A taste of Haskell?

Siddharth Bhat

IIIT Open Source Developers group

October 18th, 2019

What's programming like?

A lot like building a cathedral.

What's programming like?

A lot like building a cathedral.

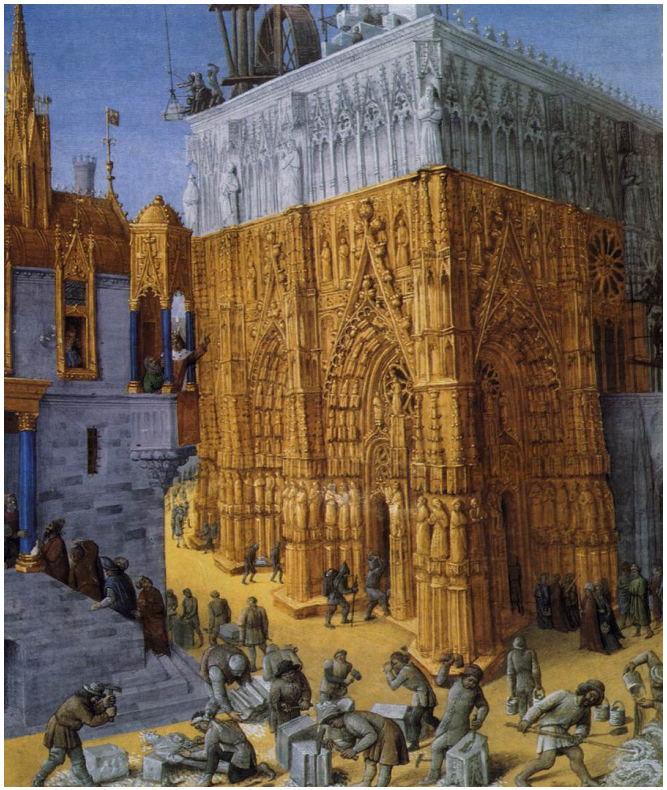


Figure: First we build

What's programming like?

A lot like building a cathedral.

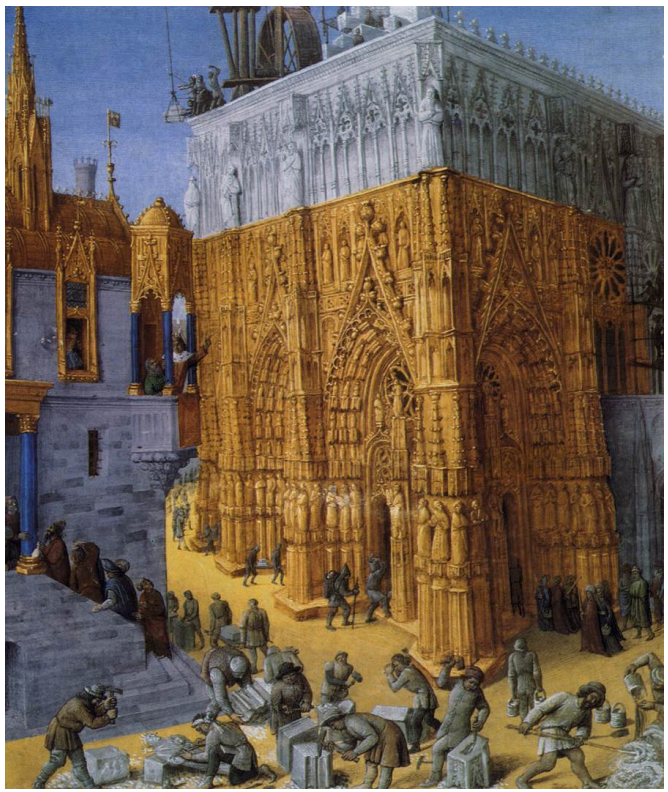


Figure: First we build

Figure: Then we pray

Why we pray

Why we pray

```
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

Why we pray

```
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

```
f(0): 1
```

Why we pray

```cpp
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

```
f(0): 1
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x+1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$

Why we pray

```
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

```
f(0): 1
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x+1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$
- $f(x) \equiv \texttt{true}$

Why we pray

```cpp
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

```
f(0): 1
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x+1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$
- $f(x) \equiv \texttt{true}$

Always going to return `true`?

Why we pray

```
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

```
f(0): 1
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x+1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$
- $f(x) \equiv \texttt{true}$

Always going to return `true`?

```
int main() {
    cout << "f(UINT32_MAX): " << f(UINT32_MAX) << "\n";
}
```

Why we pray

```
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

```
f(0): 1
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x+1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$
- $f(x) \equiv \texttt{true}$

Always going to return `true`?

```
int main() {
    cout << "f(UINT32_MAX): " << f(UINT32_MAX) << "\n";
}
```

```
f(UINT32_MAX):0
```

Why we pray

```
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

```
f(0): 1
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x+1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$
- $f(x) \equiv \texttt{true}$

Always going to return `true`?

```
int main() {
    cout << "f(UINT32_MAX): " << f(UINT32_MAX) << "\n";
}
```

```
f(UINT32_MAX):0
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x +_{2^{32}} 1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$

Why we pray

```
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

```
f(0): 1
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x+1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$
- $f(x) \equiv \texttt{true}$

Always going to return `true`?

```
int main() {
    cout << "f(UINT32_MAX): " << f(UINT32_MAX) << "\n";
}
```

```
f(UINT32_MAX):0
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x +_{2^{32}} 1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$
- $+_{2^32} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}; x +_{2^{32}} y \equiv (x +_{\mathbb{N}} y) \% 2^{32}$

Why we pray

```
#include <iostream>
#include <climits>
using namespace std;

// f(x) == true ?
bool f(unsigned x) { return (x + 1) > x; }

int main() {
    cout << "f(0): " << f(0) << "\n";
}
```

```
f(0): 1
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x+1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$
- $f(x) \equiv \texttt{true}$

Always going to return `true`?

```
int main() {
    cout << "f(UINT32_MAX): " << f(UINT32_MAX) << "\n";
}
```

```
f(UINT32_MAX):0
```

- $f : \mathbb{N} \to \mathbb{B}; f(x) \equiv \begin{cases} \texttt{true} & x +_{2^{32}} 1 > x \\ \texttt{false} & \text{otherwise} \end{cases}$
- $+_{2^32} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}; x +_{2^{32}} y \equiv (x +_{\mathbb{N}} y)\%2^{32}$; $\texttt{UINT32\_MAX} +_{2^{32}} 1 = 0$

Why we pray: A second example

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`
- `getchar` : $\emptyset \rightarrow$ `int`

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`
- `getchar` : $\emptyset \rightarrow$ `int`
- $\{\} \rightarrow \{1, 42, \ldots\}$

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`
- `getchar` : $\emptyset \rightarrow$ `int`
- $\{\} \rightarrow \{1, 42, \ldots\}$
- Such a function can't return an output!

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`
- `getchar` : $\emptyset \to$ `int`
- $\{\} \to \{1, 42, \ldots\}$
- Such a function can't return an output!
- `getchar` : $\{\star\} \to$ `int`

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`
- `getchar` : $\emptyset \to$ `int`
- $\{\} \to \{1, 42, \ldots\}$
- Such a function can't return an output!
- `getchar` : $\{\star\} \to$ `int`
- $\{\star\} \to \{1, 42, \ldots\}$

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`
- `getchar` : $\emptyset \to$ `int`
- $\{\} \to \{1, 42, \ldots\}$
- Such a function can't return an output!
- `getchar` : $\{\star\} \to$ `int`
- $\{\star\} \to \{1, 42, \ldots\}$
- $\{\star\} \to \{1, 42, \ldots\}\ \star \mapsto 42$

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`
- `getchar` : $\emptyset \to$ `int`
- $\{\} \to \{1, 42, \ldots\}$
- Such a function can't return an output!
- `getchar` : $\{\star\} \to$ `int`
- $\{\star\} \to \{1, 42, \ldots\}$
- $\{\star\} \to \{1, 42, \ldots\}$ $\star \mapsto 42$
- Such a function will always return the same output!

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`
- `getchar` : $\emptyset \to$ `int`
- $\{\} \to \{1, 42, \ldots\}$
- Such a function can't return an output!
- `getchar` : $\{\star\} \to$ `int`
- $\{\star\} \to \{1, 42, \ldots\}$
- $\{\star\} \to \{1, 42, \ldots\}$ $\star \mapsto 42$
- Such a function will always return the same output!
- `getchar` can't be a mathematical function.

Why we pray: A second example

$$\forall x \in \mathbb{Z}, 2 * x = x + x$$

(Also holds in modular arithmetic)

```
int main() {
  cout << 2 * ((int) getchar()) << "\n";
}
```

```
int main() {
  cout << ((int)getchar() + (int)getchar()) << "\n";
}
```

- `int getchar()`
- `getchar` : $\emptyset \to$ `int`
- $\{\} \to \{1, 42, \ldots\}$
- Such a function can't return an output!
- `getchar` : $\{\star\} \to$ `int`
- $\{\star\} \to \{1, 42, \ldots\}$
- $\{\star\} \to \{1, 42, \ldots\}$ $\star \mapsto 42$
- Such a function will always return the same output!
- `getchar` can't be a mathematical function.

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x
```

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x
int main() { cout << K(10, 20) << "\n" cout << 10; }
```

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x

int main() { cout << K(10, 20) << "\n" cout << 10; }
```

K(10, 20) = 10

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x

int main() { cout << K(10, 20) << "\n" cout << 10; }
```

$K(10, 20) = 10$

```
int err() { exit(1); return 0; } // start praying!
int main() { cout << K(10, err()) << "\n" cout << 10; }
```

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x

int main() { cout << K(10, 20) << "\n" cout << 10; }
```

K(10, 20) = 10

```
int err() { exit(1); return 0; } // start praying!
int main() { cout << K(10, err()) << "\n" cout << 10; }
```

K(10, err()) $\neq$ 10

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x

int main() { cout << K(10, 20) << "\n" cout << 10; }
```

K(10, 20) = 10

```
int err() { exit(1); return 0; } // start praying!
int main() { cout << K(10, err()) << "\n" cout << 10; }
```

K(10, err()) $\neq$ 10

- Mathematically, can replace $K(x, y)$ by x.

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x

int main() { cout << K(10, 20) << "\n" cout << 10; }
```

K(10, 20) = 10

```
int err() { exit(1); return 0; } // start praying!
int main() { cout << K(10, err()) << "\n" cout << 10; }
```

K(10, err()) $\neq$ 10

- Mathematically, can replace $K(x, y)$ by x.
- In C(++), impossible.

