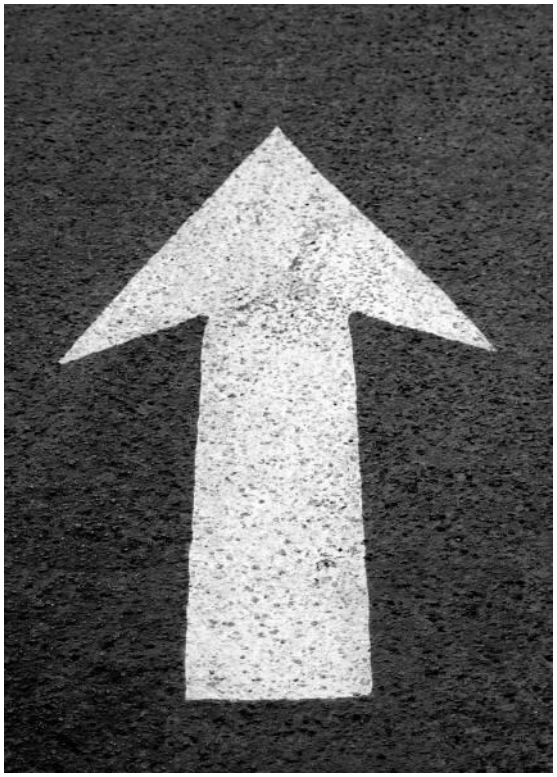
Specification, Representation, Implementation

COM2001 Advanced Programming Topics

The course so far

- Abstract Data Types
  - Used to specify behaviour
  - Language-independent
  - Three parts: sorts, syntax, semantics
- Implementing an ADT in Haskell
  - Fairly straightforward
  - Error cases can be handled in many ways



Coming up after Easter

- Program Proof

  Having implemented the ADT, how do you prove that you've done it correctly?
  - Structural induction (for recursively defined types)
  - Equational reasoning (deciding that two expressions evaluate to the same thing, even if you don't know what that is)
  - Proof by contradiction ("reductio ad absurdam")
  - Floyd-Hoare Logic

Haskell's class and type system

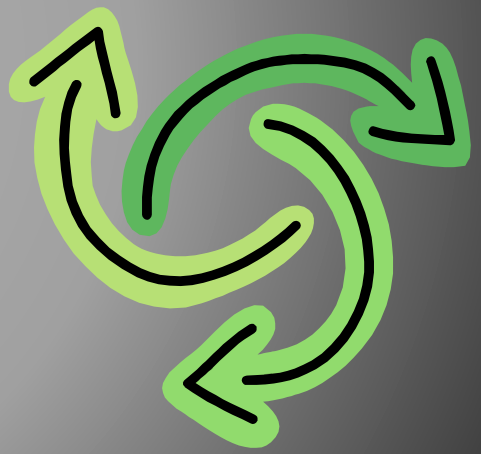
Having proven that the algorithm is correct, you need to program it. You need to understand the features of your target language.

Classes offer a convenient way to override functions, and allow us to express constraints on both data types and functions.

This section

- We return to the beginning
- How do we know **how** to implement an ADT?
- We'll consider solutions in both Haskell- and Java-like languages

From ADT to Representation



Stack a (again!)

```
createStack : Stack a
       push : a -> Stack a -> Stack a
 emptyStack : Stack a -> Bool
        top : Stack a -> (a U Msg a)
        pop : Stack a -> (Stack a U Msg (Stack a))
```

- This is a data type, but not a data *structure*
- It says nothing about logical structure of data within the stack
- We need to know how data is to be **represented** in memory

Stack a

- As an algebraic data type?
- Why not just use lists?
- What about using an array?
- Or a linked list?

All of these solutions are sensible.
All of these solutions are abstract.

There are lots of way to represent a data type. How do we identify them?

What do we represent?

