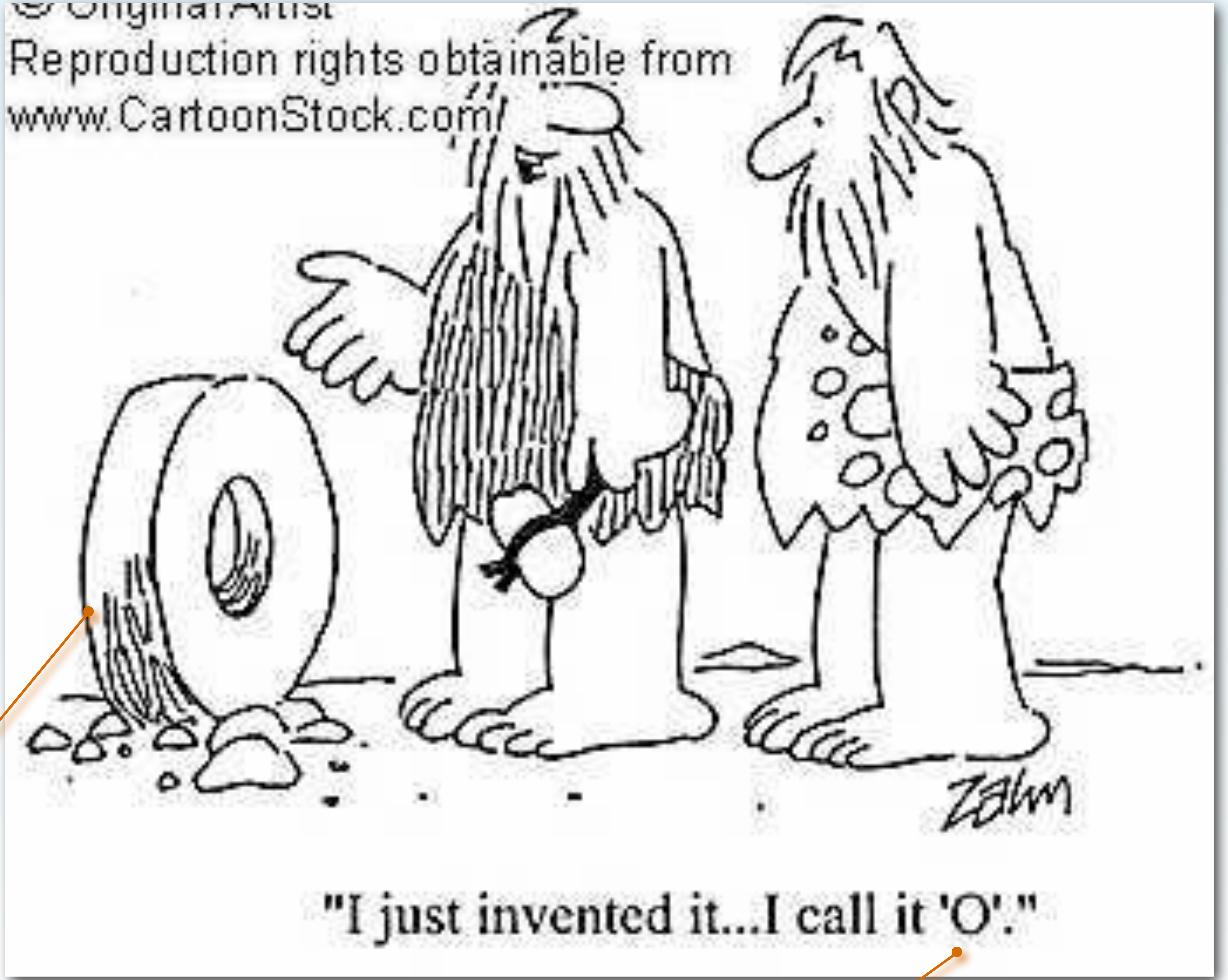
3 Semantics


HOW GOES THE OBEDIENCE TRAINING?
GREAT! WATCH THIS.
HEEL.
CLAMP
NEXT WEEK WE START ON SEMANTICS.
7-14
©2010 Creators.com
www.JohnHartStudios.com

3 Semantics

en.wikipedia.org/wiki/Blissymbols



Semantics

Syntax

3 Semantics

Why semantics?

What is semantics?

Semantics of simple expression languages

Elements of semantic definitions

Examples: Shape & Move languages

Advanced Semantic Domains

Translating Haskell into denotational semantics

Haskell as a metalanguage

Why Semantics ?

Modus Ponens

$$\frac{S \longrightarrow G \quad S}{G}$$

If this statement is true, then God exists

If this statement is true, then God does not exist

If this statement is true, then the NSA stores only meta data

Haskell B. Curry, 1900-1982

Prior, A. N., 1955. "Curry's Paradox and 3-Valued Logic", *Australasian Journal of Philosophy* 33:177-82

See also: John Allen Paulos: *Irreligion,* Hill and Wang 2008

Curry's Paradox

Why Semantics ?

Recursion without a base case

S = If *S* is true, then God exists

S = If *S* is true, then God does not exist

S = If *S* is true, then the NSA stores only meta data

Why Semantics ?

Access to non-local variables

```
{
  int x=2;
  int f(int y) {return y+x;};
  {
    int x=4;
    printf("%d", f(3));
  }
}
```

Output? 5

Why Semantics ?