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x

int main() { cout << K(10, 20) << "\n" cout << 10; }
```

`K(10, 20)` $= 10$

```
int err() { exit(1); return 0; } // start praying!
int main() { cout << K(10, err()) << "\n" cout << 10; }
```

`K(10, err())` $\neq 10$

- Mathematically, can replace $K(x, y)$ by x.
- In C(++), impossible.
- cannot **equationally reason** about programs.

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x

int main() { cout << K(10, 20) << "\n" cout << 10; }
```

K(10, 20) = 10

```
int err() { exit(1); return 0; } // start praying!
int main() { cout << K(10, err()) << "\n" cout << 10; }
```

K(10, err()) $\neq$ 10

- Mathematically, can replace $K(x, y)$ by x.
- In C(++), impossible.
- cannot **equationally reason** about programs.
- **Can we** reason about programs?

Why we pray: A third example

```
int K (int x, int y) { return x; } // K(x, y) = x

int main() { cout << K(10, 20) << "\n" cout << 10; }
```

K(10, 20) = 10

```
int err() { exit(1); return 0; } // start praying!
int main() { cout << K(10, err()) << "\n" cout << 10; }
```

K(10, err()) $\neq$ 10

- Mathematically, can replace $K(x, y)$ by x.
- In C(++), impossible.
- cannot **equationally reason** about programs.
- **Can we** reason about programs?

Can we reason about C++?

Can we reason about C++?

- Short answer: Yes.

Can we reason about C++?

- Short answer: Yes.
- Longer answer: Yes. It's complicated.

Can we reason about C++?

- Short answer: Yes.
- Longer answer: Yes. It's complicated.
- Name dropping: Operational/Denotational semantics.

Can we reason about C++?

- Short answer: Yes.
- Longer answer: Yes. It's complicated.
- Name dropping: Operational/Denotational semantics.
- Name dropping: Hoare logic/Separation logic.

Can we reason about C++?

- Short answer: Yes.
- Longer answer: Yes. It's complicated.
- Name dropping: Operational/Denotational semantics.
- Name dropping: Hoare logic/Separation logic.

How do we escape having to pray?

How do we escape having to pray?

Define a language,

How do we escape having to pray?

Define a language, where anything that happens,

How do we escape having to pray?

Define a language, where anything that happens, is what **mathematics** says should happen.

How do we escape having to pray?

Define a language, where anything that happens, is what **mathematics** says should happen.

A taste of Haskell

- `let xs = [1, 2,..10]`

A taste of Haskell

- `let xs = [1, 2,..10]`
- `xs = range(10)`

A taste of Haskell

- `let xs = [1, 2,..10]`
- `xs = range(10)`
- `let nats = [1, 2..]`

A taste of Haskell

- `let xs = [1, 2,..10]`
- `xs = range(10)`
- `let nats = [1, 2..]`
- `xs = ??`

A taste of Haskell

- `let xs = [1, 2,..10]`
- `xs = range(10)`
- `let nats = [1, 2..]`
- `xs = ??`
- `take 2 nats`

A taste of Haskell

- `let xs = [1, 2,..10]`
- `xs = range(10)`
- `let nats = [1, 2..]`
- `xs = ??`
- `take 2 nats`
- `take 10 nats`

A taste of Haskell

- `let xs = [1, 2,..10]`
- `xs = range(10)`
- `let nats = [1, 2..]`
- `xs = ??`
- `take 2 nats`
- `take 10 nats`
- `take 100 nats`

A taste of Haskell

- `let xs = [1, 2,..10]`
- `xs = range(10)`
- `let nats = [1, 2..]`
- `xs = ??`
- `take 2 nats`
- `take 10 nats`
- `take 100 nats`
- Haskell is *lazily evaluated*

A taste of Haskell

- `let xs = [1, 2,..10]`
- `xs = range(10)`
- `let nats = [1, 2..]`
- `xs = ??`
- `take 2 nats`
- `take 10 nats`
- `take 100 nats`
- Haskell is *lazily evaluated*

Why laziness?

- `let k x y = x`

Why laziness?

- `let k x y = x`
- `k 10 20`

Why laziness?

- `let k x y = x`
- `k 10 20`
- `10 + (error "urk")`

Why laziness?

- `let k x y = x`
- `k 10 20`
- `10 + (error "urk")`
- `k 10 (error "urk")`

Why laziness?

- `let k x y = x`
- `k 10 20`
- `10 + (error "urk")`
- `k 10 (error "urk") = 10`
- Laziness provides *equational reasoning*

Why laziness?

- `let k x y = x`
- `k 10 20`
- `10 + (error "urk")`
- `k 10 (error "urk") = 10`
- Laziness provides *equational reasoning*

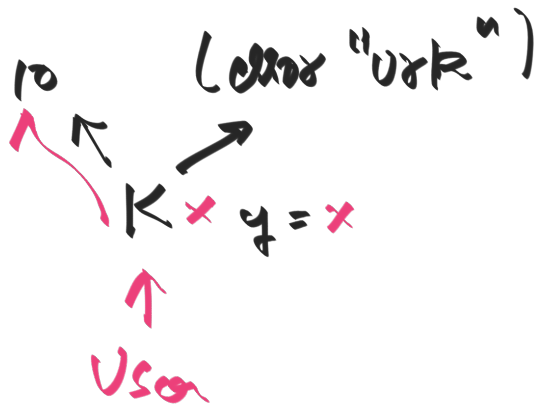


Figure: Evaluation of `K 10 (error "urk")`

Why laziness?

- `let k x y = x`
- `k 10 20`
- `10 + (error "urk")`
- `k 10 (error "urk") = 10`
- Laziness provides *equational reasoning*

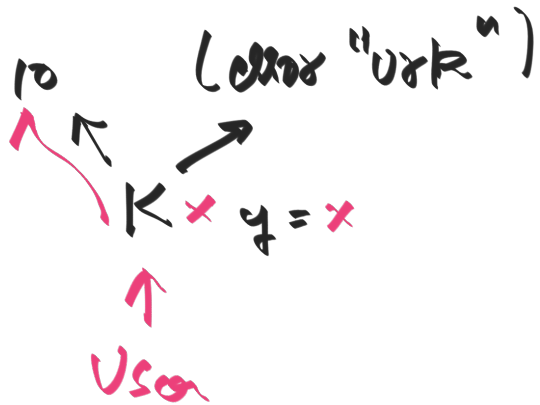


Figure: Evaluation of `K 10 (error "urk")`

Figure: Evaluation of `10 + (error "urk")`

Types

- `:t "foo"`

Types

- `:t "foo"`
- `"foo" :: [Char]`
- `take 2 "foo"`

Types

- `:t "foo"`
- `"foo" :: [Char]`
- `take 2 "foo"`
- `:t take`
- `take :: Int -> [a] -> [a]`

Types

- `:t "foo"`
- `"foo" :: [Char]`
- `take 2 "foo"`
- `:t take`
- `take :: Int -> [a] -> [a]`
- take :: $\forall(\alpha \in \texttt{TYPE}), \texttt{Int} \rightarrow \texttt{List}(\alpha) \rightarrow \texttt{List}(\alpha)$

Types

- `:t "foo"`
- `"foo" :: [Char]`
- `take 2 "foo"`
- `:t take`
- `take :: Int -> [a] -> [a]`
- take :: $\forall(\alpha \in \texttt{TYPE}), \texttt{Int} \rightarrow \texttt{List}(\alpha) \rightarrow \texttt{List}(\alpha)$
- `take 1 'a'`

Types

- `:t "foo"`
- `"foo" :: [Char]`
- `take 2 "foo"`
- `:t take`
- `take :: Int -> [a] -> [a]`
- take :: $\forall(\alpha \in \mathtt{TYPE}), \mathtt{Int} \rightarrow \mathtt{List}(\alpha) \rightarrow \mathtt{List}(\alpha)$
- `take 1 'a'`
- `Prelude> take 1 'a'`

```
<interactive>:11:8: error:
    • Couldn't match expected type ‘[a]’ with actual type ‘Char’
    • In the second argument of ‘take’, namely ‘'a'’
      In the expression: take 1 'a'
      In an equation for ‘it’: it = take 1 'a'
    • Relevant bindings include it :: [a] (bound at <interactive>:11:1)
```

Types

- `:t "foo"`
- `"foo" :: [Char]`
- `take 2 "foo"`
- `:t take`
- `take :: Int -> [a] -> [a]`
- take :: $\forall(\alpha \in \mathtt{TYPE}), \mathtt{Int} \to \mathtt{List}(\alpha) \to \mathtt{List}(\alpha)$
- `take 1 'a'`
- `Prelude> take 1 'a'`

```
<interactive>:11:8: error:
    • Couldn't match expected type ‘[a]’ with actual type ‘Char’
    • In the second argument of ‘take’, namely ‘'a'’
      In the expression: take 1 'a'
      In an equation for ‘it’: it = take 1 'a'
    • Relevant bindings include it :: [a] (bound at <interactive>:11:1)
```

- If $f : A \to B$,

Types

- `:t "foo"`
- `"foo" :: [Char]`
- `take 2 "foo"`
- `:t take`
- `take :: Int -> [a] -> [a]`
- take :: $\forall(\alpha \in \mathtt{TYPE}), \mathtt{Int} \to \mathtt{List}(\alpha) \to \mathtt{List}(\alpha)$
- `take 1 'a'`
- `Prelude> take 1 'a'`

```
<interactive>:11:8: error:
    • Couldn't match expected type ‘[a]’ with actual type ‘Char’
    • In the second argument of ‘take’, namely ‘'a'’
      In the expression: take 1 'a'
      In an equation for ‘it’: it = take 1 'a'
    • Relevant bindings include it :: [a] (bound at <interactive>:11:1)
```

- If $f : A \to B$, can ask $f(a)$ for $a \in A$.

Philosophical differences

Philosophical differences

```
" ".join(["a", "b", "c", "d"])
```

Philosophical differences

```
" ".join(["a", "b", "c", "d"])
```

`str.join(iterable)`

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` `objects`. The separator between elements is the string providing this method.

Philosophical differences

```
intercalate " " ["a", "b", "c", "d"]
```

```
" ".join(["a", "b", "c", "d"])
```

`str.join(iterable)`

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` `objects`. The separator between elements is the string providing this method.

