LOGIC I

VICTORIA GITMAN

1. The Completeness Theorem

The Completeness Theorem was proved by Kurt Gödel in 1929. To state the theorem we must formally define the notion of proof. This is not because it is good to give formal proofs, but rather so that we can prove mathematical theorems about the concept of proof.
–**Arnold Miller**

1.1. **On consequences and proofs.** Suppose that T is some first-order theory. What are the *consequences* of T? The obvious answer is that they are statements provable from T (supposing for a second that we know what that means). But there is another possibility. The consequences of T could mean statements that hold true in every model of T. Do the proof theoretic and the model theoretic notions of consequence coincide? Once, we formally define proofs, it will be obvious, by the definition of truth, that a statement that is provable from T must hold in every model of T. Does the converse hold? The question was posed in the 1920's by David Hilbert (of the 23 problems fame). The answer is that remarkably, yes, it does! This result, known as the Completeness Theorem for first-order logic, was proved by Kurt Gödel in 1929. According to the Completeness Theorem provability and semantic truth are indeed two very different aspects of the same phenomena.

In order to prove the Completeness Theorem, we first need a formal notion of proof. As mathematicians, we all know that a proof is a series of deductions, where each statement proceeds by logical reasoning from the previous ones. But what do we start with? Axioms! There are two types of axioms. First, there are the axioms of logic that are same for every subject of mathematics, and then there are the axioms particular to a given subject. What is a deduction? Modus ponens.

The first-order theory we are working over is precisely what corresponds to the axioms of a given subject. For the purposes of this section, we shall extend the notion of a theory from a collection of sentences (formulas without free variables) to a collection of formulas (possibly with free variables). What is a model for a collection of formulas? We shall say that a pair $\langle M, i\rangle$, where M is a structure of the language of T and i is a map from the variables $\{x_1, \ldots, x_n, \ldots\}$ into M, is a model of T if M is a model of T under the interpretation i of the free variables.[1]

Example 1.1. Let L be the language $\{<\}$ and T consist of a single formula $\varphi := x_1 < x_2$. Then $\mathbb{N}$ is an L-structure with the natural interpretation for $<$. Let $i(x_i) = i - 1$. Then $\langle \mathbb{N}, i\rangle$ is a model of T. On the other hand, if we define j such that $j(x_1) = 2$ and $j(x_2) = 0$, then the pair $\langle \mathbb{N}, j\rangle$ is not a model of T.

Next, we must decide on what are the axioms of logic. The naive approach would be to say that an axiom of logic is any logical validity: a formula that is true in all models under all interpretations. This is what Arnold Miller calls the *Mickey Mouse* proof system and it would be the first step toward equating the two notions

[1] A good source for material in this section is [AZ97].

of consequence.[2] The Mickey Mouse proof system surely captures everything we would ever need, but how do we check whether a given statement in a proof belongs to this category? This question, known as the Entscheidungsproblem, was posed by Hilbert and his student Wilhelm Ackermann. It was answered by the great Alan Turing, who showed that, indeed, there is no 'reasonable' way of checking whether a formula is a logical validity (we will prove this in Section 7). Since being able to verify a claim to be a proof is a non-negotiable requirement, we want a different approach. What we want is a sub-collection of the logical validities that we can actually describe, thus making it possible to check whether a statement in the proof belongs to this collection, but from which all other logical validities can be deduced. Here is our attempt:

Fix a first-order language L. First let's recall the following definition. We define the *terms* of L inductively as follows. Every variable x_n and constant c is a term. If $t_1,\ldots,t_n$ are terms and $f(x_1,\ldots,x_n)$ is a function in L, then $f(t_1,\ldots,t_n)$ is a term. Next, we define that the *universal closure* with respect to the variables $\{x_1,\ldots,x_n\}$ of a formula φ is the formula $\forall x_1,\ldots,x_n\varphi$.

The axioms of logic for L are the *closures* of the following formula schemes[3] with respect to all variables:

1. **Axioms of Propositional Logic.**
   For any formulas φ, ψ, θ:
$$\varphi\to(\psi\to\varphi)$$
$$[\varphi\to(\psi\to\theta)]\to[(\varphi\to\psi)\to(\varphi\to\theta)]$$
$$(\neg\psi\to\neg\varphi)\to(\varphi\to\psi)$$
2. **Axioms of equality.**
   For any terms t_1, t_2, t_3:
$$t_1=t_1$$
$$t_1=t_2\to t_2=t_1$$
$$t_1=t_2\to(t_2=t_3\to t_1=t_3)$$
   For every relation symbol r and every function symbol f,
$$t_1=s_1\wedge\cdots\wedge t_n=s_n\to(r(t_1,\ldots,t_n)\to r(s_1,\ldots,s_n))$$
$$t_1=s_1\wedge\cdots\wedge t_m=s_m\to f(t_1,\ldots,t_m)=f(s_1,\ldots,s_m)$$
3. **Substitution axioms.**
   For any formula φ, variable x and term t such that the substitution $\varphi(t/x)$ is proper[4]:
$$\forall x\varphi\to\varphi(t/x)$$
4. **Axioms of distributivity of a quantifier.**
   For any formulas φ, ψ, and any variable x:
$$\forall x(\varphi\to\psi)\to(\forall x\varphi\to\forall x\psi)$$

[2]One surprising logical validity is $(\varphi\to\psi)\vee(\psi\to\varphi)$. Either φ implies ψ or ψ implies φ!

[3]A *formula scheme* is a (possibly infinite) collection of formulas matching a given pattern.

[4]A substitution of a term t for a variable x in a formula φ is proper if 'you are not affecting bound variables'. More precisely, if φ is an atomic formula, then any substitution is proper; if $\varphi=\neg\psi$, then a substitution is proper for φ if and only if it is proper for ψ; if $\varphi=\psi\wedge\theta$, then a substitution is proper for φ if and only if it is proper for both ψ and θ; finally, if $\varphi=\forall y\psi$, then a substitution is proper if $x\neq y$, y does not appear in t, and the substitution is proper for ψ.

5. **Adding a redundant quantifier.**
   For any formula φ and any x not occurring in φ:
$$\varphi \to \forall x \varphi$$

Let us call LOG_L the collection of axioms of logic for L. The following result is clear.

Theorem 1.2. *Every L-structure is a model of* LOG_L.

We are now ready to formally define the notions of *proof* and *theorem* of an L-theory T.

A *proof* p from T is a finite sequence of formulas $\langle \varphi_1, \ldots, \varphi_n \rangle$ such that $\varphi_i \in$ LOG, or $\varphi_i \in T$ or there are $j, k < i$ such that $\varphi_j = \varphi_k \to \varphi_i$ (φ_i is derived by modus ponens). If there is a proof of a formula φ from T, we shall denote this by $T \vdash \varphi$. The *theorems* of T is the smallest set of formulas containing $T \cup \mathrm{LOG}$ and closed under modus ponens. Equivalently, the set T^* of theorems of T is defined by induction as
$$T^* = \bigcup_n T_n,$$
where
$$T_0 = T \cup \mathrm{LOG},$$
and
$$T_{n+1} = T_n \cup \{\psi \mid \exists \varphi(\varphi \in T_n \text{ and } \varphi \to \psi \in T_n)\}.$$
Whenever we will want to conclude a statement about all theorems of T, we shall argue by induction on the theorems of T: we will show that it is true of formulas in T, of logical axioms, and whenever it is true of φ and $\varphi \to \psi$, then it is true of ψ.

Theorem 1.3 (Soundness)**.** *Suppose that T is an L-theory. If $T \vdash \varphi$, then every model of T satisfies φ.*

In the next section, we begin to build up the machinery necessary to prove, the converse, the Completeness Theorem. Once in possession of the Completeness Theorem, we will be able to forget all about provability, which we shall soon see is a very desirable thing to do! But before we are allowed to forget provability...

1.2. **Formal provability.** Here are some examples of proofs, of analyzing provability, and of results we will need to prove the Completeness Theorem. In the background of all these results is some first-order language L.

Theorem 1.4. *For every formula φ, we have $\vdash \varphi \to \varphi$.*

Proof. The formulas:
$$\psi_1 := \varphi \to [(\varphi \to \varphi) \to \varphi]$$
$$\psi_2 := \{\varphi \to [(\varphi \to \varphi) \to \varphi]\} \to \{[\varphi \to (\varphi \to \varphi)] \to (\varphi \to \varphi)\}$$
are in LOG_L. Applying modus ponens, we get:
$$\psi_3 := [\varphi \to (\varphi \to \varphi)] \to (\varphi \to \varphi)$$
Also,
$$\psi_4 := \varphi \to (\varphi \to \varphi)$$
is in LOG_L, and so applying modus ponens, we derive $\varphi \to \varphi$. Thus, the sequence
$$\langle \psi_1, \psi_2, \psi_3, \psi_4, \varphi \to \varphi \rangle$$

is a proof of the formula $\varphi \to \varphi$. □

Theorem 1.5 (Deduction theorem). *For every theory T and formula φ, we have $T, \varphi \vdash \psi$ if and only if $T \vdash \varphi \to \psi$.*

Proof. If $T \vdash \varphi \to \psi$, then it is clear that $T, \varphi \vdash \psi$ from the definition of provability. For the other direction, we shall argue that $T, \varphi \vdash \psi$ implies that $T \vdash \varphi \to \psi$ by induction on the theorems ψ of $T \cup \{\varphi\}$. If $\psi \in T \cup \mathrm{LOG}_L$, then $T \vdash \psi$ and also $\psi \to (\varphi \to \psi)$ is an axiom of logic. Thus, $T \vdash \varphi \to \psi$ by modus ponens. If $\psi = \varphi$, then by Theorem 1.4, we have $T \vdash \varphi \to \varphi$. Finally, suppose that the conclusion holds for theorems θ and $\theta \to \psi$. Thus, we have

$$T \vdash \varphi \to \theta \text{ and } T \vdash \varphi \to (\theta \to \psi)$$

Now we apply the axiom of logic $[\varphi \to (\theta \to \psi)] \to [(\varphi \to \theta) \to (\varphi \to \psi)]$ and modus ponens twice. □

The next few theorem will be stated without proofs because the proofs get more tedious by the theorem. For those who would like to inflict this pain on themselves please consult [AZ97] for details. We will need these results for the proof of the Completeness Theorem.

We shall say that a theory T is *consistent* if for no statement φ do we have that $T \vdash \varphi$ and $T \vdash \neg\varphi$. Otherwise, we shall say that T is *inconsistent*.

Theorem 1.6. *If T is inconsistent, then $T \vdash \psi$ for every formula ψ.*

Theorem 1.7 (Reductio ad absurdum). *If $T \cup \{\neg\varphi\}$ is inconsistent, then $T \vdash \varphi$.*

We shall say that a theory T is *complete* if for every formula φ, we have either $T \vdash \varphi$ or $T \vdash \neg\varphi$. We show below, using Zorn's Lemma, that every consistent theory can be extended to a complete consistent theory. Recall that Zorn's Lemma states that whenever you have a nonempty partially ordered set such that every linearly ordered subset has an upper bound, then there is a maximal element. Zorn's Lemma (1935) is equivalent to the Axiom of Choice.

Theorem 1.8. *Every consistent theory T has a complete consistent extension T^*.*

Proof. Let $\mathbb{P}$ be the partially ordered set consisting of all consistent extensions of T ordered by inclusion, so that if $T_1, T_2 \in \mathbb{P}$, then $T_1 \leq T_2$ if and only if $T_1 \subseteq T_2$. Clearly, $\mathbb{P}$ is nonempty since $T \in \mathbb{P}$. If $X \subseteq \mathbb{P}$ is linearly ordered, then $U = \bigcup\{S \mid S \in X\}$ is an extension of T. We shall argue that it is consistent and hence in $\mathbb{P}$. If U was inconsistent, then there would be an inconsistent $\{\varphi_1, \ldots, \varphi_n\} \subseteq U$. Let each $\varphi_i \in S_i$ in X and let S_j be the largest of the S_i by linearity. Then all $\varphi_i \in S_j$ which contradicts that it is consistent. Thus, U is consistent and hence in $\mathbb{P}$. By Zorn's Lemma, we let $T^* \in \mathbb{P}$ be some maximal element. By the Reductio ad absurdum theorem, it follows that T^* must be complete. □

The usefulness of the next series of theorems will not become apparent until we get to the proof of the Completeness Theorem.

Theorem 1.9 (The generalization rule). *If a variable x does not occur in any formula $\varphi \in T$, then $T \vdash \varphi$ implies $T \vdash \forall x\varphi$.*

Proof. We shall argue by induction on the theorems φ of T. If $\varphi \in \mathrm{LOG}_L$, then $\forall x\varphi \in \mathrm{LOG}_L$ since we assumed that all universal closures of formulas in LOG_L

are also in LOG_L. If $\varphi \in T$, then x does not occur in φ by assumption and so $\varphi \to \forall x\varphi$ is an axiom of logic. Applying modus ponens, it follows that $T \vdash \forall x\varphi$. Now we suppose that the assumption holds for $\psi \to \varphi$ and ψ: $T \vdash \forall x\psi$ and $T \vdash \forall x(\psi \to \varphi)$. The formula $\forall x(\psi \to \varphi) \to (\forall x\psi \to \forall x\varphi)$ is an axiom of logic and so $\forall x\varphi$ now follows by modus ponens. □

Theorem 1.10. *Suppose that T is a theory and that a constant c does not occur in any formula of T. If $T \vdash \varphi$, then $T \vdash \forall y\varphi(y/c)$ for some variable y, and moreover, there is a proof from T of the formula $\forall y\varphi(y/c)$ in which the constant c does not occur.*

Sketch of proof. We fix a proof of φ, choose a variable y not occurring in the proof and argue that the sequence which results by replacing every occurrence of c by y is a proof of $\varphi(y/c)$. Now use the generalization rule. □

Corollary 1.11 (Elimination of constants)**.** *Suppose T is a theory and a constant c does not occur in any formula of T. If c does not occur in φ, then $T \vdash \varphi(c/x)$ implies that $T \vdash \forall x\varphi$.*

Sketch of proof. By Theorem 1.10, we have $T \vdash \forall y\varphi(y/x)$ for some variable y not occurring in φ. Now argue that $\forall y\varphi(y/x)$ and $\forall x\varphi$ are logically equivalent (this is obvious but tedious to argue). □

Corollary 1.12. *Suppose T is a theory in a language L, L^* is some expansion of L by adding constants, and φ is a formula in L. If $T \vdash \varphi$ in L^*, then $T \vdash \varphi$ in L.*

Proof. Let $c_1, \dots, c_m$ be all new constants occurring in the proof of φ from T in L^*. By the proof of Theorem 1.10, there is a proof of $\forall y_1, \dots, y_m\varphi(y_1/c_1, \dots, y_m/c_m)$, for some variables y_i not occurring in φ, from T in L. Since φ is a formula of L, we have that $\varphi(y_1/c_1, \dots, y_m/c_m)$ is the same as φ. Thus, $T \vdash \forall y_1, \dots, y_m\varphi$ in L. Now since the y_i do not appear in φ, we use the axiom of logic $\forall x\psi \to \psi(t/x)$ with any term t to conclude that $T \vdash \varphi$. □

Corollary 1.13. *Suppose L^* is an expansion of the language L by adding constants. If T is consistent in L, then T is consistent in L^*.*

Proof. If T were inconsistent in L^*, then for any formula φ of L, we would have proofs of φ and $\neg\varphi$ from T in L^* by Theorem 1.6. But then by Corollary 1.12, there would be proofs of φ and $\neg\varphi$ in L. □

1.3. **The Completeness Theorem.** We are now in possession of all the machinery needed to prove the Completeness Theorem. The proof we will give is due to Leon Henkin (1949). Henkin's construction has many applications, some of which we will encounter later on in the course.

Theorem 1.14. *Every consistent theory has a model.*

Corollary 1.15 (The Completeness Theorem)**.** *If T is a theory and every model of T satisfies φ, then $T \vdash \varphi$.*

Proof. If T does not prove φ, then $T \cup \{\neg\varphi\}$ is consistent by Reductio ad absurdum theorem. □

Our proof strategy will consist of two parts:
Part 1: We shall say that a theory S in a language L has the *Henkin property* if whenever φ is a formula and x is a variable, then for some constant $c \in L$, the theory S contains the formula:

$$\psi_{\varphi,x} := \exists x\varphi \to \varphi(c/x).$$

That is, every existential formula is witnessed by a constant in the theory S! We will show that every complete consistent theory with the Henkin property has a model. Intuitively, since every existential formula is witnessed by some constant, we can hope to build the model out of the constants of L.
Part 2: We will show that every consistent theory T in a language L can be extended to a complete consistent theory T^* with the Henkin property in some extended language L^*.

Now for the details.

