2. Elm

```
> "Hello World"
"Hello World" : String
```

2. Elm

What is Elm?

Elm Practice

Tracing

Function Definitions

What is Elm?

1. Elm is a ***functional programming language***
   - *Strongly and statically typed*
   - *Compiles to JavaScript*
   - *Language design focused on simplicity*

2. Elm supports the implementation of ***web applications***
   - *Simple, declarative model/view/update pattern*
   - *No runtime errors*

http://elm-lang.org

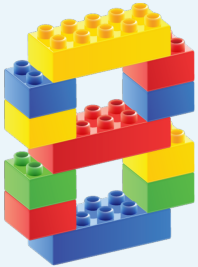
Why Elm (and not, say, Haskell)?

Elm is easier to learn

1. Elm is simpler than Haskell
   - *Elm has **no** type classes*
   - *Elm has fewer programming constructs*
   - *Elm provides better error messages*

2. Elm solutions are more straightforward
   - *No need to explain why different solutions are equivalent*
   - *No need to explain when to choose what approach*

A New Programming Model



Change Your Mental Model

When thinking about values and data:

- ***Stop** thinking "put into boxes/fetch from boxes"*
- ***Instead** think "construct & deconstruct"*

A Simple Elm Example

```
type alias Points = Int

type Grade = Pass | Fail


alice : Points
alice = 65


grade : Points -> Grade
grade p = if p>50 then Pass else Fail
```

`Point` *type synonym*

`Grade` *new (data) type*
`Pass`, `Fail` *constructors (w/o arguments)*

`:` *"has type"*
`alice` *name of a value*

`grade` *function name*
`Point -> Grade` *function type*
`p` *parameter (of type* `Points`*)*
`if ...` *expression (of type* `Grade`*)*

Why Elm?

1. Metalanguage for PL theory
   - ***Abstract syntax*** *represented by* ***algebraic data types***
   - ***Denotational semantics*** *represented by* ***functions***
   - ***Type checking*** *represented by* ***functions***

2. Makes you a better (imperative) programmer
   - *Forces you to think* ***recursively*** *and* ***compositionally***
   - *Forces you to* ***minimize use of state***

… essential skills for solving ***big*** *problems*

2. Elm

What is Elm?

Elm Practice

Tracing

Function Definitions

Elm Learning Path

Elm Language Concepts

- *Expressions, Values, and Types*[2.2]
  - *Names*[2.2.1]*; Tracing Evaluations*[2.2.2]*; Tuples*[2.2.3]
- *Functions*[2.3]
  - *Application*[2.3.1]*; Currying*[2.3.2]*; Definitions*[2.3.1]
- *Iteration and Recursion*[2.4]
- *Lists and Pattern Matching*[2.5]
- *Data Types*[2.6]
- *Higher-Order Functions*[2.7]

Exercise ...

Question 1

Write an expression that computes the larger value of `5` *and* `2*3` *(without using the function* `max`*).*

Question 2

Write an expression that tests whether `2*3` *is equal to* `3+3`

Exercise ...

Question 3

Determine the result of `let x=1 in x+let x=2 in x+x`

Question 4

Determine the result of `let x=1 in x+(let x=2 in x)+x`

Exercise …

Question 5

Determine the result of

```
let x=1 in (let y=x in (let x=y+1 in x))+x
```

Question 6

Determine the result of `let x=let y=1 in y+1 in 2*x`

2. Elm

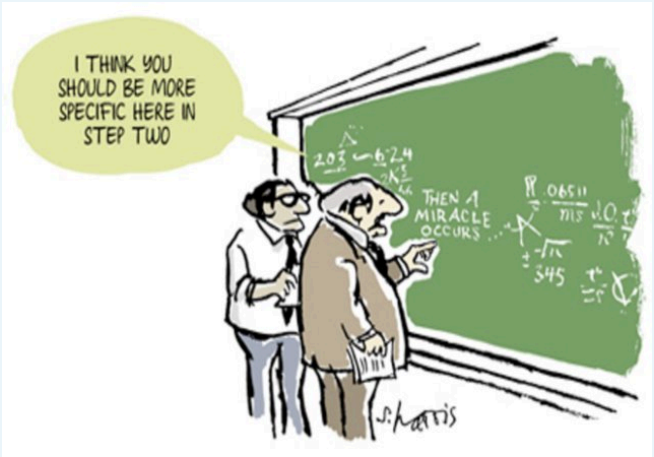
What is Elm?

Elm Practice

Tracing

Function Definitions

Why Tracing?


I THINK YOU SHOULD BE MORE SPECIFIC HERE IN STEP TWO
THEN A MIRACLE OCCURS

Trace = Decomposition

What is a Trace?

