First Order Predicate Calculus

$\nexists\sqrt{-1} \in$ TEAM

- Predicates
- Domains
- Quantifiers
- Substitutions
- Tying it all back together

Making complicated sentences

We've learned how to turn a sentence into logic.
If a and b then c and $\neg d$.

What if we want to talk about more complicated sentences?

- There is no largest prime.
- every planer graph is four colorable.
- There is a simple group of order 8×10^{53}
- Everything is awesome!

- $\nexists(y \in \mathbb{P}).\forall(x \in \mathbb{P}).x < y$.
- $\forall(g \in \mathit{Graphs}).\mathit{Planer}(g) \to \mathit{Colorable}(g, 4)$
- $\exists(G \in \mathit{Groups}).|G| = 8 \times 10^{53}$
- $\forall(t \in \mathit{Things}).\mathit{Awesome}(t)$

Making complicated sentences

to logically construct complicated sentences we need three ideas.

- Domains: What kinds of things are we talking about?
- Predicates: What can we say about these things?
- Quantifiers: How many of these things are there?

These aren't very good definitions, so lets fix that.

Domains

When I say forall x, what do I mean?

- Forall x where x is a number?
- Forall x where x is a string?
- Forall x where x is a chicken?
- Forall x where x anything at all?

This is important information

$$\forall (x \in \mathbb{N})(y \in \mathbb{N}).x \leq y \vee y \leq x.$$
$$\forall (x \in \Sigma^*)(y \in \Sigma^*).x \leq y \vee y \leq x.$$

The first statement is true. The second statement is not.
Domains give our statements meaning.

Domains

A domain is simply a set.
We usually have a specific set in mind.
$a^2 + b^2 = c^2$ doesn't make sense if a, b, c are graphs.

In Propositional logic our Domain was $\{\top, \bot\}$
Now, we'll let it be anything.

As a consequence we can't use truth tables anymore.
Try to draw the truth table for every possible thing ever.

Domains

Domains can be tricky to get right.

Is the domain well defined? $\{S : S \notin S\}$

Is the domain empty? $\{G : \sum_{v \in G} d(v) = 7\}$

are domains the same? $\{x : x \in \mathbb{N}, 2|x, n \in \mathbb{P}_{<n} \to n \not| x\}$ and $\{2\}$

These are all really set theory questions, so we'll ignore them.

Predicates

Predicates allow us to talk about properties and relations of variables.

We've seen several predicates already

- $a = b$
- $a < b$
- $a \in S$
- $Parent(a, b)$
- $Path(p_1, p_2 \dots p_n)$

Predicates

Formally:
A **predicate** is a name that has zero or more parameters.

The **arity** of a predicate is the number of parameters.

$P(v_1, v_2, \ldots v_n)$ is a predicate named P with arity n.

Informally a predicate is a function that returns a boolean.

Functions

Functions are like predicates, but they return a value.

A **function** is a name that has zero or more parameters and returns a value.

The **arity** of a function is the number of parameters.

$f(v_1, v_2, \ldots v_n) = \ldots$ is a function named f with arity n and returns $\ldots$.

Important! We can call a function inside a predicate.
We CAN NOT call a predicate inside a predicate
$P(v_1, f(v_2, v_3))$ OK
$P(v_1, Q(v_2, v_3))$ NOT OK

Back to Predicates

We want our predicate names to mean something in our domain.

$x = x$ is a binary predicate (predicate with arity 2).

If our domain is $\mathbb{R}$ then $x = x$ should be equality

But, it doesn't HAVE to be!

Quantifiers

We have two quantifiers
$\forall x. \dots$: for all x such that ...
$\exists x. \dots$: There exists x such that ...

To interpret this formula we need to pick a domain.

$\forall (x \in S). \dots$: for all x in S such that ...
$\exists (x \in S). \dots$: There exists x in S such that ...

We will often ignore these if the domain is clear, or doesn't matter.

Quantifiers: bound variables

Let's look at a formula.
$\forall x.P(x, y)$

There's something tricky going on here.

x is quantified in this formula, but y isn't.

We say a variable is **bound** if it is under a quantifier.
A variable is **free** otherwise.

