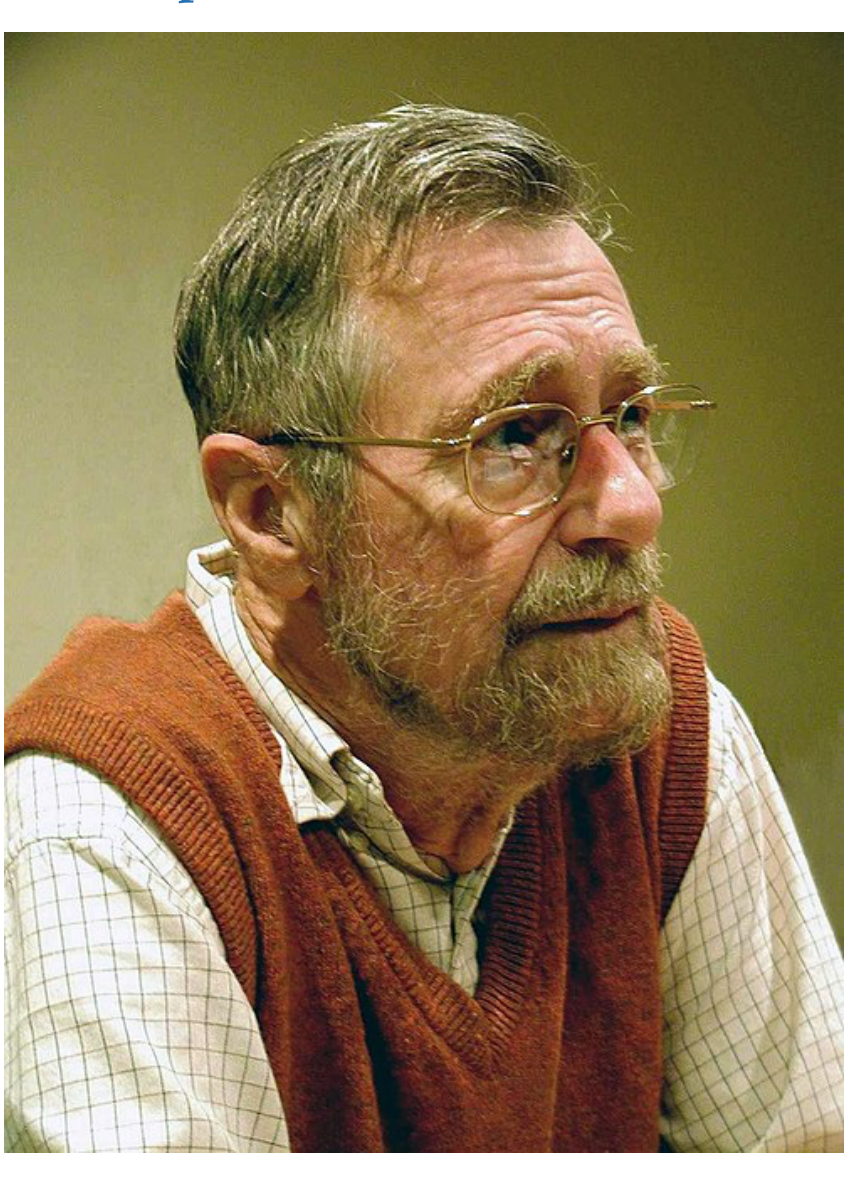
Hello, world!

Hello, World!

