Recursion

Dr. Greg Lavender
Department of Computer Sciences
University of Texas at Austin
lavender@cs.utexas.edu
Types of Recursion

- Recursive statements (also called self-referential)
- Recursively (inductively) defined sets
- Recursively defined functions and their algorithms
- Recursively defined data structures and recursive algorithms defined on those data structures
- Recursion vs Iteration
Recursive Statements

- In order to understand recursion, one must first understand recursion.
- This sentence contains thirty-eight letters.
- GNU = GNU’s Not Unix!

The Ouroboros is an ancient symbol implying self-reference or a “vicious circle.”
Paradoxical Statements

This is not a pipe

The Barber Paradox

A barber shaves all and only those men who do not shave themselves

if the barber does not shave himself, he must shave himself

if the barber does shave himself, he cannot shave himself

The Treachery of Images (1928-29), by René Magritte
Challenge Problem

Just as there are self-referential statements in English, you can write a self-reproducing program in a programming language, which is called a “quine” after the Harvard logician W. V. O. Quine.

A quine is a program that accepts no input and outputs an exact syntactic replica of itself.

As a corollary to a famous theorem called Kleene’s (second) Recursion Theorem, or the “Fixed Point” Theorem, there exist many such programs. Try writing one in your favorite language!

```c
int main() { printf("int main() { printf( ...```
Exploring the Boundaries of Our Ability to Understand
Notions of Truth

Propositions:
Statements that can be either True or False

Truth: \( \Rightarrow \)
Are there well formed propositional formulas (i.e., Statements) that return True when their input is True

<table>
<thead>
<tr>
<th>Clausal Form</th>
<th>Conventional Syntax</th>
<th>PROLOG</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ p }</td>
<td>p</td>
<td>p.</td>
</tr>
<tr>
<td>{ q }</td>
<td>q</td>
<td>q.</td>
</tr>
<tr>
<td>{ r }</td>
<td>r</td>
<td>r.</td>
</tr>
<tr>
<td>{ s, \neg p, \neg q }</td>
<td>p \land q \rightarrow s</td>
<td>s:-p,q.</td>
</tr>
<tr>
<td>{ t, \neg s, \neg r }</td>
<td>s \land r \rightarrow t</td>
<td>t:-s,r.</td>
</tr>
<tr>
<td>{ \neg t }</td>
<td>\neg t \rightarrow \bot</td>
<td>?-t.</td>
</tr>
</tbody>
</table>

\( (\lambda p \rightarrow \neg p) \)
\( (\lambda p \ q \rightarrow (p \land q) \lor (\neg p \land \neg q)) \)
\( (\lambda p \rightarrow \neg p \rightarrow q) \)
\( (\lambda p \ q \rightarrow (\neg p \land q) \land (\neg p \rightarrow q)) \)

If it was never possible for it not to be True that something was going to exist, and it will never be possible for it not to be True that something existed in the past then it is impossible for Truth ever to have had a beginning or ever to have an end. That is, it was never possible that Truth cannot be conceived not to exist.

If R is something that can be conceived not to exist and T is something that cannot be conceived not to exist and T is greater than R and God is that, than which nothing greater can be conceived, then God exists and is Truth.
There are Predicate Logic Statements that are True that can’t be proved True (Incompleteness) and/or there are Predicate Logic Statements that can be proved True that are actually False (Inconsistent Axioms or Unsound inference rules).

i.e., If Gödel's statement is true, then it is a example of something that is true for which there is no proof. If Gödel's statement is false, then it has a proof and that proof proves the false Gödel statement true.
There are things that are True that cannot be proved to be True. But what judges them to be True. “That then which nothing greater can be thought” judges them to be True. There are things that are False that can be proved True. But what judges them to be False. “That then which nothing greater can be thought” judges them to be False. [This is my twist on Gödel's Incompleteness Theorem, Anslem and Gödel below. Maybe one of you will be able to show that there is something else that can judge these things True and False and then manifest it in a programming language. This would be worth an A++.]

Anselm (1033-1109):


Gödel (1906-1978):

Gödel's ontological proof is a formalization of Saint Anselm’s ontological argument for God's existence by the mathematician Kurt Gödel. St. Anselm's ontological argument, in its most succinct form, is as follows: "God, by definition, is that than which a greater cannot be thought. God exists in the understanding. If God exists in the understanding, we could imagine Him to be greater by existing in reality. Therefore, God must exist." A more elaborate version was given by Gottfried Leibniz; this is the version that Gödel studied and attempted to clarify with his ontological argument. The first version of the ontological proof in Gödel's papers is dated "around 1941". Gödel is not known to have told anyone about his work on the proof until 1970, when he thought he was dying. In February, he allowed Dana Scott to copy out a version of the proof, which circulated privately. In August 1970, Gödel told Oskar Morgenstern that he was "satisfied" with the proof, but Morgenstern recorded in his diary entry for 29 August 1970 that Gödel would not publish because he was afraid that others might think "that he actually believes in God, whereas he is only engaged in a logical investigation (that is, in showing that such a proof with classical assumptions (completeness, etc.) correspondingly axiomatized, is possible)." Gödel died in 1978. Another version, slightly different from Scott's, was found in his papers. It was finally published, together with Scott's version, in 1987.
