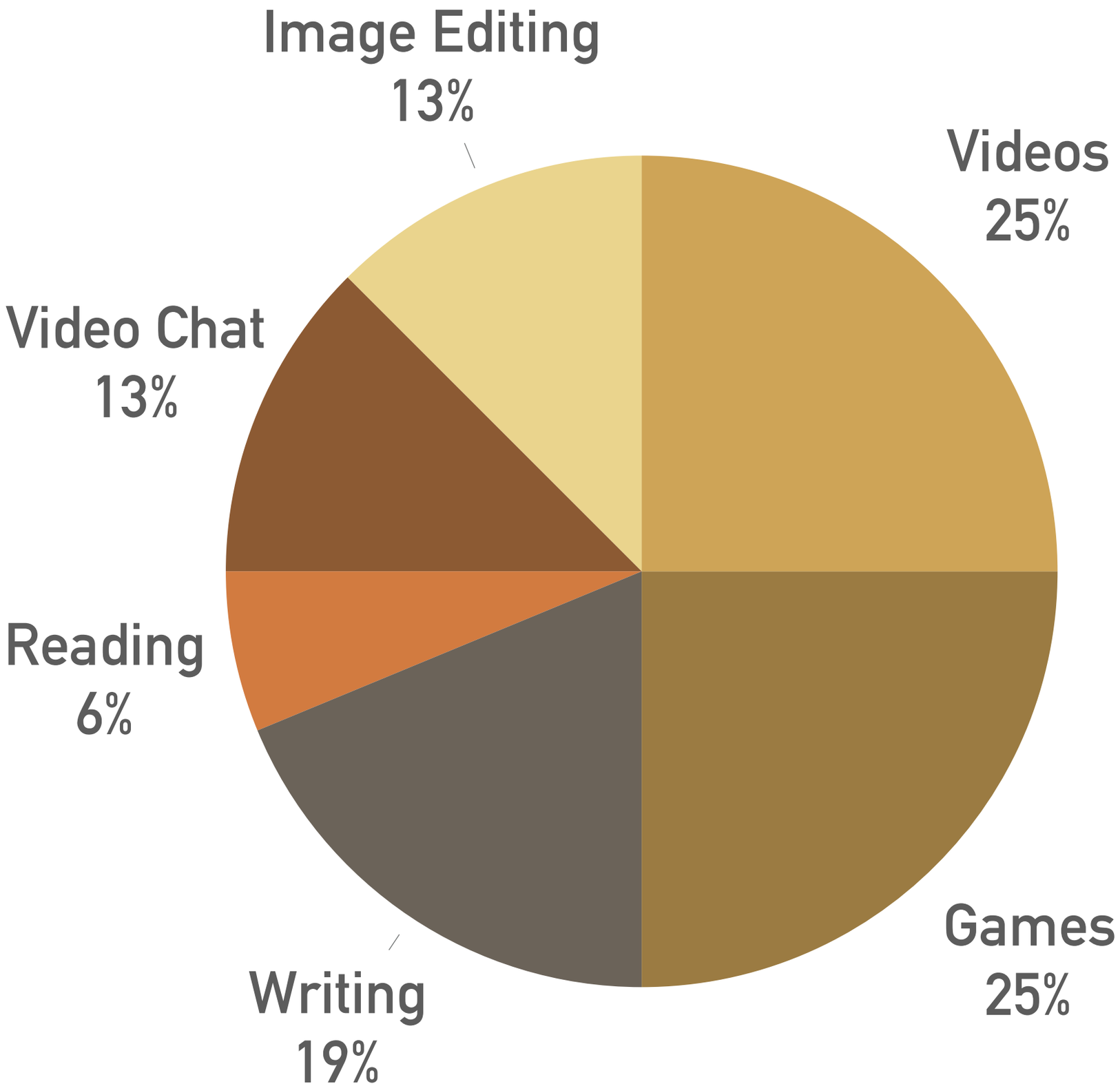
UNIT TESTS

subexpressions

COVD-19 COMMENTS

- We'll be paying attention to university guidelines, of course
- Traveling students are being repatriated, if you haven't heard yet
- If you feel bad, stay away
- GDC cancelled
- My son's trip to Italy rescheduled, so I actually have to deal with him over spring break…

WHAT ARE COMPUTERS GOOD FOR, ANYHOW?



