These are my notes for my 3TP3 "Truth and Provability" course at McMaster university, 2012.

The course textbook is Hofstadter's "Gödel-Escher-Bach".

These notes roughly follow Hofstadter's exposition of basic logic and the proof of Gödel's Incompleteness Theorems, adding a bit of rigour and some further details where mandated.

The course is loosely based on notes from Matt Valeriote's version of the course, which he taught in 2008 and 2010, and owe much to him.

The notes are written in plaintext, and should be read with a fixed-width font.

The pdf versions have some automatic highlighting of things which look like mathematical expressions, based on a simple regexp... this isn't very reliable, but kind of works.

[Comments that occur in square brackets, like these, are things I don't intend to say in class and are intended as bonus extra material for those who care to read these notes, or as reminders to myself]

-- Martin Bays, McMaster University, 2012

I: Formal systems

Definition:

An _alphabet_ is a finite set of _symbols_ (or _letters_ or **_characters_**).
e.g. {'a','b',...,'z'}

A _string_ (or _word_) in an alphabet **\Sigma** is a finite sequence of elements of **\Sigma**.
e.g. "aardvark", "word"

A _formal system_ on **\Sigma** comprises:
- a finite set of strings in the alphabet, called the **_axioms_**;
- a finite set of _production **rules_**.

A _derivation_ in a formal system is a finite sequence of strings (the _lines_ of the derivation) such that each line is an axiom or can be produced by a production rule from some preceding lines.

A string is a _theorem_ (or **_production_**) of a formal system if it is the last line of a derivation.

The _length_ of a derivation is the number of lines it has.

Before we define what production rules are, we must define patterns.
A _pattern_ in **\Sigma** is a string in the alphabet you get by adding to **\Sigma** some new symbols called _variables_ (as many of them as we need). We'll write these variables 'x', 'y', 'z', and use subscripts **'x_2'** and so on if we need more.

So if the original alphabet **\Sigma** is { '-', 'p', 'q' }, then patterns are strings like "-xyp--x".

A _production rule_ comprises:
- **A** finite sequence of patterns in **\Sigma**, called the **_inputs_**;
- **A** single pattern, called the output. Each variable appearing in the output pattern must appear in at least one input pattern.

To define how production rules are applied, we should first define matching.

To _match_ a pattern to a string in **\Sigma** means to find strings in **\Sigma** which can substitute for the variables in the pattern so as to produce the string. e.g. "-xyp--x" matches "---qp----" by substituting "--" for "**x**" and "**q**" for "**y**".

To match a sequence of patterns to a sequence of strings means to match each pattern to the corresponding string, with the same substitutions being made when the same variable appears in more than one pattern. For example, **("-xyp--x","xy")** matches **("---qp----","--q")**.

Finally, to apply a production rule to a sequence of strings means to match the input patterns to the strings, and produce as output the output pattern with variables substituted for strings according to the substitutions made in the matching. For example, the rule

("-xyp--x","xy") |-> "-yypx"

could be applied to the sequence of strings **("---qp----","--q")**. to produce "-qqp--".

Note that sometimes there will be more than one way to match the given strings to the input patterns, resulting in different outputs. For example, the simple rule

"xxy" **|->** "**y**"

when applied to the string "--p--pq-" could produce "**p**--pq-", but it could also produce "**q**-".

Remark:
This notion of formal system is due to Emil Post.
We will sometimes refer to them as "Post formal systems", when we want to be clear that we have this precise definition in mind.
The systems described in Hofstadter do not always fit rigidly into this definition; in these notes, I aim to explain how they can be tweaked so as to do so.

The MIU-system

```
The MUI-system:
    Alphabet: {'M', 'I', 'U'}
    Axioms: {"MI"}
    Production rules:
        (I)     xI   |->  xIU
        (II)    Mx   |->  Mxx
        (III)   xIIIy  |->  xUy
        (IV)    xUUy   |->  xy
```

MU-puzzle: is "MU" a theorem?

```
Example: the following is a derivation in the MIU system:
    1.  MI
    2.  MIU         (produced by rule (I) from line 1)
    3.  MIUIU       (by (II) from 2)
    4.  MIUIUIUIUI  (by (II) from 3)

    so "MIUIUIUIUI" is a theorem of the MIU system.
```

```
Example: the following is a derivation in the MIU system:
    MI
    MII
    MIIII
    MUI     (by (III) with x="M", y="I")
    MUIU
    MUIUUIU
    MUIIU
    MUIIUUIIU
    MUIIIIU
    MUIUU
```

Remark:
If we cut a derivation short, taking just the first **n** lines, what we have is also a derivation. So every line of a derivation is a theorem.

IU-puzzle: is "IU" an MIU-theorem?

Theorem: any MIU-theorem starts with 'M'
Proof by induction on the length of a derivation:
We show that for every natural number **k**

(*)_k every theorem with a derivation of length **<=k** starts with **M.**

(***)_0** is trivially true, as there are no theorems with derivations of length **<=0**!

Assume (***)_k**, and consider a derivation of length **k+1**.

Each of the first **k** lines have derivations of length **<=k**, so they all start with 'M'.

The last line is an axiom or is produced from a previous line by one of (**I)-(IV)**.

If it is an axiom, it is "MI", which starts with 'M'.

If it was produced by (I) xI **|->** xIU:
"xI" starts with 'M', hence **x** does, hence "xIU" does.

Similar arguments apply for (**II)-(IV)**.

So (***)_{k+1}** holds.

Example: The **MIU+** system is formed by adding a new production rule
(**V**) (MUx, MUy) **|->** MUxy

A derivation in this system:

```
1. MI
2. MII
3. MIIII
4. MUI
5. MUII         (by (V) from (4) and (4))
6. MUIII        (by (V) from (4) and (5))
```

Deciding theoremhood

Question: which strings in {'M', 'I', 'U'} are MIU-theorems?

First answer: those for which there exist derivations.

This is unsatisfactory!

We would like a _decision procedure_ for theoremhood:
a **procedure/algorithm/program** which we can carry out on any string, and which will (eventually) stop and give us an answer "yes" or "no", and which answers "yes" iff the string is a theorem.

We have "half" of that:
given a string, we can run through all possible derivations in order of length (see below), and stop with answer "yes" if the last line is equal to the given string.

This is a _semi-decision procedure_ for theoremhood:
an algorithm which, given a string **S**, answers "yes" if **S** is a theorem, but needn't stop at all if **S** isn't a theorem!

Algorithm to produce all derivations of a formal system, in order of length:
The only derivation of length 0 is the empty derivation.
Suppose we have produced all derivations of length **k**. To produce all derivations of length **k+1**:

- For each axiom and each length **k** derivation:
  append the axiom to the derivation, giving a length **k+1** derivation.
- For each production rule and each length **k** derivation:
  Say the production rule takes **n** strings as input.
  Run through each set of **n** lines from the derivation, and all the (finitely many!) ways to apply the production rule to them (choices for substitutions of variables). In each case, append the output, giving a length **k+1** derivation.

Remark:
For this argument to work, it's crucial that there be only finitely many

```
    axioms and finitely many production rules.

Remark:
    If we remove rules (III) and (IV) of the MIU-system, we have an easy
    decision procedure: each rule increases the length of a string it acts on,
    so if a string S is a theorem it has a derivation of length at most the
    length of S. So just check all those derivations.

Why do we call the above procedure a "semi"-decision procedure?
    Suppose we find a formal system Anti-MIU whose theorems are precisely the
    non-theorems of the MIU-system. Then we would have a decision procedure
    for MIU-theoremhood:
        Given a string S, *simultaneously* run our semi-decision procedures
        for MIU and for Anti-MIU.
        The first stops and says "yes" if S is an MIU-theorem;
            the second stops and says "yes" if S is an Anti-MIU-theorem, i.e.
            if S is *not* an MIU-theorem.
        So precisely one of them will eventually stop and say "yes"!
        Then we stop, and say "yes" or "no" appropriately.

We'll come back to this idea later.

Solution to the MU-puzzle
-------------------------

Definition: For an MIU-string S, let I(S) be the number of occurences of 'I'.

Theorem: If S is an MIU-theorem, then I(S) is not divisible by 3
    (i.e.  I(S) != 0 mod 3)
Proof:
    By induction on length of derivations.
    Suppose  I(S') !=~ 0 mod 3  for any theorem S' having a derivation of
        length <= k, and suppose S has a derivation of length k+1.
    If S is an axiom, S="MI" so  1 = I(S) !=~ 0 mod 3.
    Else, S is produced by one of (I)-(IV) from some S' with I(S') !=~ 0 mod 3.
        (I): I(S) = I(S').
        (II): I(S) = 2I(S'), so I(S) =~ 2I(S') !=~ 0 mod 3.
        (III): I(S) = I(S')-3, so I(S) =~ I(S') !=~ 0 mod 3.
        (IV) I(S) = I(S').
    So I(S) !=~ 0 mod 3.

See assignment 1 for the converse.

Semantics
---------

The pq-system:
    Alphabet: {'p', 'q', '-'}
    Axioms: {"-p-q--"}
    Production Rules:
        (I) xpyqz |-> xpy-qz-
        (II) xpyqz |-> x-pyqz-

Producing some theorems, it looks like every theorem is of the form
    "-^np-^mq-^{n+m}" (where "-^n" abbreviates n dashes).
So it's tempting to *read* e.g. "---p--q-----" as "3 plus 2 equals 5".

Is that what it "really means"?
Is "---p--q-----" *true*, and "--p--q-----" *false*?
What about "qpqpqq-"?

Definition:
    A _language_ in an alphabet is a set of strings, called the _well-formed
        strings_ (wfss).
    An _interpretation_ of a language is a way to assign a truth value (True
        or False) to each wfs.

So here, we're suggesting a language where the wfss are "-^np-^mq-^k" with
    n,m,k>=1, and the plus-equals interpretation:
        "p"  -->  "plus"
        "q"  -->  "equals"
        "-"  -->  "one"
        "--" -->  "two"
```

```
        "---" --> "three"
        etc;
    so e.g. we assign True to "--p---q-----" because "three plus two equals
        five" is true.
```

We were led to this interpretation by noting that all theorems appeared to be true according to it.

Definition: **A** formal system is _consistent_ (or _**sound**_) with respect to an interpretation if all its theorems are wfss and are true under the interpretation.

Theorem:
    The pq-system is sound wrt the plus-equals interpretation.
Proof:
    The axiom "-p-q--" is true, since **1+1=2**.
    The production rule (I) preserves truth of wffs:
        if "**-^np-^m-^k**" is true, then **k=n+m**,
        so "**-^np-^m-q-^k-**" = "**-^np-^{m+1}q-^{k+1}**" is true,
            since **k+1 = (n+m)+1 = n+(m+1)**.
    Similarly, so does (II).
    So (by an induction on length of derivations) every theorem is true.

Caution: "two plus three plus one equals six" makes sense, but
    "--p---p-q------" is ***not*** well-formed!

Remark:
    Consider
        "**p**" **-->** "equals"
        "**q**" **-->** "subtracted from"
        "-" **-->** "one"
        etc.
    This gives wfss the same truth values as the plus-equals interpretation.
    Does that mean it's the ***same*** interpretation? This question is of no importance to us, and we will not give an answer.

Remark:
    Consider the **plus-at_least** intepretation:
        "**p**" **-->** "plus"
        "**q**" **-->** "at least"
        "-" **-->** "one"
        etc.
    The pq-system is also consistent wrt this interpretation!
    But...

Definition:
    A system is _complete_ with respect to an interpretation if every wfs which is true according to the interpretation is a theorem of the system.

Example: The pq-system is ***not*** complete with respect to the **plus-at_least** interpretation. Indeed, "-p-q-" is clearly not a theorem.

Theorem: The pq-system ***is*** complete wrt the plus-equals interpretation.
Proof:
    We want to show that for any **n,m>=1**, **-^np-^mq-^{n+m}** is a theorem.
    But indeed, **n-1** applications of (I) starting with the axiom yields "**-^np-q-^{n+1}**", and then **m-1** applications of (II) yields "**-^np-^m-^{n+m}**".

So the pq-system "captures" addition of two positive numbers.

More arithmetic in formal systems

Big question:
    Can we find a language which we can interpret as making interesting statements in mathematics, and a complete consistent formal system for it?

Examples of the kinds of "interesting statements" we might want to express:
    2+2 = 5   (we've got this covered, thanks to the pq-system!)
    3*7 = 21
    4^3 = 64
    2+2 != 5

2*3 = 7 or 2 *** 3 = 6**
1337 is prime
For any integer **n**, **n*1 = n**
Every even number is the sum of two primes

Let's see what we can do!

The tq−system:
Alphabet: **{t,q}**
Axiom: −t−q−
Rules:
(I) xt−qz **|->** −xt−qz−
(II) xtyqz **|->** xty−qzx

Language: **−^nt−^mq−^k**
Interpretation: **−^nt^mq−^k --> n*t=k**

Soundness:
The axiom is true (**1*1=1**)
(I) and (II) preserve truth:
(I): **n*1 = m => (n+1)*1 = m+1**
(II): **n*m = k => n*(m+1) = k+n**
Completeness:
If **n*m=k**, we derive **−^nt−^mq−^k** from **−^nt−q−^n** by applying (II) **m−1** times, and we derive **−^nt−q−^n** from the axiom −t−q− by applying (I) **n−1** times.

