Intro

We are going to focus on:

- type systems (specifications)
- type checkers (algorithms)

First we need to cover some preliminary material on:

- inductively defined sets
- inductively defined properties

Inductively Defined Sets

In this class you've seen the natural numbers defined like so:

```
ℕ ≜ {0,1,…}
```

This definition of the natural numbers is "informal". How do we know what really comes after the …? A more pedantic definition of the natural numbers would be as an inductively constructed set, using the same notation we have already used to construct syntax. Just like we have already seen:

```
e ∈ expr ::= i | e + e
```

we could define the natural numbers as:

```
n ∈ ℕ ::= 0 | succ 0
```

Here `succ` isn't an operation, it's a piece of uninterpreted syntax. The number three is then a linked list:

```
three ∈ ℕ ≜ succ (succ (succ 0))
```

Let's write the natural numbers one last time using a new type of funny notation:

```
┌─────┐
│n ∈ ℕ│
└─────┘


──────── (1)
0 ∈ ℕ

n ∈ ℕ
──────────── (2)
succ n ∈ ℕ
```

The first rules reads: "zero is a natural number." The second rule reads "*if* `n` is a natural number, *then* `succ n` is a natural number". These are called inference rules, and the general idea for reading them is "if all of the stuff above the line is true, then the thing below the line is true". Together these two rules (1–2) are equivalent to the writing:

```
n ∈ ℕ ::= 0 | succ n
```

So why would we use "inference rules"? They appear for now just a more verbose way of defining tree-like sets. Let's do a quick case study on trees.

Well-balanced Trees

Let's define (inductively) the set of trees:

```
t ∈ tree ::= L | ⟨t,t⟩
```

we are interested in *well-balanced* trees; for example this tree is well-balanced:

```
⟨⟨L,L⟩,⟨L,L⟩⟩
```

and these trees are not:

```
⟨⟨L,L⟩,L⟩

⟨L,⟨L,L⟩⟩
```

next we create an *inductively defined relation* using the inference rules described above, written `t △ n` which is read "tree `t` is well balanced at height `n`":

```
┌─────┐
│t △ n│
└─────┘


───── (leaf)
L △ 1

t₁ △ n    t₂ △ n
──────────────── (node)
⟨t₁,t₂⟩ △ (n + 1)
```

it is the case that if `t △ n` then `t` is well-balanced. We can write a *proof* of this fact directly using inference rules. E.g., to prove that this tree is well-balanced:

```
⟨⟨L,L⟩,⟨L,L⟩⟩
```

we can write this proof that it well-balanced at height 3:

```
─────   ─────   ─────   ─────  (leaf × 4)
L △ 1   L △ 1   L △ 1   L △ 1
─────────────   ─────────────  (node × 2)
  ⟨L,L⟩ △ 2       ⟨L,L⟩ △ 1
─────────────────────────────  (node)
     ⟨⟨L,L⟩,⟨L,L⟩⟩ △ 3
```

this is the primary reason why inference rules are written in this funny way–so we can stack them together to create a *proof tree* of some inductively defined property.

Scoping

Let's define an inductive relation for well-scoping for a tiny programming language. The relation is `S ⊢ e` and is pronounced "in a scope with `S` bound variable, program fragment `e` is well-scoped":

```
x ∈ var ≈ symbol
e ∈ expr ::= i | e + e
         | x | LET x = e IN e

S ∈ scope ≜ 𝜘(var)

┌─────┐
│S ⊢ e│
└─────┘

──────── (int)
S ⊢ i

S ⊢ e₁
S ⊢ e₂
──────────── (plus)
S ⊢ e₁ + e₂

x ∈ S
──────── (var)
S ⊢ x

S ⊢ e₁
{x}∪S ⊢ e₂
──────────────────── (let)
S ⊢ LET x = e₁ IN e₂
```

And we have a pair of theorems:

- If `S ⊢ e` then `FV(e) ⊆ S`
- If `FV(e) ⊆ S` then `S ⊢ e`

and a pair of corollaries:

- If `∅ ⊢ e` then `e` is closed.
- If `e` is closed then `∅ ⊢ e`.

(The proofs are by structural induction on the term `e`—we won't get into the proof details in this class.)

Simple Types (no variables)

Let's start with a simple type system for a programming language with just integers, plus, booleans, and if:

```
e ∈ expr ::= i | e + e
         | b | IF e THEN e ELSE e

τ ∈ type ::= int | bool
```

a *simple type system* for this language relates a term to a type, written `e : τ` and pronounced "program fragment `e` has type `τ`". The type system is defined inductively (using inference rules) and is designed such that if `e : τ`, then `e` is guaranteed to (1) not crash the interpreter due to a scope or runtime type error, and (2) return a value of type `τ` as the final result.

```
┌─────┐
│e : τ│
└─────┘


──────── (int)
i : int

e₁ : int
e₂ : int
──────────── (plus)
e₁ + e₂ : int

──────── (bool)
b : bool

e₁ : bool
e₂ : τ
e₃ : τ
──────────────────────── (if)
IF e₁ THEN e₂ ELSE e₃ : τ
```

The (if) rule is worth discussing. First, the use of the same `τ` above the line can be read to say "when `e₂` and `e₃` have the same type, and that type is `τ`, then the resulting type of the `IF` expression is that same `τ`". That is, it doesn't matter what type the result of the `IF` expression has, so long as it is the same as the sub-terms `e₂` and `e₃`.

Specification vs Algorithm

The relation `e : τ` defined above is an *inductively defined relation* which tells you that a good thing is true about `e` when `e : τ`. However, we are interested in an *algorithm* that *checks* whether or not `e : τ` for some `τ`. This algorithm is called a *type checker*, whereas the inductively defined relation is called a *type system*. The type checker is (1) algorithmic, and (2) should return results consistent with the definition of the type system (the specification).

Here is a type checker written in Haskell, with commented out code that used to be the interpreter. In particular notice the check in the `IF` case that "unifies" the types which are returned by `e2` and `e3`:

```
-- x ∈ var ≈ symbol
-- e ::= i
--   | e + e
--   | b
--   | IF e THEN e ELSE e
data Expr = IntE Integer
          | PlusE Expr Expr
          | BoolE Bool
          | IfE Expr Expr Expr
  deriving (Eq,Ord,Show)

-- v ∈ value ::= i
--           | b
data Value = IntV Integer
           | BoolV Bool
  deriving (Eq,Ord,Show)

-- τ ∈ type ::= int
--          | bool
data Type = IntT
          | BoolT
  deriving (Eq,Ord,Show)

check :: Expr -> Maybe Type
check e0 = case e0 of
  IntE i -> Just IntT
         -- Just (IntV i)
  PlusE e1 e2 ->
      case (check e1,check e2) of
        (Just IntT, Just IntT) -> Just IntT
        _ -> Nothing
   -- case (interp e1,interp e2) of
   --   (Just (IntV i1),Just (IntV i2)) -> Just (IntV (i1 + i2))
   --   _ -> Nothing
  BoolE b -> Just BoolT
          -- Just (BoolV b)
  IfE e1 e2 e3 ->
       case check e1 of
         Just BoolT -> case (check e2, check e3) of
           (Just t2, Just t3) -> if t2 == t3 then Just t2 else Nothing
           _ -> Nothing
         _ -> Nothing
    -- case interp e1 of
    --   Just (BoolV b) ->
    --     if b
    --     then interp e2
    --     else interp e2
    --   _ -> Nothing
```

Last updated: 4/2/2019