Programming Language Concepts
Control Flow

Janyl Jumadinova

20–22 March, 2023

Conditional Branches

Familiar to most novice programmers:

- "if" and "if-else" statements "switch" statements
- *Basic idea*: `if (condition) then ... else ...`
- It wasn't always quite this easy, though

Conditional branches–switch statements

In C and Java:

```
switch(i) {
   case 0:
   case 2:
   case 4: System.out.println(i+": even, <= 4");
           break;
   case 1: System.out.println(i+" is one");
           break;
   default:
}
```

Conditional branches–switch statements

Without `break` statements?

```
i=0;
switch(i) {
   case 0:
   case 2:
   case 4: System.out.println(i+": even, <= 4");
   case 1: System.out.println(i+" is one");
   default: System.out.println(i+": odd or > 4");
}
```

Short Circuit Evaluation

According to the laws of logic, order doesn't matter in "and":

"p AND q" is the same as "q AND p".
Similarly, for OR.

Short Circuit Evaluation

According to the laws of logic, order doesn't matter in "and":

"p AND q" is the same as "q AND p".
Similarly, for OR.

But in Java and C, order of evaluation is important:

Short Circuit Evaluation

According to the laws of logic, order doesn't matter in "and":

"p AND q" is the same as "q AND p".
Similarly, for OR.

But in Java and C, order of evaluation is important:

```
int i = 10, j = 0, k = 0;
if (i > 10 && 5/j < 3) {
   k = 5;
}
```

Since $i > 10$ is false, there is no need to look at the second condition–we already know that the "&&" will be false.

Short Circuit Evaluation

If we switch the ordering:

Short Circuit Evaluation

If we switch the ordering:

```
int i = 10, j = 0, k = 0;
if ( 5/j < 3 && i > 10 ) {
   k = 5;
}
```

If we start with $5/j < 3$, we"ll get a "division by zero" error.

Short Circuit Evaluation

Short circuit evaluation is used often in situations like this:

```
if (i >= 0 && sqrt(i) > 5.0) ...
```

By checking $i >= 0$ first, we guarantee that we won't try taking square root of a negative value.

Short Circuit Evaluation

Short circuit evaluation is used often in situations like this:

```
if (i >= 0 && sqrt(i) > 5.0) ...
```

By checking $i >= 0$ first, we guarantee that we won't try taking square root of a negative value.

More generally,

`if (valid(data) && meets_criteria(data)) ...` It is more efficient than evaluating both operands and then performing an "and" or an "or" on them.

Short Circuit Evaluation

- What if, for some reason, we WANT both operands to be evaluated?

Short Circuit Evaluation

- What if, for some reason, we WANT both operands to be evaluated?
- Languages like Ada provide for both full evaluation of all operands and also short-circuit operations:

```
if (a and b) : full evaluation of both a and b
if (a and then b) : short-circuit--quit if a is false
```

Old FORTRAN Days

```
      if (i+j-k)10,20,30
10    print *,"i+j-k is negative"
      go to 40
20    print *,"i+j-k is zero"
      go to 40
30    print *,"i+j-k is positive"
40    stop
      end
```

Evaluate `i+j-k` and take one of three branches:

statement 10 if `i+j-k` < 0,
statement 20 if `i+j-k` = 0,
statement 30 if `i+j-k` > 0

(You can run this in the lab -- look for file “arith-if.for” in the repository and follow instructions in comments.)

The "go to" Statement

- "go to" is an UNCONDITIONAL branch.
- Most early programming languages had "go to" statements.
- Later languages like C also adopted them.
- But, they were easy to misuse.

The “go to” Statement

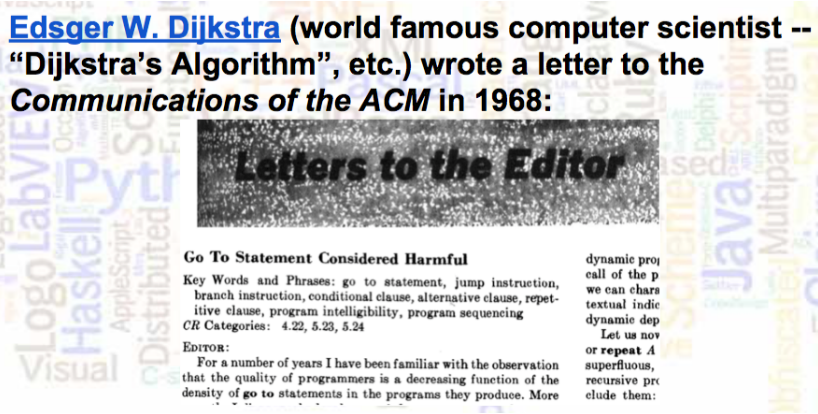
(Contrived) Example (in C):

```
   for (i = 0; i < 5; i++) {
      if (i==3) goto OUTSIDE;
 INSIDE: printf(“inside\n”);
   }
   goto FINISH;
 OUTSIDE: printf(“outside\n”);
   goto INSIDE;
 FINISH: ...
```