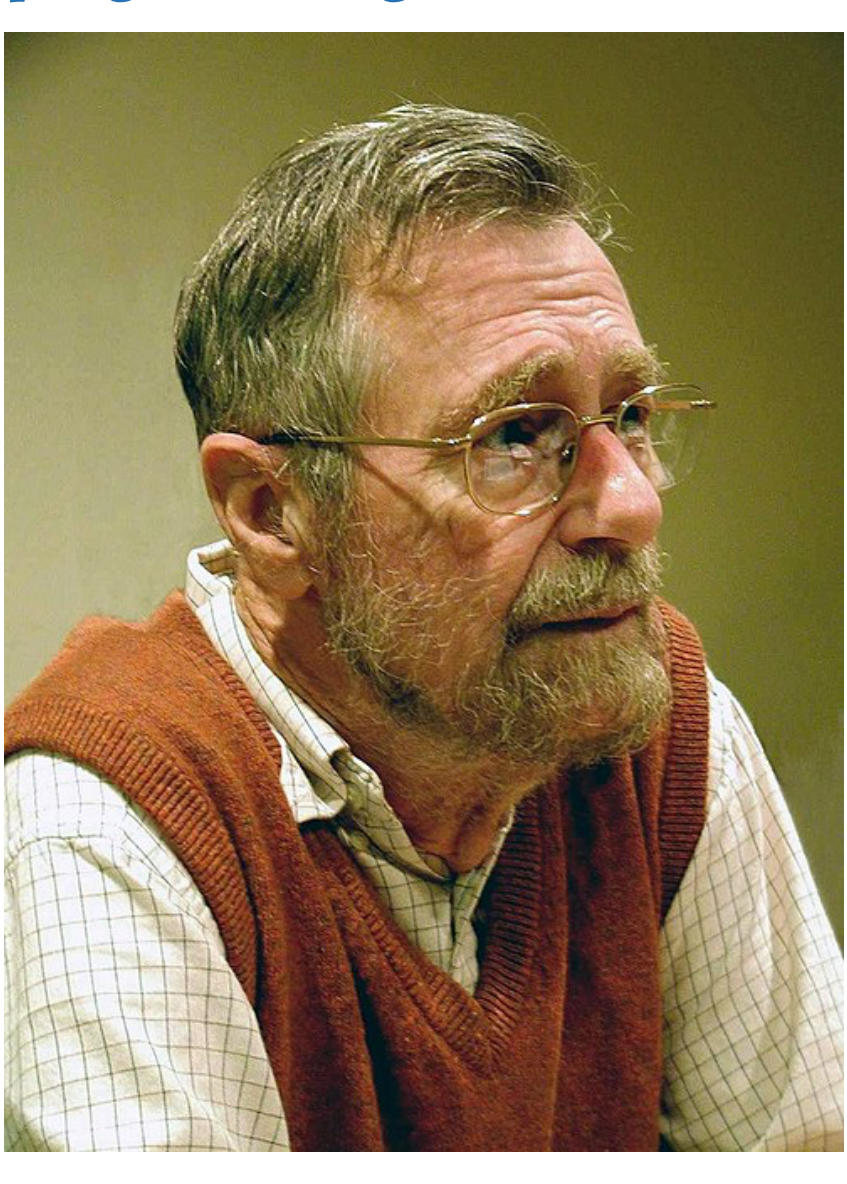
Welcome to CSE 230!

Principles of Programming Languages

Name this Computer Scientist (1)



Edgser Dijkstra "On the foolishness of natural language programming", 1978



> *From one gut feeling I derive much consolation: I suspect that machines to be programmed in our native tongues — be it Dutch, English, American, French, German, or Swahili— are as damned difficult to make as they would be to use.*

1978

Fifty years later …

the elephant in the room

… Won't LLMs like `ChatGPT`, `Claude`, `Grok`, … let us **vibe code** all the programs?

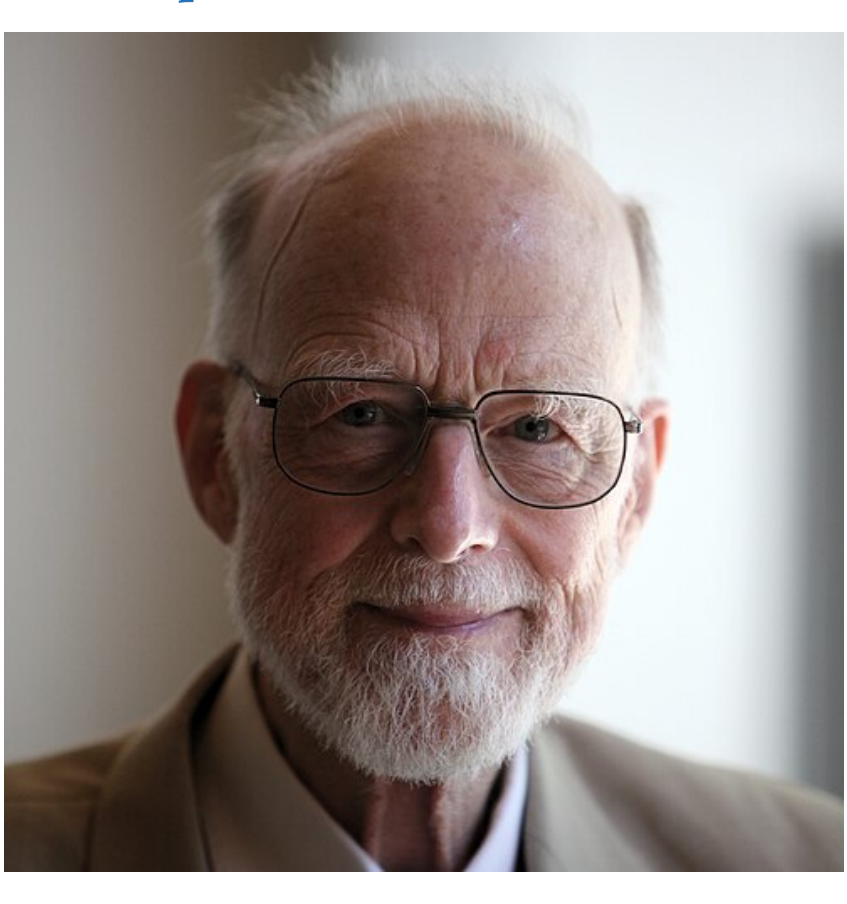
… why bother with Programming Languages?

