Computer Programming

Recursion. Decision.

Marius Minea

marius@cs.upt.ro

2 October 2017

Review: ways to write a function

Computes a value

```
double discrim(double a, double b, double c)
{
  return b*b - 4*a*c;
}
```

Produces an effect (e.g. prints a message)

```
void myerr(int code)  // void type: returns nothing
{
  printf("error code %d\n", code);
}
```

Has *effect + value* (computes + writes: several statements)

```
int sqrprint(int x)
{
  printf("Computing the square of %d\n", x);
  return x * x;
}
```

Review: structure of a simple program

```c
#include <stdio.h>    // if we need to read/write
#include <math.h>     // if we use math functions

// function definition: third side of a triangle
double thirdside(unsigned a, unsigned b, double phi)
{
  // the expression contains 2 function calls: cos, sqrt
  return sqrt(a*a + b*b - 2*a*b*cos(phi));
} // define before main, call in main

int main(void)
{
  // function call with values for its arguments
  printf("third side: %f\n", thirdside(3, 5, atan(1)));
  return 0;
}
```

Program structure: separating concerns

passing an argument is NOT *reading* from input

computing a value is NOT *writing* it

A function will typically NOT ask for input.
The smallest functions will *receive arguments* and *return results*

This allows them to be composed and used anywhere.

A function will typically NOT print its result, just return it.
(printing is inflexible: may want different format, language, etc.)

We might write "wrapper" functions that ask for input, then call the computation function.

We might also write display functions that get a value and print it.

Functions with and without result

(Computational) problems are solved by writing *functions*.
data: usually given as arguments: `f(3, 7)`, *NOT* read from input

Functions *with* result
produced with the statement `return` *expression* ;
must appear at end of any path (`if` branch) through function
else the function won't return a result!
`warning: control reaches end of non-void function`

CAUTION! in statement `f(5);` returned value is *not used*
use it: `return f(5);` , as parameter `printf("%d", f(5))` , etc.

Functions that *don't return a value*: return type `void`

```
void print_int(int n) { printf("integer %d\n", n); }
```

returns on reaching closing brace OR `return;` (NO expression)
use: standalone in an expression statement: `print_int(7);`

Recursion

any solvable complex problem can be solved using recursion

$\Rightarrow$ recursion is *fundamental in computer science*

Computing arithmetic expressions

Take some expression using integer arithmetic:
$(2 + 3) * (4 + 2 * 3) - 5 * 6/(7 - 2) + (4 + 3 - 2)/(7 - 3)$

Can we compute it?

YES, once we realize the *expression* is the *sum* of two *expressions*

$$\begin{array}{ll} & (2 + 3) * (4 + 2 * 3) - 5 * 6/(7 - 2) \\ + & (4 + 3 - 2)/(7 - 3) \end{array}$$

We then compute the simpler expressions decomposing similarly:

$(2 + 3) * (4 + 2 * 3) - 5 * 6/(7 - 2) = 44$
$(4 + 3 - 2) \ / \ (7 - 3) = 1$
$44 + 1 = 45$

Problem-solving steps

What was essential to compute the expression ?

- *Recognizing the recursive structure*
  expression is sum of two simpler *expressions*
- Expressing the *simplest computation steps*
  we can add, divide, etc. two *numbers*
- Deciding *when to stop*
  if expression is a number, need to do nothing

Recursion: definition, examples

From mathematics, we know recurrence relations for *sequences*:

arithmetic sequence: $\begin{cases} x_0 = b & \text{(i.e.: } x_n = b \text{ for } n = 0) \\ x_n = x_{n-1} + r & \text{for } n > 0 \end{cases}$

Example: $1, 4, 7, 10, 13, \ldots$ $(b = 1,\ r = 3)$

geometric sequence: $\begin{cases} x_0 = b & \text{(i.e.: } x_n = b \text{ for } n = 0) \\ x_n = x_{n-1} \cdot r & \text{for } n > 0 \end{cases}$

Example: $3, 6, 12, 24, 48, \ldots$ $(b = 3,\ r = 2)$

x_n is not computed *directly*, but *step by step*, using x_{n-1}.

