that this definition terminates all we have to do is show that there is some decreasing non-negative number that bounds the number of times the function executes. Well since for given input `n` our procedure will run |`n`|`+2` times (including zero). This means the bound value is clearly |`n`|`+2` where `n` is the number being squared.

Question 5:

Why can we claim that at each step the value of `n` is non-negative and decreasing.

Question 6:

Why can we claim that if the number of times a loop executes is non-negative and decreasing that it eventually the loop will halt?

Example 4 (verifying another `factorial`)

To show that this verification is not always trivial or obvious we will define another recursive factorial function as follows:

```
(1)    fact_iter := proc(c,p)
(2)          if (evalb(c=0)) then
(3)                p;
(4)          else
(5)                fact_iter(c-1,p*c);
(6)          end if;
(7)    end:

(8)    fact2:=n->fact_iter(n,1)
```

Question 7:

What is `fact2(3)` and `fact2(5)`? Do these cases yield correct results?

Question 8:

What does `c` do in this function? If `c=n` originally, at any step, what is the relation of `c` and `p` to the desired value of `n!`

We first claim that in order for `fact2` to be a valid definition of factorial `fact_iter` must be a valid valid defintion of factorial also. In order for `fact_iter` to give the factorial at each step its iteration invariant must be is `c!*p.`

Case `(n=0)`		`=fact_iter(0,1)`	
	definition	`=1`	(by line `(3)` in `fact_iter`)
	check LHS	`=0!*1=1`	
	check RHS	`=1`	
Case `(n>0)`		`=fact_iter(n,1)`	
	definition	`=fact_iter(n-1,1*n)`	(by line `(5)` in `fact_iter`)
	check LHS	`=n!*1` `=n!`	
	check RHS	`=(n-1)!*n` `=n!`	

And since `fact(2)=fact_iter(n,1)=n!*1=n`! we claim that `fact2` also gives the correct results.

Problem Set:

Question 1:

What is the bound value for the factorial procedure given in Example 4? What can we now claim about this procedure.

Question 2:

Let `a` and `b` non-negative integers, define `mult(a,b)` be a recursive function that returns `a*b` using only addition.

a) What is the base case?

b) What is the recursive step?

c) Implement this routine in Maple

d) What is this algorithms iteration invariant?

e) What is this algorithm's bound value?

f) What can we conclude from d) and e)

Question 3:

Let `a` and `b` non-negative integers, define `pow(a,b)` be a recursive function that returns `a^b` using only multiplication.

Repeat steps a) – f) as given in Question 2:

Question 4:

Give a recursive definition for `dot_product` that calculates the dot product of two vectors (given as Maple lists). Recall that `dot_product([a,b,c],[e,f,g])=a*e+b*f+c*g`.