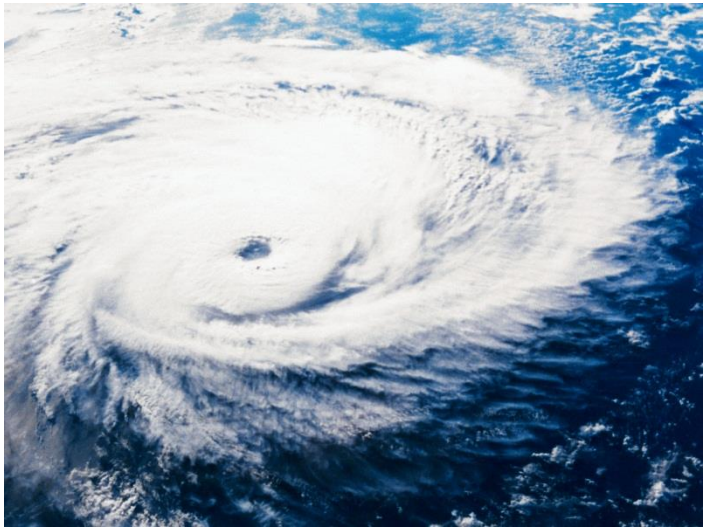
- What features need to be included in a logical representation of a stack?
- To answer this question, look at the functions and consider what they tell us.

createStack	: Stack a
push	: a -> Stack a -> Stack a
emptyStack	: Stack a -> Bool
top	: Stack a -> (a U Msg a)
pop	: Stack a -> (Stack a U Msg (Stack a))

createStack

```
createStack : Stack a
```

This function tells us nothing about how a stack should be represented as a data structure. All it tells us is that a createStack function needs to be defined.



push

```
push : a -> Stack a -> Stack a
```

Again, this function says nothing about what needs to be stored in the data structure, except that a push function should be definable.

emptyStack

```
emptyStack : Stack a -> Bool
```

This function **does** tell us something about the data that needs to be stored. It says that information must be accessible which tells us whether or not the stack is empty.

top

```
top : Stack a -> (a U Msg a)
```

This function tells us that it should be possible to identify the top of the stack. Relevant information must be accessible.

pop

```
pop : Stack a -> (Stack a U Msg (Stack a))
```

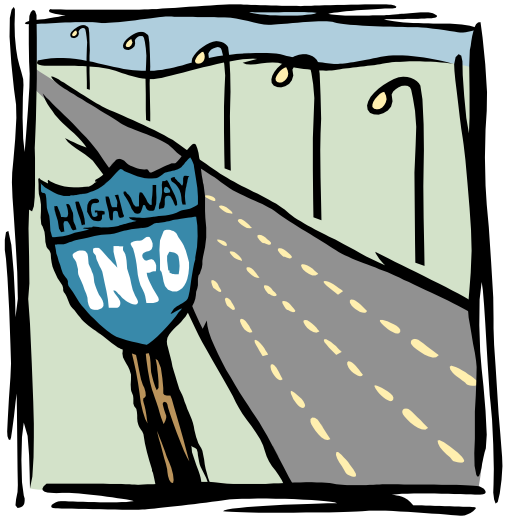
This function tells us nothing useful, except that a pop function needs to be definable.

Constructors vs Observers

- When defining semantics for an algebraic data type, we focus on the constructors
  - createStack
  - push
- Here we focus instead on the **observers**.
- Observers tell us what information has to be accessible in the data structure

Stack a's observers

- The observers are **emptyStack** and **top**
- In addition to representing the stack's entries itself, the representation should also include information stating
  - is the stack empty?
  - what's the top entry?



Using an array as a stack

CreateStack					**empty**
Push 1	**1**				**not empty**
Push 2	**1**	**2**			**not empty**
Push 3	**1**	**2**	**3**		**not empty**
Pop	**1**	**2**	**3**		**not empty**
Pop	**1**	**2**	**3**		**not empty**
Pop	**1**	**2**	**3**		**empty**
Push 6	**6**	**2**	**3**		**not empty**

The top element is shown in orange

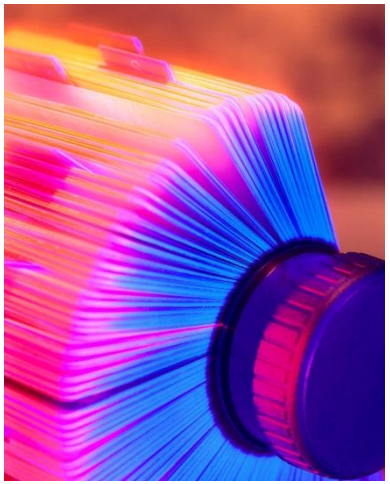
Required data components

The analysis shows that any representation of a stack has to contain at least the following components (the names stackrep, empty and top are arbitrary).