So the tq−system "captures" multiplication of two positive integers.

Compositeness:
Add to the tq system a character 'C' and a rule of inference
xqy **|->** Cy
and interpret **C−^n** as "**n** is composite".

Completeness and soundness are easily checked.

Primeness:
Can we find a system where **P−^n** is a theorem iff **n** is prime,
i.e. iff **n** is ***not*** composite?
The system for compositeness is no use to us here!
(cf trying to find an anti−MIU system given only the MIU system...)
We have to develop a new system.

First, we capture "**n** does not divide **m**":
Axioms: −−DND−
Rules:
xDND− **|->** **x**−DND−
xyDNDx **|->** xy−DNDx−
xDNDy **|->** xDNDxy
Interpretation:
−^nDND−^m ---> n does not divide **m**
(i.e. **m !=~ 0** mod **n**)

(the first two rules give **−^nDND−^m** as a theorem whenever **n>m**)

Secondly, we capture "**n** has no divisors among 2,3,...,**m**", which we can phrase as "**n** is Divisor Free up to **m**". Add the rules
−−DNDx **|->** xDF−−
(yDFx, **x**−DNDy) **|->** yDFx−

Finally, add a rule:
x−DFx **|->** Px−
and an axiom:
P−−

II: Propositional logic

Examples:
Socrates is a man or Socrates is a woman.
Socrates is not a woman.
Therefore: Socrates is a man.

    If Socrates is a vampire and vampires are immortal, then Socrates is
        alive.
    Socrates is not alive.
    Therefore: Either Socrates is not a vampire, or vampires are not immortal.

We will develop a formal system, the _propositional **calculus_**, implementing this kind of logic.

Our system ***won't*** have strings interpreted as "Socrates" or "is a vampire" (we'll have to wait for the predicate calculus for that!). Rather, we use _propositional variables_ to stand in for whole propositions - e.g. **P** could stand for "Socrates is a vampire". For our purposes, a _proposition_ is just something which is true or false.

The language of propositional logic

Alphabet: **<**, **>**, **P**, **Q**, **R**, ', /\, \/, **=)**, **~**
(Note: I'm using "**=)**" as an ascii representation of the horseshoe character)

Well-formedness:
Well-formed strings in propositional logic are called _well-formed formulas_ (wffs).

Rules to determine well-formedness:
- "**P**", "**Q**" and "**R**" are well-formed, as are "**P'**", "**R''**" and so on. These are the _propositional **variables_**. We also refer to them as **_atoms_**.
- If **x** is a wff, then **~x** is a wff
- If **x** and **y** are wffs, then **<x /\ y>**, **<x \/ y>** and **<x =) y>** are wffs.
- Nothing else is a wff!

Unique readability:
A wff is of _precisely one_ of the forms given above, so we can tell exactly how it was built up from variables. This is called _parsing_ the wff, and we can draw the result as a _parse **tree_**.

For example, the wff **\sigma = "<<P /\ ~Q> =) ~<<P \/ R> /\ P'>>**" has the following parse tree:

```
                      \sigma
                       / \
                     /     \
                   /   =)    \
             <P /\ ~Q>      ~<<P \/ R> /\ P'>>
                /  \                 |
              /      \               | ~
            /   /\     \             |
          P             ~Q      <<P \/ R> /\ P'>>
                         |              / \
                         | ~          /     \
                         |          /   /\    \
                         Q      <P \/ R>       P'
                                   / \
                                 /     \
                               /   \/    \
                             P             R
```

Digression:
Contrast with natural languages, where parses are often not unique - sentences are often syntactically **_ambiguous_**.
e.g. "pretty little girls' school" has many parses (a school for girls which is quite little? **A** school owned by girls who are small and pretty? etc)

Interpretations

Suppose we have an interpretation of the propositional variables (e.g. **P -->** "Socrates is a vampire" etc). We extend the interpretation to determine truth of arbitrary wffs by requiring, for **x** and **y** wffs:

- **~x** is true iff **x** is false;
- **<x /\ y>** is true iff **x** and **y** are both true;
- **<x \/ y>** is true iff at least one of **x** and **y** are true;
- **<x =) y>** is ***false*** iff **x** is true and **y** is false.

Since every wff has a unique parse, these rules decide the truth of every wff.

Example:
According to an interpretation in which **P** and **Q** are true but **Q** and **P'** is false, determine from the parse tree whether **\sigma** is true.

So
~ --> "not"
/\ --> "and"
\/ --> "or"
=) --> "implies", "if [...] then [...]"

Regarding "or":
In English, "or" is sometimes _inclusive_
("Don't touch anything which is hot or which has sharp points!"
applies to things which are hot _and_ have sharp points)
and sometimes _exclusive_
(e.g. "a person is either male or female"
makes the (contentious!) claim that no-one can be both or neither)
("either" is mostly needed to clearly signal an exclusive or in english);
<x \/ y> --> "**x** or **y**" in the ***inclusive*** sense.

Regarding "if":
It seems we are declaring that "if **P** then **Q**" is false iff **P** is true and **Q** is false.

e.g. "If 4 is prime then there is a god" is true!

Consider:
"For every natural number **n**, if **n** is prime then **n=2** or **n** is odd." (*****)

This is true, precisely because:
for those **n** for which "**n** is prime" is true, "**n=2** or **n** is odd" is true.

For **n** for which **n** is ***not*** prime, "**n=2** or **n** is odd" is sometimes true and sometimes false.

So in other words, (*****) is true precisely because
for all **n**, **<"n** is prime" **=) <"n=2 \/ "n** is odd"**>>** is true.

Digression:
What about natural language conditionals?
"If I had a million dollars, then I would be guilty of theft."

We can analyse this as
"For all imaginable situations **s**: if I have a million dollars in **s**, then I am guilty of theft in **s**"

So is "if 4 were prime, then there would be a god" true? Not if it's imaginable that 4 is prime and there is no god!

Tautologies, contradictions and satisfiability

Definition:
A _truth assignment_ is an assignment of a _truth **value_**, True or False, to each propositional variable.

As above, a truth assignment determines truth values for all wffs.

// Truth assignments are the austere cousins of interpretations - we
// explicitly don't care about giving any "meaning" to the variables, we just
// give them truth values.

Definition:

A wff is a _tautology_ if it is True for every truth assignment.

A wff is a _contradiction_ if it is False for every truth assignment.

A wff is _satisfiable_ if it is not a contradiction, i.e. if it is True for some truth assignment.

Examples:
<P \/ ~P> is a tautology
<P /\ ~P> is a contradiction
<P =) ~P> is satisfiable, but not a tautology

Remark:
x is a contradiction iff **~x** is a tautology.
x is satisfiable iff **~x** is not a tautology.

Remark:
There is a decision procedure for being a tautology:
Given a wff **\sigma**, only finitely many propositional variables occur in **\sigma.**
For each possible assignment of True and False to those propositional variables, follow the parse tree of **\sigma** to determine whether **\sigma** is assigned True or False.
\sigma is a tautology iff it is True for all such truth assignments.

Similarly, we can decide being a contradiction and being satisfiable.

Note that if **n** different propositional variables occur in **\sigma**, we must check **2^n** assignments.

Truth tables

// Truth tables give a neat way to write down the above algorithm.

Truth table for the basic logical operators:

P	**Q**	**<P /\ Q>**	**<P \/ Q>**	**<P =) Q>**	**~P**
T	**T**	**T**	**T**	**T**	**F**
T	**F**	**F**	**T**	**F**	**F**
F	**T**	**F**	**T**	**T**	**T**
F	**F**	**F**	**F**	**T**	**T**

Truth table for **\sigma := <<~P =) <Q /\ R>> =) <<~R \/ ~Q> =) P>>**

P	**Q**	**R**	**~P**	**~Q**	**~R**	**<Q/\R>**	**<~P=)<Q/\R>>**	**<~R\/~Q>**	**<<~R\/~Q>=)P>**	**\sigma**
T	**T**	**T**	**F**	**F**	**F**	**T**	**T**	**F**	**T**	**T**
T	**T**	**F**	**F**	**F**	**T**	**F**	**T**	**T**	**T**	**T**
T	**F**	**T**	**F**	**T**	**F**	**F**	**T**	**T**	**T**	**T**
T	**F**	**F**	**F**	**T**	**T**	**F**	**T**	**T**	**T**	**T**
F	**T**	**T**	**T**	**F**	**F**	**F**	**F**	**T**	**F**	**T**
F	**T**	**F**	**T**	**F**	**T**	**F**	**F**	**T**	**F**	**T**
F	**F**	**T**	**T**	**T**	**F**	**F**	**F**	**T**	**F**	**T**
F	**F**	**F**	**T**	**T**	**T**	**F**	**F**	**T**	**F**	**T**

So **\sigma** is a tautology.

Example Zen interpretation (after Hofstadter):
P --> "You are close to the way"
Q --> "This mind is Buddha"
R --> "The flax weighs three pounds"
\sigma --> "If your not being close to the way implies that this mind is Buddha and this flax weighs three pounds, then you are close to the way if this mind is not Buddha or this flax does not weigh three pounds".
\sigma has truth-nature.

Notation:
We write **|=\sigma** to mean that **\sigma** is a tautology.

Remark:
Tautologies of the form **<\tau =) \theta>** express _valid **reasoning_**:
whatever propositions the variables stand for, if **\tau** is true then **\theta**
is true.

Exercise:
The decision procedure for tautologicalness of a wff **\sigma** described
above requires us to check each of **2^n** truth assignments, where **n** is the
number of variables appearing in **\sigma.**

Find a more efficient algorithm - one which, for some **c** and **k**, takes at
most **cn^k** cpu cycles to run. Alternatively, prove that no such algorithm
exists.

Note that you've determined whether **P=NP**, solving the most important
problem in computer science. Claim plaudits, prizes, fame, and 7 RPs.

Example:
Using truth tables to solve a Smullyan-style knight-knave puzzle.

You are lost in a maze on Smullyan Island. Each inhabitant of this strange
island is either a _knight_ or a **_knave_.** Everything a knight says is
true, while everything a knave says is false.

Walking along a corridor while trying to find the way out, you come across
an inhabitant of the island. You ask him for directions, and he says "If I
am a knight, then the exit lies behind me".

Should you continue past him?

Solution:
Write **P** for the proposition "The inhabitant is a knight".
Write **Q** for the proposition "The exit is past the knight".
So the inhabitant is claiming
\sigma := <P =) Q> .
So **\sigma** is true iff the inhabitant is a knight; i.e. we know that
<<P =) \sigma> /\ <\sigma =) P>> is true.
Now write a truth table, and see what this being true tells us about
Q's truth value.

A formal system for propositional logic

We develop a formal system, PROP, to capture tautologies:
\sigma will be a theorem of PROP iff **|=\sigma.**

[We follow Hofstatder, Ch. VII. It's a Fitchish natural deduction system]

Alphabet:
The alphabet of propositional logic, with two new symbols '[' and ']'.
Axioms:
None!
Production Rules:
Joining:
(**x**, **y**) **|-> <x /\ y>**
Separation:
<x /\ y> |-> x
<x /\ y> |-> y
Double-Tilde:
~~x |-> x
x |-> ~~x
Detachment:
(**x**, **<x =) y>**) **|-> y**
Contrapositive:
<x =) y> |-> <~y =) ~x>
<~x =) ~y> |-> <y =) x>
De Morgan:
<~x /\ ~y> |-> ~<x \/ y>
~<x \/ y> |-> <~x /\ ~y>
Switcheroo:
<x \/ y> |-> <~x =) y>
<~x =) y> |-> <x \/ y>

// No axioms so no theorems!
// That's because we're missing the informal rule!

Fantasy rule

At any point during a derivation, we may "push into a fantasy":
we write "[" on a line, and then ***any*** wff **x** on the next line.
We then proceed as if this is an entirely new derivation. Say we derive **y**.
We may then "pop out of the fantasy":
we write "]" on the line after **y**, and then "**<x =) y>**" on the line after that, and proceed as if the fantasy never happened (no lines from a popped fantasy may be used in production rules).

Example:

```
    [
        P        (pushing in to a fantasy)
        ~~P      (double-tilde)
    ]
    <P =) ~~P>  (fantasy rule)
```

So the fantasy rule implements the reasoning
"if from **x** we can prove **y**, then **<x =) y>** must be true".

Note we may push into a new fantasy within a fantasy, and we must pop out of the inner fantasy before popping out of the outer fantasy (indentation helps to keep track!).

Example:

```
    [
        <<P =) P> =) Q>
        [
            P
        ]
        <P =) P>            (fantasy)
        Q                   (detachment)
    ]
    <<<P =) P> =) Q> =) Q>  (fantasy)
```

Carry-over rule: inside a fantasy, we may write any line which appeared in the "reality one level up".

