COM2001

Advanced Programming Techniques

Mike Stannett (m.stannett@sheffield.ac.uk)
Department of Computer Science
The University of Sheffield

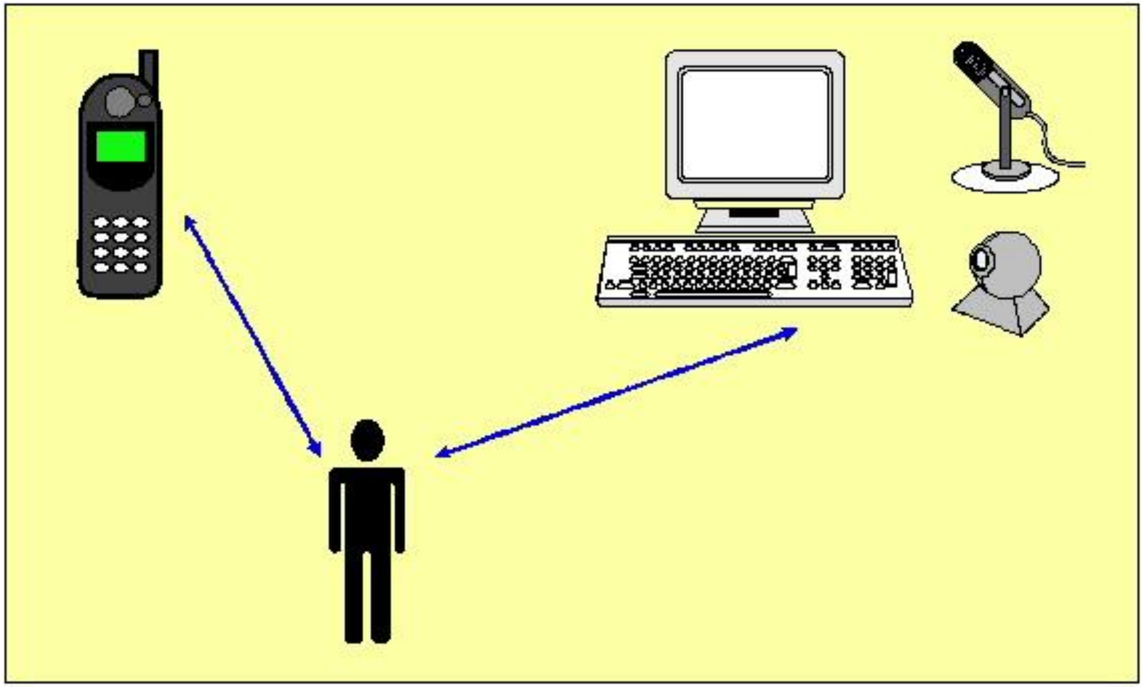
Spring Semester

ABSTRACT DATA TYPES, PROGRAM COMPLEXITY, AND CORRECTNESS

Some basic questions

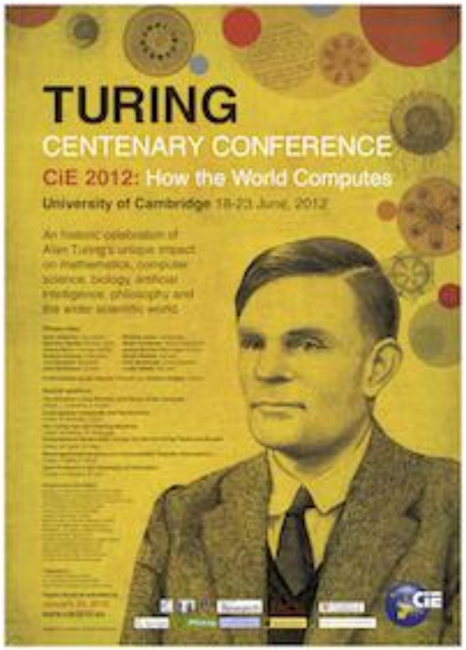
- How do we know whether a program written in Java **does the same thing** as a program written in Haskell (or Perl or PHP or Ruby, or…)?
- How do we know whether two hardware systems have **the same behaviour**?
- How do we specify behaviour in a **language-independent** way?
- Can we **prove** that our program is correct?
- How **efficient** is our program?
- Can it be made **more** efficient?
- Is there a **limit** to efficiency?

