Exercise [16.18]

In the text, Penrose states:

"Now it follows immediately from Turing's argument above that the family of true Π_1-sentences constitutes a non-recursively enumerable set"

In my opinion, given the actual description of Turing's argument presented in the preceding text, this rather stretches the meaning of the word "immediately"!

Accordingly, part 1 of my solution to this problem is to show *in detail* how the Turing argument given in the text can be used to "immediately" (!) show that true Π_1-sentences are not r.e. (recursively enumerable).

Part 2 of my solution is to adapt the part 1 proof to find a *particular* Π_1-sentence G(***F***), that must be true if a given formal proof system ***F*** is trustworthy, but that nevertheless can't be proved using ***F***.

As an addendum, I provide a much more direct route to the final proof, that also utilizes a Turing/Cantor style argument, but a different one from that given in the text.

Part 1:

The goal of this part is to show that true Π_1-sentences are not r.e. (recursively enumerable) by any formal proof system ***F***. I'll do this in two stages. Part (1a) will demonstrate that non-terminating Turing machine computations are not r.e.. Then in part (1b) I'll explain the relationship between Turing machine computations and Π_1-sentences, and use this relationship to complete the part (1) goal.

Part 1a: Proof that non-terminating Turing machine computations are not r.e.

The text discusses using Cantor's diagonal slash to prove that a particular subset of effective Turing machines (the ones that output only either 0 or 1) are not r.e., but it doesn't go into detail. So here are the details:

For convenience, let's call a Turing machine whose output is restricted to only 0 and 1 a "membership machine" (since each such machine defines the membership of a particular subset of $\mathbb{N}$).

Let us ASSUME that the set of effective membership machines is r.e. - i.e. that there is an effective Turing machine ***MEff*** that lists *every* effective membership machine (in no particular order, and perhaps with duplicates). We can state this more formally: $\forall n \in \mathbb{N}$, $\boldsymbol{T}_{\boldsymbol{MEff}(n)}$ is an effective membership machine, and for every effective membership machine $\boldsymbol{M}$, there is at least one n such that $\boldsymbol{M} = \boldsymbol{T}_{\boldsymbol{MEff}(n)}$. But now consider the particular membership machine defined as follows:

$$\boldsymbol{M}(x) = \boldsymbol{T}_{\boldsymbol{MEff}(x)}(x) + 1 \pmod 2$$

It's clearly possible to construct a Turing machine that performs the calculation on the RHS (it would involve computing $\boldsymbol{MEff}(x)$, then using the universal Turing machine $\boldsymbol{U}$ to compute $\boldsymbol{T}_{\boldsymbol{MEff}(x)}(x) = \boldsymbol{U}(\boldsymbol{MEff}(x), x)$. Since all the elements involved in calculating the RHS are effective (i.e. all the computations are guaranteed to halt and produce a result), $\boldsymbol{M}$ is clearly also an *effective* membership machine. But by construction it differs from *every* $\boldsymbol{T}_{\boldsymbol{MEff}(n)}$. Therefore, it is not part of the list enumerated by ***MEff***. This contradicts the assumption that ***MEff*** recursively enumerates every effective membership machine. So effective membership machines are not r.e.. Since effective membership machines are just a subset of effective Turing machines, this means that effective Turing machines are not r.e. either. ■

(Actually, the last sentence only follows if it's a computable matter to distinguish between Turing machines that are or are not membership machines. It is *iff* we consider only those Turing machines who's algorithm *explicitly* outputs only either a zero or a one as a membership machine. But in any case, the argument above, although phrased so as to apply to the recursive enumeration of effective *membership* machines, in fact works just fine if we apply it to the recursive enumeration of effective *Turing* machines generally. That approach gives us a more direct proof that effective Turing machines are not r.e.; it just doesn't happen to be how it was done in the text).

So, we've established that the set of *effective* Turing machines is not r.e.. What does this tell us about the *complement* of that set, i.e. the set of *faulty* Turing machines? Or about the set of non-terminating Turing machine *computations*? At first glance, it would seem that the answer might be "not much". Knowing that some set **S** is not r.e. doesn't (in general) tell us whether or not its complement is or is not r.e.. However it turns out that we can show that if non-terminating Turing machine computations were r.e., then this would imply that effective Turing machines were also r.e.. And since we've just shown that the latter isn't true, the former can't be either. It remains, then, to demonstrate this implication.