Example:

```
    [
        P
        [
            Q
            P               (carry-over)
            <Q /\ P>        (joining)
        ]
        <Q =) <Q /\ P>>     (fantasy)
    ]
    <P =) <Q =) <Q /\ P>>>  (fantasy)
```

Whee!

Remark:
Please note that by introducing this rule, we've broken the feature of our previous systems that every line of a derivation is a theorem. With the fantasy rule, ***any*** wff can appear as a line! The theorems are the wffs on lines which aren't part of any fantasy (i.e. the unindented lines, if we indent as above).

Notation:
We write "**|- \sigma**" to mean that **\sigma** is a PROP-theorem.

Waiter, waiter, there's an informal rule in my formal system!

Don't worry!

Fact: We can find a Post formal system, in the strict sense we've been using, which has the same theorems as the system described above.

How to do that (omitted in class):

Actually, there are two sensible ways to do this. The traditional approach would be to scrap the natural deduction scheme described above, and instead use a Hilbert-style deduction system. In these, the only rule of inference is detachment ("modus ponens"), and we have some well-chosen axiom schemes. You can look this up if you're interested.

But we don't need to do that. We can implement the fantasy rule directly in syntax. Here's a way to do that; the basic idea is just to keep track of the premises of the fantasies we're inside:

```
Alphabet: as above, but add new symbols |- ? W F :
Axioms: |-, WFF:P, WFF:Q, WFF:R
Production rules:
    (x|-y, WFF:z)  |->  x?z|-z      (pushing into a fantasy)
    (x|-y, WFF:z)  |->  x?z|-y      (carry-over)
    (x?y|-z, WFF:y) |->  x|-<y=)z>  (popping out of a fantasy)

    x|-<y/\z> |-> x|-y
    x|-<y/\z> |-> x|-z
    (x|-y, x|-z) |-> x|-<y/\z>
        and so on for the other rules in the original system

    WFF:Px  |->  WFF:P'x
    WFF:Qx  |->  WFF:Q'x
    WFF:Rx  |->  WFF:R'x            (variables are well-formed)

    WFF:x   |->  WFF:~x
    (WFF:x, WFF:y) |-> WFF:<x/\y>
    (WFF:x, WFF:y) |-> WFF:<x\/y>
    (WFF:x, WFF:y) |-> WFF:<x=)y>   (formation rules for wffs)

    |-x  |->  x                     (deriving wffs)
```

The last example of the previous section, derived in this system:

```
    |-
    WFF:P
    ?P|-P
    WFF:Q
    ?P?Q|-Q
    ?P?Q|-P
    ?P?Q|-<Q/\P>
    ?P|-<Q=)<Q/\P>>
    |-<P=)<Q=)<Q/\P>>>
    <P=)<Q=)<Q/\P>>>
```

Examples

Give derivations of the following tautologies.

```
    <<P =) Q> =) <~Q =) ~P>>
        (contraposition)
    <P \/ ~P>
        ("excluded middle")
    <<P /\ ~P> =) Q>
        (you can prove anything from a contradiction!)
    ~<P /\ ~P>
        Hint: first prove <<P /\ ~P> =) ~<P \/ ~P>>
    <<P =) <Q /\ ~Q>> =) ~P>
        (proof by contradiction)
    <<<P \/ Q> /\ <<P =) R> /\ <Q =) R>>> =) R>
        (cases)
    <~<P /\ Q> =) <~P \/ ~Q>>
        (more De Morgan)
    <<<P =) <P =) Q>> /\ <<P =) Q> =) P>> =) <P /\ Q>>
        (cf knight-knave example above)
```

Substitution

Definition:

Let **\sigma** be a wff, and let **p_1**, ..., **p_n** be propositional variables appearing in **\sigma**. Let **\phi_1**, ..., **\phi_n** be wffs. Then if we replace each occurence of **p_i** in **\sigma** with **\phi_i**, we get a new wff. Such a wff is called a _substitution instance_ of **\sigma**.

Lemma: Suppose **\tau** is a substitution instance of **\sigma**. Then
(a) if **|= \sigma** then **|= \tau**
(**b**) if **|- \sigma** then **|- \tau**

Proof:
(a) Exercise
(**b**) Make the substitution throughout a derivation of **\sigma**; the result is also a derivation.

Example:
We saw that
|- <P \/ ~P>.
So by substituting **<P =) Q>** for **P**, it follows that
|- <<P =) Q> \/ ~<P =) Q>>.
Similarly for **|=**.

Soundness

Theorem [Soundness]:
For any wff **\tau**, if **|- \tau** then **|= \tau**.

Lemma:
The production rules correspond to tautologies:
|= <<P/\Q> =) P> **(separation)**
|= <<P /\ <P =) Q>> =) Q> **(detachment)**
|= <<P /\ Q> =) <P /\ Q>> **(joining)**
|= <~~P =) P> **(double-tilde)**
etc

Proof:
Check truth tables. Exercise.

We would like now to prove the theorem by induction on the length of a derivation - but the induction hypothesis tells us nothing about lines which occur within fantasies...

Definition:
A set of wffs **\Sigma _necessitates_** a wff **\tau**, written
\Sigma |= \tau,
if **\tau** is true for all truth assignments for which every **\sigma** in **\Sigma** is true.

|= \tau abbreviates **\emptyset |= \tau.**

[Hoping to avoid giving this, as it just seems obfuscatory
Notation:
(just to clarify)
Recall that a truth assignment is a map
f : {propositional variables} -> {T,F} .
Write **f*** for the unique extension
f* : {wffs} -> {T,F}
such that for all wffs **\sigma,\tau**:
f*(~\sigma)=T iff **f*(\sigma)=F**,
f*(<\sigma /\ \tau>)=T iff **f*(\sigma)=T=f*(\tau)**,
f*(<\sigma \/ \tau>)=F iff **f*(\sigma)=F=f*(\tau)**,
and **f*(<\sigma =) \tau>)=F** iff **f*(\sigma)=T** and **f*(\tau)=F.**

(so "**\sigma** is true for **f**" means **f*(\sigma)=T**).

Then we can write the definition of **\Sigma |= \tau** more formally as:
for all **f**, if **f*(\sigma)=T** for all **\sigma \in \Sigma** then **f*(\tau)=T.**
]

Definition:
The _premise_ of a fantasy is its first line.

The _premises of a line_ of a PROP-derivation are the premises of the fantasies the line appears within.

Claim:
Let **\tau** be a wff occuring as a line of a PROP-derivation.
Let **\Sigma** be the set of premises of the line.
Then **\Sigma |= \tau**.

Proof:

Assume the claim holds for the first **k** lines of any derivation, we show it holds for the first **k+1**. So suppose the (**k+1**)th line of a derivation is a wff **\tau** with premises **\Sigma**.

If **\tau** is the premise of a fantasy, then **\tau \in \Sigma**, so clearly **\Sigma |= \tau**.

If **\tau** is a carry-over, then **\tau** appears as a previous line with premises **\Sigma'** a subset of **\Sigma**; by the inductive hypothesis, **\Sigma' |= \tau**, so also **\Sigma |= \tau**.

If **\tau** is the result of the fantasy rule, then **\tau = <\phi =) \psi>**, and **\psi** appears on a previous line with premises **\Sigma \union {\phi}**, so by the inductive hypothesis

\Sigma \union {\phi} |= \psi.

Now for any truth assignment for which all **\sigma\in\Sigma** are true:

if **\phi** is true then **\psi** is true since **\Sigma \union {\phi} |= \psi**;
hence **\tau = <\phi =) \psi>** is true.

So **\Sigma |= \tau**.

[Phrasing that argument with the fs:
Now let **f** be a truth assignment, and suppose **f*(\sigma)=T** for all **\sigma \in \Sigma**. Suppose **f*(<\phi =) \psi>)=F**. Then **f*(\phi)=T** and **f*(\psi)=F**, contradicting **\Sigma \union {\phi} |= \psi**. So **f*(<\phi =) \psi>)=T**. So **\Sigma |= \tau**.
]

Else, **\tau** is the result of a production rule. Say it has two inputs, **\phi** and **\psi**. Each appears as a previous line in the derivation with the same premises **\Sigma**, so by the inductive hypothesis,

\Sigma |= \phi and **\Sigma |= \psi**.

By the Lemma,

|= <<\phi /\ \psi> =) \tau>.

It follows easily that **\Sigma |= \tau**.

(if the production rule has only one input, the argument is similar)

Completeness

Definition:

For a set **\Sigma**, write

\Sigma |- \tau ("\Sigma proves \tau")

to mean **\tau** is a theorem of the system **PROP+\Sigma** we get by adding **\Sigma** as axioms to PROP.

Lemma ["strong soundness"]:

If **\Sigma |- \tau** then **\Sigma |= \tau**

Proof:

Suppose **\Sigma |- \tau**. So there is a derivation of **\tau** using **\Sigma** as axioms. The derivation can use only finitely many of the axioms, say **\sigma_1**, ..., **\sigma_n**. Let **\phi** be the conjunction

\phi := <\sigma_1 /\ <\sigma_2 /\ ... /\ \sigma_n>...>>

Then by separation and the fantasy rule,

|- <\phi =) \tau>.

By soundness,

|= <\phi =) \tau>.

It follows easily that

\Sigma |= \tau.

Lemma 1:

For each of ~, /\, \/, **=)**, the tautologies corresponding to its truth table are theorems; i.e.

<P/\Q>:

|- <<P/\Q> =) <P/\Q>>
|- <<~P/\Q> =) ~<P/\Q>>
|- <<P/\~Q> =) ~<P/\Q>>

```
            |- <<~P/\~Q> =) ~<P/\Q>>
        ~P:
            |- <P =) ~~P>
            |- <~P =) ~P>
    and similarly for \/ and =).
Proof:
    All fairly straightforward. See exercises.

Lemma 2:
    |- <<<P =) Q> /\ <~P =) Q>> =) Q>
Proof:
    Here's a PROP-derivation:
    [
        <<P =) Q> /\ <~P =) Q>>
        <P =) Q>
        <~P =) Q>
        [
            ~Q
            <P =) Q>
            <~Q =) ~P>
            ~P
            <~P =) Q>
            Q
        ]
        <~Q =) Q>
        <Q \/ Q>
        [
            ~Q
            <~Q /\ ~Q>
            ~<Q \/ Q>
        ]
        <~Q =) ~<Q \/ Q>>
        <<Q \/ Q> =) Q>
        Q
    ]
    <<<P =) Q> /\ <~P =) Q>> =) Q>


Theorem [completeness of PROP]:
    For any wff \tau, if |= \tau then |- \tau.

Proof:
    Notation:
        If PV = {p_1, ..., p_n} is a set of propositional variables and
        f : PV -> {T,F}, write
            \Sigma^f := { +/- p_i | 1 <= i <= n }
        where +/- p_i = p_i if f(p_i)=T, and +/- p_i = ~p_i if f(p_i)=F.
    Claim:
        If \sigma is a wff and all propositional variables occuring in \sigma
        are in PV, then for any f,
            \Sigma^f |- \sigma  or  \Sigma^f |- ~\sigma  (*)
    Proof:
        By induction on depth of \sigma's parse tree.

        If \sigma is a propositional variable, (*) is clear.

        Else, clear by Lemma 1 and the inductive hypothesis.

    Now let PV = {p_1, ..., p_n} be the set of propositional variables
    occuring in \tau.

    So by the claim, "strong soundness" and the fact that \tau is a tautology,
    for any f : PV -> {T,F},
        \Sigma^f |- \tau .

    For k<=n, let PV_k := {p_i | i > k} = {p_{k+1}, ..., p_n}, so PV_0 = PV
    and PV_n = \emptyset. We show inductively that for any k<=n:
        (*)_k: for any f : PV_k -> {T,F},
                \Sigma^f |- \tau .

    We've seen (*)_0. Suppose (*)_{r-1}, 0<r<=n; we show (*)_r.
    Let f : PV_r -> {T,F}. Then we know
```

{p_r} \union \Sigma^f |- \tau
and
{~p_r} \union \Sigma^f |- \tau.

So by the fantasy rule,
\Sigma^f |- <p_r =) \tau>
and
\Sigma^f |- <~p_r =) \tau>,
so
\Sigma^f |- <<p_r =) \tau> /\ <~p_r =) \tau>> .
But then, by Lemma 2,
\Sigma^f |- \tau .

So (***)_n** holds, i.e.
|- \tau .
QED

So we have proven

Theorem [soundness and completeness of PROP]:
For any wff **\tau**, **|= \tau** iff **|- \tau.**

Digression:
The strong version of completeness
\Sigma |= \tau implies **\Sigma |- \tau**
is true. For ***finite* \Sigma**, this follows by a similar argument to that for strong soundness.

To show it for infinite **\Sigma**, the only difficulty is to see that if **\Sigma |= \tau**, then actually there's some _finite_ **\Sigma' (= \Sigma** such that **\Sigma' |= \tau.** That takes a little thought; it's equivalent to compactness of Cantor space **2^\omega.**

III: Typographical Number Theory

In this section, we define Hofstadter's TNT [with some subtle modifications].