Philosophical differences

```
" ".join(["a", "b", "c", "d"])
```

```
str.join(iterable)
```

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` `objects`. The separator between elements is the string providing this method.

```
intercalate " " ["a", "b", "c", "d"]
```

```
intercalate ::
  String -> [String] -> String
```

Philosophical differences

```
" ".join(["a", "b", "c", "d"])
```

`str.join(iterable)`

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` `objects`. The separator between elements is the string providing this method.

```
intercalate " " ["a", "b", "c", "d"]
```

```
intercalate ::
  String -> [String] -> String
```

`intercalate xs xss` is *equivalent* to `(concat (intersperse xs xss))`. It inserts the list `xs` in between the lists in `xss` and concatenates the result.

Philosophical differences

```
" ".join(["a", "b", "c", "d"])
```

```
str.join(iterable)
```

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` `objects`. The separator between elements is the string providing this method.

```
intercalate " " ["a", "b", "c", "d"]
```

```
intercalate ::
  String -> [String] -> String
```

`intercalate xs xss` is *equivalent* to `(concat (intersperse xs xss))`. It inserts the list `xs` in between the lists in `xss` and concatenates the result.

```
intercalate ::
  [Char] -> [[Char]] -> [Char]
intercalate :: [a] -> [[a]] -> [a]
```

Philosophical differences

```
" ".join(["a", "b", "c", "d"])
```

`str.join(iterable)`

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` objects. The separator between elements is the string providing this method.

```
intercalate " " ["a", "b", "c", "d"]
```

```
intercalate ::
  String -> [String] -> String
```

`intercalate xs xss` is *equivalent* to `(concat (intersperse xs xss))`. It inserts the list `xs` in between the lists in `xss` and concatenates the result.

```
intercalate ::
  [Char] -> [[Char]] -> [Char]
intercalate :: [a] -> [[a]] -> [a]
```

```
intercalate [1, 2]
  [[3,4], [30, 40], [300, 400]]
```

Philosophical differences

```
" ".join(["a", "b", "c", "d"])
```

`str.join(iterable)`

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` objects. The separator between elements is the string providing this method.

```
intercalate " " ["a", "b", "c", "d"]
```

```
intercalate ::
  String -> [String] -> String
```

`intercalate xs xss` is *equivalent* to `(concat (intersperse xs xss))`. It inserts the list `xs` in between the lists in `xss` and concatenates the result.

```
intercalate ::
  [Char] -> [[Char]] -> [Char]
intercalate :: [a] -> [[a]] -> [a]
```

```
intercalate [1, 2]
  [[3,4], [30, 40], [300, 400]]
```

```
= [3, 4, 1, 2, 30, 40, 1, 2, 300, 400]
```

Philosophical differences

```
" ".join(["a", "b", "c", "d"])
```

`str.join(iterable)`

Return a string which is the concatenation of the strings in *iterable*. A `TypeError` will be raised if there are any non-string values in *iterable*, including `bytes` objects. The separator between elements is the string providing this method.

docs.python.org/3/library/stdtypes.html#str.join

```
intercalate " " ["a", "b", "c", "d"]
```

```
intercalate ::
  String -> [String] -> String
```

`intercalate xs xss` is *equivalent* to `(concat (intersperse xs xss))`. It inserts the list `xs` in between the lists in `xss` and concatenates the result.

```
intercalate ::
  [Char] -> [[Char]] -> [Char]
intercalate :: [a] -> [[a]] -> [a]
```

```
intercalate [1, 2]
  [[3,4], [30, 40], [300, 400]]
```

```
= [3, 4, 1, 2, 30, 40, 1, 2, 300, 400]
```

hackage.haskell.org/package/base-4.14.0.0/docs/Data-List.html#v:intercalate

Summing over a list of numbers

```
sum([1, 2, 3, 4])
```

Summing over a list of numbers

```
sum([1, 2, 3, 4])
```

```
sum(iterable, /, start=0)
```

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

Summing over a list of numbers

```
sum([1, 2, 3, 4])
```

```
sum(iterable, /, start=0)
```

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

```
sum [1, 2, 3, 4]
```

Summing over a list of numbers

```
sum([1, 2, 3, 4])
```

```
sum(iterable, /, start=0)
```

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

```
sum [1, 2, 3, 4]
```

```
sum :: (Foldable t, Num a) => t a -> a
sum :: [Int] -> Int]
```

Summing over a list of numbers

```
sum([1, 2, 3, 4])
```

```
sum(iterable, /, start=0)
```

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

```
sum [1, 2, 3, 4]
```

```
sum :: (Foldable t, Num a) => t a -> a
sum :: [Int] -> Int]
```

The `sum` function computes the sum of the numbers of a structure.

Summing over a list of numbers

```
sum([1, 2, 3, 4])
```

```
sum(iterable, /, start=0)
```

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

- `sum [1, 2, 3]`

```
sum [1, 2, 3, 4]
```

```
sum :: (Foldable t, Num a) => t a -> a
sum :: [Int] -> Int]
```

The `sum` function computes the sum of the numbers of a structure.

Summing over a list of numbers

```
sum([1, 2, 3, 4])
```

```
sum(iterable, /, start=0)
```

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

- `sum [1, 2, 3]`
- `sum [1.1, 2.1, -3.2]`

```
sum [1, 2, 3, 4]
```

```
sum :: (Foldable t, Num a) => t a -> a
sum :: [Int] -> Int]
```

The `sum` function computes the sum of the numbers of a structure.

Summing over a list of numbers

```
sum([1, 2, 3, 4])
```

```
sum(iterable, /, start=0)
```

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

- `sum [1, 2, 3]`
- `sum [1.1, 2.1, -3.2]`
- `let minus_1_by_12 = sum [1, 2..]`

```
sum [1, 2, 3, 4]
```

```
sum :: (Foldable t, Num a) => t a -> a
sum :: [Int] -> Int]
```

The `sum` function computes the sum of the numbers of a structure.

Summing over a list of numbers

```
sum([1, 2, 3, 4])
```

```
sum(iterable, /, start=0)
```

Sums *start* and the items of an *iterable* from left to right and returns the total. The *iterable*'s items are normally numbers, and the start value is not allowed to be a string.

- `sum [1, 2, 3]`
- `sum [1.1, 2.1, -3.2]`
- `let minus_1_by_12 = sum [1, 2..]`

https://docs.python.org/3/library/functions.html#sum

```
sum [1, 2, 3, 4]
```

```
sum :: (Foldable t, Num a) => t a -> a
sum :: [Int] -> Int]
```

The `sum` function computes the sum of the numbers of a structure.

hackage.haskell.org/package/base-4.14.0.0/docs/Data-List.html#v:intercalate

Num?

```
sum :: (Foldable t, Num a) => t a -> a
```

Num?

```
sum :: (Foldable t, Num a) => t a -> a
```

```
class  Num a  where -- Typeclass: `a` is a Num if...
```

Num?

```
sum :: (Foldable t, Num a) => t a -> a
class  Num a  where -- Typeclass: `a` is a Num if...
    (+), (-), (*)       :: a -> a -> a
```

Num?

```
sum :: (Foldable t, Num a) => t a -> a

class  Num a  where -- Typeclass: `a` is a Num if...
    (+), (-), (*)       :: a -> a -> a
    -- | Unary negation.
    negate              :: a -> a
    -- | Absolute value.
    abs                 :: a -> a
```

Num?

```
sum :: (Foldable t, Num a) => t a -> a
class  Num a  where -- Typeclass: `a` is a Num if...
    (+), (-), (*)       :: a -> a -> a
    -- | Unary negation.
    negate              :: a -> a
    -- | Absolute value.
    abs                 :: a -> a
    -- | Sign of a number.
    -- The functions 'abs' and 'signum' should satisfy the law:
    --
    -- > abs x * signum x == x
    --
    -- For real numbers, the 'signum' is either `-1` (negative), `0` (zero)
    -- or `1` (positive).
    ...
```

Num?

```
sum :: (Foldable t, Num a) => t a -> a
class  Num a  where -- Typeclass: `a` is a Num if...
    (+), (-), (*)       :: a -> a -> a
    -- | Unary negation.
    negate              :: a -> a
    -- | Absolute value.
    abs                 :: a -> a
    -- | Sign of a number.
    -- The functions 'abs' and 'signum' should satisfy the law:
    --
    -- > abs x * signum x == x
    --
    -- For real numbers, the 'signum' is either `-1` (negative), `0` (zero)
    -- or `1` (positive).
    ...
```

- Associativity of + `(x + y) + z = x + (y + z)`

Num?

```
sum :: (Foldable t, Num a) => t a -> a
class  Num a  where -- Typeclass: `a` is a Num if...
    (+), (-), (*)       :: a -> a -> a
    -- | Unary negation.
    negate              :: a -> a
    -- | Absolute value.
    abs                 :: a -> a
    -- | Sign of a number.
    -- The functions 'abs' and 'signum' should satisfy the law:
    --
    -- > abs x * signum x == x
    --
    -- For real numbers, the 'signum' is either `-1` (negative), `0` (zero)
    -- or `1` (positive).
    ...
```

- Associativity of + `(x + y) + z = x + (y + z)`
- Commutativity of +: `x + y = y + x`

Num?

```
sum :: (Foldable t, Num a) => t a -> a
class  Num a  where -- Typeclass: `a` is a Num if...
    (+), (-), (*)       :: a -> a -> a
    -- | Unary negation.
    negate              :: a -> a
    -- | Absolute value.
    abs                 :: a -> a
    -- | Sign of a number.
    -- The functions 'abs' and 'signum' should satisfy the law:
    --
    -- > abs x * signum x == x
    --
    -- For real numbers, the 'signum' is either `-1` (negative), `0` (zero)
    -- or `1` (positive).
    ...
```

- Associativity of + `(x + y) + z = x + (y + z)`
- Commutativity of +: `x + y = y + x`
- negate gives the additive inverse: `x + negate x = fromInteger 0`

Num?

```
sum :: (Foldable t, Num a) => t a -> a
class  Num a  where -- Typeclass: `a` is a Num if...
    (+), (-), (*)       :: a -> a -> a
    -- | Unary negation.
    negate              :: a -> a
    -- | Absolute value.
    abs                 :: a -> a
    -- | Sign of a number.
    -- The functions 'abs' and 'signum' should satisfy the law:
    --
    -- > abs x * signum x == x
    --
    -- For real numbers, the 'signum' is either `-1` (negative), `0` (zero)
    -- or `1` (positive).
    ...
```

- Associativity of + `(x + y) + z = x + (y + z)`
- Commutativity of +: `x + y = y + x`
- negate gives the additive inverse: `x + negate x = fromInteger 0`

Num?

```
sum :: (Foldable t, Num a) => t a -> a
class  Num a  where -- Typeclass: `a` is a Num if...
    (+), (-), (*)       :: a -> a -> a
    -- | Unary negation.
    negate              :: a -> a
    -- | Absolute value.
    abs                 :: a -> a
    -- | Sign of a number.
    -- The functions 'abs' and 'signum' should satisfy the law:
    --
    -- > abs x * signum x == x
    --
    -- For real numbers, the 'signum' is either `-1` (negative), `0` (zero)
    -- or `1` (positive).
    ...
```

- Associativity of + `(x + y) + z = x + (y + z)`
- Commutativity of +: `x + y = y + x`
- negate gives the additive inverse: `x + negate x = fromInteger 0`

https://hackage.haskell.org/package/base-4.14.0.0/docs/Prelude.html#t:Num

Foldable?

```
sum :: (Foldable t, Num a) => t a -> a
```

Foldable?

```
sum :: (Foldable t, Num a) => t a -> a
```

An object one can "fold over" / "accumulate answers on".

