Programming Paradigms

Unit 5 — Recursion

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE

Outline

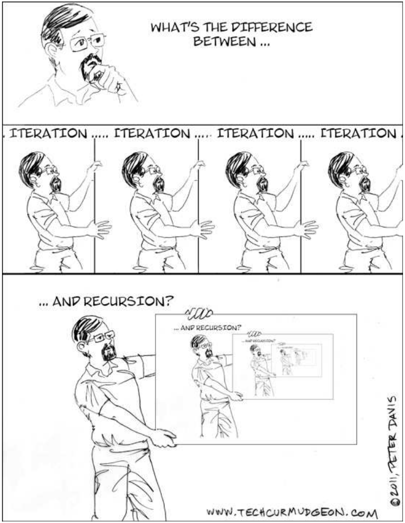
Outline

1. Recursion

2. Writing Recursive Functions

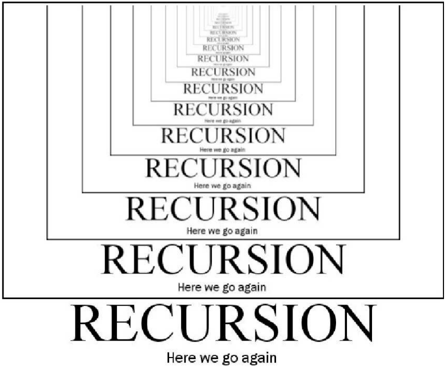
A Different Kind of Loop/1

- The languages we are covering next have a lack of iterative constructs
  - That means, they have no loops
- That does not mean that they are less expressive, they use recursion, instead



A Different Kind of Loop/2

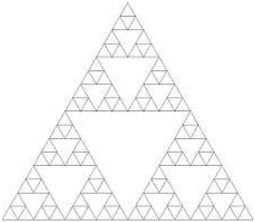
- It depends on the language how easy and efficient one or the other is
  - Some languages lack recursion: Fortran77, Assembler
  - Some languages allow recursion, but aren't very efficient with it: C++, Java
  - Languages we cover next are optimized for recursion

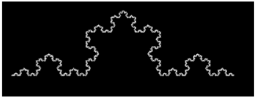


Basic Idea of Recursion

- The basic idea of recursion is “Divide et Impera”, and is composed of three elements:
  - Recursive case: Divide a problem P into subproblems with the same structure, but smaller
  - Base case: At some point, the subproblem is small enough to solve it
  - Composition: Once the subproblems are solved, they can be composed to solve P
- Many problems can be expressed very elegantly and naturally with recursion

To iterate is human, to recurse divine (Anonymous)





$$n! = n * (n-1)!$$

General Structure of a Recursive Solution/Program

```
solve( problem $p$ )
   if ( $p$ is simple )
      Solve the problem $p$ directly
   else
      Divide $p$ into new sub-problems $p_1$, $p_2$, ...
      $r_1$ = solve($p_1$)
      $r_2$ = solve($p_2$)
         ⋮
      Reassemble $r_1$, $r_2$, ... to solve $p$
   end
end
```

Example: The Handshake Problem

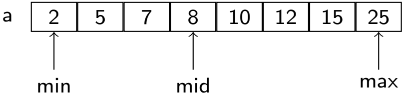
- There are n people in a room, and each person shakes hands once with every other person.
- What is the total number of $h(n)$ of handshakes?

- Recursive solutions seems very natural
  - 2 persons: $h(2) = 1$
  - 3 persons: $h(3) = h(2) + 2$

    $\vdots$
  - n persons: $h(n) = h(n-1) + (n-1)$
- That is, the *n-th* person shakes $n-1$ hands in addition to the number of handshakes of the previous $n-1$ people
- Same as sum of $1 + 2 + \cdots + (n-1) = \frac{n \cdot (n-1)}{2}$

Example: Binary Search

- Binary search for an element *v* in a sorted array *a*
    - Compare *v* with the middle element of *a*
    - If not equal, apply binary search to half of *a* where *v* would be

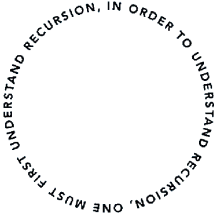


```
bsearch(a, i, j, v)
   if ( a[mid] = v )
      return mid
   else if ( i ≥ j )
      return not found
   else if ( v ≤ a[mid] )
      return bsearch(a, min, mid-1, v)
   else
      return bsearch(a, mid+1, max, v)
   end
end
```

Iterative Sum Example

- Let's convert a simple loop into recursion
- We're going to stay with Ruby for a while longer and write a function that computes a sum:

```
def sum(n)
   total = 0
   while(n != 0)
      total += n
      n -= 1
   end
   return total
end
```