[TNT is PA]

Language of TNT

Alphabet:
0 **S + * ( ) =**
~ /\ \/ =)
a **b c d e x y z** '
A E

(We no longer have propositional variables.)

Variables:
a, **b**, **c**, **d**, **e**, **x**, **y**, **z** are variables
If **v** is a variable, so is **v'.**

Terms:
any variable is a term;
0 is a term;
if **t** and **s** are terms, then so are
St, (**t+s**), (**t*s**);
nothing else is a term.

wffs:
if **t** and **s** are terms, then **t=s** is a wff (an _atomic **formula_**);
if **\phi** and **\psi** are wffs, so are
~\phi, **<\phi /\ \psi>**, **<\phi \/ \psi>**, **<\phi =) \psi>**;
if **\phi** is a wff and **v** is a variable, then
Av:\phi and **Ev:\phi**
are wffs;
nothing else is a wff.

Remark:

```
    We have unique parse trees.

Bound and free variables:
    An _occurrence of a variable_ v in a wff is a location in the wff where
    the variable appears, where appearances in substrings of the form "Av:" or
    "Ev:" do not count.
        (e.g. there are *no* occurrences of y in "Ay:y'=y'", but two of y')

    An occurrence of a variable v in a wff is _bound_ if it occurs within a
    substring of the form "Av:\phi" or "Ev:\phi" (\phi a wff). Else, the
    occurrence is _free_.

    The _free variables_ of a wff are those variables which occur free in the
    wff.

The standard interpretation:
    Variables stand for natural numbers.

    Call a choice of natural number for each variable, i.e. a map
    f : [variables] -> N, a "variable assignment".

    Given a variable assignment f, we evaluate terms as natural numbers:
        eval_f(v) = f(v)
        eval_f(0) = 0
        eval_f(St) = eval_f(t) + 1          ("Successor")
        eval_f((t+s)) = eval_f(t) + eval_f(s)
        eval_f((t*s)) = eval_f(t) * eval_f(s)

    Now we determine truth of a wff wrt a variable assignment f:

    * An atomic formula "t=s" is true wrt f iff eval_f(t) = eval_f(s).
    * "<\phi /\ \psi>" is true wrt f iff \phi and \psi are both true wrt f.
    * Similarly for ~, \/, =), as in propositional logic.
    * Av:\phi is true wrt f iff \phi is true for any
        variable assignment f' which is the same as f except possibly on v.
            (i.e. f'(w)=f(w) if w!=v)
        [ with notation:
            Av:\phi ^ f = T  iff  \forall f'. \forall w\in Variables.
                                  ((w!=v -> f'(w) = f(w)) -> \phi ^ f' = T)
        ]
    * Ev:\phi is true wrt f iff \phi is true for some
        variable assignment f' which is the same as f except possibly on v.


    Clearly, whether a wff \phi is true wrt a variable assignment f depends
    only on the values of f at the free variables of \phi.

    A wff with no free variables is a _sentence_, and is just true or false.

    A wff with 1 free variable expresses a _property_ of a natural number
        (e.g. primeness, oddness...).

    A wff with n free variables expresses an _n-ary relation_ (aka _predicate_)
        (1-ary == unary, 2-ary == binary etc)
        (e.g. "x is less than y"; "x is between y and z").

Examples:
    Ax: Ey: Sx=y
        (first think what Ey: Sx=y says about x
            (first think what Sx=y says about x,y))
    Ex: Ay: Sx=y
        (remark: cf ambiguity of english
            "every number is the predecessor of some number")
    Ax: Ey: x=Sy
    Ax: <Ey: x=Sy \/ x=0>

    Ey: (y+y)=x
    Ey: S(y+y)=x
    <Ey: (y+y)=x \/ S(y+y)=x>
    Ax: <Ey: (y+y)=x \/ S(y+y)=x>

    Ax: <Ey: (y+y)=x =) ~(x*x)=x>
    Ax: Ey: <(y+y)=x =) ~(x*x)=x>
```

Ez: x=(y+z)
Ax: Ez: (x*x)=(x+z)
Ez: x=(y+Sz)
Ax: Ez: (x*x)=(x+Sz)

<~x=0 /\ <~x=S0 /\ Ay:Az:<(y*z)=x -> <y=x \/ z=x>>>>
<Ey:x=SSy /\ ~Ey:Ez:x=(SSy*SSz)>

Euclid:
Ax: Ey: Ez: <y=(x+Sz) /\ ~Ey:Ez:x=(SSy*SSz)>

Fermat (n=3):
~Ex:Ey:Ez:(x*(x*x))+(y*(y*y))=(z*(z*z))

Goldbach:
Ax:Ey:Ez:<<~Ey':Ez':y=(SSy'*SSz') /\ ~Ey':Ez':z=(SSy'*SSz')> /\ (y+z)=(x+x)>

Non-standard interpretations:
A _structure in the language of arithmetic_ consists of
a set N';
an element 0' \in N';
a unary function S' : N' -> N';
binary functions +', *' : N'^2 -> N'.
We denote the structure by <N';0',S',+',*'>, or just N'.

The _standard_ arithmetic structure is <N;0,S,+,*>, i.e. the set of natural numbers N, with usual 0, successor, addition and multiplication.

An assignment of variables for N' assigns an element _of N'_ to each variable; terms evaluate to elements of N' using 0', S', +', and *'; wffs evaluate, wrt a variable assignment, to True/False as above (so "Ax:" now means "for all x in N'").

For a sentence \sigma, we write
N' |= \sigma
to mean that \sigma is true when interpreted in N'.

Example: the integers Z with usual zero, successor, addition, and multiplication is a structure in the language of arithmetic.
"Ex:Sx=0" is true in Z but not in N.
(Z |= Ex:Sx=0, but N |/= Ex:Sx=0)

PRED

Axioms:
Axiom 0: Ax:x=x
Rules:
Rules of the propositional calculus; premises of fantasies may now be arbitrary *TNT*-wffs.

Generalisation: \phi |-> Av:\phi
where v is a variable.
RESTRICTION: v must not occur free in any premise of \phi.

Specification: Av:\phi |-> \phi[t/v]
where \phi[t/v] is the result of replacing each free occurrence of the variable v in \phi with the term t.
RESTRICTION: any new occurrences of variables resulting from the substitution must be free.

Interchange: XAv:~Y <-|-> X~Ev:Y
X~Av:Y <-|-> XEv:~Y
where v is a variable;
i.e. whenever "Av:~" occurs within a wff, it may be rewritten as "~Ev:", and vice-versa.

Existence: \phi[t/v] |-> Ev:\phi
where \phi is a wff, v is a variable, t is a term, and \phi[t/v] is the result of replacing each free occurrence of v in \phi with t.

```
        RESTRICTION: the substitution must meet the restriction imposed in the
            specification rule: any occurrences of variables created in
            passing from \phi to \phi[t/v] must be free.


    Symmetry: t = s |-> s = t
    Transitivity: (t = s, s = r) |-> t = r
    Congruence:
        t=s |-> St = Ss
        (t_1=s_1, t_2=s_2) |-> (t_1+t_2) = (s_1+s_2)
        (t_1=s_1, t_2=s_2) |-> (t_1*t_2) = (s_1*s_2)

        where t,s,r,t_i,s_i are terms.

Notation:
    |- \phi  means \phi is a PRED-theorem
        (note \phi can have free variables)

Examples:
    |- <Ax:Ay:x=y =) Ay:Sy=y>:
    [
        Ax:Ay:x=y
        Ay:Sy=y
    ]

    |- <Ax:Ay:x=y =) Ax:Sx=x>:
    [
        Ax:Ay:x=y
        Ay:Sy=y
        Sx=x
        Ax:Sx=x
    ]

    |- Ax:<Ey:<y=Sx /\ x=Sy> =) x=SSx>:
    [
        ~x=SSx
        [
            <y=Sx /\ x=Sy>
            x=Sy
            y=Sx
            Sy=SSx
            x=SSx
        ]
        <<y=Sx /\ x=Sy> =) x=SSx>
        <~x=SSx =) ~<y=Sx /\ x=Sy>>
        ~<y=Sx /\ x=Sy>
        Ay:~<y=Sx /\ x=Sy>
        ~Ey:<y=Sx /\ x=Sy>
    ]
    <~x=SSx =) ~Ey:<y=Sx /\ x=Sy>>
    <Ey:<y=Sx /\ x=Sy> =) x=SSx>
    Ax:<Ey:<y=Sx /\ x=Sy> =) x=SSx>


Non-examples (demonstrating necessity of the restrictions):
    [
        x=0
        Ax:x=0
    ]
    <x=0 =) Ax:x=0>
    Ax:<x=0 =) Ax:x=0>
    Uhoh!


    [
        Ax:Ey:~x=y
        Ey:~y=y
    ]
    <Ax:Ey:~x=y =) Ey:~y=y>
    Uhoh!


    Ax:x=x
```

Ey:Ax:y=x
Uhoh!

[
Ax:x=(x*S0)
Ex:Ax:x=(x*x)
]
Uhoh!

[
Ax:~x=Sx
~x=Sx
Ex:~x=x (existence, t:=Sx)
]
Uhoh!

Definition:
A _TNT-tautology_ is a substitution instance of a propositional tautology, obtained by replacing propositional variables with TNT-wffs.
Remark:
By completeness of PROP, any TNT-tautology is a PRED-theorem.
(Make the substitution in a PROP-derivation, yielding a PRED-derivation)

Example:
|- <Ex:~x=x =) (SS0+SS0)=SSSSS0>:
[
Ex:~x=x
~Ax:x=x (interchange)
Ax:x=x (axiom 0)
<Ax:x=x /\ ~Ax:x=x> (joining)
[...lines proving following tautology omitted...]
<<Ax:x=x /\ ~Ax:x=x> =) (SS0+SS0)=SSSSS0>
(**SS0+SS0)=SSSSS0 (detachment)**
]
<Ex:~x=x =) (SS0+SS0)=SSSSS0> (fantasy rule)

For convenience, we add all TNT-tautologies to PRED as axioms.

[omitting this for now; might put it in later if it surviving without it is too annoying:
Lemma:
Suppose **|- <<\phi =) \phi'> /\ <\phi' =) \phi>>**,
and suppose **\theta** is a formula in which **\phi** occurs as a subformula, and **\theta'** is the result of replacing that subformula with **\phi'.**
Then **|- <<\theta =) \theta'> /\ <\theta' =) \theta>>.**
Proof:
[omitted, but straightforward by induction on length of **\theta**, and using the previous remark]

For convenience, we add as a rule to PRED:
Substitution: **\theta |-> \theta'**
whenever **\theta** and **\theta'** are as in the previous lemma.
]

Remark:
The existence rule can be deduced from the specification rule and interchange:
[
~Ev:\phi
Av:~\phi
~\phi[t/v]
]
<~Ev:\phi =) ~\phi[t/v]>
<\phi[t/v] =) Ev:\phi>

Soundness and completeness

Notation:
For **\Sigma** a set of sentences and **\tau** a sentence, write
\Sigma |- \tau
if **\tau** is a theorem of the system **PRED+\Sigma** obtained by adding **\Sigma** as axioms to PRED, and
\Sigma |= \tau
if **\tau** is satisfied by every structure in the language of arithmetic which satisfies all the sentences in **\Sigma**; i.e.
\Sigma |= \tau <=> for all **N'**: **N'|=\Sigma => N'|=\tau**

Fact (Soundness):
If **\Sigma |- \tau**, then **\Sigma |= \tau**

Fact (Gödel's Completeness Theorem):
If **\Sigma |= \tau**, then **\Sigma |- \tau**

(So **\Sigma |= \tau <=> \Sigma |- \tau**)

// Allaying possible confusion concerning the term 'complete':
Definition:
A system in the language of TNT is _complete for the standard **interpretation**_, abbreviated "**N**-complete" or "complete for **N**", if every TNT-sentence which is true in the standard interpretation is a theorem.

It is _negation complete_ if for every TNT-sentence **\sigma**, at least one of **\sigma** and **~\sigma** is a theorem.

Remark:
N-complete **=>** negation complete.

PRED is not negation complete!
e.g. neither **0+0=0** nor **~0+0=0** is a theorem!

GIT1 proves negation incompleteness, hence **N**-incompleteness.

TNT'

Definition:
TNT' is the system obtained by adding the following axioms to PRED:
Axiom 1: **Ax:~Sx=0**
Axiom 2: **Ax:(x+0)=x**
Axiom 3: **Ax:Ay:(x+Sy)=S(x+y)**
Axiom 4: **Ax:(x*0)=0**
Axiom 5: **Ax:Ay:(x*Sy)=((x*y)+x**)
Axiom 6: **Ax:Ay:<Sx=Sy =) x=y>**

We will also write "TNT'" to refer to the set of these 6 axioms, so
TNT' **|- \sigma**
means that **\sigma** is a TNT'-theorem.

Remark:
Hofstadter has the rule
Drop **S**: **Sx=Sy |-> x=y**
in place of Axiom 6; this makes no real difference - the two systems prove the same theorems.

