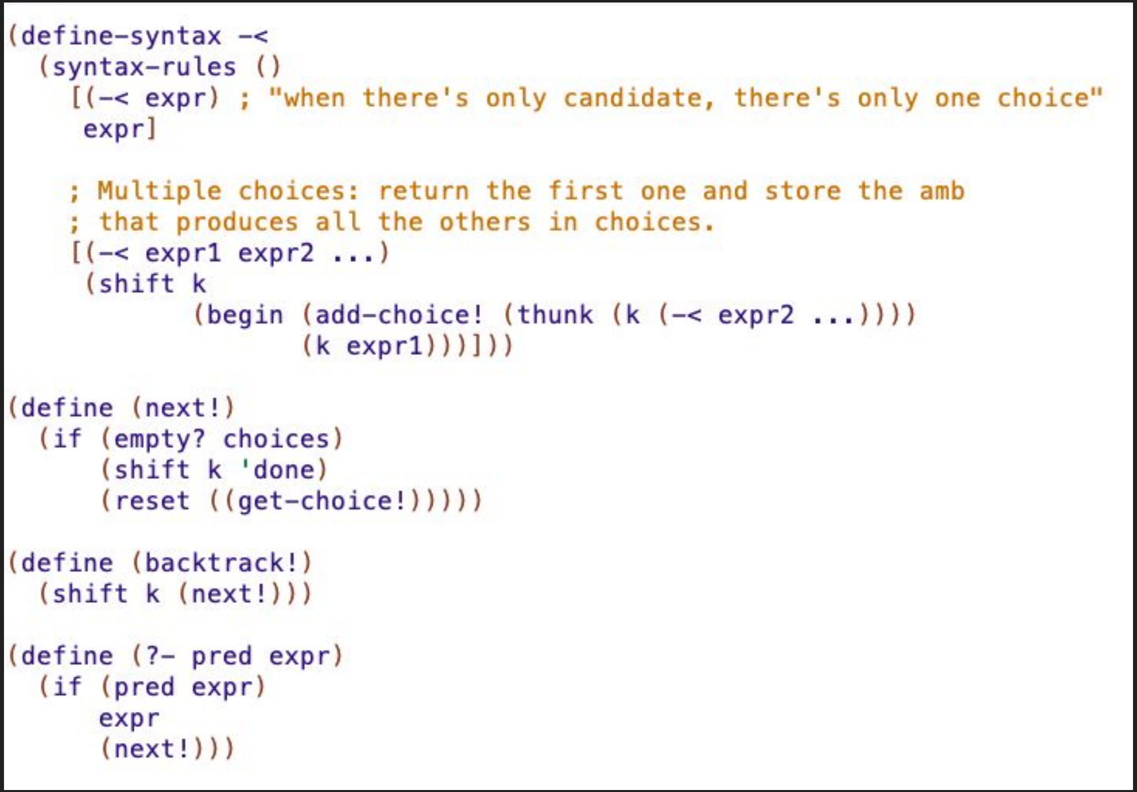
CSC324 Lecture 15

17 July 2020

Last time...

- We built all this mechanism… but what's it good for?



```
(define-syntax -<
  (syntax-rules ()
    [(-< expr) ; "when there's only candidate, there's only one choice"
     expr]

    ; Multiple choices: return the first one and store the amb
    ; that produces all the others in choices.
    [(-< expr1 expr2 ...)
     (shift k
            (begin (add-choice! (thunk (k (-< expr2 ...))))
                   (k expr1)))]))

(define (next!)
  (if (empty? choices)
      (shift k 'done)
      (reset ((get-choice!)))))

(define (backtrack!)
  (shift k (next!)))

(define (?- pred expr)
  (if (pred expr)
      expr
      (next!)))
```

