Computer Programming

Decision. Recursion. Characters

Marius Minea

marius@cs.upt.ro

3 October 2016

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);
}
```

effect + value (computes + writes: several statements)

```
int sqr(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));
} // will be called in main --> define before

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

We solve a (computational) problem by writing a *function*.

function *parameters*: the *input data*, used to compute result
NOT read from input, but given in function call: `f(3, 7)`

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* (e.g., just print)
declare function with 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: 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)

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

```
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

```c
#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*) or if (*expression*)
  statement1   *statement1*
else
  statement2

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* `{ }`

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?)

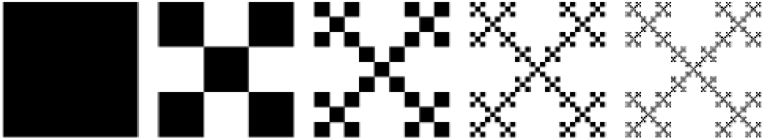
Can define Fibonacci words (strings):
$S_0 = 0, S_1 = 01, S_n = S_{n-1}S_{n-2}$
(formed by string *concatenation*)

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!)

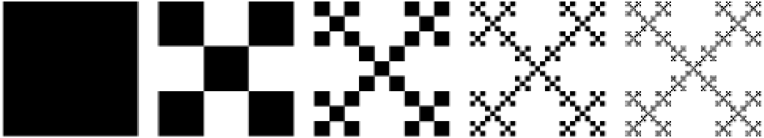
Box fractal:



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

3. Proof/argument that recursion stops in a 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 $n/10$	$1457/10 = 145$

Problems 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?

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

```
#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)

When calling a function, must remember *where to return*
(right after call)

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

These are placed on the *stack*
since nested calls return in opposite order made
must restore values in reverse order of saving (last in, first out)

If recursion is very deep, stack may be insufficient $\Rightarrow$ program crash
even otherwise, save/call/restore may be expensive

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

⇒ parameter and local values no longer needed
⇒ *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

Characters. ASCII code

ASCII = American Standard Code for Information Interchange

Characters are represented as a numeric code = index in this table

e.g. `'0' == 48`, `'A' == 65`, `'a' == 97`, etc.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x0	\0							\a	\b	\t	\n	\v	\f	\r		
0x10:																
0x20:		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
0x30:	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0x40:	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0x50:	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0x60:	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0x70:	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

Prefix `0x` denotes *hexazecimal constants* (in base 16)

Characters < `0x20` (space): *control characters*

digits; uppercase letters; lowercase letters: 3 contiguous sequences

ASCII: only up to `0x7f` (127); then national chars, multi-byte, etc.

The character type

The standard type `char` is used to represent characters
`char` is an *integer type*, with smaller range than `int` or `unsigned`
⇒ can be stored in a byte (`CHAR_BIT` $\geq$ 8 bits)

`char` can be `signed char`, at least -128 to 127,
or `unsigned char`, at least 0 to 255. Both are included in `int`.

character constants are written betweeen (single) *quotes* `' '`
They are *integer values*. In expressions: *implicitly converted to* `int`
Digits, lowercase letters and uppercase letters are *consecutive* ⇒
`'7' == '0' + 7   '5' - '0' == 5   'E' - 'A' == 4   'f' == 'a' + 5`

Escape sequences (textual representation) for special chars:

`'\0'`	null	`'\n'`	newline
`'\a'`	alarm	`'\r'`	carriage return
`'\b'`	backspace	`'\f'`	form feed
`'\t'`	tab	`'\''`	single quote
`'\v'`	vertical tab	`'\\'`	backslash

Writing a character: `putchar`

Declaration, in `stdio.h` : `int putchar(int c);`
Call (sample use): `putchar('7')`

Writes an `unsigned char` (given as `int`); returns its value, or `EOF` (constant -1) on error

```
#include <stdio.h>
int main(void)
{
  putchar('A'); putchar(':'); // writes A then :
  putchar(getchar());        // prints character read
  return 0;
}
```

Chars are just ints (stored in one byte).
'A' is just another way of writing 65 .

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 in `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              int c = getchar();
    ...                    printf("let's print the char: ");
    statement              putchar(c);
}                        }
```

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, the value of the expression is that of *expr2*

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
}
```

Example with `if`: printing a number

```c
#include <stdio.h>

void printnat(unsigned n) { // recursive, digit by digit
  if (n >= 10)              // if it has several digits
    printnat(n/10);         // write first part
  putchar('0' + n % 10);    // always write last digit
}

int main(void)
{
  printnat(312);
  return 0;
}
```