[Remark for the initiated: TNT' is basically Robinson's **Q**, although we're missing the axiom that only 0 has no predecessor (which is ok for our purposes, as this is implied by the induction axioms)]

Note **N |=** TNT', so by soundness if TNT' **|- \sigma** then **N |= \sigma**.

Example:
TNT' **|- S0+S0=SS0**:

Ax:Ay:(x+Sy)=S(x+y)
Ay:(S0+Sy)=S(S0+y)
(S0+S0)=S(S0+0)
Ax:(x+0)=x

S0+0=S0
S(S0+0)=SS0
(S0+S0)=SS0

Fact:
TNT' can prove every sentence which is true in **N** of the form **t=s**, where **t** and **s** are terms.

Example:
TNT' |- Ax:(x*(S0+S0))=((x*S0)+x):

(S0+S0)=SS0 (shown above)
Ax:x=x
x=x
(x*(S0+S0))=(x*SS0)
Ax:Ay:(x*Sy)=((x*y)+x)
Ay:(x*Sy)=((x*y)+x)
(x*SS0)=((x*S0)+x)
(x*(S0+S0))=((x*S0)+x)
Ax:(x*(S0+S0))=((x*S0)+x)

TNT' is still not **N**-complete!

In other words, there are "non-standard" structures in the language of arithmetic which satisfy axioms 1-6, but satisfy sentences **N** does not.

Example:
Let **Mat_2(N)** be the structure in the language of arithmetic consisting of 2x2 matrices with natural number entries, with matrix addition and matrix multiplication and with **S(M) := M+I**, where I is the identity matrix.

Then **Mat_2(N) |=** TNT'.

Now let **\sigma** be the sentence Ax:<x*x=0 =) x=0>.

Clearly **N |= \sigma**.

But **M |= ~\sigma**, since

```
      2
(01)     =  (00)
(00)        (00)
```

So by soundness of PRED, {Ax 1-6} **|/- \sigma**.

However, by the Fact above, whenever **t** is a numeral (i.e. one of 0, S0, SS0, ...),
<(t*t)=0 =) t=0>
is a TNT'-theorem!

Similarly, the following are *not* TNT'-theorems:
Ax:(0+x)=x
Ax:Ay:(x+y)=(y+x)
Ax:Ay:(x*y)=(y*x).

[Remark: **Mat_2(\N) |/= Q**, though]

TNT
===

What's missing?

In **N**, if **\phi[0/v]** and **\phi[S0/v]** and **\phi[SS0/v]** and so on all hold, then so does **Av:\phi**.

But e.g. if \phi := <(x*x)=0 =) x=0>, then **\phi[0/x]** and **\phi[S0/x]** and **\phi[SS0/x]** and so on are all theorems, but **Av:\phi** is not. Similarly with **\phi := (0+x)=x**.

Proposed "Rule of All":
(**\phi[0/v]**, **\phi[S0/v]**, **\phi[SS0/v]**, ...) **|-> Av:\phi**

BUT rules of formal systems have ***finitely*** many inputs, this has ***infinitely*** many. You could never use this rule as part of a finite derivation!

Consider how we prove statements of the form "for all **n**" in everyday mathematics...

Induction rule:
        (**\phi[0/v]**, **Av:<\phi =) \phi[Sv/v]>**) **|-> Av:\phi**
    where **v** is a variable and **\phi** is a wff, and **\phi[t/v]** is the result of replacing each free occurence of **v** in **\phi** with the term **t**).

// But let's use axioms rather than a rule, so we have access to soundness and
// completeness.

Definition:
    TNT is the system obtained by adding to TNT' the following infinite set of axioms:
        Induction axioms: for each wff **\phi** with one free variable **v**, the axiom
            <<\phi[0/v] /\ Av:<\phi =) \phi[Sv/v]>> =) Av:\phi> .

    (Again, we will also use "TNT" to refer to the set of axioms)

Remark:
    TNT is more commonly known as PA (**"(first-order)** Peano arithmetic")

Example: **Ax:(0+x)=x** is a TNT-theorem:

```
    1.  Ax:(x+0)=0
    2.  (0+0)=0
    3.  [
    4.      (0+x)=x
    5.      Ax:Ay:(x+Sy)=S(x+y)
    5.      Ay:(0+Sy)=S(0+y)         (spec x->0)
    6.      (0+Sx)=S(0+x)            (spec y->x)
    7.      S(0+x)=Sx
    8.      (0+Sx)=Sx
    9.  ]
    10. <(0+x)=x =) (0+Sx)=Sx>
    11. Ax:<(0+x)=x =) (0+Sx)=Sx>   (gen)
    12. Ax:(0+x)=x                 (induction: lines 2, 11)
```

Remark:
    N |= TNT, so any TNT-theorem is true in **N (i.e. TNT is sound for N).**

Question:
    Does the converse hold? i.e. is TNT complete for **N**?

    We will answer this presently!

Related question:
    If a structure **N'=<N';S',+',*'>** satisfies TNT, must every element of **N'** be one of 0', S'0', S'S'0', ...?

    Answer: no! However, there aren't any easily described examples like **Mat_2(N).** [See Tennenbaum's theorem]

Fact:
    There is a Post formal system, FormalTNT, such that a wff is a theorem of TNT iff it is a theorem of FormalTNT.

Appendices:
===========

A: deviations from Hofstadter

PRED is set up so as to satisfy Gödel's completeness theorem - this meant adding **Ax:x=x** as an axiom, and inserting the congruence rules. I also added the other form of interchange, because surviving without it is painful (though possible).

The existence rule got rewritten. Here's an equivalent version which looks more like Hofstadter's:
    Existence: **\phi |-> Ev:\phi'**
        where **\phi** is a wff, **v** is a variable, **t** is a term, and **\phi'** is the

result of replacing one or more occurrences of **t** in **\phi** with **v**.
RESTRICTION: no bound occurrences of variables may be created or destroyed in passing from **\phi** to **\phi'**, and there may be no occurrences of **v** in **\phi'** other than those introduced through replacing occurrences of **t** in **\phi** with **v**.

"Drop **S**" became Axiom 6, and the induction rule became a set of axioms, to ensure that TNT is of the form **PRED+\Sigma**.

B: FormalTNT

We can implement TNT in a Post formal system.

It's more than a little ugly! But conceptually it's straight-forward.

[Again, I'm omitting this from the lectures, but including it here for the curious.]

(note that 'x', 'y' and 'z' are now in our alphabet, so we use 'X', 'Y', 'Z', 'X1', 'Z37' and so on for variables when giving production rules.

Let's simplify things by removing **E** from our formal system, considering "Ev:" to be just an abbreviation for "**~Av:~**".

Alphabet: as above, but add new symbols **|-** ? **|**, and all the roman alphabet in lower case and in upper case, except **X Y** and **Z**.
Axioms and Production rules:

```
Var|x
Var|y
Var|z
Var|X |-> Var|X'
Var|X |-> Term|X

VarNeq|x|y
VarNeq|y|z
VarNeq|z|x
VarNeq|X|Y |-> VarNeq|Y|X
(Var|X, Var|XY') |-> VarNeq|X|XY'

Term:0
Term|X |-> Term|SX
(Term|X, Term|Y) |-> Term|(X+Y)
(Term|X, Term|Y) |-> Term|(X*Y)

(Term|X, Term|Y) |-> WFF|X=Y

WFF|X   |->  WFF|~X
(WFF|X, WFF|Y) |-> WFF|<X/\Y>
(WFF|X, WFF|Y) |-> WFF|<X\/Y>
(WFF|X, WFF|Y) |-> WFF|<X=)Y>

// NoFree|Z|Y : variable Z doesn't appear free in wff Y
// NoFreeT|Z|Y : variable Z doesn't appear in term Y
VarNeq|Z|Y |-> NoFreeT|Z|Y
NoFreeT|Z|0
NoFreeT|Z|X |-> NoFreeT|Z|SX
(NoFreeT|Z|X, NoFreeT|Z|Y) |-> NoFreeT|Z|(X+Y)
(NoFreeT|Z|X, NoFreeT|Z|Y) |-> NoFreeT|Z|(X*Y)
(NoFreeT|Z|X, NoFreeT|Z|Y) |-> NoFree|Z|X=Y
(NoFree|Z|X, NoFree|Z|Y) |-> NoFree|Z|<X/\Y>
(NoFree|Z|X, NoFree|Z|Y) |-> NoFree|Z|<X\/Y>
(NoFree|Z|X, NoFree|Z|Y) |-> NoFree|Z|<X=)Y>
NoFree|Z|X |-> NoFree|Z|~X
(Var|Y, NoFree|Z|X) |-> NoFree|Z|AY:X
WFF|X |-> NoFree|Z|AZ:X

// NoFreePrems|Z|X : variable Z doesn't appear in any of the wffs in the
// ?-separated list X
Var|Z |-> NoFreePrems|Z|
(NoFree|Z|X, NoFreePrems|Z|Y) |-> NoFreePrems|Z|Y?X

(X|-Y, NoFreePrems|Z|X) |-> X|-AZ:Y     // (generalisation)
```

```
// Sub|X|Z|Z1|Y : Y is the result of validly substituting all free
// occurrences of the variable Z in the wff X with the term Z1, where
// "valid" means that no variable occurring in Z1 gets put in somewhere it
// gets bound.
// SubT|X|Z|Z1|Y : same, but X is a term.
(Var|Z, Term|Z1) |-> SubT|Z|Z|Z1|Z1
(Var|Z, Term|Z1) |-> SubT|0|Z|Z1|0
SubT|X|Z|Z1|Y |-> SubT|SX|Z|Z1|SY
(SubT|X|Z|Z1|Y, SubT|X1|Z|Z1|Y1) |-> SubT|(X+X1)|Z|Z1|(Y+Y1)
(SubT|X|Z|Z1|Y, SubT|X1|Z|Z1|Y1) |-> SubT|(X*X1)|Z|Z1|(Y*Y1)
(SubT|X|Z|Z1|Y, SubT|X1|Z|Z1|Y1) |-> Sub|X=X1|Z|Z1|Y=Y1
(Sub|X|Z|Z1|Y, Sub|X1|Z|Z1|Y1) |-> Sub|<X/\X1>|Z|Z1|<Y/\Y1>
(Sub|X|Z|Z1|Y, Sub|X1|Z|Z1|Y1) |-> Sub|<X\/X1>|Z|Z1|<Y\/Y1>
(Sub|X|Z|Z1|Y, Sub|X1|Z|Z1|Y1) |-> Sub|<X=)X1>|Z|Z1|<Y=)Y1>
Sub|X|Z|Z1|Y |-> Sub|~X|Z|Z1|~Y
(VarNeq|Z|Z2, NoFreeT|Z2|Z1, Sub|X|Z|Z1|Y) |-> Sub|AZ2:X|Z|Z1|AZ2:Y
(VarNeq|Z|Z2, Sub|X|Z|Z1|X) |-> Sub|AZ2:X|Z|Z1|AZ2:X
(Var:Z, WFF:X) |-> Sub|AZ:X|Z|Z1|AZ:X

(X|-AZ:Y, Term|Z1, Sub|Y|Z|Z1|Y1) |-> X|->Y1    // (specification)

(Term|X, Term|Y, Z|-X=Y) |-> Z|-Y=X (symmetry)
// other equality rules similar and omitted

// (the following is the same as for PROP)

|-
(X|-Y, WFF:Z)  |->  X?Z|-Z      // (pushing into a fantasy)
(X|-Y, WFF:Z)  |->  X?Z|-Y      // (carry-over)
(X?Y|-Z, WFF:Y) |->  X|-<Y=)Z>  // (popping out of a fantasy)

X|-<Y/\Z> |-> X|-Y
X|-<Y/\Z> |-> X|-Z
(X|-Y, X|-Z) |-> X|-<Y/\Z>
    // and so on for the other deduction rules of PROP

|-Ax:x=x
    // and similarly for axioms 1-6

// Induction axiom scheme:
(Sub|X|Z|0|Y, Sub|X|Z|SZ|Y1) |-> |-<<Y/\AZ:<X=)Y1>>=)AZ:X>

|-X  |->  X                     // (deriving wffs)
```

IV: Semantic Incompleteness

In this section, we will prove the following weak "semantic" version of Gödel's First Incompleteness Theorem:

Theorem [Semantic G1T, Post formal system version]
Arithmetical truth is not captured by any Post formal system, i.e. there is no Post formal system **S** such that for all TNT-sentences **\sigma**, **\sigma** is true in **N** iff **\sigma** is a theorem of **S**.

In particular, TNT is **N**-incomplete.

Idea of proof:
Let **S** be a Post formal system which is sound for **N**, i.e. if **\sigma** is an **S**-theorem then **\sigma** is true in **N**.

We find a sentence **G** which "says":
"**G** is not derivable in **S**"
i.e. **G** is true in **N** iff there is no derivation of **G** in **S**.

If **G** is false in **N**, then **G** is an **S**-theorem, hence is true in **N** - contradiction.

So **G** is true in **N**. So **G** is not an **S**-theorem.

For the rest of this section, we work with the standard interpretation of TNT-wffs - "true" means "true in **N**", and variables take values in **N**.

