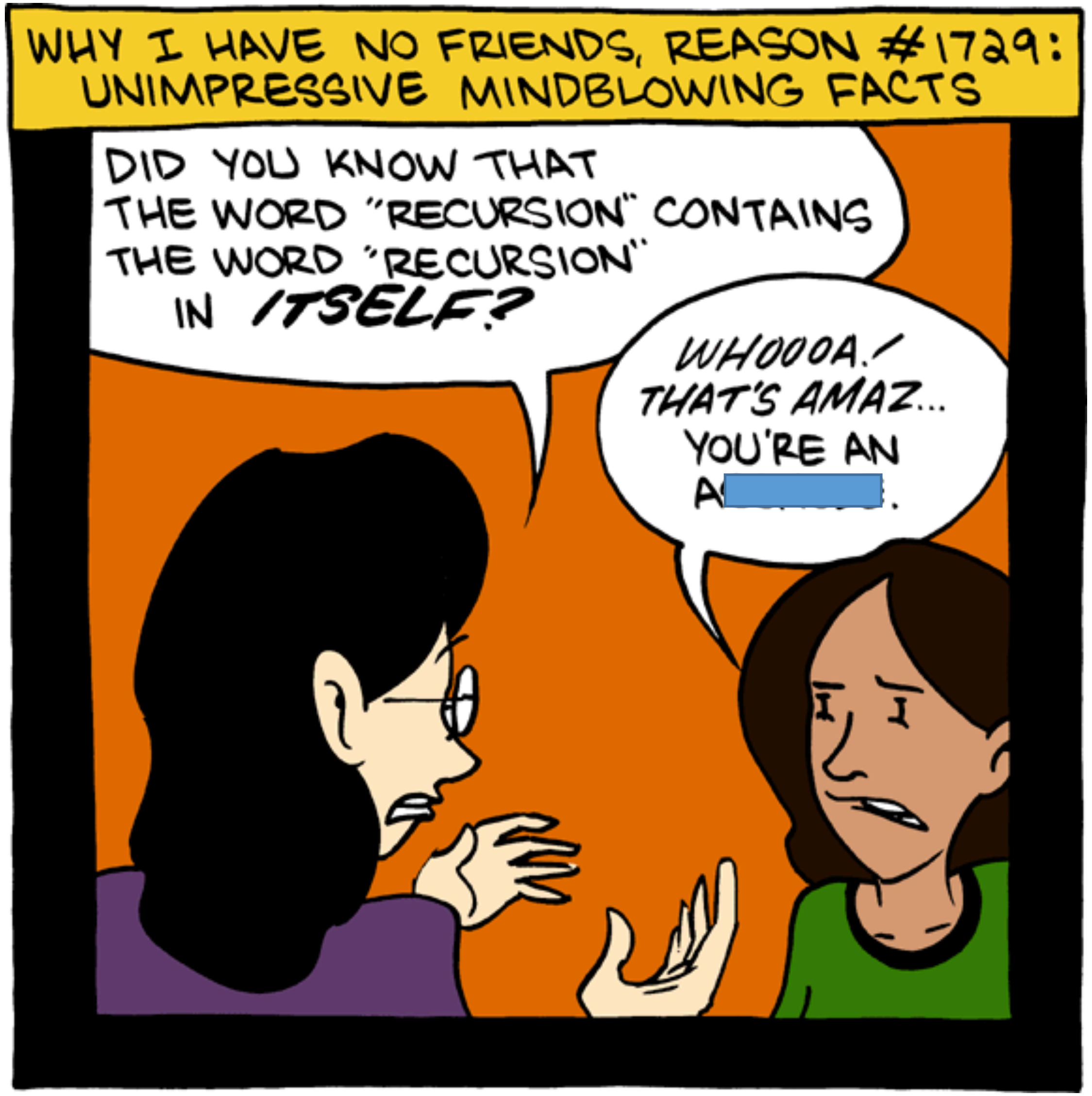
Recursion

What is recursion?

- In order to understand recursion, you first need to understand recursion
  - Whoops, need a base case!
- From the Google dictionary:
  "the repeated application of a recursive procedure or definition."
- An example from Linux — what does GNU stand for?
  - GNU's Not Unix
- In math and computer science, recursion is the use of a function in defining the values the function returns
  - $Factorial(n) = \begin{cases} nFactorial(n-1) & \text{when } n \geq 1 \\ 1 & \text{when } n = 0 \end{cases}$



https://www.smbc-comics.com/comic/2011-08-02

Exercise

- Write an inductive definition for the Fibonacci sequence
  - $Fib(n) = \begin{cases} Fib(n-1) + Fib(n-2) & \text{when } n > 1 \\ 1 & \text{when } n \leq 1 \end{cases}$
- Write an inductive definition for multiplication over the naturals
  - $Mult(n, m) = \begin{cases} n + Mult(n, m-1) & \text{when } m > 0 \\ 0 & \text{when } m = 0 \end{cases}$
- Write an inductive definition for addition over the naturals using the successor function (*succ*)
  - $Add(n, m) = \begin{cases} succ(Add(n, m-1)) & \text{when } m > 0 \\ n & \text{when } m = 0 \end{cases}$