Foldable?

```
sum :: (Foldable t, Num a) => t a -> a
```

An object one can "fold over" / "accumulate answers on". A list, a set, a binary tree, ...

- `let x = Data.Set.fromList [1, 2, 3]`

Foldable?

```
sum :: (Foldable t, Num a) => t a -> a
```

An object one can "fold over" / "accumulate answers on". A list, a set, a binary tree, ...

- `let x = Data.Set.fromList [1, 2, 3]`
- `fromList [1,2,3]`

Foldable?

```
sum :: (Foldable t, Num a) => t a -> a
```

An object one can "fold over" / "accumulate answers on". A list, a set, a binary tree, ...

- `let x = Data.Set.fromList [1, 2, 3]`
- `fromList [1,2,3]`
- `let y = union x x`

Foldable?

```
sum :: (Foldable t, Num a) => t a -> a
```

An object one can "fold over" / "accumulate answers on". A list, a set, a binary tree, ...

- `let x = Data.Set.fromList [1, 2, 3]`
- `fromList [1,2,3]`
- `let y = union x x`
- `fromList [1,2,3]`

Foldable?

```
sum :: (Foldable t, Num a) => t a -> a
```

An object one can "fold over" / "accumulate answers on". A list, a set, a binary tree, ...

- `let x = Data.Set.fromList [1, 2, 3]`
- `fromList [1,2,3]`
- `let y = union x x`
- `fromList [1,2,3]`
- `sum y`
- 6

Foldable in detail

```
sum :: (Foldable t, Num a) => t a -> a
```

Foldable in detail

```
sum :: (Foldable t, Num a) => t a -> a
```

```
class Foldable t where -- a data structure t is foldable if...
  -- | Map each element of the structure to a monoid, and combine the results.
  foldMap :: Monoid m => (a -> m) -> t a -> m
```

Foldable instances are expected to satisfy the following laws:

```
foldr f z t = appEndo (foldMap (Endo . f) t ) z
foldl f z t = appEndo (getDual (foldMap (Dual . Endo . flip f) t)) z
fold = foldMap id
length = getSum . foldMap (Sum . const  1)
```

https://hackage.haskell.org/package/base-4.14.0.0/docs/Data-Foldable.html#t:Foldable

Philosophical differences: Abstract algebra

- Why do we study linear algebra?

Philosophical differences: Abstract algebra

- Why do we study linear algebra?
- Because many things turn out to be linear algebra!

Philosophical differences: Abstract algebra

- Why do we study linear algebra?
- Because many things turn out to be linear algebra!
- https://math.stackexchange.com/questions/5233/vivid-examples-of-vector-spaces

Philosophical differences: Abstract algebra

- Why do we study linear algebra?
- Because many things turn out to be linear algebra!
- https://math.stackexchange.com/questions/5233/vivid-examples-of-vector-spaces
- Study things at the right level of generality!

Philosophical differences: Abstract algebra

- Why do we study linear algebra?
- Because many things turn out to be linear algebra!
- https://math.stackexchange.com/questions/5233/vivid-examples-of-vector-spaces
- Study things at the right level of generality!
- `Foldable`, `Monoid`, `Num`, `[a]`, ...

Philosophical differences: Abstract algebra

- Why do we study linear algebra?
- Because many things turn out to be linear algebra!
- https://math.stackexchange.com/questions/5233/vivid-examples-of-vector-spaces
- Study things at the right level of generality!
- `Foldable`, `Monoid`, `Num`, `[a]`, ...
- `str.join(iterable)` versus `intercalate :: [a] -> [[a]] -> [a]`

Philosophical differences: Abstract algebra

- Why do we study linear algebra?
- Because many things turn out to be linear algebra!
- https://math.stackexchange.com/questions/5233/vivid-examples-of-vector-spaces
- Study things at the right level of generality!
- `Foldable`, `Monoid`, `Num`, `[a]`, ...
- `str.join(iterable)` versus `intercalate :: [a] -> [[a]] -> [a]`
- What is an iterable anyway?

Philosophical differences: Abstract algebra

- Why do we study linear algebra?
- Because many things turn out to be linear algebra!
- https://math.stackexchange.com/questions/5233/vivid-examples-of-vector-spaces
- Study things at the right level of generality!
- `Foldable`, `Monoid`, `Num`, `[a]`, ...
- `str.join(iterable)` versus `intercalate :: [a] -> [[a]] -> [a]`
- What is an iterable anyway?

Fibonacci, Take 1

- `fib :: Int -> Int`. Produces *n*th fibonacci number
- `fib 0 = 0`

Fibonacci, Take 1

- `fib :: Int -> Int`. Produces *n*th fibonacci number
- `fib 0 = 0`
- `fib 1 = 1`

Fibonacci, Take 1

- `fib :: Int -> Int`. Produces *n*th fibonacci number
- `fib 0 = 0`
- `fib 1 = 1`
- `fib n = fib (n - 1) + fib (n - 2)`

Fibonacci, Take 1

- `fib :: Int -> Int`. Produces *n*th fibonacci number
- `fib 0 = 0`
- `fib 1 = 1`
- `fib n = fib (n - 1) + fib (n - 2)`
- `fibs :: [Int]`

Fibonacci, Take 1

- `fib :: Int -> Int`. Produces *n*th fibonacci number
- `fib 0 = 0`
- `fib 1 = 1`
- `fib n = fib (n - 1) + fib (n - 2)`
- `fibs :: [Int]`
- `fib = ???`

Fibonacci, Take 1

- `fib :: Int -> Int`. Produces *n*th fibonacci number
- `fib 0 = 0`
- `fib 1 = 1`
- `fib n = fib (n - 1) + fib (n - 2)`
- `fibs :: [Int]`
- `fib = ???`
- Empty list: `[]`

Fibonacci, Take 1

- `fib :: Int -> Int`. Produces *n*th fibonacci number
- `fib 0 = 0`
- `fib 1 = 1`
- `fib n = fib (n - 1) + fib (n - 2)`
- `fibs :: [Int]`
- `fib = ???`
- Empty list: `[]`
- append an element using `:` — `[1] = 1:[]`

Fibonacci, Take 1

- `fib :: Int -> Int`. Produces *n*th fibonacci number
- `fib 0 = 0`
- `fib 1 = 1`
- `fib n = fib (n - 1) + fib (n - 2)`
- `fibs :: [Int]`
- `fib = ???`
- Empty list: `[]`
- append an element using `:` — `[1] = 1:[]`
- append an element using `:` — `[1, 2] = 1:2:[]`

Fibonacci, Take 1

- `fib :: Int -> Int`. Produces *n*th fibonacci number
- `fib 0 = 0`
- `fib 1 = 1`
- `fib n = fib (n - 1) + fib (n - 2)`
- `fibs :: [Int]`
- `fib = ???`
- Empty list: `[]`
- append an element using `:` — `[1] = 1:[]`
- append an element using `:` — `[1, 2] = 1:2:[]`
- append an element using `:` — `[1, 2, 3] = 1:2:3:[]`

Fibonacci, Take 1

- `fib :: Int -> Int`. Produces *n*th fibonacci number
- `fib 0 = 0`
- `fib 1 = 1`
- `fib n = fib (n - 1) + fib (n - 2)`
- `fibs :: [Int]`
- `fib = ???`
- Empty list: `[]`
- append an element using `:` — `[1] = 1:[]`
- append an element using `:` — `[1, 2] = 1:2:[]`
- append an element using `:` — `[1, 2, 3] = 1:2:3:[]`
- Convert a list `[0, 1, 2...]` into a list `[fib 0, fib 1, fib 2, ...]`

Fibonacci, Take 1

- `fib :: Int -> Int`. Produces *n*th fibonacci number
- `fib 0 = 0`
- `fib 1 = 1`
- `fib n = fib (n - 1) + fib (n - 2)`
- `fibs :: [Int]`
- `fib = ???`
- Empty list: `[]`
- append an element using `:` — `[1] = 1:[]`
- append an element using `:` — `[1, 2] = 1:2:[]`
- append an element using `:` — `[1, 2, 3] = 1:2:3:[]`
- Convert a list `[0, 1, 2...]` into a list `[fib 0, fib 1, fib 2, ...]`

- `ys = map f xs`
- `xs=[x1, x2 ... xn]` $\Longrightarrow$
- `[f x1, f x2, ..., f xn] = ys`
- `fibs = map fib [0, 1..]`

Fibonacci, Take 1

- `fib :: Int -> Int`. Produces *n*th fibonacci number
- `fib 0 = 0`
- `fib 1 = 1`
- `fib n = fib (n - 1) + fib (n - 2)`
- `fibs :: [Int]`
- `fib = ???`
- Empty list: `[]`
- append an element using `:` — `[1] = 1:[]`
- append an element using `:` — `[1, 2] = 1:2:[]`
- append an element using `:` — `[1, 2, 3] = 1:2:3:[]`
- Convert a list `[0, 1, 2...]` into a list `[fib 0, fib 1, fib 2, ...]`