Computation is *specified* by Programming Languages

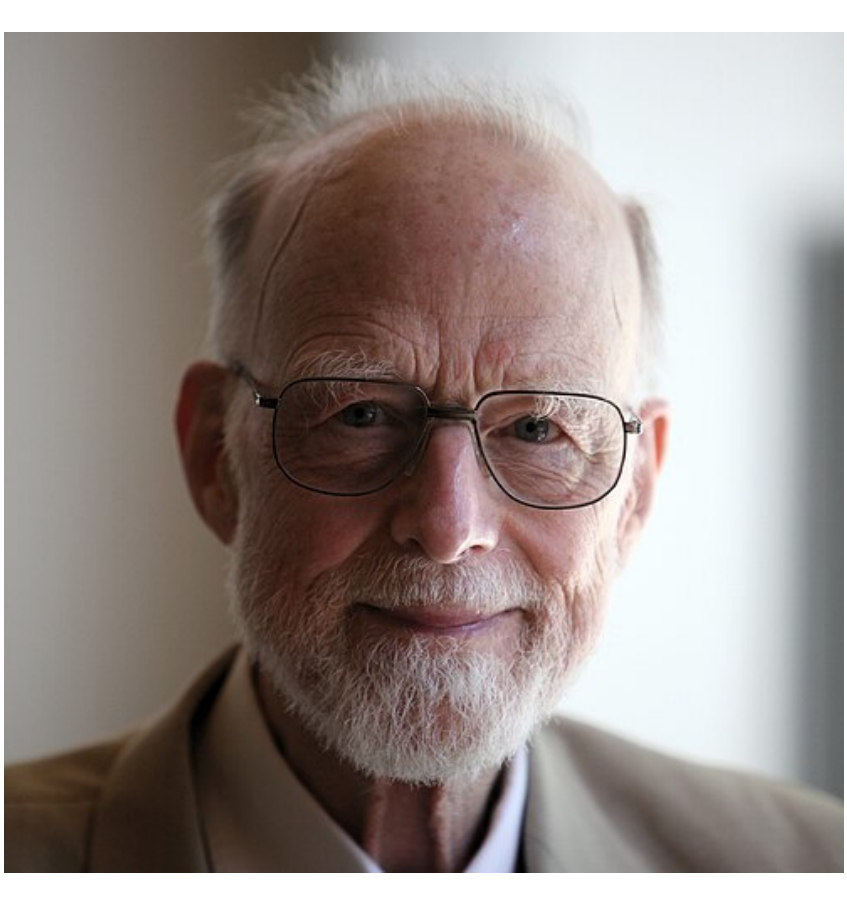
Increased dependence implies increased need for *getting code right*

- **Safety** Will this code *crash*?
- **Security** Will this code broadcast my *social security number*?
- **Performance** Will this code run in the appropriate time/space *constraints*?