Dr. Philip Cannata

Anselm starts with the psalmist’s fool (Psalms 14 and 53 - Fools say in their hearts, "There is no God.‘‘ - i.e., an atheist). God (“that then which nothing greater can be thought”) does not exist in the fool’s mind. But then if the fool is convinced to truly think of “that then which nothing greater can be thought” (i.e., examine it with the mind’s eye) it then exists in the fool’s mind. But, did it just begin to exist when it entered the fool’s mind? That’s not possible because “that then which nothing greater can be thought” cannot have a beginning. Therefore, it had to have existed before it entered the fool’s mind.

Gödel is possibly the quintessential psalmist’s fool who, as an atheist, truly inspected “that then which nothing greater can be thought” with his mind’s eye and found it to exist at least in his mind and in his logic.

Immanuel Kant, in his Critique of Pure Reason. claims that ontological arguments are vitiated by their reliance upon the implicit assumption that “existence” is a predicate. However, as Bertrand Russell observed, it is much easier to be persuaded that ontological arguments are no good than it is to say exactly what is wrong with them. This helps to explain why ontological arguments have fascinated philosophers for almost a thousand years.

Critique of Pure Reason - If we regard the sum of the cognition of pure speculative reason as an edifice, the idea of which, at least, exists in the human mind, it may be said that we have in the Transcendental Doctrine of Elements examined the materials and determined to what edifice these belong, and what its height and stability. We have found, indeed, that, although we had purposed to build for ourselves a tower which should reach to Heaven, the supply of materials sufficed merely for a habitation, which was spacious enough for all terrestrial purposes, and high enough to enable us to survey the level plain of experience, but that the bold undertaking designed necessarily failed for want of materials—not to mention the confusion of tongues, which gave rise to endless disputes among the labourers on the plan of the edifice, and at last scattered them over all the world, each to erect a separate building for himself, according to his own plans and his own inclinations. Our present task relates not to the materials, but to the plan of an edifice; and, as we have had sufficient warning not to venture blindly upon a design which may be found to transcend our natural powers, while, at the same time, we cannot give up the intention of erecting a secure abode for the mind, we must proportion our design to the material which is presented to us, and which is, at the same time, sufficient for all our wants. [ Immanuel Kant ]

Dr. Philip Cannata
Recursion is Definitely Odd!

Or is it Even?!

odd n = 
  if (n == 0) then
    False
  else
    even (n-1)

even n = 
  if (n == 0) then
    True
  else
    odd (n-1)
Beginners often fail to appreciate that a recursion must have a conditional statement or conditional expression that checks for the “bottom-out” condition of the recursion and terminates the recursive descent.

We call the bottom-out condition the “base case” of the recursion.

If you fail to do this properly, you end up lost in Recursion Land and you never return!
Classic Recursive Functions

- Euclid’s Greatest Common Divisor (GCD) function
- Factorial function
- Fibonacci function
GCD Function

The GCD of a pair of integers \((x, y)\) is defined by taking the remainder \(r\) of \((\text{abs } x)\) divided by \((\text{abs } y)\). If \(r\) is 0, return \(x\). Otherwise compute GCD of \(y\) and \(r\).

\[
gcd :: (\text{Int, Int}) \rightarrow \text{Int}
gcd (x, y) = gcd' (\text{abs } x) (\text{abs } y)
\]

where

\[
gcd' \ x \ 0 = x
\]

\[
gcd' \ x \ y = gcd' \ y \ (x \mod \ y)
\]

\[
gcd (-98, 16) = gcd' \ 16 \ (98 \mod \ 16)
\]

\[
= gcd' \ 16 \ 2
\]

\[
= gcd' \ 2 \ (16 \mod \ 2)
\]

\[
= gcd' \ 2 \ 0
\]

\[
= 2
\]
Factorial Function

A recursive factorial algorithm implementing the function $n!$ first counts down from $n$ to 0 by recursively descending to the bottom-out condition, then performs $n$ multiplications as the recursion ascends back up.

$0! = 1$

$n! = n \times (n-1)!$ for all $n > 0$

\[
\text{fact} :: \text{Integer} \rightarrow \text{Integer} \\
\text{fact} \ n \mid n == 0 = 1 \quad -- \text{base case terminates recursion} \\
\mid n > 0 \quad = n \times \text{fact} \ (n-1) \\
\mid \text{otherwise} = \text{error} \ "\text{fact: negative value for n}\"
\]

\[
\text{fact} \ 3 \Rightarrow 3 \times \text{fact}(2) \Rightarrow 3 \times (2 \times \text{fact}(1)) \Rightarrow \\
3 \times (2 \times (1 \times \text{fact}(0))) \Rightarrow 3 \times (2 \times (1 \times 1)) \Rightarrow \\
3 \times (2 \times 1) \Rightarrow 3 \times 2 \Rightarrow 6
\]
Fibonacci Function

- The Fibonacci numbers are the infinite integer sequence 0,1,1,2,3,5,8,13,21,..., in which each item is formed by adding the previous two, starting with 0 and 1. E.g., 0+1→1, 1+1→2, 1+2→3, 2+3→5

- The Fibonacci function is defined recursively as:

  ```haskell
  fib :: Integer -> Integer
  fib 0 = 0
  fib 1 = 1
  fib n = fib(n-2) + fib(n-1)
  ```

- Notice that in computing fib n, we do two recursive calls and the sum up their results. Furthermore, the two calls are duplicative in the sense that computing fib(n-1) necessarily computes fib(n-2) all over again! Using this kind of “double” recursion is terribly inefficient.
Recursive List Data Type

A list is a recursively defined data type with elements of some type `a`, e.g., [1,2,3] is a list of type `int`.

[] constructs the empty list; `:` is an infix right associative list constructor operator (cons), that constructs a new list from an element of type `a` on the left and a list `[a]` on the right.

data [a] = [] | a : [a]

3:[] = [3]; 2:[3] = [2,3]; 1:[2,3] = [1,2,3] = 1:2:3:[]

let head [a1,a2,...,an] = a1; tail [a1,a2,...,an] = [a2,...,an]
head [] = error; tail [] = error

let (x:xs) match [a1,a2,...,an], then x = a1, xs = [a2,...,an]

let `++` be list concatenation: [1,2] ++ [3,4] = [1,2,3,4]
Recursive List Functions

length :: [a] -> Int
length [] = 0  -- empty list is the base case
length (x:xs) = 1 + length xs

sum :: (Num a) => [a] -> Int
sum [] = 0  -- empty list is the base case
sum (x:xs) = x + sum xs

mean :: (Num a) => [a] -> Float
mean lst = sum lst / length lst

-- Note: mean requires 2 traversals of the list!
-- Can we compute the mean using just one traversal?
-- let (x,y) be an ordered pair, then
-- fst (x,y) = x; snd (x,y) = y

sumlen :: (Num a) => [a] -> (Int,Int) -> (Int,Int)
sumlen [] = p
sumlen (x:xs) p = sumlen xs (x + fst(p), 1 + snd(p))

mean lst = fst(p) / snd(p) where p = sumlen lst (0,0)
Branching Recursion

- “Divide & Conquer” strategy
  - split a problem into two or more sub-problems; solve each sub-problem recursively; then combine the sub-results to obtain the final answer

- Classic examples
  - binary tree traversal
  - sorting a list of numbers
Branching Recursion

Binary Tree

Preorder Traversal: [1, 2, 3, 4, 5, 6, 7]
Inorder Traversal: [3, 2, 4, 1, 6, 5, 7]
Postorder Traversal: [3, 4, 2, 6, 7, 5, 1]
Recursive Binary Tree Traversal Algorithms