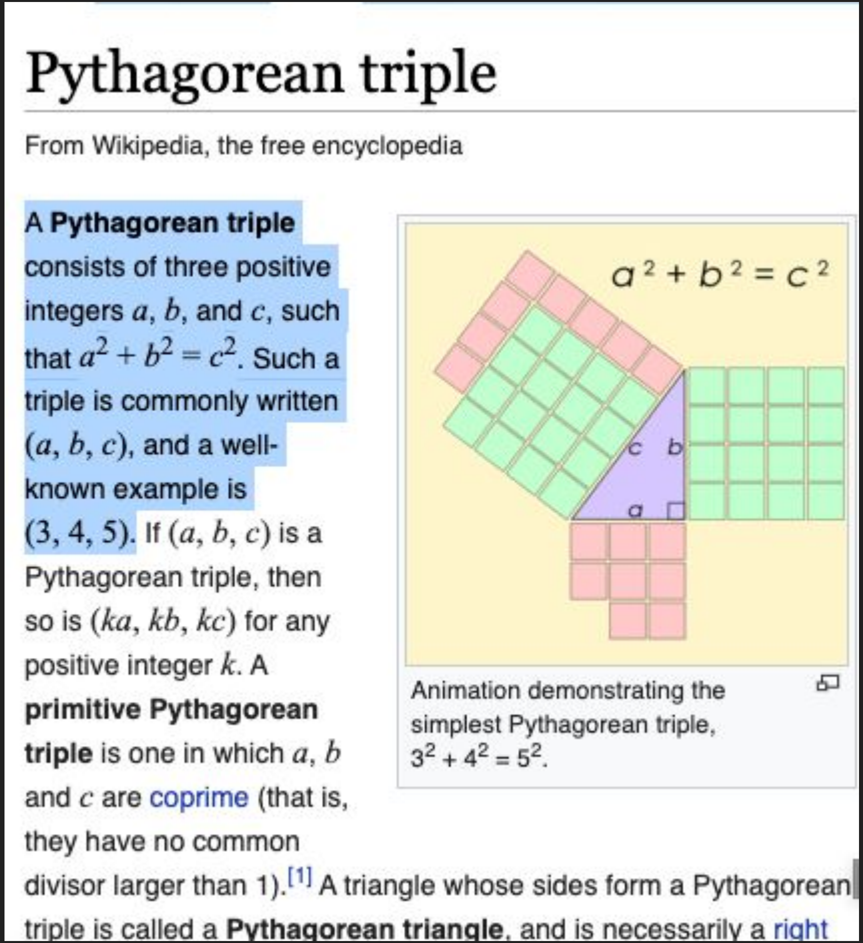
A worked example with amb

We will:

- Describe the problem we want to solve
- Express it in terms of -<
- Implement any missing pieces

Our goal is to move towards **declarative programming,** where we don't instruct the computer how to solve a problem but rather describe what the solution to the problem should look like, and have it solve it itself

Pythagorean triples



Pythagorean triple

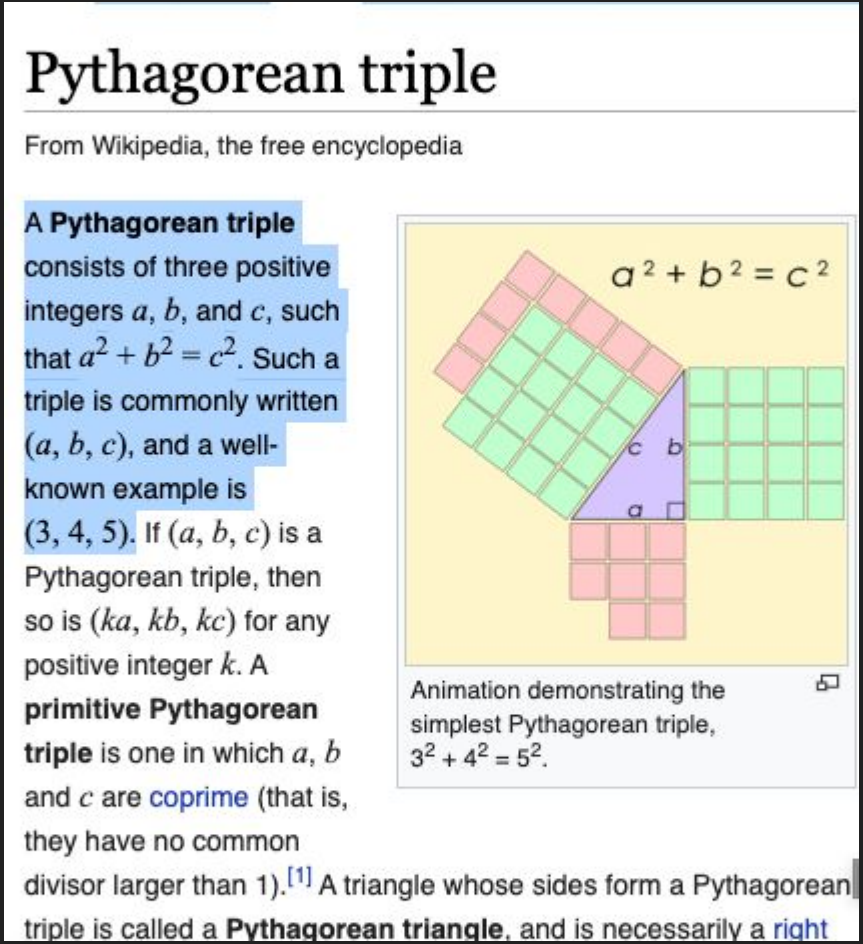
From Wikipedia, the free encyclopedia

A **Pythagorean triple** consists of three positive integers a, b, and c, such that $a^2 + b^2 = c^2$. Such a triple is commonly written (a, b, c), and a well-known example is $(3, 4, 5)$. If (a, b, c) is a Pythagorean triple, then so is (ka, kb, kc) for any positive integer k. A **primitive Pythagorean triple** is one in which a, b and c are coprime (that is, they have no common divisor larger than 1).[1] A triangle whose sides form a Pythagorean triple is called a **Pythagorean triangle**, and is necessarily a right

Animation demonstrating the simplest Pythagorean triple, $3^2 + 4^2 = 5^2$.

Pythagorean triples

"Computer, generate all pythagorean triples up to some point...