I'll do this formally by constructing a series of Turing machines with particular properties, given here as pseudo-code algorithms, culminating in one that recursively enumerates all effective Turing machines. To make this rather cumbersome set of constructions easier to follow, a brief summary of the overall plan might be helpful before I get into the details:

Basically the idea is that if you could list all the non-terminating computations, you could determine for sure whether any computation will stop or not: Just run the computation and at the same time step through the list. Either the computation will terminate, or it'll be in the list - and therefore within in a finite time you'll know if it terminates or not. So now consider the following operation: For a given (validly coded) Turing machine ***T***, consider its action on each natural number 0, 1, 2,... in turn, testing to see if each computation terminates, until you find one that doesn't. This operation is *itself* a computation, and it terminates iff ***T*** is faulty. So by testing whether *this* computation terminates, you can determine for sure within a finite time whether an arbitrary Turing machine is effective or not. To recursively list all effective Turing machines then, you could first list ALL Turing machines, and then just replace all the faulty ones in the list (which you can now detect) with some particular effective one.

That's the summary. The details are as follows:

Let us ASSUME that the set of non-terminating Turing machine computations is r.e. - i.e. that there is some effective Turing machine ***TCNTlist*** that lists *every* non-terminating Turing machine computation. More formally, for $\forall n \in \mathbb{N}$, ***TCNTlist***(n) successfully outputs an ordered pair (t,x) such that the computation $\boldsymbol{T}_t(x)$ never halts; and for every possible non-halting computation, there is at least one n such that ***TCNTlist*** outputs the (t,x) pair corresponding to that particular computation.

(Note Penrose's comments on p375 about encoding/decoding pairs of numbers as a single number - they apply equally well to the *outputs* of Turing machines as well as their inputs; also we can in fact encode larger n-tuples of numbers and not just pairs, a fact which we will make use of later).

Now, given ***TCNTlist***, we can define the following series of Turing machines:

TCNT takes arguments (*t*,*x*) and outputs a 1 if the computation $T_t(x)$ never halts; if it DOES halt, ***TCNT*** will not. Note that ***TCNT*** works by simply attempting to recursively enumerate all non-terminating computations until it finds the one that matches its argument. Note that the argument '*t*' is always assumed to represent (encode) a *validly coded* Turing machine, for this and all the following Turing machines; except for the last one, ***TEffList***, they don't include any logic to handle the case when the argument representing a Turing machine does not represent a syntactically valid (correctly coded) one.

```
FUNCTION TCNT(t,x)
{
  i ← 0
  WHILE (TCNTlist(i) ≠ (t,x)) DO i ← i+1
  OUTPUT 1
}
```

TCHalts is an *effective* Turing machine that takes arguments (*t*,*x*) and outputs a 1 if the computation $T_t(x)$ halts, or zero if it does not. It works by computing both ***TCNT***(*t*,*x*) and $T_t(x)$ in parallel until one or the other halts; given the construction of (and assumptions behind) ***TCNT***, it is guaranteed that this will always happen. Note that ***U*** is the universal Turing machine, so ***U***(*t*,*x*) ≡ $T_t(x)$.

```
FUNCTION TCHalts(t,x)
{
  Initiate both the computations A = TCNT(t,x) and B = U(t,x)
  REPEAT
  {
    perform a single step of computation A
    perform a single step of computation B
  }
  UNTIL (A has halted OR B has halted)
  IF (A has halted) OUTPUT 0
  IF (B has halted) OUTPUT 1
}
```

TNT takes argument *t* and outputs a 1 if the Turing machine T_t is faulty; otherwise ***TNT*** does not halt. Note that ***TNT*** works by checking the action of T_t on each natural number in turn using ***TCHalts*** until one of these computations is found to be non-terminating.

```
FUNCTION TNT(t)
{
  x ← 0
  WHILE (TCHalts(t,x) = 1) DO x ← x+1
  OUTPUT 1
}
```

TEff is an *effective* Turing machine that takes argument *t*, and outputs a 1 if the Turing machine T_t is effective, or a zero if its faulty. Its existence would prove that the set of effective Turing machines is not just r.e., but *recursive* (a stronger condition). It works by using ***TCHalts*** to determine if the computation ***TNT***(*t*) halts or not. (Since ***TNT*** is a turing machine, it corresponds to some T_t where $t = t_{TNT}$, the constant that encodes ***TNT***).

```
FUNCTION TEff(t)
{
  OUTPUT 1-TCHalts(t_TNT, t)     // t_TNT is the constant that encodes
}                                // Turing machine TNT
```