Proof of Theorem 1.14.
Part 1: Suppose that a theory S in a language L is consistent, complete and has the Henkin property. We will assume for ease of presentation that S contains only sentences. We can also assume without loss of generality that for every φ, we have $\varphi \in S$ or $\neg\varphi \in S$, by extending S to contain all its consequences. Let X consist of all constant symbols in L. As we mentioned earlier, it seems reasonable to try to build the requisite model out of X. We cannot quite let X be the universe of our proposed model because constants do not necessarily represent distinct elements. It could be that the sentence $c = d$ is in S for some distinct constants c and d in L, and so they should represent the same element of a model of S. We can easily overcome this obstacle though by accordingly defining an equivalence relation on the constants and letting our model consist of the equivalence classes. We define the relation $\sim$ on X by $c \sim d$ if and only if the sentence $c = d$ is in S. Using the equality axioms of logic and the fact that S is both consistent and complete, it immediately follows that $\sim$ is an equivalence relation. Thus, we let $M = X/\sim$ be the set of all equivalence classes.

For a relation $r(x_1, \ldots, x_n)$ of L, we define that $M \models r([c_1], \ldots, [c_n])$ if $r(c_1, \ldots, c_n) \in S$. It follows from the equality axioms and the completeness and consistency of S that this is well-defined. For a function $f(x_1, \ldots, x_m)$ of L, we define that $M \models f([c_1], \ldots, [c_m]) = [c]$ if $f(c_1, \ldots, c_m) = c \in S$. This is again clearly well-defined, but we also have to argue that such a constant c exists. For some constant c, the sentence

$$\exists x f(c_1, \ldots, c_m) = x \to f(c_1, \ldots, c_m) = c$$

is in S by the Henkin property, and so it suffices to argue that $\exists x f(c_1, \ldots, c_m) = x$ is in S as well. If it were not, then by completeness of S, we would have $\forall x \neg f(c_1, \ldots, c_m) = x$ in S. But this cannot be since

$$\forall x \neg f(c_1, \ldots, c_m) = x \to \neg f(c_1, \ldots, c_m) = f(c_1, \ldots, c_m)$$

is an axiom of logic, as well as

$$f(c_1, \ldots, c_m) = f(c_1, \ldots, c_m).$$

Thus, our satisfaction definition for functions makes sense. Finally, we define that $M \models c = [c]$.

Now that we have a proposed model M, we will argue that $M \models \varphi(c_1, \ldots, c_n)$ if and only if $\varphi(c_1, \ldots, c_n)$ is in S. We start by showing, by induction on complexity

of terms t, that we have $M \models t = c$ if and only if $t = c$ is in S. So consider a term $t = f(t_1, \ldots, t_n)$. By the same argument as above, there is a constant c and constants $c_1, \ldots, c_n$ such that $f(t_1, \ldots, t_n) = c$ and $t_i = c_i$ are all in S. By the inductive assumption, we have that $M \models t_i = c_i$ and $M \models f(c_1, \ldots, c_n) = c$. It follows that $M \models f(t_1, \ldots, t_n) = d$ if and only if $M \models c = d$ if and only if $f(t_1, \ldots, t_n) = d$ is in S. Next, we argue that for atomic formulas φ, we have $M \models \varphi$ if and only if $\varphi \in S$. Suppose φ has the form $r(t_1, \ldots, t_n)$ for some terms t_i. By what we already showed, there are constants c_i such that $t_i = c_i$ are all in S and $M \models t_i = c_i$. Thus, $M \models r(t_1, \ldots, t_n)$ if and only if $M \models r(c_1, \ldots, c_n)$ if and only if $r(c_1, \ldots, c_n)$ is in S if and only if $r(t_1, \ldots, t_n)$ is in S. Suppose next that φ has the form $t = s$. Again, there are constants c, d such that $t = c$ and $s = d$ are in S and $M \models t = c, s = d$. Thus, $M \models t = s$ if and only if $M \models c = d$ if and only if $c = d$ is in S if and only if $t = s$ is in S. This completes the argument for atomic formulas. Now suppose inductively that $M \models \varphi, \psi$ if and only $\varphi, \psi \in S$. Thus, we have that $M \models \neg\varphi$ if and only if M is not a model of φ if and only if $\varphi \notin S$ if and only if $\neg\varphi \in S$. We also have that $M \models \varphi \wedge \psi$ if and only if $M \models \varphi$ and $M \models \psi$ if and only if φ and ψ are in S if and only if $\varphi \wedge \psi$ is in S. The most involved argument is the quantifier case. So suppose that $M \models \exists x\varphi(x)$. It follows that $M \models \varphi(c)$ for some constant c and hence by the inductive assumption, we have $\varphi(c)$ in S. If $\exists x\varphi(x)$ were not in S, then $\forall x\neg\varphi(x)$ would be in S by completeness, and thus so would $\neg\varphi(c) \in S$, meaning that S is inconsistent. Thus, $\exists x\varphi(x) \in S$. Now suppose that $\exists x\varphi(x)$ is in S. By the Henkin property, it follows that $\exists x\varphi \to \varphi(c/x)$ is in S for some constant c. Thus, by completeness, we have $\varphi(c)$ is in S and hence by the inductive assumption, $M \models \varphi(c)$, which implies that $M \models \exists x\varphi(x)$. This completes the proof that M is a model of S.

To modify the proof for the case of formulas, we would need to provide an interpretation of the variables x_n by the elements of M. For each variable x_i, there is a constant c such that $x_i = c$ is in S. So we interpret x_i by $[c]$ in M.

Part 2: It remains to show that any consistent theory T in a language L can be extended to a theory T^* in some language L^* such that T^* is a complete consistent theory with the Henkin property in L^*. Let $T = T_0$ and $L = L_0$. Let the language L_1 be the expansion of L_0 by adding constants $c_{\varphi,x}$ for every formula φ in L_0 and every variable x. Let T_1' consist of T_0 together with the formulas

$$\psi_{\varphi,x} := \exists x\varphi \to \varphi(c_{\varphi,x}/x)$$

for every formula φ of L_0 and every variable x. Let us argue that theory T_1' is consistent in L_1. Enumerate the formulas $\psi_{\varphi,x}$ as $\{\psi_1, \psi_2, \ldots, \psi_n, \ldots\}$ and let $S_m = T \cup \{\psi_1, \ldots, \psi_m\}$. Clearly it suffices to show that all S_m are consistent in L_1. The theory $S_0 = T$ is consistent in the language L_0 and hence in L_1 as well by Corollary 1.13. So we assume that S_m is consistent for some m. Then $S_{m+1} = S_m \cup \{\psi_{m+1}\}$, where

$$\psi_{m+1} := \exists x\varphi \to \varphi(c/x).$$

If S_{m+1} were inconsistent, then we would have $S_m \vdash \neg(\exists x\varphi \to \varphi(c/x))$ by Reductio ad absurdum. Thus, $S_m \vdash \exists x\varphi \wedge \neg\varphi(c/x)$. But since c does not appear in S_m, it follows that $S_m \vdash \forall x\neg\varphi$, meaning that S_m was inconsistent, a contradiction. This completes the proof that T_1' is consistent. Next, we extend T_1' to a complete consistent theory T_1 in L_1 by Theorem 1.8. Now inductively, we define T_n and L_n

as above and let

$$T^* = \bigcup_n T_n \text{ and } L^* = \bigcup_n L_n.$$

It should be clear that T^* is a complete consistent theory in L^* with the Henkin property. □

The Compactness Theorem, due to Gödel (1930) for countable theories and to Malcev (1936) for uncountable theories, is an immediate corollary of the Completeness Theorem.

Corollary 1.16 (Compactness Theorem)**.** *If every finite subset of a theory T has a model, then T has a model.*

Proof. If every finite subset of a theory T has a model, then it is consistent, but then it has a model by the Completeness Theorem! □

Giving our thanks to the Completeness Theorem, we will endeavor, for the remainder of course, to forget all about formal provability. In the future, whenever we will be required to show that a theory $T \vdash \varphi$, we will argue that every model of T is a model of φ.

1.4. **Homework.**

Sources

(1) Arnold Miller's Logic Course Notes [Mil]
(2) Logic of Mathematics [AZ97]

Question 1.1. Prove that if $T, \neg G \vdash \neg F$, then $T \vdash F \to G$.

Question 1.2. Suppose we adopt the Mickey Mouse proof system (where the axioms of logic are all logical validities). Under this assumption, show that the Completeness Theorem follows from the Compactness Theorem.

Question 1.3. The alphabet of propositional logic consists of a countable set of propositional variables $\mathcal{P} = \{P_0, P_1, \ldots, P_n, \ldots\}$, the logical connectives $\wedge$, $\vee$, $\neg$, and $\to$ and grammar symbols (,). The propositional formulas are defined inductively in the same manner as those of first-order logic. The semantics of propositional logic is a truth valuation function $v : \mathcal{I} \to \{T, F\}$ assigning truth values to all propositional variables. The truth valuation extends to all formulas as in first-order logic. A collection of formulas Σ of propositional logic is said to be realizable if there is a truth valuation such that all formulas in Σ evaluate to T. A collection of formulas is said to be finitely realizable if every finite sub-collection of it is realizable. Prove the Compactness Theorem for propositional logic.

2. Models of Peano Arithmetic

"...it's turtles all the way down."

2.1. **Peano's Axioms.** In 1889, more than two millennia after ancient Greeks initiated a rigorous study of number theory, Guiseppe Peano introduced the first axiomatization for the theory of the natural numbers. Incidentally, Peano is also famous for his space filling curve, a continuous surjection from the unit interval onto the unit square. Peano originally formulated his axioms in second-order logic, but in the next century, Peano's axioms were reformulated and studied in the context of first-order logic, the newly accepted language of formal mathematics. Proving the

consistency of Peano Arithmetic (PA), as the first-order reformulation of Peano's axioms become known, by 'finitary' means was the second of Hilbert's famous 23 problems for the 20th-century.[5]

The language of PA is $L_A = \{+, \cdot, <, 0, 1\}$, where $+, \cdot$ are 2-ary functions, $<$ is a 2-ary relation, and 0, 1 are constant symbols.

Here is the theory *Peano Arithmetic*.[6]

Ax1-Ax5 express the associativity and commutativity of $+$, $\cdot$ and the distributive law.

Ax1: $\forall x, y, z((x + y) + z = x + (y + z))$
Ax2: $\forall x, y(x + y = y + x)$
Ax3: $\forall x, y, z((x \cdot y) \cdot z = x \cdot (y \cdot z))$
Ax4: $\forall x, y(x \cdot y = y \cdot x)$
Ax5: $\forall x, y, z(x \cdot (y + z) = x \cdot y + x \cdot z)$

The group Ax6-Ax7 expresses that 0 is the identity for $+$ and zero for $\cdot$, and 1 is the identity for $\cdot$.

Ax6: $\forall x(x + 0 = x) \wedge (x \cdot 0 = 0)$
Ax7: $\forall x(x \cdot 1 = x)$

The group Ax8-Ax10 expresses that $\leq$ $(x \leq y \leftrightarrow x < y \vee x = y)$ is a linear order.

Ax8: $\forall x, y, z((x < y \wedge y < z) \rightarrow x < z)$
Ax9: $\forall x \, \neg x < x$
Ax10: $\forall x, y(x < y \vee x = y \vee y < x)$

The group Ax11-Ax12 expresses that the functions $+$ and $\cdot$ respect the order $<$.

Ax11: $\forall x, y, z(x < y \rightarrow x + z < y + z)$
Ax12: $\forall x, y, z(0 < z \wedge x < y \rightarrow x \cdot z < y \cdot z)$

The axiom Ax13 expresses that we can define subtraction.

Ax13: $\forall x, y(x < y \rightarrow \exists z \, x + z = y)$

The group Ax14-Ax15 expresses that 0 is the least element and the ordering $<$ is discrete.

Ax14: $0 < 1 \wedge \forall x(x > 0 \rightarrow x \geq 1)$
Ax15: $\forall x(x \geq 0)$

So far we have written down finitely many axioms, 15 to be precise. The theory consisting of these is known as PA^-. Now we will write down the axiom scheme

[5]A discussion of the meaning of 'finitary' consistency proofs will have to wait until Section 5.
[6]A good source for material in this section is [Kay91].

expressing induction. If $\varphi(x,\overline{y})$ is an L_A-formula[7], then the axiom of induction for φ on x is:

$$\forall \overline{y}(\varphi(0,\overline{y}) \wedge \forall x(\varphi(x,\overline{y}) \rightarrow \varphi(x+1,\overline{y})) \rightarrow \forall x \varphi(x,\overline{y})).$$

Can every property of the natural numbers be proved from the PA?

Question 2.1. Is Peano Arithmetic complete?

Gödel showed in his famous First Incompleteness Theorem (1931) that Peano Arithmetic is not complete, albeit through no fault of its own. Indeed, the First Incompleteness Theorem shows that no 'reasonably defined' theory extending Peano Arithmetic could ever be complete! We will learn all about the First Incompleteness Theorem in Section 4.

By design, the natural numbers $(\mathbb{N},+,\cdot,0,1)$ is a (countable) model of PA. It is known as the *standard model* of PA.

Question 2.2. Are there countable *nonstandard models* of PA? What do they look like?

We will find out more about this in the next few sections.

Question 2.3. Are there uncountable models of PA?

Yes, surprising as it may seem, there are uncountable models of our familiar number theory! Here is one way to argue it. Let us expand L_A to the language L_A' by adding uncountably many new constants. Next, let us extend PA to the theory PA$'$ in L_A' by adding the sentences $c \neq d$ for every two new distinct constants $c, d \in L_A'$. Clearly PA$'$ is finitely realizable (in $\mathbb{N}$) and thus has a model by the Compactness Theorem and clearly this model must be uncountable.

2.2. **The Arithmetical Hierarchy.** Before we dive deeper into the world of models of PA, we will introduce a structure on L_A-formulas known as the *arithmetical hierarchy*.

Let $\forall x < y(\ldots)$ be an abbreviation for the L_A formula $\forall x(x < y \rightarrow \ldots)$ and $\exists x < y(\ldots)$ be an abbreviation for the L_A-formula $\exists x(x < y \wedge \ldots)$. We will say that such quantifiers are *bounded*. An L_A-formula is said to be Δ_0 if all its quantifiers are bounded. The Δ_0-formulas are at the bottom of the arithmetical hierarchy. For convenience, we shall also call Δ_0-formulas Σ_0-formulas and Π_0-formulas. Now we define inductively that an L_A-formula is Σ_{n+1} if it has the form $\exists \overline{y}\varphi$ with φ a Π_n-formula. A Σ_n-formula looks like

$$\exists \overline{x}_1 \forall \overline{x}_2 \exists \overline{x}_3 \ldots Q\overline{x}_n \underbrace{\varphi(\overline{x}_1,\ldots,\overline{x}_n,\overline{y})}_{\Delta_0}.$$

We further define that a formula is Π_{n+1} if it has the form $\forall \overline{y}\varphi$ with φ a Σ_n-formula. A Π_n-formula looks like

$$\forall \overline{x}_1 \exists \overline{x}_2 \forall \overline{x}_3 \ldots Q\overline{x}_n \underbrace{\varphi(\overline{x}_1,\ldots,\overline{x}_n,\overline{y})}_{\Delta_0}.$$

We allow blocks of quantifiers to be empty, so that every Σ_n-formula is both Σ_m and Π_m for every $m \geq n+1$. In this way, we obtain the arithmetical hierarchy of formulas.

[7] The notation $\overline{y}$ is a shorthand for a tuple of elements $y_1,\ldots,y_n$.

If T is an L_A-theory, then a formula is said to be $\Sigma_n(T)$ or $\Pi_n(T)$ if it is equivalent over T to a Σ_n-formula or a Π_n-formula respectively. A formula is said to be $\Delta_n(T)$ if it is both $\Sigma_n(T)$ and $\Pi_n(T)$. Similarly, if M is an L_A-structure, then a formula is said to be $\Sigma_n(M)$ or $\Pi_n(M)$ if it is equivalent over M to such a formula and it is said to be $\Delta_n(M)$ if it is both $\Sigma_n(M)$ and $\Pi_n(M)$.

Over PA, we can show that a Σ_n-formula that is followed by a bounded quantifier is equivalent to a Σ_n-formula, that is, we can 'push the bounded quantifier inside' and similarly for Π_n-formulas.

Theorem 2.4. *Suppose that $\varphi(\overline{y}, x, z)$ is a Σ_n-formula and $\psi(\overline{y}, x, z)$ is a Π_n-formula. Then the formulas $\forall x < z\, \varphi(\overline{y}, x, z)$ and $\exists x < z\, \psi(\overline{y}, x, z)$ are $\Sigma_n(\text{PA})$ and $\Pi_n(\text{PA})$ respectively.*

The proof is left as homework.