Swap the values of two variables

```
{
   int x=1;
   int y=8;

   y = x;
   x = y;
}
```

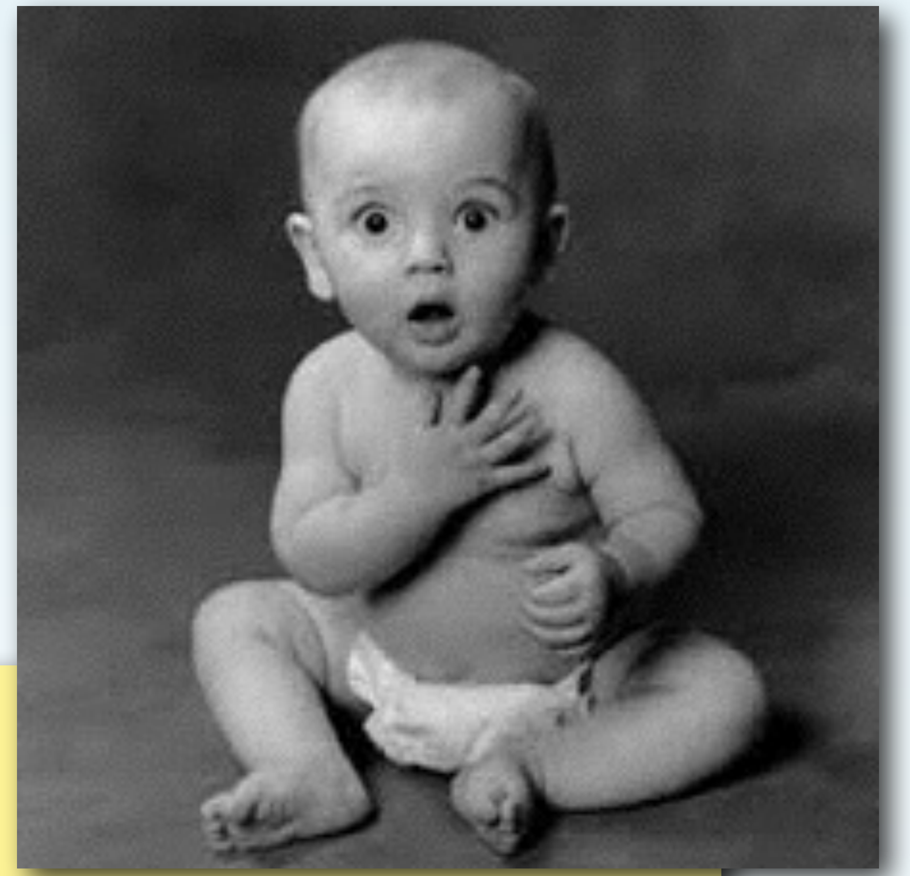
Effect? y: 1 x: 1

```
{
   int x=1;
   int y=8;
   int z;

   z = y;
   y = x;
   x = z;
}
```

Effect? y: 1 x: 8

What?!



```
{
   int x=1;
   int y=8;

   y = x + 0*(x = y);
}
```

Effect? y: 1 x: 8

Why Semantics ?

- Understand what program constructs do
- Judge the correctness of a program
  (compare expected with observed behavior)
- Prove properties about languages
- Compare languages
- Design languages
- Specification for implementations

Syntax: Form of programs
Semantics: Meaning of programs

The Meaning of Programs

What is the meaning of a program?

It depends on the language!

Language	*Meaning*
Boolean expressions	Boolean value
Arithmetic expressions	Integer
Imperative Language	State transformation
Logo	Picture

Denotational Semantics of a language:
Transformation of representation
(abstract syntax → semantic domain)

Simple Examples

BoolSyn.hs
BoolSem.hs

ExprSyn.hs
ExprSem.hs

Exercises

(1) Extend the boolean expression language by an and operation (abstract syntax and semantics)

(2) Extend the arithmetic expressions by multiplication and division (abstract syntax and semantics)

(3) Define a Haskell function to apply DeMorgan's laws to boolean expression, i.e., a function to transform any expression not (x and y) into (not x) or (not y) (and accordingly for not (x or y))

Defining Semantics in 3 Steps

Example Language
"*Arithmetic expressions*"