OUTPUT:

```
inside
inside
inside
outside
inside
inside
```

The "go to" Statement



[Edsger W. Dijkstra](#) (world famous computer scientist -- "Dijkstra's Algorithm", etc.) wrote a letter to the *Communications of the ACM* in 1968:

Letters to the Editor

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

EDITOR:
For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More

dynamic pro
call of the p
we can chara
textual indic
dynamic dep
Let us nov
or **repeat** *A*
superfluous,
recursive pro
clude them:

The “go to” Statement

HUGE response. Letter is now famous; many imitations. “Considered harmful” essays appear about almost every topic in computer science:

- XMLHttpRequest Considered Harmful
- Csh Programming Considered Harmful
- Turing Test Considered Harmful
- Considered Harmful Essays Considered Harmful
- ... etc ...

The "go to" Statement

But why?

- We can "break out of scope" with a goto (the for-loop block might have its own local variables)
- We can write incomprehensible code ("spaghetti code")

Class Activity: C and goto

Loops

```
while: while (condition) {
   loop body
}
```

The loop body is executed zero or more times (the condition might be false from the very beginning).

Loops

```
do: do {
   loop body
} while (condition);
```

Loops

```
do: do {
   loop body
} while (condition);
```

The loop body is executed one or more times (the condition is not tested until after the loop body has been executed at least once).

do...while Is Syntactic Sugar

We can achieve the same effect as a “do while” using a plain while loop, for instance:

```
while (true) {
   loop body
   if (! condition) break;
}
```

Iterators

In Java we can do things like this:

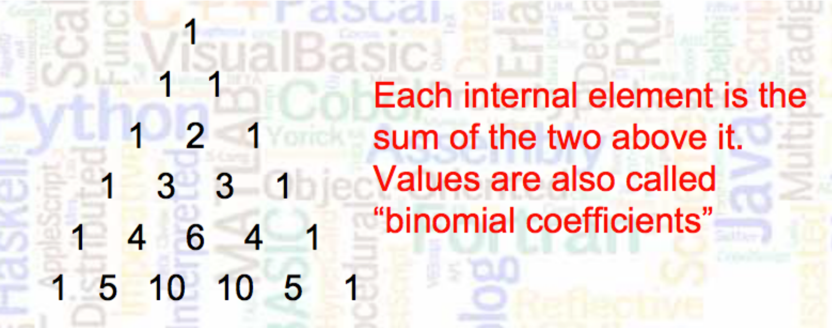
```
String[] words = {"cat", "dog", "bird", ...};
...
for (String s : words) {

}
```

Most compound data types in Java include an iterator feature.

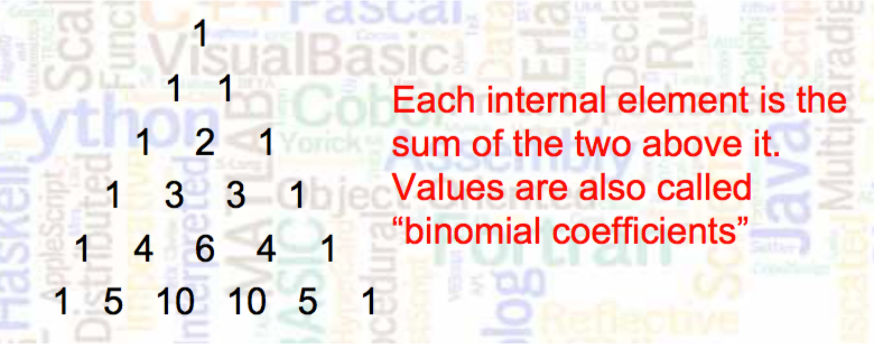
Recursion

Recursion represents a certain special kind of “control flow”.