2.3. **What** PA **proves.** Let's prove a few basic theorems of PA. For the future, your general intuition should be that pretty much anything you are likely to encounter in a number theory course is a theorem of PA. All our 'proofs' from PA^- will be carried out model theoretically, using the Completeness Theorem. First, we get some really, really basic theorems out of the way.

Theorem 2.5. PA^- *proves that the ordering $<$ is discrete:* $\text{PA}^- \vdash \forall x, y(x < y \to x + 1 \leq y)$.

Proof. Suppose that $M \models \text{PA}^-$ and $a < b$ in M. By Ax13, there is z such that $a + z = b$ and by Ax9, $\neg a = b$. Thus, $z \neq 0$, and so by Ax14, Ax15, we have $z \geq 1$. Now finally, by Ax11, we have $a + 1 \leq a + z = b$. □

Let us call $\underline{0}$ the constant term 0 and $\underline{n}$ the term $\underbrace{1 + \cdots + 1}_{n \text{ times}}$ (by Ax1, we don't have to worry about parenthesization). We think of the terms $\underline{n}$ as representing the natural numbers in a model of PA^-. Indeed, we show below that the terms $\underline{n}$ behave exactly as though they were natural numbers.

Theorem 2.6. *If $n, m, l \in \mathbb{N}$ and $n + m = l$, then* $\text{PA}^- \vdash \underline{n} + \underline{m} = \underline{l}$.

Proof. Suppose that $M \models \text{PA}^-$. We shall show by an external induction on m that $M \models \underline{m} + \underline{n} = \underline{l}$ whenever $m + n = l$ in $\mathbb{N}$. If $m = 0$, then $n = l$ and so it follows by Ax6 that $M \models \underline{0} + \underline{n} = \underline{n}$. So assume inductively that $M \models \underline{n} + \underline{m} = \underline{l}$, whenever $n + m = l$ in $\mathbb{N}$. Now suppose that $n + (m + 1) = l$ and let $n + m = k$. By the inductive assumption, $M \models \underline{n} + \underline{m} = \underline{k}$. By Ax1, $M \models \underline{n} + \underline{m+1} = (\underline{n} + \underline{m}) + 1 = \underline{k} + 1 = \underline{k+1} = \underline{l}$, and thus $M \models \underline{n} + \underline{m+1} = \underline{l}$. □

Similarly, we have:

Theorem 2.7. *If $n, m, l \in \mathbb{N}$ and $n \cdot m = l$, then* $\text{PA}^- \vdash \underline{n} \cdot \underline{m} = \underline{l}$.

Next, we have:

Theorem 2.8. *If $n, m \in \mathbb{N}$ and $n < m$, then* $\text{PA}^- \vdash \underline{n} < \underline{m}$.

Finally, we have:

Theorem 2.9. *For all $n \in \mathbb{N}$,* $\text{PA}^- \vdash \forall x(x \leq \underline{n} \to (x = \underline{0} \vee x = \underline{1} \vee \cdots \vee x = \underline{n}))$

Proof. Suppose that $M \models \mathrm{PA}^-$. Again, we argue by an external induction on n. If $n = 0$, then $M \models \forall x(x \leq \underline{0} \to x = \underline{0})$ by Ax15. So assume inductively that $M \models \forall x(x \leq \underline{n} \to (x = \underline{0} \vee x = \underline{1} \vee \cdots \vee x = \underline{n}))$. Now observe that if $x \leq \underline{n+1} = \underline{n} + 1$ in M, then $x \leq \underline{n} \vee x = \underline{n} + 1 = \underline{n+1}$ by Theorem 2.5. □

Combining these results, we make a first significant discovery about model of PA. But first, we need to review (introduce) some definitions and notation. If M and N are L_A-structures, then we say that M *embeds* into N if there is a an injective map $f : M \to N$ such that $f(0^M) = 0^N$, $f(1^M) = 1^N$, $f(a +^M b) = f(a) +^N f(b)$, $f(a \cdot^M b) = a \cdot^N b$, and finally $a <^M b \to f(a) <^N f(b)$. If the embedding is the identity map, we say that M is a *submodel* of N, and denote it by $M \subseteq N$. If $M \subseteq N$ and whenever $x \in M$ and $y < x$, then $y \in M$, we say that M is an *initial segment* of N or that N is an *end-extension* of M and denote it by $M \subseteq_e N$.

Theorem 2.10. *The standard model* $(\mathbb{N}, +, \cdot, 0, 1)$ *embeds as an initial segment into every model of* PA^- *via the map* $f : n \mapsto \underline{n}$.

By associating $\mathbb{N}$ with its image, we will view it as a submodel of every model of PA^-. So from now on, we will drop the notation $\underline{n}$ and simply use n. Because $\mathbb{N}$ is an initial segment submodel of every a model M of PA^-, we can make a stronger statement about its connection to these M.

If $M \subseteq N$, we say that M is a Δ_0-elementary submodel of N, denoted by $M \prec_{\Delta_0} N$, if for any Δ_0-formula $\varphi(\overline{x})$ and any tuple $\overline{a} \in M$, we have $M \models \varphi(\overline{a}) \leftrightarrow N \models \varphi(\overline{a})$. We similarly define $M \prec_{\Sigma_n} N$.

Theorem 2.11. *The standard model* $\mathbb{N}$ *is a* Δ_0*-elementary submodel of every* $M \models \mathrm{PA}^-$.

Proof. We argue by induction on the complexity of Δ_0-formulas. Since $\mathbb{N}$ is a submodel of M, the statement is true of atomic formulas. Clearly, if the statement is true of φ and ψ, then it is true of $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$. So suppose inductively that the statement is true of $\varphi(x, y, \overline{z})$ and consider the formula $\exists x < y\, \varphi(x, y, \overline{z})$. Let $a, \overline{b} \in \mathbb{N}$. If $\mathbb{N} \models \exists x < a\, \varphi(x, a, \overline{b})$, then there is $n < a$ such that $\mathbb{N} \models \varphi(n, a, \overline{b})$ and so by our inductive assumption, $M \models \varphi(n, a, \overline{b})$ from which it follows that $M \models \exists x < a\, \varphi(x, a, \overline{b})$. If $M \models \exists x < a\, \varphi(x, a, \overline{b})$, then there is $n < a$ such that $M \models \varphi(n, a, \overline{b})$. Since $n < a$ is in $\mathbb{N}$, by our inductive assumption, $\mathbb{N} \models \varphi(n, a, \overline{b})$ and hence $\mathbb{N} \models \exists x < a\, \varphi(x, a, \overline{b})$. □

An identical proof gives the following more general result.

Theorem 2.12. *If* $M, N \models \mathrm{PA}^-$ *and* $M \subseteq_e N$, *then* $M \prec_{\Delta_0} N$.

Thus, we have our first glimpse into the structure of models of PA:

$$\underbrace{\vdash\!\!+\!\!+\!\!+\!\!+ \cdots)}_{\mathbb{N}} \;\; ?$$

To finish off the section, let us prove a first day undergraduate number theory theorem using PA.

Theorem 2.13 (Division Algorithm)**.** *Let* $M \models \mathrm{PA}$ *and* $a, b \in M$ *with* $a \neq 0$. *Then there exist unique elements* $q, s \in M$ *such that* $M \models b = aq + r \wedge r < a$.

Proof. First we prove existence. We will use induction to show that the formula $\varphi(b) := \exists q, r(b = aq + r \wedge r < a)$ holds for all b in M. Clearly we have $0 = a \cdot 0 + 0 \wedge 0 < a$ in M since $a \neq 0$. So suppose that $b = aq + r \wedge r < a$ in M. Then $b + 1 = aq + (r + 1)$ and either $r + 1 < a$ or $r + 1 = a$. If $r + 1 < a$, we are done. Otherwise, we have $b + 1 = aq + a = a(q + 1) + 0$ and we are also done. Now we prove uniqueness. Suppose that $b, q, r, q', r' \in M$ with $b = aq + r = aq' + r'$ with $r < a$ and $r' < a$. If $q < q'$, then $b = aq + r < a(q + 1) \leq aq' + r' = b$, which is impossible. Once we have that $q' = q$, it is easy to see that $r' = r$ as well. □

In Theorem 2.13, we made our first use of an induction axiom. Standard proofs of the division algorithm in Number Theory texts also use induction in the guise of the *Least Number Principle*, a scheme equivalent to induction which asserts that every definable subset has a least element. Indeed, PA^- is not sufficient to prove the division algorithm. For instance, consider the ring $\mathbb{Z}[x]$ with the following ordering. First, we define that a polynomial $p(x) > 0$ if its leading coefficient is positive. Now we define that polynomials $p(x) > q(x)$ if $p - q > 0$. It is not difficult to verify that the non-negative part of $\mathbb{Z}[x]$ is a model of PA^- but the polynomial $p(x) = x$ is neither even nor odd: there is no polynomial $q(x)$ such that $x = 2q(x)$ or $x = 2q(x) + 1$.

Remark 2.14. Suppose that φ is a formula where every quantifier is bounded by a term, $\exists x < t(\overline{y})$ or $\forall x < t(\overline{y})$. Over PA such formulas are equivalent to Δ_0-formulas. The argument is a tedious carefully crafted induction on complexity of Δ_0-formulas, where for every complexity level, we argue by induction on complexity of terms that we can transform one additional quantifier bounded by a term. We will not give the complete proof here but only hint at the details. Consider, for instance, a formula $\exists x < t(\overline{y})\, \psi(x, \overline{y})$, where $t(\overline{y}) = t_1(\overline{y}) + t_2(\overline{y})$. Then the formula

$$\exists x < t_1(\overline{y}) + t_2(\overline{y})\, \psi(x, \overline{y})$$

is equivalent over PA^- to the formula

$$\begin{array}{l} \exists b < t_2(\overline{y})\, \psi(b, \overline{y}) \vee \\ \exists a < t_1(\overline{y})\, \psi(a, \overline{y}) \vee \\ \exists a < t_1(\overline{y}) \exists b < t_2(\overline{y})\, \psi(a + b + 1, \overline{y}). \end{array}$$

Notice that both the new quantifiers are bounded by terms of lower complexity and even though we potentially introduced a new quantifier bounded by $a + b + 1$ into the formula ψ, if we assume inductively that we can remove all such quantifiers from formulas of the complexity of ψ, then we would still have that $\psi(a + b + 1, \overline{y})$ is equivalent to a Δ_0-formula. The issue that must be addressed though is that when we try to apply the inductive assumption on complexity of terms to

$$\exists a < t_1(\overline{y}) \exists b < t_2(\overline{y})\, \psi(a + b + 1, \overline{y})$$

we encounter a formula having one more quantifier ($\exists b$), then the formula complexity for which we have the inductive assumption. This is where a carefully crafted inductive assumption comes into play.

Now suppose that $s = t_1 \cdot t_2$. It follows from PA (Theorem 2.13) that every $x < t_1 \cdot t_2$ has the form $bt_1 + a$ (unless $t_1 = 0$). Thus, the formula

$$\exists x < t_1(\overline{y}) \cdot t_2(\overline{y})\, \psi(x, \overline{y})$$

is equivalent over PA to the formula

$$\exists a < t_1(\overline{y})\exists b < t_2(\overline{y}) + 1\, \psi(bt_1 + a, \overline{y}).$$

2.4. **Nonstandard models of** PA. The first nonstandard model of PA was constructed by Thoralf Skolem in the early 1930's as a *definable ultrapower* of $\mathbb{N}$. A definable ultrapower of a model M is constructed only out of functions that are definable over the model, thus ensuring that this version of the ultrapower has the same cardinality as the original model. Because it follows from PA that every definable subset has a least element, the Łoś Theorem holds for definable ultrapowers. Here, we will construct a countable nonstandard model of PA using our proof of the Completeness Theorem.

Let L_A' be the language of arithmetic expanded by adding a constant symbol c. Let PA$'$ be the theory PA together with the sentences $c > \underline{n}$ for every $n \in \mathbb{N}$. The theory PA$'$ is finitely realizable (in $\mathbb{N}$) and hence consistent. Thus, we can run the construction of the proof the Completeness Theorem to build a model of PA$'$. By carefully examining the construction, we see that the language L^* of that proof is countable and thus so is the model M consisting of the equivalence classes of constants in L^*. What does this model M (or any other countable model of PA for that matter) look like order-wise?

By construction, there is an element c above all the natural numbers.

$\vdash\!+\!+\!+\!+\cdots)_{\mathbb{N}}$ $\qquad\qquad$ $|_{c}$

We know that for every $n \in \mathbb{N}$, we must have elements $c + n$ and $c - n$. It follows that the element c sits inside a $\mathbb{Z}$-block, let's call it $\mathbb{Z}(c)$.

$\vdash\!+\!+\!+\!+\cdots)_{\mathbb{N}}$ $\qquad$ $(\cdots +\!+\!+\!+\!+ \cdots)_{c}$

The element $2c$ must also be somewhere. Where? Clearly $2c > c + n$ for every $n \in \mathbb{N}$ and so $2c$ sits above the $\mathbb{Z}(c)$-block. Indeed, the entire $\mathbb{Z}(2c)$-block sits above the $\mathbb{Z}(c)$-block.

$\vdash\!+\!+\!+\!+\cdots)_{\mathbb{N}}$ $\qquad$ $(\cdots +\!+\!+\!+\!+ \cdots)_{c}$ $\qquad$ $(\cdots +\!+\!+\!+\!+ \cdots)_{2c}$

More generally, this shows that for every $\mathbb{Z}$-block, there is a $\mathbb{Z}$-block above it. Let's assume without loss of generality that c is even (if not we can take $c + 1$). The element $\frac{c}{2}$ must be somewhere. Where? Clearly $\frac{c}{2} < c - n$ for every $n \in \mathbb{N}$ and so $\frac{c}{2}$ sits below the $\mathbb{Z}(c)$-block. Indeed, the entire $\mathbb{Z}(\frac{c}{2})$-block sits below the $\mathbb{Z}(c)$-block, but above $\mathbb{N}$.

$\vdash\!+\!+\!+\!+\cdots)_{\mathbb{N}}$ $(\cdots +\!+\!+\!+\!+ \cdots)_{\frac{c}{2}}$ $(\cdots +\!+\!+\!+\!+ \cdots)_{c}$ $\qquad$ $(\cdots +\!+\!+\!+\!+ \cdots)_{2c}$

More generally, this shows that for every $\mathbb{Z}$-block, there is a $\mathbb{Z}$-block below it but above $\mathbb{N}$. The element $\frac{3c}{2}$ must also be somewhere (we assumed that c was even). Where? The $\mathbb{Z}(\frac{3c}{2})$-blocks sits between the $\mathbb{Z}(c)$ and $\mathbb{Z}(2c)$ blocks.

$\vdash\!+\!+\!+\!+\cdots)_{\mathbb{N}}$ $(\cdots +\!+\!+\!+\!+ \cdots)_{\frac{c}{2}}$ $(\cdots +\!+\!+\!+\!+ \cdots)_{c}$ $(\cdots +\!+\!+\!+\!+ \cdots)_{\frac{3c}{2}}$ $(\cdots +\!+\!+\!+\!+ \cdots)_{2c}$

More generally, given blocks $\mathbb{Z}(c)$ and $\mathbb{Z}(d)$, the block $\mathbb{Z}(\frac{c+d}{2})$ (assuming without loss of generality that $c + d$ is even) sits between them. Thus, the $\mathbb{Z}$-blocks form a dense liner order without endpoints.

Now we have the answer. Order-wise, a countable nonstandard model of PA looks like $\mathbb{N}$ followed by densely many copies of $\mathbb{Z}$ without a largest or smallest $\mathbb{Z}$-block.

Can we similarly determine how addition and multiplication works in these nonstandard models? The answer is NO! This is the famous Tennenbaum's Theorem (1959) and we will get to it in Section 7.

For a quick glimpse into the strangeness that awaits us as we investigate the properties of the *nonstandard* elements of a model $M \models \text{PA}$, we end the section with the following observation.