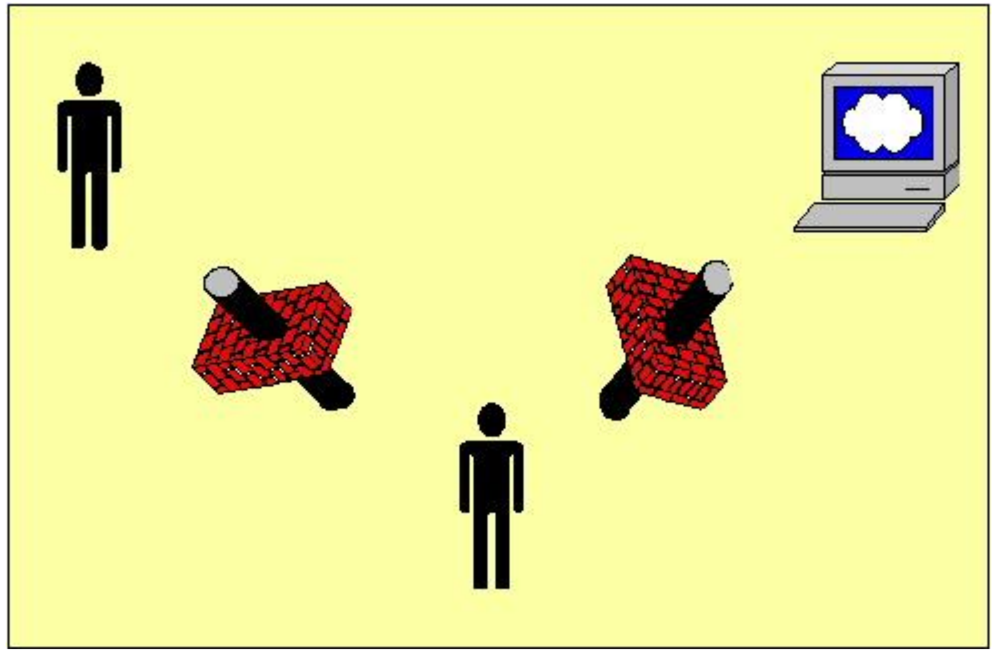
Equivalent behaviours?



If you can't tell which system you're talking to,
they are behaviourally equivalent (for that particular task)

Familiar example: Turing Test





If you can't distinguish between a human and a machine when testing for intelligence, the machine is intelligent in the same way the human is (in the sense being tested for)