Finally, ***TEffList*** is an *effective* Turing machine that recursively enumerates all effective Turing machines; its argument is any natural number. I include its definition here only because the book mentions but doesn't explicitly show, that all recursive sets are recursively enumerable. It simply returns its argument if that argument represents (encodes) an effective Turing machine. If not, it returns a constant ('f_0' in the pseudocode) representing some particular effective Turing machine. Which one doesn't matter; for example you could take the Turing machine that immediately halts and returns 0, performing no actual computation.

```
FUNCTION TEffList(n)
{
  IF (n encodes a syntactically valid Turing machine)
     AND (TEff(n) = 1)
    OUTPUT n
  ELSE OUTPUT f_0           // f_0 is a constant encoding some effective
}                           // Turing machine (any one will do)
```

This last Turing machine, ***TEffList***, recursively enumerates all effective Turing machines, establishing the implication we wanted to show.

Earlier, we proved that this recursive enumeration is impossible, and so from this contradiction, we can conclude that the initial assumption – that the set of non-terminating Turing machine computations is recursively enumerable – must be false!

■ part (1a)

Part 1b: Demonstration that the set of true Π_1-sentences are not r.e. by any formal proof system

Penrose introduces the terminology of a "formal proof system" without actually defining it. For our purposes here, we need only assume that a "formal proof system" ***F*** consists of some formal language for representing mathematical statements, as well as axioms and a set of inference rules that can be "mechanically implemented": That is to say, an effective Turing machine can be defined that will *either* recursively enumerate all mathematically correct proofs in ***F***, *or else* that will check the correctness of a supplied 'proof' against the rules of ***F***. Note that the latter implies the former: Given the latter, we could just recursively enumerate *all* possible proofs, and then use our 'correctness checker' to replace those that are not correct (with some particular correct proof) - thus enabling us to recursively enumerate all *correct* proofs. So a formal system ***F*** enables us to recursively enumerate all correct proofs in ***F***. We can discard the "proof" part and keep only the *conclusions* of the proofs, and if we do that, we are left with a recursive enumeration of all mathematical statements provable in ***F***. For our purposes here, that is the key property possessed by a "formal proof system" ***F***: It permits the recursive enumeration of all statements provable in ***F***.

Now, if we have constructed ***F*** so that its axioms are "mathematically true", and if its inference rules preserve mathematical truth, then we can expect that the "provable statements in ***F***" are in fact all *true* statements of mathematical fact. The statements *provable in* ***F*** are r.e.. We aim to demonstrate that the set of *true* mathematical statements is NOT r.e., and that therefore, there are *true* statements expressible in the language of ***F*** that nevertheless cannot be *proved* using (only) the rules of ***F***.

We will perform this demonstration by showing that a certain class of statements, the "Π_1-sentences" are each equivalent to the assertion that a particular Turing machine computation does not terminate, and make use of the result from part (1a), that the set of non-terminating Turing computations is not r.e.. Penrose once again introduces the terminology of a "Π_1-sentence" without properly defining it, but from the examples he gives it's clear that (for our purposes here at least) we can take the "Π_1-sentences" to be the set of statements of the form: "$\forall n \in \mathbb{N}$, *some statement about n*". Because we've already seen how a single natural number can be used to encode a pair of numbers (or more generally an n-tuple), this automatically also includes all statements of the form: "$\forall x, y, ..., z \in \mathbb{N}$, *some statement about x, y, ..., z*". An example is Fermat's Last Theorem: "$\forall a, b, c, n \in \mathbb{N}$, $(a+1)^{n+3} + (b+1)^{n+3} \neq (c+1)^{n+3}$". We can denote a "statement about n" as '$P(n)$' - a *predicate* that is (potentially) either true or false for each value of n. Then the Π_1-sentences are just the set of statements of the form "$\forall n \in \mathbb{N}$, $P(n)$ (is true)"; different Π_1-sentences are distinguished only by different choices of P.

There is a natural way to associate a given (arbitrary) Π_1-sentence, "$\forall n \in \mathbb{N}$, $P(n)$" with a particular Turing machine computation (given as pseudocode):

```
FUNCTION PI1P(x)
{
  n ← 0
  WHILE P(n) DO n ← n+1
  OUTPUT 0
}
```

This turing machine tests the truth of $P(n)$ for each natural number 0, 1, 2, ... etc. in turn, and halts when it finds an n for which $P(n)$ is false. (Note that it makes no use of its argument x). So the computation of PI1P(0) (taking x=0 say) halts **iff** the given Π_1-sentence is false.

