Hacking the Rust object system at Mozilla

Lindsey Kuper
Grinnell College
April 5, 2012

Me and how I got here

Me and how I got here

- Graduated from Grinnell (CS and music) in 2004

Me and how I got here

- Graduated from Grinnell (CS and music) in 2004
- Web development at a (failed) startup, 2004–2006

Me and how I got here

- Graduated from Grinnell (CS and music) in 2004
- Web development at a (failed) startup, 2004–2006
- Perl plumbing at a publishing company, 2006–2008

Me and how I got here

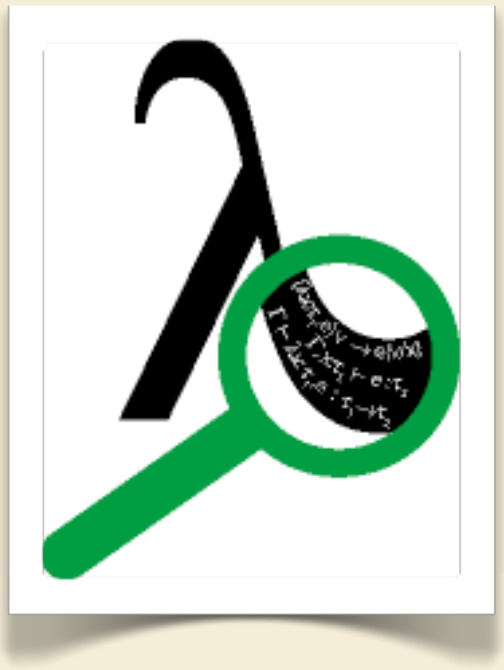
- Graduated from Grinnell (CS and music) in 2004
- Web development at a (failed) startup, 2004–2006
- Perl plumbing at a publishing company, 2006–2008
  - but in 2007, I moved in with a couple of Haskell hackers...

Me and how I got here

- Graduated from Grinnell (CS and music) in 2004
- Web development at a (failed) startup, 2004–2006
- Perl plumbing at a publishing company, 2006–2008
  - but in 2007, I moved in with a couple of Haskell hackers...
- Ph.D. student at Indiana studying PL since fall 2008

My field: programming language semantics

- Questions we might want to answer:



Source: Andrew Myers' CS 611 course at Cornell

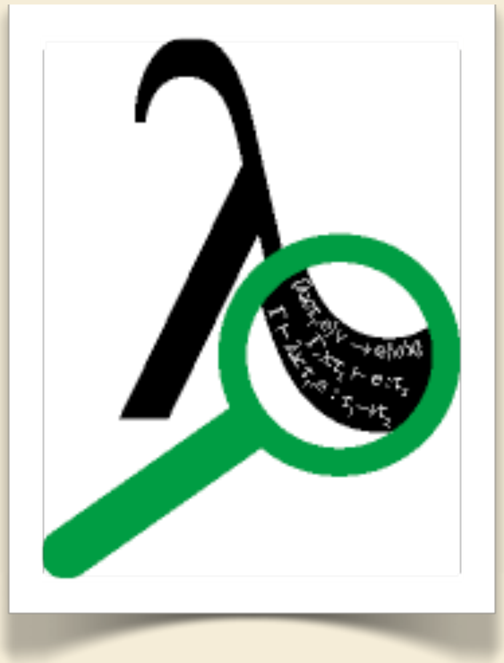
My field: programming language semantics

- Questions we might want to answer:
  - Is this program correct?

Source: Andrew Myers' CS 611 course at Cornell

My field: programming language semantics

- Questions we might want to answer:
  - Is this program correct?
  - Will this program encounter a run-time type error?



Source: Andrew Myers' CS 611 course at Cornell

My field: programming language semantics

- Questions we might want to answer:
  - Is this program correct?
  - Will this program encounter a run-time type error?
  - Is one program indistinguishable from another?

Source: Andrew Myers' CS 611 course at Cornell

My field: programming language semantics

- Questions we might want to answer:
  - Is this program correct?
  - Will this program encounter a run-time type error?
  - Is one program indistinguishable from another?
  - Is this optimization a safe program transformation?

Source: Andrew Myers' CS 611 course at Cornell

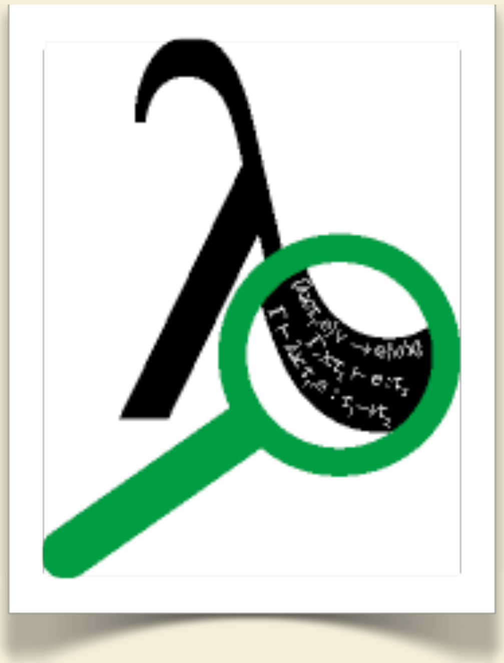
My field: programming language semantics

- Questions we might want to answer:
  - Is this program correct?
  - Will this program encounter a run-time type error?
  - Is one program indistinguishable from another?
  - Is this optimization a safe program transformation?
  - Can programs written in this language crash?

Source: Andrew Myers' CS 611 course at Cornell

My field: programming language semantics

- Questions we might want to answer:
  - Is this program correct?
  - Will this program encounter a run-time type error?
  - Is one program indistinguishable from another?
  - Is this optimization a safe program transformation?
  - Can programs written in this language crash?
  - Is this compiler translation correct?



Source: Andrew Myers' CS 611 course at Cornell

My field: programming language semantics

- Questions we might want to answer:
  - Is this program correct?
  - Will this program encounter a run-time type error?
  - Is one program indistinguishable from another?
  - Is this optimization a safe program transformation?
  - Can programs written in this language crash?
  - Is this compiler translation correct?
  - Can source language A be translated into target language B?

Source: Andrew Myers' CS 611 course at Cornell

Why Rust?

- What do we want in a programming language?

Source: Michael Sullivan, "Closures for Rust"

Why Rust?

- What do we want in a programming language?
  - Fast: generates efficient machine code

Source: Michael Sullivan, "Closures for Rust"

Why Rust?

- What do we want in a programming language?
  - Fast: generates efficient machine code
  - Safe: type system provides guarantees that prevent certain bugs

Source: Michael Sullivan, "Closures for Rust"

Why Rust?

- What do we want in a programming language?
  - Fast: generates efficient machine code
  - Safe: type system provides guarantees that prevent certain bugs
  - Concurrent: easy to build concurrent programs and to take advantage of parallelism

Source: Michael Sullivan, "Closures for Rust"

Why Rust?

- What do we want in a programming language?
  - Fast: generates efficient machine code
  - Safe: type system provides guarantees that prevent certain bugs
  - Concurrent: easy to build concurrent programs and to take advantage of parallelism
  - "Systemsy": fine-grained control, predictable performance characteristics

Source: Michael Sullivan, "Closures for Rust"

Why Rust?

- What do we have now?

Source: Michael Sullivan, "Closures for Rust"

Why Rust?

- What do we have now?
  - Firefox is in C++, which is Fast and Systemsy

Source: Michael Sullivan, "Closures for Rust"

Why Rust?

- What do we have now?
  - Firefox is in C++, which is Fast and Systemsy
  - ML is (sometimes) Fast and (very) Safe

Source: Michael Sullivan, "Closures for Rust"

Why Rust?

- What do we have now?
  - Firefox is in C++, which is Fast and Systemsy
  - ML is (sometimes) Fast and (very) Safe
  - Erlang is Safe and Concurrent

Source: Michael Sullivan, "Closures for Rust"

Why Rust?

- What do we have now?
  - Firefox is in C++, which is Fast and Systemsy
  - ML is (sometimes) Fast and (very) Safe
  - Erlang is Safe and Concurrent
  - Haskell is (sometimes) Fast, (very) Safe, and Concurrent

Source: Michael Sullivan, "Closures for Rust"

Why Rust?

- What do we have now?
  - Firefox is in C++, which is Fast and Systemsy
  - ML is (sometimes) Fast and (very) Safe
  - Erlang is Safe and Concurrent
  - Haskell is (sometimes) Fast, (very) Safe, and Concurrent
  - Java and C# are Fast and Safe

Source: Michael Sullivan, "Closures for Rust"

Why Rust?

A systems language
pursuing the trifecta:
fast, concurrent, safe

You worked on the *what* system?!

You worked on the *what* system?!

- I didn’t arrive with the intention of working on the object system, but...

You worked on the *what* system?!

- I didn’t arrive with the intention of working on the object system, but...
- I was intrigued by the idea of a classless object model and flexible prototype-based objects

You worked on the *what* system?!

- I didn’t arrive with the intention of working on the object system, but...
- I was intrigued by the idea of a classless object model and flexible prototype-based objects
    - and was told, “None of that’s implemented yet; go for it!”

You worked on the *what* system?!

- I didn’t arrive with the intention of working on the object system, but...
- I was intrigued by the idea of a classless object model and flexible prototype-based objects
  - and was told, “None of that’s implemented yet; go for it!”
- When I started: no object extension, method overriding, or self-dispatch

You worked on the *what* system?!

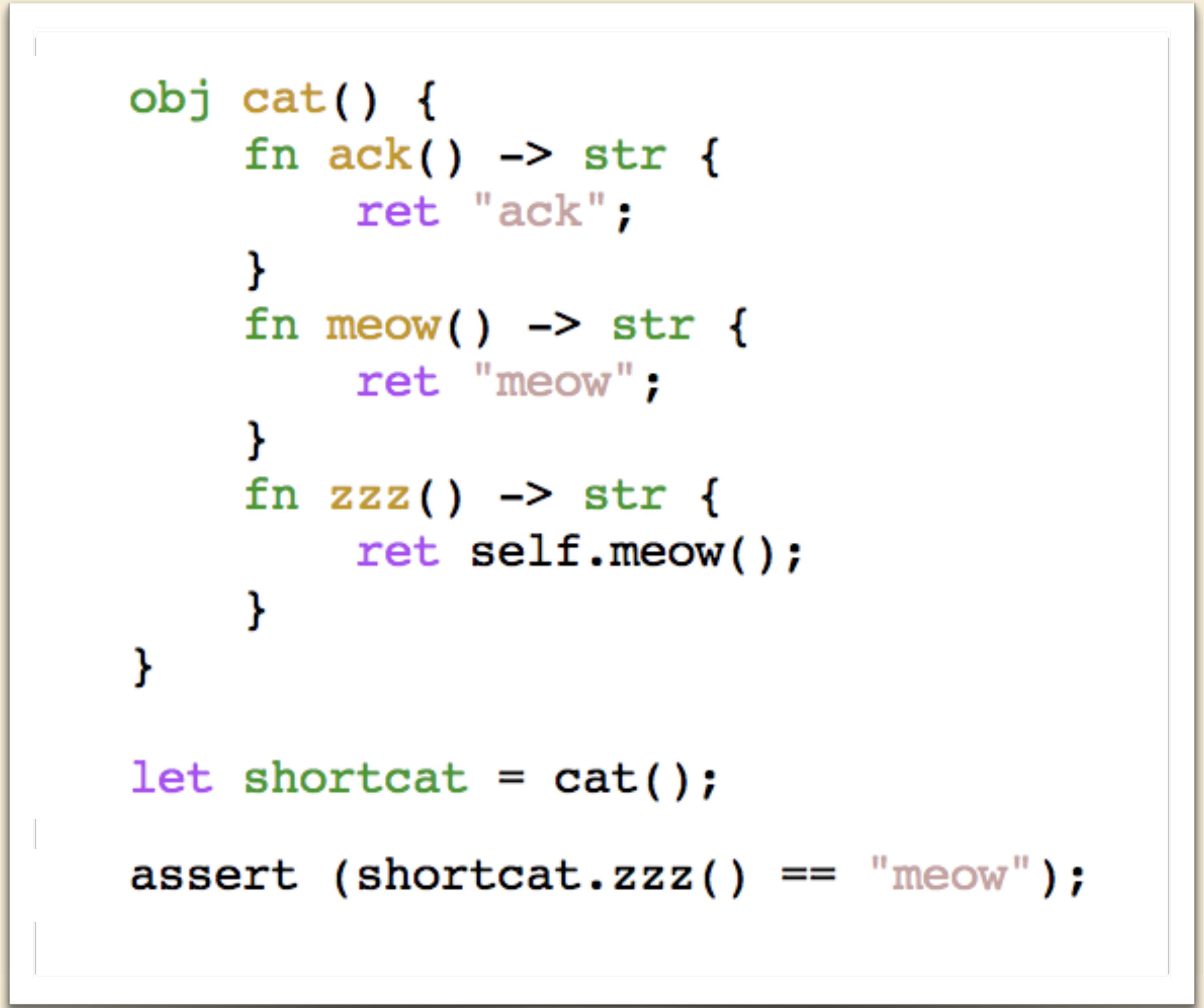
- I didn't arrive with the intention of working on the object system, but...
- I was intrigued by the idea of a classless object model and flexible prototype-based objects
  - and was told, "None of that's implemented yet; go for it!"
- When I started: no object extension, method overriding, or self-dispatch
- During my internship, I implemented those things

You worked on the *what* system?!

- I didn't arrive with the intention of working on the object system, but...
- I was intrigued by the idea of a classless object model and flexible prototype-based objects
    - and was told, "None of that's implemented yet; go for it!"
- When I started: no object extension, method overriding, or self-dispatch
- During my internship, I implemented those things
    - and learned that they interact with each other in interesting ways

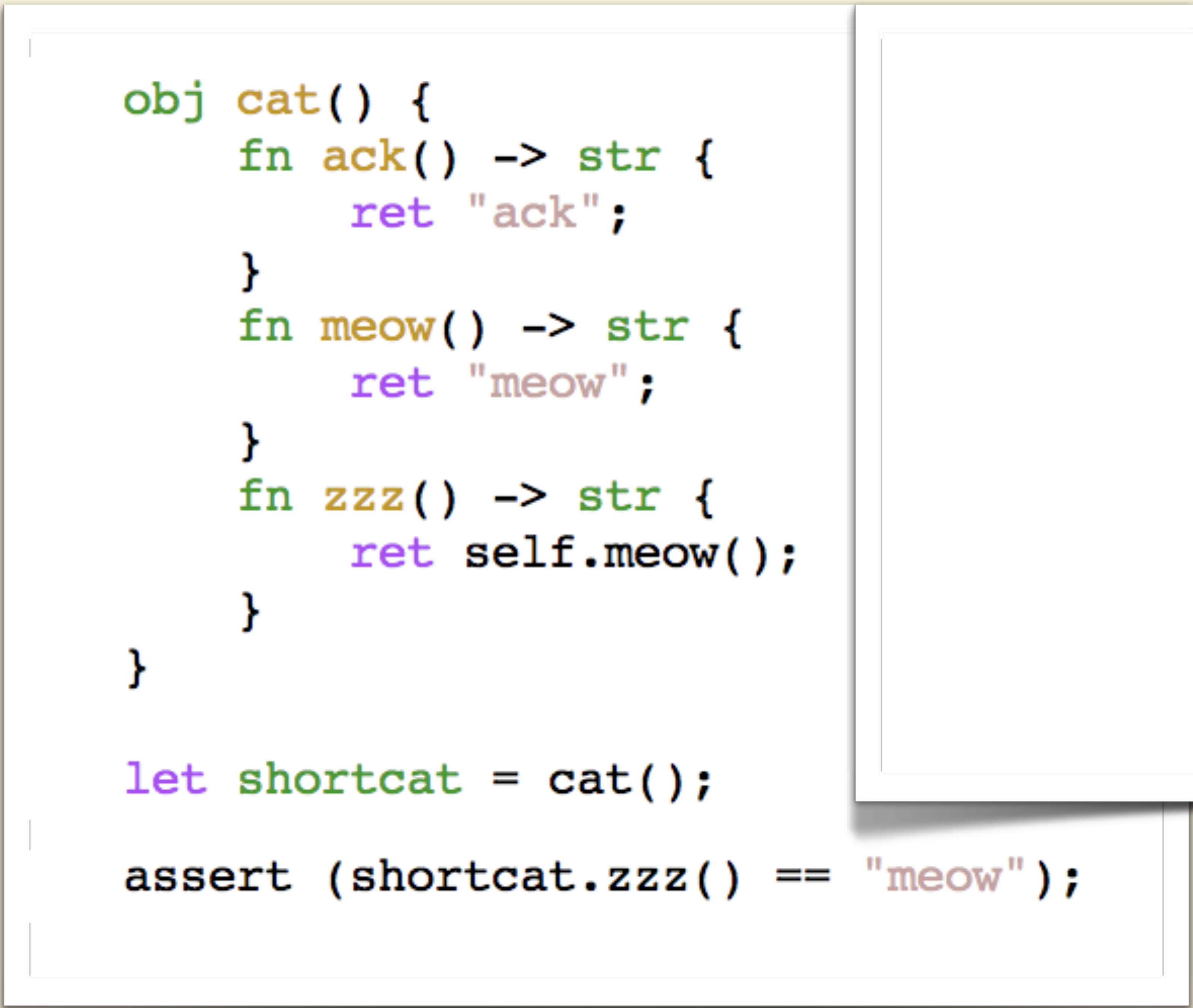
Self-dispatch

Self-dispatch



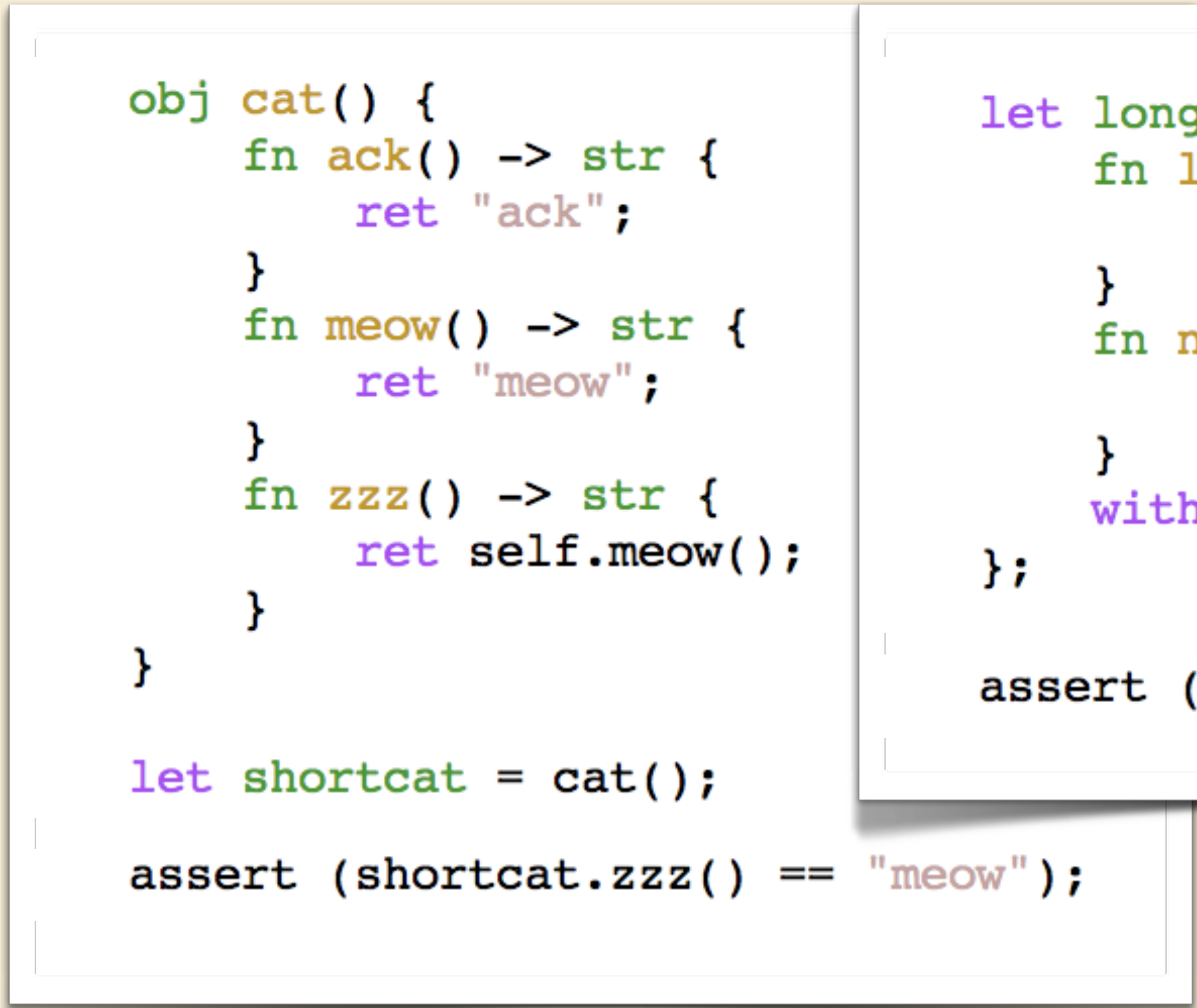
```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();

assert (shortcat.zzz() == "meow");
```