Theorem 2.15. *Suppose that $M \models \text{PA}$ is nonstandard. Then there is $a \in M$ such that for all $n \in \mathbb{N}$, we have that n is a divisor of a.*

Proof. We shall use induction to argue that for every $b \in M$, there is $a \in M$ that is divisible by all elements $\leq b$. Thus, we show by induction on b that the formula

$$\varphi(b) := \exists a \forall x \leq b \exists c\, a = cx$$

holds for all $b \in M$. Obviously $\varphi(0)$ holds. So suppose inductively that there is $a \in M$ that is divisible by all elements $\leq b$. But then $a(b+1)$ is divisible by all elements $\leq b+1$. Now, by choosing b to be nonstandard, it will follow that a is divisible by all $n \in \mathbb{N}$. □

2.5. **Definability in models of** PA. Suppose that M is a first-order structure. For $n \in \mathbb{N}$, we let M^n denote the n-ary product $\underbrace{M \times \cdots \times M}_{n \text{ times}}$. A set $X \subseteq M^n$ is said to be *definable* (with parameters) over M if there is a formula $\varphi(\overline{x}, \overline{y})$ and a tuple of elements $\overline{a}$ in M such that $M \models \varphi(\overline{x}, \overline{a})$ if and only if $\overline{x} \in X$. A function $f : M^n \to M$ is said to be *definable* over M if its graph is a definable subset of M^{n+1}. Let's look at some examples of sets and functions definable over a model $M \models \text{PA}$.

Example 2.16. The set of all even elements of M is definable by the formula

$$\varphi(x) := \exists y \leq x\, 2y = x.$$

Example 2.17. The set of all prime elements of M is definable by the formula

$$\varphi(x) := \neg x = 1 \wedge \forall y \leq x\, (\exists z \leq x\, y \cdot z = x \to (y = 1 \vee y = x)).$$

Can $\mathbb{N}$ be a definable subset of M? It should be obvious after a moment's thought that this is impossible because any formula $\varphi(x, \overline{a})$ defining $\mathbb{N}$ would have the property that it is true of 0 and whenever it is true of x, it is also true of $x+1$, meaning that it would have to be true of all x in the nonstandard M. This observation leads to a widely applied theorem known as the *Overspill Principle.*

Theorem 2.18 (The Overspill Principle)**.** *Let $M \models \text{PA}$ be nonstandard. If $M \models \varphi(n, \overline{a})$ for all $n \in \mathbb{N}$, then there is a $c > \mathbb{N}$ such that*

$$M \models \forall x \leq c\, \varphi(x, \overline{a}).$$

Proof. Suppose not, then for every $c > \mathbb{N}$, there is $x \leq c$ such that $M \models \neg\varphi(x, \overline{a})$. But then we could define $n \in \mathbb{N}$ if and only if $\forall x \leq n \varphi(x, \overline{a})$. □

Example 2.19. Every polynomial function $f(x) = a_n x^n + \cdots + a_1 x + a_0$ (with parameters $a_0, \ldots, a_n \in M$) is definable over M.

Addition and multiplication are part of the language of arithmetic. What about exponentiation? Do we need to expand our language by an exponentiation function? Or...

Question 2.20. Suppose that $M \models \mathrm{PA}$. Is there always a function that extends standard exponentiation and preserves its properties that is definable over M?

For instance, it will follow from results in Section 4 that multiplication cannot be defined from addition in $\mathbb{N}$. A curious result due to Julia Robinson is that addition is definable in $\mathbb{N}$ from the successor operation $s(x) = x + 1$ and multiplication. Robinson observed that in $\mathbb{N}$, we have that

$$a + b = c$$

if and only if the triple $\langle a, b, c\rangle$ satisfies the relation

$$(1 + ac)(1 + bc) = 1 + c^2(1 + ab).$$

There are two ways that exponentiation is usually presented in a standard math textbook. One way uses the intuitive ... notation ($2^n = \underbrace{2 \cdot 2 \cdot \ldots \cdot 2}_{n}$) and the more rigorous approach relies on induction ($a^0 = 1$ and $a^{b+1} = a^b \cdot a$). Neither approach is immediately susceptible to the formalities of first-order logic, but with a decent amount of work, the inductive definition will formalize.

Suppose someone demanded of you a formal argument that $2^4 = 16$. You might likely devise the following explanation. You will write down a sequence of numbers: the first number in the sequence will be 1 (because $2^0 = 1$), each successive number in the sequence will be the previous number multiplied by 2, and the 5th number better be 16. This is our key idea! An exponentiation calculation is correct if there is a witnessing *computation sequence*. If s is a sequence, then we let $(s)_n$ denote the nth element s. Clearly it holds that $a^b = c$ if and only if there is a witnessing computation sequence s such that $(s)_0 = 1$, for all $x < b$, we have $(s)_{x+1} = a \cdot (s)_x$, and $(s)_b = c$. Here is a potential formula defining $a^b = c$:

$$\varphi(a, b, c) := \exists s\, (s)_0 = 1 \wedge \forall x < b\, (s)_{x+1} = a \cdot (s)_x \wedge (s)_b = c.$$

Unfortunately, this makes no sense over a model of PA. Or does it? For it to make sense, an element of a model of PA would have be definably interpretable as a sequence of elements.

2.6. **Coding in models of** PA**.** Let's first think about how a natural number can be thought of as coding a sequence of natural numbers. Can this be done in some reasonable way? Sure! For instance, we can use the uniqueness of primes decomposition. Let p_n denote the nth prime number. We code a sequence $a_0, \ldots, a_{n-1}$ with by the number

$$p_0^{a_0+1} \cdot p_1^{a_1+1} \cdot \ldots \cdot p_{n-1}^{a_{n-1}+1}.$$

We need to add 1 to the exponents because otherwise we cannot code 0. Notice that under this coding only some natural numbers are codes of sequences.

Example 2.21. The sequence $\langle 5, 1, 2\rangle$ gets coded by the number $2^6 \cdot 3^2 \cdot 5^3$.

Example 2.22. The number $14 = 2 \cdot 7$ does not code a sequence because its decomposition does not consist of consecutive prime numbers.

This coding is relatively simple (although highly inefficient), but it does not suit our purposes because it needs exponentiation. We require a clever enough coding that does not already rely on exponentiation. The way of coding we are about to introduce relies on the Chinese Remainder Theorem (from number theory) and is due to Gödel.

Let's introduce some standard number theoretic notation. We abbreviate by $b|a$ the relation b divides a. We abbreviate by $(\frac{a}{b})$ the remainder of the division of a by b. We abbreviate by $(a, b) = c$ the relation expressing that c is the largest common devisor of a and b.

Theorem 2.23 (Chinese Remainder Theorem)**.** *Suppose $m_0, \ldots, m_{n-1} \in \mathbb{N}$ are pairwise relatively coprime. Then for any sequence $a_0, \ldots, a_{n-1}$ of elements of $\mathbb{N}$, the system of congruences*

$$\begin{gathered} x \equiv a_0 (\text{mod } m_0) \\ x \equiv a_1 (\text{mod } m_1) \\ \vdots \\ x \equiv a_{n-1} (\text{mod } m_{n-1}) \end{gathered}$$

has a solution.

Example 2.24. The system of congruences

$$\begin{gathered} x \equiv 2 (\text{mod } 3) \\ x \equiv 1 (\text{mod } 5) \end{gathered}$$

has a solution $x = 11$.

The next theorem, which makes up the second piece of the coding puzzle, states that there are easily definable arbitrarily long sequences of coprime numbers whose first element can be made arbitrarily large.

Theorem 2.25. *For every $n, k \in \mathbb{N}$, there is $m > k$ such that elements of the sequence*

$$m + 1, 2m + 1, \ldots, nm + 1$$

are pairwise relatively coprime.

Hint of proof. Let m be any multiple of $n!$ chosen so that $m > k$. □

Now suppose we are given a sequence of numbers $a_0, \ldots, a_{n-1}$. Here is the coding algorithm:

1. We find $m > \max\{a_0, \ldots, a_{n-1}\}$ so that

$$m + 1, 2m + 1, \ldots, nm + 1$$

are relatively coprime.

2. By the Chinese Remainder Theorem, there is $a \in \mathbb{N}$ such that for $0 \leq i < n$, we have

$$a \equiv a_i (\text{mod } (i + 1)m + 1)$$

3. Since $a_i < (i + 1)m + 1$ for all $0 \leq i < n$, it follows that

$$a_i = \left(\frac{a}{(i + 1)m + 1} \right).$$

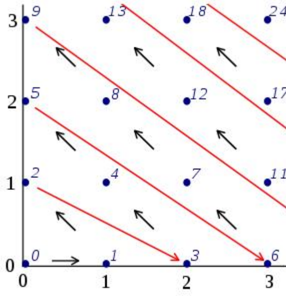
4. The pair (a, m) codes the sequence $a_0, \ldots, a_{n-1}$!

Most significantly, the decoding mechanism is expressible using only operations of addition and multiplication. Also, conveniently, we may view *every* pair of numbers (a, m) as coding a sequence s, whose i^{th}-element is

$$(s)_i = \left(\frac{a}{(i+1)m+1}\right).$$

Armed with the Chinese Remainder Theorem coding algorithm, we can now code any finite sequence of numbers by a pair of numbers. This is good enough for the purpose of defining exponentiation over $\mathbb{N}$ using just addition and multiplication. The quantifier $\exists s$ would be replaced by $\exists a, m$ and the expression $(s)_i$ would be replaced with $\left(\frac{a}{(i+1)m+1}\right)$, which is easily expressible in L_A. But with a minor bit more work, we can see how to code pairs of numbers by a single number, so that every number codes a pair of numbers. One way to accomplish this is using *Cantor's pairing function*

$$\langle x, y\rangle = \frac{(x+y)(x+y+1)}{2} + y.$$



Theorem 2.26. *Cantor's pairing function* $\langle x, y\rangle$ *is a bijection between* $\mathbb{N}^2$ *and* $\mathbb{N}$.

The proof is left to the homework. Cantor's pairing function $z = \langle x, y\rangle$ is definable over $\mathbb{N}$ by the quantifier-free formula

$$2z = (x+y)(x+y+1) + 2y.$$

It also easy to see that the code of a pair is at least as large as its elements, $x, y \leq \langle x, y\rangle$. Using Cantor's pairing function, we can inductively define, for every $n \in \mathbb{N}$, the bijection $\langle x_1, \ldots, x_n\rangle : \mathbb{N}^n \to \mathbb{N}$ by $\langle x_1, \ldots, x_n\rangle = \langle x_1, \langle x_2, \ldots, x_n\rangle\rangle$. Arguing inductively, it is easy to see that functions $z = \langle x_1, \ldots, x_n\rangle$ are all Δ_0-definable over $\mathbb{N}$. Supposing that $z = \langle x_1, \ldots, x_n\rangle$ is Δ_0-definable, we define $z = \langle x_1, \ldots, x_n, x_{n+1}\rangle$ by the Δ_0-formula

$$\exists w \leq z\, w = \langle x_2, \ldots, x_{n+1}\rangle \wedge 2z = (x_1 + w)(x_1 + w + 1) + 2w.$$

So far we have worked exclusively with the natural numbers and so it remains to argue that this coding method works in an arbitrary model of PA. Once that is done, we will have that any element of a model of PA may be viewed as a sequence of elements of a model of PA and with some additional arguments (using induction) we will be able to show that elements coding sequences witnessing exponentiation (and other fun stuff) exist.

The full details of formalizing the coding arguments over a model of PA will be left as homework. Here we just give a brief overview of how they proceed.

First, we show that Theorem 2.25, asserting that easily definable arbitrarily long sequences of coprime elements exist, is provable from PA.

Theorem 2.27.

$$\mathrm{PA} \vdash \forall k, n \exists m(m > k \wedge \forall i, j < n(i \neq j \to ((i+1)m+1, (j+1)m+1) = 1)).$$

Hint of proof. Suppose that $M \models \mathrm{PA}$ and $k, n \in M$. We let m' be any element of M that is divisible by all $x \leq n$. The element m' plays the role of $n!$. Now we let m be any multiple of m' chosen so that $m > k$.

□

We cannot hope to express the Chinese Remainder Theorem in L_A until we develop the sequence coding machinery, so luckily we do not need its full power for our purposes. Instead, we will argue that given a, m coding a sequence s and an element b, there are a', m' coding s extended by b. This property will allow us to argue inductively that a code of some desired sequence exists. The next result is an auxiliary theorem for that argument. We show that whenever an element z is relatively coprime with elements of the form $(i+1)m+1$ for all $i < n$, then there is an element w that is relatively coprime with z and which divides all $(i+1)m+1$ for $i < n$.

Theorem 2.28.

$$\mathrm{PA} \vdash \forall m, n, z(\forall i < n\, ((i+1)m+1, z) = 1 \to \exists w((w, z) = 1 \wedge \forall i < n\, ((i+1)m+1)|w)).$$

Hint of proof: We argue by induction on k up to n in the formula

$$\exists w(w, z) = 1 \wedge \forall i < k\, ((i+1)m+1)|w,$$

using that if w worked for stage k, then $w((k+1)m+1)$ will work for stage $k+1$. □

Theorem 2.29. $\mathrm{PA} \vdash \forall a, m, b, n \exists a', m'$

$$\forall i < n \left(\frac{a'}{(i+1)m'+1}\right) = \left(\frac{a}{(i+1)m+1}\right) \wedge \left(\frac{a'}{(n+1)m'+1}\right) = b$$

Hint of proof: We choose $m' > \max\{b, m\}$ satisfying Theorem 2.25. First, we show by induction on k up to n that

$$M \models \forall k \leq n \exists c \forall i < k \left(\frac{c}{(i+1)m'+1}\right) = \left(\frac{a}{(i+1)m+1}\right)$$

(this uses Theorem 2.28). Finally, we construct a' from c using the same argument as for the inductive step of above. □

Theorem 2.30. PA *proves that Cantor's pairing function* $\langle x, y \rangle$ *is a bijection:*

(1) $\forall z \exists x, y\, \langle x, y \rangle = z,$
(2) $\forall x, y, u, v(\langle x, y \rangle = \langle u, v \rangle \to x = y \wedge y = v).$

Thus, every model $M \models \mathrm{PA}$ has Δ_0-definable bijections $z = \langle x_1, \ldots, x_n \rangle$ between M and M^n.

Remark 2.31. Contracting Quantifiers
Using the function $\langle x_1,\ldots,x_n\rangle$, we show that every Σ_{m+1}-formula with a block of existential quantifiers $\exists y_1,\ldots,y_n\,\varphi(y_1,\ldots,y_n)$ is equivalent over PA to a Σ_{m+1}-formula with a single existential quantifier

$$\exists y\,\underbrace{\exists y_1,\ldots,y_n\leq y\,y=\langle y_1,\ldots,y_n\rangle\wedge\varphi(y_1,\ldots,y_n)}_{\Pi_m}.$$

The same holds for Π_{m+1}-formulas as well.

Now finally, given any $x,i\in M$, we define that

$$(x)_i=\left(\frac{a}{m(i+1)+1}\right)$$

where $x=\langle a,m\rangle$. The relationship $(x)_i=y$ is expressible by the Δ_0-formula

$$\exists a,m\leq x\left(\begin{array}{l}(a+m)(a+m+1)+2m=2x\wedge\\ \exists q\leq a\,a=q((i+1)m+1)+y\wedge\\ y<(i+1)m+1\end{array}\right).$$

The function $(x)_i$ is better know as the Gödel β-function, $\beta(x,i)$.
The next theorem summarizes the coding properties we have proved.

Theorem 2.32. PA *proves that*
(1) $\forall a\exists x\,(x)_0=a$
(2) $\forall x,a,b\exists y(\forall i<a\,(x)_i=(y)_i\wedge(y)_a=b)$
(3) $\forall x,i\,(x)_i\leq x$.

Example 2.33. Suppose that $M\models\text{PA}$. In M, consider the definable relation

$$\exp(a,b,c):=\exists s((s)_0=1\wedge\forall x<b\,(s)_{x+1}=a(s)_x\wedge(s)_b=c).$$

We shall argue that exp defines a total function on M, meaning that for every $a,b\in M$, there is a $c\in M$ such that $M\models\exp(a,b,c)$ and such c is unique. To prove uniqueness, we suppose that there are two witnessing sequences s and s' and argue by induction up to b that $(s)_x=(s')_x$ for all $x\leq b$. To prove existence, we fix $a\in M$ and argue by induction on b. By Theorem 2.32, there is $s\in M$ such that $(s)_0=1$, which witnesses that $a^0=1$. So suppose inductively that there is $s\in M$ witnessing that $\exp(a,b,c)$ holds for some c. By Theorem 2.32, there is s' such that $(s)_x=(s')_x$ for all $x\leq b$ and $(s')_{b+1}=a\cdot(s)_b$. Thus, s' is a sequence witnessing that $\exp(a,b+1,ac)$ holds. Similarly, using induction, we can show that exp satisfies all expected properties of exponentiation. Finally, we argue that for $a,b\in\mathbb{N}$, if $a^b=c$, then $M\models\exp(a,b,c)$ and thus exp extends true exponentiation to all of M. Suppose that $a^b=c$, then there is a sequence $s\in\mathbb{N}$ witnessing this and since $\mathbb{N}\prec_{\Delta_0}M$, the model M must agree.