Pythagorean triple

From Wikipedia, the free encyclopedia

A **Pythagorean triple** consists of three positive integers a, b, and c, such that $a^2 + b^2 = c^2$. Such a triple is commonly written (a, b, c), and a well-known example is $(3, 4, 5)$. If (a, b, c) is a Pythagorean triple, then so is (ka, kb, kc) for any positive integer k. A **primitive Pythagorean triple** is one in which a, b and c are coprime (that is, they have no common divisor larger than 1).[1] A triangle whose sides form a Pythagorean triple is called a **Pythagorean triangle**, and is necessarily a right

Animation demonstrating the simplest Pythagorean triple, $3^2 + 4^2 = 5^2$.

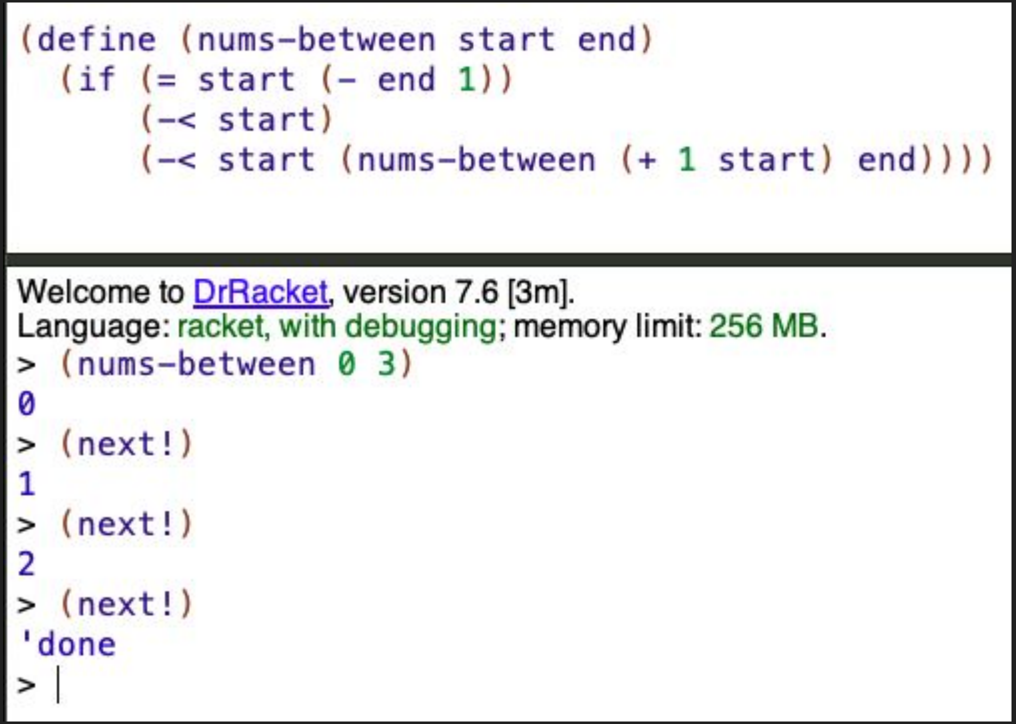
What do we need to solve this problem?

Decompose the problem into a "generate and filter" form:

- Have a way of producing all triple (a,b,c), where a,b,c are positive integers
  - This implies we need a way of producing all integers from 1 to n, which we already have!
- Have a way of testing whether some triple (a,b,c) satisfies (a^2 + b^2 = c^2)

What do we need to solve this problem?

- A way of producing all triple (a,b,c), where a,b,c are positive integers
  - **So we need a way of producing "all" positive integers**
- A way of testing whether some triple (a,b,c) satisfies (a^2 + b^2 = c^2)

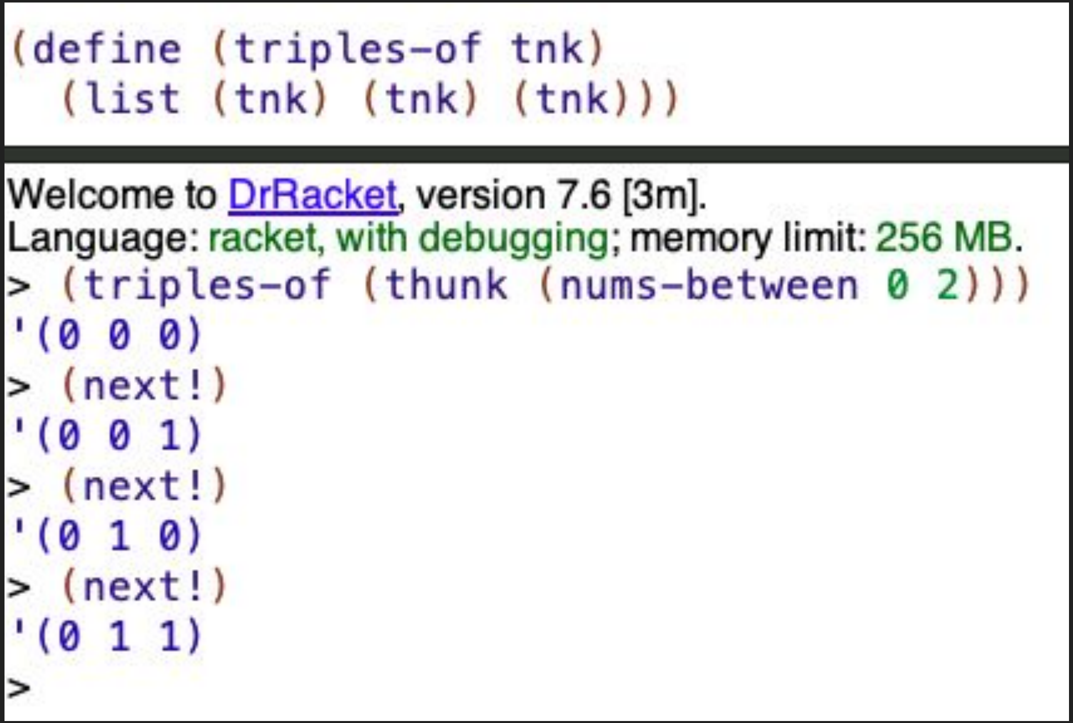


```
(define (nums-between start end)
  (if (= start (- end 1))
      (-< start)
      (-< start (nums-between (+ 1 start) end))))
```

```
Welcome to DrRacket, version 7.6 [3m].
Language: racket, with debugging; memory limit: 256 MB.
> (nums-between 0 3)
0
> (next!)
1
> (next!)
2
> (next!)
'done
> 
```