Name this Computer Scientist (2)



Sir Tony Hoare

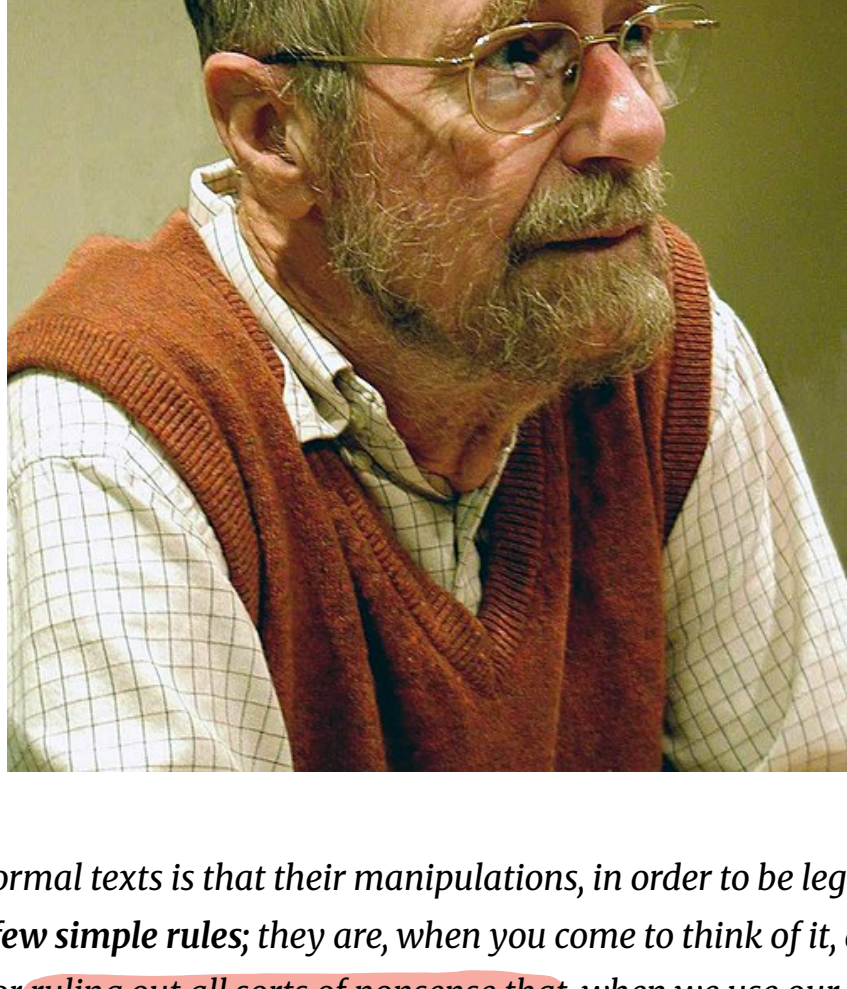


> *There are two ways of constructing a software design: One way is to make it so simple that there are **obviously no deficiencies**, and the other way is to make it so complicated that there are **no obvious deficiencies.***

1980

*How to ensure **obviously no deficiencies** ?*

*How to ensure **obviously no deficiencies** ?*



> *The virtue of formal texts is that their manipulations, in order to be legitimate, **need to satisfy only a few simple rules;** they are, when you come to think of it, an amazingly effective tool for ruling out all sorts of nonsense that, when we use our native tongues, are almost impossible to avoid.*

Name this Computer Scientist (3)

Greg Brockman, President, CTO & Co-Founder, OpenAI

x.com/gdb/status/2007228511363444905

Post

Reply

Greg Brockman
@gdb

Follow

rust is a perfect language for agents, given that if it compiles it's ~correct

3:12 PM · Jan 2, 2026 · **1.2M** Views

477 573 5.5K 1.1K

Not just OpenAI...

Nov 2025

security.googleblog.com/2025/11/rust-in-android-move-fast-fix-things.html

We adopted Rust for its security and are seeing **a 1000x reduction in memory safety vulnerability density compared to Android's C and C++ code**. But the biggest surprise was Rust's impact on software delivery. With Rust changes having **a 4x lower rollback rate** and spending **25% less time in code review**, the safer path is now also the faster one.