```haskell
data BinTree a = Leaf a | Root a (BinTree a) (BinTree a)

preorder :: BinTree a -> [a]
preorder (Leaf v) = [v]
preorder (Root v l r) = [v] ++ preorder l ++ preorder r

inorder :: BinTree a -> [a]
inorder (Leaf v) = [v]
inorder (Root v l r) = inorder l ++ [v] ++ inorder r

postorder :: BinTree a -> [a]
postorder (Leaf v) = [v]
postorder (Root v l r) = postorder l ++ postorder r ++ [v]
```
Simple Recursive Sorting

Given a list of values of type `a` on which there is an ordering relation defined, permute the elements of the list so that they are ordered in either ascending or descending order.

Example: Given the input list `[16, -99, 25, 71, 9, 3, 28]`, sort it into ascending order.

Step 1: select the first element of the list as a “pivot”

Step 2: partition the list into two sublists “left” and “right” where `left = [x | x <= pivot]` and `right = [y | y > pivot]`

Step 3: Recursively sort the left sublist and prepend that result to the singleton list `[pivot]`, and recursively sort the right sublist and append the result to the `left++pivot` list.
Recursive Sorting Example

sort [16, -99, 25, 71, 9, 3, 28]

sort [-99, 9, 3] ++ [16] ++ sort [25, 71, 28]

(sort [] ++ [-99] ++ [9, 3]) ++ [16] ++ sort [25, 71, 28]

(sort [] ++ [-99] ++ sort [9, 3]) ++ [16] ++ sort [25, 71, 28]


[-99, 3, 9] ++ [16] ++ sort [25, 71, 28]

[-99, 3, 9] ++ [16] ++ (sort [] ++ [25] ++ sort [28, 71])


[-99, 3, 9] ++ [16] ++ [25, 28, 71]

[-99, 3, 9, 16, 25, 28, 71]
Recursive Sort Algorithm

Polymorphic recursive sorting function that sorts a list of elements of type `a`, where `a` is required to be ordered (i.e., has the relational operators `==`, `<`, `>`, `<=` and `>=` defined)

sort :: (Ord a) => [a] -> [a]

sort [] = [] -- base case
sort (x:[]) = [x] -- singleton list
sort (pivot:rest) = sort left ++ [pivot] ++ sort right
where
  left = [x | x <- rest, x <= pivot]
  right = [y | y <- rest, y > pivot]
Recursion vs Iteration

- Iterative factorial requires O(1) stack space
- update n and m in place within one stack activation frame each time through the loop

```c
int ifact (int n) {
    int m = 1;
    while (n > 1) { m = m * n; n = n-1; }
    return m;
}
```

Frame #1:
- n=3
- m=1

Frame #1:
- n=3-1
- m=1*3

Frame #1:
- n=2-1
- m=3*2
Recursion vs Iteration

Recursive factorial requires $O(n)$ stack space

“Count down” from $n$ and delay doing any multiplications until the recursion bottoms out, then do the multiplications on the way “back up”