- `ys = map f xs`
- `xs=[x1, x2 ... xn]` $\Longrightarrow$
- `[f x1, f x2, ..., f xn] = ys`
- `fibs = map fib [0, 1..]`
- `map [] = []`

Fibonacci, Take 1

- `fib :: Int -> Int`. Produces *n*th fibonacci number
- `fib 0 = 0`
- `fib 1 = 1`
- `fib n = fib (n - 1) + fib (n - 2)`
- `fibs :: [Int]`
- `fib = ???`
- Empty list: `[]`
- append an element using `:` — `[1] = 1:[]`
- append an element using `:` — `[1, 2] = 1:2:[]`
- append an element using `:` — `[1, 2, 3] = 1:2:3:[]`
- Convert a list `[0, 1, 2...]` into a list `[fib 0, fib 1, fib 2, ...]`

- `ys = map f xs`
- `xs=[x1, x2 ... xn]` $\Longrightarrow$
- `[f x1, f x2, ..., f xn] = ys`
- `fibs = map fib [0, 1..]`
- `map [] = [] ; map (x:xs) = (f x):(map f xs)`

Fibonacci, Take 1

- `fib :: Int -> Int`. Produces *n*th fibonacci number
- `fib 0 = 0`
- `fib 1 = 1`
- `fib n = fib (n - 1) + fib (n - 2)`
- `fibs :: [Int]`
- `fib = ???`
- Empty list: `[]`
- append an element using `:` — `[1] = 1:[]`
- append an element using `:` — `[1, 2] = 1:2:[]`
- append an element using `:` — `[1, 2, 3] = 1:2:3:[]`
- Convert a list `[0, 1, 2...]` into a list `[fib 0, fib 1, fib 2, ...]`

- `ys = map f xs`
- `xs=[x1, x2 ... xn]` $\Longrightarrow$
- `[f x1, f x2, ..., f xn] = ys`
- `fibs = map fib [0, 1..]`
- `map [] = [] ; map (x:xs) = (f x):(map f xs)`
- `map :: (t1 -> t2) -> [t1] -> [t2]`

Fibonacci, Take 1

- `fib :: Int -> Int`. Produces *n*th fibonacci number
- `fib 0 = 0`
- `fib 1 = 1`
- `fib n = fib (n - 1) + fib (n - 2)`
- `fibs :: [Int]`
- `fib = ???`
- Empty list: `[]`
- append an element using `:` — `[1] = 1:[]`
- append an element using `:` — `[1, 2] = 1:2:[]`
- append an element using `:` — `[1, 2, 3] = 1:2:3:[]`
- Convert a list `[0, 1, 2...]` into a list `[fib 0, fib 1, fib 2, ...]`

- `ys = map f xs`
- `xs=[x1, x2 ... xn]` $\Longrightarrow$
- `[f x1, f x2, ..., f xn] = ys`
- `fibs = map fib [0, 1..]`
- `map [] = [] ; map (x:xs) = (f x):(map f xs)`
- `map :: (t1 -> t2) -> [t1] -> [t2]`

Fibonacci, Take 2

- `fibs` satisfies a recurrence:
- `fibs ! 0 = 0`
- `fibs ! 1 = 1`
- `fibs ! n = fibs ! (n - 1) + fibs ! (n - 2)`

Fibonacci, Take 2

- `fibs` satisfies a recurrence:
- `fibs ! 0 = 0`
- `fibs ! 1 = 1`
- `fibs ! n = fibs ! (n - 1) + fibs ! (n - 2)`

```
[fib 0, fib 1, fib 2, fib 3, fib 4, ..., fib (n-1), fib n]
```

Fibonacci, Take 2

- `fibs` satisfies a recurrence:
- `fibs ! 0 = 0`
- `fibs ! 1 = 1`
- `fibs ! n = fibs ! (n - 1) + fibs ! (n - 2)`

```
[fib 0, fib 1, fib 2, fib 3, fib 4, ..., fib (n-1), fib n]
[fib 1, fib 2, fib 3, fib 4, fib 5, ..., fib (n-2), fib (n-1)]
```

Fibonacci, Take 2

- `fibs` satisfies a recurrence:
- `fibs ! 0 = 0`
- `fibs ! 1 = 1`
- `fibs ! n = fibs ! (n - 1) + fibs ! (n - 2)`

```
  [fib 0, fib 1, fib 2, fib 3, fib 4, ..., fib (n-1), fib n]
  [fib 1, fib 2, fib 3, fib 4, fib 5, ..., fib (n-2), fib (n-1)]
= [fib 2, fib 3, fib 4, fib 5, fib 6, ..., fib  n   , fib (n+1)]
```

Fibonacci, Take 2

- `fibs` satisfies a recurrence:
- `fibs ! 0 = 0`
- `fibs ! 1 = 1`
- `fibs ! n = fibs ! (n - 1) + fibs ! (n - 2)`

```
  [fib 0, fib 1, fib 2, fib 3, fib 4, ..., fib (n-1), fib n]
  [fib 1, fib 2, fib 3, fib 4, fib 5, ..., fib (n-2), fib (n-1)]
= [fib 2, fib 3, fib 4, fib 5, fib 6, ..., fib  n   , fib (n+1)]
```

- `fibs =`

Fibonacci, Take 2

- `fibs` satisfies a recurrence:
- `fibs ! 0 = 0`
- `fibs ! 1 = 1`
- `fibs ! n = fibs ! (n - 1) + fibs ! (n - 2)`

```
  [fib 0, fib 1, fib 2, fib 3, fib 4, ..., fib (n-1), fib n]
  [fib 1, fib 2, fib 3, fib 4, fib 5, ..., fib (n-2), fib (n-1)]
= [fib 2, fib 3, fib 4, fib 5, fib 6, ..., fib  n   , fib (n+1)]
```

- `fibs =`
- `fibs = 0`

Fibonacci, Take 2

- `fibs` satisfies a recurrence:
- `fibs ! 0 = 0`
- `fibs ! 1 = 1`
- `fibs ! n = fibs ! (n - 1) + fibs ! (n - 2)`

```
  [fib 0, fib 1, fib 2, fib 3, fib 4, ..., fib (n-1), fib n]
  [fib 1, fib 2, fib 3, fib 4, fib 5, ..., fib (n-2), fib (n-1)]
= [fib 2, fib 3, fib 4, fib 5, fib 6, ..., fib  n   , fib (n+1)]
```

- `fibs =`
- `fibs = 0`
- `fibs = 0:1`

Fibonacci, Take 2

- `fibs` satisfies a recurrence:
- `fibs ! 0 = 0`
- `fibs ! 1 = 1`
- `fibs ! n = fibs ! (n - 1) + fibs ! (n - 2)`

```
  [fib 0, fib 1, fib 2, fib 3, fib 4, ..., fib (n-1), fib n]
  [fib 1, fib 2, fib 3, fib 4, fib 5, ..., fib (n-2), fib (n-1)]
= [fib 2, fib 3, fib 4, fib 5, fib 6, ..., fib  n   , fib (n+1)]
```

- `fibs =`
- `fibs = 0`
- `fibs = 0:1`
- `fibs = 0:1:fib_rec fibs`

Fibonacci, Take 2

- fibs satisfies a recurrence:
- `fibs ! 0 = 0`
- `fibs ! 1 = 1`
- `fibs ! n = fibs ! (n - 1) + fibs ! (n - 2)`

```
  [fib 0, fib 1, fib 2, fib 3, fib 4, ..., fib (n-1), fib n]
  [fib 1, fib 2, fib 3, fib 4, fib 5, ..., fib (n-2), fib (n-1)]
= [fib 2, fib 3, fib 4, fib 5, fib 6, ..., fib  n   , fib (n+1)]
```

- `fibs =`
- `fibs = 0`
- `fibs = 0:1`
- `fibs = 0:1:fib_rec fibs`
- `fib_rec fibs = zipAdd fibs (tail fibs)`

Fibonacci, Take 2

- fibs satisfies a recurrence:
- `fibs ! 0 = 0`
- `fibs ! 1 = 1`
- `fibs ! n = fibs ! (n - 1) + fibs ! (n - 2)`

```
[fib 0, fib 1, fib 2, fib 3, fib 4, ..., fib (n-1), fib n]
[fib 1, fib 2, fib 3, fib 4, fib 5, ..., fib (n-2), fib (n-1)]
= [fib 2, fib 3, fib 4, fib 5, fib 6, ..., fib  n   , fib (n+1)]
```

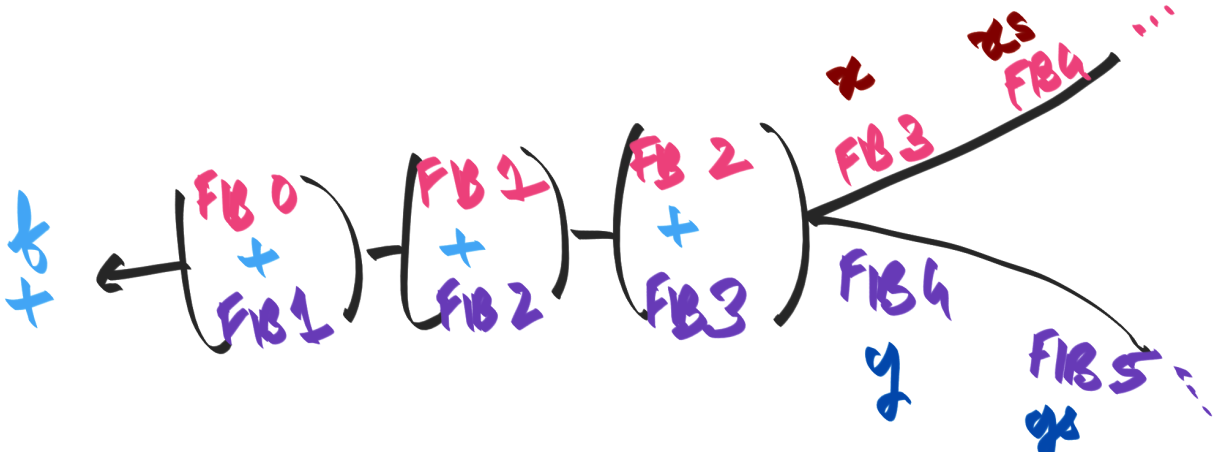
- `fibs =`
- `fibs = 0`
- `fibs = 0:1`
- `fibs = 0:1:fib_rec fibs`
- `fib_rec fibs = zipAdd fibs (tail fibs)`
- `zipAdd (x:xs) (y:ys) = (x+y):(zipAdd xs ys)`