A notion is *recursive* if it is *used in its own definition*.

Exercise: write recurrences for: C_n^k, Fibonacci sequence, ...

Recursion: definition, examples

Recursion is fundamental in computer science:
it reduces a problem to a simpler case of the *same* problem

objects: a *sequence* is

- a single element $\bigcirc$
- an element followed by a *sequence* $\bigcirc\overbrace{\bigcirc\bigcirc\bigcirc}^{\text{sequence}}$

e.g. word (sequence of letters); number (sequence of digits)

actions: a *path* is

- a step $\longrightarrow$
- a *path* followed by a step $\overbrace{\longrightarrow\longrightarrow\longrightarrow}^{\text{path}}\longrightarrow$

e.g. traversing a path in a graph

An *expression*:

- number (7)
- identifier (`x`)
- *expression* + *expression*
- *expression* - *expression*
- (*expression*), etc

Example: power function

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

```
#include <stdio.h>
double pwr(double x, unsigned n)
{
 return n==0 ? 1 : x * pwr(x, n-1);
}
int main(void)
{
  printf("-2 raised to 3 = %f\n", pwr(-2.0, 3));
  return 0;
}
```

`unsigned`: type of nonnegative integers (natural numbers)

The *header* of `pwr` is a *declaration* of the function
so it can be used in its own function body (*recursive call*)

Even if we write `pwr(-2, 3)`, -2 (int) will be *converted* to float
(the type declared for each parameter is known)

The mechanism of a recursive call

Same code executed *many times* with *different values*.

The `pwr` function does two computations:
– a *test* (`n == 0` ? *base case* ?) if so, return 1
– else, a multiply; the right operand requires a *new recursive call*

```
pwr(5, 3)
      call↓ ↑125
         5 * pwr(5, 2)
               call↓ ↑25
                  5 * pwr(5, 1)
                        call↓ ↑5
                           5 * pwr(5, 0)
                                 call↓ ↑1
                                    1
```

The mechanism of a recursive call

In the recursive computation of the power function:

Every call makes *a new call*, until the base case it reached

Every call executes *the same code*, but with *other data*
(own values for parameters)

When reaching the base case, all started calls are still *unfinished*
(each has to perform the multiplication with the result of the call)

Returning is done *in opposite order* of the calls
(call with exponent 0 returns, then the one with exponent 1, etc.)

Recursion: power by repeated squaring

Recursion = reduction to a *simpler* case of the *same* problem

Base case is simple enough for direct computation
(can / need no longer be reduced)

$$x^n = \begin{cases} 1 & n = 0 \\ (x^2)^{n/2} & n > 0 \text{ even} \\ x \cdot (x^2)^{n/2} & n > 0 \text{ odd} \end{cases}$$

```
double pow2(double x, unsigned n)
{
  return n == 0 ? 1
    : n % 2 == 0 ? pow2(x*x, n/2) : x * pow2(x*x, n/2);
}
```

Recursion: power by repeated squaring (v. 2)

What happens for $n = 1$?

needless computation of $(x^2)^0$ (which is 1) $\Rightarrow$ rewrite:

$$x^n = \begin{cases} 1 & n = 0 \\ \mathrm{x} & n = 1 \\ (x^2)^{n/2} & n > 1 \text{ even} \\ x \cdot (x^2)^{n/2} & n > 1 \text{ odd} \end{cases}$$

```
double pow2(double x, unsigned n)
{
  return n < 2 ? n == 0 ? 1 : x
    : n % 2 == 0 ? pow2(x*x, n/2) : x * pow2(x*x, n/2);
}
```