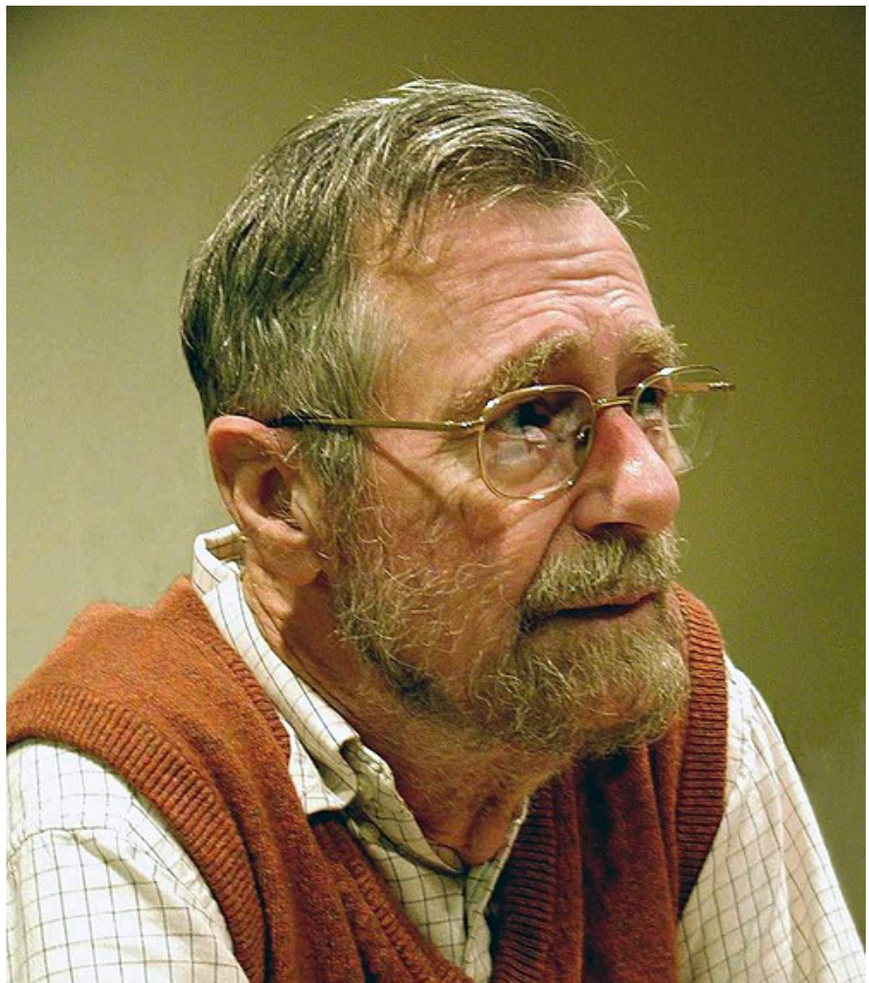
... similar stories from Microsoft, Meta, Amazon, etc.

Learning Goals for CSE 230

Abstraction: Intellectual and Algorithmic tools for reasoning about program behavior

- How to specify what a program **does**?
- How to be sure of what a program **does not**?
- How to ensure *"if it compiles, it is ~correct"*?

Back to Esger Dijkstra...



> *"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be **absolutely precise**."*

Course Outline

Part I: Abstraction Foundations in Haskell

How to create abstractions via types and equational reasoning FP

- Algebraic Data & Functions
- Type classes
- Effects via Monads and Transformers

Part II: Zero-Cost Abstractions in Rust

How to enable zero-cost abstractions via ownership

- Effects via Ownership & Borrowing
- "Fearless" Parallelism & Concurrency
- Property based Testing & Verification

Logistics!

Course Staff

Instructor

- Ranjit Jhala

Teaching Assistants

- Kyle Thompson
- Aronya Baksy

Policies

- No podcasting.

- No screens (phones, laptops) in lecture.
- Yes attendance & class participation (worksheets).
- Yes exams must be done at allotted time and location.
- Yes you may use AI tools for assignments ...

Grading

- Class participation/worksheets (20%)
- Programming Assignments x 4 (20%)
- Midterms x 2 (40%)
- Project (20%)

Class participation (20%)

- "In class" worksheets handed out each lecture
- Handed in at the end of the lecture
- Turn in 75% of the worksheets to get full credit

- Responses will be graded on participation (not correctness)

Programming Assignments (20%)

- **Four** assignments released online
- At least a week before due date
- Via github classroom + codespaces
- **Submitted individually** via github
- Ok to discuss with classmates, but solution must be your own
- Four **late days** used as **whole unit**
- AI use is ok ... but ...

Midterms x 2 (40%)

- Must be done at allotted time and location,
- **Two** midterms (each 20%)

- Closed-book, pencil-and-paper midterm exams,
- During lecture on **Th 2/5** and **~~Th~~ TU 3/10** respectively
- May use a letter-sized **"notes sheet"** of paper

Will heavily test understanding of assignments...