Fibonacci, Take 2

- fibs satisfies a recurrence:
- `fibs ! 0 = 0`
- `fibs ! 1 = 1`
- `fibs ! n = fibs ! (n - 1) + fibs ! (n - 2)`

```
  [fib 0, fib 1, fib 2, fib 3, fib 4, ..., fib (n-1), fib n]
  [fib 1, fib 2, fib 3, fib 4, fib 5, ..., fib (n-2), fib (n-1)]
= [fib 2, fib 3, fib 4, fib 5, fib 6, ..., fib  n   , fib (n+1)]
```

- `fibs =`
- `fibs = 0`
- `fibs = 0:1`
- `fibs = 0:1:fib_rec fibs`
- `fib_rec fibs = zipAdd fibs (tail fibs)`
- `zipAdd (x:xs) (y:ys) = (x+y):(zipAdd xs ys)`



Fibonacci, Take 3

- `fibs = 0:1:fib_rec fibs`

Fibonacci, Take 3

- `fibs = 0:1:fib_rec fibs`
- `fib_rec fibs = zipAdd fibs (tail fibs)`

Fibonacci, Take 3

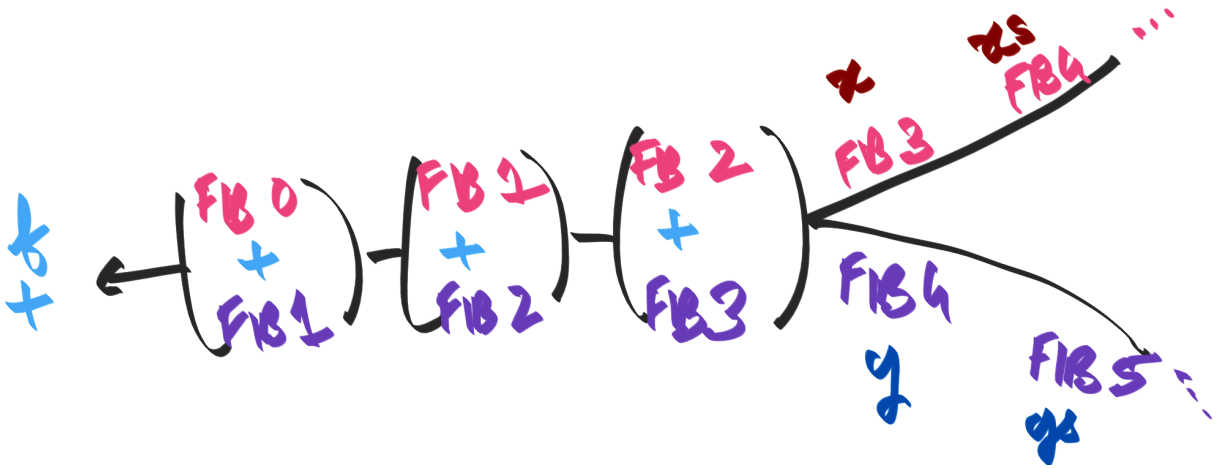
- `fibs = 0:1:fib_rec fibs`
- `fib_rec fibs = zipAdd fibs (tail fibs)`
- `zipAdd (x:xs) (y:ys) = (x+y):(zipAdd xs ys)`

Fibonacci, Take 3

- `fibs = 0:1:fib_rec fibs`
- `fib_rec fibs = zipAdd fibs (tail fibs)`
- `zipAdd (x:xs) (y:ys) = (x+y):(zipAdd xs ys)`

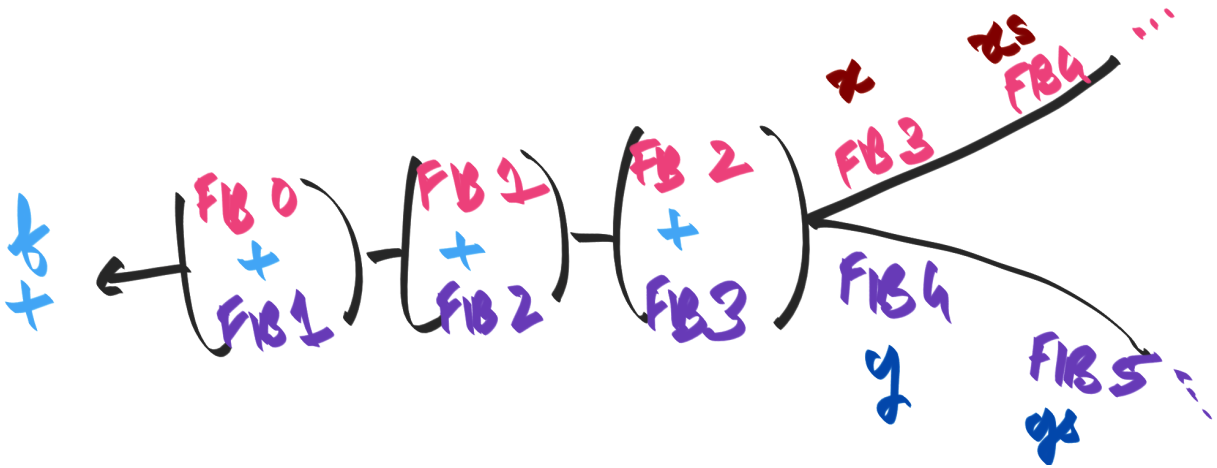
Fibonacci, Take 3

- `fibs = 0:1:fib_rec fibs`
- `fib_rec fibs = zipAdd fibs (tail fibs)`
- `zipAdd (x:xs) (y:ys) = (x+y):(zipAdd xs ys)`



Fibonacci, Take 3

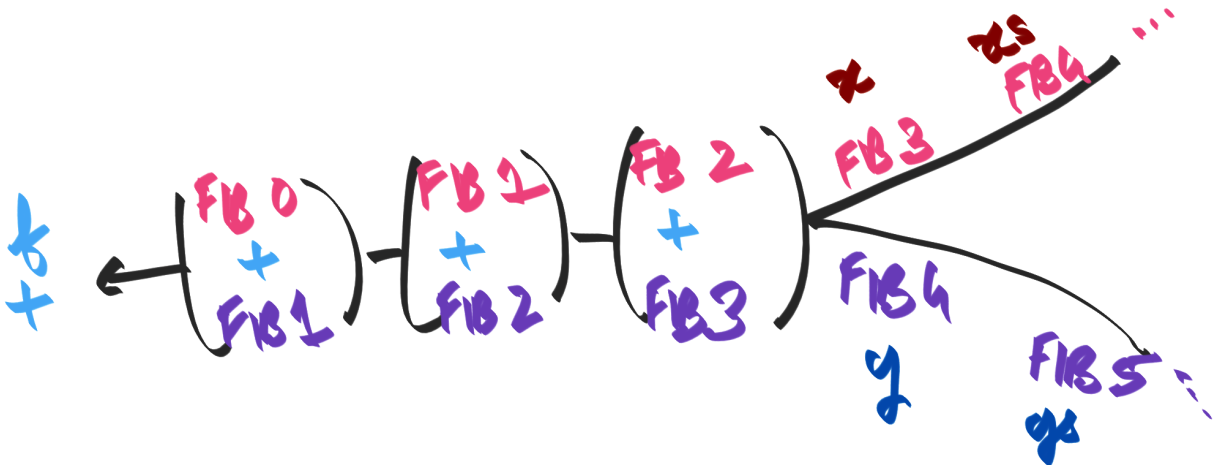
- `fibs = 0:1:fib_rec fibs`
- `fib_rec fibs = zipAdd fibs (tail fibs)`
- `zipAdd (x:xs) (y:ys) = (x+y):(zipAdd xs ys)`



- `fibs = 0:1:(zipAdd fibs (tail fibs))`

Fibonacci, Take 3

- `fibs = 0:1:fib_rec fibs`
- `fib_rec fibs = zipAdd fibs (tail fibs)`
- `zipAdd (x:xs) (y:ys) = (x+y):(zipAdd xs ys)`



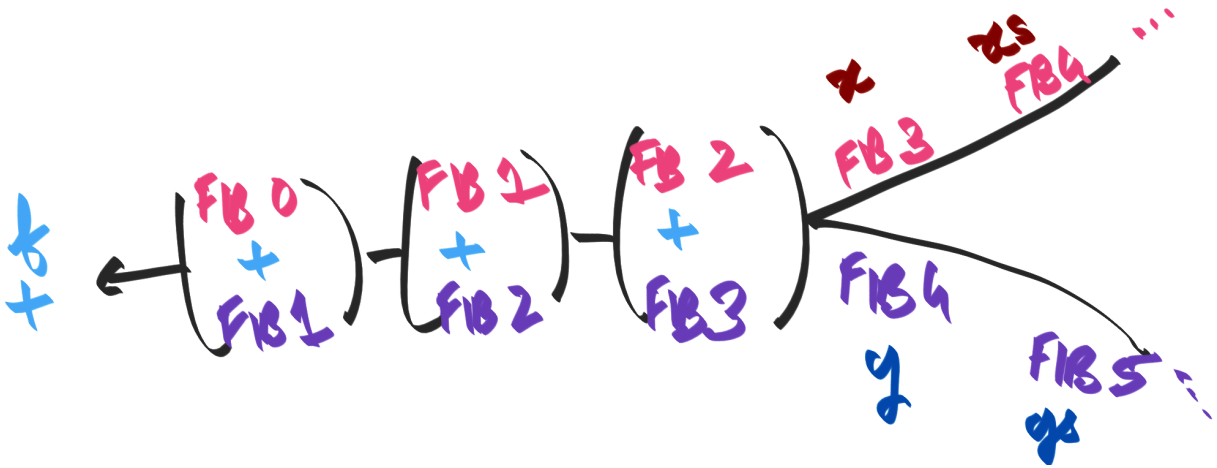
- `fibs = 0:1:(zipAdd fibs (tail fibs))`

Definition

What is the Fibonacci sequence? `fibs = 0:1:(zipWith (+) fibs (tail fibs))`

Fibonacci, Take 3

- `fibs = 0:1:fib_rec fibs`
- `fib_rec fibs = zipAdd fibs (tail fibs)`
- `zipAdd (x:xs) (y:ys) = (x+y):(zipAdd xs ys)`