Let's follow the recursive calls

```
#include <stdio.h>

double pow2(double x, unsigned n)
{
  printf("base %f exponent %u\n", x, n);
  return n < 2 ? n == 0 ? 1 : x
    : n % 2 == 0 ? pow2(x*x, n/2) : x * pow2(x*x, n/2);
}
int main(void)
{
  printf("5 to the 6th = %f\n", pow2(5, 6));
  return 0;
}
```

Each call halves the exponent $\Rightarrow \lceil \log_2(n+1) \rceil$ calls

`pow2(5, 6)` $\rightarrow$ `pow2(25, 3)` $\rightarrow$ `pow2(625, 1)`

How to use recursion

Recursion solves a problem by reducing it to a simpler case of the same problem.

To use recursion, we must express the problem as a *function*

things given/known to the function are *parameters*

(index of recursive sequence; problem size; etc.)

the answer to the problem is the function *result*

Sometimes, the problem asks to *produce an effect* (print) rather than compute a result.

Block statements and sequencing

A function body may have several statements *in sequence*

```
{
  printf("This is a line\n");
  printf("Line 2: ");
  printf("cos(0)=%f\n", cos(0));
  return 0;
}
```

{
statement
...
statement
}

Function returns on reaching closing brace OR `return` statement.

More generally, a *block* (compound statement) can appear in place of any statement.

This is an example of *recursion* in the *definition of statements*:

statement ::= `return` $expression_{optional}$;
$expression_{optional}$; (incl. function call)
{ *statement* ... *statement* }

The `if` statement

Conditional operator `? :` selects from two *expressions* to *evaluate*

Conditional statement selects between two *statements* to *execute*

Syntax:

```
if ( expression )
    statement1
else
    statement2
```

or

```
if ( expression )
    statement1
```

Effect: If the expression is *true* (nonzero) *statement1* is executed, else *statement2* is executed (or nothing, if the latter is missing)

Each branch has only *one* statement. If several statements are needed, these must be grouped in a *compound statement* `{ }`

An `else` belongs to the closest `if`:
`if`$_1$ (*exp*$_1$) `if`$_2$ (*exp*$_2$) *stmt_then* `else`$_2$ *stmt_else*

The *parantheses* `( )` around the condition are mandatory.

Example with the `if` statement

Printing roots of a quadratic equation:

```
void printsol(double a, double b, double c)
{
  double delta = b * b - 4 *a * c;
  if (delta >= 0) {
    printf("root 1: %f\n", (-b-sqrt(delta))/2/a);
    printf("root 2: %f\n", (-b+sqrt(delta))/2/a);
  } else printf("no solution\n"); // puts("no solution");
}
```

Can rewrite the *conditional operator* `? :` using the `if` *statement*

```
int abs(int x)
{
  return x > 0 ? x : -x;
}
```

```
int abs(int x)
{
  if (x > 0) return x;
  else return -x;
}
```

Recursion: Fibonacci words

Fibonacci sequence: $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ for $n > 1$
inefficient to do direct recursion (exercise: how many calls?)

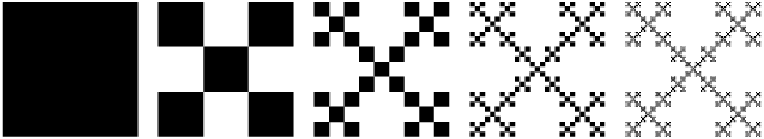
Can define Fibonacci words (strings):
$S_0 = 0, S_1 = 01, S_n = S_{n-1}S_{n-2}$
(formed by string *concatenation*)

Write a function that prints S_n
problem = function; effect = print; concatenation = sequencing

More recursion: fractals

Fractals are *self-similar* figures
(a part of the figure looks like the whole figure = recursion!)

Box fractal:



What is the base case?
What defines a part of the figure?

http://mathworld.wolfram.com/BoxFractal.html

Elements of a recursive definition

1. *Base case*: no recursive call
= simplest case, defined directly
    e.g. in sequences: initial term x_0 of the recurrence
    the empty list (for a list of elements)

A missing base case is an *ERROR* $\Rightarrow$ recursion never stops!