Can we make use of this particular relationship between Π_1-sentences and Turing machines to demonstrate the result we are looking for? Actually we cannot. The problem is that we have mapped the set of all true Π_1-sentences onto a *subset* of all non-terminating Turing machine computations. It's easy to find a non-terminating computation that doesn't match the above code, for any choice of P; the simplest is probably just "LOOP INDEFINITELY". And knowing that the *entire set* of non-terminating computations is not r.e. doesn't guarantee that *subsets* of it won't be! (E.g. we showed that effective membership functions were not r.e., but if we take only those membership functions corresponding to singleton subsets - i.e. the subsets {0}, {1}, {2}, ... etc, then this *subset* of effective membership functions certainly *is* r.e.). In fact the relations between r.e. sets and subsets is this: If a set is r.e., its subsets are r.e.. Inverting this, we see that if a subset is not r.e., than neither is the complete set. But knowing that a set is *not* r.e. tells us nothing about its subsets.

What we need to do is find some association between the set of *all* non-termination Turing computations, on the one hand, and some set of true Π_1-sentences on the other. Then we could infer that this set of true Π_1-sentences is not r.e., and hence neither is the set of all true Π_1-sentences, or the set of all true mathematical statements generally.

Put another way, given an arbitrary Turing machine computation, can we come up with a Π_1-sentence that's true iff the computation never terminates? It's easy enough to do if you remember that Turing machine computations are carried out in discrete steps. Then for an an arbitrary computation $\boldsymbol{T}_t(x)$, non-termination is equivalent to the sentence "$\forall n \in \mathbb{N}$, $\boldsymbol{T}_t(x)$ does not halt after step n". To make this into a statement representable in a formal mathematical language, you'd need to capture the mechanisms by which Turing machines function in the form of mathematical expressions. Penrose never describes in this book exactly what Turing machines are defined to be, except to say they're "idealized computers", so it might be pretty tough to proceed with nothing but this vague idea to go on. I won't supply those details here either, since they're largely unnecessary. The only really important thing to know is that Turing machines have a well-defined "internal state" that changes in a fairly simple way at each step of their computations. (For those who know how Turing machines are actually defined, I'm lumping together the tape contents, the tape position, the machine state, and the state transition function together into what I'm calling "internal state"). If

we call the space of all possible Turing machine internal states '**TMS**' then we can quite easily define the functions:

I: $\mathbb{N}$ x $\mathbb{N}$ → **TMS**

which maps a (*t*,*x*) pair representing the computation $\boldsymbol{T}_t(x)$ onto the internal state that represents the starting point of that computation,

S: **TMS** → **TMS**

which maps any given internal state to the state that follows it after a single computation step, and

H: **TMS** → {*true*, *false*}

a predicate (i.e. a function returning either *true* or *false*) that is true when its argument represents the internal state of a computation that has halted.

In this way we can replace the informal English sentence:

> "Turing machine computation $\boldsymbol{T}_t(x)$ does not halt"

with the formal mathematical Π_1-sentence

> $\forall n \in \mathbb{N}, \neg H(S^n(I(t,x)))$

(Where S^n denotes repeated function composition, i.e. $S^3(q)$ means $S(S(S(q)))$, and '¬' means 'not', i.e. $\neg H(r)$ means "$H(r)$ is false").

Of course, to do this we must be using a formal proof system ***F*** with axioms and syntax powerful enough to express all the elements of this sentence, and to define the three functions I, S, and H that we require. However it turns out that this isn't a very strong restriction on ***F***, and that any formal proof system capable of representing certain basic properties of arithmetic will suffice.

What we have just achieved is to show that there is a true Π_1-sentence that uniquely corresponds to every non-terminating Turing machine computation (and also a false one that corresponds to each terminating computation… but that's not important right now). Since the non-terminating computations are not r.e., neither are their corresponding true Π_1-sentences. So **the set of true Π_1-sentences is not recursively enumerable**, which is the result we were trying to demonstrate in this part.

■ part (1b)

(INTERESTING SIDE NOTE: Our first attempt to relate the set of Π_1-sentences to the set of Turing machines resulted in an injective function from the former to the latter; the second attempt, which we used in the proof, resulted in an injection from the latter to the former. According to exercise [16.10], this means that it's possible to define a bijection between the two sets. I'll leave it to the reader to figure out what that bijection might look like!).