```c
int rfact (int n)
{
    return (n == 0 || n == 1) ? 1 : n * fact(n-1);
}
```

Frame #1

<table>
<thead>
<tr>
<th>Frame #1</th>
<th>n = 3</th>
<th>delay 3 * fact(2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame #2</td>
<td>n = 2</td>
<td>delay 2 * fact(1)</td>
</tr>
<tr>
<td>Frame #3</td>
<td>n = 1</td>
<td>recursion bottoms out at 1</td>
</tr>
</tbody>
</table>
Tail Recursion

“Recursion is the root of computation since it trades description for time”
– Alan Perlis

The benefits of an elegant recursive description, but equivalent in space and time to an iteration

requires $O(1)$ stack space instead of $O(n)$ stack space
Tail Recursive Factorial

Written recursively, but only requires \( O(1) \) stack space like an iteration. We just need to invoke \( \text{tfact} \) with \( m = 1 \) and compute the expression \( m \times n \) before the recursive call. The compiler can then do “tail call” optimization and turn the recursion into an iteration automatically.

```c
int tfact (int n, int m=1)
{
    return (n == 0 || n == 1) ? m : tfact(n-1, m*n);
}
```
No Tail Call Optimization

Use “g++ -S tfact.cc”. Note the “call” instruction

```c
_tfact:  pushl  %ebp
         movl  %esp, %ebp
         subl  $40, %esp
         cmpl  $1, 8(%ebp)
         je     L14
         cmpl  $0, 8(%ebp)
         jne   L16

L14:   movl   12(%ebp), %eax
       movl   %eax, -12(%ebp)
       jmp    L17

L16:   movl   12(%ebp), %eax
       imull  8(%ebp), %eax
       movl   8(%ebp), %edx
       subl   $1, %edx
       movl   %eax, 4(%esp)
       movl   %edx, (%esp)
       call   _tfact
       movl   %eax, -12(%ebp)

L17:   movl   -12(%ebp), %eax
       leave
       ret
```
Tail Call Optimization

Recompile tfact using “g++ -O2 -S tfact.cc” to verify that a loop is generated using one stack frame, not call insn using $O(n)$ stack frames

```
_tfact:  pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    $1, %edx
    jbe     L22
    L26:    imull   %edx, %eax
    subl    $1, %edx
    cmpl    $1, %edx
    ja      L22
    L22:    popl    %ebp
    ret
```
Inductive Sets

The set of Sponges is the smallest set satisfying the following rules, known as the Sponge Bob Axioms:

- ‘Bob’ is a Sponge
- If ‘s’ is a Sponge, then the successor of ‘s’ is a Sponge.
- Bob is not the successor of any Sponge
- Induction Axiom: For all sets S, if Bob is in S and for every Sponge s in S, the successor of s is in S, then every Sponge is in S

A recursively defined abstract data type that captures this inductive set:

```haskell
data Sponge = Bob | Succ Sponge
```
Peano Arithmetic

Using the Sponge Bob Axioms, we can define arithmetic on the Natural Numbers, but let’s equate the data type Sponge to Nat and Bob to Zero. We then define “Peano Arithmetic,” named after Guiseppe Peano (1858-1932), who defined the set Nat inductively using such axioms (called Peano’s Axioms of course)

-- inductive “Nat” data type
-- Ex: Zero, Succ(Zero), Succ(Succ(Zero)) ...
-- i.e., counting up: 0, 1, 2, ...

data Nat = Zero | Succ Nat

-- boolean function to test for the base case
iszero :: Nat -> Bool
iszero Zero = True
iszero (Succ n) = False
Basic Arithmetic Functions

Arithmetic can then be defined recursively in terms of counting up (successor) and counting down (predecessor)

succ, pred :: Nat -> Nat  -- unary functions
succ n = Succ n    -- count up by prepending Succ to n
pred (Succ n) = n  -- count down by removing a Succ from n
pred Zero = error "no predecessor of Zero"

add, mult :: (Nat, Nat) -> Nat  -- binary functions
add (n, Zero) = n
add (Zero, m) = m
add (n, m) = succ(add(n, pred m)) -- succ of n, m times

mult (n, Zero) = Zero
mult (Zero, m) = Zero
mult (n, m) = add(n, mult(n, pred m)) -- succ of n, n+m times
Examples

A Peano Arithmetic calculator:

PA> pred Zero
*** Exception: no predecessor of Zero

PA> pred (Succ Zero)
Zero

PA> succ Zero
Succ (Zero)

PA> add(Succ Zero, Succ (Succ (Succ Zero)))
Succ (Succ (Succ (Succ (Zero)))))

PA> add(Succ Zero, Succ (Succ (Succ (Succ (Succ Zero))))))
Succ (Succ (Succ (Succ (Succ (Succ (Zero)))))))

PA> mult(Succ (Succ Zero), Succ (Succ (Succ (Succ (Succ Zero))))))
Succ (Succ (Succ (Succ (Succ (Succ (Succ (Succ (Succ (Zero))))))))))
Induction vs Recursion

For beginners, induction is intuitive, but recursion is often counter-intuitive

Induction is like “ascending”

  e.g., counting up: 1, 2, 3, ...

Recursion is like “descending”

  e.g., counting down: n, n-1, n-2, ...

But they often go hand-in-hand to solve a problem
Show that the closed formula \( n(n+1)/2 = \text{sum}(n) \) for all \( n > 0 \), where \( n \) is a positive integer and \( \text{sum}(n) = 1 + 2 + \ldots + n \)

**Base case:** evaluate \( n(n+1)/2 \) for \( n = 1 \)

\[
1(1+1)/2 = 2/2 = 1 = \text{sum}(1)
\]

**Inductive hypothesis:** assume \( k(k+1)/2 = \text{sum}(k) \), for all \( i \) where \( 1 \leq i \leq k \)

**Inductive step:** show \( (k+1)((k+1)+1)/2 = \text{sum}(k) + 1 \)

\[
(k+1)(k+2)/2 = (k+1)/2 + k/2 + 1 = k(k+1)/2 + 1 = \text{sum}(k) + 1
\]

**Conclusion:** \( n(n+1)/2 = \text{sum}(n) \) for all \( n \geq 1 \)
A Simpler Algebraic Proof

Due to Carl Friedrich Gauss (1777-1855)

he was told to sum the first 100 positive integers at a young age while in a class on arithmetic

Gauss combined counting up with counting down

\[
(1 + 2 + 3 + \ldots + n) + (n + n-1 + n-2 + \ldots + 1)
= (1+n) + (2+n-1) + (3+n-2)\ldots + (n+1)
= n+1 + n+1 + n+1 + \ldots + n+1
= n \times (n+1)
= 2 \times \text{sum}(n)
\]

Therefore, \( \text{sum}(n) = n \times (n+1)/2 \) for all \( n \geq 1 \)

Ex: \( \text{sum}(100) = (100 \times 101)/2 = 50 \times 101 = 5050 \)
Summing Up vs Summing Down

- A well-ordered ascending sequence
  \[ \text{isum}(n) = 1 + 2 + 3 + \ldots + n \]

- A well-ordered descending sequence
  \[ \text{rsum}(n) = n + n-1 + n-2 + \ldots + 1 \]

Both isum and rsum compute the same value for a given n, but isum does so in O(1) stack space while rsum requires O(n) stack space.

```c
// inductive sum (count up)
int isum(int n)
{
    int sum = 0;
    for (int i=1; i<=n; ++i)
        sum += i;
    return sum;
}

// recursive sum (count down)
int rsum(int n)
{
    assert(n > 0);
    if (n == 1)
        return 1;
    else
        return n + rsum(n-1);
}
```
Recommended Reading

Recursion Theory and Logic:


Recursive Programming:

- “Recursive Functions of Symbolic Expressions and their Computation by Machine (Part I),” by John McCarthy (see his website)
- Thinking Recursively, 2nd Ed., by Eric Roberts
- Recursive Programming Techniques, by W. H. Burge
- Programming in Haskell, by Graham Hutton
Other Fun Reading

- Godel, Escher, Bach, by Douglas Hofstadter
- Books by Raymond Smullyan
  - What is the Name of this Book?
  - To Mock a Mockingbird
  - Forever Undecided
  - Recursion Theory for Metamathematics (advanced)
Life Itself is Recursive, So Self-Reflect On It!

© M. C. Escher