2. *Recurrence relation*
   defines a notion using a simpler case of the same notion

3. *Proof* (argument) that recursion stops in finite number of steps
(e.g. a nonnegative measure that decreases on each application
   for sequences: the index (smaller in definition body but ≥ 0)
   for recursive objects: size (component objects are smaller)

Are the following definition recursive and correct ?

? $x_{n+1} = 2 \cdot x_n$
? $x_n = x_{n+1} - 3$
? $a^n = a \cdot a \cdot \ldots \cdot a$ (n times)
? a sentence is a sequence of words
? a sequence is the concatenation of two smaller sequences
? a string is a character followed by a string

A recursive definition must be *well formed* (conditions 1-3)

something cannot be defined only in terms of itself

one can only use other notions which are already defined

computation has to stop at some point

Recursion in numbers: sequences of digits

A natural number (in base 10) can be defined/viewed recursively:
a number is a *single digit*
or: *last digit* preceded by *another number* (in base 10)

We can find the two parts using integer division (with remainder)

$n = 10 \cdot (n/10) + n\%10$	$1457 = 10 \cdot 145 + 7$
the last digit of n is $n\%10$	$1457\%10 = 7$
the number remaining in front is $n/10$	$1457/10 = 145$

Exercises with a simple recursive solution:
sum of a number's digits
number of digits; largest/smallest digit, etc.

Solution: always *follow the structure of the recursive definition*
base case: *directly give result* for single-digit number
recurrence: *combine* last digit with result for *remaining number* (n/10)

How many digits in a number?

1, if number < 10
else, one digit more than the number without its last digit ($n/10$)

```
unsigned ndigits(unsigned n)
{
  return n < 10 ? 1 : 1 + ndigits(n / 10);
}
```

Alternative: use an *accumulator* for the digits already counted
  start from 1 (last digit already counted; surely has one)
  if the number is single-digit, return the digits already counted
  else, $n/10$ still has (at least) one digit, add 1 to parameter

```
unsigned ndigs2(unsigned n, unsigned r)
{
  return n < 10 ? r : ndigs2(n / 10, r + 1);
}
```

Need function with only one parameter: wrap auxiliary function
(called with starting value 1: single-digit number)

```
unsigned ndig(unsigned n) { return ndigs2(n, 1); }
```

Largest digit in a number

base case: single-digit number (digit is also max)
else, max of last digit and result for the remaining number

```
unsigned max(unsigned a, unsigned b) { return a > b ? a : b; }
unsigned maxdigit(unsigned n)
{
  return n < 10 ? n : max(n%10, maxdigit(n/10));
}
```

Variant with accumulator: maximal digit seen so far: `md`
- if 0 (no more digits), return the maximum so far: `md`
- else, continue with maximum of last digit and previous max

```
unsigned maxdig2(unsigned n, unsigned md)
{
  return n == 0 ? md : maxdig2(n/10, max(md, n%10));
}
unsigned maxdig(unsigned n) { return maxdig2(n/10, n%10); }
```

Two ways of writing recursion

```
unsigned max(unsigned a, unsigned b) { return a > b ? a : b; }

unsigned maxdig(unsigned n) {
  return n < 10 ? n : max(n%10, maxdig(n/10));
} // directly from: number ::= digit | number digit

unsigned maxdig2(unsigned n, unsigned maxd) {
  unsigned md1 = max(n%10, maxd);
  return n < 10 ? md1 : maxdig2(n/10, md1);
} // keep maxd found so far

unsigned maxdig1(unsigned n) {
  return n < 10 ? n : maxdig2(n/10, n%10);
} // 1-arg wrapper for function above
```

Is recursion efficient?

