Lecture 4

Numbers

Peano arithmetic

1. 0 is a natural number.
2. For every natural number x, x = x. That is, equality is **reflexive**.
3. For all natural numbers x and y, if x = y, then y = x. That is, equality is **symmetric**.
4. For all natural numbers x, y and z, if x = y and y = z, then x = z. That is, equality is **transitive**.
5. For all a and b, if b is a natural number and a = b, then a is also a natural number. That is, the natural numbers are **closed** under equality.

Peano arithmetic

6. The naturals are assumed to be closed under a single-valued **successor** function S. For every natural number n, S(n) is a natural number.
7. For all natural numbers m and n, m = n if and only if S(m) = S(n). That is, S is an **injection**.
8. For every natural number n, S(n) = 0 is false. That is, there is no natural number whose successor is 0.

Addition

- Defined as:

$$x+0 = x \quad \text{for all } x \in \mathbb{N}$$

$$x+1 = S(x) \quad \text{for all } x \in \mathbb{N}$$

$$x+S(y)=S(x+y) \quad \text{for all } x, y \in \mathbb{N}$$

- can then be proven to be both associative and commutative.

Multiplication

- Defined as:

  x*1=x for all $x \in \mathbb{N}$ and $x \neq 0$

  x*S(y)=x*y+x for all $x, y \in \mathbb{N}$ and $x, y \neq 0$

- can also be proven to be both associative and commutative.
- It can also be shown to be distributive over addition.

Mathematical Induction

- Mathematical Induction is a method of proof normally used to prove that a proposition is true for all $\mathbb{N}$.
- Principle of Mathematical Induction consists of successfully carrying out the following two steps:

1. **Base Case**: Prove that P(1) is true.
2. **Induction Step**: Assume that P(n) is true for an arbitrary n, then prove that P (n+1) is true.

Structural Induction-Also known as

- Structural induction is used to show results about **recursively** defined sets.
- *Basis Step*: Prove that the statement holds for all elements specified in the basis step of the set definition.
- *Recursive Step*: Prove that if the statement is true for each of the elements used to construct elements in the recursive step of the set definition, then the result holds for these new elements.
- Induction = Recursion often

Recursion in Rust-Which solution is better?

```
fn fac(n: u128) -> u128 {
    if n > 1 {
        n * fac(n - 1)
    } else {
        n
    }
}
```

A

```
fn fac(n: u128) -> u128 {
    //!
    fn fac_with_acc(n: u128, acc: u128) -> u128 {
        if n > 1 {
            fac_with_acc(n - 1, acc * n)
        } else {
            1
        }
    }
    fac_with_acc(n, 1)
}
```

B

Recursion vs. Iteration

```
pub fn fibonacci(n: u64) -> u64 {
 fn fibonacci_lr(n: u64, a: u64, b: u64) -> u64
 {
  match n {
    0 => a,
     _ => fibonacci_lr(n - 1, a + b, a),
    }
   }
  fibonacci_lr(n, 1, 0)
}
```

A

```
pub fn fibonacci(mut n: u64) -> u64 {
    let (mut a, mut b) = (1, 0);
    while n > 0 {
        n -= 1;
        a = a + b;
        b = a - b;
    }
    a
}
```

B

Compiler Explorer – Rust vs. LLVM IR

```
pub fn square(num:i32) -> i32{
    num * num
}
```

C

```
define i32 @square(i32)
local_unnamed_addr #0 {
    %2 = mul nsw i32 %0, %0
    ret i32 %2
}
```

D

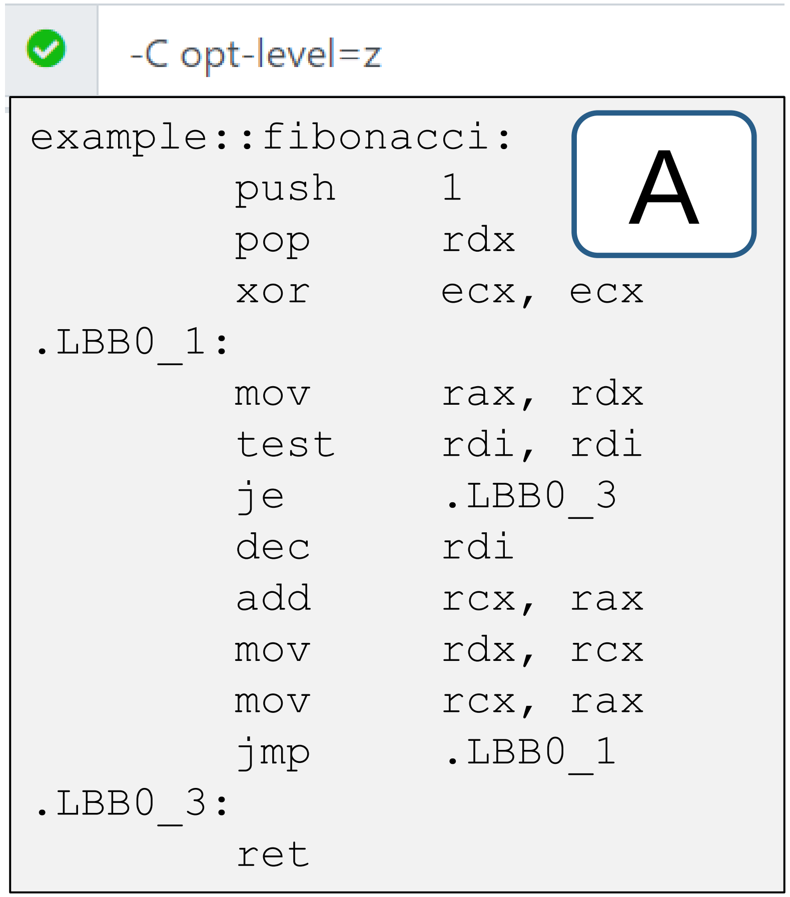
```
example::square:
        mov     eax, edi
        imul    eax, edi
        ret
```