Self-dispatch + object extension

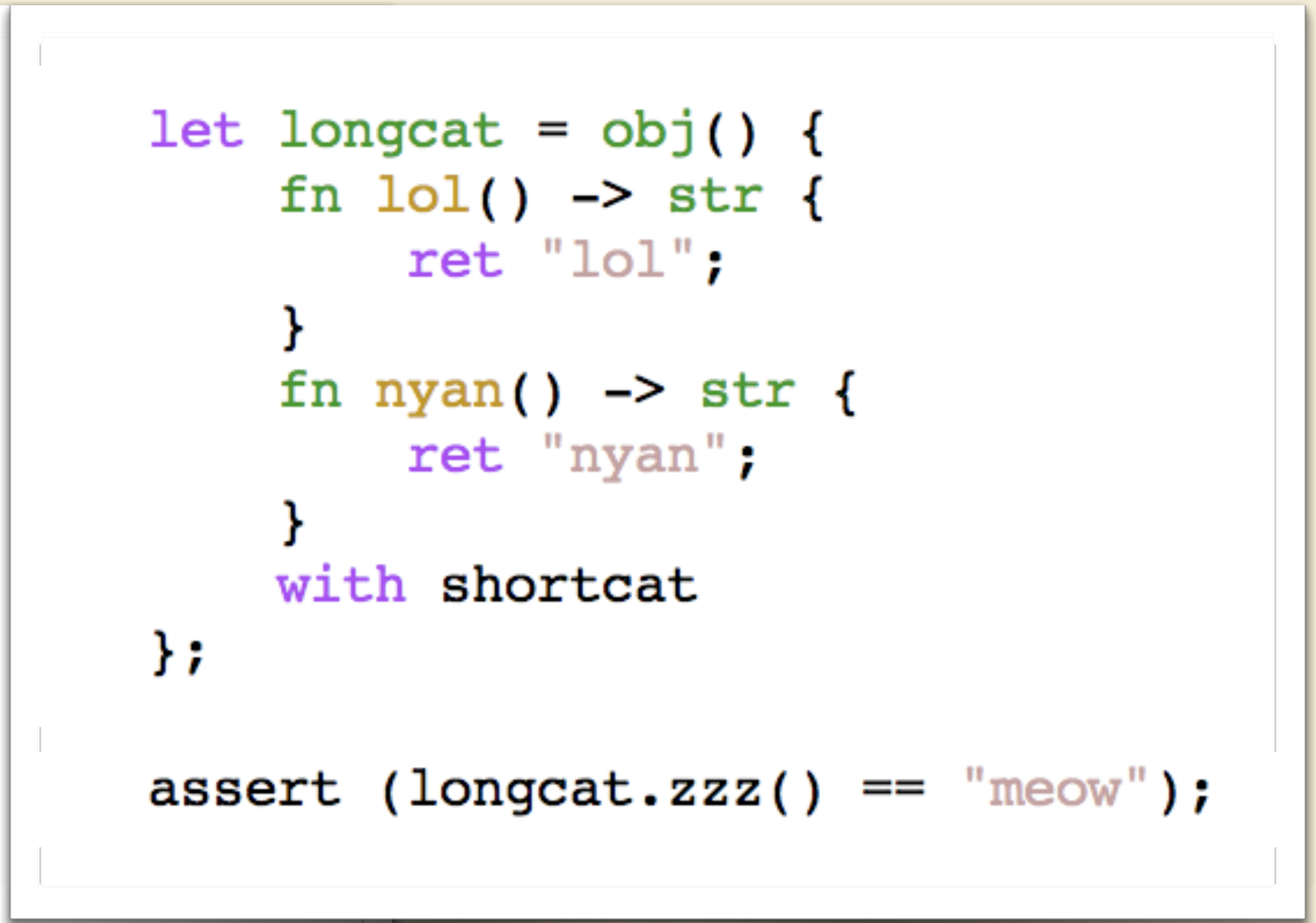


```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();

assert (shortcat.zzz() == "meow");
```

Self-dispatch + object extension

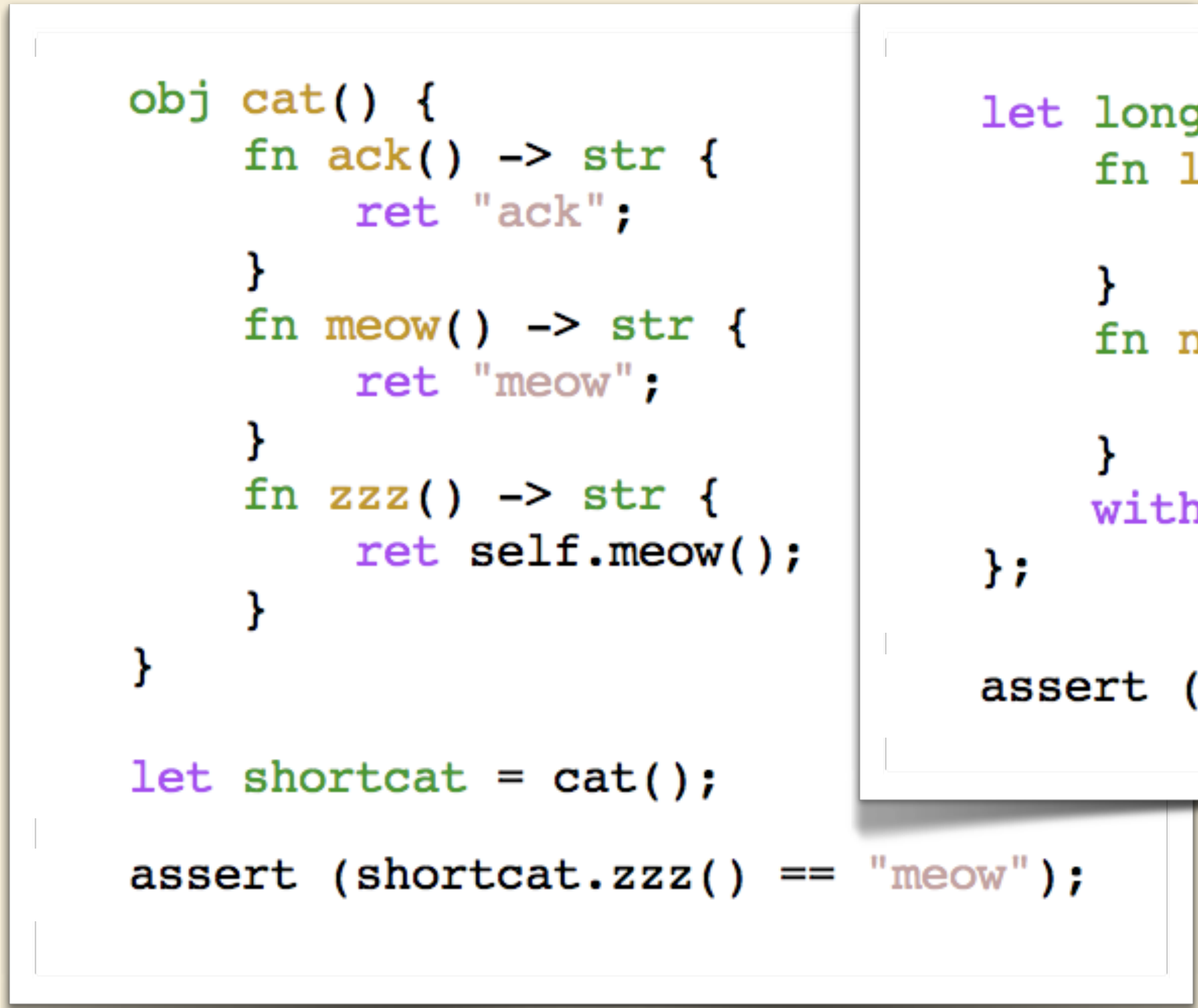


```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();

assert (shortcat.zzz() == "meow");
```

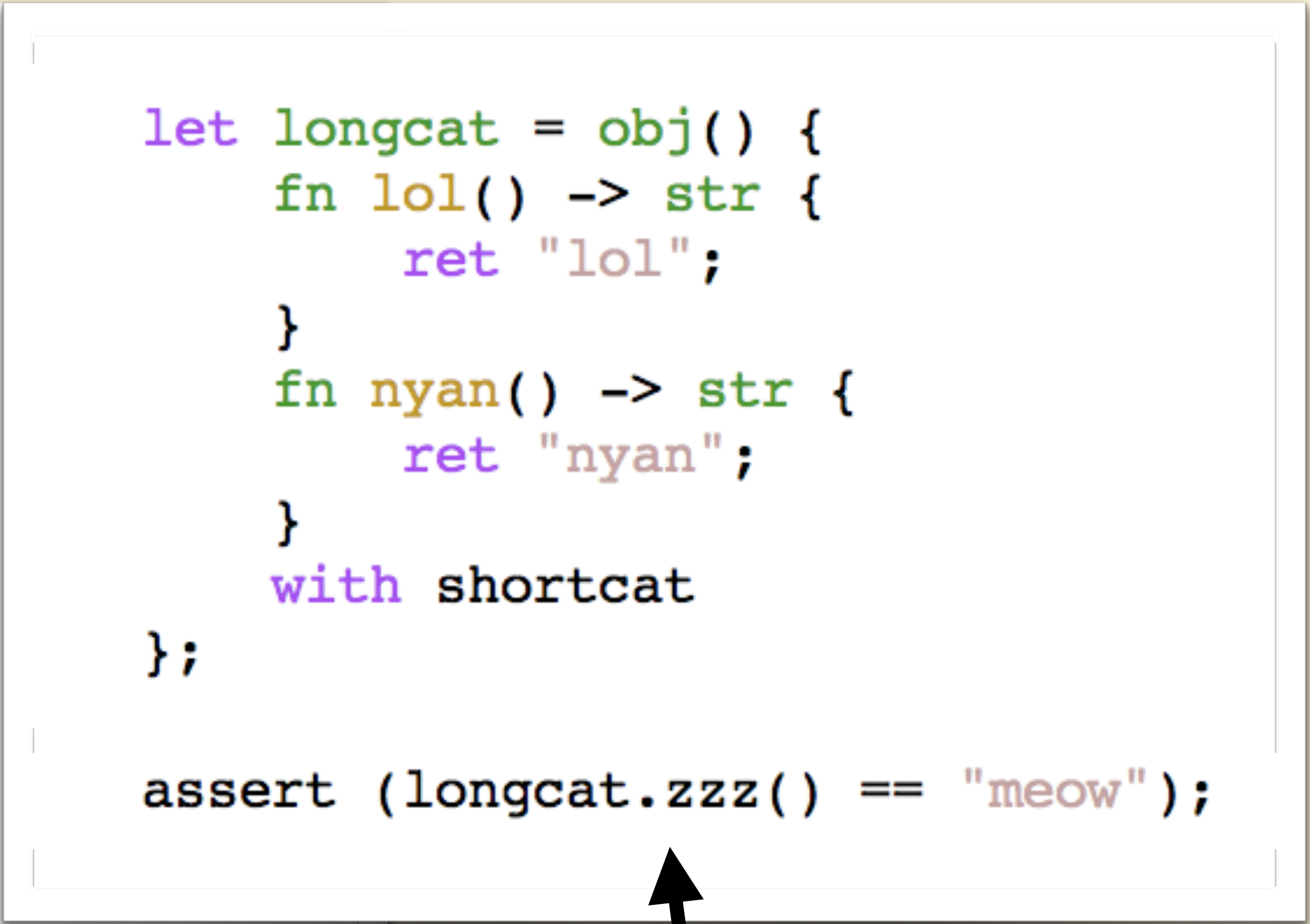


```
let longcat = obj() {
    fn lol() -> str {
        ret "lol";
    }
    fn nyan() -> str {
        ret "nyan";
    }
    with shortcat
};

assert (longcat.zzz() == "meow");
```

A brainteaser...



```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();

assert (shortcat.zzz() == "meow");
```

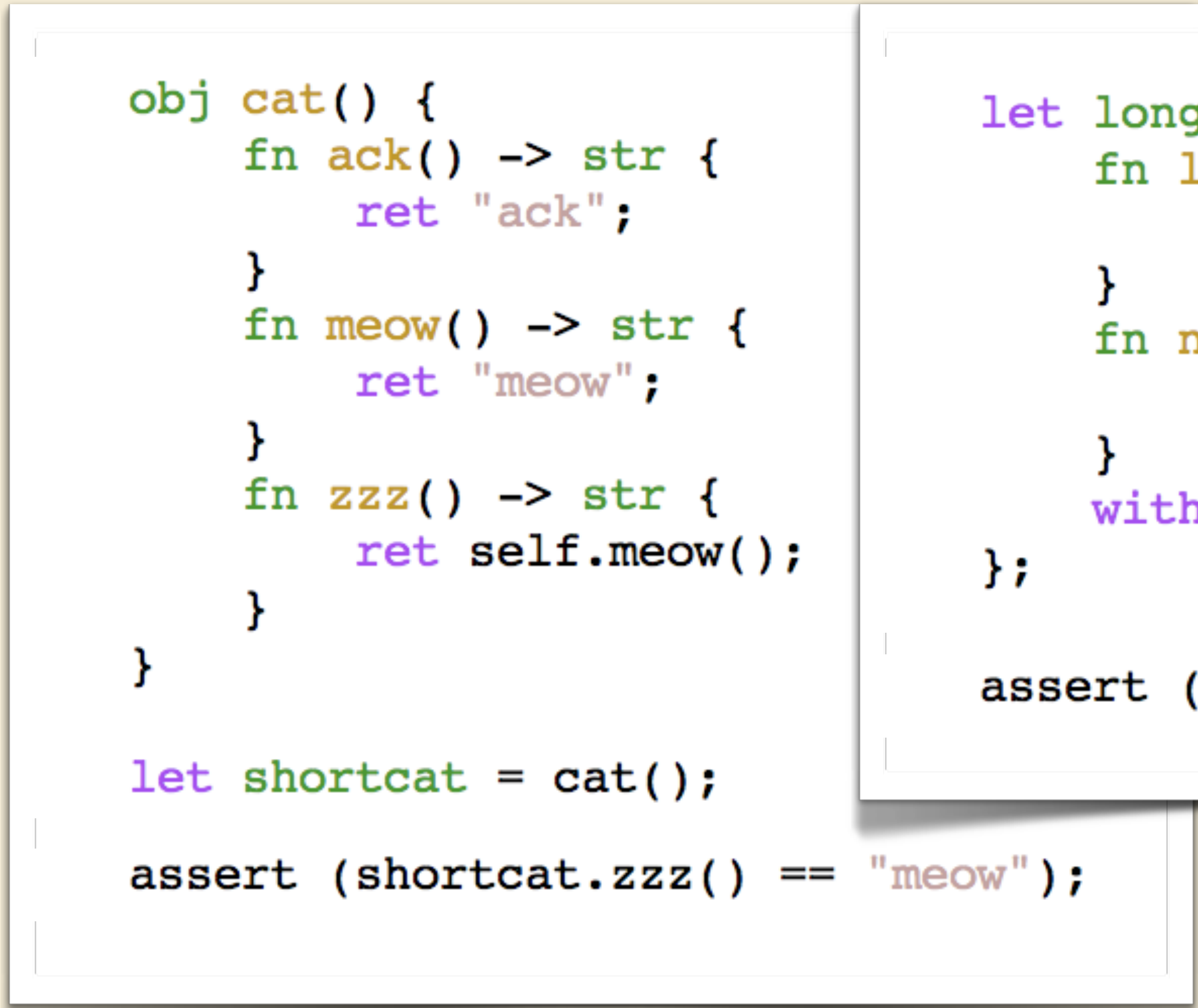


```
let longcat = obj() {
    fn lol() -> str {
        ret "lol";
    }
    fn nyan() -> str {
        ret "nyan";
    }
    with shortcat
};

assert (longcat.zzz() == "meow");
```

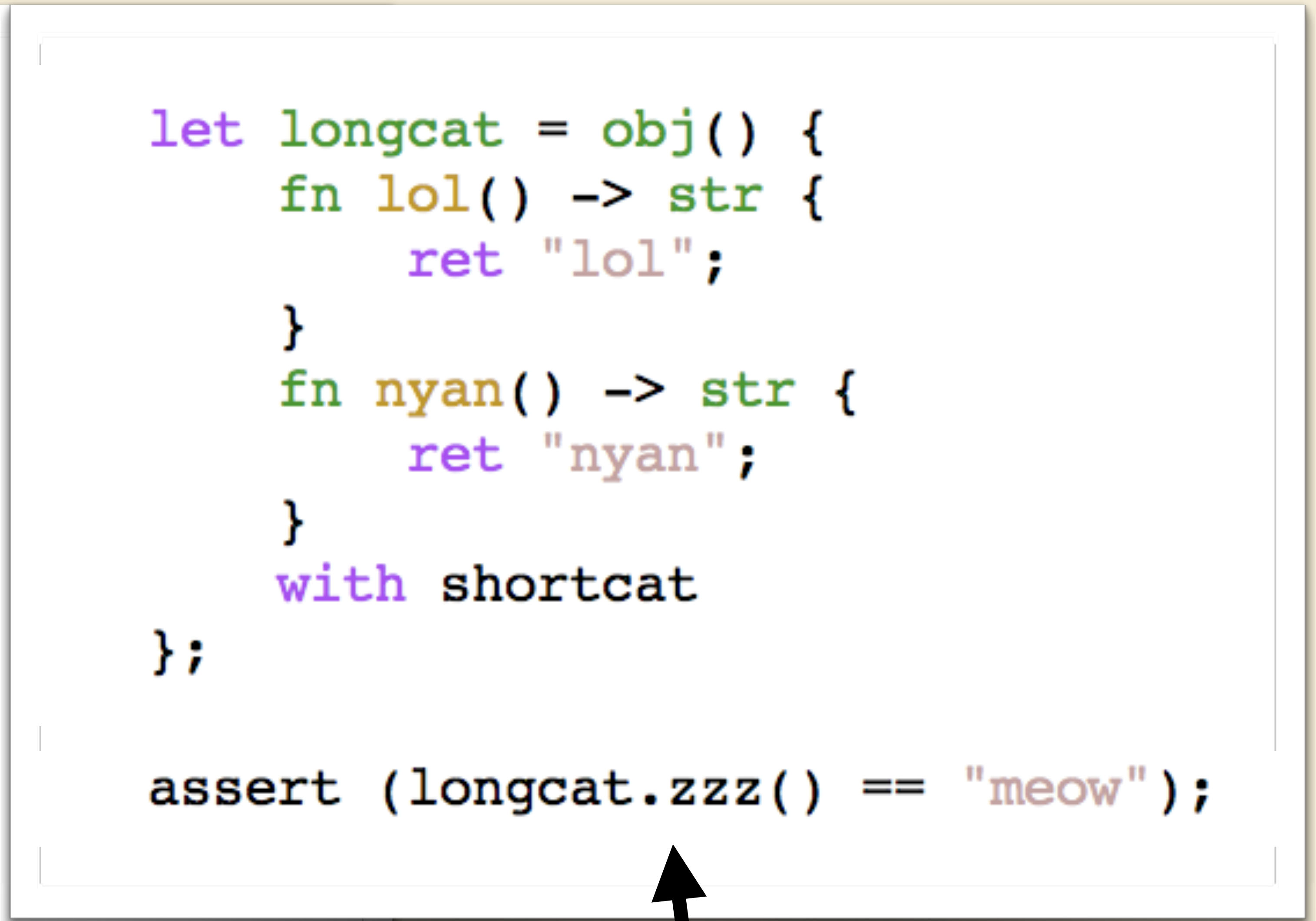
After my first implementation attempt, this assertion failed. Why?

A hint...

A brainteaser...