Notation:

[defining some abbreviations to make our formal language actually usable, making our lives much easier as we explore what can be expressed in the language of arithmetic]

If **n** is a natural number, then **\overline{n}** is an abbreviation for the TNT term SS...S0 with **n** S's.

[in these ascii notes, I'll miss out the overline... don't get confused!]

e.g. **"Ax:(2*x) = ((1*x)+(1*x)**)" is just an abbreviation for **"Ax:(SS0*x) = ((S0*x)+(S0*x)**)".

If we denote a wff by **\phi(x,y)**, we are indicating that the free variables of the wff are precisely **x** and **y**.

We then write **\phi(t,s)**, where **t** and **s** are terms, as an abbreviation for the wff obtained by substituting **t** for each free occurrence of **x** and **s** for each free occurrence of **y**, and adding primes to quantified variables in **\phi** as necessary to avoid conflicts.

e.g. let **Lteq(x,y)** be the wff
    Ez:(x+z)=y.

Then **Lteq(S0,SSS0)** is the wff
    Ez:(S0+z)=SSS0
and **Lteq(y,x)** is the wff
    Ez:(y+z)=x.
and **Lteq(z,y)** is the wff
    Ez':(y+z')=z.
and **Lteq(z,(S0+z')**) is the wff
    Ez'':((S0+z')+z'')=z.

We will write
    t <= s
as an abbreviation for the wff
    Lteq(t,s)
i.e. for the wff
    Ez:(t+z)=s.

Similarly, let **LessThan(x,y)** be the wff Ez: (**x+z)=y**, and let "**t<s**" abbreviate **LessThan(t,s)**.

Gödel numbering: coding Post formal systems in arithmetic

Recall that a Post formal system consists of:
- An alphabet consisting of finitely many symbols;
- a finite set of axioms;
- a finite set of "pattern matching" production rules.

We will _code_ strings and derivations as natural numbers, and show that syntactic operations are expressible by wffs. In particular,

* given a rule **R** with 1 input, we will find a formula **Produces_R(x,y)** such that if **x** is the code of a string **X**, then **Produces_R(x,y)** is true precisely when **y** is the code of a string which can be produced by **R** with input **X**.

* Similarly for rules with many inputs.

* Using this, we will find a formula **ProofPair(x,y)** true precisely when **x** codes for a valid **S**-derivation of which **y** is a line.

* Hence the formula
    Theorem(y) := Ex:Proves(x,y)
  will be true precisely when **y** codes for an **S**-theorem.

Coding strings:

Example - MIU system:

Symbols coded as numbers:

```
I ==> 1
U ==> 2
M ==> 3
```

Strings coded as numbers:

```
MIU ==> 123
MUMUMU ==> 131313

empty string ==> 0
    (this is why I'm not following Hofstadter's choice U ==> 0!)
```

Example - (Formal)TNT:

```
A ==> 626
: ==> 636
a ==> 262
= ==> 111

Aa:a=a  ==>  626262636262111262
```

Generally:

Code the symbols of the alphabet by natural numbers which are all of the same length when written as decimals.

Then code a string **S** by the natural number with decimal representation the concatenation of the decimal representations of the codes for the symbols. This is the _Gödel number_ of **S**, written **[S]**

[well actually it's written with only the top halves of '[' and ']', but we'll have to live with '[S]' in ASCII!]

Coding rules:

Example - MIU system:

```
(I)     XI   |->  XIU
(II)    MX   |->  MXX
(III)   XIIIY  |->  XUY
(IV)    XUUY   |->  XY
```

We want a formula **Produces_I(x,y)** which is true precisely when **x** codes a string of the form "XI" and **y** codes the corresponding string "XIU". So let **Produces_I(x,y)** be

```
Ez:<x = ((10*z)+1) /\ y = ((100*z) + 12)>.
```

How about **Produces_II**? How do we check that a number's decimal representation starts with '3'?

We need exponentiation...

Lemma:

There is a formula **Exp(x,y,z)** which is true precisely when **z = x^y**

Proof:

Later!

Let **HasLength(x,y)** be

```
<Ez: <Exp(10,y,z) /\ <x < z /\ z <= 10*x>>
    \/ <x=0 /\ y=0>>    // ugly special case for the empty string
```

Now let **Concat(x,y,z)** be

```
Ey':<HasLength(y,y') /\ Ey'':<Exp(10,y',y'') /\ z = ((x*y'')+y)>>.
    (rewritten as normal maths: z = x*10^{length(y)}+y)
```

So given strings **X** and **Y**, **Concat([X],[Y],z)** is true iff **z = [XY]**.

Now **x** codes for a string of the form MX iff **Ez:Concat(3,z,x)** holds, and we can define **Produces_II(x,y)** to be

```
Ez:< Concat(3,z,x) /\ Concat(x,z,y) >
```

Similarly, let **Produces_III(x,y)** be

```
Ez:Ez':< Ex': < Concat(z,111,x') /\ Concat(x',z',x) > /\
           Ey':< Concat(z,3,y') /\ Concat(y',z',y) > >
```

And **Produces_IV** is similar.

Generally:
The same formula Concat works for any coding of any Post formal system, and arbitrary rules can be expressed by formulas similar to those above.

Coding derivations:
A derivation is a sequence of strings ("lines"). We need something new!

Lemma [Gödel's **\beta Lemma]**:
We can code arbitrarily long lists of arbitrarily large natural numbers as pairs of natural numbers:

There is a formula **ListElement(x,y,z)** such that for any finite sequence of natural numbers **a_0,a_1,...,a_n**, there is a natural number **c** such that for any **i <= n**,
```
    ListElement(c,i,z)
```
is true precisely when **z = a_i**.

Notation: for terms **s**,**t**,**r**, we will write
```
    [s]_t = r
```
as an abbreviation for
```
    ListElement(s,t,r)
```

Now we can code a derivation by a number **D** such that the **i**-th line of the derivation is the string with Gödel number **[D]_{i-1}**.

[Technical remark: this doesn't give us the ***length*** of the derivation, and may give us junk if we look at **[D]_i** for **i** greater than the length of the derivation. We could complicate things to handle that - but it won't actually matter for our definition of **Theorem(x)**, so we won't worry.]

We can also give the promised:

Proof of expressibility of exponentiation:
"exists a sequence **1=a_0,a_1,a_2,...,a_y=z** such that **a_{i+1} = xa_i** for all **i<y**";
```
Exp(x,y,z)  :=  Ex': <
        < [x']_0 = 1 /\ [x']_y = z > /\
        Ay':< y'<y =) Ez':< [x']_y' = z' /\ [x']_Sy' = (x*z') > > >
```

Expressing theoremhood:
Example: MIU-system
```
ProofPair_MIU(x,y):
    Ez:< [x]_z = y /\
        Az':< z' <= z  =)
            <[x]_z' = [MI] \/
                Ez'':<z'' < z' /\
                    Ey':Ey'':< <[x]_z' = y' /\ [x]_z'' = y''> /\
                        <Produces_I(y'',y') \/
                        <Produces_II(y'',y') \/
                        <Produces_III(y'',y') \/
                         Produces_IV(y'',y')>>>>>>>>
```
True precisely when **x** codes for the sequence of lines of a valid MIU-derivation, and **y** is the Gödel number of the last line.

```
Theorem_MIU(x):
    Ez:ProofPair_MIU(z,x)
```

Generally:
Similar!

See exercises!

Proof of **\beta** lemma:
Recall: Chinese Remainder Theorem:
Suppose **m_1,...,m_n** are pairwise coprime (i.e. **gcd(m_i,m_j)=1** if **i!=j**).

Then given **a_i** such that 0 **<= a_i < m_i**, we can find **c** such that
c == a_i mod **m_i** for all **i**.

["Right" way to think about it: the point is that if **M** is the product
M := \Pi_i m_i,
then the obvious map
Z/MZ --> \Pi_i Z/m_iZ
x/MZ |-> (x/m_1Z, ..., x/m_nZ)
is a ring isomorphism.]

Now define, for **c**,**d**,**i** in **N**,
\beta(c,d,i) := rem(c, (d(i+1)+1)))
where **rem(n,m)** is the unique natural number in [0,**m**) such that
n == rem(n,m) mod **m**

Claim: given a finite sequence **a_0,...,a_n**, there exist **c** and **d** such that
for **i=0**,...,**n**,
\beta(c,d,i) = a_i
Proof:
Let **d** be greater than all **a_i** and divisible by 1,...,n; e.g. we could
set **d := (n+1)!*\Pi_i a_i**.

Then as **i** ranges through 0,...,**n**, the numbers (**d(i+1)+1**) are pairwise
coprime.

Indeed: suppose **p** is prime, **p | d(i+1)+1** and **p | d(j+1)+1**,
with **i < j <= n**.
Then **p** does not divide **d(i+1)**, hence **p** does not divide **d**.
But **p | (d(j+1)+1 - d(i+1)+1) = d(j-i)**, so **p | (j-i)**.
But 0 **< (j-i) <= n**, so (j-i) **| d**. Contradiction.

So by the Chinese remainder theorem, we can find a **c** as required.

It remains to code the pair **(c,d)** as a single natural number...
Here's a direct approach:
t(x,y) = (x+y)(x+y+1)/2 + y = [(x+y)th triangular number] **+ y**

```
                                             .
                                             ..
the graph of which, with x increasing        ...
to the right and y increasing upwards,       9...
starts off as shown to the right:            58...
                                             247...
                                             0136...
```

So now we can let **ListElement(x,y,z)** be
Ez':Ez'':<t(z',z'')=x /\ \beta(z',z'',y)=z>

Arithmoquining

"yields falsehood when preceded by its own quotation" yields falsehood when
preceded by its own quotation.

Does it?

Abstractly:
If we have an "incomplete" sentence - one which requires a noun to make it
a sentence
e.g.
* yields falsehood.
* I like **x.**
* is missing a noun.
* The string _ has an underscore in it.

- we can _quine_ it: put the quotation of the incomplete sentence in for the
missing noun
resulting quines:
* "yields falsehood" yields falsehood.
* I like "I like **x.**".
* "is missing a noun" is missing a noun.
* The string "The string _ has an underscore in it." has an
underscore in it.

.

Now if the incomplete sentence says something about the quine of the missing noun, then its quine will say that thing about itself!

Simple example:
Let **U** be the string:
The quine of **x** is a self-referential sentence.
Then the quine of **U** is the string **S**:
The quine of
"The quine of **x** is a self-referential sentence"
is a self-referential sentence.
So **S** says that the quine of **U** is self-referential.
i.e. **S** says that **S** is self-referential!

Referring by name to quining is arguably cheating... we can give a more explicit recipe, like:

The string resulting from replacing the underscore in the string _ with the quotation of that string is 248 characters long.

|
| Quine
v

The string resulting from replacing the underscore in the string "The string resulting from replacing the underscore in the string _ with the quotation of that string is 246 characters long." with the quotation of that string is 246 characters long.

[Etymology: Willard Quine, philosopher; via Hofstadter]

Implementing this trick in arithmetic ("arithmoquining"):
Given a wff **\phi** whose only free variable is **x**, the _arithmoquine_ of **\phi** is the formula **AQ_\phi**:
Ex:<x = [\phi] /\ \phi>

So this is a sentence which claims of **[\phi]** whatever **\phi** claims of **x**.

(analogy:
incomplete sentence **<==>** wff with a free variable
sentence **<==>** sentence
noun **<==>** numeral
quotation **<==>** Gödel number)

[why the trick with Ex:? Why not just use substitution, letting **AQ_\phi(x)** be **\phi([\phi])**? Answer: because the following claim would then be much harder to prove.]

Claim: Arithmoquining is expressible:
There is a formula **Arithmoquine(x,y)** such that if **\phi** is a formula whose only free variable is **x**, then **Arithmoquine([\phi],z)** holds iff **z = [AQ_\phi]**.
Proof:
[AQ_\phi] = [Ex:<x = [\phi] /\ \phi>]

So **AQ_\phi** is the concatenation of "**Ex:<x=**", the numeral of **[\phi]**, "**/**", **\phi**, and "**>**".

So the only tricky part is getting the Gödel number of the numeral **\overline{[\phi]}**...

Let **GödelNumeral(x,y)** say that there exists **z** such that **[z]_0 = [0]**, **[z]_x = y**, and for all **x'**, **[z]_Sx'** is the Gödel number of the concatenation of "**S**" and the string coded by **[z]_x'**.