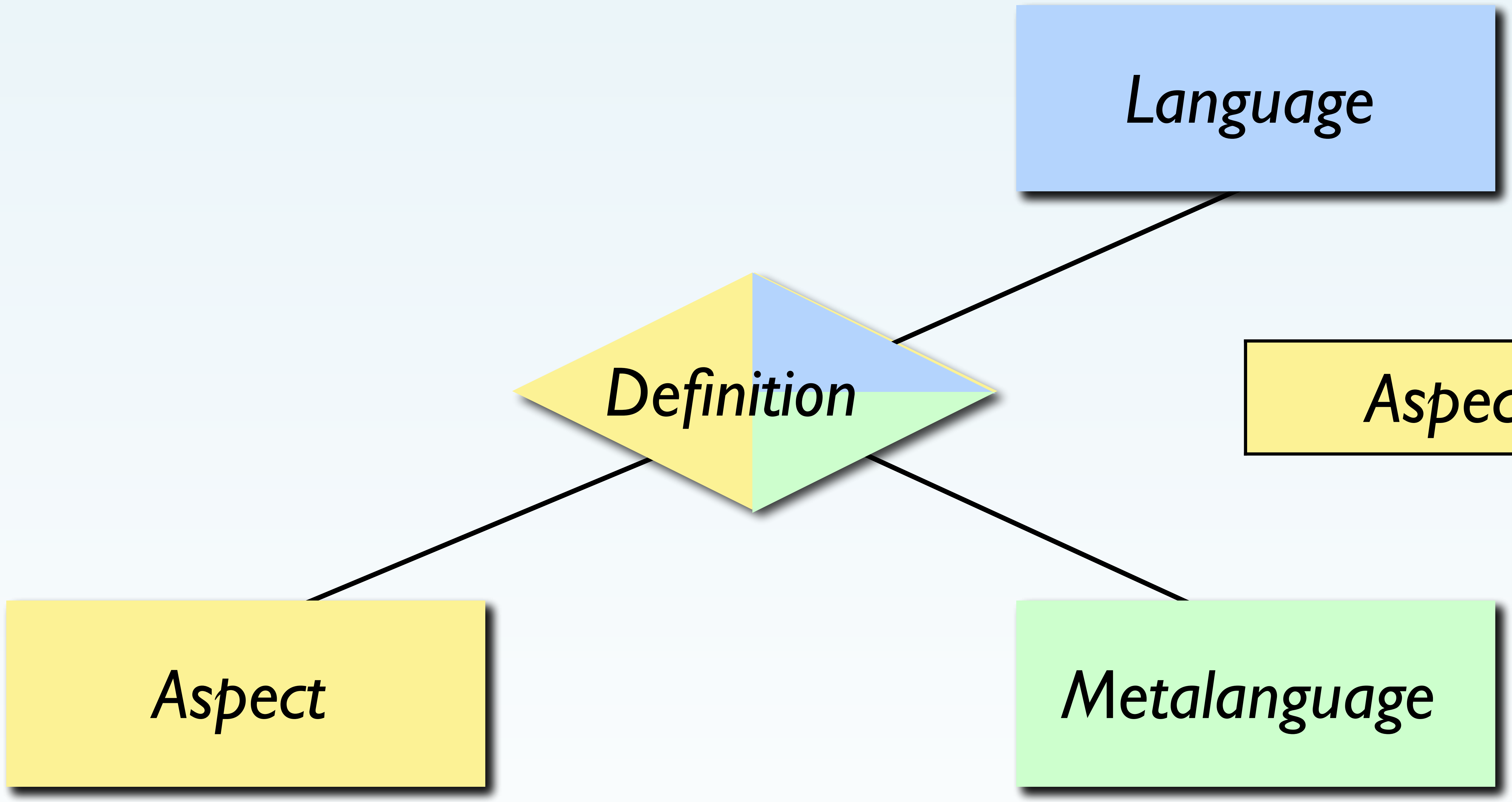
S:	`Expr`
D:	`Int`
$[\![\cdot]\!]$:	`sem :: Expr → Int`

(1) Define the *abstract syntax* *S*,
i.e. set of syntax trees

(2) Define the *semantic domain* *D*,
i.e. the representation of semantic values

(3) Define the *semantic function* / *valuation* $[\![\cdot]\!] : S \longrightarrow D$
that maps trees to semantic values

Language Definitions



Definition Profile

	Language
Aspect	Metalanguage

Example Expression Language

Expr
Syntax · Haskell

```haskell
data Expr = N Int
          | Plus Expr Expr
          | Neg Expr
```

Expr
Semantics · Haskell

Semantic Domain

```haskell
sem :: Expr → Int
sem (N i)        = i
sem (Plus e e') = sem e + sem e'
sem (Neg e)      = -(sem e)
```

Semantic Function

Expr
Syntax · Grammar

$$Expr \quad ::= \quad Num \mid Expr\texttt{+}Expr \mid \texttt{-}Expr$$

Expr
Semantics · Math

$$\begin{aligned} [\![\cdot]\!] &: Expr \to Int \\ [\![n]\!] &= n \\ [\![e\texttt{+}e']\!] &= [\![e]\!] + [\![e']\!] \\ [\![\texttt{-}e]\!] &= -[\![e]\!] \end{aligned}$$