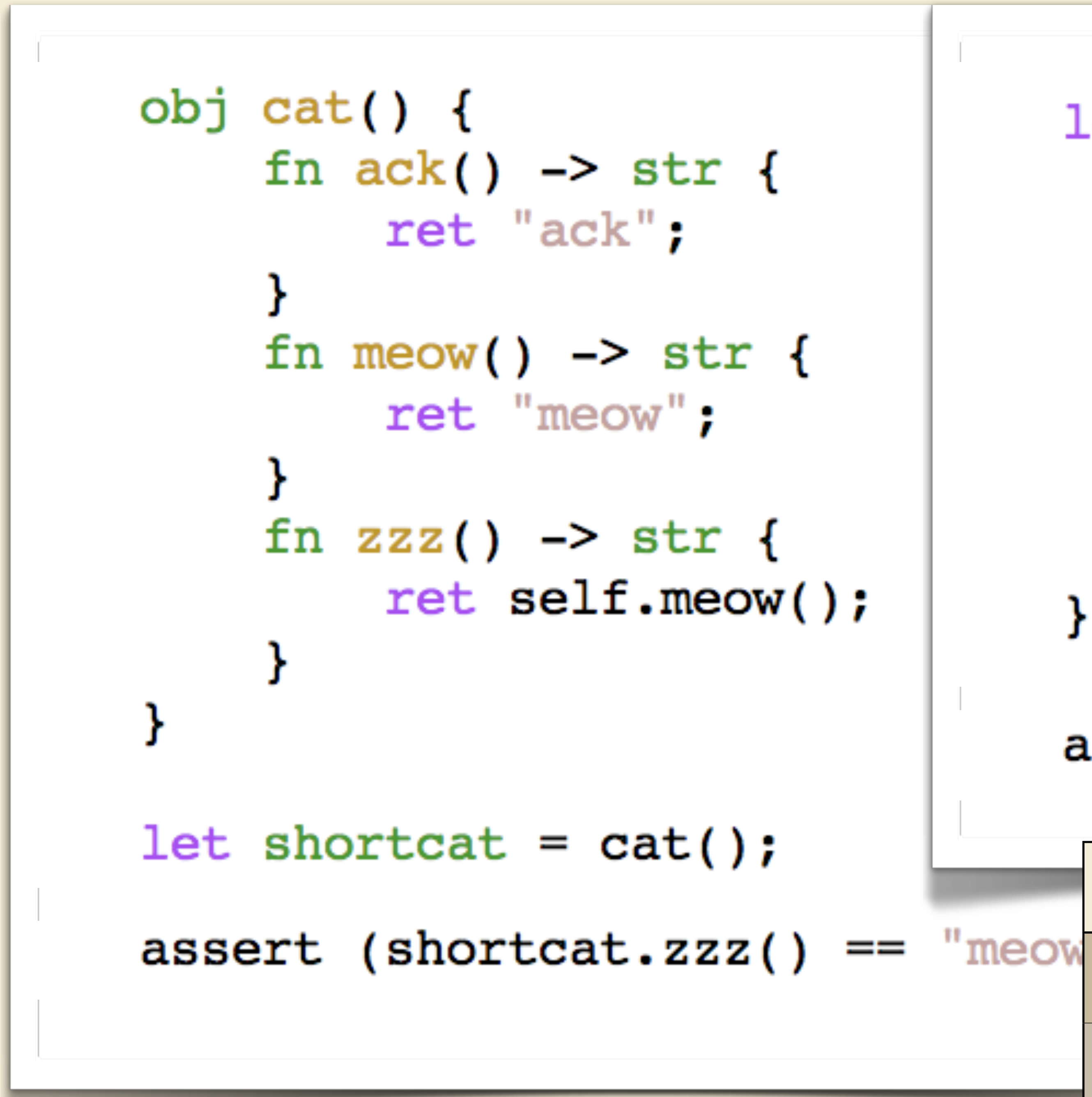
```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();

assert (shortcat.zzz() == "meow");
```



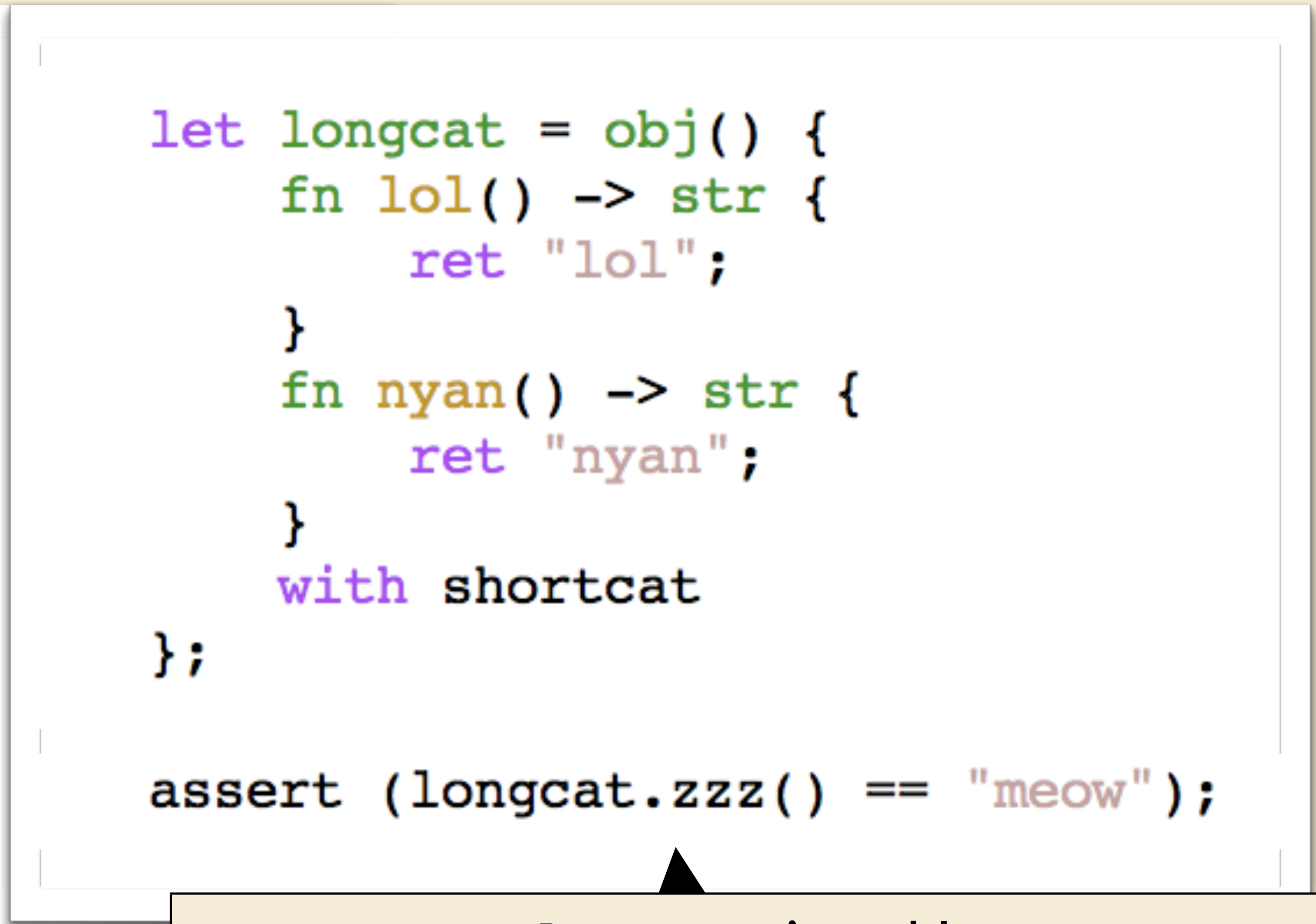
```
let longcat = obj() {
    fn lol() -> str {
        ret "lol";
    }
    fn nyan() -> str {
        ret "nyan";
    }
    with shortcat
};

assert (longcat.zzz() == "meow");
```

After my first implementation attempt, this assertion failed. Why?

A brainteaser...



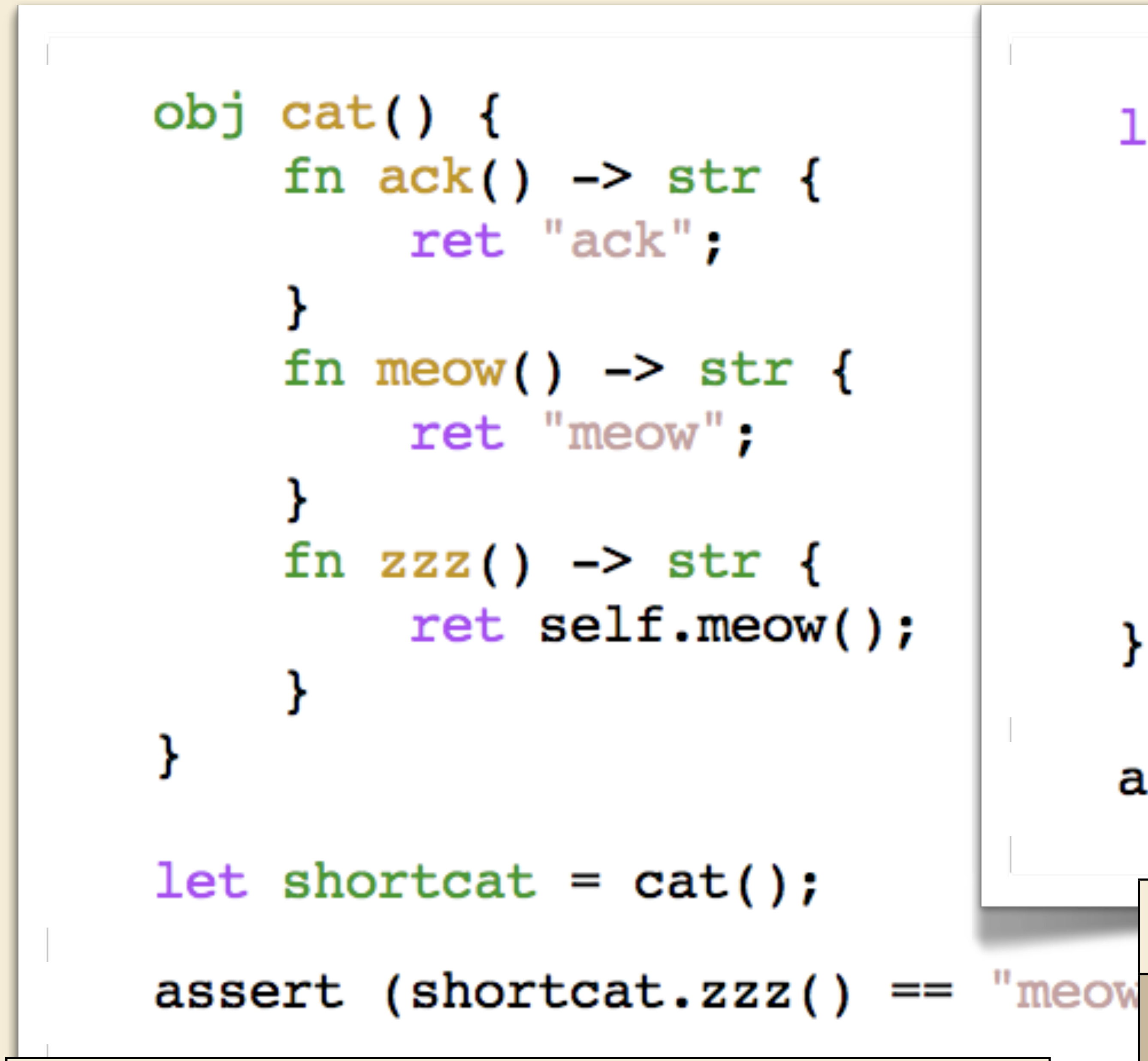
```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();

assert (shortcat.zzz() == "meow
```



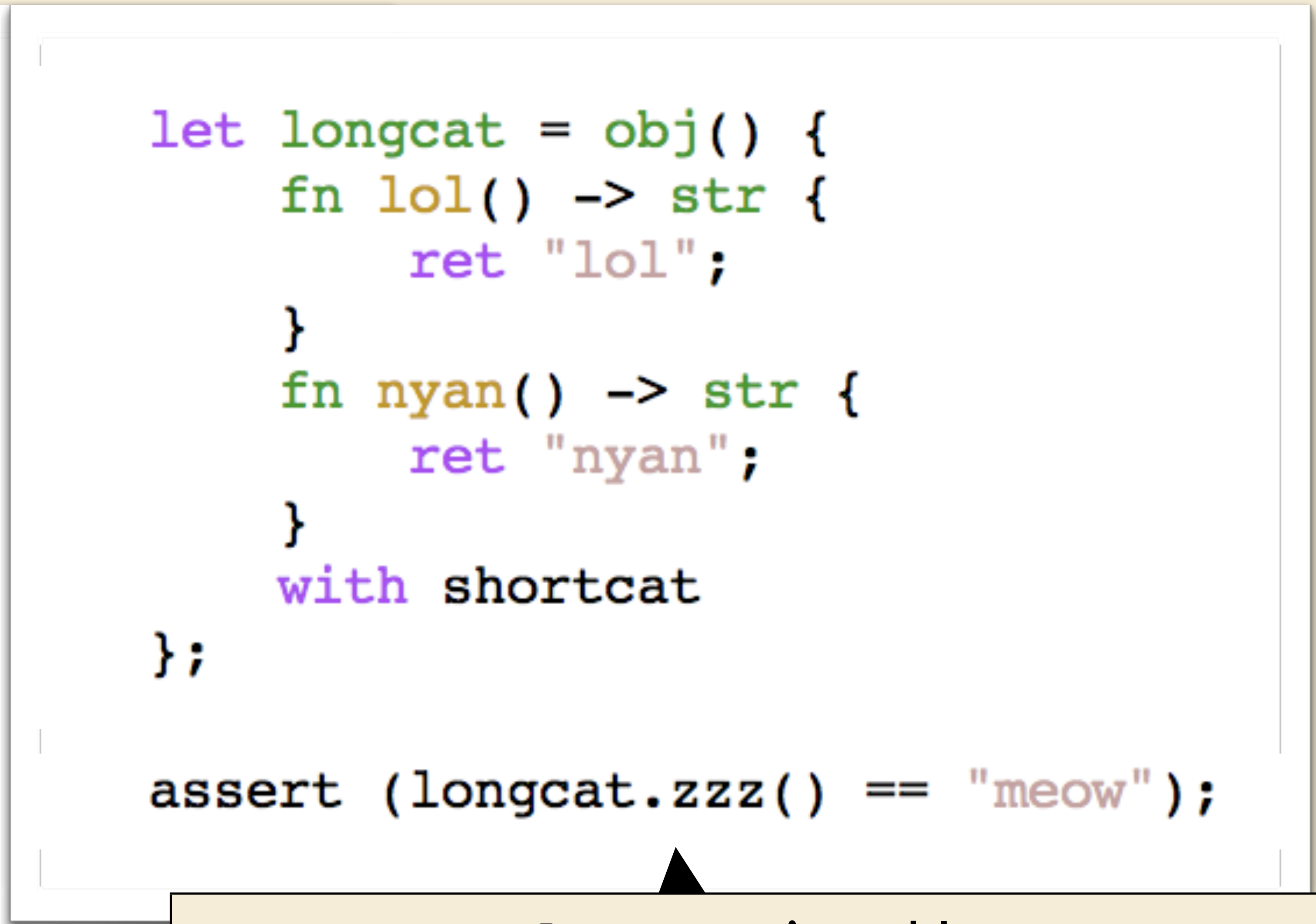
```
let longcat = obj() {
    fn lol() -> str {
        ret "lol";
    }
    fn nyan() -> str {
        ret "nyan";
    }
    with shortcat
};

assert (longcat.zzz() == "meow");
```

longcat's vtable		
0	ack	forward to shortcat.ack()
1	lol	ret "lol"
2	meow	forward to shortcat.meow()
3	nyan	ret "nyan"
4	zzz	forward to shortcat.zzz()

A brainteaser...



```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();

assert (shortcat.zzz() == "meow
```

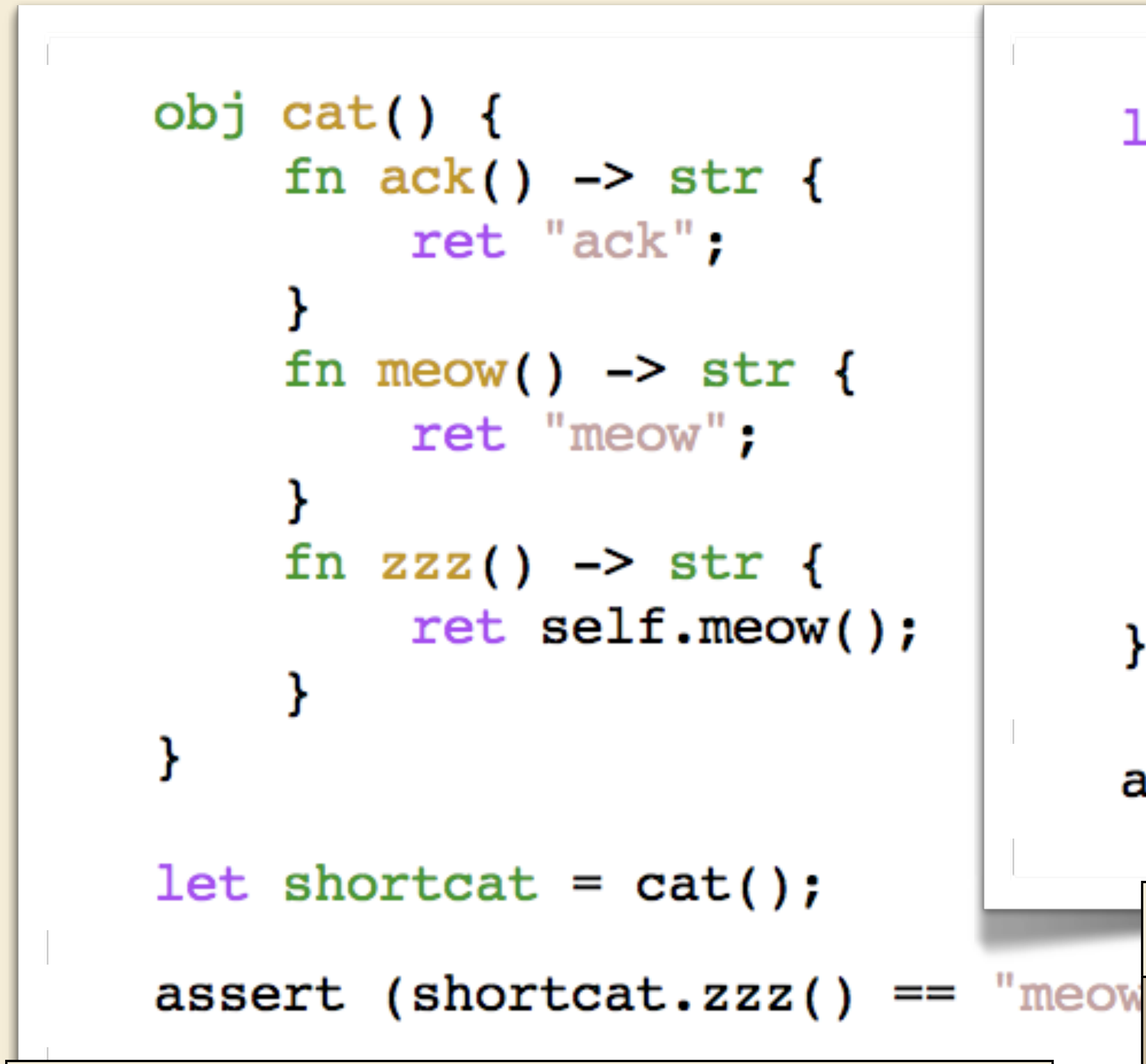


```
let longcat = obj() {
        fn lol() -> str {
            ret "lol";
        }
        fn nyan() -> str {
            ret "nyan";
        }
        with shortcat
};

assert (longcat.zzz() == "meow");
```

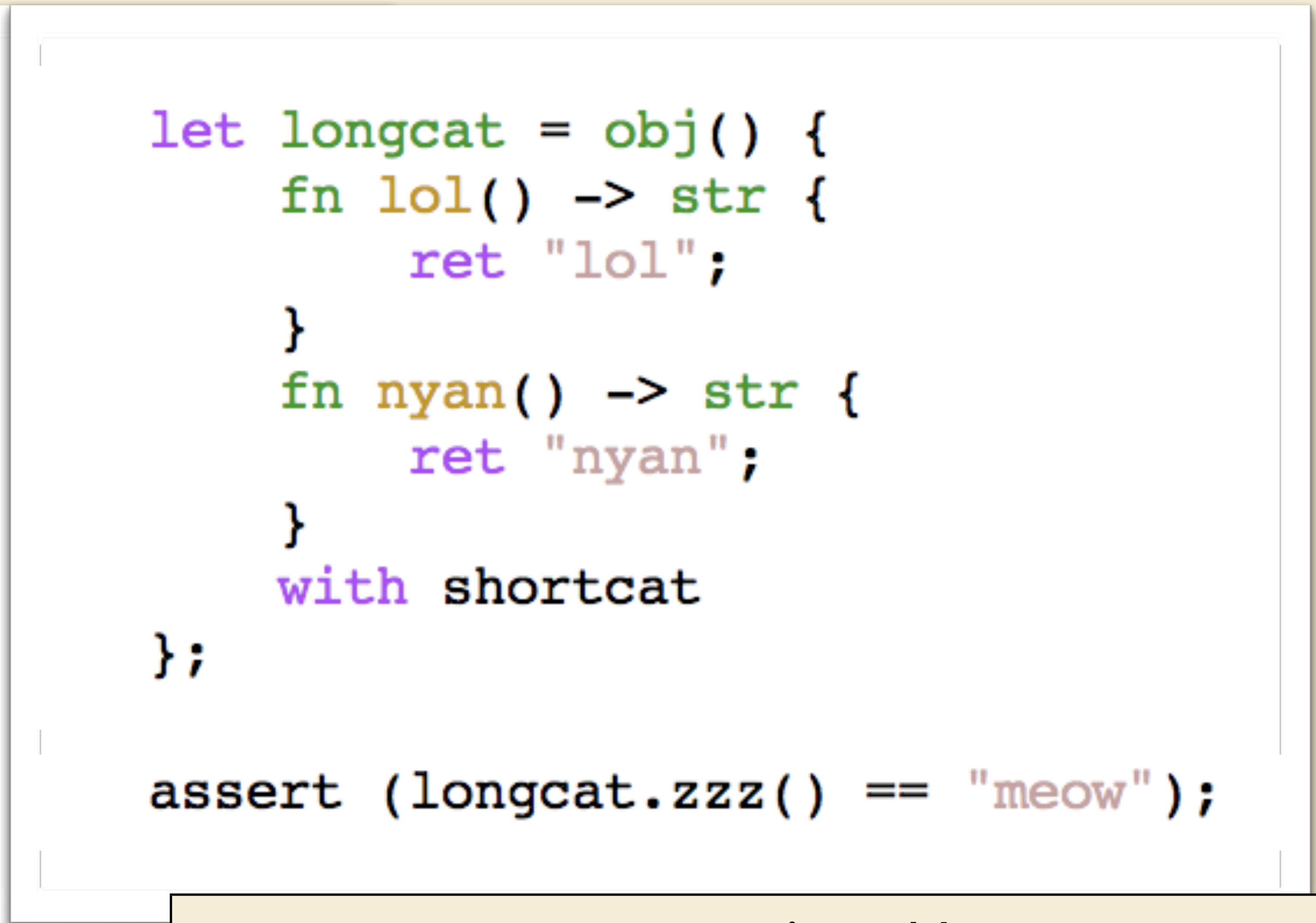
shortcat’s vtable		
0	ack	ret “ack”
1	meow	ret “meow”
2	zzz	ret self.meow()

longcat’s vtable		
0	ack	forward to shortcat.ack()
1	lol	ret “lol”
2	meow	forward to shortcat.meow()
3	nyan	ret “nyan”
4	zzz	forward to shortcat.zzz()

How to fix it



```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();

assert (shortcat.zzz() == "meow
```