WHAT ABOUT ME?

- Programming is life
- Can’t play games any more.
- I do a *lot* of Keynote. I wonder why.

HOMEWORK REVIEW

MULTIPLE POSSIBILITIES

- Rectangles (right ones) are interesting
- What's more interesting is how many different approaches y'all used
- Some were top-left-to-bottom-right, some were lower-left-to-top-right
- One was center with width and height
- This demonstrates the flexibility of using software contracts!

CHECKING WORK

- I'm also intrigued by how many of you felt compelled to verify your homework against an actual interpreter
- I don't personally feel that this kind of review is *needed*, but I can appreciate why you might do so

SKETCH OUT AN ANSWER ON THE WHITEBOARD

BUILD ON WHAT CAME BEFORE

- Don't define functions in terms of implementation!
- cons / car / cdr only belong in constructors and selectors, then *use* those constructors and selectors as you're implementing computations!
  - SEPERATION
  - OF
  - CONCERNS

SUB EXPRESSIONS

BEGIN/END PRIMITIVES

- One way we can approach sub-expressions is by wrapping each set of primitives inside a new type of primitive, *Begin / End*
- Our primitives become building blocks for the language

BEGIN/END

```
enum Primitive {
    Add,
    Multiply,
    Number(i32),
    Begin,
    End
}
```

A PROBLEM WITH THIS

- This ends up having some serious problems, however
- For example, the begin/end tags end up becoming part of our "fold" during addition and so on
- Secondly, how do we trigger evaluating a sub expression?

SO SIMILAR TO UNARY MINUS

- This problem is indeed similar to the unary minus problem—we are solving the problem at an incorrect time!
- In fact, our entire set of primitives seems to be incorrect!
- Let's simplify this some…

SIMPLIFIED PRIMITIVES

```
#derive[(Clone)]
#derive[(Debug)]
enum Primitive {
    Add,
    Number(i32)
}
```

DON'T EAT THE ITER

- Add shouldn't be eating the iterator as it works through the list of operands
- In fact, Add should have its *own* iterator for the operands
- It would be better if it had a vector of operands on its own

SIMPLIFIED PRIMITIVES

```
#derive[(Clone)]
#derive[(Debug)]
enum Primitive {
    Add(Vec<Primitive::Number>),
    Number(i32)
}
```

NO NEED TO LIMIT IT TO NUMBERS

- Addition shouldn’t be limited to numbers, though
- It’d be trivial to make addition work with any type of primitive

SIMPLIFIED PRIMITIVES

```
#derive[(Clone)]
#derive[(Debug)]
enum Primitive {
    Add(Vec<Primitive>),
    Number(i32)
}
```

UPDATED EVALUATION

```
fn evaluate(primitive: &Primitive) → i32 {
    match primitive {
        Primitive::Add(primitives) ⇒ {
            let iter = primitives.iter();
            iter.fold(0, |total, next| total + evaluate(next))
        },
        Primitive::Number(val) ⇒ *val,
    }
}
```

REINSTATING MULTIPLICATION

```
fn evaluate(primitive: &Primitive) → i32 {
    match primitive {
        Primitive::Add(primitives) ⇒ {
            let iter = primitives.iter();
            iter.fold(0, |total, next| total + evaluate(next))
        },
        Primitive::Multiply(primitives) ⇒ {
            let iter = primitives.iter();
            iter.fold(1, |total, next| total * evaluate(next))
        },
        Primitive::Number(val) ⇒ *val,
    }
}
```

TESTING IN MAIN()

```
fn main() {
    let mut primitives = Vec::new();
    primitives.push(Primitive::Number(3));
    primitives.push(Primitive::Number(4));
    primitives.push(Primitive::Number(5));
    let add = Primitive::Add(primitives);
    let add2 = Primitive::Multiply(vec![Primitive::Number(2), add]);
    let result = evaluate(&add2);
    println!("result was {}", result);
    let sub = Primitive::Subtract(vec![add2, Primitive::Number(2)]);
    println!("result was {}", evaluate(&sub));
}
```

MAYBE THESE AREN'T PRIMS

- Primitive no longer seems appropriate as a name
- These are in fact expressions.
  - A number is an expression.
  - The value of adding a set of numbers is the evaluation of an expression
  - Expressions are actually a good name!
- Time for a quick refactor
  - What does "refactor" mean?