What do we need to solve this problem?

- **A way of producing all triples (a,b,c), where a,b,c are positive integers**
  - So we need a way of producing "all" positive integers
- A way of testing whether some triple (a,b,c) satisfies (a^2 + b^2 = c^2)



```
(define (triples-of tnk)
  (list (tnk) (tnk) (tnk)))
```

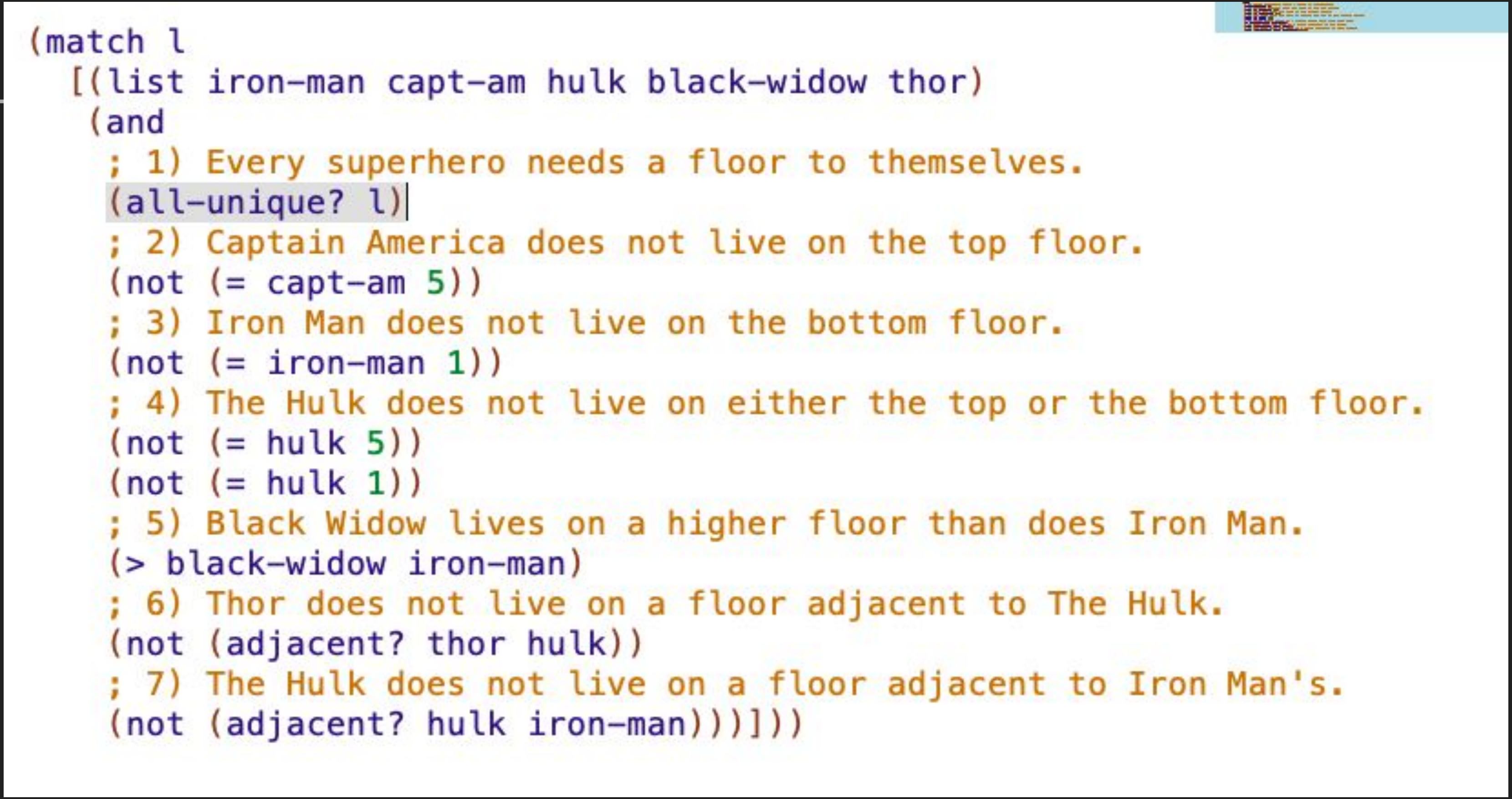
```
Welcome to DrRacket, version 7.6 [3m].
Language: racket, with debugging; memory limit: 256 MB.
> (triples-of (thunk (nums-between 0 2)))
'(0 0 0)
> (next!)
'(0 0 1)
> (next!)
'(0 1 0)
> (next!)
'(0 1 1)
>
```