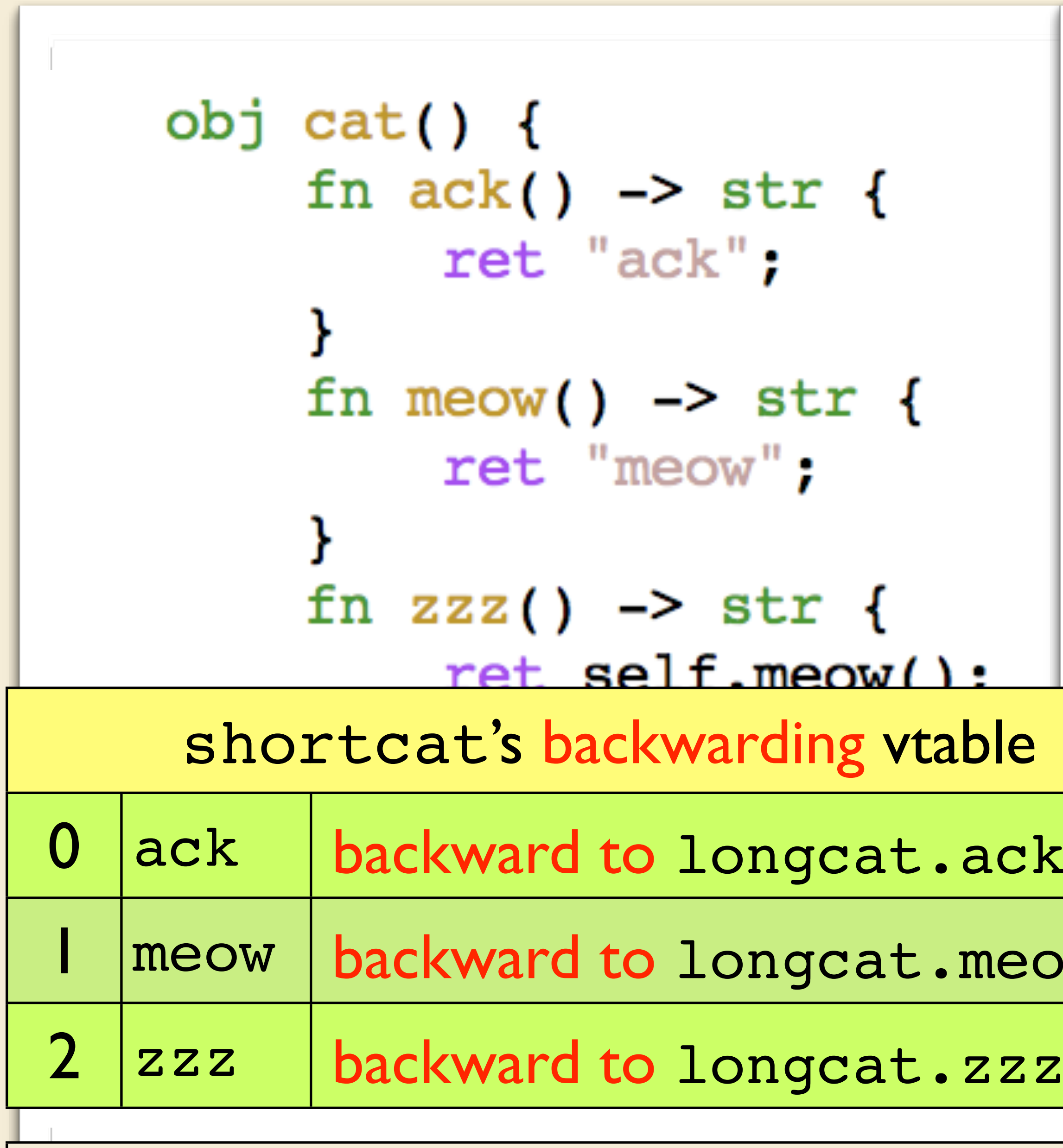


```
let longcat = obj() {
    fn lol() -> str {
        ret "lol";
    }
    fn nyan() -> str {
        ret "nyan";
    }
    with shortcat
};

assert (longcat.zzz() == "meow");
```

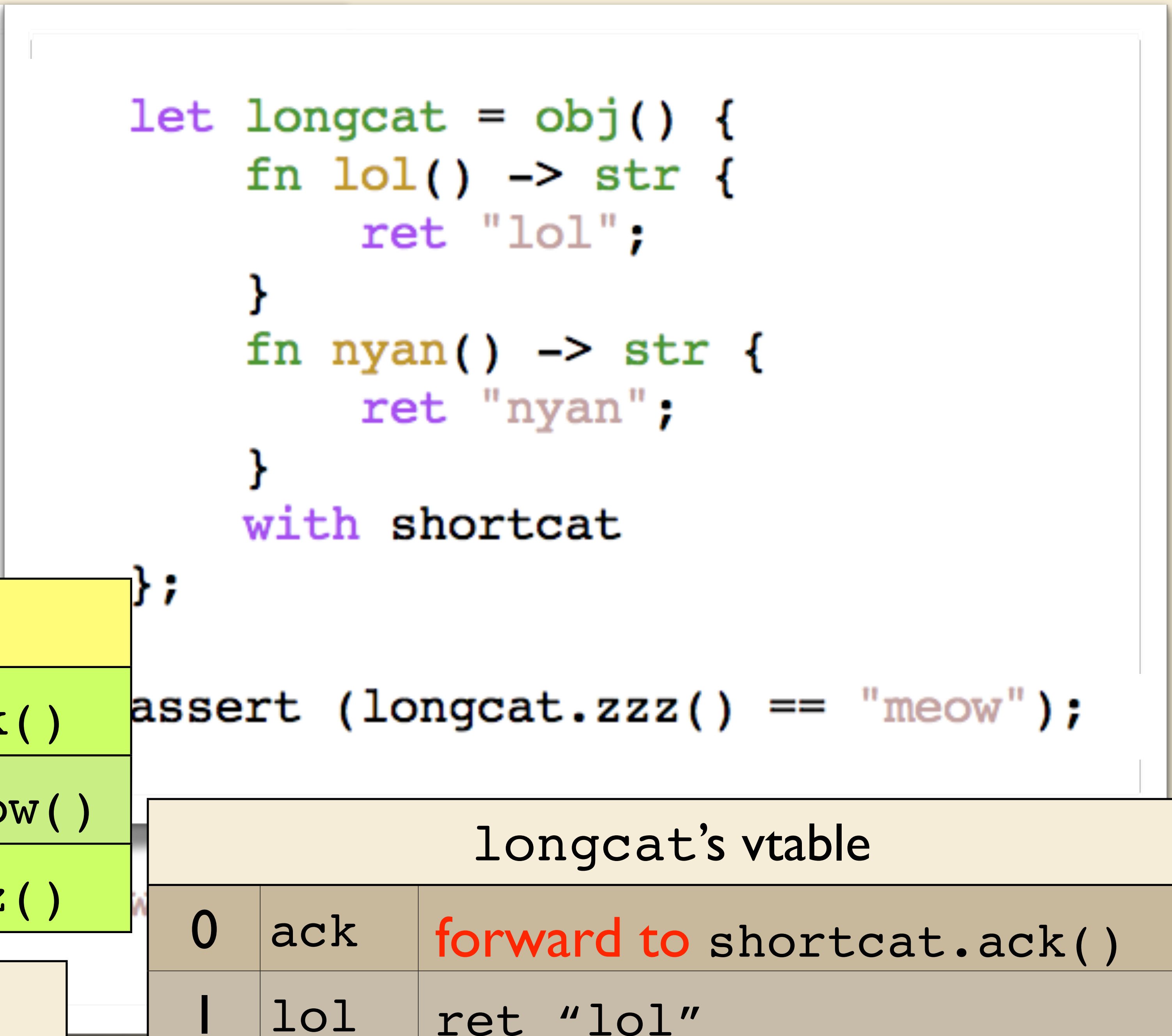
shortcat's vtable		
0	ack	ret "ack"
1	meow	ret "meow"
2	zzz	ret self.meow()

longcat's vtable		
0	ack	forward to shortcat.ack()
1	lol	ret "lol"
2	meow	forward to shortcat.meow()
3	nyan	ret "nyan"
4	zzz	forward to shortcat.zzz()

How to fix it



```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
```



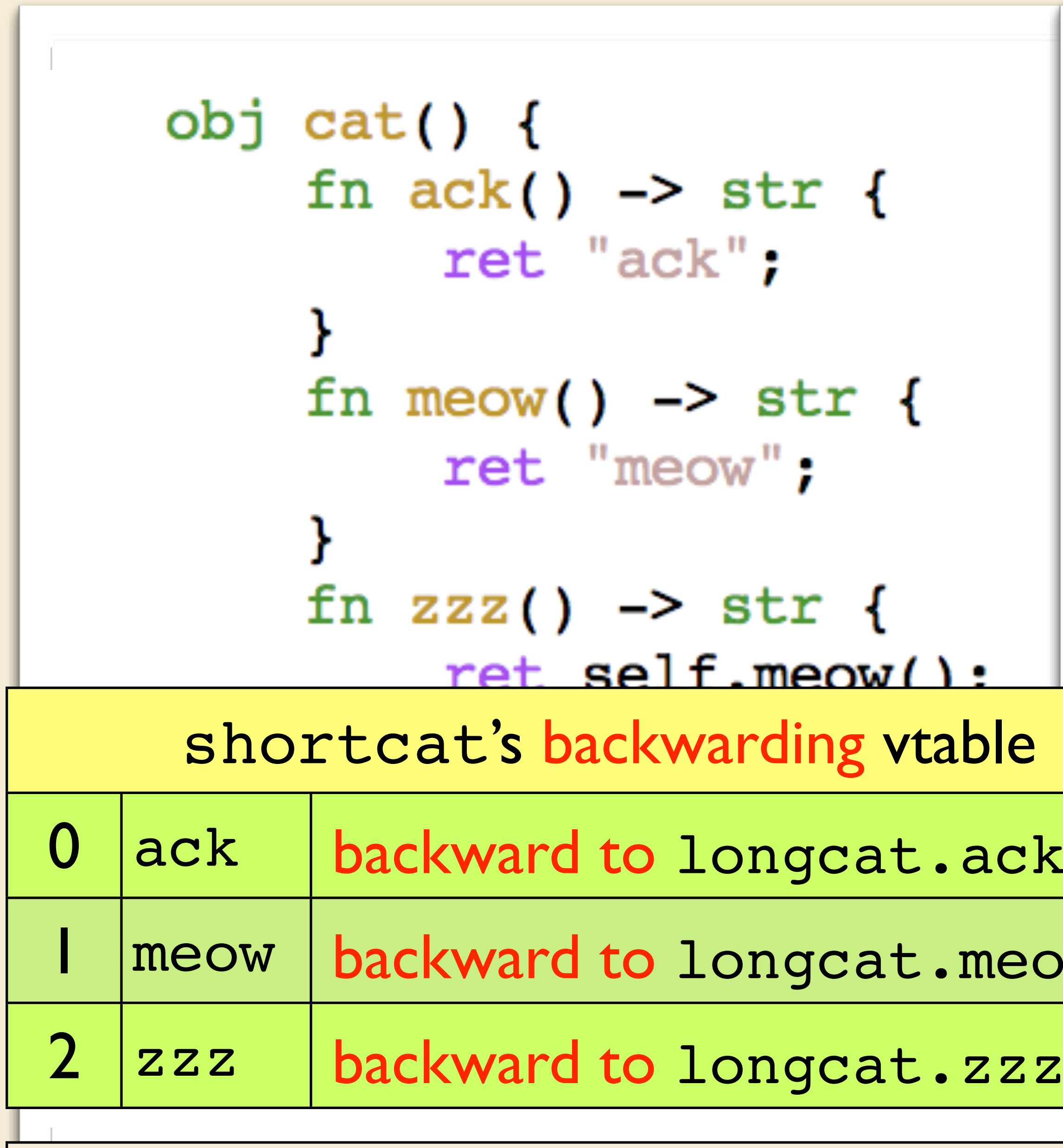
```
let longcat = obj() {
    fn lol() -> str {
        ret "lol";
    }
    fn nyan() -> str {
        ret "nyan";
    }
    with shortcat
};

assert (longcat.zzz() == "meow");
```

shortcat's backwarding vtable		
0	ack	backward to longcat.ack()
1	meow	backward to longcat.meow()
2	zzz	backward to longcat.zzz()

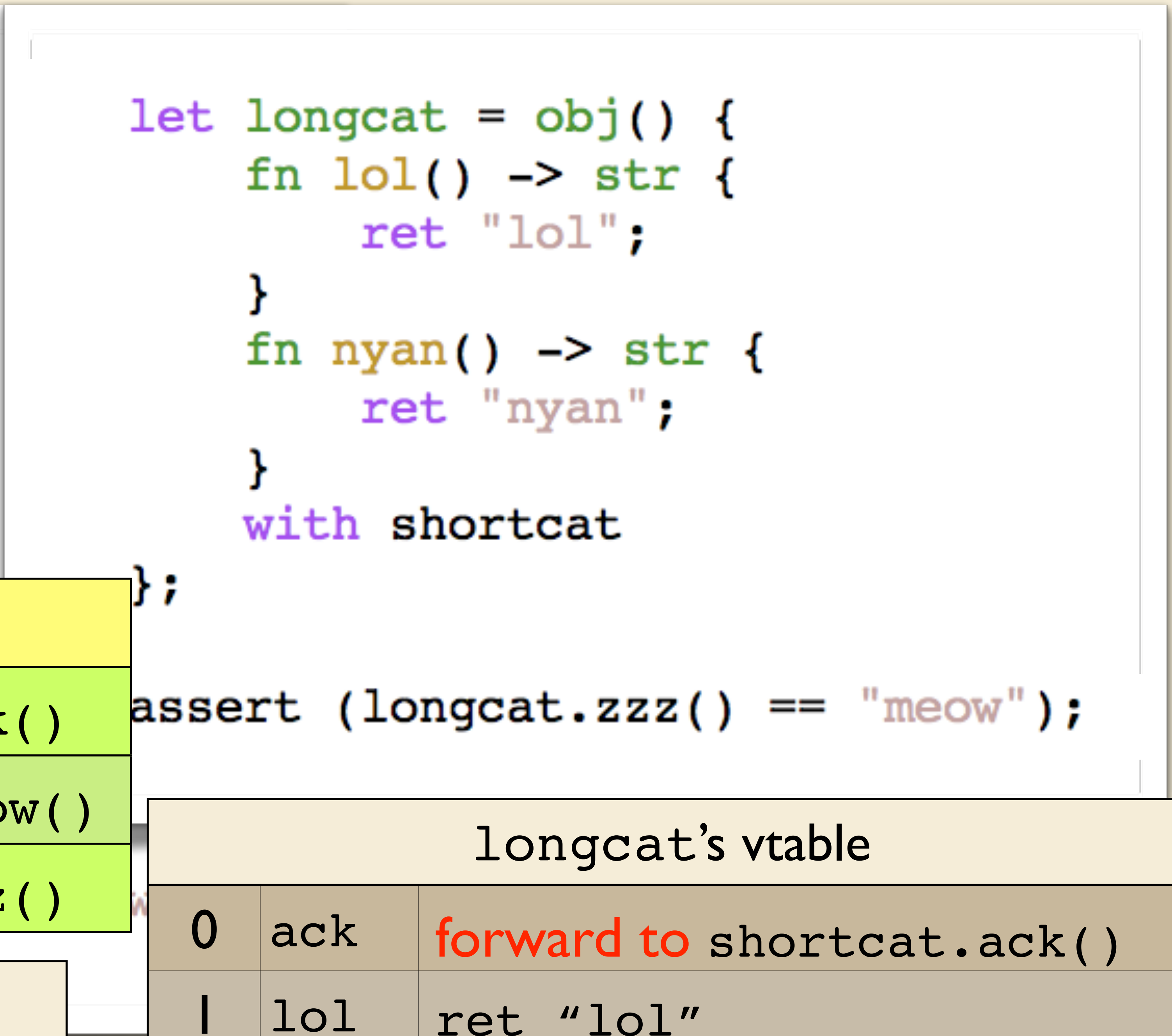
shortcat's vtable		
0	ack	ret "ack"
1	meow	ret "meow"
2	zzz	ret self.meow()

longcat's vtable		
0	ack	forward to shortcat.ack()
1	lol	ret "lol"
2	meow	forward to shortcat.meow()
3	nyan	ret "nyan"
4	zzz	forward to shortcat.zzz()

How to fix it



```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
```



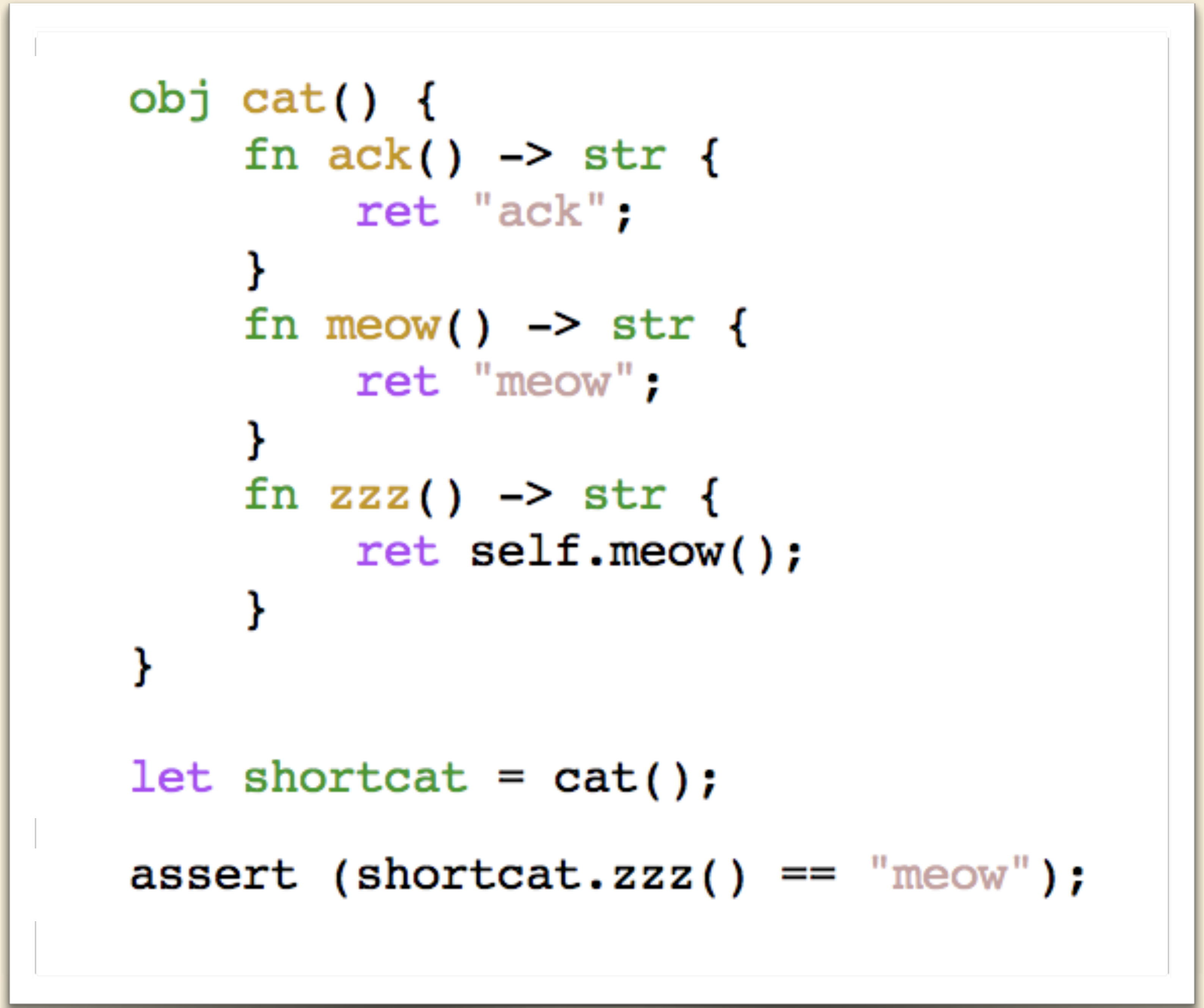
```
let longcat = obj() {
    fn lol() -> str {
        ret "lol";
    }
    fn nyan() -> str {
        ret "nyan";
    }
    with shortcat
};

assert (longcat.zzz() == "meow");
```

shortcat's backwarding vtable		
0	ack	backward to longcat.ack()
1	meow	backward to longcat.meow()
2	zzz	backward to longcat.zzz()

shortcat's vtable		
0	ack	ret "ack"
1	meow	ret "meow"
2	zzz	ret self.meow()