Problem: compute the values in the Pascal’s triangle



Recursion


1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
Each internal element is the sum of the two above it.
Values are also called "binomial coefficients"

Recursion

Reorganize and number rows and columns:

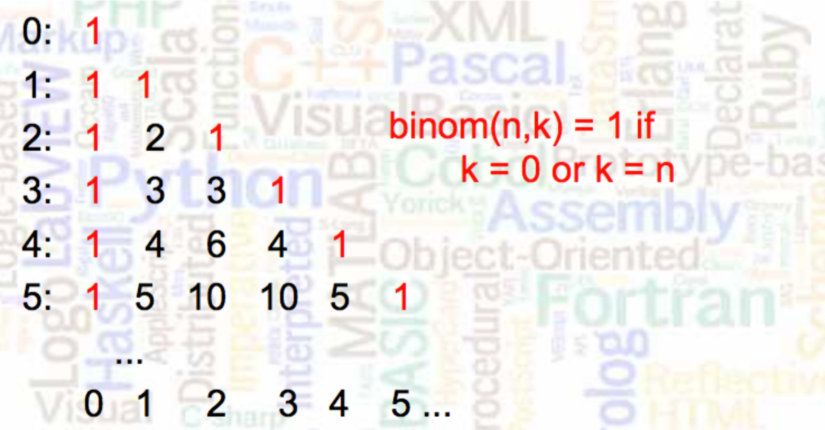
0:	1					
1:	1	1				
2:	1	2	1			
3:	1	3	3	1		
4:	1	4	6	4	1	
5:	1	5	10	10	5	1
...						
	0	1	2	3	4	5 ...

Rows: n

Columns: k

binom(4,2) = 6

Recursion


0: 1
1: 1 1
2: 1 2 1
3: 1 3 3 1
4: 1 4 6 4 1
5: 1 5 10 10 5 1
...
0 1 2 3 4 5 ...
binom(n,k) = 1 if
k = 0 or k = n

Recursion

binom(n,k) = binom(n-1,k-1) + binom(n-1,k)

```
0:  1
1:  1   1
2:  1   2   1
3:  1   3   3   1
4:  1   4   6   4   1
5:  1   5  10  10   5   1
   ...
    0   1   2   3   4   5 ...
```

Recursion

```
public static int binom(int n, int k) {
  if (k == 0 || k == n) {
      return 1;
  }
  else {
      return binom(n-1,k-1)+ binom(n-1,k);
  }
}
```

Recursion

```
public static int binom(int n, int k) {
    int retvalue, temp1, temp2;
    if (k == 0 || k == n) {
        retvalue = 1;
    }
    else { // recursive case:
      temp1 = binom(n-1,k-1);
      temp2 = binom(n-1,k);
      retvalue = temp1+temp2;
    }
    return retvalue;
}
```

Recursive calls always take us back to the beginning of the function

"Returns" could take us back to a location in the function or to some external location.

Recursion

- Let's eliminate explicit recursion and instead simulate the behavior of the frame stack.
- We will need a "frame" to hold values of local variables `n, k, temp1, temp2` (and `retvalue`, but in this example we don't need it so we'll skip it).
- The frame must also hold a "return address", which we will simulate with an integer value.

Recursion

```
private int n, k, t1,t2; // parameters and local variables
private int ra;          // return address
// Constructor
public Frame (int n, int k, int ra, int t1, int t2) {
   this.n = n; this.k = k; this.ra = ra;
   this.t1 = t1; this.t2 = t2;
}
public int n() {return n;}
public int k() {return k;}
public int ra() {return ra;}
public int t1() {return t1;}
public int t2() {return t2;}
```

Recursion

And we'll need a stack:

```
import java.util.Stack;
     ...
Stack<Frame> stack = new Stack<Frame>();
```

Recursion

And we'll need a stack:

```
import java.util.Stack;
     ...
Stack<Frame> stack = new Stack<Frame>();
```

– Each recursive call is replaced with a "push" to the stack; execution then goes back to the top of the function.
– Each "return" is replaced by a "pop" and a return to the location in the (popped) return address.

Recursion

The heart of the "binom" function is an infinite loop that uses the return address variable ra to "goto" the correct section of code to simulate a return from a recursive call.

```
  ...
while (true) {
   switch(ra) {
      case 0: // base case test: go here when first entering the function
             ...
      case 1: // First recursive call to binom.
              ...
      case 2: // Second recursive call to binom.
              ...
      case 3: // We just returned from the second recursive call.
              ...
      case 4:...
   }
}
```

Recursion

To prepare to simulate a recursive call, we save values onto the stack, update to new values, and return to the top of the loop by setting ra to 0. E.g., here's the first recursive call to `binom(n-1,k-1)`:

```
stack.push(new Frame(n,k,2,temp1,temp2));
n=n-1; k=k-1;ra=0;
   continue;
```

Recursion

To simulate a "return", we see if there is anything in the stack (if not, then binom was called from an external function). Pop the stack, restore old variable values, and go to the popped return address:

```
// Is this a top-level call? Then return:
if (stack.empty())
   return retvalue;
else {
     Frame s = stack.pop();
     n = s.n(); k = s.k(); ra = s.ra(); // go here next
     temp1 = s.t1(); temp2 = s.t2();
}
```