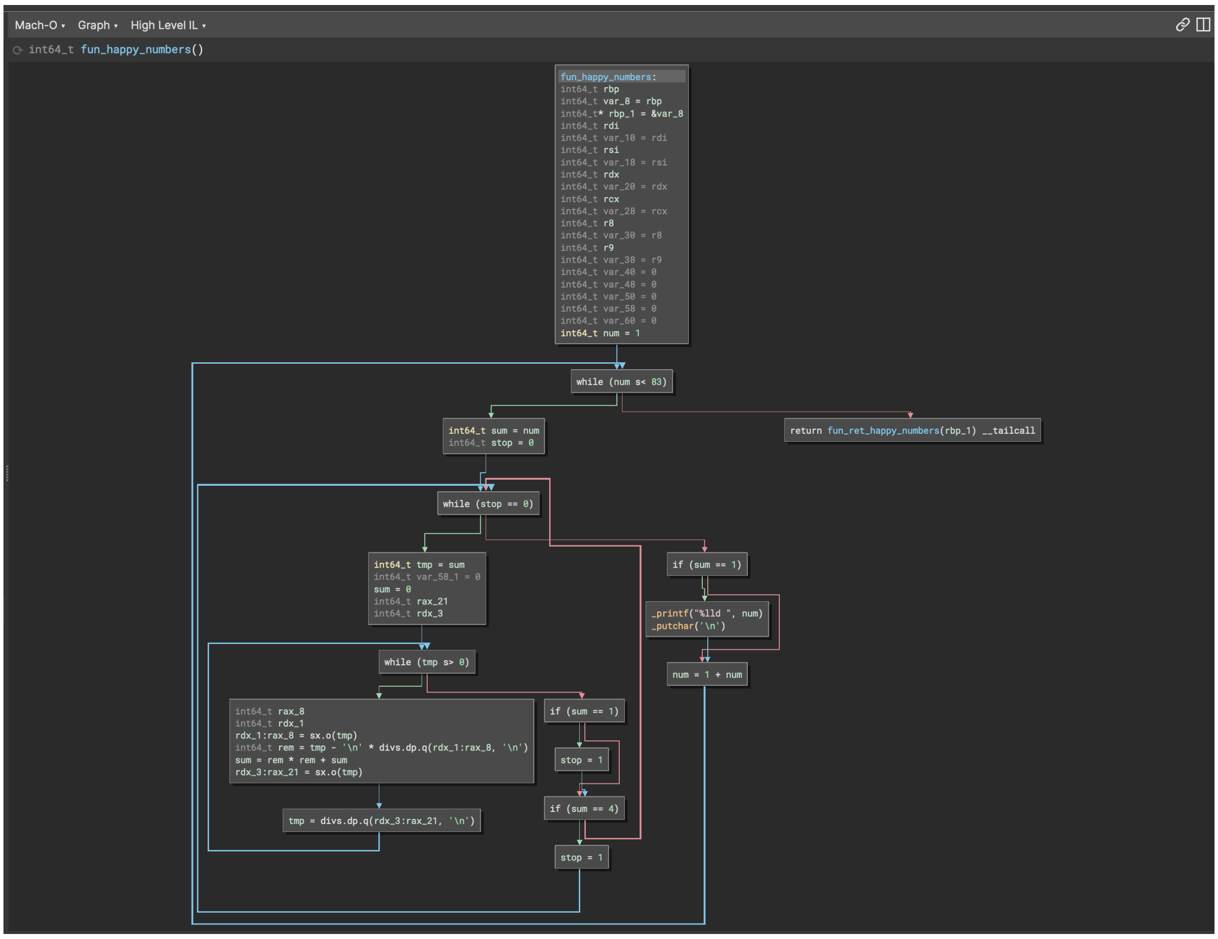
TDT4205 - Ex 6

Jonas Elvedal Hole

Data flow analysis



```
int64_t fun_happy_numbers()
```

```
fun_happy_numbers:
int64_t rbp
int64_t var_8 = rbp
int64_t* rbp_1 = &var_8
int64_t rdi
int64_t var_10 = rdi
int64_t rsi
int64_t var_18 = rsi
int64_t rdx
int64_t var_20 = rdx
int64_t rcx
int64_t var_28 = rcx
int64_t r8
int64_t var_30 = r8
int64_t r9
int64_t var_38 = r9
int64_t var_40 = 0
int64_t var_48 = 0
int64_t var_50 = 0
int64_t var_58 = 0
int64_t var_60 = 0
int64_t num = 1
```

```
while (num s< 83)
```

```
int64_t sum = num
int64_t stop = 0
```

```
return fun_ret_happy_numbers(rbp_1) __tailcall
```

```
while (stop == 0)
```

```
int64_t tmp = sum
int64_t var_58_1 = 0
sum = 0
int64_t rax_21
int64_t rdx_3
```

```
if (sum == 1)
```

```
_printf("%lld ", num)
_putchar('\n')
```

```
num = 1 + num
```

```
while (tmp s> 0)
```

```
int64_t rax_8
int64_t rdx_1
rdx_1:rax_8 = sx.o(tmp)
int64_t rem = tmp - '\n' * divs.dp.q(rdx_1:rax_8, '\n')
sum = rem * rem + sum
rdx_3:rax_21 = sx.o(tmp)
```

```
tmp = divs.dp.q(rdx_3:rax_21, '\n')
```

```
if (sum == 1)
```

```
stop = 1
```

```
if (sum == 4)
```

```
stop = 1
```

I simply decompiled the program and added variable names to produce this graph of execution

Code generation

For a challenge I redid the compiler i Rust, but to save time I started with the .symbol files generated by the previous iteration of the C compiler. In the end i the code ended up pretty clean, and i got `while` and `if` working.

It could be even cleaner but I didn't have time to fix everything. This ended up taking way more time than I expected.

I have added snippets of the code so it's easier to grade since you're not familiar with my codebase at all. The main parts are just the struct impls, but there is also some basic logic to construct those at the bottom of `src/ast/types.rs`.

While

```rust
#[derive(Debug, Clone)]
pub struct WhileStatement {
    condition: Relation,
    statement: Block,
}

impl WhileStatement {
    pub fn compile<W: Write>(&self, function: &Function, out: &mut W) {
        emit!("").compile(out);
        emit!("// while statement").compile(out);

        let nonce = rand::random::<u16>();

        let inverse_instruction = match self.condition.operator {
            '>' => "jle",