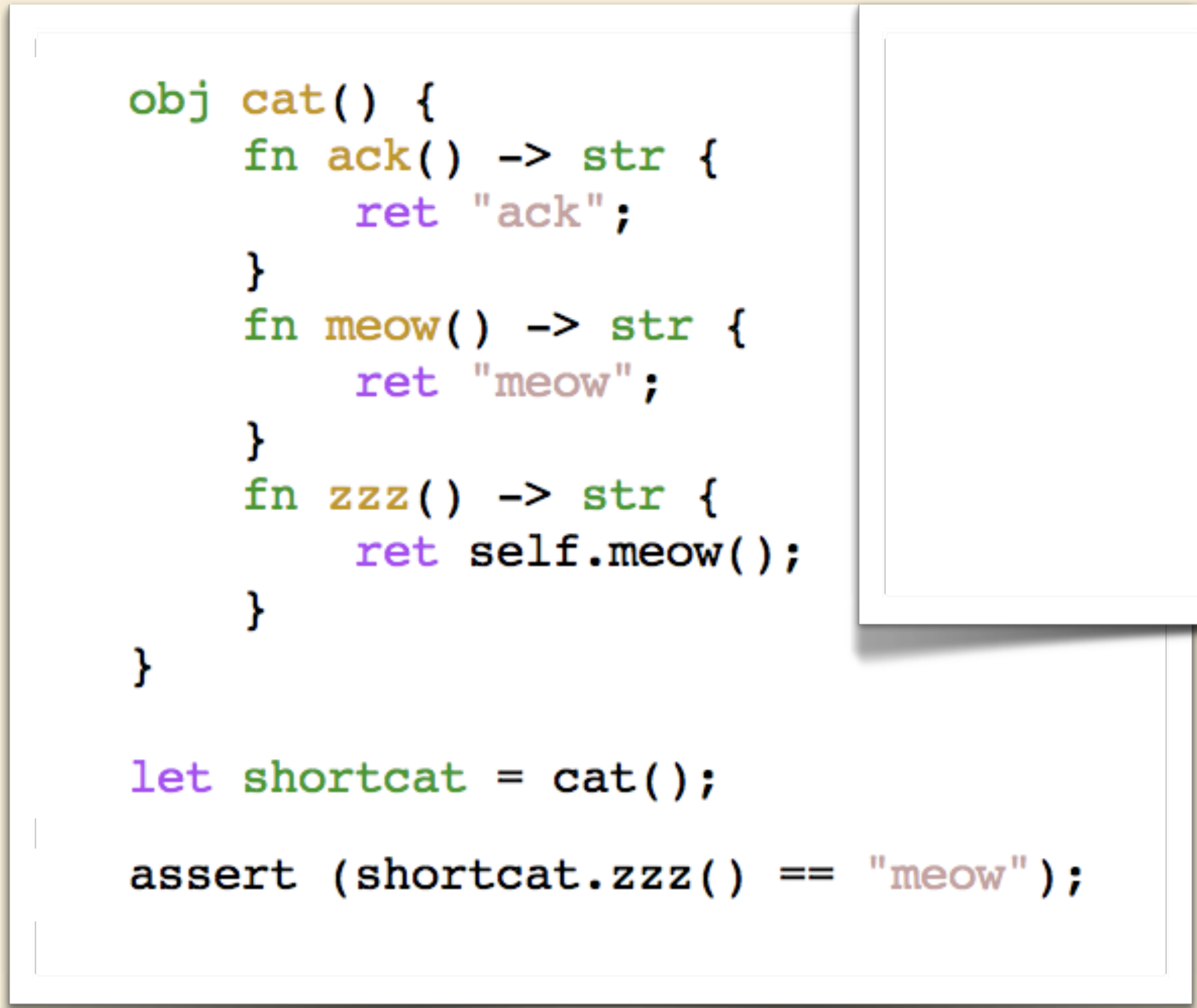
longcat's vtable		
0	ack	forward to shortcat.ack()
1	lol	ret "lol"
2	meow	forward to shortcat.meow()
3	nyan	ret "nyan"
4	zzz	forward to shortcat.zzz()

Self-dispatch + object extension + overriding



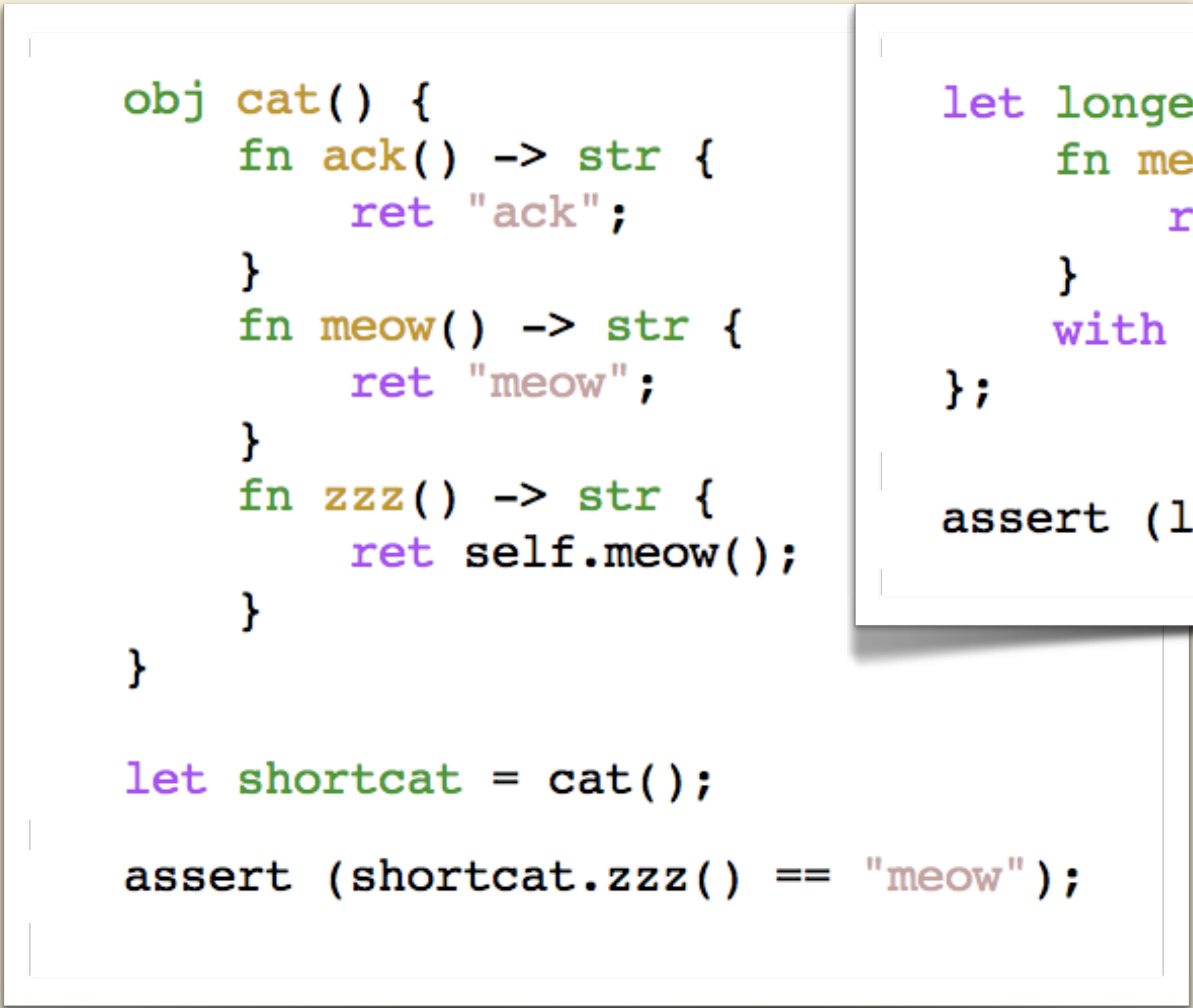
```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();

assert (shortcat.zzz() == "meow");
```

Self-dispatch + object extension + overriding

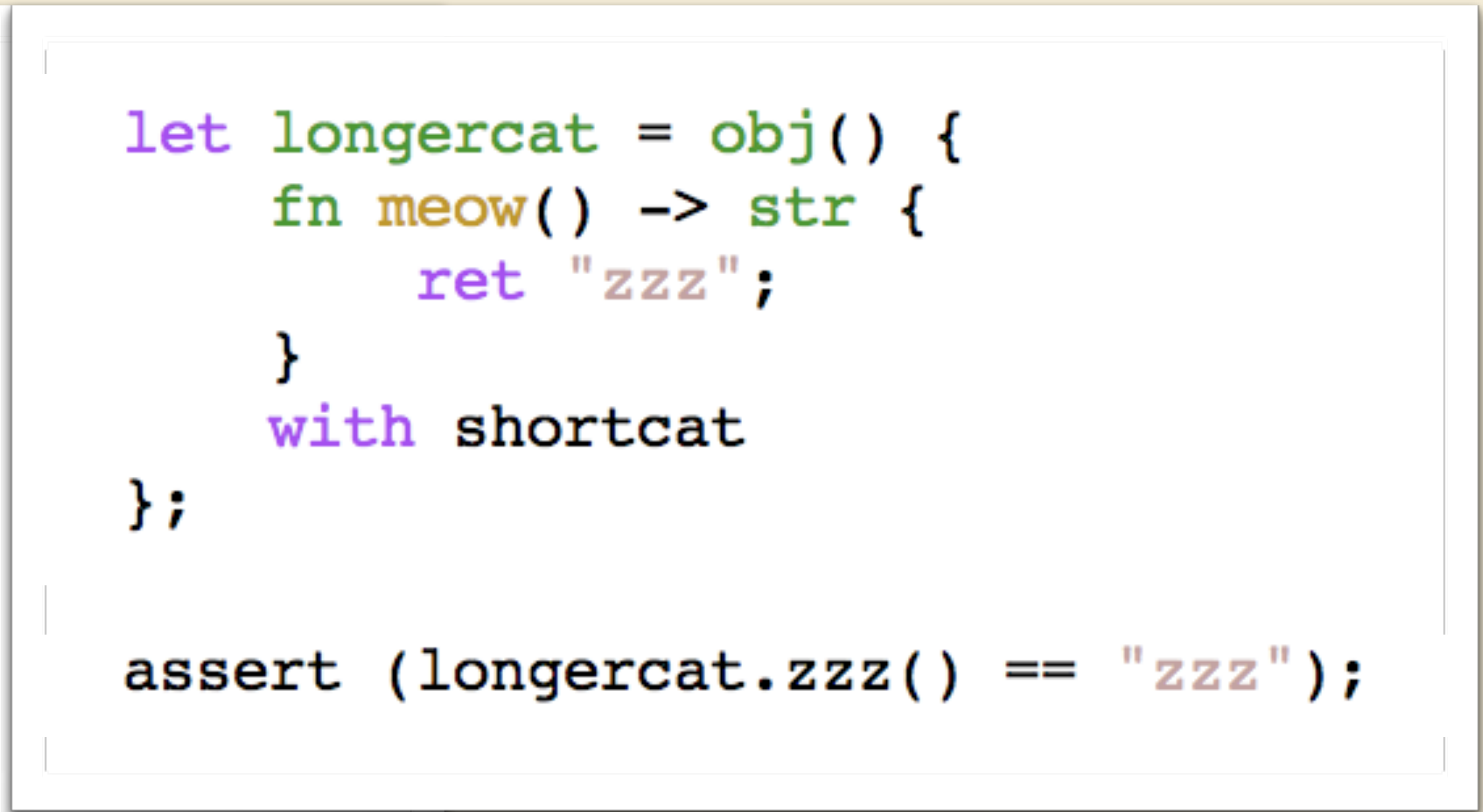


```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();

assert (shortcat.zzz() == "meow");
```

Self-dispatch + object extension + overriding

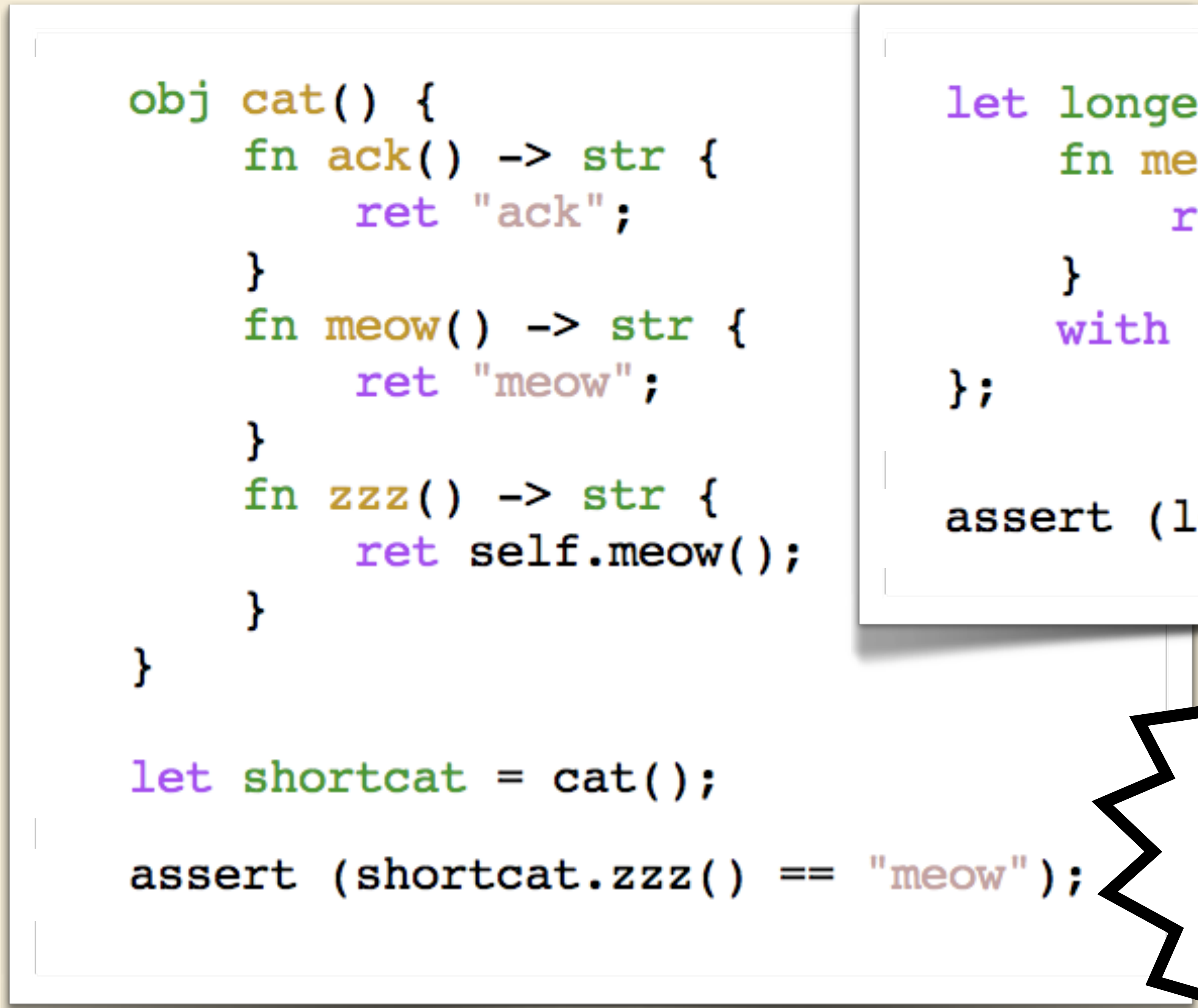


```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();

assert (shortcat.zzz() == "meow");
```

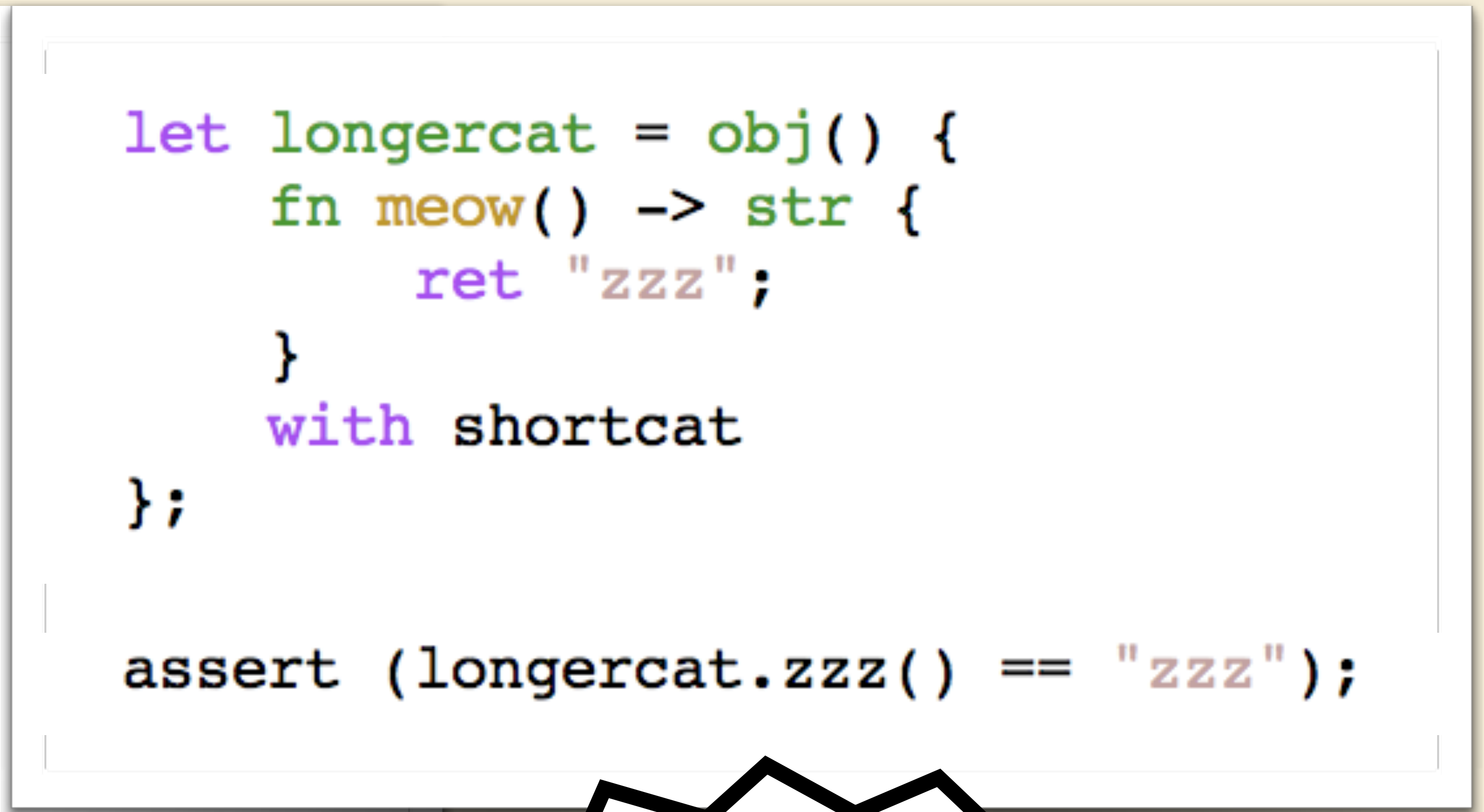


```
let longercat = obj() {
    fn meow() -> str {
        ret "zzz";
    }
    with shortcat
};

assert (longercat.zzz() == "zzz");
```

Self-dispatch + object extension + overriding



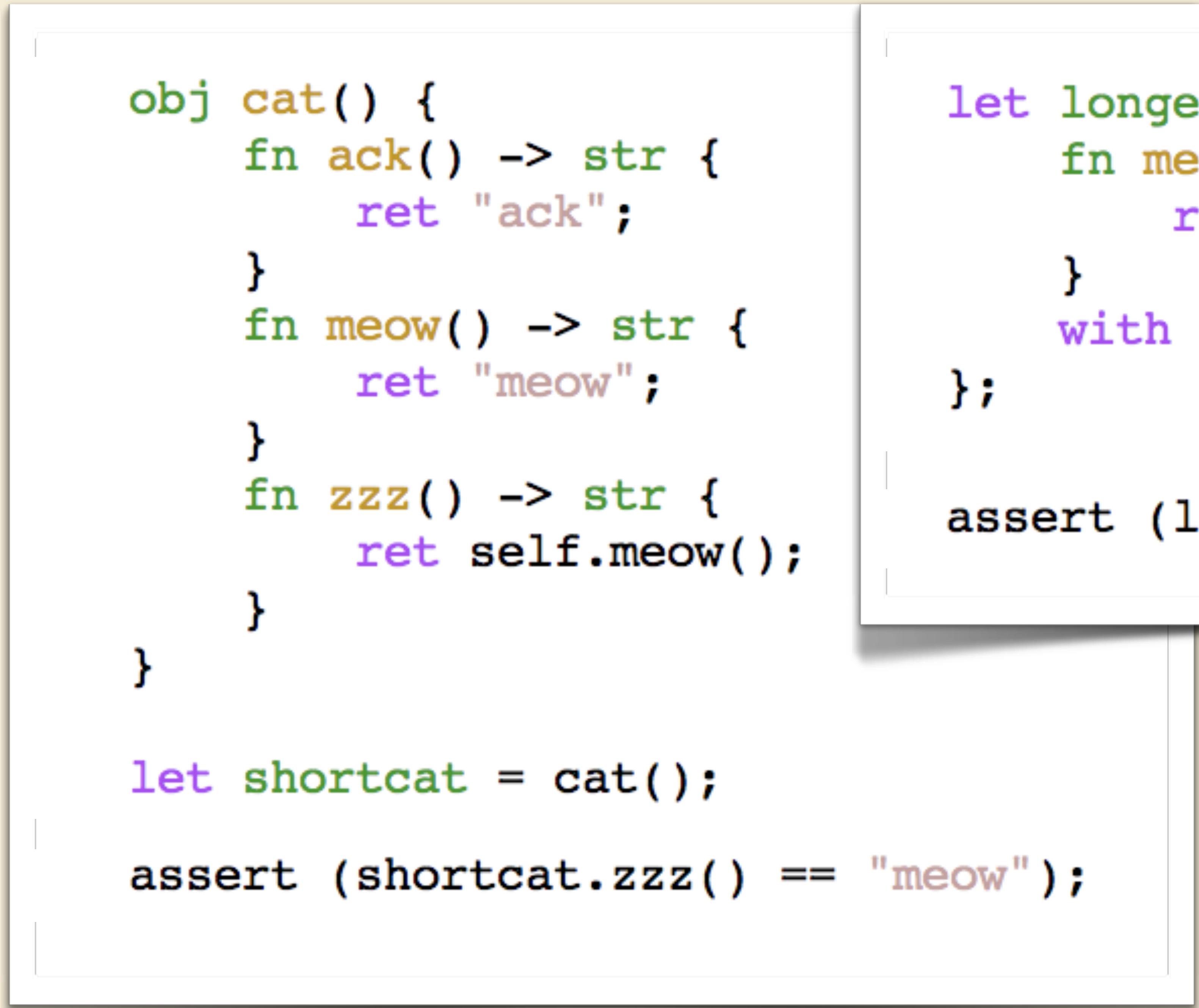
```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();

assert (shortcat.zzz() == "meow");
```