- `fibs = 0:1:(zipAdd fibs (tail fibs))`

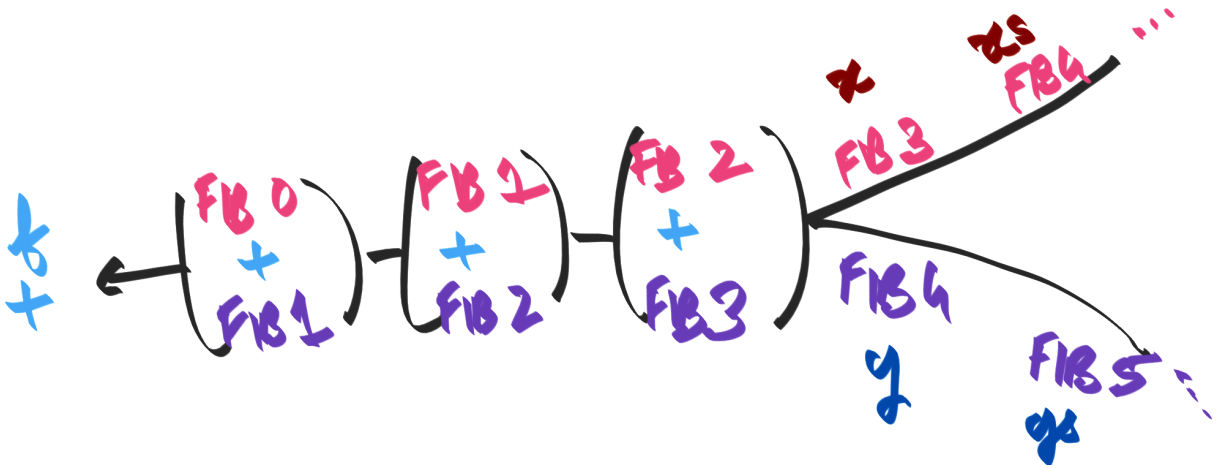
Definition

What is the Fibonacci sequence? `fibs = 0:1:(zipWith (+) fibs (tail fibs))`

- `zipWith (x:xs) (y:ys) = (f x y):(zipWith xs ys)`
- Replace + with `f` in `zipAdd`

Fibonacci, Take 3

- `fibs = 0:1:fib_rec fibs`
- `fib_rec fibs = zipAdd fibs (tail fibs)`
- `zipAdd (x:xs) (y:ys) = (x+y):(zipAdd xs ys)`



- `fibs = 0:1:(zipAdd fibs (tail fibs))`

Definition

What is the Fibonacci sequence? `fibs = 0:1:(zipWith (+) fibs (tail fibs))`

- `zipWith (x:xs) (y:ys) = (f x y):(zipWith xs ys)`
- Replace + with `f` in `zipAdd`
- This was not meant to enlighten! Only show off Haskell!

Should I learn hasell?

Should I learn hasell?

- Will it help me get a job?

Should I learn hasell?

- Will it help me get a job? Unlikely

Should I learn hasell?

- Will it help me get a job? Unlikely
- Will it help me do better at CP?

Should I learn hasell?

- Will it help me get a job? Unlikely
- Will it help me do better at CP? Ask Anurudh

Should I learn hasell?

- Will it help me get a job? Unlikely
- Will it help me do better at CP? Ask Anurudh
- Will it help me get a GSoC project?

Should I learn hasell?

- Will it help me get a job? Unlikely
- Will it help me do better at CP? Ask Anurudh
- Will it help me get a GSoC project? Maybe. The haskell community is small!

Should I learn hasell?

- Will it help me get a job? Unlikely
- Will it help me do better at CP? Ask Anurudh
- Will it help me get a GSoC project? Maybe. The haskell community is small!
- Will it help me learn math?

Should I learn hasell?

- Will it help me get a job? Unlikely
- Will it help me do better at CP? Ask Anurudh
- Will it help me get a GSoC project? Maybe. The haskell community is small!
- Will it help me learn math? Yes.

Should I learn hasell?

- Will it help me get a job? Unlikely
- Will it help me do better at CP? Ask Anurudh
- Will it help me get a GSoC project? Maybe. The haskell community is small!
- Will it help me learn math? Yes.
- Will it help me **understand what computation is**?

Should I learn hasell?

- Will it help me get a job? Unlikely
- Will it help me do better at CP? Ask Anurudh
- Will it help me get a GSoC project? Maybe. The haskell community is small!
- Will it help me learn math? Yes.
- Will it help me **understand what computation is**? Yes. It was designed to do so

Should I learn hasell?

- Will it help me get a job? Unlikely
- Will it help me do better at CP? Ask Anurudh
- Will it help me get a GSoC project? Maybe. The haskell community is small!
- Will it help me learn math? Yes.
- Will it help me **understand what computation is**? Yes. It was designed to do so

Should I learn hasell?

- Will it help me get a job? Unlikely
- Will it help me do better at CP? Ask Anurudh
- Will it help me get a GSoC project? Maybe. The haskell community is small!
- Will it help me learn math? Yes.
- Will it help me **understand what computation is**? Yes. It was designed to do so

The Haskell motto: Avoid success at all costs

Should I learn hasell?

- Will it help me get a job? Unlikely
- Will it help me do better at CP? Ask Anurudh
- Will it help me get a GSoC project? Maybe. The haskell community is small!
- Will it help me learn math? Yes.
- Will it help me **understand what computation is**? Yes. It was designed to do so

The Haskell motto: Avoid success at all costs

The Haskell motto: Avoid success, at all costs

Should I learn hasell?

- Will it help me get a job? Unlikely
- Will it help me do better at CP? Ask Anurudh
- Will it help me get a GSoC project? Maybe. The haskell community is small!
- Will it help me learn math? Yes.
- Will it help me **understand what computation is**? Yes. It was designed to do so

The Haskell motto: Avoid success at all costs

The Haskell motto: Avoid success, at all costs

The Haskell motto: Avoid " success at all costs "

The end: What is haskell?

- Haskell is a programming language

The end: What is haskell?

- Haskell is a programming language
- It is lazy: `k 10 (error "urk")`

The end: What is haskell?

- Haskell is a programming language
- It is lazy: `k 10 (error "urk")`
- It is strongly typed: `take :: Int -> [a] -> [a]`

The end: What is haskell?

- Haskell is a programming language
- It is lazy: `k 10 (error "urk")`
- It is strongly typed: `take :: Int -> [a] -> [a]`
- It is pure (We didn't talk about this)

The end: What is haskell?

- Haskell is a programming language
- It is lazy: `k 10 (error "urk")`
- It is strongly typed: `take :: Int -> [a] -> [a]`
- It is pure (We didn't talk about this)
- Its community is fanatical

The end: What is haskell?

- Haskell is a programming language
- It is lazy: `k 10 (error "urk")`
- It is strongly typed: `take :: Int -> [a] -> [a]`
- It is pure (We didn't talk about this)
- Its community is fanatical . . . about having the right abstractions: `Foldable`, `Num`

The end: What is haskell?

- Haskell is a programming language
- It is lazy: `k 10 (error "urk")`
- It is strongly typed: `take :: Int -> [a] -> [a]`
- It is pure (We didn't talk about this)
- Its community is fanatical . . . about having the right abstractions: `Foldable`, `Num`

Where to learn?

- The Freenode IRC channel `####haskell`

The end: What is haskell?

- Haskell is a programming language
- It is lazy: `k 10 (error "urk")`
- It is strongly typed: `take :: Int -> [a] -> [a]`
- It is pure (We didn't talk about this)
- Its community is fanatical . . . about having the right abstractions: `Foldable`, `Num`

Where to learn?

- The Freenode IRC channel `####haskell`
- CIS 194: Introduction to Haskell (course at UPenn)

The end: What is haskell?

- Haskell is a programming language
- It is lazy: `k 10 (error "urk")`
- It is strongly typed: `take :: Int -> [a] -> [a]`
- It is pure (We didn't talk about this)
- Its community is fanatical . . . about having the right abstractions: `Foldable`, `Num`

Where to learn?

- The Freenode IRC channel `####haskell`
- CIS 194: Introduction to Haskell (course at UPenn)
- The book `learn you a Haskell for great good`

The end: What is haskell?

- Haskell is a programming language
- It is lazy: `k 10 (error "urk")`
- It is strongly typed: `take :: Int -> [a] -> [a]`
- It is pure (We didn't talk about this)
- Its community is fanatical . . . about having the right abstractions: `Foldable`, `Num`

Where to learn?

- The Freenode IRC channel `####haskell`
- CIS 194: Introduction to Haskell (course at UPenn)
- The book `learn you a Haskell for great good`
- In-house: Watch anurudh solve CP problems in Haskell

The end: What is haskell?

- Haskell is a programming language
- It is lazy: `k 10 (error "urk")`
- It is strongly typed: `take :: Int -> [a] -> [a]`
- It is pure (We didn't talk about this)
- Its community is fanatical . . . about having the right abstractions: `Foldable`, `Num`

Where to learn?

- The Freenode IRC channel `####haskell`
- CIS 194: Introduction to Haskell (course at UPenn)
- The book `learn you a Haskell for great good`
- In-house: Watch anurudh solve CP problems in Haskell
- In-house: Ping folks in the Theory group with questions

The end: What is haskell?

- Haskell is a programming language
- It is lazy: `k 10 (error "urk")`
- It is strongly typed: `take :: Int -> [a] -> [a]`
- It is pure (We didn't talk about this)
- Its community is fanatical . . . about having the right abstractions: `Foldable`, `Num`

Where to learn?

- The Freenode IRC channel `####haskell`
- CIS 194: Introduction to Haskell (course at UPenn)
- The book `learn you a Haskell for great good`
- In-house: Watch anurudh solve CP problems in Haskell
- In-house: Ping folks in the Theory group with questions
- In-house: DM me with questions, or Pick a time to chat with me

The end: What is haskell?

- Haskell is a programming language
- It is lazy: `k 10 (error "urk")`
- It is strongly typed: `take :: Int -> [a] -> [a]`
- It is pure (We didn't talk about this)
- Its community is fanatical . . . about having the right abstractions: `Foldable`, `Num`

Where to learn?