Iron-Man, Captain-America, The-Hulk, Black-Widow, and Thor live on different floors of Stark Tower that contains only five floors.

Captain America does not live on the top floor. Iron Man does not live on the bottom floor. The Hulk does not live on either the top or the bottom floor. Black Widow lives on a higher floor than does Iron Man. Thor does not live on a floor adjacent to The Hulk's. Fletcher does not live on a floor adjacent to Iron Man.

Where does everyone live?

<T



```
(match l
  [(list iron-man capt-am hulk black-widow thor)
   (and
    ; 1) Every superhero needs a floor to themselves.
    (all-unique? l)
    ; 2) Captain America does not live on the top floor.
    (not (= capt-am 5))
    ; 3) Iron Man does not live on the bottom floor.
    (not (= iron-man 1))
    ; 4) The Hulk does not live on either the top or the bottom floor.
    (not (= hulk 5))
    (not (= hulk 1))
    ; 5) Black Widow lives on a higher floor than does Iron Man.
    (> black-widow iron-man)
    ; 6) Thor does not live on a floor adjacent to The Hulk.
    (not (adjacent? thor hulk))
    ; 7) The Hulk does not live on a floor adjacent to Iron Man's.
    (not (adjacent? hulk iron-man)))]))
```

Part 3: Type Systems

The third and final component of the course!!

Part 3: Type Systems

In Computer Science, we consider **Formal Methods** to be the study of tools to help ensure that a program behaves correctly with respect to some specification.

Some examples of formal methods:

- logic systems to express general correctness properties (more in CSC465)
- dynamic monitoring to detect when a subsystem is not behaving correctly
- lightweight syntactic methods like **type systems**

Definition: A **type system** is a tractable syntactic method for proving the absence of certain program behaviours by classifying expressions according to the kinds of values that they evaluate to.

A type system can be regarded as calculating a kind of "static approximation" of the dynamic (ie. at runtime) behaviour of the program.

(adapted from Pierce, B. *Types and Programming Languages*.)

Static vs Dynamic type systems

Definition: We say a type system is **static** if the aforementioned "static approximation" is computed before execution of the program occurs.

Definition: We can say that a type system that is checked while the program executes is **dynamic**, but the preferred term to **"dynamically-typed"** is often "**dynamically-checked".**

For our purposes, we are only considered with discussing static type systems.

Conservatism in Type Systems

By virtue of being **static** (ie. we know some function `int -> int` is called with some integer but cannot always know with which particular integer that will be), type systems are **conservative**.

That is to say: They may catch certain classes of errors...

- (ie. passing a string to a function that expects an int)

...but not all ...

- (ie. passing an int that might lead to a division by zero exception)

What type systems are good for:

By this point in your study of computer science, you should have some sense of what these reasons might be:

- Error detection
  - "You passed a string to the function that expects a number")
- Documentation
  - "Ah, I see that this function expects a number"
- *Are there any other reasons?*

What type systems are good for:

By this point in your study of computer science, you should have some sense of what these reasons might be:

- Error detection
  - "You passed a string to the function that expects a number")
- Documentation
  - "Ah, I see that this function expects a number"
- Abstraction
  - "Thinking in types forces me to decompose my program into meaningful subparts"
  - "Languages with more expressive type systems allow me to write more expressive programs"

Our goal in this class is to build your skills with the third point.

From set theory to type theory

Informally, we're used to thinking of types as **sets of objects**:

- "an Int is all integers from zero to 2^^32-1"
- "a string is an array of characters"

This is convenient because it gives us a mechanism to imagine typechecking, where we conceptually say "I have 42, and 42 is contained in the set of Integers, so therefore 42 is a valid Integer"

From set theory to type theory

However, thinking of types just as sets isn't ideal for a few reasons:

- Sets are definable but not necessarily constructable
- More expressive type systems need typechecking beyond checking "membership in a set"
    - And, indeed, higher-order types can suffer from Russell's paradox
    - Type theory was in fact invented partially to circumvent Russell's paradox!
- For the moment, a value only ever has one type, but set theory lets objects be members of an arbitrary number of sets.

(Note: when we get to polymorphism, I may refer to "the set of all types", which is different from "the values in some given type, contained in a set".)