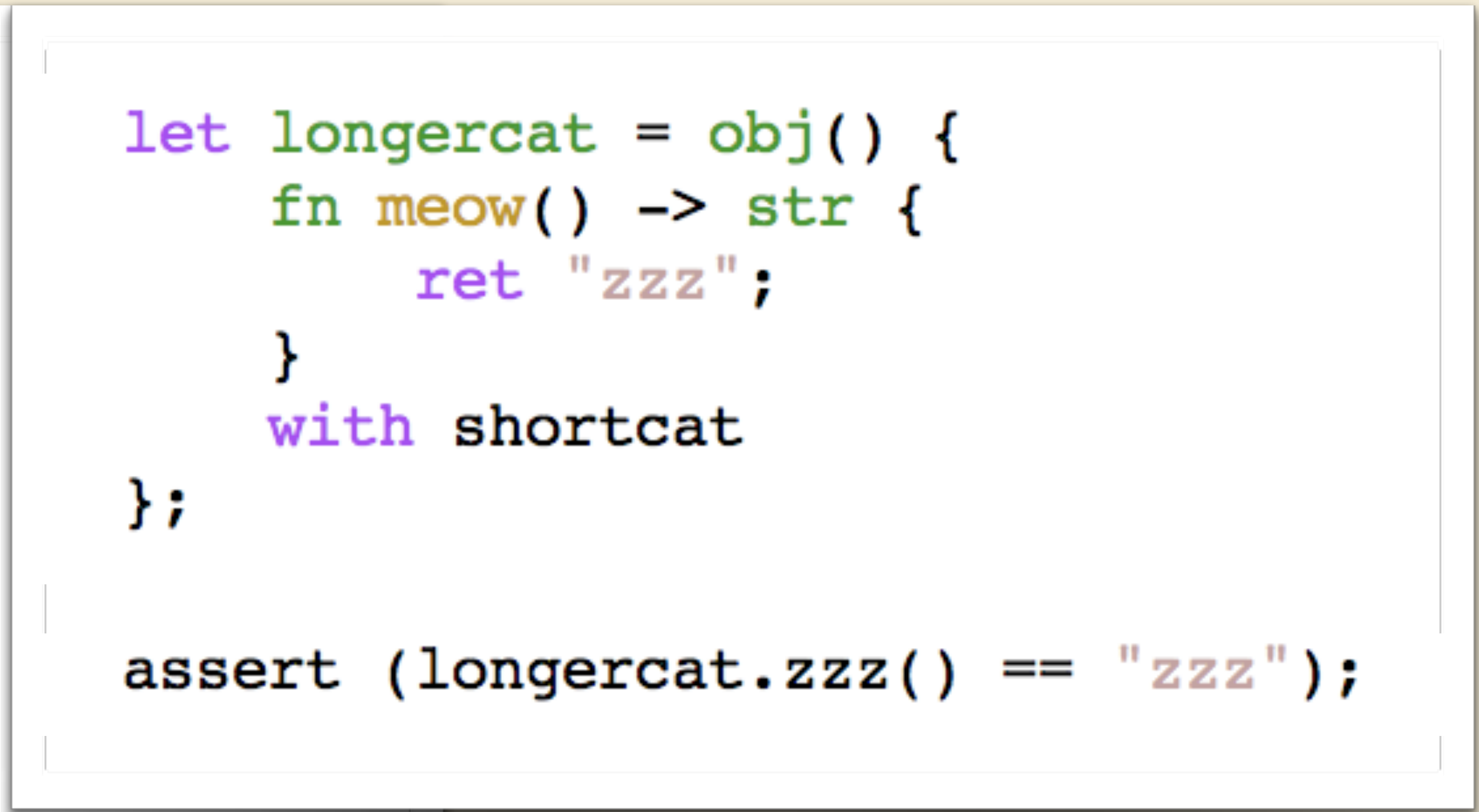
```
let longercat = obj() {
    fn meow() -> str {
        ret "zzz";
    }
    with shortcat
};

assert (longercat.zzz() == "zzz");
```

Caveat: Some disagreement on whether it should work this way (see: Aldrich, "Selective Open Recursion")

Self-dispatch + object extension + overriding

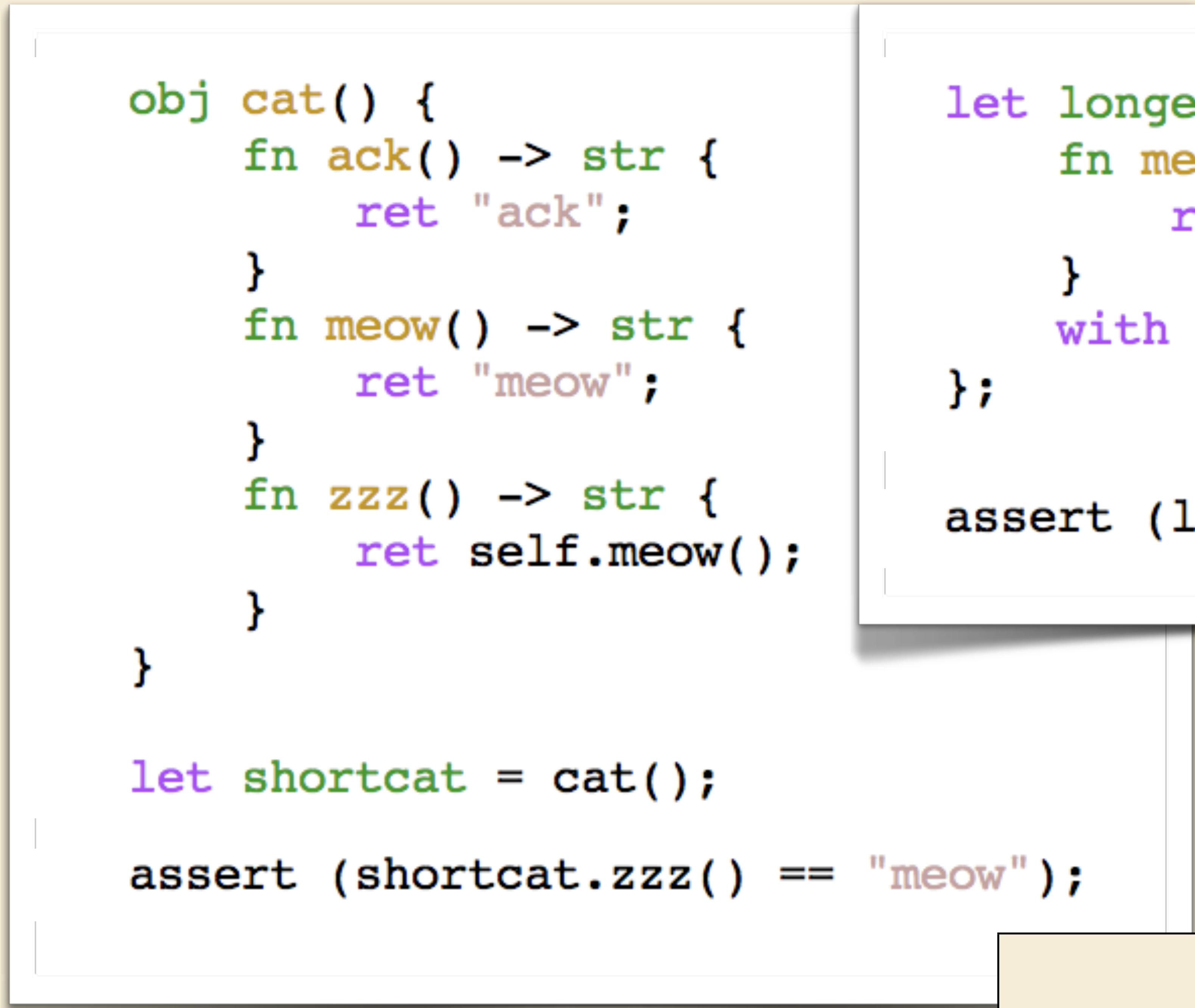


```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();

assert (shortcat.zzz() == "meow");
```

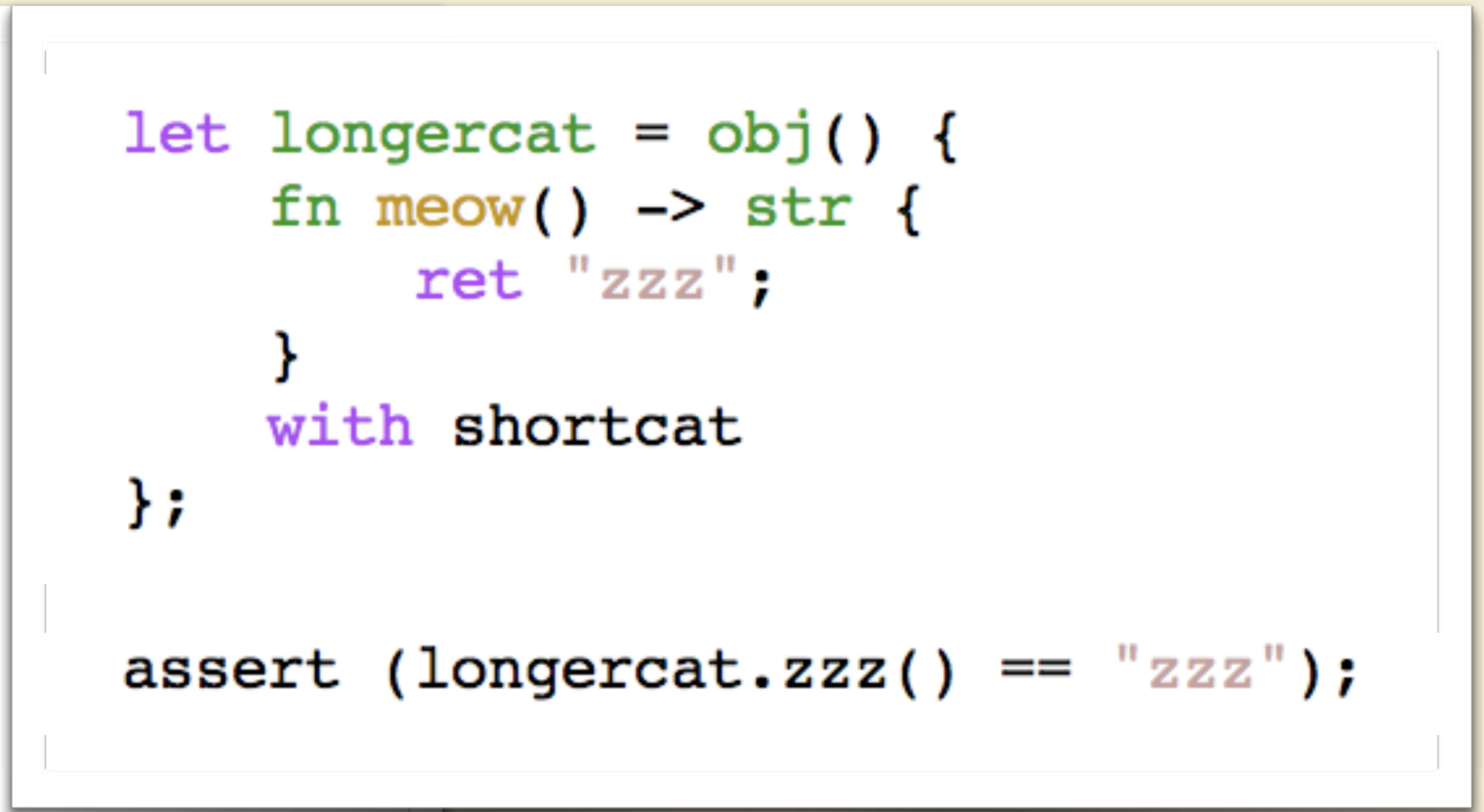


```
let longercat = obj() {
    fn meow() -> str {
        ret "zzz";
    }
    with shortcat
};

assert (longercat.zzz() == "zzz");
```

Self-dispatch + object extension + overriding



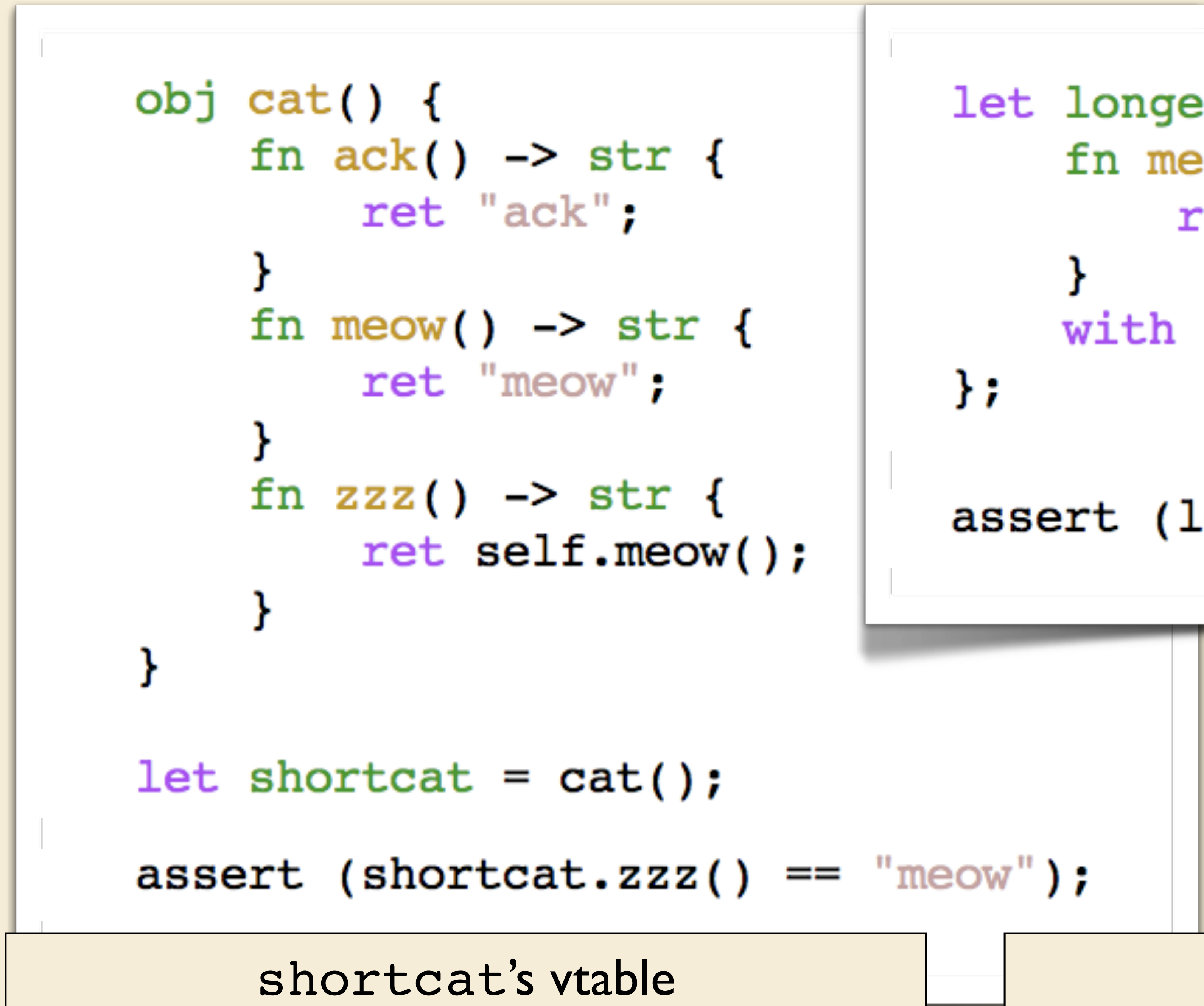
```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();

assert (shortcat.zzz() == "meow");
```



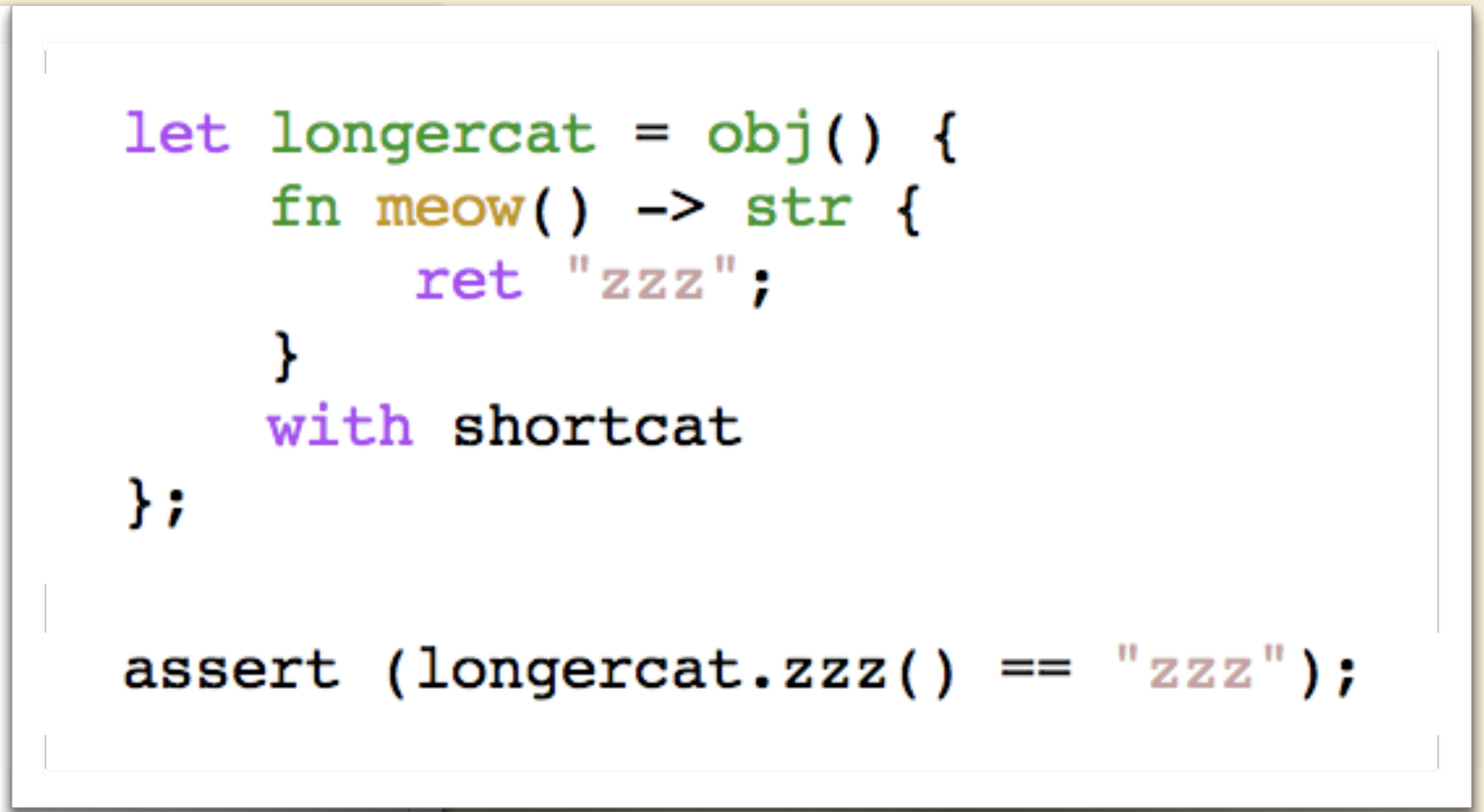
```
let longercat = obj() {
    fn meow() -> str {
        ret "zzz";
    }
    with shortcat
};

assert (longercat.zzz() == "zzz");
```

longercat's vtable		
0	ack	forward to shortcat.ack()
1	meow	ret "zzz"
2	zzz	forward to shortcat.zzz()

Self-dispatch + object extension + overriding



```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();

assert (shortcat.zzz() == "meow");
```