Part 2:

We have demonstrated in part 1 that the set of true mathematical statements expressible in English as "Turing computation $\boldsymbol{T}_t(x)$ does not halt" is not r.e.; and hence neither are mathematical truths generally. We did this by assuming this set *was* r.e., and showing that this implied that the set of effective Turing machines was also r.e., which we know to be false. However, unlike (say) the Cantor's diagonal slash argument, which contradicts it's assumptions by constructing a concrete counterexample, our method of proof provided no such concrete counterexample. We seek one now. Specifically, let us suppose ***F***(*n*) is the effective Turing machine associated with the formal proof system ***F***, that recursively enumerates all (true) mathematical statements provable by ***F***. Then we seek to construct some mathematical statement, G(***F***), that isn't enumerated by ***F***(*n*) (and hence

can't be proven by ***F***), but which is true nonetheless. We'll do this by modifying and simplifying the part 1 proof in several steps until we have constructed such a statement.

The first rather trivial adjustment step we obviously need is to take ***F***(*n*) as the source of our recursive enumeration of non-halting computations. To do this we simply need to replace the first Turing machine in our series, ***TCNT***, with this new version:

```
FUNCTION TCNT(t,x)
{
  i ← 0
  WHILE (F(i) ≠ "∀n∈ℕ, ¬H(S^n(I(t,x)))") DO i ← i+1
  OUTPUT 1
}
```

Now, since ***F***(*n*) recursively enumerates *all* provable statements (in ***F***), this obviously includes all the provable statements of the specific form "$\forall n \in \mathbb{N}, \neg H(S^n(I(t,x)))$" – i.e. in English "Turing computation $T_t(x)$ does not halt". The fact that many other kinds of statement will be enumerated as well needn't concern us; it does not affect the validity of the algorithm.

The next step is to continue the proof all the way through the Cantor diagonal slash argument. Previously we used the *result* of that argument to contradict the *existence* of ***TEffList*** (or rather, to show that this Turing machine could not work as intended, and hence that the assumption behind its construction was false). However, actually working through the diagonal slash argument using ***TEffList*** to construct a "diagonal" Turing machine turns out to be a fruitful avenue of investigation, as we will see. We can construct such a Turing machine, ***TDiag***, by simply adding 1 to each of the diagonals in the list of Turing machines given by ***TEffList***. The algorithm is as follows:

```
FUNCTION TDiag(n)
{
  OUTPUT U(TEffList(n),n)+1
}
```

Expanding out the contents of ***TEffList*** and then ***TEff*** in turn directly into the algorithm yields this:

```
FUNCTION TDiag(n)
{
  IF (n encodes a syntactically valid Turing machine)
     AND (TCHalts(t_TNT,n) = 0)
    OUTPUT U(n,n)+1
  ELSE OUTPUT F_0(n)+1    // F_0 is some effective Turing machine
}                         // (any one will do)
```

Note that these are the same Turing machines (hence the same name) – all I've done is to present the same algorithm in a slightly different way.

Now, there are three simplifications we can make here. The first is to assume that all values of *n* encode valid Turing machines. This could be achieved by using an encoding that produces a unique, valid Turing machine for each value of *n* (i.e. the n^{th} valid Turing machine), or else, perhaps more easily, by simply substituting some particular validly-coded machine whenever an invalid code is provided. That is to say, we could define $\boldsymbol{U}(t,x) \equiv \boldsymbol{T}_t(x)$ when *t* is a valid code, and $\boldsymbol{U}(t,x) \equiv \boldsymbol{T}_{\text{default}}(x)$ when it isn't. It doesn't matter what $\boldsymbol{T}_{\text{default}}$ actually does, only that it's a respectable Turing machine. It could simply be the "`LOOP INDEFINITELY`" machine (making all invalid codes faulty, as Penrose suggests), or the "`HALT AND OUTPUT 0`" machine (making all invalid codes effective). It really doesn't matter. The only important thing to note is that the behaviour of our function $I(t,x)$ must also change accordingly, i.e. we must have $I(t,x) \equiv I(t_{\text{default}},x)$ whenever *t* is an invalid code. (Here t_{default} is the *valid* code that normally encodes Turing machine $\boldsymbol{T}_{\text{default}}$, obviously). At this point it might be worth mentioning that $\boldsymbol{U}(t,x)$ can actually be constructed using functions I, S, and H:

```
FUNCTION U(t,x)
{
  c ← I(t,x)
  WHILE (¬H(c)) DO c ← S(c)
  OUTPUT R(c)
}
```

Here *c* holds the internal state of the Turing machine computation we are performing, and R is the function:

R: **TMS** → $\mathbb{N}$

That returns the output value corresponding to a given Turing machine internal state, if that state is a halted state (i.e. it returns the output value of the corresponding completed computation). If it's not a halted state, the return value of R is undefined.