Syntactic Symbol

Semantic Operation

Related: Expressions vs. Values

`tail [1 .. 4]` $\xrightarrow{\textit{semantics}}$ `[2,3,4]` *or* `2:3:4:[]`

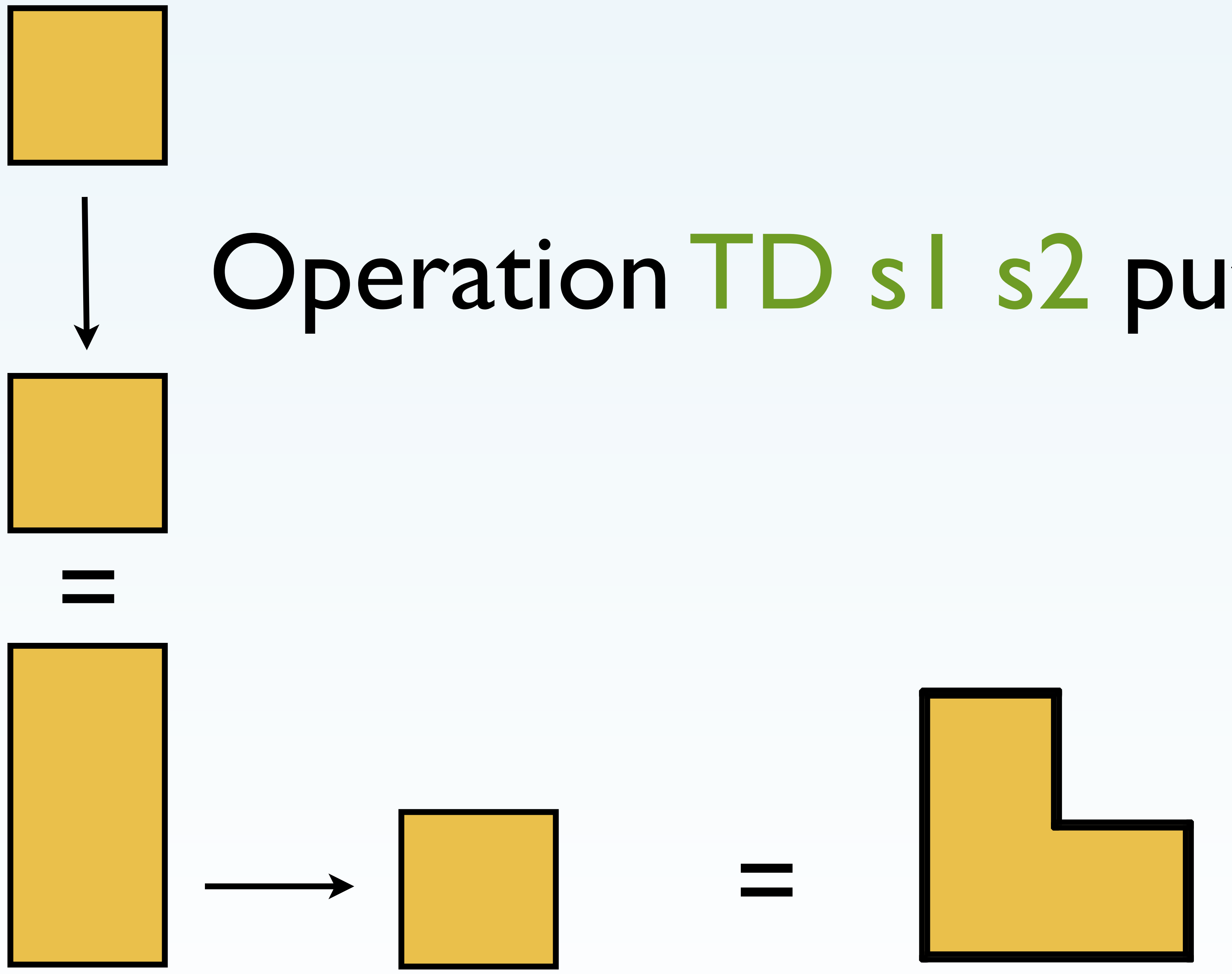
~~`[2 .. 4]`~~

`3 + 5` $\xrightarrow{\textit{semantics}}$ `8`

~~`2 + 6`~~

Example: Shape Language

A language for constructing bitmap images: an image is either a pixel or a vertical or horizontal composition of images

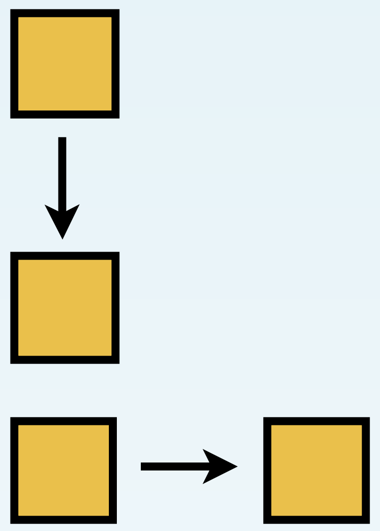


Operation TD s1 s2 puts s1 on top of s2

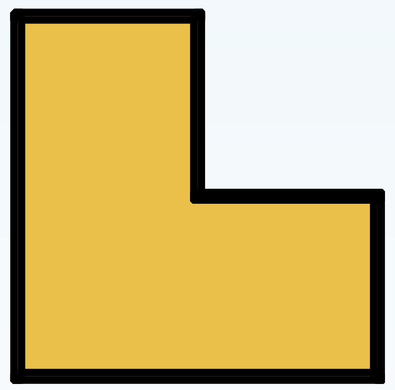
Operation LR s1 s2 puts s1 left next to s2

Abstract Syntax

```
data Shape = X
           | TD Shape Shape
           | LR Shape Shape
```



Example:



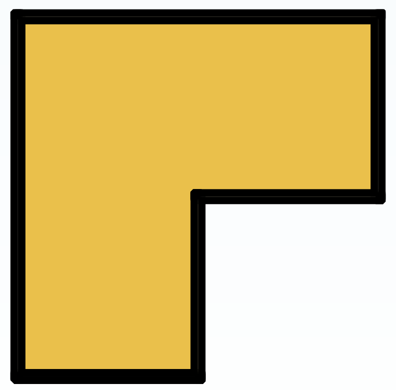
```
LR (TD X X) X
```

```
TD X (LR X X)
```

LR aligns at bottom

TD aligns at left

```
TD (LR X X) X
```



... part of semantics

Semantic Domain

How to represent a bitmap image?

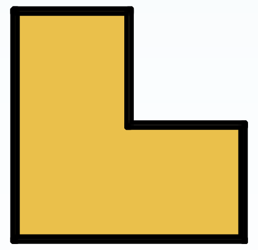
```
data Shape = X
           | TD Shape Shape
           | LR Shape Shape
```

```
type Image = Array (Int,Int) Bool
```

Drawback: size is fixed, operations require complicated bit shifting

```
type Pixel = (Int,Int)
type Image = [Pixel]
```

`LR (TD X X) X` —*semantics*→ `[(1,1),(1,2),(2,1)]`



Semantic Function (1)