            '<' => "jge",
            '=' => "jne",
            '!' => "je",
            _ => todo!("unknown operator {}", self.condition.operator),
        };

        label!("WHILE_START{nonce}").compile(out);

        // should end up with a cmp
        self.condition.compile(function, out);

        // jump to end if condition failed
        emit!("{inverse_instruction} WHILE_DONE{nonce}").compile(out);

        // otherwise, run the body
        self.statement.compile(function, out);

        // then jump up again
        jmp!("WHILE_START{nonce}").compile(out);

        // and when done, continue:
        label!("WHILE_DONE{nonce}").compile(out);
    }
}
```

If

```
#[derive(Debug, Clone)]
pub struct IfStatement {
    condition: Relation,
    statement: Box<Statement>,
    else_statement: Option<Box<Statement>>,
}

impl IfStatement {
    fn compile<W: Write>(&self, function: &Function, out: &mut W) {
        emit!("").compile(out);
        emit!("// if statement").compile(out);

        let nonce = rand::random::<u16>();

        self.condition.compile(function, out);

        let inverse_instruction = match self.condition.operator {
            '>' => "jle",
            '<' => "jge",
            '=' => "jne",
            '!' => "je",
            _ => todo!("unknown operator {}", self.condition.operator),
        };

        // jump to else, if condition failed (or end if there is no else)
        let fail_label = match &self.else_statement {
            Some(_) => format!("IF_ELSE{}", nonce),
            None => format!("IF_END{}", nonce),
        };
        emit!("{inverse_instruction} {fail_label}").compile(out);

        // otherwise run body
        self.statement.compile(function, out);
        jmp!("IF_END{nonce}").compile(out);

        // compile else if it exists
        if let Some(else_statement) = &self.else_statement {
            label!("IF_ELSE{nonce}").compile(out);
            else_statement.compile(function, out);
        }

        // end label
        label!("IF_END{nonce}").compile(out);
    }
}
```

Break

Didn't do this in time. Would be pretty trivial to use some global state to do it, but I don't want to mess up the codebase too much and it wouldn't be too hard to keep track of parents of nodes, but again, no time.