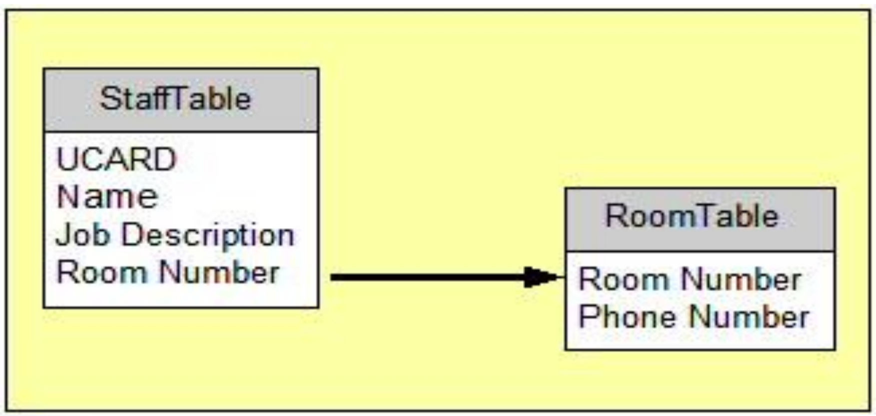
Engineering for equivalence

AddPhone

$\Delta PhoneDB$
$name? : NAME$
$number? : PHONE$

$name? \notin known$

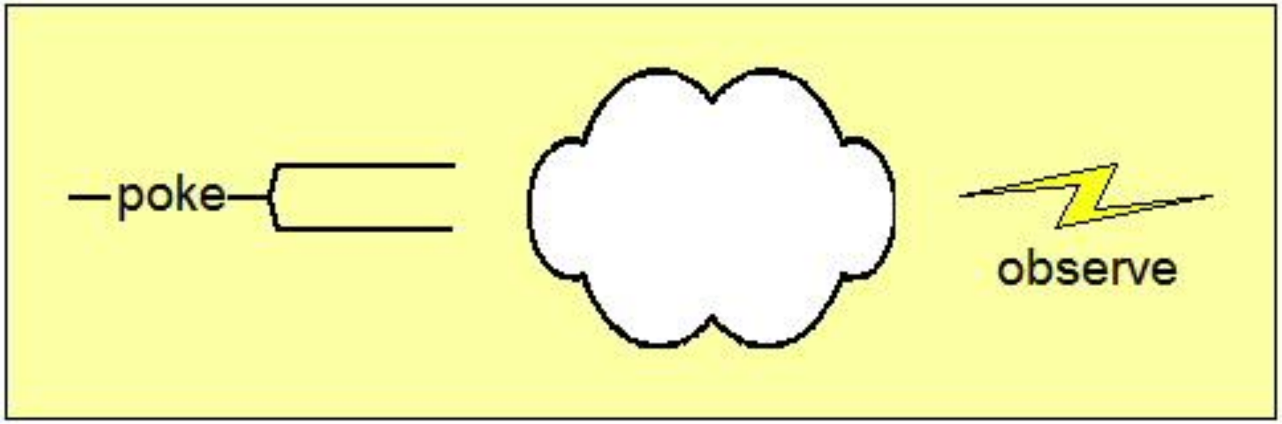
$phone' = phone \oplus \{name? \mapsto number?\}$



Specifications and implementations use different languages.

How do we know whether the specification is satisfied?