$$S_0 = 1, \quad S_n = S_{n-1} + \cos n \qquad S_{1000000} = ?$$

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

double s(unsigned n) {
  return n == 0 ? 1 : s(n-1) + cos(n);
}

int main(void) {
  printf("%f\n", s(1000000));
  return 0;
}
```

```
./a.out
Segmentation fault
```

Recursion and the stack

Code executes sequentially (except for branch/call/return)

On function call, must remember *where to return* after call

Must store *function parameters and locals* to keep using them

These are placed on the *stack*

Each function activation has its *stack frame*: arguments, return address, local vars

Nested calls return in opposite order made
⇒ stack frames popped in reverse order of saving (last in, first out)

For deep recursion, stack may be insufficient
⇒ program crash

locals of f(0)
retaddr: to f(1)
args to f: n=0
locals of f(1)
retaddr: to f(2)
args to f: n=1
locals of f(2)
retaddr: to main
args to f: n=2
locals: main

Tail recursion

$S_0 = 1, \quad S_n = S_{n-1} + \cos n$
We *know* we'll have to add $\cos n$ (but not yet to what)
$\Rightarrow$ can *anticipate* and *accumulate* values we need to add

When reaching the base case, add accumulator (partial result)

```
double s2(double acc, unsigned n)
{
  return n == 0 ? acc : s2(acc + cos(n), n-1);
}

double s1(unsigned n) { return s2(1, n); } // call w/ S0=1
```

Program now works!

Tail recursion is iteration!

A function is *tail-recursive* if recursive call is *last* in the function.
no computation done after call (e.g., with result)
result (if any) is returned unchanged between calls

$\Rightarrow$ parameter and local values no longer needed
$\Rightarrow$ *no need for stack*: replace *recursive* call with jump,
return value at end (base case)

(Optimizing) compiler converts tail recursion to iteration (loop)
need not worry about efficiency

Recursion can express arbitrary repetition

Base case: are we done? return (result)

Recursive case (not done):
  compute new partial result
  call recursive function with new partial result
    (usually an extra parameter, besides initial input)

Exercise: rewrite Fibonacci
  extra parameters: last, previous number
  stopping condition: all iterations done

Recursion: reverse digits in number

Often, problem restated with explicit *partial result* (accumulator)

n	r
1465	empty(0)
146	5
14	56
1	564
empty(0)	5641

What is the result of reverting
given that
the end has already been reverted
the resulting number is `r`
and remaining part is `n`?

```
unsigned rev2(unsigned n, unsigned r) {
  return n == 0 ? r : rev2(n/10, 10*r + n % 10);
}
// initial reversed part is zero
unsigned rev(unsigned n) { return rev2(n, 0); }
```

Careful: `return` in base case *must use accumulator*
(else computation is thrown away!)

Recursion for computing approximations: square root

Babylonian method: $a_0 = 1$, $a_{n+1} = \frac{1}{2}(a_n + \frac{x}{a_n})$

recurrent sequence of approximations $\Rightarrow$ recursive solution

given (parameters): x and the current approximation

result = a satisfactory approximation (precision ϵ)

Re-state problem: compute $\sqrt{x}$ *given current approximation* a_n

In recursion, partial result is usually carried as parameter

Computation:

if precision good $|a_{n+1} - a_n| < \epsilon$ return *current approximation* a_n (base case)

else, return value computed starting from *new approximation* a_{n+1} (recursive call)

We no longer need an index n, and the base case is not $n = 0$
(but it's still the case when nothing left to compute)

Can prove: error to $\sqrt{x}$ is less than distance between last two terms

Square root by approximation

```c
#include <math.h>
// needed for double fabs(double x); (abs. value for reals)