Approach: Translate individual shapes separately into bitmaps and then compose bitmaps

```haskell
data Shape = X
           | TD Shape Shape
           | LR Shape Shape
```

semantics →

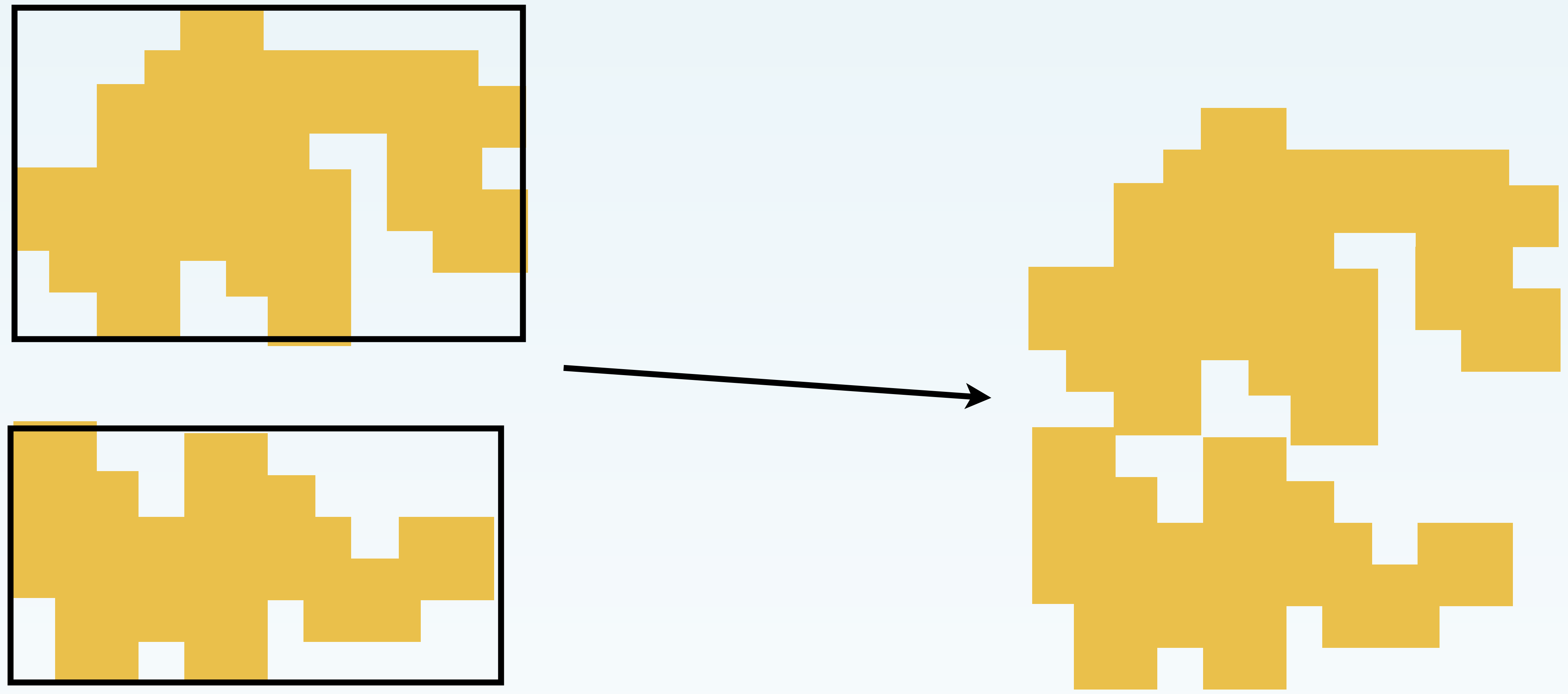
```haskell
type Pixel = (Int,Int)
type Image = [Pixel]
```

Base case: Individual pixel

```haskell
sem :: Shape -> Image
sem X = [(1,1)]
```

Semantic Function (2)

How can we compose (horizontally and vertically) two bitmap images without overlapping?



Take bounding boxes and adjust y-coordinates of top shape by height of bottom shape

```
sem (TD s1 s2) = adjustY ht p1 ++ p2
    where p1 = sem s1
          p2 = sem s2
          ht = maxY p2
```

Semantic Function (3)


```
sem (TD s1 s2) = adjustY ht p1 ++ p2
    where p1 = sem s1
          p2 = sem s2
          ht = maxY p2
```

```
maxY :: [(Int,Int)] -> Int
maxY p = maximum (map snd p)
```

```
adjustY :: Int -> [(Int,Int)] -> [(Int,Int)]
adjustY ht p = [(x,y+ht) | (x,y) <- p]
```

Exercise

```haskell
sem (TD s1 s2) = adjustY ht p1 ++ p2
    where p1 = sem s1
          p2 = sem s2
          ht = maxY p2

maxY :: [(Int,Int)] -> Int
maxY p = maximum (map snd p)

adjustY :: Int -> [(Int,Int)] -> [(Int,Int)]
adjustY ht p = [(x,y+ht) | (x,y) <- p]
```

(1) Define the functions:

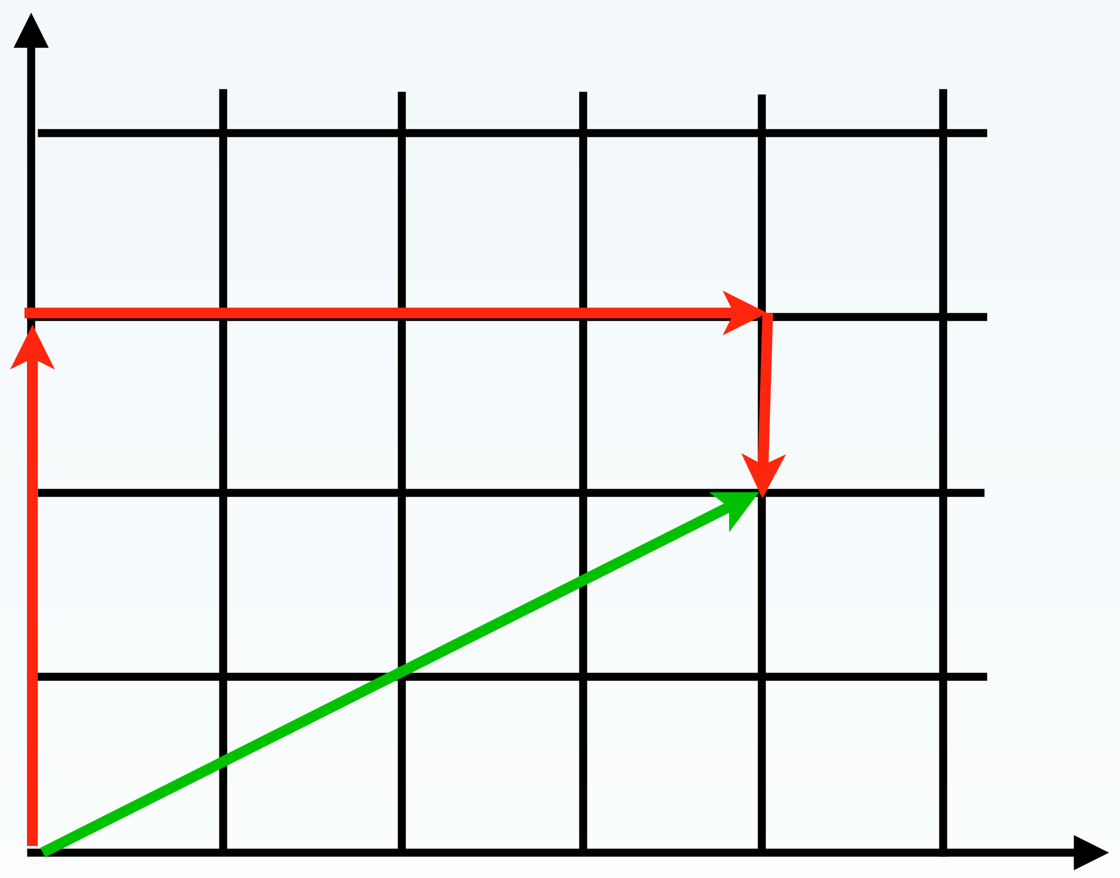
`sem (LR s1 s2)`   `maxX`   `adjustX`

Example

Shape.hs
ShapePP.hs

Example: Move Language

A language describing vector-based movements in the 2D plane.
A *step* is an *n*-unit horizontal or vertical move,
a *move* is a sequence of steps.



Go Up 3;
Go Right 4;
Go Down 1

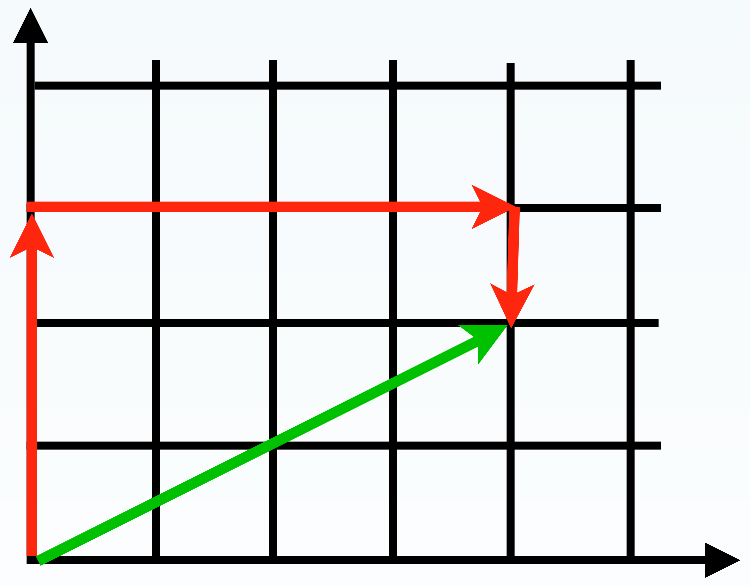
Abstract Syntax


```
data Dir  = Lft | Rgt | Up | Dwn

data Step = Go Dir Int

type Move = [Step]
```

Example:



```
[Go Up 3, Go Rgt 4, Go Dwn 1]
```

Exercises

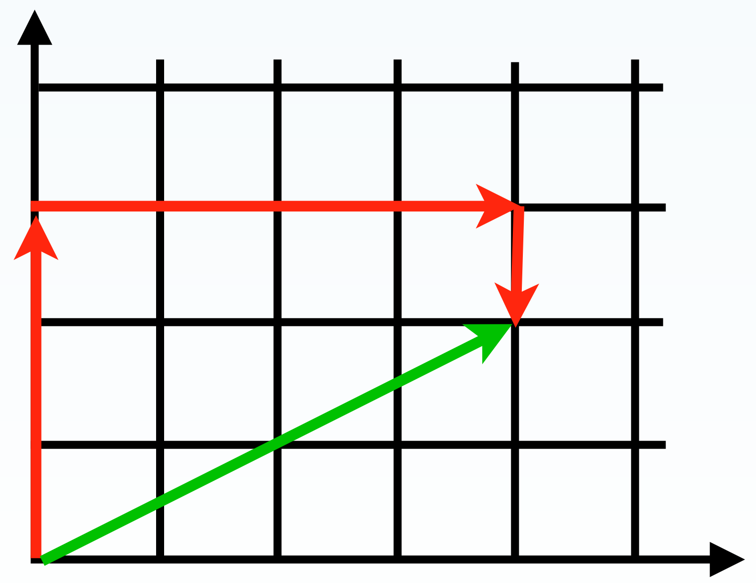
(1) Give a type definition for the data type `Step`

```
data Step = Go Dir Int
```