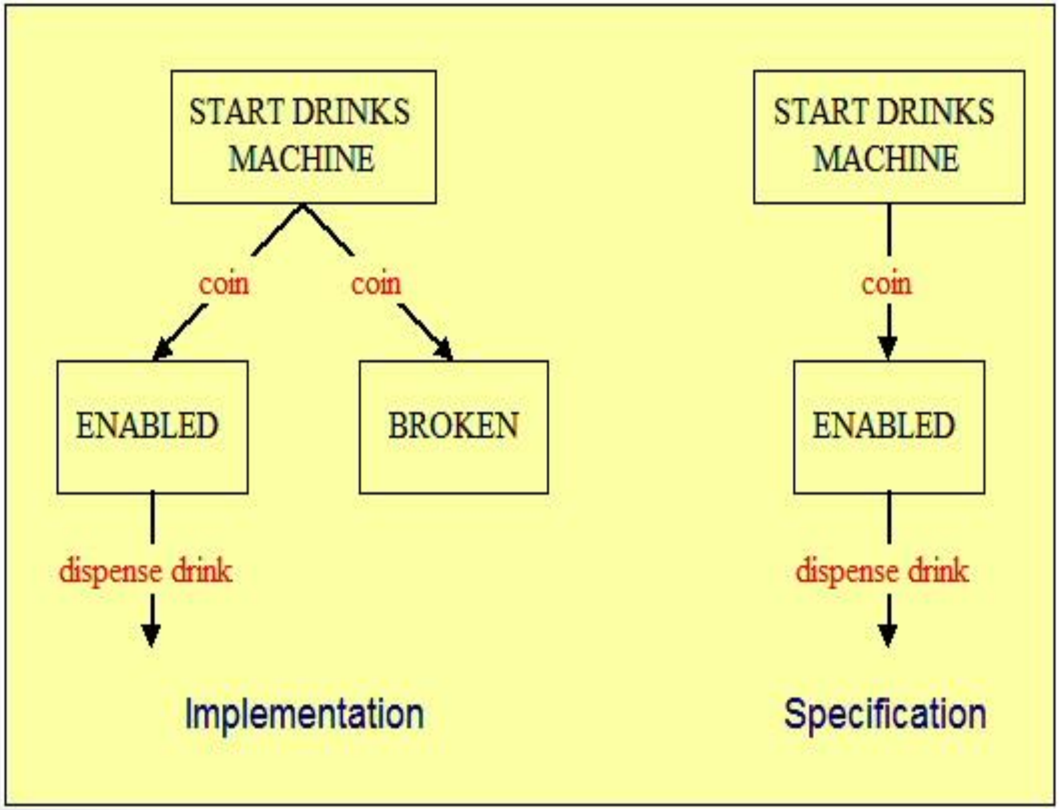
General principle



- In order to identify the specification's behaviour: poke it and see how it responds
- If the implementation never responds differently to what's specified, we say it is **correct**
- We say the implementation **satisfies** (simulates) the specification; it may do extra stuff as well.

Warning!

The situation is more subtle than this suggests



Each of these systems simulates the other - but they're NOT equal as drinks machines.

The BROKEN state produces no output, so **it cannot produce the wrong output**. To observe this problem, you need to consider **termination** as well (**total correctness**).

Termination and correctness

SPEC: Produce a function that takes an int as input, and whose output is 0

```
int foo(int n) {
  if (n == 3) {
    loop_forever();
  }
  return 0;
}
```

CORRECT: Whenever it terminates, it does so with the correct output.

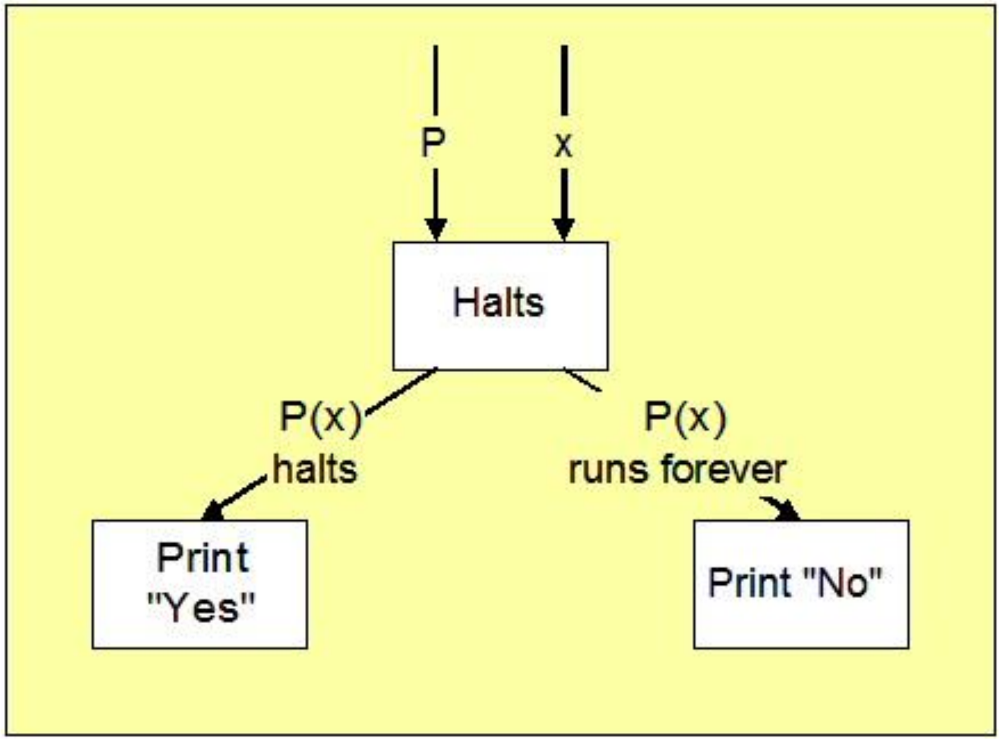
TOTALLY CORRECT
It's correct
and also
whenever it's supposed to terminate, it does so.

```
int foo(int n) {
  if (n == 3) {
    wait(10^10 years);
  }
  return 0;
}
```