Quantifiers: renaming

At any point we can *rename* the variable in a quantifier.
For example
$\forall x.P(x) \to P(x)$ is the same as $\forall z.P(z) \to P(z)$.

There is a catch.
We can't rename x if it is below another quantifier over x.
$\forall x.(\forall x.P(x))$ is NOT the same as $\forall z.(\forall x.P(z))$

We will denote replacing a variable x with z in a formula f with $f[x \mapsto z]$ (You may also see $f[x/z]$ or $f[z/x]$).

The **scope** of a variable is the nearest quantifier that defines it.
So, we only rename variables in the same scope.

ASTs in predicate logic

There's not a lot to say here.
It's similar to representing formulas in propositional logic.

$$\begin{array}{rcl} E & := & E \to E \\ & | & E \vee E \\ & | & E \wedge E \\ & | & \neg E \\ & | & \forall V.E \\ & | & \exists V.E \\ & | & \top \\ & | & \bot \end{array}$$

ASTs in predicate logic

There is one more thing we need to add.
We need predicates.

Remember a predicate can have variables, constants, and functions.
But a predicate can't have a another predicate inside of it.

We can solve this by adding another level.
a *term* is a variable, constant, or function that contains terms.

$$\begin{array}{rcl} T & := & V \\ & | & C \\ & | & f(T, T \ldots T) \end{array}$$

ASTs in predicate logic

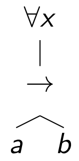
Putting this all together we have:

$$
\begin{array}{rcl}
E & := & E \to E \\
& | & E \vee E \\
& | & E \wedge E \\
& | & \neg E \\
& | & \forall V.E \\
& | & \exists V.E \\
& | & \top \\
& | & \bot \\
& | & P(T, T, \dots T) \\
\\
T & := & V \\
& | & C \\
& | & f(T, T \dots T)
\end{array}
$$

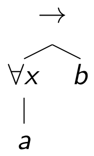
Where P is a predicate, f is a funciton, and C is a constant.

Predidence

All precidence from predicate logic remains the same.
The only difference is that $\forall$ and $\exists$ have the lowest precidence.
$\forall x.a \rightarrow b$ would give the tree



Not



If you want to have $\forall$ as a Subexpression, then you need parentheses.
$(\forall x.a) \rightarrow b$

Representing Formulas

So, what changes do we make to Python?

```
class Forall():
    def __init__(self, v, e):
        self.var = v
        self.expr = e

    def sub(self, x, v):
        raise SubException("Forall")

    def type(self):
        return Node.FORALL
```

The structure isn't too surprising.
A $\forall$ node has a variable, and an expression.

Representing Formulas

Predicates are also pretty easy

```
class Pred():
    def __init__(self, n, vs):
        self.name = n
        self.vars = vs

    def sub(self, x, v):
        raise SubException("Pred")

    def type(self):
        return Node.PRED
```

A *Pred* node has a name, and a list of variables.
We aren't dealing with functions, so a predicate can only have variables in it.

Substitution

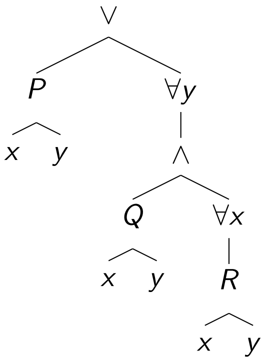
You may have noticed that one of the methods raise an exception.
A function that remains variables is called a substitution.

Part of your homework is to implement a substitution for ASTs.

Let's look at how this works.
We'll use the expression $P(x, y) \lor \forall y.Q(x, y) \land \forall x.R(x, y)$

Substitution

$P(x, y) \vee \forall y.Q(x, y) \wedge \forall x.R(x, y)$
We get the syntax tree

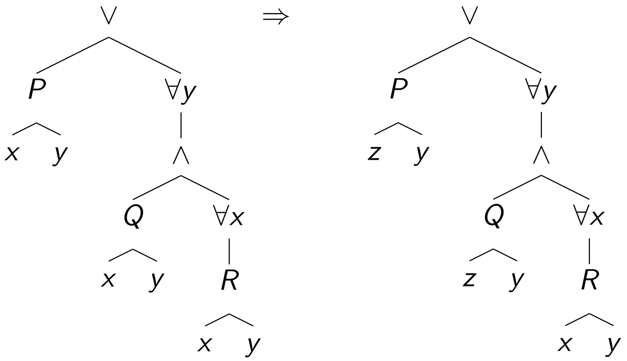


Remember $\forall$ goes until the end of the formula

Substitution

$P(x, y) \vee \forall y.Q(x, y) \wedge \forall x.R(x, y)$

Now I want to rename x to z.