Project (20%)

- Implement a **concurrent**, **real-time**, **multiplayer** game
- Using `tokio + ratatui` (terminal UI library)
- **In a group of 2-3**, in the *second half* of the quarter.
- Evaluated with **live demo and Q&A** ...
- ... during the final exam block on **~~Thu 03/19~~ 8:00a – 11:00a**.

TU 3/17 800-11am

And now, lets begin...!

Haskell Crash Course Part I

Programming in Haskell

Computation by Calculation ABSTRACTION

Substituting equals by equals

Computation via Substituting Equals by Equals

```
(1 + 3) * (4 + 5)      -- subst 1 + 3 = 4
==>
4 * (4 + 5)            -- subst 4 + 5 = 9
==>
4 * 9
==>
36
```

Computation via Substituting Equals by Equals

Equality-Substitution enables **Abstraction** via **Pattern Recognition**

Abstraction via Pattern Recognition

Repeated Expressions

```
31 * (42 + 56)
78 * (12 + 99)
98 * (68 + 12)
```

Recognize Pattern as a *Function Definition*

```
pat   x y z   =  x * (y + z)
```

Instantiate Pattern as a Function Call

```
pat 31 42 56 =*> 31 * (42 + 56) =*> 31 * 98  =*> 3038
pat 78 12 99 =*> 78 * (12 + 99) =*> 78 * 107 =*> 7498
pat 98 68 12 =*> 98 * (68 + 12) =*> 98 * 80  =*> 7280
```

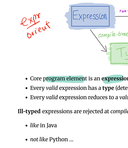
Key Idea: Computation is substitute equals by equals.

Programming in Haskell

Substitute Equals by Equals

Thats it! (*Do not* think of registers, stacks, frames etc.)

Elements of Haskell



- Core program element is an **expression**
- Every *valid* expression has a **type** (determined at compile-time)
- Every *valid* expression reduces to a *value* (computed at run-time)

Ill-typed expressions are rejected at *compile-time* before execution

- *like* in Java
- *not like* Python ...

The Haskell Eco-System

- **Batch compiler**: `ghc` Compile and run large programs
- **Interactive Shell** `ghci` Shell to interactively run small programs online
- **Build Tool** `stack` Build tool to manage libraries etc.

Interactive Shell: `ghci`

```
$ stack ghci

:load file.hs
:type expression
:info variable
```

A Haskell Source File

A sequence of **top-level definitions** `x1`, `x2`,...

- Each has type `type_1`, `type_2`,...
- Each defined by expression `expr_1`, `expr_2`,...

```
x_1 :: type_1
x_1 = expr_1

x_2 :: type_2
x_2 = expr_2

.
.
.
```

Basic Types

```
ex1 :: Int
ex1 = 31 * (42 + 56)     -- this is a comment

ex2 :: Double
ex2 = 3 * (4.2 + 5.6)    -- arithmetic operators "overloaded"

ex3 :: Char
ex3 = 'a'                -- 'a', 'b', 'c', etc. built-in `Char` values

ex4 :: Bool
ex4 = True               -- True, False are builtin Bool values

ex5 :: Bool
ex5 = False
```

QUIZ: Basic Operations

```
ex6 :: Int
ex6 = 4 + 5

ex7 :: Int
ex7 = 4 * 5

ex8 :: Bool
ex8 = 5 > 4

quiz :: ???
quiz = if ex8 then ex6 else ex7
```

What is the type of `quiz` ?

A. `Int`

B. `Bool`

C. Error!

QUIZ: Basic Operations

```
ex6 :: Int
ex6 = 4 + 5

ex7 :: Int
ex7 = 4 * 5

ex8 :: Bool
ex8 = 5 > 4

quiz :: ???
quiz = if ex8 then ex6 else ex7
```

What is the value of `quiz` ?