If $\boldsymbol{U}$ is defined in this way, that leaves the function I as the sole entity for translating (or associating) integer values with particular Turing machines. So if I is modified (as above) to produce a valid Turing machine for any input value, then effectively, there are no longer any "invalid" values, and we needn't bother checking for them specifically in our algorithm.

The second simplification is to note that for the purposes of our diagonal slash argument, the output of ***TDiag***(*n*) doesn't matter when $\boldsymbol{T}_n$ is not an effective Turing machine. Our list of effective Turing machines is constructed by first listing *all* Turing machines (by code), and then simply replacing the faulty ones with $\boldsymbol{F}_0$, which is just some (arbitrarily chosen) effective Turing machine. That means that $\boldsymbol{F}_0$ will appear many times in the list. It will appear in its "proper" position, the f_0'th position, as well as in every position *n* that encodes a faulty Turing machine. Our goal in constructing ***TDiag*** is that it should differ from every entry in the list, including $\boldsymbol{F}_0$. However, so long as $\boldsymbol{TDiag}(f_0) \neq \boldsymbol{F}_0(f_0)$, we are guaranteed that ***TDiag*** and $\boldsymbol{F}_0$ differ; we do not additionally need $\boldsymbol{TDiag}(n) \neq \boldsymbol{F}_0(n)$ to guarantee this, for those other positions *n* in which $\boldsymbol{F}_0$ appears. We can therefore replace the line "**ELSE OUTPUT** $\boldsymbol{F}_0(n)+1$" with the simpler line "**ELSE OUTPUT** `0`".

The third simplification follows the same logic as the previous one: The output value is only important when n encodes an effective Turing machine. This is what the condition "***TCHalts***(t_{TNT},n) = 0" is (supposedly) testing for. ("Supposedly" because this depends on assumptions that ultimately prove to be false). By definition, when $\boldsymbol{T}_n$ is effective, the computation $\boldsymbol{T}_n(n)$ halts. The reverse is not necessarily true though: $\boldsymbol{T}_n(n)$ might halt (for the specific value n) even if $\boldsymbol{T}_n$ is not effective. What we can say, though, is that if we ensure we have the correct (modified diagonal) output value whenever $\boldsymbol{T}_n(n)$ halts, that will certainly cover all the cases when $\boldsymbol{T}_n$ is effective… and the other cases don't matter. So we can replace the "***TCHalts***(t_{TNT},n) = 0" condition, which (supposedly) tests to see if $\boldsymbol{T}_n$ is effective (i.e. that $\boldsymbol{T}_n(x)$ halts for all $x \in \mathbb{N}$), with the much simpler condition "***TCHalts***(n,n) = 1", which only tests (supposedly) whether $\boldsymbol{T}_n(x)$ halts for $x=n$. (The new condition doesn't look any simpler written in the form above, but if we expand out the algorithm, it now avoids using Turing machine ***TNT*** altogether).

Incorporating these three simplifications into ***TDiag*** yields a new Turing machine, (call it ***TD2***) that can serve the same purpose in our proof, and can be written thus:

```
FUNCTION TD2(n)
{
  IF (TCHalts(n,n) = 1) OUTPUT U(n,n)+1
  ELSE OUTPUT 0
}
```

Expanding out the contents of ***TCHalts*** and then further incorporating the algorithms of ***U*** and ***TCNT*** directly (rather than performing abstracted "steps" of these computations until they halt) gives:

```
FUNCTION TD3(x)
{
  c ← I(x,x)
  i ← 0
  REPEAT
  {
    IF (¬H(c)) THEN c ← S(c) ELSE OUTPUT R(c)+1
    IF (F(i) ≠ "∀n∈ℕ, ¬H(S^n(I(x,x)))") THEN i ← i+1 ELSE OUTPUT 0
  }
}
```

This single Turing machine replaces (in our proof) every other Turing machine discussed up till now! Noting that "`OUTPUT z`" means "output the value z and then halt", we can interpret the function of this Turing machine as performing two separate computations "in parallel", until one of them halts. (This is actually performed by interleaving the steps of the two computations). The two computations correspond to the two "`IF`" statements inside the "`REPEAT {…}`" loop. The first computation attempts to compute and output the value of $\boldsymbol{T}_x(x)+1$. The second computation recursively enumerates all statements provable by ***F***, and outputs "0" if ***F*** can prove that the computation of $\boldsymbol{T}_x(x)$ never terminates (i.e. that the first computation never terminates – the "+1" part of it makes no difference in that regard, since the final step of adding 1 is guaranteed to terminate, if that point is reached).