[In gory detail:
GödelNumeral(x,y) :=
Ez:<<[z]_x = y /\ [z]_0 = [0]> /\ Ax'< x'<x =)
Ez':Ez'':<< [z]_x' = z' /\ [z]_Sx' = z''> /
Concat([S],z',z'')>>>
]

```
        Then Arithmoquine(x,y) :=
            Ez:<GödelNumeral(x,z) /\ Concat( [Ex:<x=], z, [/\], x, [>], y)>

            (where Concat(x,x',x'',x''',x'''',y) says that y codes the
            concatenation of the five strings coded by x-x''''; we can define
            Concat(x,x',x'',x''',x'''',y) to be

                Ez':Ez'':Ez''':<Concat(x,x',z')
                    /\ <Concat(z',x'',z'')
                    /\ <Concat(z'',x''',z''')
                    /\ Concat(z''',x'''',y) >>>>
            )
```

Now let **S** be a Post formal system, and let **U** be the wff

```
    Ey:<Arithmoquine(x,y) /\ ~Theorem_S(y)>
```

"The arithmoquine of **x** is not a **S** theorem"

Let **G := AQ_U** be the arithmoquine of **U**:

```
    Ex:<x=[U] /\ U>
```

in full:

```
    Ex:<x=[Ey:<Arithmoquine(x,y) /\ ~Theorem_S(y)>]
        /\ Ey:<Arithmoquine(x,y) /\ ~Theorem_S(y)>
```

So **G** is true iff the arithmoquine of **U** is not a **S** theorem.

But **G** is the arithmoquine of **U.**

So **G** is true iff **G** is not a **S** theorem.

Now, the argument at the start of the section applies to **G**:

Theorem [Semantic G1T, Post formal system version]:
No Post formal system is both sound and complete for **N.**
Proof:
Suppose **S** is **N**-sound.

If **G** is false in **N**, then **G** is an **S**-theorem.
So **G** is true in **N** - contradiction.

So **G** is true in **N.** So **G** is not an **S**-theorem.

So **S** is **N**-incomplete!

Incompletability

So, TNT is not **N**-complete. It fails to prove the true sentence **G_TNT.**

But we know that **G_TNT** is true, so we can just add it as an axiom!

Let **TNT_2 :=** TNT **\cup { G_TNT }**.

Problem: if we add the axiom **G_TNT** to our Post formal system, we get another Post formal
system! So again, we can find a sentence **G_{TNT_2}** which is true, but not a
theorem of **TNT_2.**

Fine... let's add that too!

Let **TNT_3 :=** TNT **\cup { G_TNT**, **G_{TNT_2} }**.

But... again, the theorem applies, and we get **G_{TNT_3}** which is true but not
provable in **TNT_3.**

But! This procedure defines **TNT_n** for all **n**, so we can define

TNT_\omega := TNT **\cup { G_TNT**, **G_{TNT_2}**, **G_{TNT_3}**, **...** }.

A Post formal system is only allowed to have finitely many axioms, so we appear to have broken free of the incompleteness theorem!

This is a slightly ugly set to have as axioms, but it isn't too bad - we can tell whether or not a sentence is one of the axioms, because there's a definite pattern to the sentences **G_{TNT_n}.** So if **TNT_\omega** were complete, we'd be happy!

But. Precisely because there is this pattern, we could find a Post-formal system which produces { **G_TNT**, **G_{TNT_2} ...** } as theorems (and no other TNT-wffs). If we add this to FormalTNT, we'll have a Post formal system which proves precisely the TNT-sentences which **TNT_\omega** does... and hence by G1T, **TNT_\omega** isn't complete either!

That's a bit of an ad-hoc argument. We can be much more general:

Computability

The question arises: how strong is this theorem? Our notion of a Post formal system looked pretty restrictive, after all. So should we be surprised or worried that arithmetic truth is not captured by one?

To explore this issue, we will need to consider the concept of an algorithm.

"Definition": an _algorithm_ is an explicit, deterministic, step-by-step
    procedure for performing a calculation on some input data.
    Given input, it may _return_ a result, or it may never return anything
    (because the procedure keeps going forever, or because it fails at some
    point).

Definition:
    A partial function **f : N -> N** is _computable_ (synonyms: _**effective**_, _**recursive**_) if there is an algorithm which takes a natural number **n** as input, and
        * if **f** is defined at **n**, it returns **f(n)**.
        * if **f** is not defined at **n**, it never returns anything.

    A subset **X** of **N** is _computable_ (synonyms: _**decidable**_, _**recursive**_) if there is an algorithm which takes a natural number **n** as input and
        * if **n** is in **X**, returns True
        * if **n** is not in **X**, returns False

    A subset **X** of **N** is _computably enumerable_ (synonyms: _**semidecidable**_, _recursively **enumerable**_) if there is an algorithm which takes a natural number **n** as input and
        * if **n** is in **X**, returns True
        * if **n** is not in **X**, never returns anything.

    Similarly for functions **N^n -> N** and subsets of **N^n**, using algorithms which take **n** inputs (or using a coding function **N^n --> N**).

Lemma:
    (**i**) **X (= N** is computable iff **X** and its complement **N\X** are **c**.e.
    (ii) a nonempty set **X (= N** is **c**.e. iff it is the range of a total
        computable **f : N -> N.**
    (iii) **f : N -> N** is computable iff its graph **\Gamma_f** is **c**.e.
Proof:
    (**i**) **=>**: clear
        <=: given **n**, simultaneously run the algorithms which semidecide **X** and
            N\X; one will eventually return, telling you whether **n\in X.**

    (ii)
        <=: given **n**, compute **f(0)**, **f(1)**, ...; if ever **f(i)=n**, return True.
        =>: First, suppose **X** is infinite. Consider the following procedure for
            producing a list of elements of **X**:
                do the following with **i=0**, then with **i=1**, then 2,3,...:
                    1) start the semidecision procedure for testing if **i\in X.**
                    2) for each currently running semidecision procedure:
                        run it for one step; if it returns True, meaning
                        that **j\in X**, add **j** to our output list.
            Every element of **X** will eventually appear on the output list, with

no repetitions.

Now to compute **f**: given **n**, run the above listing algorithm until it has output **n** numbers. Return the nth.

In the case that **X** is finite (which is an uninteresting degenerate case), say **X = {a_0,...,a_k}**, define **f(n) := a_n** if **n <= k**, and for **n > k** define **f(n) := a_0**. This is clearly computable.

(iii) **=>**: easy
<=: given **n**, enumerate **\Gamma_f** as in (ii); if ever **(n,m)** is produced, return **m**.

Fact:
Many precise definitions of "algorithm" have been given; they are all equivalent: whichever notion of "algorithm" you use to define which functions and sets are computable, you get the same collection of functions and sets.

Moreover, they are precisely those which are "intuitively computable"!

A system for computation which computes precisely these functions and sets is called _Turing **complete_**.

"computable" means "computable by some (any) Turing complete system".
(sim **c**.e.)

Examples of Turing complete systems:
mathematical abstractions: **\mu**−recursive functions, **\lambda**−calculus, Turing machines, register machines, string rewriting systems;
physical systems: digital computers (with infinite RAM), Babbage's Analytical Engine (never built);
programming languages (FLooP, **C**, Scheme, Prolog, etc);
cellular automata: Conway's game of life, Rule 110;
esoteric programming languages (befunge, **brainf*ck** etc);
surprising places: molecular biology, **MtG(?)**, asciiportal...

Example of a Turing complete system:
Register machines (see below)

Church−Turing Thesis:
"There is nothing beyond Turing completeness"

Any function which can be calculated, in any reasonable sense of the word, is computable by any Turing complete system.

Fact: Post formal systems are Turing complete:

Let **A** be a finite alphabet. Fix a Gödel numbering of **A**−strings.

Let **\Sigma** be a set of **A**−strings.

Then the set of Gödel numbers of elements of **\Sigma** is **c**.e. iff there exists a Post formal system **S** in an alphabet **A'** containing **A** such that **\Sigma** is the set of **A**−strings which are **S**−theorems.

So we obtain:

Theorem [Semantic G1T]:
The set of true TNT−sentences is not decidable, or even **c**.e..
Proof:
If it were **c**.e., there would be a Post formal system **S** such that a string in the alphabet of TNT is an **S**−theorem iff it is a true sentence.
But this contradicts the Post formal systems version of Semantic G1T.

Remark:
If the set **Th(N)** of true sentences ***were* c**.e., then it would be computable. Indeed: **\sigma** is false iff **~\sigma** is true, so the complement of **Th(N)** would also be **c**.e.

[Decided to omit this... it's a more conventional statement, but giving it as well as the above statements would I think be obfuscatory. It's also a bit limiting, since it restricts us to the language of arithmetic (whereas we might want to consider e.g. ZF). The notion of a "logically adequate" formal system in section 5 substitutes for this.

Definition:
A _recursive axiomatisation_ for a structure **N'** in the language of arithmetic is a computable set of sentences **\Sigma** such that for any sentence **\sigma**,
N' |= \sigma iff **\Sigma |= \sigma**

Theorem [Semantic G1T, axiomatisability version]:
N does not have a recursive axiomatisation.
Proof:
By Gödel's completeness theorem,
\Sigma |= \sigma <=> \Sigma |- \sigma,
where recall the latter means that **\sigma** is a theorem of **PRED+\Sigma.**

But the set of theorems of **PRED+\Sigma** is computably enumerable, by enumerating derivations.
]

In particular, adding a **c**.e. set of true axioms to TNT will not yield completeness. In this sense, TNT is "incompletable".

See Figure 18 in Hofstadter.

For contrast, let me mention:

Fact [Tarski]:
The set of true sentences in the real field **<R;0,+,\cdot> *is*** decidable!

Same for the complex field **<C;0,+,\cdot>**.

Register machines

Theoretical computer, comprising infinitely many "registers" **R_0**, **R_1**, ... each containing a natural number.

A register machine program is a finite string in the alphabet
+ - ( ) ; . 0 1 2 3 4 5 6 7 8 9,
interpreted as instructions to alter the contents of the registers:
"**n+**" means "increment the contents of **R_n** by 1"
"**n**-" means "decrement the contents of **R_n** by 1 (or leave it at 0)"
"x;y" means "do **x** then do **y**"
"**n(x)**" means "do **x** while **R_n** does not contain 0"
"." means "stop".

A (well-formed) program implements a partial function **f:N->N** as follows:
to determine **f(n)**, first set **R_0** to **n** and all other **R_i** to 0. Then run the program. If the program stops, then **f(n)** is the contents of **R_0** when it stops; else, **f(n)** is undefined.

(Similarly, it implements partial functions **N^n -> N** for any **n**, using **R_0,...,R_{n-1}** for the inputs.)

Example - a program computing **f(n) := 2*n**
```
    0(1+; 0-); 1(1-; 0+; 0+).
```
Example - a program computing **f(n) := n*n**
```
    0(1+; 2+; 0-);
    1( 1-;
       2(0+; 3+; 2-);
       3(2+; 3-)
     ).
```

Example - a program computing **f(n) := 1** if **n** is prime, 0 else
```
    0(1+; 2+; 0-); 2(0+; 2-);
```

```
    1( 1-;
        0(2+; 3+; 0-); 3(0+; 3-);
        2( 2-;
            2( 2-; 3+; 4+ ); 4(4-; 2+);
            3( 3-;
               1(4+; 5+; 1-);
               5(1+; 5-);
             );
            // we've set R_4 := R_1*R_2; now check if R_4 == R_0:
            4(4-; 5+; 6+);
            0(0-; 7+; 8+; 9+); 9(9-; 0+);
            5(5-; 7-);
            8(8-; 6-);
            7(6(.)); // return 0 if composite
         )
     );
    0(0-); 0+.
```

Fact: register machine programs are Turing complete - any computable function is computed by a register machine program.

[So one way to prove that Post formal systems are Turing complete would be to show that any register machine program can be simulated by a Post formal system, or equivalently that we can find a "universal" Post system which produces a string of the form "**n,m,k**" iff the nth register machine program (according to some Gödel numbering; see below) returns **k** on input **m.** Emil Post did something similar, but for the lambda calculus (which is also known to be Turing complete) rather than register machine programs]

The Halting problem

Fix a Turing complete system, e.g. register machine programs.

Via Gödel numbering, we can code the programs by natural numbers such that each number codes a program and
Run(n,m) := f_n(m) where **f_n** is the function computed by the program with code **n (undefined iff f_n(m) is)**
is itself computable.

["computation is computable!"]

Theorem [Turing]:
(**i**) The Halting Problem is undecidable:

Define Halts : **N^2 -> N** by
Halts(n,m)=1 if the program with code **n** ever returns anything when given input **m (i.e. Run(n,m) is defined)**, and
Halts(n,m)=0 otherwise.

Then Halts is not computable.

(ii) There exists a **c**.e. subset **H** of **N** which is not computable.
Proof:
(**i**) Suppose Halts is computable. Then so is **h : N -> N** defined by
h(n) := 1 if **Halts(n,n)=0**; undefined else.
But then **h** is computed by some program, say with Gödel number **n.**
Then **h(n) = 1** iff **Halts(n,n)=0** iff **h(n)** is undefined. Contradiction.

(ii) Let **H := { n | Halts(n,n)=1 }**. Then **H** is **c**.e. since Halts is, but if **H** were computable then **h** would be computable.

Remark:
We can use this to give an alternative proof of Semantic G1T:
since **H** is **c**.e., there is a formula **\phi(x)** such that **\phi(n)** is true iff **n\in H**
(this follows from Turing completeness of Post systems and the existence of formulas **Theorem_S(x)**; there are some technical details to fill in; see Assignment 10)

So if arithmetic truth is computable, then so is **H** - contradiction!

