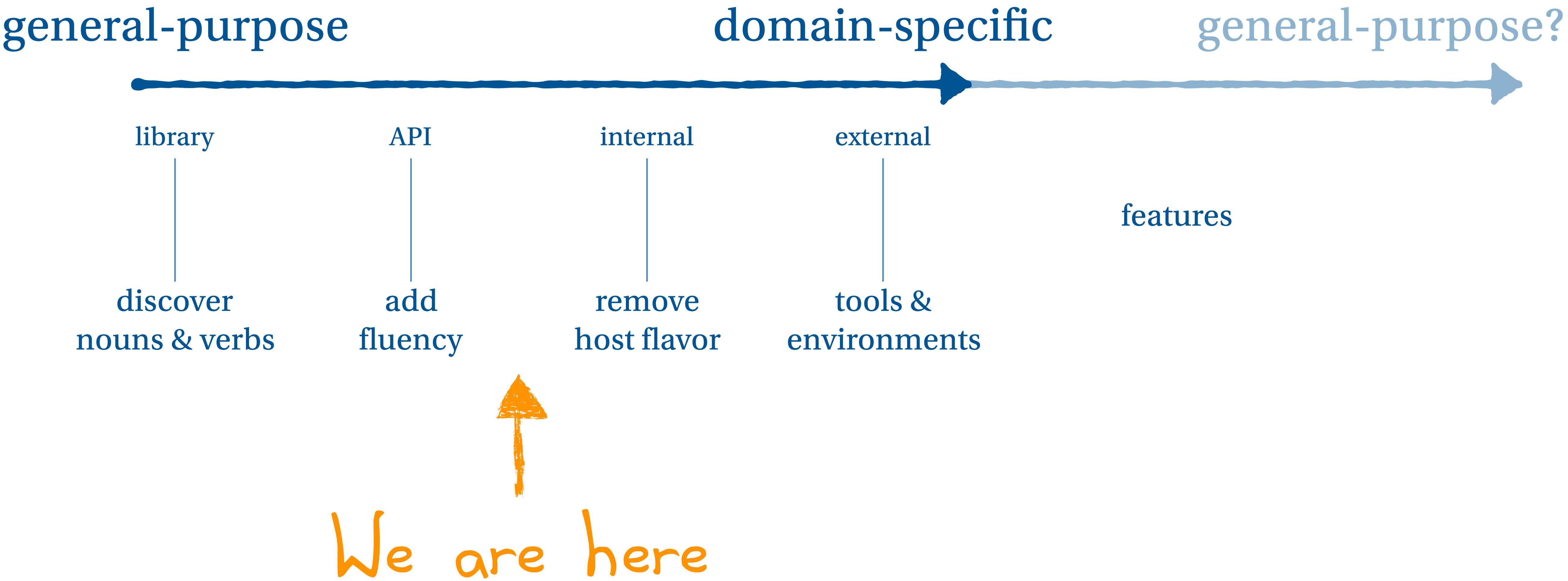
Internal Data Structures

The evolution of a DSL?



general-purpose

domain-specific

general-purpose?

library

discover
nouns & verbs

API

add
fluency

internal

remove
host flavor

external

tools &
environments

features

We are here

Simple techniques for adding fluency

Most general-purpose languages support these features.

names

including Unicode

```
sin(θ)
```

ASK: If the DSL supports Unicode, how will the user write programs?

whitespace

```
computer();
  processor();
    cores(2);
  disk();
    size(150);
```