C

```
square:
# @square
        mov     eax, edi
        imul    eax, edi
        ret
```

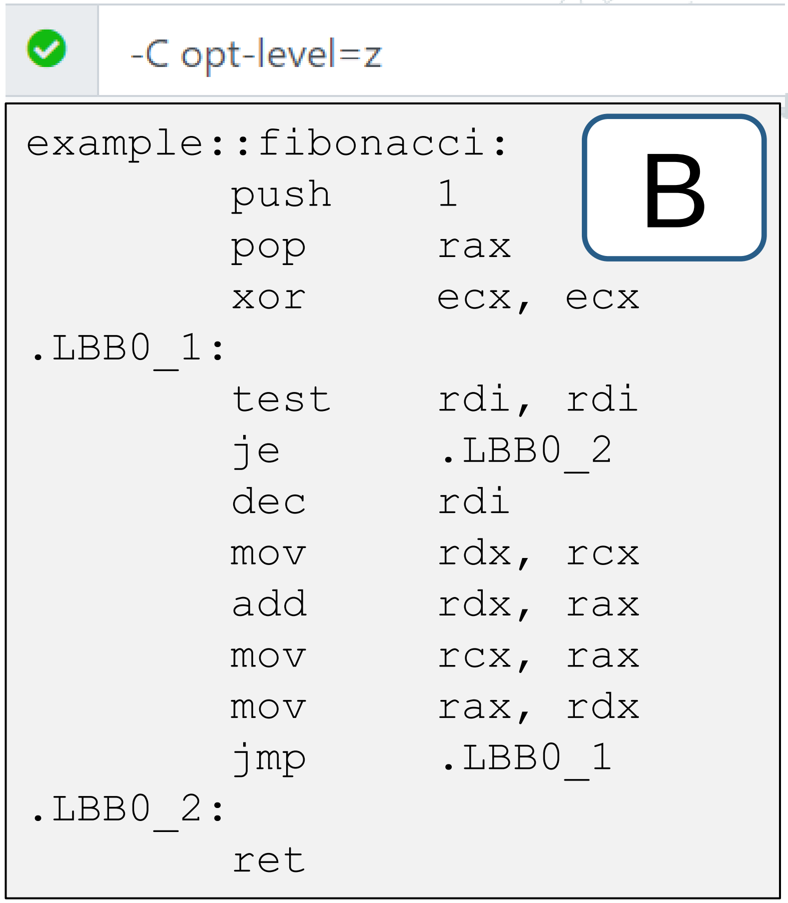
D

Recursion vs. Iteration – Compiler Explorer



-C opt-level=z

```
example::fibonacci:
        push    1
        pop     rdx
        xor     ecx, ecx
.LBB0_1:
        mov     rax, rdx
        test    rdi, rdi
        je      .LBB0_3
        dec     rdi
        add     rcx, rax
        mov     rdx, rcx
        mov     rcx, rax
        jmp     .LBB0_1
.LBB0_3:
        ret
```



-C opt-level=z

```
example::fibonacci:
        push    1
        pop     rax
        xor     ecx, ecx
.LBB0_1:
        test    rdi, rdi
        je      .LBB0_2
        dec     rdi
        mov     rdx, rcx
        add     rdx, rax
        mov     rcx, rax
        mov     rax, rdx
        jmp     .LBB0_1
.LBB0_2:
        ret
```

Macro!

- A macro can call itself, like a function recursion:

```
macro_rules! sum {
    ($base:expr) => { $base };
    ($a:expr, $($rest:expr),+) => { $a + sum!($($rest),+) };
}
```

- Let's go though the expansion of sum!(1, 2, 3):

```
sum!(1, 2, 3)
//      ^  ^~~~
//     $a  $rest
=> 1 + sum!(2, 3)
//          ^  ^
//         $a  $rest
=> 1 + (2 + sum!(3))
//               ^
//               $base
=> 1 + (2 + (3))
```