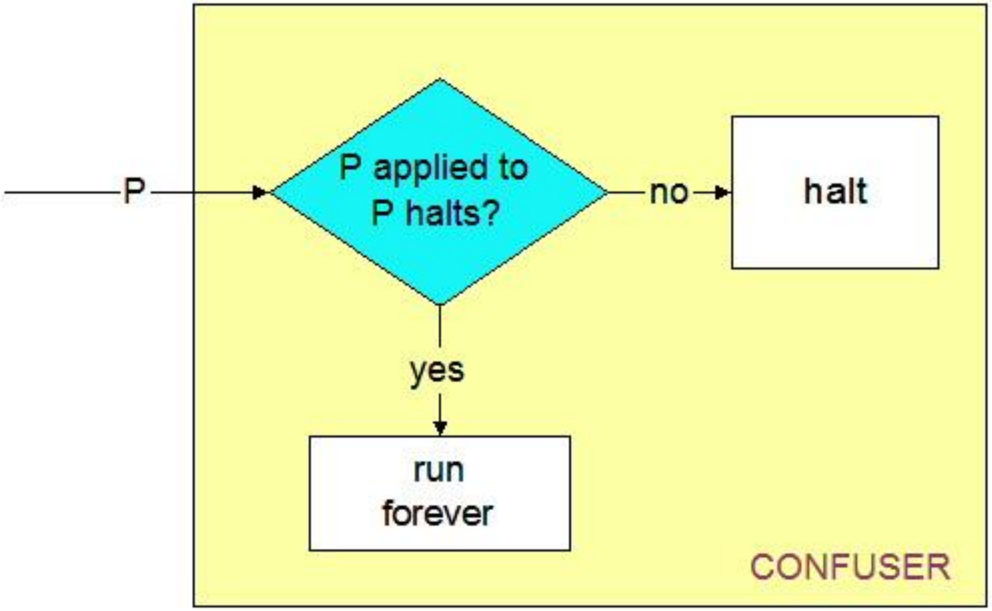
Reminder: The halting problem

Why not simply insist on termination anyway?

Can't we simply write a program that tests for termination?