Example 2.34. Suppose that $M\models\text{PA}$. In M, the relation

$$\exists s((s)_0=1\wedge\forall x<a(s)_{x+1}=(x+1)(s)_x\wedge(s)_a=b)$$

defines a total function extending the factorial function to M and satisfying all its expected properties.

Our sequences are still missing one useful feature, namely length. Given an element x, we know how to obtain the ith-element coded by x, but we don't know how many of the elements x codes are meaningful. Up to this point, we did not

require this feature but it is feasible that it might yet prove useful. To improve our coding by equipping sequences with length, we simply let $(x)_0$ tell us how many elements of x we are interested in. Given x, let us adopt the convention that $(x)_0$ is the length of x and for all $i < \text{len}(x)$, we denote by $[x]_i = (x)_{i+1}$ the ith-element of x. It will also be convenient to assume that $[x]_i = 0$ for all $i \geq \text{len}(x)$.

One other potential deficiency of our coding is that a given sequence does not have a unique code. But this is also easily remedied. Let us define a predicate $\text{Seq}(s)$ that will be true only of s that are the smallest possible codes of their sequences

$$\text{Seq}(s) := \forall s' < s\,(\text{len}(s') \neq \text{len}(s) \vee \exists i < \text{len}(s)[s]_i \neq [s']_i).$$

Now every sequence s has a unique code s': $\forall s \exists s'(\text{Seq}(s') \wedge \forall i < \text{len}(s)([s]_i = [s']_i))$.

Finally, here is a coming attraction. So far using coding, we gained the ability to definably extend to models of PA, recursively expressible functions on the natural numbers. What else can coding be used for? Here is something we can code: languages! Letters of the alphabet can be represented by numbers, words can be represented by sequences of numbers, and sentences by sequences of sequences of numbers. If we are dealing with a formal language, we should also be able to define whether a given number codes a letter, a word, or a sentence. In this way we should in principle be able to code first-order languages, such as L_A. Below is a preview.

Suppose that $M \models \text{PA}$. First, we assign number codes to the alphabet of L_A in some reasonable manner.

L_A-symbol	Code
0	0
1	1
$+$	2
$\cdot$	3
$<$	4
$=$	5
$\wedge$	6
$\vee$	7
$\neg$	8
$\exists$	9
$\forall$	10
$($	11
$)$	12
x_i	$\langle 13, i\rangle$

This allows us to define the alphabet of L_A as

$$A(x) := x \leq 12 \vee \exists y\, x = \langle 13, y\rangle.$$

Using $A(x)$, we next definably express whether an element of M codes a term of L_A. Loosely, an element $t \in M$ codes a term of L_A if it is a sequence consisting of elements satisfying $A(x)$ and there is a *term building* sequence s witnessing its construction from basic terms. An element of s is either a singleton sequence coding one of the basic terms $0, 1, x_i$, or it is a sequence coding one of the expressions $x+y$, $x\cdot y$ where x, y previously appeared in s, and the last element of s is t. Notice that for the natural numbers, this definition holds exactly of those numbers coding terms, but if M is nonstandard, then the definition also holds of *nonstandard* terms, such

as $\underbrace{1+\cdots+1}_{a}$ for some nonstandard $a \in M$. Similarly, we define that an element of M codes a formula of L_A if there is a *formula building* sequence witnessing this. We can also definably express whether the code of a formula represents a Peano Axiom. In this way, we have made a model of PA reason about models of PA. This sort of self-reference is bound to get us into some variety of trouble. Stay tuned for the Gödel Incompleteness Theorems. In a similar way, we can code finite (and even certain infinite) languages inside models of PA and make these models reason about the properties of formulas of such a language. Indeed, as we will see in Section 5, models of PA can even prove the Completeness Theorem!

2.7. **Standard Systems.** One of the central concepts in the study of models of PA is that of a standard system. Suppose that $M \models \text{PA}$ is nonstandard. The *standard system* of M, denoted by $SSy(M)$, is the collection of all subsets of $\mathbb{N}$ that arise as traces of definable subsets of M on the natural numbers. More precisely

$$SSy(M) = \{A \cap \mathbb{N} \mid A \text{ is a definable subset of } M\}.$$

Example 2.35. All sets that are Δ_0-definable over $\mathbb{N}$ are in the standard system of every nonstandard $M \models \text{PA}$. Suppose that $A \subseteq \mathbb{N}$ is definable over $\mathbb{N}$ by a Δ_0-formula $\varphi(x)$ and $M \models \text{PA}$ is nonstandard. If $A' \subseteq M$ is defined by $\varphi(x)$ over M, then, since $\mathbb{N} \prec_{\Delta_0} M$, we have $A = A' \cap \mathbb{N}$. Thus, in particular, the following sets are in $SSy(M)$ of every nonstandard $M \models \text{PA}$:

(1) the set of all even numbers,
(2) the set of all prime numbers,
(3) $\mathbb{N}$ (definable by $\varphi(x) := x = x$).

Example 2.36. All sets that are definable over $\mathbb{N}$ by a $\Delta_1(\text{PA})$-formula are in the standard system of every $M \models \text{PA}$. Suppose that $A \subseteq \mathbb{N}$ is definable over $\mathbb{N}$ by a Σ_1-formula $\exists y\varphi(x,y)$, where $\varphi(x,y)$ is Δ_0, that is equivalent over PA to a Π_1-formula $\forall y\psi(x,y)$, where $\psi(x,y)$ is Δ_0. Let A' be the set definable over M by the formula $\exists y\varphi(x,y)$. We shall argue that $A = A' \cap \mathbb{N}$. Suppose that $n \in A$. Then $\mathbb{N} \models \exists y\varphi(n,y)$, and so there is $m \in \mathbb{N}$ such that $\mathbb{N} \models \varphi(n,m)$. Since $\mathbb{N} \prec_{\Delta_0} M$, it follows that $M \models \varphi(n,m)$, and thus $M \models \exists y\varphi(n,y)$. This demonstrates that $n \in A'$. Now suppose that $n \in \mathbb{N} \cap A'$. Then $M \models \exists y\varphi(n,y)$ and, since $\exists y\varphi(x,y)$ is equivalent over PA to $\forall y\psi(x,y)$, we have that $M \models \forall y\psi(n,y)$. In particular, for every $m \in \mathbb{N}$, we have that $M \models \psi(n,m)$. Thus, by Δ_0-elementarity, it follows that $\mathbb{N} \models \forall y\psi(n,y)$. But then, $\mathbb{N} \models \exists y\varphi(n,y)$. This demonstrates that $n \in A$.

In Section 3, we will generalize Example 2.36 to show that every subset of $\mathbb{N}$ that is definable by a $\Delta_1(\mathbb{N})$-formula is in $SSy(M)$ of every nonstandard $M \models \text{PA}$. This is not an immediate consequence of Example 2.36 because a formula that is equivalent over $\mathbb{N}$ to both a Σ_1 and a Π_1-formula has no a priori reason to have similar equivalent variants over some nonstandard $M \models \text{PA}$.

Question 2.37. What are other general properties of standard systems?

In Section 7, we will show that standard systems of countable models of PA are completely characterized by three properties, meaning that any standard system must possess those properties and any collection of subsets of $\mathbb{N}$ that possesses those properties is the standard system of some model $M \models \text{PA}$. The question of whether the same characterization holds for uncountable models of PA has been open for

over half a century! Two of these properties are presented below, but the third will have to wait until Section 7. First though, it is useful to observe that in studying $SSy(M)$, we do not need to examine all definable subsets of a model $M \models \mathrm{PA}$, but merely its Δ_0-definable subsets.

Let us say that $A \subseteq \mathbb{N}$ is *coded* in a nonstandard $M \models \mathrm{PA}$, if there is $a \in M$ such that $n \in A$ if and only if $M \models (a)_n \neq 0$.

Theorem 2.38. *Suppose that $M \models \mathrm{PA}$ is nonstandard. Then $SSy(M)$ is the collection of all subsets of $\mathbb{N}$ that are coded in M.*

Proof. First, suppose that $A \subseteq \mathbb{N}$ is coded in M. Then there is $a \in M$ such that $n \in A$ if and only if $M \models (a)_n \neq 0$. Let $\varphi(x, a) := (a)_x \neq 0$, and observe that the intersection of the set defined by $\varphi(x, a)$ with $\mathbb{N}$ is exactly A. Now suppose that there is a formula $\varphi(x, y)$ and $\bar{b} \in M$ such that $A = \{n \in \mathbb{N} \mid M \models \varphi(n, \bar{b})\}$. Let $c \in M$ be nonstandard. We argue by induction on y up to c in the formula

$$\psi(y, \bar{b}) := \exists a \forall x \leq y\, (a)_x \neq 0 \leftrightarrow \varphi(x, \bar{b})$$

that there is $a \in M$ such that for all $x \leq c$, we have $M \models (a)_x \neq 0 \leftrightarrow \varphi(x, \bar{b})$. Define that $i_x = 1$ if $M \models \varphi(x, \bar{b})$ and $i_x = 0$ otherwise. By Theorem 2.32, there is $a \in M$ such that $(a)_0 = i_0$. So suppose inductively that there is $a \in M$ such that for all $x \leq y$, we have $M \models (a)_x \neq 0 \leftrightarrow \varphi(x, \bar{b})$. By Theorem 2.32, there is $a' \in M$ such that for all $x \leq y$, we have $(a)_x = (a')_x$ and $(a')_{y+1} = i_{y+1}$. □

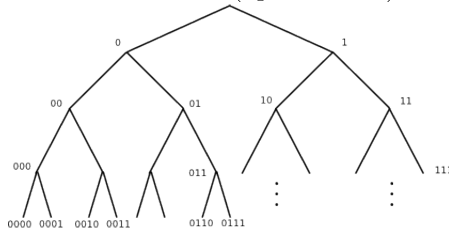
Now back to properties of standard systems. The following result follows almost immediately from the definition of standard systems.

Theorem 2.39 (Property (1) of standard systems)**.** *Suppose that $M \models \mathrm{PA}$ is nonstandard. Then $SSy(M)$ is a Boolean algebra of subsets of $\mathbb{N}$.*

To introduce the second property of standard systems, we first need some background on finite binary trees.

Let *Bin* denote the collection of all finite binary sequences. Using, the Chinese Remainder Theorem coding developed in the previous section, every finite binary sequence has a unique natural number code, that is the least natural number coding that sequence. Via this coding, we shall view every subset of *Bin* as a subset of $\mathbb{N}$.

A subset T of *Bin* is called a (*binary*) *tree* if whenever $s \in T$, then so are all its initial segments (if $k \leq \mathrm{len}(s)$, then $s \restriction k \in T$). We will refer to elements of T as *nodes* and define the *level* n of T to consist of all nodes of length n. The collection *Seq* is itself a binary tree, called the *full binary tree*. A tree T comes with a natural partial ordering defined by $s \leq u$ in T if $u \restriction \mathrm{len}(s) = s$. A set $b \subseteq T$ is a called a *branch* of T if b is a tree and and its nodes are linearly ordered. Below is a visualization of a (upside down) binary tree.



Theorem 2.40 (König's Lemma, 1936)**.** *Every infinite binary tree has an infinite branch.*

Proof. Suppose that T is an infinite binary tree. We define the infinite branch b by induction. Let $s_0 = \emptyset$. Next, observe that since T is infinite, either the node $\langle 0\rangle$ or the node $\langle 1\rangle$ has infinitely many nodes in T above it. If $\langle 0\rangle$ has infinitely many nodes above it, we let $s_1 = \langle 0\rangle$, otherwise, we let $s_1 = \langle 1\rangle$. Now suppose inductively that we have chosen s_n on level n of T with infinitely many nodes in T above it. Again either $s_n{}^\frown 0$ or $s_n{}^\frown 1$ has infinitely many nodes in T above it. If $s_n{}^\frown 0$ has infinitely many nodes above it, we let $s_{n+1} = s_n{}^\frown 0$, otherwise, we let $s_{n+1} = s_n{}^\frown 1$. It is clear that $b = \{s_n \mid n \in \mathbb{N}\}$ is an infinite branch of T. □

The second property of standard systems assets that whenever a tree T is an element of a standard system, then at least one of its infinite branches must end up in the standard system as well. All infinite trees in a standard systems have branches there!

Theorem 2.41 (Property (2) of standard systems)**.** *Suppose that $M \models \mathrm{PA}$ is nonstandard. If $T \in SSy(M)$ is an infinite binary tree, then there is an infinite branch B of T such that $B \in SSy(M)$.*

Proof. First, we define that s is an *M-finite binary sequence* if

$$M \models \mathrm{Seq}(s) \wedge \forall i < \mathrm{len}(s)\, ([s]_i = 0 \vee [s]_i = 1).$$

It is clear that (the code of) every finite binary sequence is an M-finite binary sequence and, indeed, any M-finite binary sequence of length n for some $n \in \mathbb{N}$ is (the code of a) finite binary sequence.

Now suppose that $T \in SSy(M)$ is an infinite binary tree. Let

$$n \in T \leftrightarrow M \models \varphi(n, \bar{b}).$$

Since $\mathbb{N}$ is not definable in M, it is not possible that $\varphi(x, \bar{b})$ looks like a tree only on the natural numbers. The tree properties of $\varphi(x, \bar{b})$ must overspill into the nonstandard part, meaning that there will be an M-finite binary sequence a satisfying $\varphi(x, \bar{b})$ such that all its initial segments also satisfy $\varphi(x, \bar{b})$. But then the truly finite initial segments of a form a branch through T. Now for the details.

Since T has a node on every level $n \in \mathbb{N}$, we have that, for every $n \in \mathbb{N}$, there is $m \in \mathbb{N}$ such that M satisfies

(1) (the sequence coded by) m has length n,
(2) $\varphi(m, \bar{b})$ holds,
(3) $\varphi(x, \bar{b})$ holds of all initial segments of (the sequence coded by) m.

By the Overspill Principle, there must be some nonstandard $a, c \in M$ such that M satisfies

(1) (the sequence coded by) a has length c,
(2) $\varphi(a, \bar{b})$ holds,
(3) $\varphi(x, \bar{b})$ holds of all initial segments of (the sequence coded by) a.

Let $\psi(x, a)$ define the set of all initial segments of a and let B be the intersection of this set with $\mathbb{N}$. By our earlier observations, B consists precisely of all initial segments of a of length n for $n \in \mathbb{N}$. Thus, $B \in SSy(M)$ and $B \subseteq T$ is an infinite branch as desired. □

2.8. **Homework.**
Sources:

(1) *Models of Peano Arithmetic* [Kay91]

Question 2.1. Prove the Chinese Remainder Theorem in $\mathbb{N}$.

Question 2.2. Prove Theorem 2.25 in $\mathbb{N}$.

Question 2.3. Prove Theorem 2.27.

Question 2.4. Prove Theorem 2.28.

Question 2.5. Prove Theorem 2.29.

Question 2.6. Prove Theorem 2.4.

3. Recursive Functions

"To understand recursion, you must understand recursion."

3.1. **On algorithmically computable functions.** In his First Incompleteness Theorem paper (see Section 4), Gödel provided the first explicit definition of the class of *primitive recursive* functions on the natural numbers ($\mathbb{N}^k \to \mathbb{N}$). Properties of functions on the natural numbers defined by the process of recursion have been studied at least since the middle of the 19$^{\text{th}}$ century by the likes of Dedekind and Peano, and then in the early 20$^{\text{th}}$ century by Skolem, Hilbert, and Ackermann. In the 20$^{\text{th}}$ century, the subject began to be associated with the notion of effective, or as we will refer to it here, algorithmic computability. Algorithmically computable functions, those that can be computed by a finite mechanical procedure[8], came to play a role in mathematics as early as in Euclid's number theory. Attempts to study them formally were made by Leibniz (who built the first working calculator), Babbage (who attempted to build a computational engine), and later by Hilbert and Ackermann. But it was Gödel's definition of primitive recursive functions that finally allowed mathematicians to isolate what they believed to be precisely the class of the algorithmically computable functions and precipitated the subject of computability theory. Realizing along with several other mathematicians that there were algorithmically computable functions that were not primitive recursive, Gödel extended the class of primitive recursive functions to the class of *recursive* functions. Led by Alonzo Church, mathematicians began to suspect that the recursive functions were precisely the algorithmically computable functions. Several radically different definitions for the class of algorithmically computable functions, using various models of computation, were proposed close to the time of Gödel's definition of the recursive functions. All of them, including the physical modern computer, proved to compute exactly the recursive functions. The assertion that any reasonably defined model of computation will compute exactly the recursive functions became known as *Church's Thesis.* In this section we will study the primitive recursive and recursive functions, while in Section 7, we will encounter another model of computation, the Turing machines, introduced by the great Alan Turing.

[8]Traditionally the notion of algorithmic computability applies to functions on the natural numbers and we will study it only in this context. It should nevertheless be mentioned that there is much fascinating work extending the algorithmic computability to functions on the real numbers.