REFACTOR PRIMITIVE TO EXPRESSION

```
#[derive(Clone)]
#[derive(Debug)]
enum Expression {
    Add(Vec<Expression>),
    Multiply(Vec<Expression>),
    Subtract(Vec<Expression>),
    Number(f64),
}
```

EVALUATE

```
fn evaluate(expression: &Expression) → f64 {
    match expression {
        Expression::Add(expressions) ⇒ {
            let iter = expressions.iter();
            iter.fold(0.0, |total, next| total + evaluate(next))
        },
        Expression::Multiply(expressions) ⇒ {
            let iter = expressions.iter();
            iter.fold(1.0, |total, next| total * evaluate(next))
        },
        Expression::Number(val) ⇒ *val,
    }
}
```

SUBTRACTION

```
Expression::Subtract(expressions) ⇒ {
    let mut iter = expressions.iter();
    let first = iter.next().unwrap();
    iter.fold(evaluate(first),
              |total, next|
              total - evaluate(next))
},
```

TEST VIA MAIN

```
fn main() {
    let mut expressions = Vec::new();
    expressions.push(Expression::Number(3.0));
    expressions.push(Expression::Number(4.0));
    expressions.push(Expression::Number(5.0));
    let add = Expression::Add(expressions);
    let add2 = Expression::Multiply(vec!
                                    [Expression::Number(2.5), add]);
    let result = evaluate(&add2);
    println!("result was {}", result);
    let sub = Expression::Subtract(vec![add2,
                                    Expression::Number(2.2)]);
    println!("result was {}", evaluate(&sub));
}
```

UNIT TESTING

WHAT IS A UNIT TEST?

- Unit testing is a way to write a test against one specific area of code to ensure that it works and continues to work as development continues
- Unit tests are a source of great arguments amongst programmers
- Unit tests can be used to demonstrate that parts of a program are correct

WHAT TO TEST

- A unit means different things to different people
  - In procedural languages, a "unit" is usually a single function
  - In object-oriented languages, a "unit" is usually a struct / class, though it can be a single function as well
- We write tests for the smallest meaningful units within our programs

WHY UNIT TEST

- Unit testing can help prove parts of the application are correct
- Unit tests help drive us to ensure that our code is *modular* — it is not as meaningful to test a single mega-method with many different combinations of parameters
- Unit tests help us truly define what our code is *supposed to do*; if we can't articulate a test, we don't understand the requirements

DESIGNING A GOOD TEST

- A good unit test has three components:
  - Arrange
  - Act
  - Assert
- During the *Arrange* phase, we set up whatever variables or environments are needed in order to establish the test
  - Please note that if too much state is needed, this can be a code smell!

DESIGNING A GOOD TEST

- During the *Act* phase, the method or object being tested is called to do whatever it is that it does
- During the *Assert* phase, the results of the action are interpreted
  - If asserts cannot be written, then perhaps the test is poorly formed
- *Arrange, Act, Assert*

DESIGNING A GOOD TEST

- A good test will verify a *single specific combination* of elements
- Many good tests are better than one *great* test, because one *great* test ain't a thing
- Don't confuse *unit* tests with *integration* tests

WHEN TO TEST

- Tests should be written as a program is begun, and should be maintained throughout development
- Some days, a programmer may run unit tests exclusively and never run the actual application they're working on
- Tests are best when they use some kind of automated framework, where they are run every time that a file is saved, or at least *very* often

TEST DRIVEN DEVELOPMENT

- In TDD, tests are written prior to the code that they test.
- A *software contract* is written, then a test that *intentionally fails* is written, and then that test is refined as code is written, until the tests pass and the code is well written
- A strict application of TDD implies 100% *code coverage*

CODE COVERAGE

- Code coverage is the primary means of defining how well our code is tested
- Code coverage refers to how much of our code is tested: not just functions, but within each section of branching logic
- TDD aspires to 100%
- Different parts of a program might aspire to different levels of coverage

CARGO TESTING

- Rust provides testing mechanisms via *cargo test*
- Tests can be written in the same module as the code that they're testing!