Sets are definable but not necessarily constructable

$$\{ f \mid (f \in NP) \land (x \notin P) \}$$

Whether the above set is empty or not is the most famous open problem in computer science (the P=NP problem).

I can define the set without having a clue of how to figure out what is contained in the set!

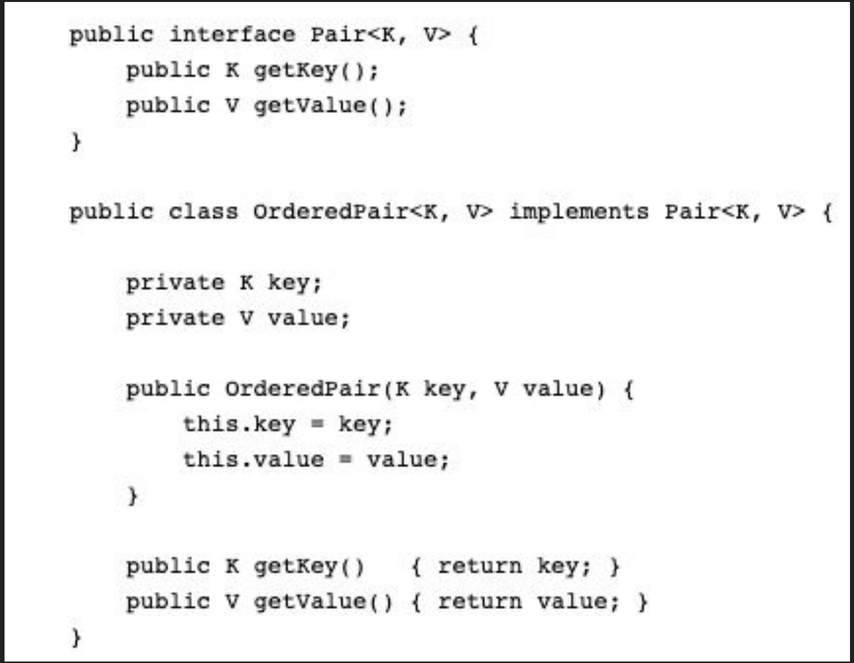
$$\{ f \mid (f \in NP) \land (x \notin P) \}$$

For our purposes, we need a formalism that is always going to be able to give us a tangible way of operating on types, so this suggests set theory isn't it.

"membership" in a set: Consider generics in Java

There isn't a single "set of all elements of the type" anymore, but now our types depend on type variables K and V.

We probably want to treat "the set of `Pair<String, String>`" differently than "the set of `Pair<Int, Int>`".



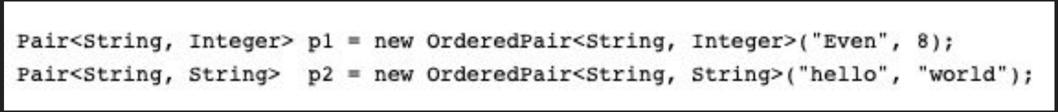
```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```

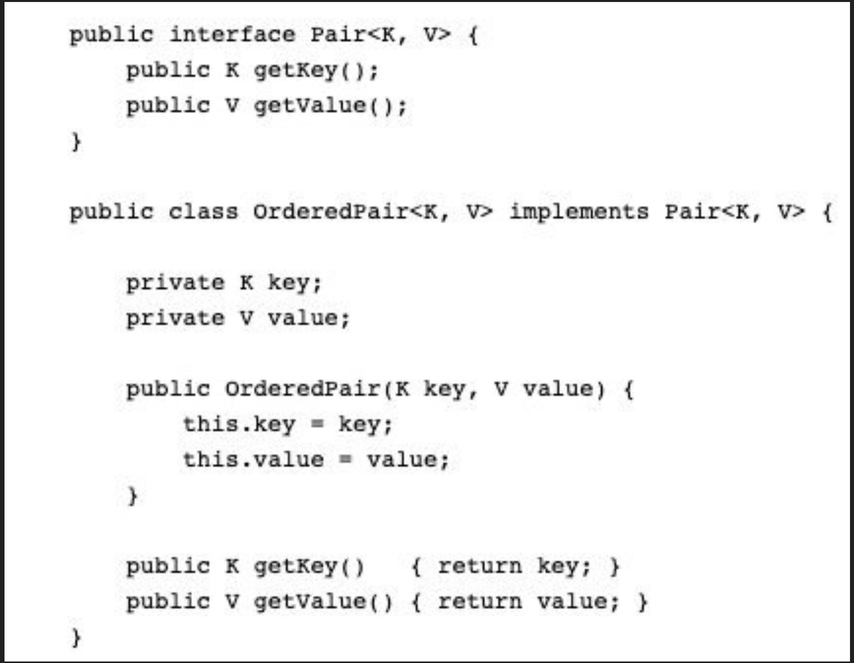


```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
```

Consider generics in Java

Do we extend this definition to "Pair<K,V> describes an infinite number of sets for each possible type K and V"?

And what rules do we use to compare the key and value types of an object against K and V?



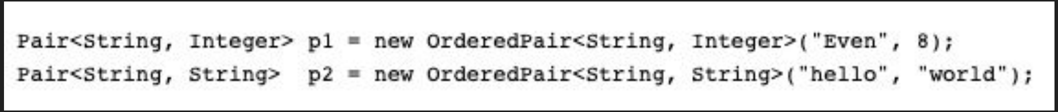
```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```



```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
Pair<String, String>  p2 = new OrderedPair<String, String>("hello", "world");
```

What is a type, then?