3.2. **Primitive recursive functions.** Since the recursive functions purport to be exactly the algorithmically computable functions, it is often instructive to retranslate the definitions we are about to give into familiar programming concepts. After all, we are claiming that any function computable by any program in any programming language must turn out to be one of the class we are about to define in this and the next section.[9]

The primitive recursive functions consist of the so-called *basic* functions together with two closure operations.

Basic Functions

(1) Constant zero function: $Z : \mathbb{N} \to \mathbb{N}$, $Z(x) = 0$
(2) Projection functions: $P_i^n : \mathbb{N}^n \to \mathbb{N}$, $P_i^n(x_1, \ldots, x_n) = x_i$
(3) Successor function: $S : \mathbb{N} \to \mathbb{N}$, $S(x) = x + 1$.

Closure Operations

(1) **Composition**
Suppose that $g_i : \mathbb{N}^m \to \mathbb{N}$, for $1 \leq i \leq n$, and $h : \mathbb{N}^n \to \mathbb{N}$. We define a new function $f : \mathbb{N}^m \to \mathbb{N}$ by *composition*:
$$f(\overline{x}) = h(g_1(\overline{x}), \ldots, g_n(\overline{x})).$$
(2) **Primitive Recursion**
Suppose that $h : \mathbb{N}^{n+2} \to \mathbb{N}$ and $g : \mathbb{N}^n \to \mathbb{N}$. We define a new function $f : \mathbb{N}^{n+1} \to \mathbb{N}$ by *primitive recursion*:
$$f(\overline{y}, 0) = g(\overline{y})$$
$$f(\overline{y}, x + 1) = h(\overline{y}, x, f(\overline{y}, x)).$$
Here, we allow $n = 0$, in which case, we have
$$f(0) = k \text{ for some fixed } k \in \mathbb{N}$$
$$f(x + 1) = h(x, f(x)).$$

A function $f : \mathbb{N}^n \to \mathbb{N}$ is said to be *primitive recursive* if it can be built in finitely many steps from the basic functions using the operations of composition and primitive recursion. Thus, the class of primitive recursive functions is the smallest class containing the basic functions that is closed under the operations of composition and primitive recursion. The next several examples will illustrate that many familiar functions are primitive recursive.

Let $Z^n : \mathbb{N}^n \to \mathbb{N}$ denote the n-ary constant zero function, $Z^n(x_1, \cdots, x_n) = 0$.

Example 3.1. The functions Z^n are primitive recursive for every $n \in \mathbb{N}$ since
$$Z^n(x_1, \ldots, x_n) = Z(P_1^n(x_1, \ldots, x_n)).$$

Let $C_i^n : \mathbb{N}^n \to \mathbb{N}$ denote the n-ary constant function with value i, $C_i^n(x_1, \ldots, x_n) = i$.

Example 3.2. The functions C_i^n are primitive recursive for every i, n.
Using projections, it suffices to show all C_i^1 are primitive recursive. We argue by induction on i. The function $C_0^1(x) = Z(x)$ is primitive recursive. So suppose inductively that C_i^1 is primitive recursive. Then
$$C_{i+1}^1(x) = S(C_i^1(x))$$
is primitive recursive as well.

[9] A good source for material in this section is [Coo04].

Example 3.3. Addition $\mathrm{Add}(m,n)=m+n$ is primitive recursive.
We define Add by primitive recursion, using that $m+0=m$ and $m+(n+1)=(m+n)+1$, as

$$\mathrm{Add}(m,0)=P_1^1(m),$$
$$\mathrm{Add}(m,n+1)=S(P_3^3(m,n,\mathrm{Add}(m,n))).$$

We must make use of projections to satisfy the formalism that h must be a 3-ary function.

Example 3.4. Multiplication $\mathrm{Mult}(m,n)=m\cdot n$ is primitive recursive.
We define Mult by primitive recursion, using that $m\cdot 0=0$ and $m\cdot(n+1)=(m\cdot n)+m$, as

$$\mathrm{Mult}(m,0)=Z(m),$$
$$\mathrm{Mult}(m,n+1)=\mathrm{Add}(P_3^3(m,n,\mathrm{Mult}(m,n)),P_1^3(m,n,\mathrm{Mult}(m,n))).$$

Again, we use projections to stay within the formalism of primitive recursion.

Example 3.5. Exponentiation $\mathrm{Exp}(m,n)=m^n$ is primitive recursive.

Example 3.6. The factorial function $\mathrm{Fact}(m)=m!$ is primitive recursive.

Example 3.7. The predecessor function $\mathrm{Pred}(n)=n\dot{-}1$, where

$$n\dot{-}1=\begin{cases} n-1 & \text{if } n>0,\\ 0 & \text{if } n=0,\end{cases}$$

is primitive recursive.
We define Pred by primitive recursion, using that $\mathrm{Pred}(0)=0$ and $\mathrm{Pred}(n+1)=n$, as

$$\mathrm{Pred}(0)=0,$$
$$\mathrm{Pred}(n+1)=P_1^2(n,\mathrm{Pred}(n)).$$

Example 3.8. The truncated subtraction function $\mathrm{Sub}(m,n)=m\dot{-}n$, where

$$m\dot{-}n=\begin{cases} m-n & \text{if } m\geq n,\\ 0 & \text{if } m<n,\end{cases}$$

is primitive recursive.
We define Sub by primitive recursion, using that $\mathrm{Sub}(m,0)=m$ and $\mathrm{Sub}(m,n+1)=\mathrm{Pred}(\mathrm{Sub}(m,n))$, as

$$\mathrm{Sub}(m,0)=m,$$
$$\mathrm{Sub}(m,n+1)=\mathrm{Pred}(P_3^3(m,n,\mathrm{Sub}(m,n))).$$

The next two technical functions will prove very useful.

Example 3.9. The functions

$$\mathrm{sg}(n)=\begin{cases} 0 & \text{if } n=0,\\ 1 & \text{if } n\neq 0\end{cases}$$

and

$$\overline{\mathrm{sg}}(n)=\begin{cases} 1 & \text{if } n=0,\\ 0 & \text{if } n\neq 0\end{cases}$$

are primitive recursive.

Example 3.10. If $f(\overline{m}, n)$ is primitive recursive, then so is the *bounded sum*

$$s(\overline{m}, k) = \Sigma_{n<k} f(\overline{m}, n),^{10}$$

and the *bounded product*

$$p(\overline{m}, k) = \Pi_{n<k} f(\overline{m}, n).^{11}$$

If we intend to capture all algorithmically computable functions, then it seems necessary that we are able to compute a *definition by cases*, as for example the function

$$f(m, n) = \begin{cases} m + n & \text{if } m < n, \\ m \cdot n & \text{otherwise.} \end{cases}$$

In programming languages this is handled by the 'if, else' construct. If a function is defined by cases, then computing its value depends on computing the truth value of the relations on which the cases are based. So, we begin by introducing the notion of a primitive recursive relation. A relation $R(\overline{x})$ is said to be *primitive recursive* if its characteristic function

$$\chi_R(\overline{x}) = \begin{cases} 1 & \text{if } R(\overline{x}) \text{ holds,} \\ 0 & \text{otherwise} \end{cases}$$

is primitive recursive.

Example 3.11. The relation $m = n$ is primitive recursive since

$$\chi_=(m, n) = \overline{\text{sg}}((m \dot{-} n) + (n \dot{-} m)).$$

The relation $m < n$ is also primitive recursive since

$$\chi_<(m, n) = \text{sg}(n \dot{-} m).$$

Theorem 3.12. *The primitive recursive relations are closed under complement, union, and intersection.*

Proof. Suppose that R and S are primitive recursive relations of the same arity.

$$\chi_{R\cap S} = \chi_R \cdot \chi_S$$
$$\chi_{R\cup S} = \text{sg}(\chi_R + \chi_S)$$
$$\chi_{\neg R} = 1 \dot{-} \chi_R$$

□

Theorem 3.13 (Definition by Cases)**.** *Suppose that $g(\overline{x})$ and $h(\overline{x})$ are primitive recursive functions and $R(\overline{x})$ is a primitive recursive relation. Then the function*

$$f(\overline{x}) = \begin{cases} g(\overline{x}) & \text{if } R(\overline{x}) \text{ holds,} \\ h(\overline{x}) & \text{otherwise} \end{cases}$$

is primitive recursive.

Proof.

$$f(\overline{x}) = \chi_R(\overline{x}) \cdot g(\overline{x}) + \chi_{\neg R}(\overline{x}) \cdot h(\overline{x})$$

□

As in programming, once you have the basic 'if, else', you automatically get the 'if, else if' construct.

[10] If $k = 0$, then the value of the bounded sum is defined to be 0.

[11] If $k = 0$, then the value of the bounded product is defined to be 1.

Corollary 3.14. *Suppose that $g_i(\overline{x})$, for $1 \leq i \leq n$, are primitive recursive functions and $R_i(\overline{x})$, for $1 \leq i < n$, are primitive recursive relations such that exactly one of the R_i holds for any tuple $\overline{x}$. Then the function*

$$f(\overline{x}) = \begin{cases} g_1(\overline{x}) & \text{if } R_1(\overline{x}) \text{ holds,} \\ g_2(\overline{x}) & \text{if } R_2(\overline{x}) \text{ holds,} \\ \vdots & \\ g_{n-1}(\overline{x}) & \text{if } R_{n-1}(\overline{x}) \text{ holds,} \\ g_n(\overline{x}) & \text{otherwise} \end{cases}$$

is primitive recursive.

Primitive recursive functions also allow *bounded* conditional loops. As we will see soon, what they do not allow is unbounded conditional loops!

Theorem 3.15. *Suppose that $g(u, \overline{y})$ is a primitive recursive function. Then the bounded search function*

$$f(x, \overline{y}) = \mu u_{u<x}[g(u, \overline{y}) = 0] = \begin{cases} \min\{u < x \mid g(u, \overline{y}) = 0\} & \text{if } u \text{ exists,} \\ x & \text{otherwise} \end{cases}$$

is primitive recursive.

Proof. First, we observe that the relation $g(u, \overline{y}) = 0$ is primitive recursive since

$$\chi_{g=0}(u, \overline{y}) = \overline{\text{sg}}(g(u, \overline{y})).$$

Next, we let

$$p(x, \overline{y}) = \Pi_{u \leq x} \chi_{\neg g=0}(u, \overline{y}).$$

Suppose that $\overline{b}$ is fixed and a is least such that $g(a, \overline{b}) = 0$. Then $p(x, \overline{b}) = 1$ for all $x < a$ and $p(x, \overline{b}) = 0$ for all $x \geq a$. Now observe that to compute $\mu u_{u<x}[g(u, \overline{y}) = 0]$, we simply add up $p(u, \overline{b})$ for all $u < x$,

$$f(x, \overline{y}) = \Sigma_{u<x} p(u, \overline{y}).$$

□

We can replace the relation $g(u, \overline{y}) = 0$ by any primitive recursive relation $R(u, \overline{y})$ since we have $R(u, \overline{y})$ if and only if $\overline{\text{sg}}(\chi_R(u, \overline{y})) = 0$. This gives us a more general version of the bounded search function

$$f(x, \overline{y}) = \mu u_{u<x}[R(u, \overline{y})] = \begin{cases} \min\{u < x \mid R(u, \overline{y}) \text{ holds}\} & \text{if } u \text{ exists,} \\ x & \text{otherwise.} \end{cases}$$

The general intuition one should have is that any function computable by an algorithm requiring only bounded searches is primitive recursive. Also, using composition, the search can be bounded by some primitive recursive function $h(x, \overline{y})$.

3.3. **Recursive functions.** Even before Gödel formally defined primitive recursive functions, mathematicians realized that such a class failed to capture algorithmic computability. In the 1920's Wilhelm Ackermann defined a 3-argument function computable by an explicit algorithmic procedure and showed that it was not primitive recursive. Ackermann's function is best expressed using Knuth's modern *up-arrow* operators. Let us define that $a \uparrow b = a^b$, $a \uparrow\uparrow b = \underbrace{a \uparrow (a \uparrow (\ldots \uparrow a))}_{b\text{-copies of } a}$, and more generally $a \uparrow^{n+1} b = \underbrace{a \uparrow (a \uparrow^n (\ldots \uparrow^n a))}_{b\text{-copies of } a}$.

Example 3.16. We have that $2 \uparrow 3 = 8$ and $2 \uparrow (2 \uparrow 2) = 2^{2^2} = 2^4 = 16$.

The up-arrow operators extend the pattern that multiplication is iterated addition and exponentiation is iterated multiplication by defining iterated exponentiation, iterated iterated exponentiation, and so on, ad infinitum. Ackermann defined his 3-argument function $a(n, m, p)$ as follows:

$$\begin{aligned} a(n, m, 0) &= n + m, \\ a(n, m, 1) &= nm, \\ a(n, m, 2) &= n^m, \\ a(n, m, p) &= n \uparrow^{p-1} (m + 1) \text{ for } p > 2. \end{aligned}$$

A modern reformulation of Ackermann's function, $A(n, m)$, which we will refer to as *the* Ackermann function, is defined by the following three recursion equations:

(1) $A(0, m) = m + 1$,
(2) $A(n + 1, 0) = A(n, 1)$,
(3) $A(n + 1, m + 1) = A(n, A(n + 1, m))$.

Let's compute $A(n, m)$ for $n < 4$.
By definition we have: $A(0, m) = m + 1$.

	0	1	2	3	4	...	m
0	1	2	3	4	5	...	$m + 1$

Next, we compute some values of $A(1, m)$:

$$A(1, 0) = A(0, 1) = 2,$$
$$A(1, 1) = A(0, A(1, 0)) = A(0, 2) = 3,$$
$$A(1, 2) = A(0, A(1, 1)) = A(0, 3) = 4,$$

and guess that general formula is: $A(1, m) = m + 2 = 2 + (m + 3) - 3$.

	0	1	2	3	4	...	m
0	1	2	3	4	5	...	$m + 1$
1	2	3	4	5	6	...	$m + 2$

Next, we compute some values of $A(2, m)$:

$$A(2, 0) = A(1, 1) = 3,$$
$$A(2, 1) = A(1, A(2, 0)) = A(1, 3) = 5,$$
$$A(2, 2) = A(1, A(2, 1)) = A(1, 5) = 7,$$

and guess that the general formula is: $A(2, m) = 2m + 3 = 2(m + 3) - 3$.

	0	1	2	3	4	...	m
0	1	2	3	4	5	...	$m + 1$
1	2	3	4	5	6	...	$m + 2$
2	3	5	7	9	11	...	$2m + 3$

Finally, we compute some values of $A(3, m)$:

$$A(3, 0) = A(2, 1) = 5,$$
$$A(3, 1) = A(2, A(3, 0)) = A(2, 5) = 13,$$

$$A(3,2) = A(2, A(3,1)) = A(2,13) = 29,$$

and guess that the general formula is: $A(3,m) = 2^{m+3} - 3$.

	0	1	2	3	4	...	m
0	1	2	3	4	5	...	$m+1$
1	2	3	4	5	6	...	$m+2$
2	3	5	7	9	11	...	$2m+3$
3	5	13	29	61	125	...	$2^{n+3}-3$

It can be shown that the function $A(4,m) = \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{m+3} - 3$ and more generally, that $A(n,m) = 2 \uparrow^{n-2} (m+3) - 3$.

Obviously, the Ackermann function grows very quickly! It is also intuitively clear that each computation terminates in finitely many steps. In the next theorem, we give a formal proof of this.

Theorem 3.17. *Every computation of the Ackermann function terminates after finitely many steps.*

Proof. Let us define a linear ordering on $\mathbb{N} \times \mathbb{N}$ by $(n,m) \leq (n',m')$ if $n < n'$ or $n = n'$ and $m \leq m'$. We shall argue that $\leq$ is a well-order, namely, that every nonempty subset of $\mathbb{N} \times \mathbb{N}$ has a least element under it. Suppose that A is a subset of $\mathbb{N} \times \mathbb{N}$. Let $n = a$ be the smallest such that $(n,m) \in A$ and let A' consist of all pairs of A with first coordinate a. Next, we let $m = b$ be the smallest such that (a,m) in A'. Then (a,b) is the least element of A. Now suppose towards a contradiction that there is $A(n,m)$ whose computation does not terminate in finitely many steps, and let $A(a,b)$ be such that (a,b) is least with this property. Note that a cannot be 0 and thus $a = a' + 1$. First, suppose that $b = 0$. Then $A(a,b) = A(a'+1, 0) = A(a', 1)$. But since $(a', 1) < (a,b)$, we have that the computation of $A(a',1)$ terminates in finitely many steps, contradicting that the computation of $A(a,b)$ does not terminate. So suppose that $b = b' + 1$. Then $A(a,b) = A(a'+1, b'+1) = A(a', A(a'+1, b'))$. But since $(a'+1, b') < (a'+1, b'+1)$, we have that the computation of $A(a'+1, b')$ terminates in finitely many steps. So let $c = A(a'+1, b')$. Now since $(a', c) < (a'+1, b'+1)$, we have that the computation of $A(a', c)$ terminates in finitely many steps, contradicting that the computation of $A(a,b)$ does not terminate. □

Despite the fact that the horizontal rows represent faster and faster growing functions, they all turn out be primitive recursive.