(2) Define the data type `Move` without using built-in lists

```
type Move = [Step]
```

(3) Write the move `[Go Up 3, Go Rgt 4, Go Dwn 1]` using the representation from (1) and(2)



Semantic Domain

What is the meaning of a move?

```
data Dir  = Lft | Rgt | Up | Dwn

data Step = Go Dir Int

type Move = [Step]
```

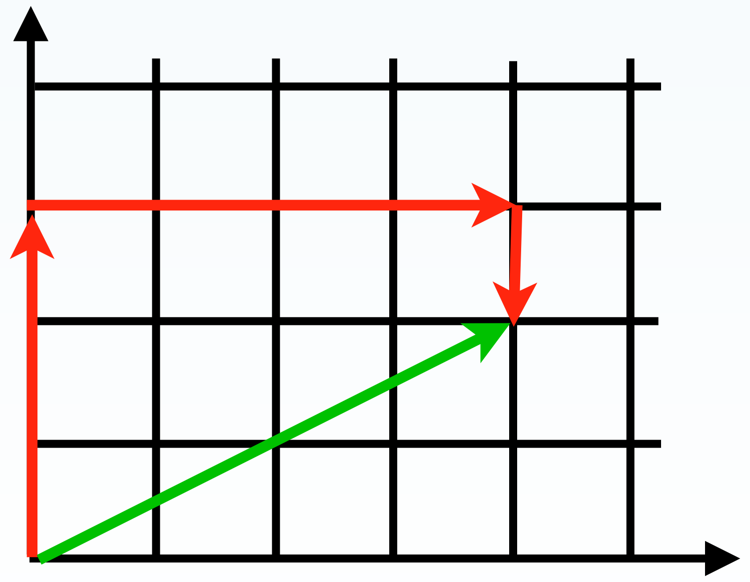
The distance traveled, the final position, or both.

```
type Pos = (Int,Int)
```

```
[Go Up 3, Go Rgt 4, Go Dwn 1]
```

semantics

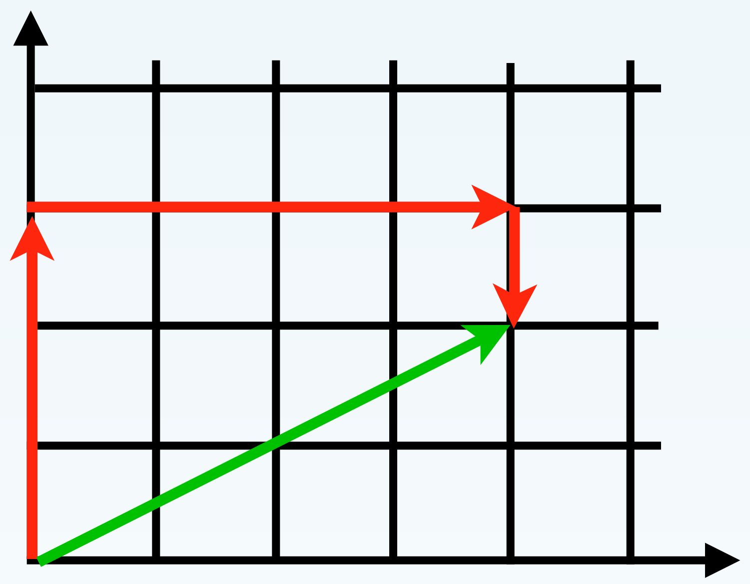
```
(4,2)
```



Semantic Function

```
sem :: Move -> Pos
sem []           = (0,0)
sem (Go d i:ss) = (dx*i+x,dy*i+y)
                   where (dx,dy) = vector d
                         (x,y)   = sem ss
```

pattern matching in definitions



```
vector :: Dir -> (Int,Int)
vector Lft = (-1,0)
vector Rgt = (1,0)
vector Up  = (0,1)
vector Dwn = (0,-1)
```

Example

Move.hs

Exercises

```
sem :: Move -> Pos
sem []           = (0,0)
sem (Go d i:ss) = (dx*i+x,dy*i+y)
                  where (dx,dy) = vector d
                        (x,y)   = sem ss
```

(1) Define the semantic function for the move language for the semantic domain

```
type Dist = Int
```

(2) Define the semantic function for the move language for the semantic domain

```
type Trip = (Dist,Pos)
```

Advanced Semantic Domains

The story so far: Semantic domains were mostly simple types (such as `Int` or `[(Int,Int)]`)

How can we deal with language features, such as *errors*, *union types*, or *state*?

(1) *Errors*: Use the `Maybe` *data type*

(2) *Union types*: Use corresponding *data types*

(3) *State*: Use *function types*

Error Domains

If `T` *is the type representing "regular" values, define the semantic domain as* `Maybe T`

regular value

error value

```
data Maybe a = Just a | Nothing
```

type of regular values

Example

ExprErr.hs

Union Domains

If `T1` *…* `Tk` *are types representing different semantic values for different nonterminals, define the semantic domain as a data type with k constructors.*

semantic domain

```
data T = C1 T1
       | ...
       | Ck Tk
```

different types of result values

Example

Expr2.hs

Exercises

(1) Extend the semantic domain for the two-type expression language to include errors

```
data Val = I Int
         | B Bool
```

(2) Extend the semantic function for the two-type expression language to handle errors