Definition: We say that, in some type system, some term t (aka an expression):

- `t : T`
- "t has type T"
- "t belongs to T"
- "t is an element of T"
- …

if there exist l**ogical implication rules** associated with the type that show:

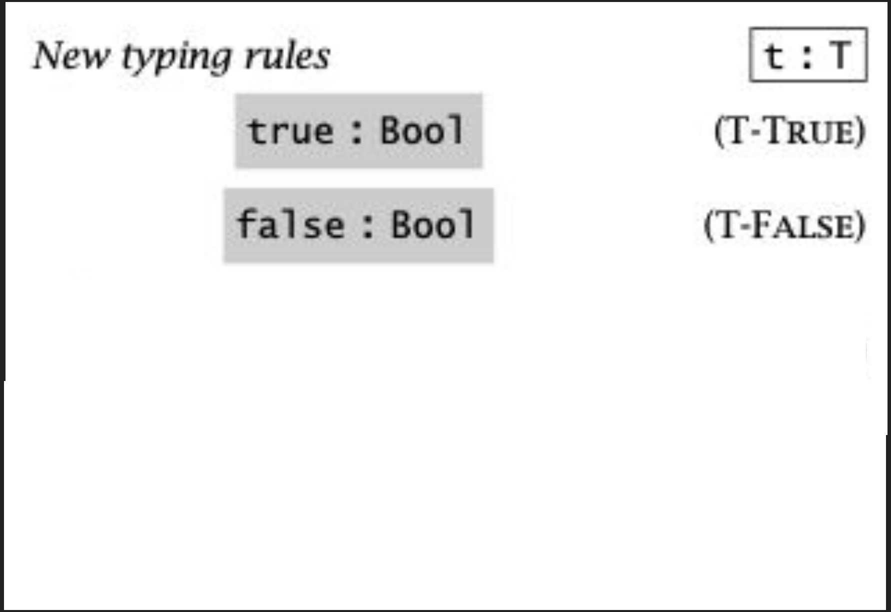
- that the term is already **axiomatically of type T**, or
- that repeatedly applying implication rules will produce a term of type T.

The type of booleans: Axioms

Here, we have the propositions that `true` and `false` are Booleans.

We read it as "true is of type Bool".

From our recursive definition, a term that isn't `true` or `false` must be able to be demonstrated that it would evaluate to a boolean axiomatic term. How do we do this?



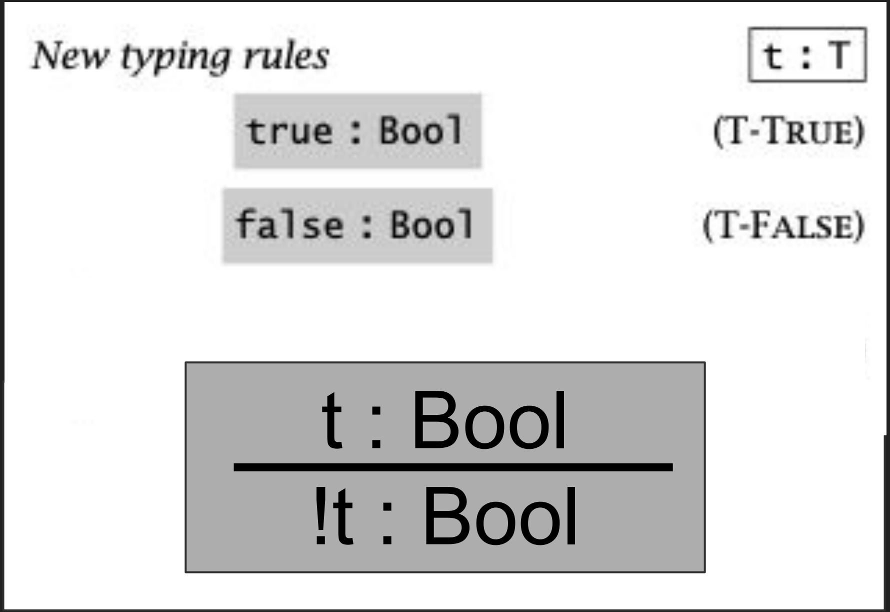
New typing rules $\boxed{\texttt{t : T}}$

`true : Bool` (T-TRUE)

`false : Bool` (T-FALSE)

The type of booleans: Typing rules

Recall: a **logical proposition** is a statement that is either true or false.