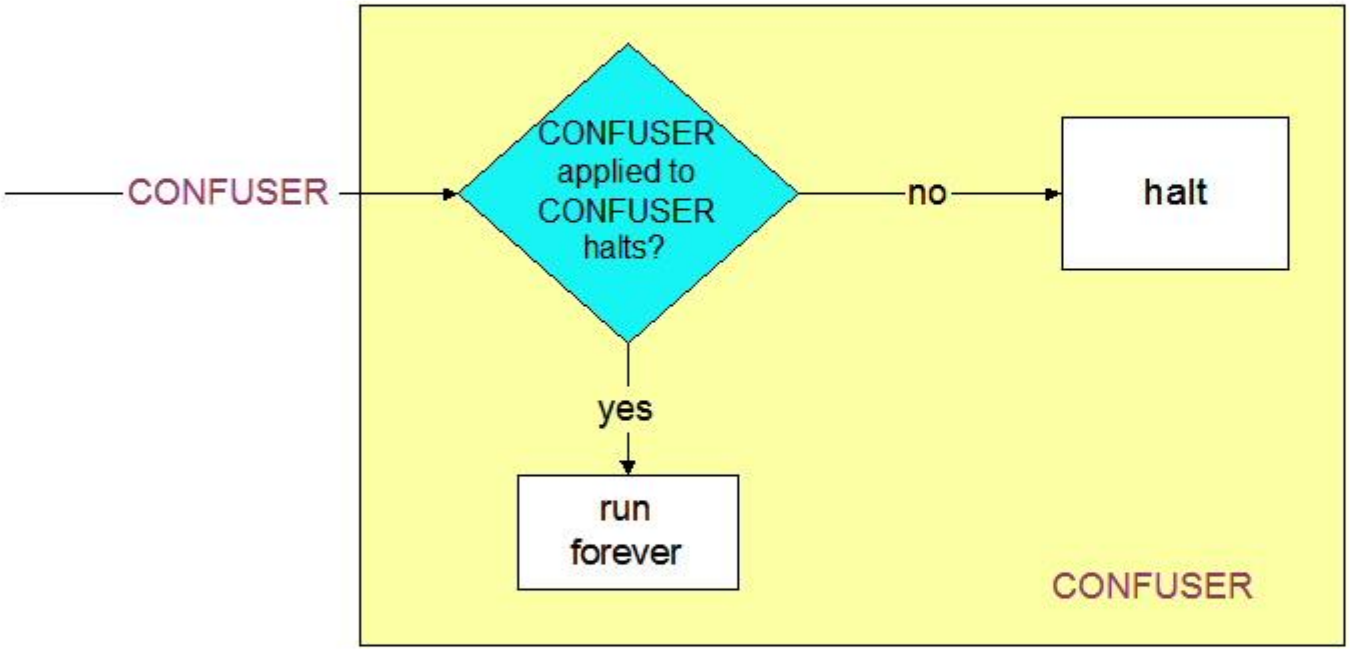


Suppose we could…



If we can write "Halts" we can use it to build the program "CONFUSER" shown above. Given a program P, it uses P both as a program and as P's own input.

Contradiction



If CONFUSER(CONFUSER) halts, it doesn't.
If it doesn't, it does.

THEREFORE: HALTS cannot be written.
We cannot decide in general whether a program will halt or run forever.

To describe a data type abstractly (without reference to any particular programming language), we list its Sorts, Syntax and Semantics. The semantics tell us the required behaviour.

ABSTRACT DATA TYPES

Sorts, Syntax, Semantics

- **Syntax**
  - What function and constant names are being defined?
- **Semantics**
  - How are the functions defined in terms of one another?
- **Sorts**
  - What other types need to be defined?

NB. **Algebraic** data types in Haskell are not the same as **abstract** data types (ADTs). ADTs are described mathematically, and are language-independent.

Example

Sorts --- Syntax --- Semantics

```
data BTree a = EmptyNode
             | Leaf a
             | Node (BTree a) (BTree a)
     deriving Show

treesum :: Num a => BTree a -> a

treesum t = case t of
  EmptyNode  -> 0
  Leaf x     -> x
  Node t1 t2 -> treesum t1 + treesum t2
```

Abstract Data Types (Maths)

We can define the behaviour of a data type using maths. This makes it independent of our choice of programming language.

```
ADT: IntStack

SORTS: Bool, Int

SYNTAX
  create  : IntStack
  push    : Int      -> IntStack -> IntStack
  top     : IntStack -> Int
  pop     : IntStack -> IntStack
  isEmpty : IntStack -> Bool
  length  : IntStack -> Int
```

Stack or Queue?