A. 9

B. 20

C. Other!

Function Types

In Haskell, a **function is a value** that has a type

```
A -> B
```

A function that

- takes input of type `A`
- returns output of type `B`

For example

```
isPos :: Int -> Bool
isPos = (x > 0)
```

Define **function-expressions** using `\` like in *λ-calculus!*

But Haskell also allows us to put the parameter on the *left*

```
isPos :: Int -> Bool
isPos n = (x > 0)
```

(Meaning is **identical** to above definition with `\n -> ...`)

Multiple Argument Functions

A function that

- takes three inputs `A1`, `A2` and `A3`
- returns one output `B` has the type

```
A1 -> A2 -> A3 -> B
```

For example

```
pat :: Int -> Int -> Int -> Int
pat = \x y z -> x * (y + z)
```

which we can write with the params on the *left* as

```
pat :: Int -> Int -> Int -> Int
pat x y z = x * (y + z)
```

QUIZ

What is the type of `quiz` ?

```
quiz :: ???
quiz x y = (x + y) > 0
```

A. `Int -> Int`

B. `Int -> Bool`

C. `Int -> Int -> Int`

D. `Int -> Int -> Bool`

E. `(Int, Int) -> Bool`

Function Calls

A function call is *exactly* like in the *λ*-calculus

```
e1 e2
```

where `e1` is a function and `e2` is the argument. For example

```
>>> isPos 12
True

>>> isPos (0 - 5)
False
```

Multiple Argument Calls

With multiple arguments, just pass them in one by one, e.g.

```
(((e e1) e2) e3)
```

For example

```
>>> pat 31 42 56
3038
```

EXERCISE

Write a function `myMax` that returns the *maximum* of two inputs

```
myMax :: Int -> Int -> Int
myMax = ???
```

When you are done you should see the following behavior:

```
>>> myMax 10 20
20

>>> myMax 100 5
100
```

How to Return Multiple Outputs?

Tuples

A type for packing `n` different kinds of values into a single "struct"

```
(T1,..., Tn)
```

For example

```
tup1 :: ???
tup1 = ('a', 5)

tup2 :: (Char, Double, Int)
tup2 = ('a', 5.2, 7)
```

QUIZ

What is the type `???` of `tup3`?

```
tup3 :: ???
tup3 = ((2, 3.2), True)
```

A. `(Int, Bool)`

B. `(Int, Double, Bool)`

C. `(Int, (Double, Bool))`

D. `((Int, Double), Bool)`

E. `(Tuple, Bool)`

Extracting Values from Tuples

We can **create** a tuple of three values `e1`, `e2`, and `e3` ...

```
tup = (e1, e2, e3)
```

... but how to **extract** the values from this tuple?

Pattern Matching

```
fst3 :: (t1, t2, t3) -> t1
fst3 (x1, x2, x3) = x1

snd3 :: (t1, t2, t3) -> t2
snd3 (x1, x2, x3) = x2

thd3 :: (t1, t2, t3) -> t3
thd3 (x1, x2, x3) = x3
```

QUIZ

What is the value of `quiz` defined as

```
tup2 :: (Char, Double, Int)
tup2 = ('a', 5.2, 7)

snd3 :: (t1, t2, t3) -> t2
snd3 (x1, x2, x3) = x2

quiz = snd3 tup2
```

A. `'a'`

B. `5.2`

C. `7`

D. `('a', 5.2)`

E. `(5.2, 7)`

Lists

Unbounded Sequence of values of type `T`

```
[T]
```

For example

```
chars :: [Char]
chars = ['a','b','c']

ints :: [Int]
ints = [1,2,3,4]

pairs :: [(Int, Bool)]
pairs = [(1,True),(2,False)]
```

QUIZ

What is the type of `things` defined as

```
things = [(1, 'a'), (2, 'b'), (3, 'c')]
```

A. `[Int]`

B. `[[Int], [Int], [Int]]`

C. `[(Int, Int, Int)]`

D. `[[Int]]`

E. `List`

List's Values Must Have The SAME Type!

The type `[T]` denotes an unbounded sequence of values of type `T`

Suppose you have a list

```
oops = [1, 2, 'c']
```

There is no `T` that we can use

- As last element is not `Int`
- First two elements are not `Char` !

Result: Mysterious Type Error!

Constructing Lists

There are two ways to construct lists

```
[]      -- creates an empty list
h:t     -- creates a list with "head" `h` and "tail" t
```

For example

```
>>> 3 : []
[3]

>>> 2 : (3 : [])
[2, 3]

>>> 1 : (2 : (3 : []))
[1, 2, 3]
```

Cons Operator `:` is Right Associative

`x1 : x2 : x3 : x4 : t` means `x1 : (x2 : (x3 : (x4 : t)))`

So we can just avoid the parentheses.