We rename the x for P and Q,

but since R is below a $\forall$, we don't rename it.

Substitution

This gives us the following Substitution algorithm.
Remember $f[x \mapsto z]$ means rename x with z in f.

$x[x \mapsto z] = z$
$y[x \mapsto z] = y$
$P(x_1, \ldots x_n)[x \mapsto z] = P(x_1[x \mapsto z], \ldots x_n[x \mapsto z])$
$(a \wedge b)[x \mapsto z] = a[x \mapsto z] \wedge b[x \mapsto z]$
$(a \vee b)[x \mapsto z] = a[x \mapsto z] \vee b[x \mapsto z]$
$(a \rightarrow b)[x \mapsto z] = a[x \mapsto z] \rightarrow b[x \mapsto z]$
$(\neg a)[x \mapsto z] = \neg a[x \mapsto z]$
$(\forall y.a)[x \mapsto z] = \forall y.a[x \mapsto z]$
$(\forall x.a)[x \mapsto z] = \forall x.a$
$(\exists y.a)[x \mapsto z] = \exists y.a[x \mapsto z]$
$(\exists x.a)[x \mapsto z] = \exists x.a$

Substitution

Let's look at a couple cases in Python
$(a \wedge b)[x \mapsto z] = a[x \mapsto z] \wedge b[x \mapsto z]$

In the case for $\wedge$, we just rename both the left and right hand sides.

```
class And():
    ...
    def sub(self, x, v):
        return And(self.lhs.sub(x,v), self.rhs.sub(x.v))
    ...
```

We rename the left, and rename the right, and put them together.

Notice that we're not modifying anything here,
we're actually creating an entirely new node.
We'll end up copying the whole tree.

Substitution

Let's look at a couple cases in Python
$P(x_1, \ldots x_n)[x \mapsto z] = P(x_1[x \mapsto z], \ldots x_n[x \mapsto z])$

This is the hardest case (although, it's really not too bad)
We want to rename each variable in the list `self.vars`
There's a cool python feature that'll help here.

In Python `[f(x) for x in list]` will apply `f` to every element in a list. So, we just need to make a function `f`.

```
[abs(x) for x in [-2,-1,0,1,2]] == [2,1,0,1,2]
```

Substitution

Let's look at a couple cases in Python

$P(x_1, \dots x_n)[x \mapsto z] = P(x_1[x \mapsto z], \dots x_n[x \mapsto z])$

```
class Pred():
    ...
    def sub(self, x, v):
        def f(n): # possibly renames a variable
            if n == x:
                return v
            else:
                return n
        return Pred(self.name, \
                    [f(var) for var in self.vars])
    ...
```

We make a new Pred with the same name.
We just renamed all of the variables.

Quantifiers: common bugs

The order of quantifiers is important!
A woman gives birth once every 7 seconds in the US.

option 1: $\forall t \in 7\ \mathit{Seconds}\ \exists w \in \mathit{Women}.\mathit{GivesBirth}(w, t)$
option 2: $\exists w \in \mathit{Women}\ \forall t \in 7\ \mathit{Seconds}.\mathit{GivesBirth}(w, t)$

One of these is going to end very badly for some poor woman!

You can only merge quantifiers if their variables are disjoint.

$\forall x.(\exists x.\neg x) \wedge x$ is NOT $\forall x \exists x.(\neg x \wedge x)$

If you have a case like this, then rename one of the variables.

Tying it together: making sentences

Now that we've seen what we can do with quantifiers
let's practice.

- There is no largest natural number.
- There is no smallest real number.
- induction
- There is no program that can tell us if another program is correct.
- a program that doens't typecheck has a bug.