- The Freenode IRC channel `####haskell`
- CIS 194: Introduction to Haskell (course at UPenn)
- The book `learn you a Haskell for great good`
- In-house: Watch anurudh solve CP problems in Haskell
- In-house: Ping folks in the Theory group with questions
- In-house: DM me with questions, or Pick a time to chat with me

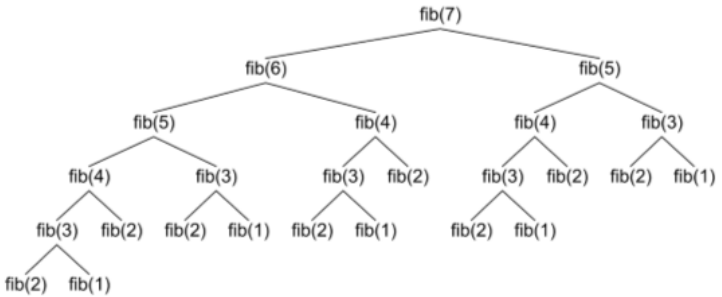
Fibonacci, Take 4

Fibonacci, Take 4

- `fib 0 = 1; fib 1 = 1; fib n = fib (n-1) + fib(n-2)`
- `fibs = map fib [0,1..]`

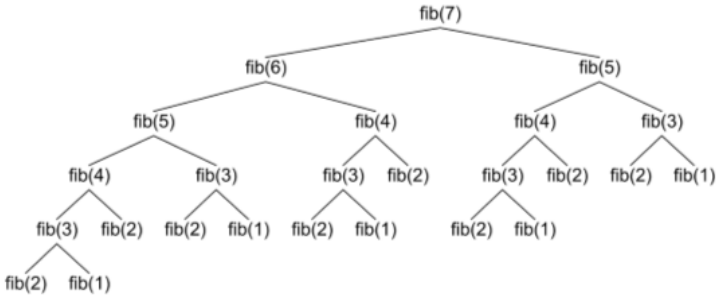
Fibonacci, Take 4

- `fib 0 = 1; fib 1 = 1; fib n = fib (n-1) + fib(n-2)`
- `fibs = map fib [0,1..]`

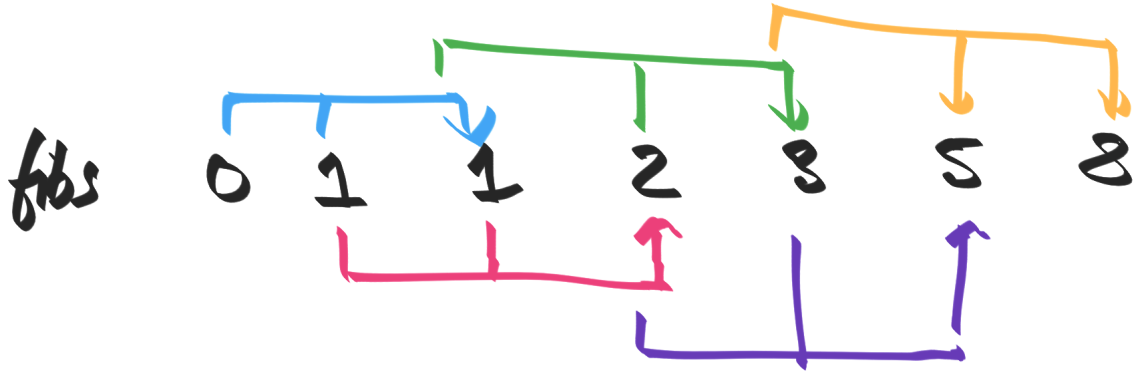


Fibonacci, Take 4

- `fib 0 = 1; fib 1 = 1; fib n = fib (n-1) + fib(n-2)`
- `fibs = map fib [0,1..]`



- `fibs = 0:1:zipWith (+) fibs (tail fibs)`



IO in haskell

- `int getchar()`

IO in haskell

- `int getchar()`
- `getChar :: IO Char`

IO in haskell

- `int getchar()`
- `getChar :: IO Char`
- `(*) :: Int -> Char -> Int`

IO in haskell

- `int getchar()`
- `getChar :: IO Char`
- `(*) :: Int -> Char -> Int`
- `1 * getChar -- does not typecheck`

IO in haskell

- `int getchar()`
- `getChar :: IO Char`
- `(*) :: Int -> Char -> Int`
- `1 * getChar -- does not typecheck`
- `do x <- getChar; print (2 * (fromEnum x))`

IO in haskell

- `int getchar()`
- `getChar :: IO Char`
- `(*) :: Int -> Char -> Int`
- `1 * getChar -- does not typecheck`
- `do x <- getChar; print (2 * (fromEnum x))`
- `x :: Char` because `getChar :: IO Char`
- `2 * (fromEnum x) == fromEnum x + fromEnum x`

IO in haskell

- `int getchar()`
- `getChar :: IO Char`
- `(*) :: Int -> Char -> Int`
- `1 * getChar -- does not typecheck`
- `do x <- getChar; print (2 * (fromEnum x))`
- `x :: Char` because `getChar :: IO Char`
- `2 * (fromEnum x) == fromEnum x + fromEnum x`
- `int x = getchar(); 2 * x == x + x`

IO in haskell

- `int getchar()`
- `getChar :: IO Char`
- `(*) :: Int -> Char -> Int`
- `1 * getChar -- does not typecheck`
- `do x <- getChar; print (2 * (fromEnum x))`
- `x :: Char` because `getChar :: IO Char`
- `2 * (fromEnum x) == fromEnum x + fromEnum x`
- `int x = getchar(); 2 * x == x + x`
- `2 * getchar() != getchar() + getchar()`

Effects, or the "M" word

Keep every element, and drop every element.

```
powerset xs = filterM (const [True, False]) xs
```

Currying

- `let peek x = take 1 x`

Currying

- `let peek x = take 1 x`
- `let peek'= take 1`

Currying

- `let peek x = take 1 x`
- `let peek'= take 1`
- `peek' x = take 1 x`

Currying

- `let peek x = take 1 x`
- `let peek'= take 1`
- `peek' x = take 1 x`
- Equational reasoning!

Currying

- `let peek x = take 1 x`
- `let peek'= take 1`
- `peek' x = take 1 x`
- Equational reasoning!
- `let x_incr = 1 + x`

Currying

- `let peek x = take 1 x`
- `let peek'= take 1`
- `peek' x = take 1 x`
- Equational reasoning!
- `let x_incr = 1 + x`
- `let x_incr = (+) 1 x`

Currying

- `let peek x = take 1 x`
- `let peek'= take 1`
- `peek' x = take 1 x`
- Equational reasoning!
- `let x_incr = 1 + x`
- `let x_incr = (+) 1 x`
- `let incr = (+) 1`

Currying

- `let peek x = take 1 x`
- `let peek'= take 1`
- `peek' x = take 1 x`
- Equational reasoning!
- `let x_incr = 1 + x`
- `let x_incr = (+) 1 x`
- `let incr = (+) 1`

Trolling the Haskell IRC channel

https://gist.githubusercontent.com/quchen/5280339/raw/a18362f99e3847351e891f2ed37872d1dfa9f942/trolling_haskell

```
xQuasar>      | HASKELL IS FOR FUCKIN FAGGOTS. YOU'RE ALL A BUNCH OF
              | FUCKIN PUSSIES
xQuasar>      | JAVASCRIPT FOR LIFE FAGS
luite>        | hello
ChongLi>      | somebody has a mental illness!
merijn>       | Wow...I suddenly see the error of my ways and feel
              | compelled to write Node.js!
genisage>     | hi
luite>        | you might be pleased to learn that you can compile
              | haskell to javascript now
xQuasar>      | FUCK YOU AND YOUR HUSSY OPS
xQuasar>      | THEY CAN'T DO SHIT TO ME CUNTS
quchen>       | xQuasar: While I don't think anything is wrong with
              | faggots to the point where "faggot" isn't insulting
              | anymore, I assure you we have heterosexuals in our
              | language as well.
quchen>       | Haskell is invariant under gender. Really!
Iceland_jack> | quchen++
merijn>       | I don't blame him, I'd be this angry to if I had to write
              | javascript all day too
xQuasar>      | FUCK YOU STRAIGHT CUNTS
xQuasar>      | BESTIALITY IS THE BEST
quchen>       | merijn: Lol. And when I write that I mean it :D
quchen>       | xQuasar: Do you have any specific questions?
merijn>       | This is sort of like a puppy trying to be angry with
              | you...it's just kinda adorable to see him think he has
              | any effect :)
quchen>       | xQuasar: You're offtopic right now. This is a Haskell
              | help channel. Do you have Haskell questions?
quchen>       | xQuasar: We'd love to help you make your first steps.
tsinnema>     | hey -- is there a proper way to whine about no one having
              | responded to a question? :)
xQuasar>      | i just want to get kicked out of a bunch of channels for
              | fun
quchen>       | Have you seen LYAH? It's a very enjoyable book on
```

Trolling the Haskell IRC channel

https://gist.githubusercontent.com/quchen/5280339/raw/a18362f99e3847351e891f2ed37872d1dfa9f942/trolling_haskell

```
quchen>       | @where lyah
xQuasar>      | why is no one cooperating with me
lambdabot>    | http://www.learnyouahaskell.com/
tsinnema>     | in soviet russia, haskell learns a you
merijn>       | tsinnema: Yeah, wait 30 minutes or more and try again :)
Iceland_jack> | xQuasar: We are cooperating with you, you're just not
              | aware that your goal is learning Haskell
ChongLi>      | xQuasar: why not learn some Haskell instead?
xQuasar>      | alright i'll admit i lose
merijn>       | Ha, sometimes I forget how hard it is to troll haskell :)
nxorg8>       | #haskell is awesome :-)
xQuasar>      | what's haskell good for though
quchen>       | xQuasar: But there is so much to win here!
xQuasar>      | i'm more into gamedev
Iceland_jack> | figures
arkeet>       | it's good for writing programs.
xQuasar>      | what kind of programs?
ChongLi>      | xQuasar: any kind
arkeet>       | "what's C++ good for?"
Iceland_jack> | xQuasar: The ones that run on comput-ars.
ChongLi>      | it's a general purpose language
luite>        | xQuasar: frp can be useful for writing games, and with
              | ghcjs you can compile them to javascript to make web
              | games
tsinnema>     | merijn, yeah, seems reasonable :)
ChongLi>      | seriously though, if you learn it you may completely
              | change your perspective on programming
xQuasar>      | i have absolutely no idea what frp and ghcjs are
...
```

Equality: Eq

Ordering: Ord

Dictionary ordering: Ord for lists

Oh, that's just monoidal accumulation of a left absorbing semigroup