The proof now runs as follows:

ASSUME that ***F*** can prove every true statement of the form "Turing computation $\boldsymbol{T}_x(x)$ never terminates" (for arbitrary x). Then within ***TD*3**, either the first computation or the second one will halt, for every argument x; i.e. ***TD*3** is effective. Now, let t_{D3} be the constant that encodes Turing machine ***TD*3**. Since ***TD*3** is effective, ***TD*3**(t_{D3}) must be halted by the completion of its *first* computation, the result of which is to output the value of ***TD*3**(t_{D3})+1. But the output value of ***TD*3**(t_{D3}) cannot equal ***TD*3**(t_{D3})+1, that's a contradiction! So the initial assumption must be false. ■

Note that in computational terms, the "contradiction" merely implies that the computation of ***TD*3**(t_{D3}) never terminates. You actually end up with an infinite recursion that can be viewed like this:

Value of ***TD*3**(t_{D3})

= (Value of ***TD*3**(t_{D3}))+1

= ((Value of ***TD*3**(t_{D3}))+1)+1

= (((Value of ***TD*3**(t_{D3}))+1)+1)+1

= ((((Value of ***TD*3**(t_{D3}))+1)+1)+1)+1

⋮

Now the proof just given demonstrates that ***F*** cannot prove every true statement of the form "Turing computation $\boldsymbol{T}_z(z)$ never terminates". Since these kind of statements are Π_1-sentences, and since the only property of ***F*** we used is that it provides a recursive enumeration of some set of true statements, we have shown (again) that the true Π_1-sentences are not r.e..

As worded above, the proof is much shorter than the version from part 1, but still doesn't give us the Gödel statement G(***F***) that we're looking for. But – and here's why using ***TD*3** is such a big advantage – we can rework it this way:

Let t_{D3} be the constant that encodes Turing machine ***TD*3**, and consider the computation of ***TD*3**(t_{D3}). Suppose it is halted via the completion of its *first* (internal) computation, the result of which is to output the value of ***TD*3**(t_{D3})+1. But the output value of ***TD*3**(t_{D3}) cannot equal ***TD*3**(t_{D3})+1, that's a contradiction! So it *cannot* halt via completion of its first internal computation. Alternatively, suppose it is halted via the completion of its *second* (internal) computation. This can only occur if ***F*** can prove that ***TD*3**(t_{D3}) does *not* halt. So either ***F*** has proved a false statement, contradicting the

assumption that ***F*** is trustworthy (proves only true statements), or else ***TD*3**(t_{D3}) does not halt via the completion of its second internal computation. Therefore ***TD*3**(t_{D3}) does not halt via the completion of its second internal computation. Therefore, ***TD*3**(t_{D3}) does not halt (at all). Furthermore, the fact that the second internal computation does not halt demonstrates that ***F*** cannot prove that ***TD*3**(t_{D3}) does not halt!

Therefore, "***TD*3**(t_{D3}) does not halt" is a true statement (a Π_1-sentence), and cannot be proved by ***F***!

So we have shown that ***F*** cannot prove all true Π_1-sentences, and we have obtained an example of a true Π_1-sentence that is not provable by ***F***. We could take this sentence to be the G(***F***) we're looking for, and we'd be finished. However, examining the last version of the proof closely, it turns out that the only role played by the "first internal computation" is that it cannot be the reason why ***TD*3**(t_{D3}) halts. Well, if that's all it's doing, we can omit it altogether, and the proof will still work!