Booleans

Definition of the booleans — see core.lean, line 253

Naturals

An inductive definition of the natural numbers — see core.lean, line 293

Definition

```
inductive mynat : Type
| zero : mynat
| succ : mynat → mynat
```

- Base case: mynat.zero
- Inductive case: mynat.succ

```
def zero := mynat.zero
def one := mynat.succ zero
def two := mynat.succ one
…
```

- `#reduce two -- mynat.succ (mynat.succ mynat.zero)`

Predecessor function

```
def pred (n : mynat) :=
  match n with
    -- define pred mynat.zero to be zero
    | mynat.zero := zero
    -- define pred (succ n’) to be n’
    | mynat.succ n' := n'
  end
#reduce pred three -- mynat.succ (mynat.succ mynat.zero)
```

- Compare with definition of mynat:

```
inductive mynat : Type
| zero : mynat
| succ : mynat → mynat
```

- Two cases in definition of mynat → two cases in definition of pred (typically)

Addition function

```
def add_mynat: mynat → mynat → mynat
-- base case
| mynat.zero m := m
-- recursive case: invokes add_mynat
| (mynat.succ n') m :=
     mynat.succ (add_mynat n’ m)
-- (n' + 1) + m = (n' + m) + 1
-- or, n + m = (n – 1 + m) + 1
-- stops when n gets to base case (zero)

#reduce add_mynat three two
```

Exercises

1. We just implemented addition as the recursive (iterated) application of the successor function. Now you are to implement multiplication as iterated addition.

```
def mul_mynat: mynat → mynat → mynat
| mynat.zero m := zero
| (mynat.succ n') m := add_mynat m (mul_mynat n' m)
```

2. Implement exponentiation as iterated multiplication.

```
def exp_mynat: mynat → mynat → mynat
| m mynat.zero := one
| m (mynat.succ n') := mul_mynat m (exp_mynat m n')
```

3. Take this pattern one step further. What function did you implement? How would you write it in regular math notation?

```
def tet_mynat: mynat → mynat → mynat
| m mynat.zero := one
| m (mynat.succ n') := exp_mynat m (tet_mynat m n')
```

Left identity for adding zero

- For our naturals, we can prove 0 + m = m

```
theorem zero_left_id:
  ∀ m : mynat,
    add_mynat mynat.zero m = m
:=
begin
intro m,
apply rfl,
-- simp [add_mynat],
end
```

Proof by induction

- How do we prove ∀`m`:`P(m)` is true?

1. Prove that `P(m)` is true for the *base case* (e.g., `P(0)`).
2. Prove that if `P(m)` is true, then `P(succ(m))` is true — e.g., `P(m)` → `P(m+1)`

- Why is this proof valid?

Right identity for adding zero

- We can also prove that m + 0 = m, but we need to use induction and simp (or some other similar tactic)

```
theorem zero_right_id :
  ∀ m : mynat,
    add_mynat m mynat.zero = m
:=
begin
  intro m,
  induction m with m’ h,
    -- base case
    apply rfl,
    -- inductive case
    simp [add_mynat],
    assumption,
end
```

Another induction example

- Prove n + (m + 1) = (n + m) + 1

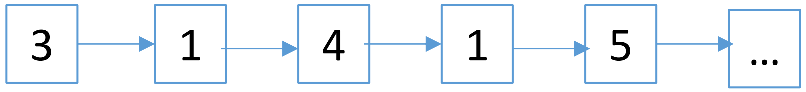
```
lemma add_n_succ_m :
  ∀ n m : mynat,
    add_mynat n (mynat.succ m) =
      mynat.succ (add_mynat n m) :=
begin
  intros n m,
  induction n with n' h,
    apply rfl,  -- base case
    simp [add_mynat], -- inductive case
    assumption,
end
```

Yet another induction example

```
example :
  ∀ m n : mynat,
    add_mynat m n = add_mynat n m :=
begin
  intros m n,
  -- by induction on m
  induction m with m’ h,
    --base case: m = zero
    simp [add_mynat],
    rw zero_right_id,
    -- inductive case: if true for m then true for succ m
    simp [add_mynat],
    rw add_n_succ_m,
    -- rewrite using induction hypothesis!
    rw h,
end
```

Linked lists

- A linked list is a structure of ”nodes” that contain a value and a “pointer” to their “next” node



- In some languages (*cough* C++), you would have a structure like this:

```
struct Node {
  int data;
  struct Node *next;
}
```

Linked lists in Lean

- In Lean, we can define a linked list of integers as an inductive type
- The base case is an empty list
- The inductive case is a node that takes an integer and a reference to the next node

Fin