Explaining Recursion

- The function from the previous slide in plain words:
  1. You have `n`, set `total` to 0
  2. If `n` is not 0 yet:
     - (a) Add `n` to `total`
     - (b) Decrement `n` by 1
     - (c) Repeat Step 2
  3. Done, return `total`

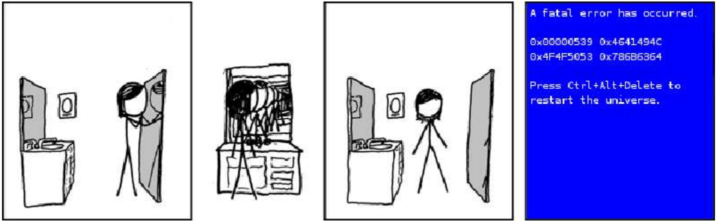
- Explaining Step 2 in recursive fashion:
  2. If `n` is not 0 yet, repeat this same step with
     - (a) `total + n` as new value for `total`
     - (b) `n-1` as new value for `n`

Recursive Sum Example/1

- How would this look like in Ruby?

```
def sum(n,total)
   if n != 0
      sum(n-1,total+n)
   end
end
```

- Something is still missing ...

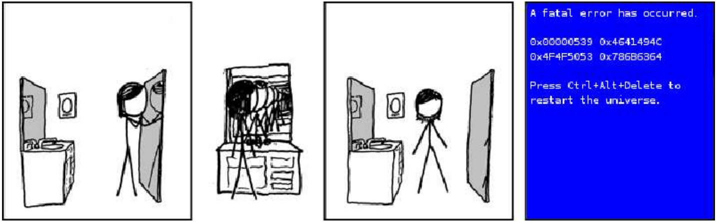


Recursive Sum Example/1

- How would this look like in Ruby?

```
def sum(n,total)
   if n != 0
      sum(n-1,total+n)
   end
end
```

- Something is still missing ... when and how do we stop?



Recursive Sum Example/2

- When `n` has reached 0, we return `total`:

```
def sum(n,total)
  if n != 0
    sum(n-1,total+n)
  else
    total
  end
end
```

- This isn't exactly the same as the iterative version

Recursive Sum Example/3

- To obtain the same signature as in the iterative version, we need a second function that initializes `total = 0`
    - Be careful that this cannot be done in the recursive function!

```
def do_sum(n)
   sum(n,0)
end


def sum(n,total)
   if n != 0
      sum(n-1,total+n)
   else
      total
   end
end
```

Tail Recursion

- A very important sub-class of recursive functions are tail recursive functions
- This means, there is nothing left to do when the recursive call returns
- The example on the previous slide is tail recursive
- Why are these functions so important?

Execution of Tail-Recursive Function

```
def sum(n,total)
   if n != 0
      sum(n-1,total+n)
   else
      total
   end
end
```

Recursive call	n	total	n != 0
sum(3,0)	3	0	yes
sum(2,3)	2	3	yes
sum(1,5)	1	5	yes
sum(0,6)	0	6	no
→ 6			
→ 6			
→ 6			
→ 6			

Non-Tail Recursion

- We implemented `sum` as a tail recursive function
- It could have also been implemented in a non-tail recursive way:

```
def sum(n)
   if n != 0
      sum(n-1) + n
   else
      0
   end
end
```

- After returning from the recursive call we still have to add `n`

Execution of Non-Tail-Recursive Function

Recursive call	n	n != 0
sum(3)	3	yes
sum(2) + 3	2	yes
sum(1) + 2	1	yes
sum(0) + 1	0	no
↪ 0 + 1		
→ 1		
↪ 1 + 2		
→ 3		
↪ 3 + 3		
→ 6		
→ 6		

```
def sum(n)
   if n != 0
      sum(n-1) + n
   else
      0
   end
end
```

Tail Recursion vs. Non-Tail Recursion/1

- Non-tail recursive function calls put parameters on the stack
  - Every call grows the stack
  - On return, the parameters are needed to compute the result (together with the partial result returned)
- In tail recursive functions, the parameters from the call before are not needed anymore
  - Instead, the result is directly handed to the parent
  - Hence, no parameters need to be put on the stack
- Hence, the main difference is
  - Tail recursion composes the (partial) result before the recursive call
  - Non-tail recursion composes the (partial) result after the recursive call
- Languages that use tail recursion optimization realize this and don't grow the stack
- The languages we cover next are optimized in this way
  - So they are much more efficient when using tail recursion