Function Domains

If a language operates on a state that can be represented by a type `T`, define the semantic domain as a function type `T -> T`

```
type D = T -> T
```

```
sem :: S -> D
```

=

```
sem :: S -> (T -> T)
```

=

```
sem :: S -> T -> T
```

Semantic function takes state as an additional argument

Example

RegMachine.hs

Exercises

(1) Extend the machine language to work on two registers A and B

```
data Op = LD Int
        | INC
        | DUP
```

(2) Define a new semantic domain for the extended language

```
type RegCont = Int

type D = RegCont -> RegCont
```

(3) Define the semantics functions for the extended language

RegMachine2.hs

Translating Haskell into Mathematical Denotational Semantics

(1) Replace *type definitions* by *sets (should actually be CPOs)*

(2) Replace *patterns* by *grammar productions*

(3) Replace *function names* by *semantic brackets* that enclose only syntactic objects

	Expr
Semantics	Haskell

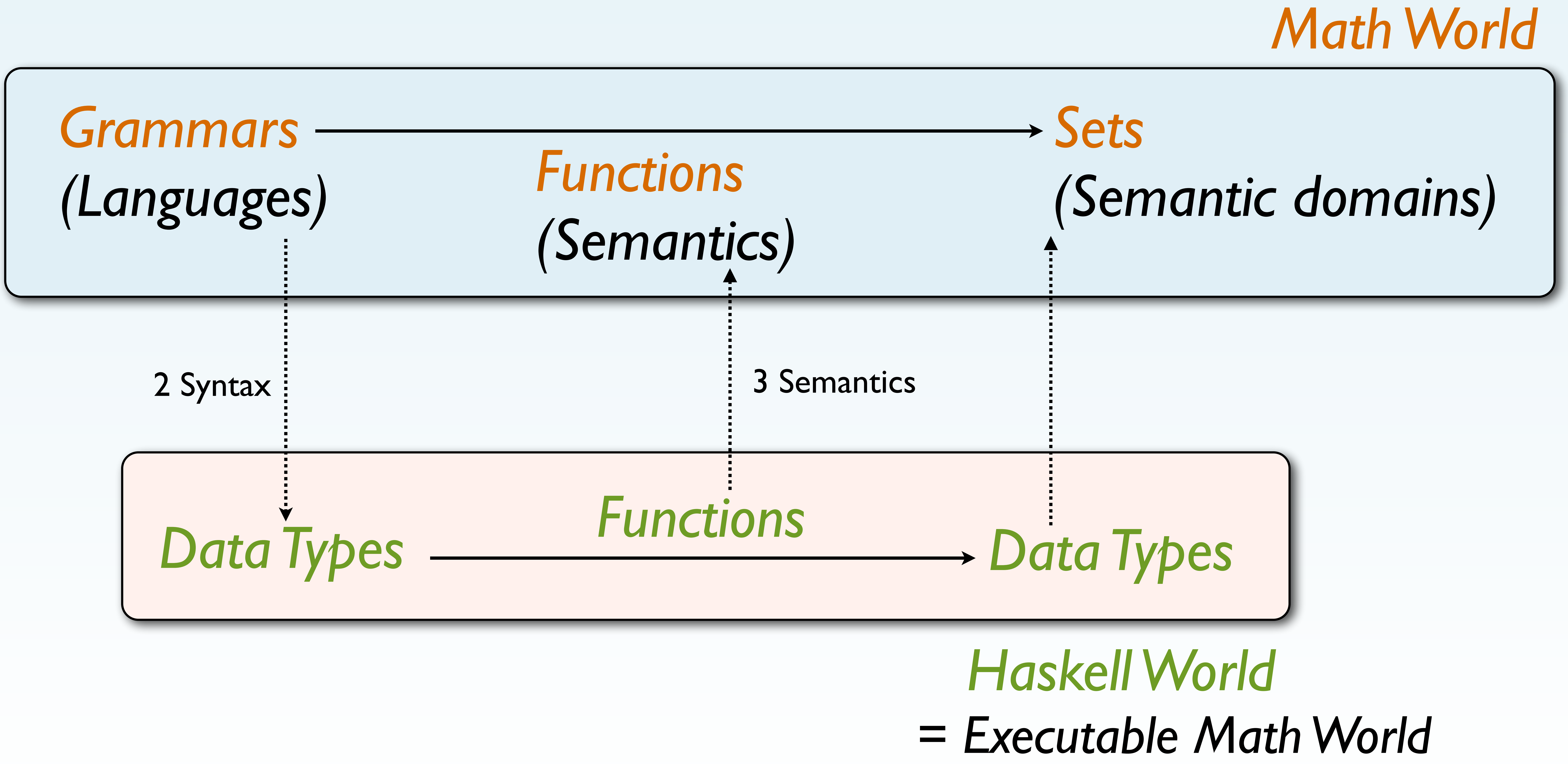
```
sem :: Expr → Int ❶
sem (N i)         = i
sem (Plus❷ e e') = sem e + sem e'
sem (Neg e)       = -(sem❸ e)
```

	Expr
Semantics	Math

$$
\begin{aligned}
[\![\cdot]\!] &: Expr \rightarrow Int^{❶} \\
[\![n]\!] &= n \\
[\![e \overset{❷}{+} e']\!] &= [\![e]\!] + [\![e']\!] \\
[\![-e]\!] &= -[\![e]\!]^{❸}
\end{aligned}
$$

$$Expr \; ::= \; Num \mid Expr+Expr \mid -Expr$$

Haskell as a Mathematical Metalanguage



--- END OF SLIDES ---

BACKUP SLIDES FOLLOW

Why Semantics ?