CS201 - PL'S

Control Flow - 02

Aravind Mohan

Allegheny College

September 29, 2021

Revisit Assignments and Expressions

- In an assignment statement such as:
  $i = i + 1$
- the variable `i` plays two different roles.
- On the left, it stands for a memory location, or reference, also called an "l-value".
- On the right, it represents a value ("r- value").
- Languages such as Java, C, etc., use a mix of values and references.

Assignment Statements in Functional Languages

- In a functional language (such as Racket in lab 4), there are no variables, hence no assignment statements.
- A program in a functional language is just a collection of expressions to be evaluated.
- More on functional languages in a few weeks.

Initialization

- Some languages allow a variable to be initialized at the same time it is declared:
  `int i = 10`
- Not all languages check to see if a variable is initialized:

```
int main() { /* C example */
    int i,j,k;
    printf("i,j,k = %d %d %d \n",i,j,k);
}
```

Output: *i,j,k = 32767 1740734558 32767*

Initialization

- Compilers can often determine whether a variable is initialized at the point where it is used as an r-value, e.g., Java

```
...
int i;
int j = i;
...
```

javac init.java

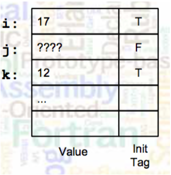
Init.java:12: error: variable i might not have been initialized

Initialization

Unitialized variables can also be detected at runtime, but it can be very expensive.

```
...
j = i*j;
...
```

if `tag(i)` then load `i`
`else` throw “uninitialized error”
if `tag(j)` then load `j`
`else` throw “uninitialized error”



Automatic Initialization

- In some languages, (some) variables are automatically initialized to a default value.
- **Example**: Java initializes all instance variables (e.g., ints and doubles are zero, objects are null, etc.), but not local variables

Expressions

Precedence and associativity rules aren't the whole story

```
x = 4;
i = f(3*x+1, x = 1, 2*x);
```

- Are the arguments evaluated from left to right?
- Is the value of `x` first stored in a register, after which each parameter is evaluated.

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.

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:

```
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.

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?
- 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
```

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);
```

The loop body is executed one or more times (the condition isn't 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

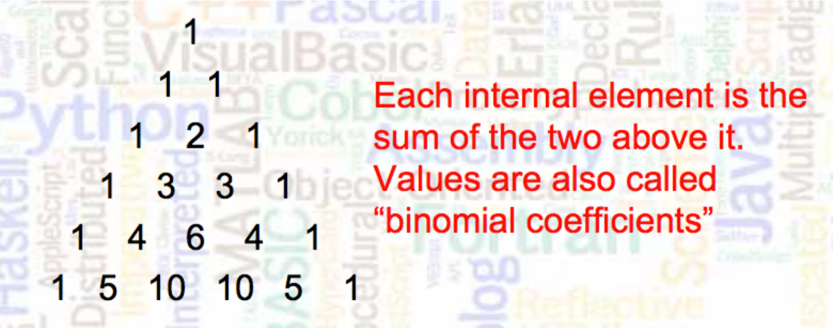
In Java we can do things like this:

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

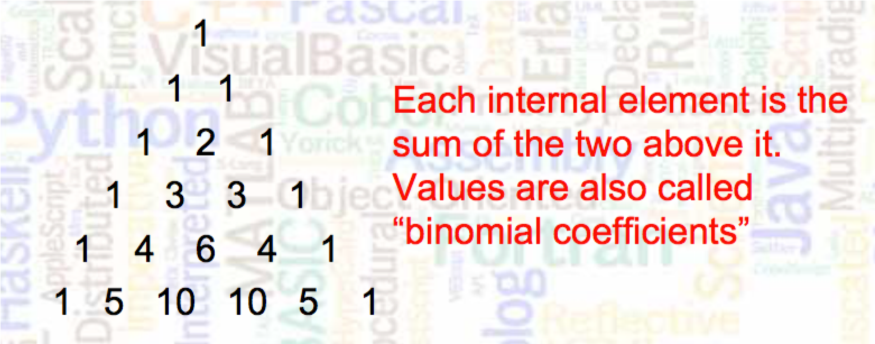
Most compound data types in Java include an iterator feature.

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

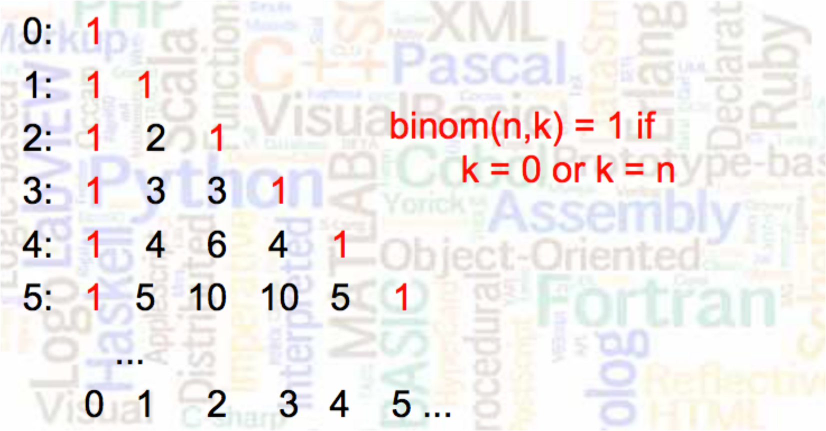
Reorganize and number rows and columns:

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

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

Recursion

```java
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;}
```

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;
```

Reading Assignment

PLP Chapter 06 [6.1.5; 6.4 - 6.6]

Questions

Do you have any questions from this class discussion?