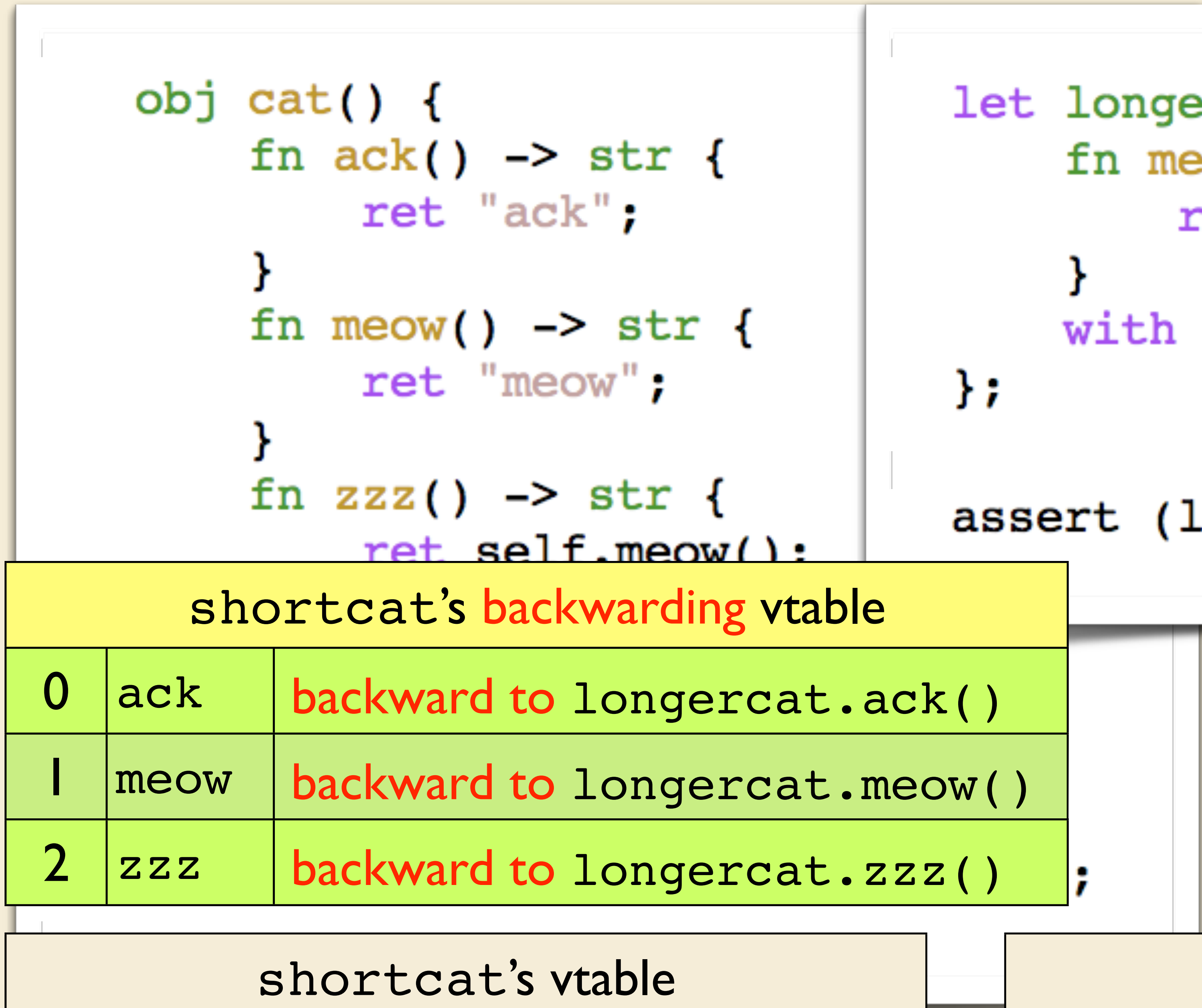


```
let longercat = obj() {
    fn meow() -> str {
        ret "zzz";
    }
    with shortcat
};

assert (longercat.zzz() == "zzz");
```

shortcat's vtable		
0	ack	ret "ack"
1	meow	ret "meow"
2	zzz	ret self.meow()

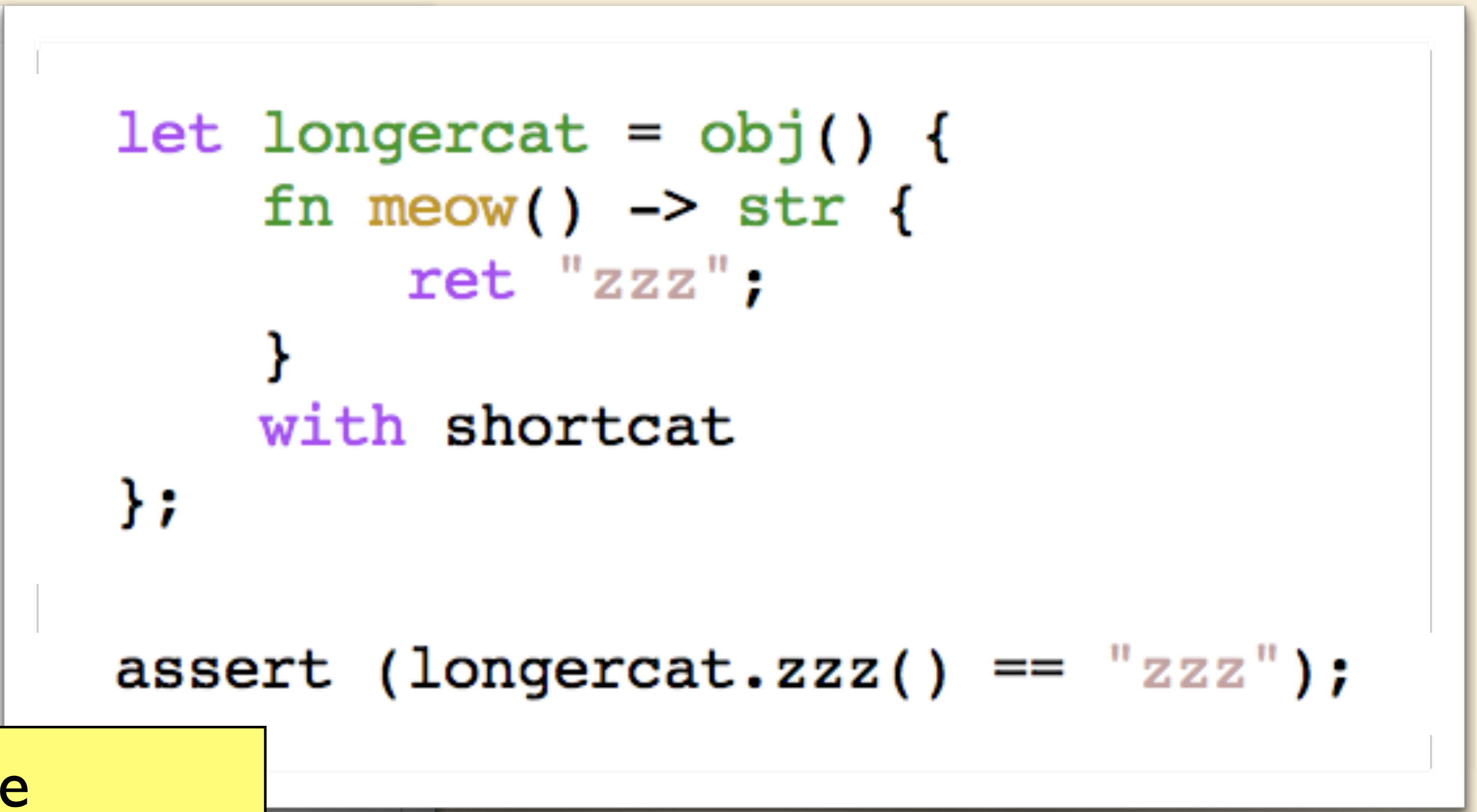
longercat's vtable		
0	ack	forward to shortcat.ack()
1	meow	ret "zzz"
2	zzz	forward to shortcat.zzz()

Self-dispatch + object extension + overriding



```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
```

```
;
```



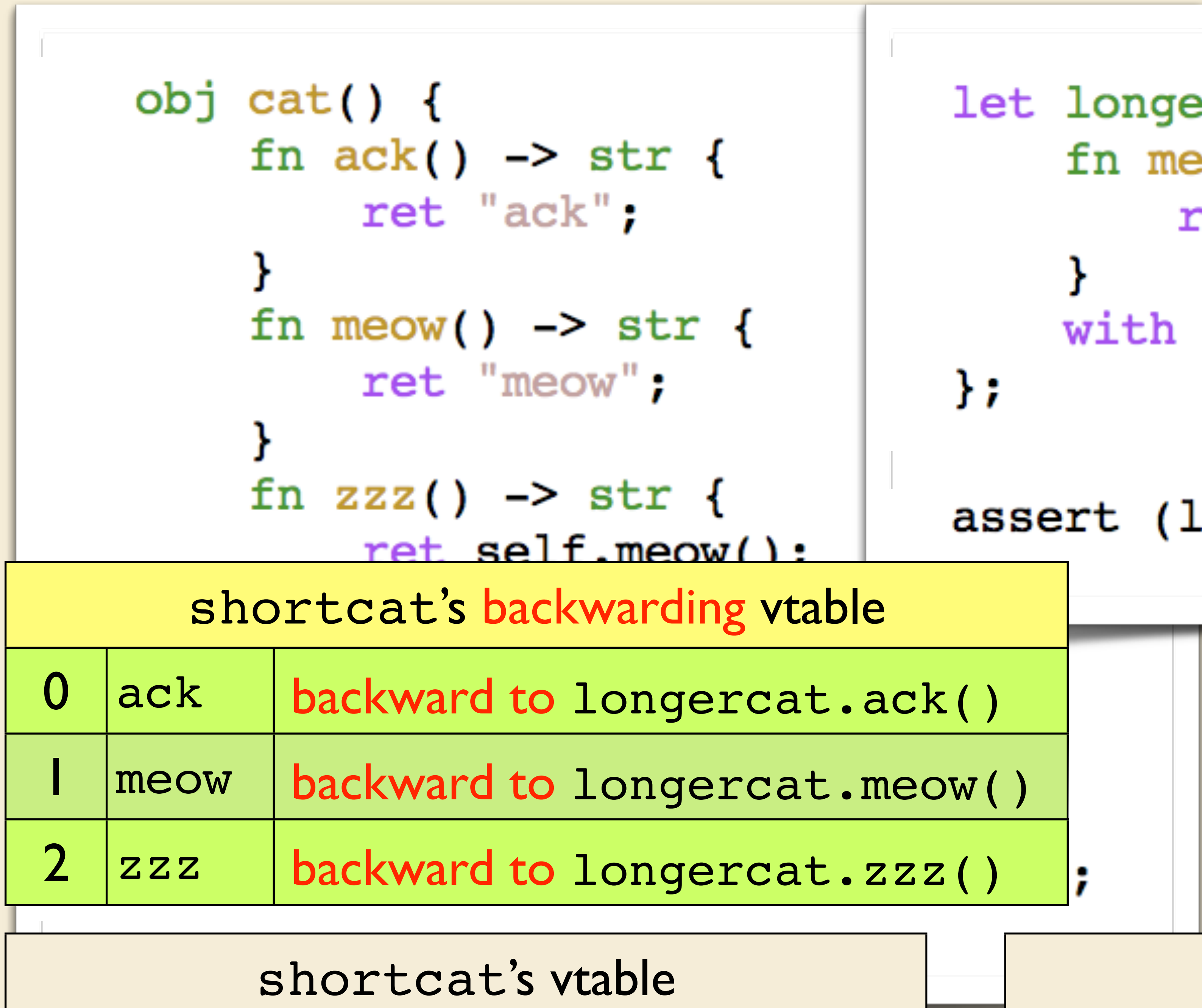
```
let longercat = obj() {
    fn meow() -> str {
        ret "zzz";
    }
    with shortcat
};

assert (longercat.zzz() == "zzz");
```

shortcat's backwarding vtable		
0	ack	backward to longercat.ack()
1	meow	backward to longercat.meow()
2	zzz	backward to longercat.zzz()

shortcat's vtable		
0	ack	ret "ack"
1	meow	ret "meow"
2	zzz	ret self.meow()

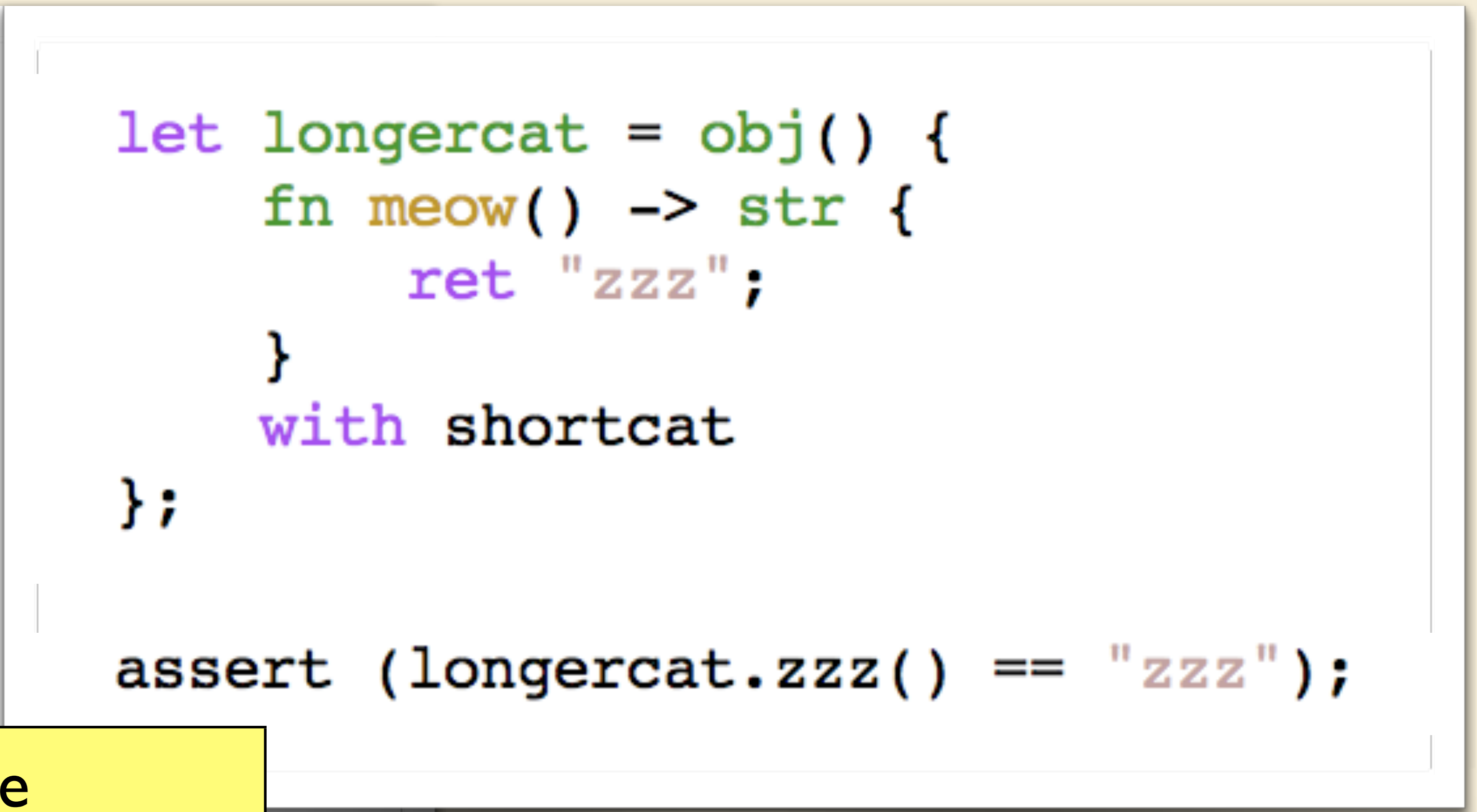
longercat's vtable		
0	ack	forward to shortcat.ack()
1	meow	ret "zzz"
2	zzz	forward to shortcat.zzz()

Self-dispatch + object extension + overriding



```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
```

```
;
```



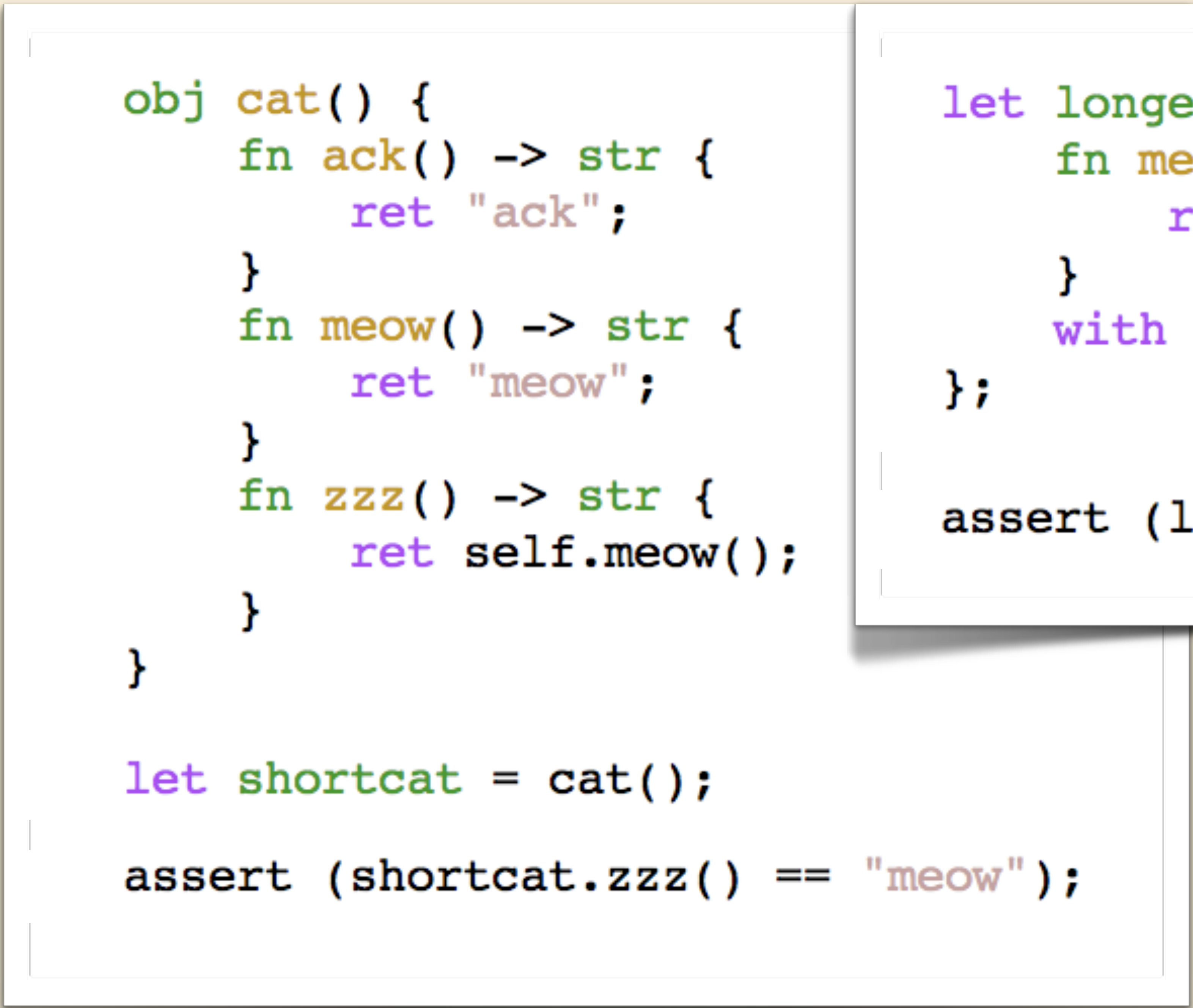
```
let longercat = obj() {
        fn meow() -> str {
            ret "zzz";
        }
        with shortcat
};

assert (longercat.zzz() == "zzz");
```

shortcat's backwarding vtable		
0	ack	backward to longercat.ack()
1	meow	backward to longercat.meow()
2	zzz	backward to longercat.zzz()

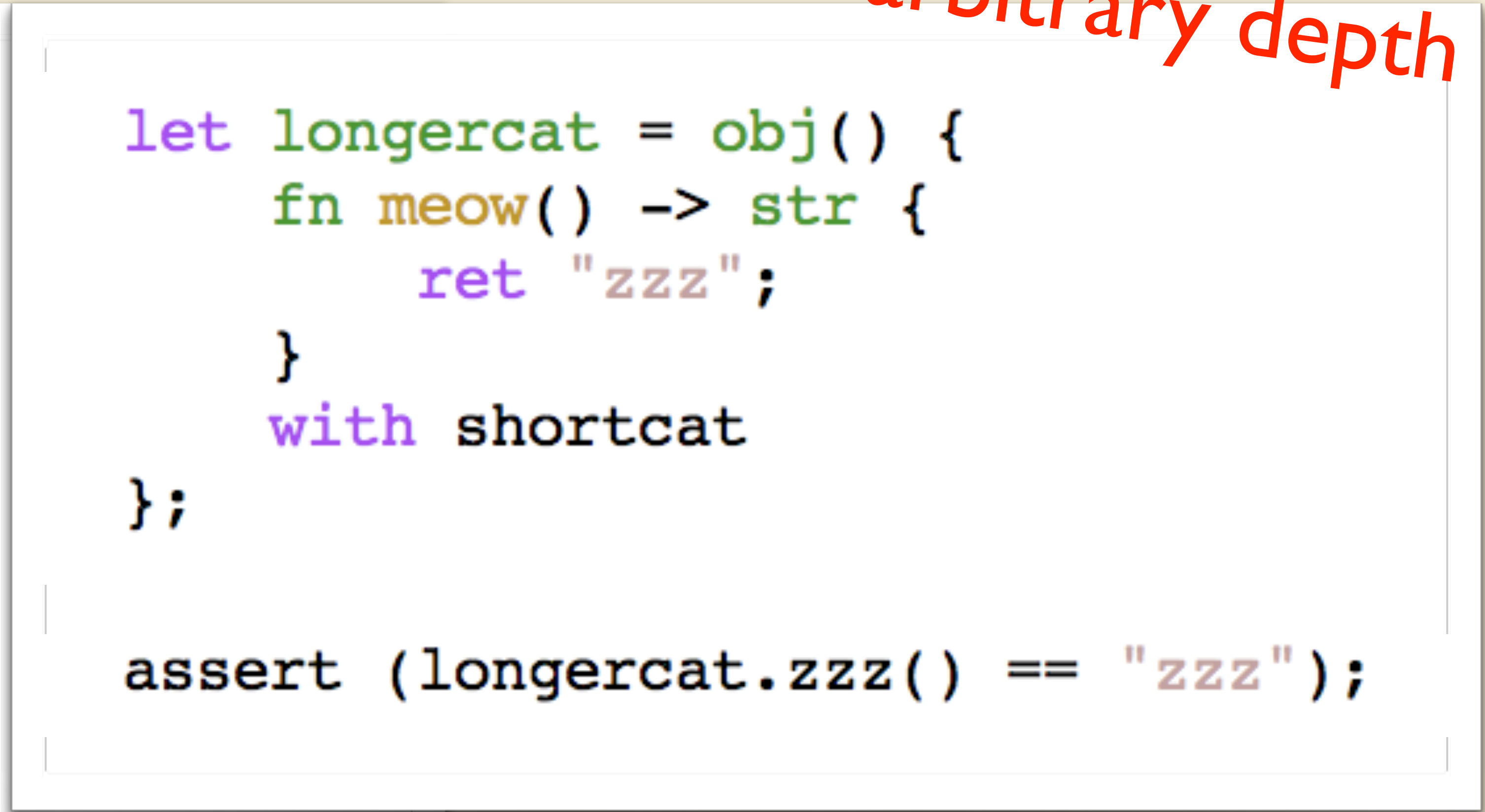
shortcat's vtable		
0	ack	ret "ack"
1	meow	ret "meow"
2	zzz	ret self.meow()

longercat's vtable		
0	ack	forward to shortcat.ack()
1	meow	ret "zzz"
2	zzz	forward to shortcat.zzz()