Tail Recursion vs. Non-Tail Recursion/2

- So we always use tail recursion and everything is fine?
- Unfortunately, it is not that simple:
    - Not every recursive function can be formulated in a tail recursive way
    - Non-tail recursive functions are usually easier to write: they store everything on the stack
    - Tail recursive functions have to track information and collect partial results in accumulator parameters, e.g. `total` in the `sum` function
- If a recursive function "loops" forever, it has to be tail recursive for obvious reasons

Outline

1. Recursion

2. **Writing Recursive Functions**

Writing Recursive Functions

- If you have no experience with recursive functions, writing them may seem difficult, but there are a few tricks
- Let's have a look at a concrete example: reversing an array
- First of all, it helps to look at examples

```
[]        -> []
[1]       -> [1]
[1,2]     -> [2,1]
[1,2,3]   -> [3,2,1]
```

- This will help you get a "feel" for the problem
- You may even be able to recognize some pattern

Base Cases

- Next, try to figure out the base cases
    - These are the cases that don't need a recursive call (e.g., empty list)

```
def rev(a)
   if a.length == 0 or a.length == 1
      return a
   else
      puts "not implemented yet"
   end
end
```

- You can already test this function by calling it with different parameters

```
rev([])                  -> []
rev([1])                 -> [1]
rev(['abc'])             -> ["abc"]
rev([1,2,3])             -> not implemented yet
rev([[1,2,3]])           -> [[1,2,3]]
```

Recursive Cases/1

- Now, you have to consider the recursive case, which is a bit more difficult
- What do we have?
  - We know there are at least two elements in the array (and possibly some rest)
  - We have to add a recursive call to `rev` somewhere
- Why not imagine you already have a working version?
- Summing up, we have
  - first two elements: `a[0]` and `a[1]`
  - the rest: `a.drop(2)`
    (`drop(n)` drops the first `n` elements, here 2)
  - a working function: `old_rev`

Recursive Cases/2

- How do we put this together?

```
def rev(a)
   if a.length == 0 or a.length == 1
      return a
   else
      old_rev(a.drop(2)).push(a[1]).push(a[0])
   end
end
```

- Basically, we reverse the rest of the array ...
- ... and append the first two elements in reverse order

Recursive Cases/3

- This should work now
- But if it works, then it is as good as `old_rev`
    - So you can replace `old_rev` with a recursive call `rev` and you're done!

```
def rev(a)
   if a.length == 0 or a.length == 1
      return a
   else
      rev(a.drop(2)).push(a[1]).push(a[0])
   end
end
```

- Well, we're not quite done yet . . .
    - We have to check that the recursion stops
    - We may be able to simplify the function

Termination/1

- Termination is crucial in recursive functions
- For simple functions it may be easy to see it won't get stuck in an endless loop
- For more complicated ones, you can check that its arguments are monotonically decreasing/increasing
    - and will eventually reach one of your base cases

Termination/2

- The function `rev` terminates
  - We keep dropping items from the array, making it smaller and smaller
  - Eventually it will contain only one or no item, i.e., base case
- However, checking the function `sum` we have overlooked a case
  - What happens if we call it with a negative number?

Termination of the Sum Function

- To make the sum function always terminate, we have to check for negative numbers
- Let's change the condition to $n > 0$

```
def sum(n,total)
   if n > 0
      sum(n-1,total+n)
   else
      return total
   end
end
```

- Alternatively, we could check for negative numbers in the initialization function `do_sum`

```
def do_sum(n)
   return sum(n,0) if n > 0
   0
end
```

Simplification/1

- If you have multiple base cases, check if you actually need all of them
- If we can handle empty arrays, do we need arrays with one element as a base case?
    - The case with one element can be rewritten into: `[1] -> rev([]).push(1)`
    - So we only need the empty array as base case

Simplification/2

- The simplified function looks like this:

```
def rev(a)
   return a if a == []
   rev(a.drop(1)).push(a[0])
end
```

- Was not that difficult, was it?

Just One More Flaw/1

- We now have a recursive function that reverses an array
- However, it is not tail recursive
    - We append an element to the return value
- Can you make it tail recursive?

Just One More Flaw/2

- We need a second parameter, which keeps the paritally reversed array (partial result) when going down the recursive calls

```
def rev(a,b)
   return b if a == []
   rev(a.drop(1), [a[0]] + b)
end
```

Summary

- Recursion is just a different kind of loop, but as expressive as loops
- Some programming languages are haevily based on recursion, others do not offer recursion at all
- Three important steps in writing recursive programs
  - Base cases
  - Recursive cases
  - Termination
- Often recursion allows you to write elegant code
- With the right language, it is even efficient
- Tail recursion is important to make recursive programs efficient
  - They essentialy don't need to store any data on the stack