What really matters is the semantics . It isn't a stack just because it's called one. It has to have the right behaviour.

```
ADT: IntQueue

SORTS: Bool, Int

SYNTAX
  create  : IntQueue
  push    : Int      -> IntQueue -> IntQueue
  top     : IntQueue -> Int
  pop     : IntQueue -> IntQueue
  isEmpty : IntQueue -> Bool
  length  : IntQueue -> Int
```

Defining the semantics

- We want to describe the behaviour of the stack/queue, but how do we do it?
- We can't say things like "adding a 2 to an empty stack gives a stack containing the value 2; adding a 3 to the result gives a stack containing 2 and 3", because the syntax doesn't include any way to say "I am a stack containing 2 and 3".

When we can’t describe the outcome of applying a function, we say instead when two functions have the **same** outcome.

(forall x : Int) (forall s : IntStack)
(length(push x s) = 1 + length(s))

Construct, mutate, observe

- **CONSTRUCTORS**: Some functions tell us how to build new objects from old ones
  - **create** - create a new empty stack
  - **push** - add a value onto an existing stack
- **MUTATORS**: Some tell us how to change (mutate) an existing object
  - **pop** - remove the top element from the stack
- **OBSERVERS**: Some tell us how to observe the object's properties without changing it
  - **top** - what entry is at the top of the stack?
  - **isEmpty** - is the stack empty?
  - **length** - how long is the stack?

Constructor or mutator?

- It can be hard to decide sometimes whether a function is a constructor or a mutator.
- push/pop
  - both convert one stack into another
  - both return the resulting stack
- If push is a constructor, maybe pop is one too?

In practice it is simply a matter of opinion. You see the same confusion when looking at definitions of "Tree" in Haskell. Some programmers include "Leaf" as a constructor, some don't.

In these notes I generally minimise the number of constructors to avoid problems later

Defining the semantics

- First identify the constructors (create, push)
- The rest are non-constructors (pop, top, isEmpty, length)
- For each **constructor** c and **non-constructor** n, write down a formula describing the effect of applying n to c; **quantify** over the **variables**.

```
(forall x:Int)(forall s:IntStack)

   isEmpty(create) = true
   isEmpty(push x s) = false

   length(create) = 0
   length(push x s) = 1 + length(s)
```

Take note of error cases

```
(forall x:Int)(forall s:IntStack)

   pop(create) = errStack
   pop(push x s) = s

   top(create) = errValue
   top(push x s) = x
```

`errStack`
an error condition produced when trying to pop an empty stack

`errValue`
an error condition produced when looking at the top of an empty stack

Any missing definitions?

The values **true**, **false**, **errStack** and **errValue** are currently undefined. We need to include them, and adjust the relevant function signatures.

SYNTAX
true : Bool
false : Bool
errStack : StackError
errValue : ValueError

top : IntStack -> **(**Int **U ValueError)**
pop : IntStack -> **(**IntStack **U StackError)**

SORTS
Int
Bool
StackError
ValueError

Any knock-on effects?

We want expressions like **push (top s) (pop s)** to be meaningful, so the types need to match up.

```
SYNTAX
  true : Bool
  false : Bool
  errStack : StackError
  errValue : ValueError

  top : (IntStack U StackError) -> (Int  U  ValueError)
  pop : (IntStack U StackError) -> (IntStack U StackError)
  length : (IntStack U StackError) -> (Int  U  ValueError)
  push : (Int  U  ValueError) -> (IntStack U StackError) -> (IntStack U StackError)
  create :: IntStack
```

More knock-on effects?

The types have changed, so we need to extend our semantics to cover all cases

```
SEMANTICS
    top (create) = errValue
    top (push x s) = x
    top (errStack) = errValue

    length (create) = 0
    length (push x s) = 1 + length s
    length (errStack) = errValue
```

Notice that the semantics we end up with look very like Haskell code. This makes it relatively easy to implement ADTs in Haskell.