Component	Type	Role
stackrep	Container	Contains the actual data
empty	Boolean flag	Represents whether or not the stack is empty
top	Pointer to entry	Points to the top entry of the stack

Choosing a container

- What container should we use?
- Different containers are natural choices depending on the language used
- The semantics also matter. The fact that stacks are LIFO means that the container's cells have to be indexable
- Haskell makes lots of use of **lists**
- But in many languages, the "natural" choice of ordered structure is an **array**



Beware: size restrictions

- The ADT definition of Stack a assumes we can always add new entries to the stack
- Arrays typically have a fixed pre-defined size
- New error conditions will be introduced if we represent stacks using arrays

Implementations can fail not because the code is wrong, but because they're based on less-than-perfect data representations

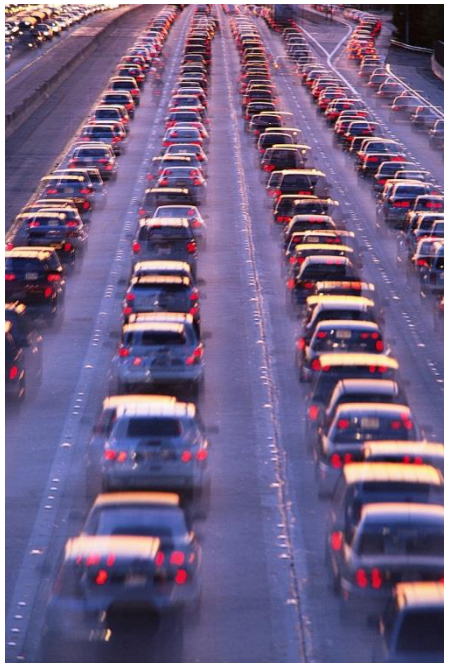
From Representation to Implementation

»

Stack as Array

- Suppose the array contains exactly N cells
- For simplicity, assume array indices start at 1
- Call the array **data[1..N]**

There are many array-based representations to choose from

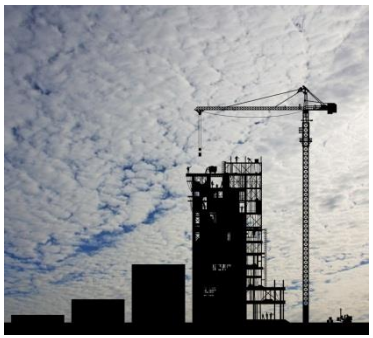


Stack as Array - 1

Component	Representation	Of type
stackrep	data[1..N]	Array of Entry
empty	flag	Boolean
top	topIndex	Integer in the range 1-N

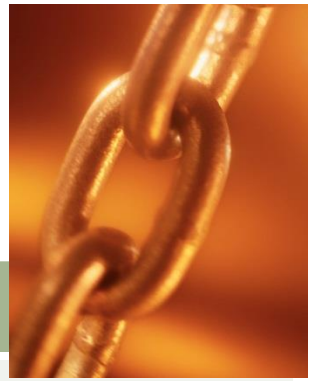
```
class DoubleStack {
  double[]  data;
  boolean empty;
  int topIndex;
}
```

The boolean flag is redundant. We can tell if the array is empty by checking whether topIndex is 0.



NB. We should also include a constructor in practice

Stack as Array - 2



Component	Representation	Of type
stackrep	data[1..N]	Array of Entry
empty	topIndex == 0	Boolean
top	topIndex	Integer in the range 1-N

```
class IntStack {
   int[] data;
   int topIndex;
}
```

We can write a function to extract **empty** information.

```
bool empty() {
  return (topIndex == 0);
}
```

Stack as Vector

Component	Representation	Of type
stackrep	data	Vector
empty	flag	Boolean
top	(built-in)	

```
class Stack {
  private Vector data;
  private boolean empty;
  private Stack() {
    data = new Vector();
    empty = true;
  }
  public createStack() { ... }
```

Stack as a list in a functional language

```
data Stack a = Stack [a]
  deriving (Eq, Show)

createStack :: Stack a
createStack = Stack []

push :: a -> Stack a -> Stack a
push x (Stack s) = Stack (x:s)
```

Summary

- ADTs are abstract. They describe behaviour.
- Representations are derived from the ADT. They describe logical data structure, and the algorithms that will be used to implement the syntax. They are still essentially language-independent.
- The implementation instantiates the representation within a particular language.