# 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, starts off as shown to the right:

```
.
..
...
9...
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.