Theorem 3.18. *Each function* $A_n(m) = A(n,m)$ *is primitive recursive.*

Proof. We argue by induction on n. Clearly $A_0(m) = m+1$ is primitive recursive. Inductively, suppose that A_n is primitive recursive. Let $A_{n+1}(0) = c$. We argue that A_{n+1} is definable using primitive recursion from A_n. Observe that

$$A_{n+1}(m+1) = A(n+1, m+1) = A(n, A(n+1, m)) = A_n(A(n+1, m)) = A_n(A_{n+1}(m)).$$

Thus, we have

$$A_{n+1}(0) = c,$$
$$A_{n+1}(m+1) = A_n(A_{n+1}(m)).$$

□

But the diagonal function $AD(n) = A(n,n)$ is not primitive recursive!

Theorem 3.19. *The Ackermann function $A(n,m)$ is not primitive recursive and the diagonal Ackermann function $AD(n) = A(n,n)$ is not primitive recursive.*

The proof of Theorem 3.19 is a homework project.

To extend primitive recursive functions to recursive functions, we add one more closure operation: the unbounded search.

Closure Operation

(3) **μ-Operator**
Suppose that $g : \mathbb{N}^{m+1} \to \mathbb{N}$. We define a new function $f : \mathbb{N}^m \to \mathbb{N}$ using the μ-operator:

$$f(\overline{y}) = \mu x[g(x,\overline{y}) = 0] = \begin{cases} \min\{x \mid g(x,\overline{y}) = 0\} & \text{if exists,} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

A function $f : \mathbb{N}^n \to \mathbb{N}$ is said to be (partial) *recursive* if it can be built in finitely many steps from the basic functions using the operations of composition, primitive recursion, and the μ-operator. We shall say that a relation is *recursive* if its characteristic function is recursive. We shall also say that $A \subseteq \mathbb{N}$ is *recursive* if the relation $n \in A$ is recursive.

Because of the μ-operator, recursive functions are potentially partial, their domain may not be all of $\mathbb{N}$. This is the same behavior exhibited by a computer program using a conditional loop such as a while loop, which may not produce a value for a given input because it cannot escape the loop. Indeed, a function $g(x,\overline{y})$ to which we apply the μ-operator may already be partial and so we need to be more careful in defining what it means for x to be the minimum value such that $g(x,\overline{y}) = 0$. Let us define more precisely that $\mu x[g(x,\overline{y}) = 0] = z$ if and only if $g(x,\overline{y})$ is *defined for all* $x \leq z$ and z is least such that $g(z,\overline{y}) = 0$. We will show in Section 7 that the alternatively defined μ-operator, where we would allow $g(x,\overline{y})$ to be undefined for some $x < z$, does not always yield a recursive function.

We can replace the relation $g(x,\overline{y}) = 0$ by any recursive relation $R(x,\overline{y})$ since we have $R(x,\overline{y})$ if and only if $\overline{\text{sg}}(\chi_R(x,\overline{y})) = 0$.

We will see in the next section that the Ackermann function is recursive.

3.4. **Recursive functions and definability over $\mathbb{N}$.** The next several theorems provide a characterization of recursive functions and relations in terms of definability over $\mathbb{N}$. Let us say that a function or relation on the natural numbers is $\Sigma_n^{\mathbb{N}}$ or $\Pi_n^{\mathbb{N}}$ if it is definable over $\mathbb{N}$ by a Σ_n-formula or a Π_n-formula respectively. Let us say that a function or a relation is $\Delta_n^{\mathbb{N}}$ if it is defined by a formula that is $\Delta_n(\mathbb{N})$.

Theorem 3.20. *$\Delta_0^{\mathbb{N}}$-relations are primitive recursive.*

Proof. We argue by induction on the complexity of Δ_0-formulas. Let $t(\overline{x})$ and $s(\overline{x})$ be L_A-terms. Since $+$ and $\cdot$ are primitive recursive and primitive recursive functions are closed under composition, it follows that the functions $f(\overline{x}) = t(\overline{x})$ and $g(\overline{x}) = s(\overline{x})$ are primitive recursive. Since the relations $=$ and $<$ are primitive recursive, it follows that the relations $t = s$ and $t < s$ are primitive recursive. Thus, relations defined by atomic formulas are primitive recursive. So assume inductively that relations defined by φ and ψ are primitive recursive. Then, by Theorem 3.12, the relations $\varphi \wedge \psi$, $\varphi \vee \psi$, and $\neg\varphi$ are primitive recursive as well. Now suppose

that the relation defined by $\varphi(x, y, \overline{z})$ is primitive recursive and let $\chi_\varphi(x, y, \overline{z})$ be its characteristic function. Consider the relation defined by

$$\psi(y, \overline{z}) := \exists x < y\, \varphi(x, y, \overline{z}).$$

Then the characteristic function of ψ is

$$\chi_\psi = \mathrm{sg}(\Sigma_{x<y}\chi_\varphi(x, y, \overline{z})).$$

□

At this point, it is easy to fall into the trap of thinking that every $\Delta_0^{\mathbb{N}}$-function is primitive recursive. But while we just showed that the graph of every Δ_0-function viewed as a relation is primitive recursive, the function itself need not be. The difference being that an output might be very difficult to compute and at the same time it can be very easy, when already given the output, to verify that it is the correct one. For instance, the Ackermann function is not primitive recursive but it is primitive recursive to check whether a particular value is the correct output of the Ackermann function given a sequence containing all Ackermann computations containing needed to compute it. Let us say that a sequence *s witnesses an Ackermann computation for* (l, p) if for every Ackermann computation $A(n, m)$ required in the computation of $A(l, p)$, we have that $[s]_{\langle n,m\rangle} = A(n, m)$ (where $\langle n, m\rangle$ is the result of applying Cantor's pairing function to (n, m)) and if $A(a, b)$ is not required in the computation of $A(l, p)$, then $[s]_{\langle a,b\rangle}$ is allowed to be either $A(a, b)$ or 0. However, if $[s]_{\langle a,b\rangle} = A(a, b)$, then the sequence must have all $A(n, m)$ needed to compute it. More precisely, we define that a sequence s witnesses the Ackermann computation for (l, p) if:

$$\begin{array}{l} \mathrm{len}(s) = k \wedge \exists x < k(x = \langle l, p\rangle \wedge [s]_x \neq 0) \wedge \\ \forall x < k\, [s]_x \neq 0 \to \\ \quad \forall n, m \leq x \left(\begin{array}{l} x = \langle 0, m\rangle \to [s]_x = m + 1 \wedge \\ x = \langle n + 1, 0\rangle \to \exists z < k(z = \langle n, 1\rangle \wedge [s]_x = [s]_z) \wedge \\ x = \langle n + 1, m + 1\rangle \to \exists z, w < k(z = \langle n + 1, m\rangle \wedge w = \langle n, [s]_z\rangle \wedge [s]_x = [s]_w) \end{array} \right). \end{array}$$

Now consider the function $f(n, m) = y$ which on input (n, m) outputs the pair $\langle s, a\rangle$, where s is the smallest sequence witnessing an Ackermann computation for $\langle n, m\rangle$ and $a = [s]_{\langle n,m\rangle}$. The function f is defined by the Δ_0-formula

$$\varphi(n, m, y) := \exists s, a \leq y \left(\begin{array}{l} y = \langle s, a\rangle \wedge \\ s \text{ is least witnessing the Ackermann computation for } (m, n) \wedge \\ a = [s]_{\langle n,m\rangle} \end{array} \right).$$

But if f was primitive recursive, then we would be able to compute the Ackermann function using a bounded search to find a such that $f(n, m) = \langle s, a\rangle$. To make this precise, it remains to argue that function the $g(x) = m$ where $x = \langle n, m\rangle$ is primitive recursive. First, we observe that the graph of $g(x)$ is Δ_0-definable by the formula

$$\varphi(x, m) := \exists n \leq x\, x = \langle n, m\rangle$$

and hence primitive recursive. Thus,

$$g(x) := \mu_{m<b}[\varphi(x, m)]$$

is primitive recursive as well. This completes the argument that $f(n, m)$ is $\Delta_0^{\mathbb{N}}$ and not primitive recursive. It is true though that every $\Delta_0^{\mathbb{N}}$-function and indeed every $\Sigma_1^{\mathbb{N}}$-function must be recursive. Remarkably, the $\Sigma_1^{\mathbb{N}}$-functions are precisely the

recursive functions! This fact provides a surprising and crucial connection between algorithmic computability and first-order definability over $\mathbb{N}$.

Theorem 3.21. *Every $\Sigma_1^{\mathbb{N}}$-function is recursive.*

Proof. Suppose that $f(\overline{x})$ is defined by the Σ_1-formula $\varphi(\overline{x},y) := \exists z\,\psi(\overline{x},z,y)$, where ψ is Δ_0. Let

$$\psi'(\overline{x},w) := \exists z,y\leq w\, w=\langle z,y\rangle\wedge\psi(\overline{x},z,y).$$

Then ψ' is Δ_0 and hence primitive recursive by Theorem 3.20. We define $g(\overline{x})$ by

$$g(\overline{x})=\mu w[\psi'(\overline{x},w)].$$

Thus, g is recursive and we have

$$g(\overline{x})=\begin{cases}\langle z,y\rangle \text{ such that } \psi(\overline{x},z,y) & \text{if exists,}\\ \text{undefined} & \text{otherwise.}\end{cases}$$

It remains to retrieve the value y from the pair $\langle z,y\rangle$ and we already showed about that this is even primitive recursive. □

Theorem 3.22. *Every recursive function is $\Sigma_1^{\mathbb{N}}$.*

Proof. The zero function $Z(x)$ is defined by $\varphi(x,y) := y=0$. The projection function $P_i^n(x_1,\ldots,x_n)$ is defined by $\varphi_i^n(x_1,\ldots,x_n,y) := y=x_i$. The successor function $S(x)$ is defined by $\varphi(x,y):=y=x+1$. Thus, the basic functions are $\Sigma_1^{\mathbb{N}}$. Now suppose that the functions $g_i(\overline{x})$, for $1\leq i\leq n$, are defined by Σ_1-formulas $\varphi_i(\overline{x},y)$ and the function $h(x_1,\ldots,x_n)$ is defined by the Σ_1-formula $\psi(x_1,\ldots,x_n,y)$. Then the composition function $h(g_1(\overline{x}),\ldots,g_n(\overline{x}))$ is defined by the formula

$$\delta(\overline{x},y) := \exists y_1,\ldots,y_n(\varphi_1(\overline{x},y_1)\wedge\cdots\varphi_n(\overline{x},y_n)\wedge\psi(y_1,\ldots,y_n,y))$$

Next, we check that a function obtained by primitive recursion from $\Sigma_1^{\mathbb{N}}$-functions is $\Sigma_1^{\mathbb{N}}$. This done using, once again, the existence of computation witnessing sequences. So suppose that the function $g(\overline{x})$ is defined by the Σ_1-formula $\varphi(\overline{x},y)$ and the function $h(\overline{x},z,w)$ is defined by the Σ_1-formula $\psi(\overline{x},z,w,y)$. The recursively defined function $f(\overline{x},z)$ such that

$$f(\overline{x},0)=g(\overline{x}),$$

$$f(\overline{x},z+1)=h(\overline{x},z,f(\overline{x},z))$$

is defined by the formula

$$\delta(\overline{x},z,y) := \exists s\begin{pmatrix}\exists y_1(\varphi(\overline{x},y_1)\wedge y_1=(s)_0)\wedge\\ \forall x<z(\exists y_2(\psi(\overline{x},x,(s)_x,y_2)\wedge(s)_{x+1}=y_2))\wedge\\ (s)_z=y\end{pmatrix}$$

Finally, we check that a function obtained by the μ-operator from $\Sigma_1^{\mathbb{N}}$-functions is $\Sigma_1^{\mathbb{N}}$. First, observe that if a function $f(\overline{x})$ is defined by the Σ_1-formula $\varphi(\overline{x},y):=\exists z\,\psi(\overline{x},z,y)$, where ψ is Δ_0, then it is also defined by the Π_1-formula $\varphi'(\overline{x},y):=\forall z\forall w(\psi(\overline{x},z,w)\to y=w)$. Thus any function that is $\Sigma_1^{\mathbb{N}}$ is also $\Pi_1^{\mathbb{N}}$ and hence $\Delta_1^{\mathbb{N}}$. Now suppose that $g(x,\overline{y})$ is defined by the Σ_1-formula $\varphi(x,\overline{y},z)$ and the function $f(\overline{y})$ is obtained from g via the μ-operator

$$f(\overline{y})=\mu x[g(x,\overline{y})=0]=\begin{cases}\min\{x\mid g(x,\overline{y})=0\} & \text{if exists,}\\ \text{undefined} & \text{otherwise.}\end{cases}$$

The function $f(\overline{y})$ is defined by the formula

$$\delta(\overline{y}, w) := \varphi(w, \overline{y}, 0) \wedge \forall u < w \, \neg\varphi'(u, \overline{y}, 0).$$

where φ' is the Π_1-definition of g. □

Corollary 3.23. *A function is recursive if and only if it is* $\Delta_1^{\mathbb{N}}$.

Note that, while every $\Sigma_1^{\mathbb{N}}$-function is $\Pi_1^{\mathbb{N}}$, the converse does not hold, namely, there are $\Pi_1^{\mathbb{N}}$-functions which are not $\Sigma_1^{\mathbb{N}}$. We will encounter an example of such a function in Section 7.

Example 3.24. The Ackermann function is recursive since it is defined by the Σ_1-formula $\varphi(n, m, y) :=$

$$\exists s, k \left(\begin{array}{l} \text{len}(s) = k \wedge \\ \exists x < k(x = \langle n, m \rangle \wedge [s]_x = y \wedge \\ s \text{ witnesses an Ackermann computation for } x) \end{array} \right).$$

Theorem 3.25. *A relation is recursive if and only if it is* $\Delta_1^{\mathbb{N}}$.

Proof. Suppose that $R(\overline{x})$ is a recursive relation. Then by Theorem 3.22 and our earlier observation, its characteristic function $\chi_R(\overline{x}, y)$ is defined by a Σ_1-formula $\varphi(\overline{x}, y)$ and by a Π_1-formula $\varphi'(\overline{x}, y)$. Thus, $R(\overline{x})$ is defined by the Σ_1-formula $\varphi(\overline{x}, 1)$ and the Π_1-formula $\varphi'(\overline{x}, 1)$.

Now suppose that a relation $R(\overline{x})$ is defined by the Σ_1-formula $\varphi(\overline{x})$ and the Π_1-formula $\psi(\overline{x})$. The characteristic function $\chi_R(\overline{x}, y)$ is defined by

$$(\varphi(z, \overline{x}) \wedge y = 1) \vee (\neg\psi(\overline{x}) \wedge y = 0).$$

Thus, $\chi_R(\overline{x})$ is recursive by Theorem 3.21 and hence $R(\overline{x})$ is recursive. □

Unlike $\Sigma_1^{\mathbb{N}}$-functions, there are $\Sigma_1^{\mathbb{N}}$-relations that are not $\Pi_1^{\mathbb{N}}$. Such relations are called *recursively enumerable*, and we will study them in great detail in Section 7.

3.5. **Recursive functions and models of** PA. Another of the remarkable facts about recursive functions (and relations) is that every model of PA (even PA^-) agrees with $\mathbb{N}$ on most of the information about them. This statement is made precise using the definitions below.

We say that a relation $R(x_1, \ldots, x_n)$ on $\mathbb{N}^n$ is *represented* in some theory T of L_A if there is an L_A-formula $\varphi(x_1, \ldots, x_n)$ such that for all $m_1, \ldots, m_n \in \mathbb{N}$, we have

(1) $R(m_1, \ldots, m_n) \leftrightarrow T \vdash \varphi(\underline{m}_1, \ldots, \underline{m}_n)$,
(2) $\neg R(m_1, \ldots, m_n) \leftrightarrow T \vdash \neg\varphi(\underline{m}_1, \ldots, \underline{m}_n)$.