function composition

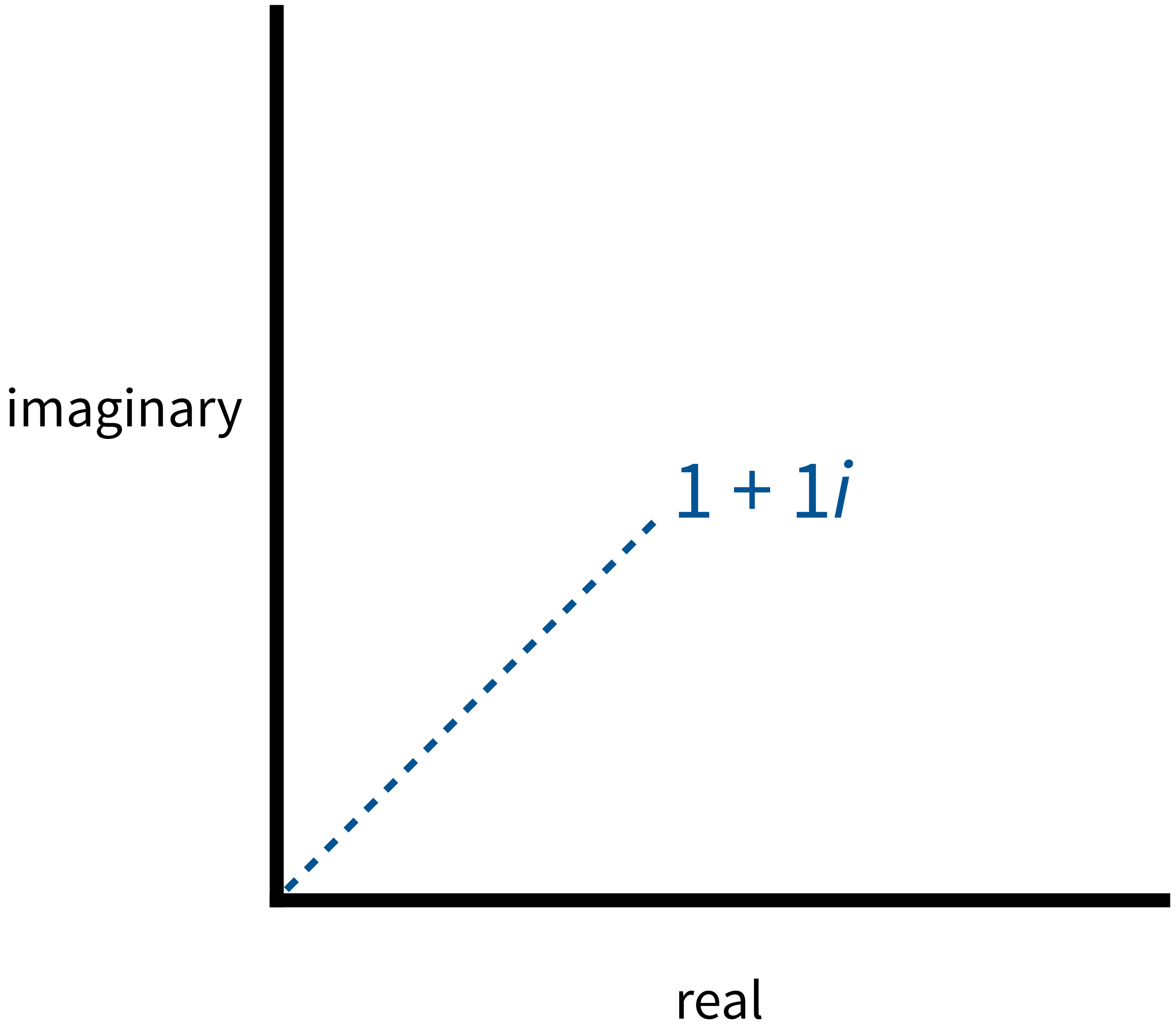
```
computer(
  processor(
    cores(2)
  ),
  disk(
    size(150)
  )
);
```

method chaining

```
computer()
  .processor()
    .cores(2)
  .disk()
    .size(150)
  .end();
```

Is this a DSL?

Complex numbers



$$(a + b\boldsymbol{i}) + (c + d\boldsymbol{i}) = (a + c) + (b + d)\boldsymbol{i}$$

$$(a + b\boldsymbol{i}) * (c + d\boldsymbol{i}) = (ac - bd) + (ad + bc)\boldsymbol{i}$$

Today's goals

- Understand Scala's building blocks for internal DSLs
- Start to recognize these building blocks in other code
- Start thinking about how to use these building blocks to make your own internal DSLs.

Techniques for hiding the host language

These features tend to be language-specific. Some languages support this ability more than others.

(re-)defining operators	`set1 ∪ set2`
	`set1 + set2`
	Different host languages gives us different amounts of control over precedence and associativity.

infix operators	`set1 union set2`
	`salaries map giveRaise`

pre- and postfix operators	`~1` `i++`
literal extension	`3 little pigs`

closures
i.e., by-name parameters in Scala
	`test("An empty Set should have size 0") {`
	`  assert(Set.empty.size == 0)`
	`}`
	Useful for defining new **control-flow structures**

Implicit conversions

See *Scala for the Impatient,* Chapter 21.4

The compiler looks for an implicit conversion when:

- the expected type differs from the inferred type
- an object does not contain an expected attribute

The compiler finds an implicit conversion when:

- a conversion is declared as `implicit`
- a conversion is in scope and is named with a single identifier
- a conversion is defined in the current class's *companion object*

The compiler does not look for an implicit conversion when:

- the code compiles without one
- the compiler has already performed one (for a given expression)
- it finds multiple conversions (i.e., conversion is ambiguous)

```
import scala.language.implicitConversions
```

```
:implicits [-v]
```

```
-Xprint:typer
```