(Note: this proof has the same "ingredients" as our original proof -

showing that arithmetic is sufficiently expressive, then using a diagonalisation trick (which in this version, is in the proof of undecidability of **H**))

Analogue of the Halting problem for Post formal systems

[turns out to be not so satisfactory... I won't present this in lectures]

Fix a countably infinite alphabet **s_0,s_1**,...; code strings as natural numbers (using the **\beta** lemma).

Also code Post systems in this alphabet as natural numbers.

Then the binary relation "**\sigma** is produced by **S_n**" (written e.g. as **s_0^[\sigma]s_1s_0^n**) is **c**.e., so is itself implemented by a Post system **U** in this alphabet
(analogue of a universal Turing machine).

Claim: the set of productions of **U** is not computable.
Proof: Suppose it is computable, and let
        X := { s_0^n | s_0^n is not a production of **S_n }**
    (where **s_0^n** is the string **s_0s_0...s_0**).
    Then **X** is computable, so is the set of productions of some **S_n**.
    But then **s_0^n \in X** iff **s_0^n** is a production of **S_n** iff **s_0^n \not\in X**.

Remark: we could probably get away with a finite alphabet (2 symbols might be enough?), but we'd need a better version of the lemma that Post systems are Turing complete.

V: Introspection, and Gödel's Second Incompleteness Theorem

In this section, we present Gödel's Second Incompleteness Theorem (G2T). Along the way, we state a stronger version of G1T which doesn't require the semantic notion of **N**-consistency. The key idea for both of these is that TNT, like us, is able to prove things about what it (and its strengthenings) can prove...

Notation: If **S** is a formal system in an alphabet including that of TNT, we say
    "**S _proves_ \sigma**" and write
        S |- \sigma
    to mean that **\sigma** is a TNT-sentence which is also an **S**-theorem.

Definition:
    S is _logically adequate_ if
    whenever **S** proves some sentences **\sigma_1,...,\sigma_n** which together necessitate a sentence **\tau**, then **S** also proves **\tau**.
    Symbolically:
        if { **\sigma_1**, ..., **\sigma_n } |= \tau**
        and if **S |- \sigma_i** for **i=1**,...,**n**,
        then **S |- \tau**.

    (So PRED is logically adequate, as is TNT)

    S is _a strengthening of TNT_ if **S** proves every TNT-sentence which TNT does, i.e. TNT **|- \sigma => S |- \sigma**.

For the remainder of this section, we assume that **S** is a logically adequate strengthening of TNT.

In particular, **S** could be (Formal)TNT itself.

Definition:
    S is **_(negation-)consistent_** iff for no TNT-sentence **\sigma** does it hold
        S |- \sigma   and   **S |- ~\sigma**.
    S is **_(negation-)complete_** iff for every TNT-sentence **\sigma**
        S |- \sigma   or   **S |- ~\sigma**.

Remark:
    N-soundness **=>** consistency

N-completeness => completeness

If **S** is inconsistent, then **S |- \sigma** for all **\sigma**.

Fact 1:
TNT proves all true sentences of the form
Theorem_S(n)
Idea of proof:
If **Theorem_S(n)** is true, then **ProofPair(m,n)** is true for some m; similarly, all the existentials involved in the (lengthy) definition of ProofPair have natural number witnesses, and it becomes a matter of checking that TNT can prove simple (quantifier-free) arithmetical truths about terms. It can. Induction isn't needed.

Let **G := G_S** be the Gödel sentence of **S**.

Fact 2:
TNT "knows what **G** is":
TNT **|- < <G =) ~Theorem_S(G)> /\ <~Theorem_S(G) =) G> >**

[
Idea of proof (omitted in lectures):
This is morally straightforward, but technically less so.
Recall that **G** is **Ex:<x=[U] /\ U>**
where **U** is **Ey:<Arithmoquine(x,y) /\ ~Theorem_S(y)>**.
Now one can check, as in Fact 1, that
TNT **|- Arithmoquine([U],[G])**.
Furthermore, TNT **|- Ay:<Arithmoquine([U],y) =) y=[G]>**;
i.e. it understands (case-by-case) that Arithmoquine is a function.
This, perhaps surprisingly, is actually the most painful bit - see the notion of "**captures***" in chapter 9 of Peter Smith's "Gödel Without Tears" (linked from website).
The rest is basic logic, which TNT can do by Gödel's completeness theorem.
]

[
Actually, for Facts 1 and 2 we don't need much of TNT at all - just TNT' and the single TNT-theorem **Ax:<x=0 \/ Ey:x=Sy>**. This system
Q := TNT'+{Ax:<x=0 \/ Ey:x=Sy>}
is known as "Robinson arithmetic" (after Raphael Robinson, who isn't Abraham or Julia!).

Technically, to get Fact 2 in **Q** you have to modify the definition of Arithmoquine, to **Arithmoquine'(x,y) :=**
< Arithmoquine(x,y) /\ Ay':< y'<y =) ~Arithmoquine(x,y') > >.
They're equivalent in **N**, and TNT proves the equivalence, but **Q** doesn't.
]

Lemma 1:
If **S** is consistent, then **S |/- G**.

i.e. **N |= <Con(S) =) ~Theorem_S([G])>**
Proof:
Suppose **S |- G**.

Then by Fact 1, since **S** strengthens TNT,
S |- Theorem_S([G]).

But by Fact 2, since **S** strengthens TNT and is logically adequate,
S |- ~Theorem_S([G]).

So **S** is inconsistent.

Historical remark:
Lemma 1 formed half of Gödel's original proof of his First Incompleteness Theorem. Assuming a stronger form of consistency ("omega-consistency"), he proved that also **S |/- ~G**, showing that **S** is incomplete. **A** few years later, Rosser found a way to remove the assumption of omega-consistency, yielding:

Gödel-Rosser First Incompleteness Theorem [G1T]:
If **S** is consistent, it is incomplete.
Proof: omitted.

Definition:
Let **Con(S)** be the sentence
~Theorem_S([~0=0])
So **Con(S)** is true iff **S** is consistent.

Fact 3: TNT is able to prove Lemma 1:
TNT **|- <Con(S) =) ~Theorem_S([G])>**
Idea of proof:
Follow the proof of Lemma 1 (and hence the proofs of Facts 1 and 2), and see that TNT can do them... for example, we have to prove an "introspective version" of Fact 1:
TNT **|- Ax:<Theorem_S(x) =) "Theorem_TNT([Theorem_S(x)])">**
(where a little work is needed even to express the bit in quotes)

[
We need more of TNT for this than for the first two facts. We still don't need all of TNT, though: we only need induction axioms for "**\Sigma_1**-formulae", being those formulas which start with zero or more existential quantifiers, then have no more quantifiers of any kind.
]

Theorem [G2T]:
If **S** is consistent, **S |/- Con(S)**
Proof:
Suppose **S |- Con(S)**.

Then by Fact 3, **S |- ~Theorem_S([G])**.

So by Fact 2, **S |- G**.

But this contradicts Lemma 1.

Appendix **A**: some remarks on Hofstadter's presentation of Gödel's theorems

It's a bit of a mess, really.

He spends a whole chapter on the difference between primitive recursion and recursion, so that he can say that TNT represents primitive recursive relations. That's true - although **Q** also represents general recursive relations, and since he isn't even sketching a proof it seems strange to mention only the easier result. But more to the point, despite making a big deal of this representability, it's never actually used in the Gödel statement he proves! He sketches a proof of only the semantic version of G1T, for which expressibility is enough. Of course, the representability is relevant to G2T, or to Syntactic G1T, but he doesn't even try to prove those. Relatedly, he seems to use soundness and consistency interchangeably.

I have great affection for the book, and on balance I don't regret using it for this course, but it has meant that the overlap between the course and the book is rather smaller than I'd have liked.

Appendix **B**: some remarks on Bays's presentation of Gödel's theorems

(Since we're on the subject of introspection...)

I hope it isn't a mess. Really.

The approach to Gödel's theorems this course has ended up taking is fairly idiosyncratic. To summarise: taking more seriously than he could possibly have intended a throwaway parenthetical at the start of Hofstadter's GEB, we build everything around the notion of a Post formal system. This makes a nice introduction to semantics of syntax, leading to the propositional and predicate calculi. From the point of view of Gödel, the main advantage of using Post systems is that, given a Post system **S**, the formula expressing "is an **S**-theorem" (i.e. **S**-production) is relatively simple. We were able to describe it essentially in full detail. This not only makes it about as clear as it could be that theoremhood is definable, but also makes it relatively easy to believe that **Q** can do the proofs.

This is to be contrasted with a more conventional approach, which would ***first*** show that recursive functions **f** are definable and that **Q** can prove true statements of the form **f(n)=k**, and then argue that since theoremhood is clearly **r**.e. (which is only clear once you've built up a solid appreciation for Turing completeness!) we get the same results.

Of course, we still need to talk about computability if we want to say that **Th(N)** is undecidable, but the Turing completeness of Post systems gives us an avenue in to that.

The main downside of avoiding talking directly about recursive functions is that we can't extract a clear statement that **Q** proves all true statements of the form **\phi(n)** for **\phi** an **r**.e. predicate, nor that **r**.e. corresponds to **\Sigma_1**.

So in retrospect, I think the Post-approach works quite well. I consider it a happy medium between (a) the truly fluffy approach of just asserting that computability is something **Q** can do, and syntactic stuff is obviously computable so duh, and (**b**) the fully Proper approach going via (primitive) recursive functions and properly demonstrating that the relevant syntactic operations are recursive (for which you might well want to go via some more appropriate Turing complete system, such as... Post systems, say?).

One regret I have regarding the use of Post systems, however, is the terminology, which I lifted from Hofstadter and kept unchanged throughout the course. Calling productions "theorems", and initial strings "axioms", is cute but not really appropriate in general. Better would be to have made a distinction at the stage when we defined 'well-formed' strings as the interpreted ones to which we assign meaning. "**S**-theorem" should have been defined as "well-formed **S**-production", explicitly leaving open the possibility of going via non-well-formed strings, perhaps using auxiliary symbols, to arrive at our theorems. This would nicely foreshadow the statement of Turing completeness of Post systems, and would be appropriately suggestive when we considering extending TNT by going outside of number theory, e.g. to ZFC; making use of auxiliary symbols (which we ***could*** optionally also interpret, yielding a consistency argument analogous to that for TNT (and a proof of **Con(ZFC)** from existence of inaccessible cardinals...)).

Appendix **C**: Idiosyncrasies

* TNT is usually known as PA.
* TNT' is almost Robinson's Q; more precisely, **Q** is TNT' along with an axiom saying that every non-zero number is a successor, **Ax:<~x=0 =)** Ey: **x=Sy>**. The point of **Q** is that it's enough for the syntactic Gödel-Rosser version of G1T we mentioned to go through: no strengthening of **Q** is both consistent and complete.
* PRED is one of numerous possible proof systems for "first-order predicate logic" in the language of arithmetic (and would not have to be changed much to accomodate other languages)
* The particular system PROP for propositional calculus was, as I hope was clear, not standard. But it's as good as any other. 'Detachment' is normally called 'Modus Ponens'. 'Switcheroo' doesn't have a common name to my knowledge, though it ought to.
* "formal system" is not normally synonymous with "Post system" as it was for us. There is no very precise definition of "formal system" in general currency, but the key idea is that the set of theorems provable in a formal system should be recursively enumerable. So by the Turing completeness of Post systems I stated, any formal system can be turned into a Post formal system.
* Relatedly, our terminology for Post systems, borrowed from Hofstadter, is non-standard, since they're not normally thought of as proof systems. What we called an 'axiom' would normally be called something like an 'initial word', and what we called a 'theorem' would normally be a 'production'. Also, they should really be called "Post canonical systems", or just "Post systems". They aren't actually particularly well-known, other theoretically simpler but less intuitive string-rewriting systems being more common, but they suited our purposes.
* "Semantic G1T" isn't common terminology; I nabbed it from Peter Smith. Normally when people (who know what they're talking about) talk about

Gödel's First Theorem they're referring to a version I never even mentioned, because it was superceded by Gödel–Rosser. The basic version, due mainly to Gödel, is this: no logically adequate strengthening of **Q** is both complete and **\omega**–consistent. Recall that 'complete' means 'negation–complete' means that for any **\sigma** it proves **\sigma** or proves **~\sigma**. We didn't define **\omega**–consistent, I'll do so now: it means that if for each **n** the system proves **\phi(n) (where that n has a bar on it)**, then the system doesn't prove **~Ax:\phi(x)**. Of course this implies that the system is consistent, because an inconsistent system proves everything, so this version is weaker than the Gödel–Rosser version which drops the "**\omega**". But people still sometimes talk about this original version anyway, essentially just for historical reasons.

* Also, it isn't normal to state G2T in terms of formal systems as we did; but doing so loses nothing. **A** more standard statement with the same power (I won't explain what all the terminology means, though... avoiding having to do so was why I only gave the Postal version!): let **T** be a recursively axiomatised theory in some language **L**, suppose **T** interprets a structure **N** in the language of arithmetic, and suppose the induced theory extends **Q**. Then if **T** is consistent, **T** does not include **Con(T)**, where the latter is considered as a sentence in **N**.