Accordingly, the complete and maximally simplified version of the proof is as follows:

Let ***F*** be a formal proof system, and let ***F***(*n*) be a recursive enumeration of all statements provable by ***F***. Let ***P*** be the Turing machine given by the following algorithm:

```
FUNCTION P(x)
{
  i ← 0
  WHILE (F(i) ≠ "∀n∈ℕ, ¬H(S^n(I(x,x)))") DO i ← i+1
  OUTPUT 0
}
```

Here I, S, and H are the Turing machine initial state and single-step functions, and halting predicate, respectively, as defined earlier. ***P*** is constructed such that ***P***(*x*) halts iff ***F*** can prove that the computation $\boldsymbol{T}_x(x)$ does not halt. Now let *p* be the constant that encodes ***P***, i.e. ***P*** ≡ $\boldsymbol{T}_p$, and consider whether or not the computation of ***P***(*p*) ≡ $\boldsymbol{T}_p(p)$ will halt. By the construction of ***P***, it will halt iff ***F*** can prove that it won't. Therefore, it will only halt if ***F*** can prove a falsehood. Otherwise, it will not halt, and furthermore ***F*** cannot prove that it will not halt.

Therefore, if ***F*** is trustworthy (proves only true statements), then ***F*** cannot prove *all* true mathematical statements, and in particular ***F*** cannot prove the Π_1-sentence G(***F***):

$$\text{G}(\boldsymbol{F}) \;=\; \text{“}\forall n\in\mathbb{N},\, \neg H(S^n(I(p,p)))\text{”}$$

G(***F***) is expressible in English as "Turing machine computation $\boldsymbol{T}_p(p)$ does not halt". Note that this depends on ***F*** because ***P***'s algorithm includes the evaluation of ***F***(*i*), and the details of ***P*** determine *p*.

■

Addendum:

We arrived at the final proof (the one on the bottom half of the last page) only after a long and circuitous route that began with the Turing/Cantor argument given in the text – a proof that the set of *effective* Turing machines is not recursively enumerable. There is *another* kind of Turing/Cantor argument that we can make, that leads to the final proof much more directly; almost immediately, in fact. Probably it's what Penrose had in mind when he refers to "Turing's argument above", though it actually isn't the same as the "argument above" in the book. It is as follows:

Let us denote the result of each Turing machine computation by its numerical output, if it terminates, and by 'N' if it does not. Then we can imagine constructing a table containing the results of *all* Turing machine computations, by listing all the Turing machines ($\boldsymbol{T}_0$, $\boldsymbol{T}_1$, $\boldsymbol{T}_2$, $\boldsymbol{T}_3$, …etc.) along the left hand side, and all possible arguments (0, 1, 2, 3, …etc.) along the top. (We may need to apply some convention to handle numbers that don't encode valid Turing machines, if our encoding method admits such: Just replace invalid entries with some 'default' Turing machine). Now apply Cantor's diagonal slash to this table: Attempt to construct a Turing machine whose result, for every argument x, differs from that of computation $\boldsymbol{T}_x(x)$ – either by producing a different numerical output, or because the result of one is 'N' and the result of the other is not. Such a Turing machine, if it were constructible, would differ from every Turing machine listed along the left hand side of the table. Therefore, *either* this list does not include every possible Turing machine, *or else* we cannot construct a Turing machine to function as described. But we know that (given an adequate coding scheme) we can encode *every* Turing machine as $\boldsymbol{T}_n$ for some value $n \in \mathbb{N}$., and that therefore the list *does* contain every possible Turing machine. So it must be the second alternative that holds: We cannot construct a Turing machine to function as described.

(Note that this argument differs from other versions of Cantor's diagonal slash that we've seen in the book so far, in that it proves that the "diagonal construction" is impossible, rather than proving that the list is incomplete).

To actually construct this impossible "diagonal" Turing machine, we'd need an algorithm for determining when the result of a computation $\boldsymbol{T}_x(x)$ was 'N' (never terminates). Since this result is equivalent to the mathematical Π_1-sentence "$\forall n \in \mathbb{N}, \neg H(S^n(I(p,p)))$", we can show that a recursive enumeration $\boldsymbol{F}(n)$ of all true Π_1-sentences would allow us to construct this (impossible) Turing machine, thus demonstrating that no such recursive enumeration is possible.

If we assume we *do* have such an $\boldsymbol{F}(n)$, and decide to construct our Turing machine to output '0' when $\boldsymbol{T}_x(x)$ results in 'N', and $\boldsymbol{T}_x(x)+1$ otherwise, then ***TD*3** (or something similar) would be the machine we construct. If instead we choose to output '0' when $\boldsymbol{T}_x(x)$ results in 'N', and to result in 'N' otherwise, then ***P*** (or similar) would be the end product of our construction.

Having thus arrived at these Turing machine constructions, it becomes very obvious where the final proof, involving ***P***, comes from (and ditto for the preceding alternative version involving ***TD*3**).