*A **step-by-step derivation** of a result from a problem*

Traces Explain 2 Things

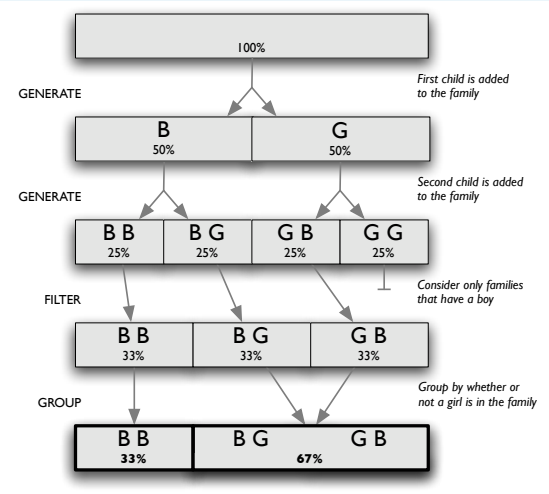
*How is the **result** obtained?*
*How does the **program** work?*

Traces as Explanations

A family has two children, one of which is a boy.

What's the probability the other child is a girl?

67%



Tracing Evaluations

Two Kinds of Actions

- ***Substitution:*** *replace a name by its definition*
- ***Simplification:*** *evaluate a subexpression*

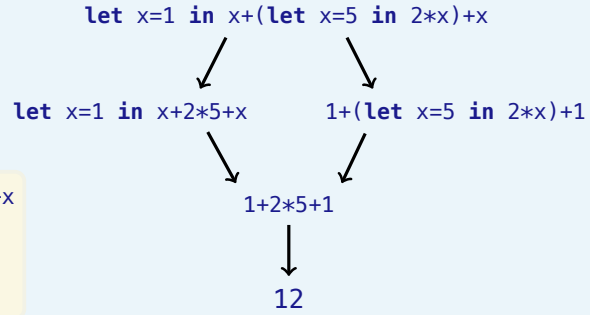
```
dbl : Int -> Int
dbl x = 2*x
```

```
   let y=2+1 in dbl y
=> let y=3 in dbl y
=> dbl 3
=> 2*3
=> 6
```

evaluate `2+1`
substitute `3` *for* `y`
substitute `2*x` *for* `dbl` *and* `3` *for* `x`
evaluate `2*3`

Multiple Evaluation Paths

```
   let x=1 in x+(let x=5 in 2*x)+x
=> let x=1 in x+2*5+x
=> 1+2*5+1
=> 12
```



Confluence

Different evaluation orders lead to the same result

A Recursive List Function

```
rev : List Int -> List Int
rev l = case l of
    []     -> []
    x::xs -> rev xs ++ [x]
```

`List Int` *is the type of integer lists*
`[]` *is the empty list*
`::` *adds single element at the front of a list*
`++` *concatenates two lists*

The ***pattern*** `x::xs` *does two things, it:*
(1) selects a case
(2) creates bindings for variables `x` *and* `xs`

Tracing Recursive Functions

Making bindings explicit:

```
   rev [1,2,3]
=> let x=1 in let xs=[2,3] in rev xs ++ [x]
=> rev [2,3] ++ [1]
=> let x=2 in let xs=[3] in rev xs ++ [x] ++ [1]
=> rev [3] ++ [2] ++ [1]
=> rev [3] ++ [2,1]
=> let x=3 in let xs=[] in rev xs ++ [x] ++ [2,1]
=> rev [] ++ [3] ++ [2,1]
=> rev [] ++ [3,2,1]
=> [] ++ [3,2,1]
=> [3,2,1]
```

```
rev : List Int -> List Int
rev l = case l of
    []    -> []
    x::xs -> rev xs ++ [x]
```

Tracing Recursive Functions

Immediately substituting parameters:

```
   rev [1,2,3]
=> rev [2,3] ++ [1]
=> rev [3] ++ [2] ++ [1]
=> rev [3] ++ [2,1]
=> rev [] ++ [3] ++ [2,1]
=> rev [] ++ [3,2,1]
=> [] ++ [3,2,1]
=> [3,2,1]
```

```
rev : List Int -> List Int
rev l = case l of
    []    -> []
    x::xs -> rev xs ++ [x]
```

Exercise ...

```
rev [1,2,3]
=> rev [2,3] ++ [1]
=> rev [3] ++ [2] ++ [1]
=> rev [3] ++ [2,1]
=> rev [] ++ [3] ++ [2,1]
=> rev [] ++ [3,2,1]
=> [] ++ [3,2,1]
=> [3,2,1]
```

```
(++) : List Int -> List Int -> List Int
l1 ++ l2 = case l1 of
     []    -> l2
     x::xs -> x::(xs ++ l2)
```

Question 7

Create a trace for the evaluation of the expression `[1,2] ++ [5]`

Solution to Exercise

Question 7

Create a trace for the evaluation of the expression `[1,2] ++ [5]`

```
   [1,2] ++ [5]
=> 1::([2] ++ [5])
=> 1::(2::([] ++ [5]))
=> 1::(2::[5])
=> 1::[2,5]
=> [1,2,5]
```

```
(++) : List Int -> List Int -> List Int
l1 ++ l2 = case l1 of
     []    -> l2
     x::xs -> x::(xs ++ l2)
```

2. Elm

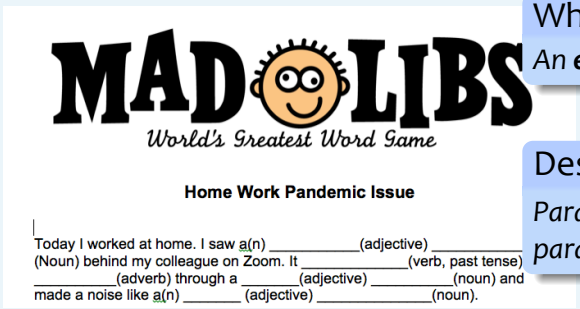
What is Elm?