New typing rules $\boxed{\texttt{t : T}}$

$$\texttt{true : Bool} \qquad \text{(T-True)}$$

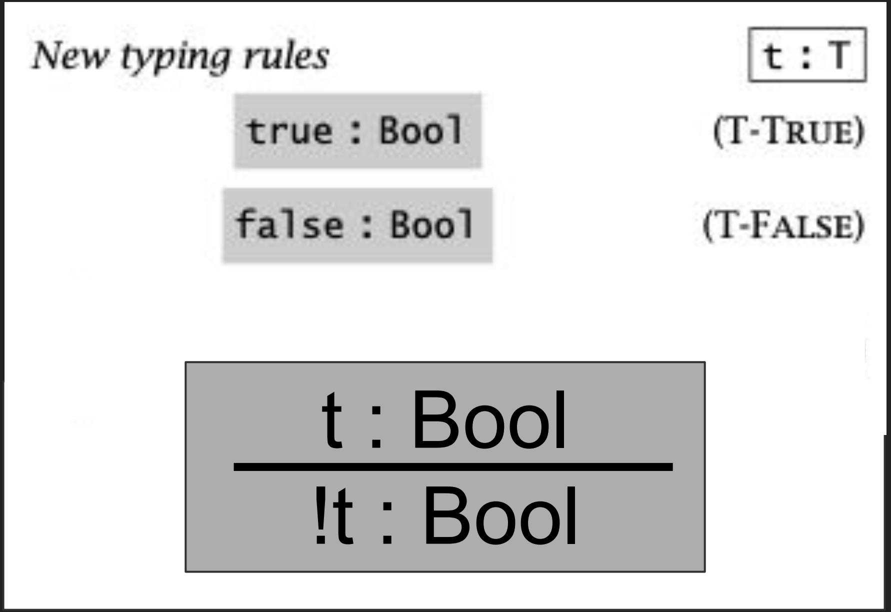
$$\texttt{false : Bool} \qquad \text{(T-False)}$$

$$\frac{t : \text{Bool}}{!t : \text{Bool}}$$

The type of booleans: Typing rules

A typing rule is a logical implication rule, where if every **premise** above the line holds, we can say that the **conclusion** below the line holds.

This says "if some term t is a Bool, then taking the negation of that t is a Bool as well."



New typing rules $\boxed{t : T}$

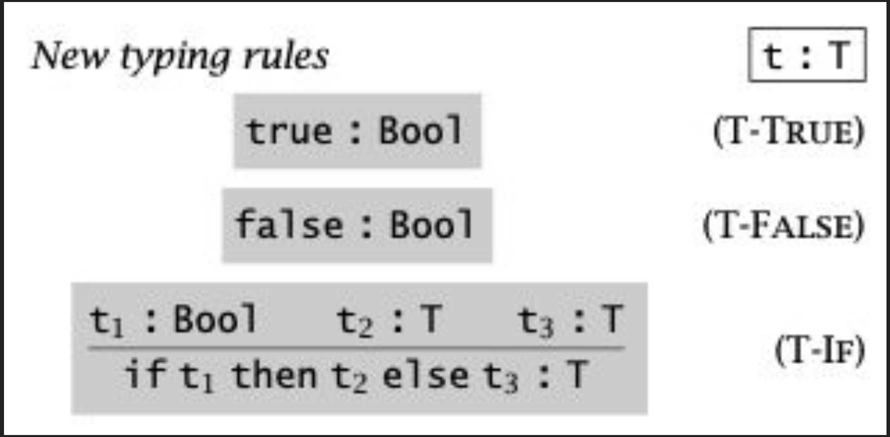
$$\texttt{true} : \texttt{Bool} \qquad \text{(T-TRUE)}$$

$$\texttt{false} : \texttt{Bool} \qquad \text{(T-FALSE)}$$

$$\frac{t : \text{Bool}}{!t : \text{Bool}}$$

The type of booleans: Typing rules

This says "if `t1` is a bool, and `t2` and `t3` are of the arbitrary type T, then the if-expression will evaluate to T"



New typing rules $\boxed{\texttt{t : T}}$

$$\texttt{true : Bool} \qquad \text{(T-TRUE)}$$

$$\texttt{false : Bool} \qquad \text{(T-FALSE)}$$

$$\frac{t_1 : \texttt{Bool} \qquad t_2 : T \qquad t_3 : T}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 : T} \qquad \text{(T-IF)}$$

The type of natural numbers

Here's an inductive way of defining the natural numbers:

a number is either

- the value zero or
- one more than a natural number.

Our one axiom is that the value `zero` is a Nat.

0 : Nat

The type of natural numbers

Here's an inductive way of defining the natural numbers:

a number is either

- the value zero or
- one more than a natural number.

Here's a typing rule that says "if some term t is a Nat, then whatever (add1 t) produces will also be a Nat

$$0 : \text{Nat}$$

$$\frac{t : \text{Nat}}{(\text{add1}\ t) : \text{Nat}}$$

The type of natural numbers

Here's an inductive way of defining the natural numbers:

a number is either

- zero or
- one more than a natural number.

Here's a typing rule that says "if some term t is a Nat, then whatever (zero? t) produces will be a Bool!

$$0 : \text{Nat}$$

$$\frac{t : \text{Nat}}{(\text{add1 } t) : \text{Nat}}$$

$$\frac{t : \text{Nat}}{(\text{zero? } t) : \text{Bool}}$$