Self-dispatch + object extension + overriding to arbitrary depth

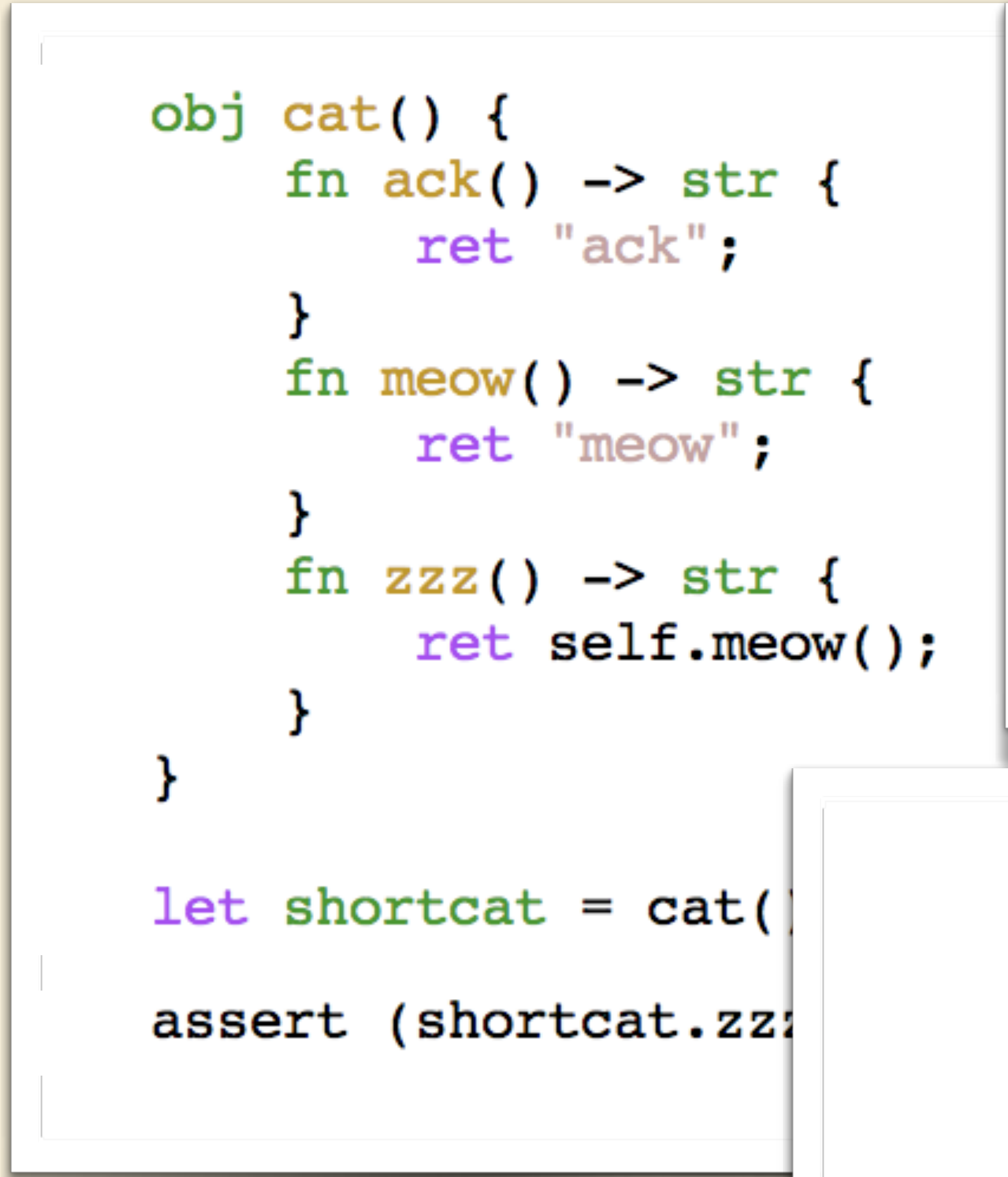


```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat();

assert (shortcat.zzz() == "meow");
```

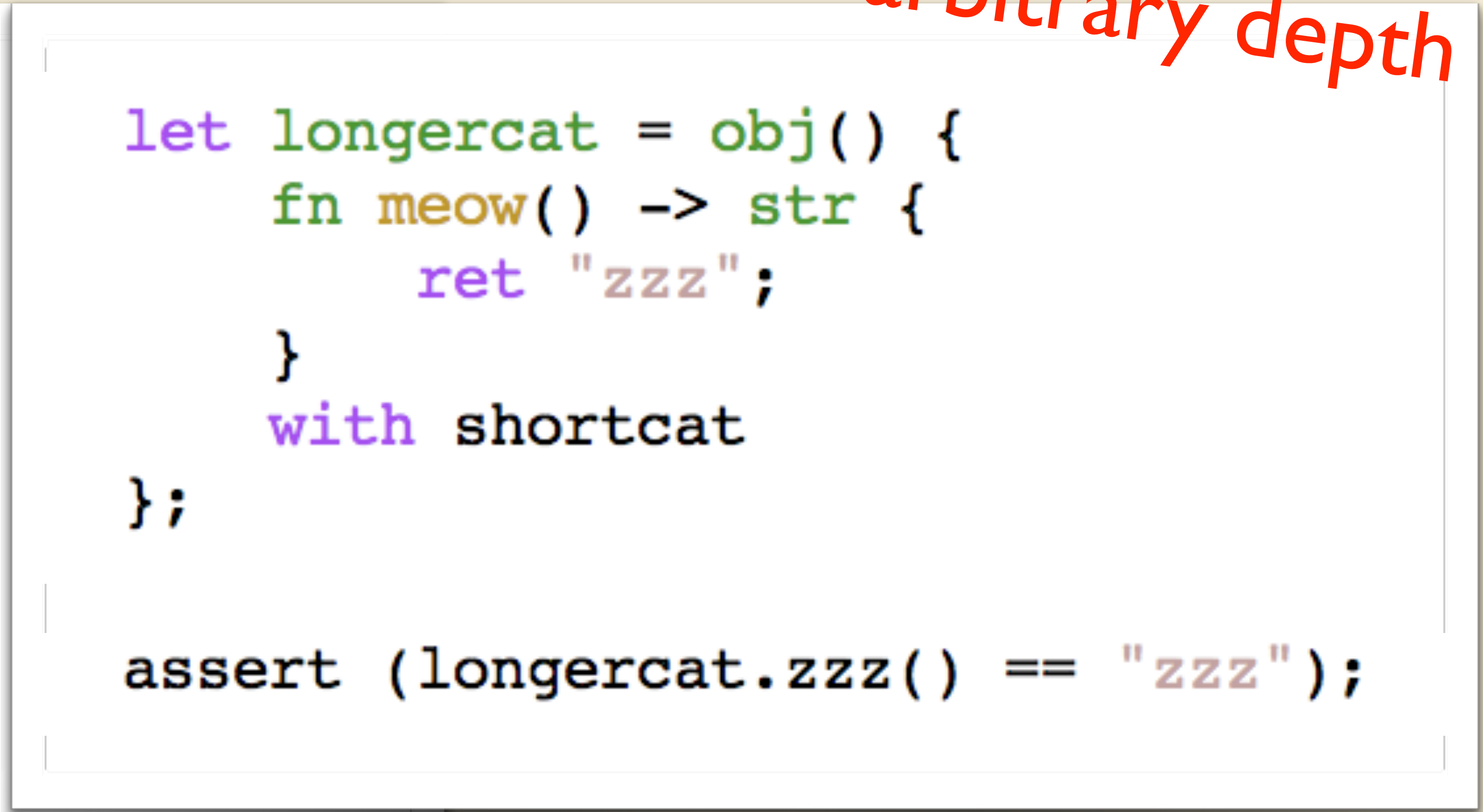


```
let longercat = obj() {
    fn meow() -> str {
        ret "zzz";
    }
    with shortcat
};

assert (longercat.zzz() == "zzz");
```

Self-dispatch + object extension + overriding ^to arbitrary depth



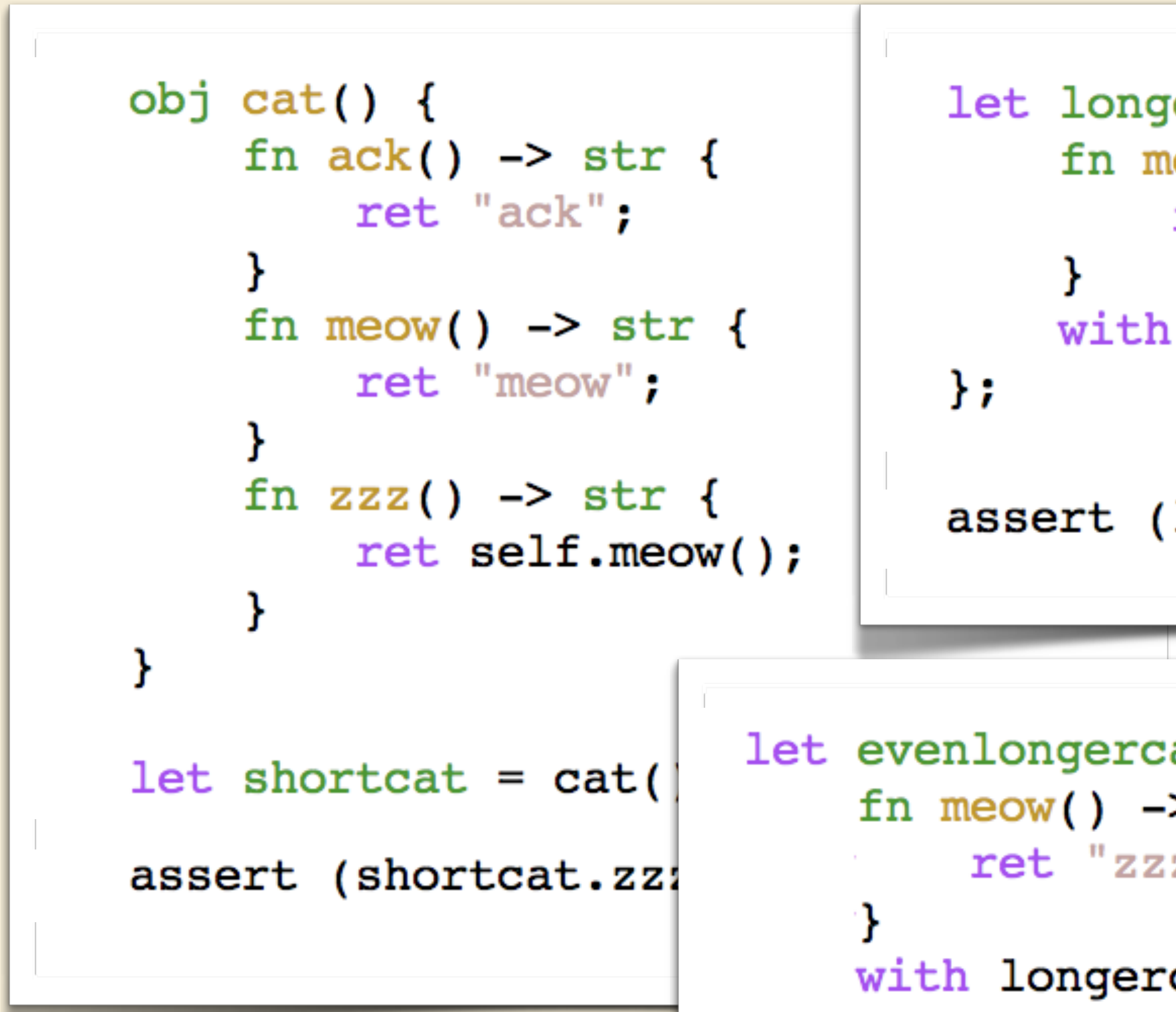
```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat(

assert (shortcat.zz
```



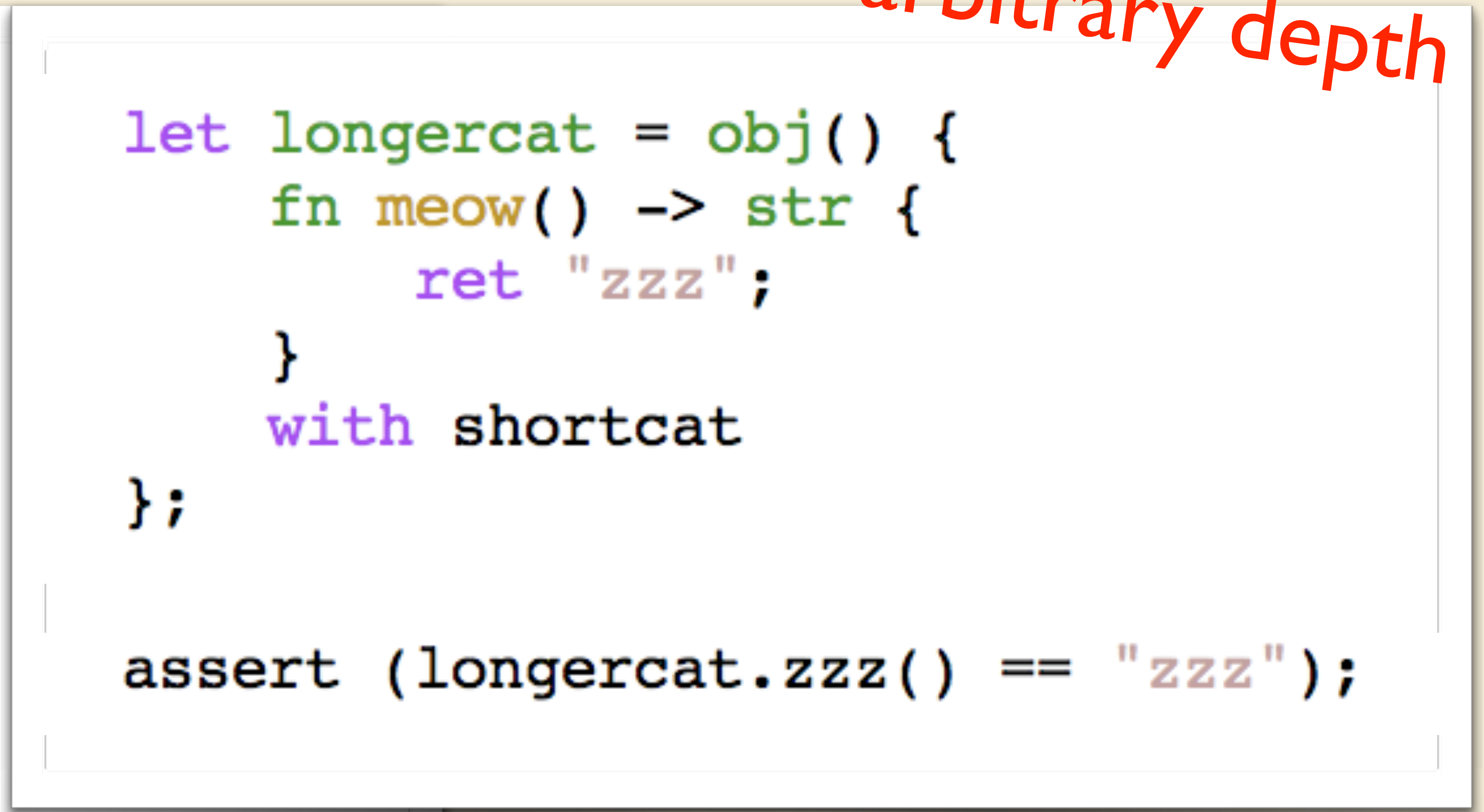
```
let longercat = obj() {
    fn meow() -> str {
        ret "zzz";
    }
    with shortcat
};

assert (longercat.zzz() == "zzz");
```

Self-dispatch + object extension + overriding

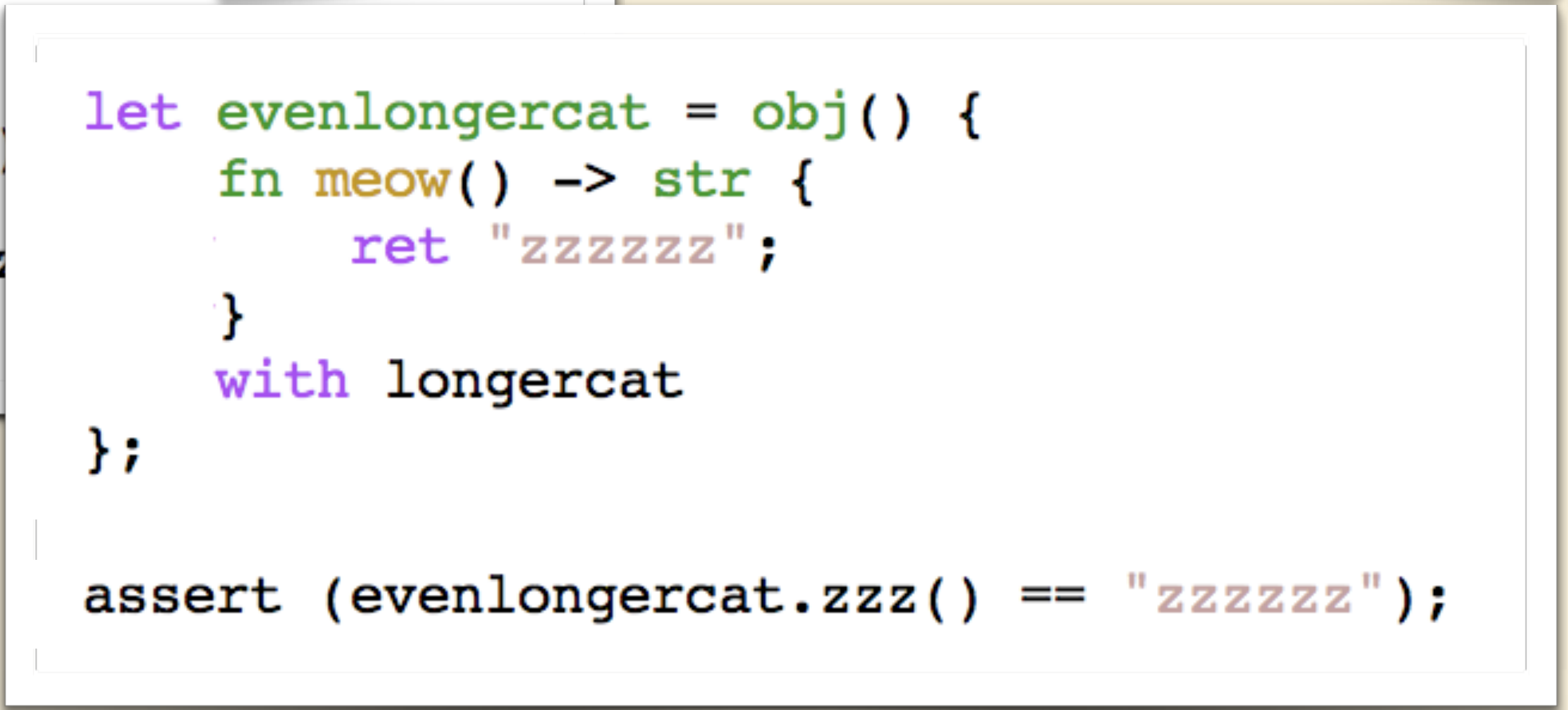
^ to arbitrary depth



```
obj cat() {
    fn ack() -> str {
        ret "ack";
    }
    fn meow() -> str {
        ret "meow";
    }
    fn zzz() -> str {
        ret self.meow();
    }
}

let shortcat = cat(

assert (shortcat.zz
```



```
let longercat = obj() {
    fn meow() -> str {
        ret "zzz";
    }
    with shortcat
};

assert (longercat.zzz() == "zzz");
```



```
let evenlongercat = obj() {
    fn meow() -> str {
        ret "zzzzzz";
    }
    with longercat
};

assert (evenlongercat.zzz() == "zzzzzz");
```

Self-dispatch + object extension + overriding

^ to arbitrary depth

Self-dispatch + object extension + overriding ^ to arbitrary depth

- We need a way to temporarily pretend that self is the inner object, while still keeping track of what the extended self is

Self-dispatch + object extension + overriding ^ to arbitrary depth

- We need a way to temporarily pretend that self is the inner object, while still keeping track of what the extended self is
- Solution: create a stack of "fake selves" threaded through the run-time stack

Self-dispatch + object extension + overriding ^ to arbitrary depth

- We need a way to temporarily pretend that self is the inner object, while still keeping track of what the extended self is
- Solution: create a stack of "fake selves" threaded through the run-time stack
- Every forwarding function allocates space in its frame for a "fake self" comprising a backwarding vtable and an inner object body

Go try it out!

rust-lang.org

Questions?

Thanks to:

Graydon Hoare and everyone on the Rust team

Dave Herman and everyone at Mozilla Research