Elm Practice

Tracing

Function Definitions

Function = Parameterized Expression



What is a Function?

*An **expression** with a **named parameter***

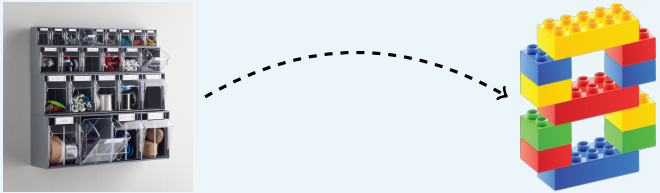
Description of a Function

Parameter & expression using the parameter

Function Definition

Giving a name to a function

But: Mind the Programming Model



```
case v of
    ...
    p -> e
```

Programming with Pattern Matching

- A ***value*** v *is a term built out of constructors*
- A ***pattern*** p *is a term built out of constructors and variables*
- A ***rule*** p -> e *consists of a pattern* p *and an expression* e
- ***Matching*** v *against* p *creates a set of* ***variable bindings***
- ***Evaluating*** *a rule for* v *means to evaluate* e *in the context of bindings produced by matching* v *against* p

Exercise ...

Question 8

Define a function `min3` *that computes the smallest of three numbers.*

First, define the function using nested **`if`–`then`–`else`** *expressions.*

Then try to find a simpler definition that reuses other functions.

Iteration and Recursion

Generic `while` loop

```
state := initValues;
while cond(state) {
    update(state)
};
print result(state)
```

Corresponding recursion

```
loop(state) = if cond(state) then
                  loop(update(state))
              else
                  result(state)

loop(initValues)
```

If the condition is true, continue updating the state.
Otherwise, return the result.
Start with some initial values.

(De/Re)Constructive Programming

Generic (linear, binary) recursion template

```
type T = Con | Bin S T

f : T -> U
f x = case x of
    Con     -> e1          -- base case
    Bin s t -> e2{s,f t}   -- inductive case
```

`Con` *constant (constructor) of type* `T`

`Bin` *binary constructor of type* `S -> T -> T`

`e1` *expression of type* `U`

`e2` *expression using* `s` *and* `f t` *of type* `U`

Inspect `T` *value, and deconstruct it into parts.*
Construct `U` *value from the (processed) parts.*

Exercise …

Question 9

Define a function `isEven : Int -> Bool` *that yields* `True` *for even numbers and* `False` *for odd numbers.*
(Hint: You need three equations: one for the base case `0`*, one for the base case* `1`*, and one case for all other numbers* `n`*, which requires a recursive definition.)*

Lists

Lists are built with two ***constructors***

`[]` *empty list*

`v::l` *adds element* `v` *at the front of list* `l`

List Constructors

```
[]   : List a
(::) : a -> List a -> List a
```

List Accessors

```
isEmpty : List a -> Bool
head    : List a -> Maybe a
tail    : List a -> Maybe (List a)
```

Programming with Lists

Lists ***constructors***

`[]` *empty list*
`v::l` *adds element* `v` *at the front of list* `l`

Template for List Processing

```
f : List T -> U
f l = case l of
    []    -> e1          -- result for empty list
    x::xs -> e2{x,xs}    -- result for nonempty list
```

`e1` *expression of type* `U`
`e2` *expression of type* `U` *using* `x` *and* `xs`

Exercise ...

Question 10

Define the function `length : List Int -> Int`

Question 11

Define the function **`member`** `: Int -> List Int -> Bool` *that checks whether a particular number is contained in a list. Give a recursive definition.*

Exercise ...

Question 12

Define the function `delete : Int -> List Int -> List Int` *that deletes all occurrences of the first argument from the list given as a second argument.*

Question 13

Define the function `insert : Int -> List Int -> List Int` *that inserts an integer at the correct position into a sorted list.*

Discussion ...

Consider the following function definition.

```
g l1 l2 = case l1 of
   []     -> l2
   x::xs -> g xs (x::l2)
```

What is the result of `g [1,2,3] [4]`*?*

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

(Paper) `[3,2,1,4]` ✓

(Scissors) *Something else*

Exercise ...

Question 14

Define the function **`preorder`** `: Tree -> List Int` *that produces a list of nodes obtained by a pre-order traversal of the tree.*

Exercise ...

Question 15

What does the function `map f . map g` *do? How could it be rewritten?*

Question 16

Define the function `last : List a -> a` *as a composition of some list function and the function* `rev`*.*

Exercise ...

Question 17

Define the function **`reflexive`** `: (a -> a -> a) -> (a -> a)` *that turns a binary function into a unary function by using the same parameter for both arguments.*

Question 18

Define the functions **`times2`** *and* **`sqr`** *as instances of* **`reflexive`**.