Syntactic Sugar

Haskell lets you write `[x1, x2, x3, x4]` instead of `x1 : x2 : x3 : x4 : []`

Functions Producing Lists

Lets write a function `copy3` that

- takes an input `x` and
- returns a list with three copies of `x`

```
copy3 :: ???
copy3 x = ???
```

When you are done, you should see the following

```
>>> copy3 5
[5, 5, 5]

>>> copy3 "cat"
["cat", "cat", "cat"]
```

PRACTICE: Clone

Write a function `clone` such that `clone n x` returns a list with `n` copies of `x`.

```
clone :: ???
clone n x = ???
```

When you are done you should see the following behavior

```
>>> clone 0 "cat"
[]

>>> clone 1 "cat"
["cat"]

>>> clone 2 "cat"
["cat", "cat"]

>>> clone 3 "cat"
["cat", "cat", "cat"]

>>> clone 3 100
[100, 100, 100]
```

How does `clone` execute?

(Substituting equals-by-equals!)

```
clone 3 100
  ==> ???
```

EXERCISE: Range

Write a function `range` such that `range i j` returns the list of values `[i, i+1, ..., j]`

```
range :: ???
range i j = ???
```

When we are done you should get the behavior

```
>>> range 4 3
[]

>>> range 3 3
[3]

>>> range 2 3
[2, 3]

>>> range 1 3
[1, 2, 3]

>>> range 0 3
[0, 1, 2, 3]
```

Functions Consuming Lists

So far: how to *produce* lists.

Next how to *consume* lists!

Example

Lets write a function `firstElem` such that `firstElem xs` returns the *first* element `xs` if it is a non-empty list, and `0` otherwise.

```
firstElem :: [Int] -> Int
firstElem xs = ???
```

When you are done you should see the following behavior:

```
>>> firstElem []
0

>>> firstElem [10, 20, 30]
10

>>> firstElem [5, 6, 7, 8]
5
```

QUIZ

Suppose we have the following `mystery` function

```
mystery :: [a] -> Int
mystery []     = 0
mystery (x:xs) = 1 + mystery xs
```

What does `mystery [10, 20, 30]` evaluate to?

A. `10`

B. `20`

C. `30`

D. `3`

E. `0`

EXERCISE: Summing a List

Write a function `sumList` such that `sumList [x1, ..., xn]` returns `x1 + ... + xn`

```
sumList :: [Int] -> Int
sumList = ???
```

When you are done you should get the following behavior:

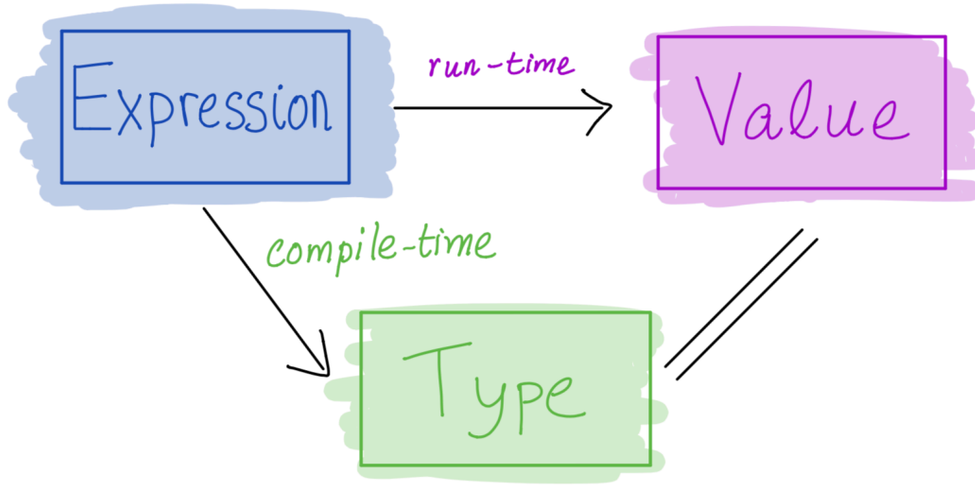
```
>>> sumList []
0

>>> sumlist [3]
3

>>> sumlist [2, 3]
5

>>> sumlist [1, 2, 3]
6
```

Recap



- Core program element is an **expression**
- Every *valid* expression has a **type** (determined at compile-time)
- Every *valid* expression reduces to a *value* (computed at run-time)

Execution

- Basic values & operators
- Execution / Function Calls just *substitute equals by equals*
- Pack data into *tuples* & *lists*
- Unpack data via *pattern-matching*
