```
[[questions]]
type = "MultipleChoice"
prompt.prompt = """
Let's consider this function that performs the division between two `Option<u32>`.
Why do we want it to return a `Result` instead of an `Option` ?
```
fn option_div (lhs: Option<u32>, rhs: Option<u32>) -> Result<u32, ByteArray> {
    match lhs {
        Option::Some(dividend) => {
            match rhs {
                Option::Some(divisor) => {
                    if divisor == 0 {
                        Result::Err("Divisor is 0")
                    } else {
                        Result::Ok(dividend / divisor)
                    }
                },
                Option::None => Result::Err("Divisor is None")
            }
        },
        Option::None => {
          Result::Err("Dividend is None")
        },
    }
}
```
"""
prompt.distractors = [
  "Because `Result` uses fewer bytes at runtime than `Option` to represent failures",
  "Because `Result` represents the possibility of failure, while `Option` cannot represent
failures",
  "Because `Result` represents errors the same way as the underlying system calls",
]
answer.answer = "Because `Result` can represent why an operation failed, the division
can fail for many reasons (e.g either at least one operand is `None` or the divisor is 0)"
context = """
`Option` can just represent that an operation has failed, but `Result` can explain why
the operation has failed.
"""
id = "f9aee0d9-6974-433d-8391-c601b9c803f5"
[[questions]]
type = "Tracing"
prompt.program = """
fn option_div (lhs: Option<u32>, rhs: Option<u32>) -> Result<u32, ByteArray> {
    match lhs {
        Option::Some(dividend) => {
```

```
            match rhs {
                Option::Some(divisor) => {
                    if divisor == 0 {
                        Result::Err("Divisor is 0")
                    } else {
                        Result::Ok(dividend / divisor)
                    }
                },
                Option::None => Result::Err("Divisor is None")
            }
        },
        Option::None => {
          Result::Err("Dividend is None")
        },
    }
}
fn try_division_by_0 () -> Option<u32> {
    let dividend = Option::Some(10);
    let divisor = Option::Some(0);
    let result = option_div(dividend, divisor)?;
    Option::Some(result)
}
fn main() {
    println!("{}", try_division_by_0().unwrap());
}
"""
answer.doesCompile = false
context = """
`option_div` returns a `Result`, but the return type of `try_division_by_0` expects an
`Option`.
Therefore it is invalid to use the `?` operator until the `Result` has been converted to an
`Option` (e.g. with the `Result::Ok` method).
"""
id = "021a3060-6076-4b40-ba4f-eb305543e449"
[[questions]]
type = "MultipleChoice"
prompt.prompt = """
Given an arbitrary expression `e` of type `Result<T, E>`, which code snippet best
represents how `e?` is translated?
"""
prompt.distractors = ["""
```
if let Result::Err(e) = e {
    return Result::Err(e);
}
```
```

"""", """
```
match e {
    Result::Ok(x) => x,
    Result::Err(err) => panic!("{}", err)
}
```
"""", "`e.unwrap()`"]
answer.answer = """
```
match e {
    Result::Ok(v) => v,
    Result::Err(e) => return Result::Err(e)
}
```
"""
context = """
If `e` is of type `Result`, then `e?` extracts the value inside `Ok` if possible; otherwise, it returns the `Err` from the current function.
"""
id = "b4ddf5d9-ee90-47b3-a183-24a7fa578669"