ADDING A MEANINGLESS TEST

```
#[cfg (test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2+2, 4);
    }
}
```

#[CFG(TEST)]

- There are 3 elements that make a "test" a *test*.
  - #[cfg(test)]
  - mod tests { … }
  - #[test]
    - in front of *each* test

- Use *assert!*, *assert_eq!*, and *assert_ne!*

REFACTORING FOR TESTABILITY

```
fn evaluate(expression: &Expression) → f64 {
    match expression {
        Expression::Add(expressions) ⇒ {
            let iter = expressions.iter();
            iter.fold(0.0, |total, next| total + evaluate(next))
        },
        Expression::Multiply(expressions) ⇒ {
            let iter = expressions.iter();
            iter.fold(1.0, |total, next| total * evaluate(next))
        },
        Expression::Subtract(expressions) ⇒ {
            let mut iter = expressions.iter();
            let first = iter.next().unwrap();
            iter.fold(evaluate(first), |total, next| total - evaluate(next))
        },
        Expression::Number(val) ⇒ *val,
    }
}
```

POOR TESTABILITY

- This method has poor testability, because so much logic is contained within each arm
- This code should be moved into separate functions, which can then be individually tested

ADDITION

```
pub fn evaluate_addition(add: &Expression) → f64 {
    if let Expression::Add(expressions) = add {
        let iter = expressions.iter();
        iter.fold(0.0, |total, next| total + evaluate(next))
    } else {
        panic!("Addition not provided")
    }
}
fn evaluate(expression: &Expression) → f64 {
    match expression {
        Expression::Add(_) ⇒ evaluate_addition(expression),
    }
}
```

PUBLIC FUNCTIONS

- We've made our function for evaluation *public* by using the *pub* keyword
- This will make more sense when we write multiple-module code in rust
- For now, even though the tests are in the same *module* (file) as this function, it still must be defined as public.
- The same is true for the *Expression* enum

A SINGLE UNIT TEST

```
#[test]
fn basic_addition() {
  // arrange
  let values = vec![Expression::Number(2.0),
                    Expression::Number(2.0)];
  // act
  let sum = evaluate_addition(&Expression::Add(values));
  // assert
  assert_eq!(4.0, sum);
}
```

MORE REFACTORING

```
pub fn evaluate_multiplication(mult: &Expression) → f64 {
    if let Expression::Multiply(expressions) = mult {
        let iter = expressions.iter();
        iter.fold(1.0, |total, next| total * evaluate(next))
    } else {
        panic!("Multiply not provided")
    }
}
pub fn evaluate_subtraction(sub: &Expression) → f64 {
    if let Expression::Subtract(expressions) = sub {
        let mut iter = expressions.iter();
        let first = iter.next().unwrap();
        iter.fold(evaluate(first), |total, next| total - evaluate(next))
    } else {
        panic!("Subtract not provided")
    }
}
```

REFACTORED EVALUATE

```
fn evaluate(expression: &Expression) → f64 {
    match expression {
        Expression::Add(_) ⇒
                        evaluate_addition(expression),
        Expression::Multiply(_) ⇒
                    evaluate_multiplication(expression),
        Expression::Subtract(_) ⇒
                    evaluate_subtraction(expression),
        Expression::Number(val) ⇒ *val,
    }
}
```

TEST FOR SUBTRACTION

```
    #[test]
    fn basic_subtraction() {
        // arrange
        let values = vec![Expression::Number(4.0),
                          Expression::Number(2.0)];
        // act
        let remainder = evaluate_subtraction(
                        &Expression::Subtract(values));
        // assert
        assert_eq!(2.0, remainder);
}
```

TEST MULTIPLICATION

```
#[test]
fn basic_multiplication() {
    // arrange
    let values = vec![Expression::Number(2.0),
                      Expression::Number(2.0)];
    // act
    let product = evaluate_multiplication(
                      &Expression::Multiply(values));
    // assert
    assert_eq!(4.0, product);
}
```

RUNNING THE TESTS

```
 ~/projects/first-interp ▶  master ● ▶ cargo test
    Finished test [unoptimized + debuginfo] target(s) in 0.00s
     Running target/debug/deps/first_interp-b9881376c65b0e85

running 4 tests
test tests::basic_addition ... ok
test tests::basic_subtraction ... ok
test tests::basic_multiplication ... ok
test tests::it_works ... ok

test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0
filtered out
```