Implementation

Suppose we want to implement the data type IntStack. Looking at the syntax, we see that implementations are required for the following auxiliary types and functions/constants:

```
Int { 0, (1+) }
Bool { true, false }
StackError { errStack }
ValueError { errValue }

IntStack { top, pop, length,
             push, create, isEmpty }
```

If we're lucky, the programming language we use will already some of these defined for us. We have to define the rest, and check that the semantics are satisfied. The main problem will be error handling.

Error handling: Maybe

```
-- Int, Bool: built-in
-- Error handling: using Maybe types

data IntStack = Create | Push Int IntStack

top :: Maybe IntStack -> Maybe Int
top s = case s of
          Nothing          -> Nothing
          Just Create      -> Nothing
          Just (Push x _)  -> Just x
```

Using **Maybe** in Haskell lets us record whether the value we're using is properly defined, or is an error value. It cannot distinguish between different types of error.

Error handling: error

```
-- Int, Bool: built-in
-- Error handling: using "error :: String -> a"

data IntStack = Create | Push Int IntStack

top :: IntStack -> Int
top s = case s of
          Create     -> error "no such value"
          Push x _   -> x
```

Using **error** in Haskell lets us output an error message; the program then stops running. We can produce different messages for different errors.

Error handling: Either

```
-- Int, Bool: built-in
-- Error handling: using Either

data IntStack = Create | Push Int IntStack
data Error = ErrStack | ErrValue

top :: IntStack -> Either Int ValueError
top s = case s of
          Create    -> Right ErrValue
          Push x _  -> Left  x
```

Using **Either** lets us take the “union” of two types. We have to say whether the result belongs to the left-hand or right-hand type.

Error handling: unions

```
// int: built-in
// bool: same as int (0 = false)
// Error handling: using unions

union value { int val; char* msg; };

int top (intStack is, value &v) {
  int r = 0;
  if (length(is)) { r = 1; v.val = is.data[0]; }
  else { strcpy(v.msg, ”no such value”); }
  return r;
}
```

Using union lets us interpret the contents of a given memory location as a value belonging to any one of several types. We can amend the error message depending on the situation.

Error handling: special values

```
-- Int, Bool: built-in
-- Error handling: using special values

data IntStack = Create | Push Int IntStack

top Create     = -1
top (Push x _) =  x
```

If we know that certain values can never occur, we can use them to indicate error conditions. For example, if our IntStack will only ever be used to store positive integers, we can return the value -1 to let the user know that the result is meaningless.

PROBLEM: Someone else may use our code in a context we weren't expecting. The "special value" might occur naturally in situations we hadn't foreseen.

Error handling: Exceptions

```
// int, bool: built-in
// Error handling: using Exceptions

class IntStack {
  ...
  IntStack create() { return new IntStack(); }
  IntStack top() {
    if (this.isEmpty()) throw IntStackException();
    else ...
  }
  ...
}
```

Exception handling is a standard way to handle error conditions gracefully.

Program Proof (more later)

No matter what language we use to implement IntStack, we can prove basic facts that emerge from the semantics (provided the implementation is totally correct).

PROVE THIS

If s of type IntStack isn't empty, then

push (top s) (pop s) = s

EXERCISE

Step 1. Write down the semantics of the function **isEmpty**.

Step 2. We're told that **isEmpty(s)** is false. Deduce that **s** has the form **push x s'**.

Step 2. Calculate **top s** and **pop s**.

Step 3. Complete the proof.

Summary

- To establish correctness, we need to know whether an implementation simulates its specification; these may be described in different languages.
- How do we define behaviour in a language-independent way?
- ADTs (Abstract Data Types)
    - Sorts – what auxiliary types are involved?
    - Syntax – what constants and functions need to be defined?
    - Semantics – what rules do they need to satisfy?

EXERCISE

Write out the sorts, syntax and semantics of the IntStack ADT (a **stack** of integers). Using exactly the same sorts and syntax, write down the semantics for IntQueue (a **queue** of integers). How does IntQueue's semantics differ from those of IntStack?