// root of x with error < 1e-6 given approximation a_n
double root2(double x, double an)
{
  return fabs(a_n - x/a_n) < 2e-6 ? a_n
    : root2(x, (a_n + x/a_n)/2);
}
double root(double x) { return x < 0 ? -1 : root2(x, 1); }
```

Two functions:
auxiliary `root2` needs two parameters (also approximation)
for user: `root` defined as required: only one parameter
returns -1 for negative numbers (error code)

Recall: this form is *tail recursion*: recursive call is *last* computation.
Compiler can convert this to *iteration* (efficient).

Review: conditional expression

condition `?` *expr1* `:` *expr2* everything is an *expression*

expr1 or *expr2* may be conditional expression themselves
(if we need more questions to find out the answer)

$$f(x) = \begin{cases} -6 & x < -3 \\ 2 \cdot x & x \in [-3, 3] \\ 6 & x > 3 \end{cases}$$

```
double f(double x)
{
  return x < -3 ? -6      // else, we know x >= -3
                : x <= 3 ? 2*x : 6;
}
```

or: `x >= -3 ? (x <= 3 ? 2*x : 6) : -6`
if $x \geq -3$ we still need to ask $x \leq 3$?

or: `x < -3 ? -6 : (x > 3 ? 6 : 2*x)`
if x is not < -3 or > 3, it must be $x \in [-3, 3]$

Conditional expression (cont'd)

The conditional expression is an expression
  ⇒ may be used *anywhere* an expression is needed

Example: as an expression of type string
puts: function that prints a string to stdout, followed by a newline

```
void printsgn(int n)
{
  puts(n == 0 ? "zero"
        : n > 0 ? "positive"
        : "negative");
}
```

Note layout for readability: one question per line.

Expressions and statements

Expression: *computes a result*

arithmetic operations: `x + 1`

function call: `fact(5)`

Statement: *executes an action*

`return n + 1;`

Any *expression* followed by `;` becomes a *statement*

`n + 3;` (computes, but does not use the result)

`printf("hello!");` we do not use the *result* of `printf`

but are interested in the *side effect*, printing

`printf` returns an `int`: number of chars written (rarely used)

Statements contain expressions. Expressions don't contain statements.

Sequencing for statements and expressions

Statements are written and executed in order (*sequentially*)

With *decision*, *recursion* and *sequencing* we can write any program

Compound statement: several statements between *braces* `{ }`
A *function body* is a compound statement (*block*).

```
{
    statement
    ...
    statement
}
```

```
{
  double pi = acos(-1);
  printf("pi = %f\n", pi);
  double diff = sqrt(.5) - sin(pi/4);
  printf("difference: %f\n", diff);
}
```

A compound statement is considered *a single statement*.
May contain declarations: anywhere (C99/C11)/at start (C89).
All other statements are *terminated* by a semicolon `;`

The *sequencing operator* is the *comma*: *expr1* `,` *expr2*
evaluate *expr1*, ignore; evaluate *expr2* ⇒ value of whole expression

Decisions with multiple branches

The branches of an `if` can be any statements

⇒ also `if` statements

⇒ can chain decisions one after another

```
void binop(int op, int a, int b) // op: operator (char)
{
  if (op == '+') printf("sum: %d\n", a + b);
  else if (op == '-') printf("diff: %d\n", a - b);
  else puts("bad operator");
}
```

Checks `op=='+'` and `op=='-'` are *not independent*. *DON'T* write

~~`if (op == '+') printf("sum: %d\n", a + b);`~~

~~`if (op == '-') printf("diff: %d\n", a - b);`~~

It is pointless do the second test if the first was true

(op cannot be both + and - at the same time)

The proper code is with chained `if`s (or a `switch` statement)

Decisions with multiple branches

If each branch ends with returning a value, the `else` is not needed:
we only get to a branch if the previous condition was false
(else the function will have returned):

```
int binop(int op, int a, int b) // op: operator (char)
{
  if (op == '+') return a + b;
  if (op == '-') return a - b; // can't be here for op == '+'
  puts("bad operator"); return 0;    // any other case
}
```

Often, we first deal with error cases, then do the actual processing:

```
int check_interval(int n) {
  if (n > 100) { puts("number too big"); return -1; }
  if (n < 0) { puts("number is negative"); return -1; }
  // do something with n here
  return 0;    // means OK
}
```