We say that a function $f : \mathbb{N}^n \to \mathbb{N}$ is *represented* in some theory T of L_A if there is an L_A-formula $\varphi(x_1, \ldots, x_n, y)$ such that for all $m, m_1, \ldots, m_n \in \mathbb{N}$, we have

(1) $f(m_1, \ldots, m_n) = m \to T \vdash \varphi(\underline{m}_1, \ldots, \underline{m}_n, \underline{m})$,
(2) $T \vdash \exists! y \, \varphi(\underline{m}_1, \ldots, \underline{m}_n, y)$[12].

Theorem 3.26. *Every total recursive function is represented in* PA^-.

[12] The notation $\exists! y(\ldots)$ is an abbreviation for $\exists y((\ldots) \wedge \forall z(z \neq y \to \neg(\ldots)))$, which expresses that y is unique.

Proof. Suppose that $f : \mathbb{N}^k \to \mathbb{N}$ is a total recursive function. By Theorem 3.22, $f(\overline{x})$ is defined over $\mathbb{N}$ by some Σ_1-formula

$$\delta(\overline{x}, y) := \exists z \varphi(\overline{x}, y, z),$$

where $\varphi(\overline{x}, y, z)$ is Δ_0. Let $\psi(\overline{x}, y, z)$ be the Δ_0-formula

$$\varphi(\overline{x}, y, z) \wedge \forall u, v \leq y + z(u + v < y + z \to \neg\varphi(\overline{x}, u, v)).$$

The formula $\psi(\overline{x}, y, z)$ asserts that the pair y, z is, in some sense, the smallest witnessing $\varphi(\overline{x}, y, z)$. Now we shall argue that $f(\overline{x})$ is represented in PA^- by the formula

$$\gamma(\overline{x}, y) := \exists z\, \psi(\overline{x}, y, z).$$

Suppose that $f(\overline{n}) = m$. Since f is defined by $\delta(\overline{x}, y)$, there is $z \in \mathbb{N}$ such that $\mathbb{N} \models \varphi(\overline{n}, m, z)$ and so we let l be the least such z. If there are $u, v \leq m + l$ such that $\varphi(\overline{x}, u, v)$, then it must be that $u = m$ and so $u < l$. But this cannot be by our minimality assumption on l. Thus, $\mathbb{N} \models \psi(\overline{n}, m, l)$. Now suppose that $M \models \text{PA}^-$. Since $\mathbb{N} \prec_{\Delta_0} M$, we have that $M \models \psi(\overline{n}, m, l)$, and so $M \models \gamma(\overline{n}, m)$. It remains to argue that M cannot satisfy $\gamma(\overline{n}, a)$ for some other $a \in M$. So suppose that $M \models \gamma(\overline{n}, a)$. Thus, for some $b \in M$, we have that

$$M \models \varphi(\overline{n}, a, b) \wedge \forall u, v \leq a + b(u + v < a + b \to \neg\varphi(\overline{n}, u, v)).$$

Observe that we cannot have $m + l < a + b$ since $M \models \varphi(\overline{n}, m, l)$, and so we must have $a + b \leq m + l$. But then both $a, b \in \mathbb{N}$, and so by Δ_0-elementarity, we have $\mathbb{N} \models \varphi(\overline{n}, a, b)$. Thus, $\mathbb{N} \models \delta(\overline{n}, a)$, from which it follows that $a = m$, as desired. □

Theorem 3.27. *Every recursive relation is represented in* PA^-.

Proof. Suppose that $R(\overline{x})$ is recursive. Then its characteristic function $\chi_R(\overline{x})$ is total recursive and therefore, by Theorem 3.26, it is represented in PA^- by a formula $\varphi(\overline{x}, y)$. We argue that R is represented by the formula $\varphi(\overline{x}, 1)$. If $R(\overline{n})$ holds, then $\chi_R(\overline{n}) = 1$ and so $\text{PA}^- \vdash \varphi(\overline{n}, 1)$. If $R(\overline{n})$ fails, then $\chi_R(\overline{n}) = 0$ and so $\text{PA}^- \vdash \varphi(\overline{n}, 0)$. Also, $\text{PA}^- \vdash \exists! y\, \varphi(\overline{n}, y)$ and so it must be that $\text{PA}^- \vdash \neg\varphi(\overline{n}, 1)$. □

Recall from Example 2.36 of Section 2, that all sets definable over $\mathbb{N}$ by a $\Delta_1(\text{PA})$-formula are in the standard system of every nonstandard model of PA. Now we can extend that result to show that every recursive ($\Delta_1^{\mathbb{N}}$) set is in the standard system of every nonstandard model of PA.

Theorem 3.28. *Suppose that* $M \models \text{PA}$ *is nonstandard. Then every recursive set is in* $SSy(M)$.

Proof. Suppose $A \subseteq \mathbb{N}$ is recursive. By Theorem 3.27, there is a formula $\varphi(x)$ such that

(1) $n \in A \to \mathbb{N} \models \varphi(n) \to \text{PA}^- \vdash \varphi(n)$,
(2) $n \notin A \to \mathbb{N} \models \neg\varphi(n) \to \text{PA}^- \vdash \neg\varphi(n)$.

We will show that $A = \{n \in \mathbb{N} \mid M \models \varphi(n)\}$. Suppose that $n \in A$, then $\mathbb{N} \models \varphi(n)$ and so $M \models \varphi(n)$. Now suppose that $n \in \mathbb{N}$ and $M \models \varphi(n)$. It must be the case that $n \in A$, since otherwise we would have $M \models \neg\varphi(n)$. It follows, as desired, that $A \in SSy(M)$. □

Question 3.29. Must a standard system contain anything more than the recursive sets?

We will argue in Section 4 that there is a recursive binary tree that has no recursive branches. It will follow that every standard system must contain some non-recursive set.

3.6. **Homework.**

Question 3.1. Show that $\text{Exp}(m, n) = m^n$ and $\text{Fact}(n) = n!$ are primitive recursive.

Question 3.2. Show that bounded sums and bounded products are primitive recursive.

Question 3.3. Show that the Ackerman function is not primitive recursive and the diagonal Ackermann function $DA(n) = A(n, n)$ is not primitive recursive.

4. The First Incompleteness Theorem

Some people are always critical of vague statements. I tend rather to be critical of precise statements; they are the only ones which can correctly be labeled 'wrong'.
–**Raymond Smullyan**

4.1. **The Beginning and the End of Hilbert's Program.** At the dawn of the 20th-century formal mathematics flourished. In the preceding centuries, mathematical concepts had became simultaneously more complex and abstract, reaching a point where their validity could no longer be justified on the intuitive ground that they mirrored the physical universe. Formal mathematics was born toward the end of the 19th-century, partly out of new developments in logic and the study of collections, and partly out of the need to provide a foundation on which abstract mathematics, divorced from physics, could safely rest. Gottlob Frege had expanded the scope of logical reasoning for the first time since it was introduced by Aristotle more than two millennia before. In his 1879 *Begriffsschrift*, Frege invented predicate logic by introducing quantification into Aristotle's propositional logic. Building on Frege's earlier work but eliminating second-order quantifiers, logicians arrived at first-order logic, a powerful formal language for mathematical concepts. Reinterpreted in first-order logic, Peano's axioms became the accepted axiomatization of number theory. In 1904, in an attempt to justify Cantor's well-ordering principle, Zermelo proposed an axiomatization for set theory, a potential foundation of all known mathematics. Mathematicians were optimistic that well-chosen axiom systems expressed in first-order logic could provide an unshakable foundation for their respective fields of mathematics. They believed that formal metamathematical arguments could be devised to show the consistency of these axioms systems, making a mathematical statement that was derived by formal logical reasoning from such a collection of axioms incontrovertibly valid. They assumed that each such axiom system, if properly chosen, could be shown to be both *consistent* and *complete*! Foremost among the enthusiasts was David Hilbert. In his famous 1900 International Congress of Mathematicians address (in which he introduced the list of 23 problems), Hilbert proclaimed:

> *When we are engaged in investigating the foundations of a science, we must set up a system of axioms which contains an exact and complete description of the relations subsisting between the elementary ideas of that science. The axioms so set up are at the same time the definitions of those elementary ideas; and no statement within the realm of the science whose foundation we are testing is held to be correct unless it can be*

> *derived from those axioms by means of a finite number of logical steps.*[...] *But above all I wish to designate the following as the most important among the numerous questions which can be asked with regard to the axioms: To prove that they are not contradictory, that is, that a definite number of logical steps based upon them can never lead to contradictory results.*

The two goals became known as *Hilbert's Program* and initial progress gave good reason for optimism. In 1905, Hilbert observed that propositional logic is consistent.

Theorem 4.1. *Propositional logic is consistent.*

Proof. The axioms of propositional logic and every statement provable from them share the property that they evaluate as true under every truth assignment to the propositional variables. Since whenever a propositional statement φ evaluates to true, its negation $\neg\varphi$ must evaluate to false, it is not possible that both φ and $\neg\varphi$ are provable from the axioms of propositional logic. □

Coming tantalizingly close to Hilbert's goal, Mojżesz Presburger showed in 1929 that Peano Arithmetic without multiplication, an axiom system which became known as Presburger arithmetic, is consistent and complete. The language of Presburger Arithmetic is L_A without multiplication.

Presburger Arithmetic

Addition Axioms

Ax1: $\forall x(\neg x+1=0)$
Ax2: $\forall x\forall y(x+1=y+1\to x=y)$
Ax3: $\forall x(x+0=x)$
Ax4: $\forall x\forall y((x+y)+1=y+(x+1))$

Induction Scheme

$$\forall\overline{y}(\varphi(0,\overline{y})\wedge\forall x(\varphi(x,\overline{y})\to\varphi(x+1,\overline{y}))\to\forall x\varphi(x,\overline{y}))$$

Theorem 4.2. *Presburger Arithmetic is consistent and complete.*

The proof of Presburger's theorem involves the technique of quantifier elimination and is beyond the scope of these notes.

Then in 1930, Kurt Gödel ended all hope for Hilbert's program with his First and Second Incompleteness theorems. Gödel showed:

Theorem 4.3 (First Incompleteness Theorem)**.** *Peano Arithmetic is not complete. Moreover, no 'reasonable' axiom system extending* PA^- *can ever be complete.*

Theorem 4.4 (Second Incompleteness Theorem)**.** *There cannot be a 'finitary' proof of the consistency of Peano Arithmetic.*

Formal statements of the incompleteness theorems will have to wait until we introduce the necessary machinery in the next few sections.

4.2. **Richard's Paradox.** Gödel credited the idea behind the proof of the First Incompleteness Theorem to Richard's Paradox, described by the French mathematician Jules Richard in 1905.

Let us consider all English language expressions that unambiguously define a property of natural numbers. Here are some examples:

(1) x is even.
(2) x is prime.
(3) x is a sum of two squares.
(4) Goldbach's Conjecture[13] fails above x.
(5) x is definable using 25 words or less.

Each such definition, call it $\varphi(x)$, can be assigned a unique numerical code $\ulcorner\varphi(x)\urcorner$ using some agreed upon reasonable coding. For instance, we can concatenate the ASCII codes of the individual characters in the expression and let the resulting (very large) number be its code. Expressions (1) and (2), from above, would end up with the following codes.

$\ulcorner x \text{ is even}\urcorner = 120032105115032101118101110$
$\ulcorner x \text{ is prime}\urcorner = 120032105115032112114105109101$

Now we make a crucial observation.

$\ulcorner x$ is even$\urcorner$ is **even**
$\ulcorner x$ is prime$\urcorner$ is **not** prime (why?)

Generalizing this phenomena, we can ask of any property whether it is true of its own code, whether $\varphi(\ulcorner\varphi(x)\urcorner)$ holds. By coding properties as numbers and evaluating them on their own codes, we have endowed our properties with the ability to self-reference and such abilities hardly ever lead to any good. Indeed, there is something suspiciously extra-ordinary about a property that holds of its own code. Thus, we define that a number n is *ordinary* if

(1) $n = \ulcorner\varphi(x)\urcorner$ for some property $\varphi(x)$,
(2) $\varphi(\ulcorner\varphi(x)\urcorner)$ does **not** hold.

It is clear (isn't it?) that being ordinary is one of Richard's unambiguously defined properties of numbers. Thus, we let $m = \ulcorner x$ is ordinary$\urcorner$ (where 'x is ordinary' is an abbreviation for the rather long English expression describing ordinariness).

Question 4.5. Is m ordinary?

We have that

$$\begin{aligned} &m \text{ is ordinary} &&\leftrightarrow \\ &\text{`}\ulcorner x \text{ is ordinary}\urcorner \text{ is ordinary'} &&\leftrightarrow \\ &m \text{ is not ordinary.} \end{aligned}$$

This is Richard's Paradox!

An obvious objection to Richard's argument is that there is no way to make precise Richard's notion of an unambiguously defined property for a non-formal language such as English. But on a deeper level, Richard's paradox exploits the same phenomena which made possible Russell's paradox of naive set theory, as well as its ancient antecedent, the Liar Paradox. Using the trick of coding properties by numbers, the purported property of ordinariness is defined in terms of the collection of all properties and therefore inadvertently references itself. Because the definition must refer to itself for its evaluation, the evaluation cannot be always carried out. Thus ordinariness, while being a perfectly valid property of properties of numbers, is not a property of numbers, in the sense that it can be evaluated for every number to yield a yes or no answer. As simply illustrated in the case of the sentence 'this sentence is false', English language allows statements which reference themselves.

[13] Goldbach's conjecture states that every even number greater than 2 can be expressed as the sum of two primes.

Because of this, using coding, we can sneak in as properties of numbers, meta-mathematical assertions: properties of properties of numbers, such as ordinariness. Any consistent formal system should prevent such inter-level reference. So what happens if we replace English by the formal language of first-order logic? What if we could assign numerical codes to first-order assertions about numbers in such a way that the model $\mathbb{N}$, in particular, and nonstandard models of PA, in general, could be made to reason about them? Then properties of properties of numbers become properties of numbers and numerous exciting consequences follow.

4.3. **The arithmetization of first-order logic.** Gödel introduced the β-function in his First Incompleteness Theorem paper, but he chose not to use it for subsequent coding. Instead Gödel used the unique primes decomposition coding, which we encountered in Section 2. Once in possession of the β-function, it is not difficult (but cumbersome) to show that the primes decomposition coding machinery is definable and indeed recursive ($\Delta_1(\mathbb{N})$). Recall that in this coding method, we we code a sequence $\langle a_0, \ldots, a_{n-1}\rangle$ by the product $p_0^{a_0+1} \cdot p_1^{a_1+1} \cdot \ldots \cdot p_{n-1}^{a_{n-1}+1}$, where p_i denotes the ith prime. The primes coding has the emotional advantage that it is trivial to compute the code of a given sequence. With the β-function coding, you convince yourself that there is the requisite code, but you would not venture to actually compute what it is. This disadvantage though is purely emotional because there is no reason for us to ever have to produce the physical code of a sequence. Indeed, one should not worry about the details of the coding beyond the knowledge that it has properties (1)-(3) of Theorem 2.32 and that its decoding properties are recursive. In the case of the β-function, they are even Δ_0. In these notes, I am willing to take the emotional discomfort of not being able to readily produce a code over having to prove that the primes decomposition coding machinery is recursive. Thus, unlike Gödel, we will stick here with the old and familiar β-function coding.

As in the sneak preview of Section 2, we begin by assigning number codes to the alphabet of L_A in some reasonable manner.

L_A-symbol	Code
0	0
1	1
$+$	2
$\cdot$	3
$<$	4
$=$	5
$\wedge$	6
$\vee$	7
$\neg$	8
$\exists$	9
$\forall$	10
(	11
)	12
x_i	$\langle 13, i\rangle$

We may now assign to every string S of L_A-symbols the unique number code, which is the least number coding, via the β-function, the string of numbers corresponding to the elements of S. We call this unique code, the Gödel-number of S and denote

it by $\ulcorner S \urcorner$. The set of all Gödel-numbers is defined by the formula

$$\mathrm{Gn}(x) := \mathrm{Seq}(x) \wedge \forall i < \mathrm{len}(x)([x]_i \leq 12 \vee \exists z \leq x\, [x]_i = \langle 13, z\rangle).$$

Via the coding, we will associate L_A-formulas with their Gödel-numbers, L_A-theories with sets of Gödel-numbers, and proofs from an L_A-theory with numbers that code a sequence of Gödel-numbers. Our goal now is to make $\mathbb{N}$ and, more generally, any model of PA^- be able to understand first-order concepts such as terms, formulas, and proofs. This will come down to showing that the relations and functions which recognize and manipulate these concepts are recursive. By Theorem 3.26, recursiveness guarantees that all models of PA^- agree on the truth value of formulas with natural number parameters expressing these relations and functions. First, though, we need to define some crucial operations on sequences.

The sequence concatenation operation $z = x\hat{\ }y$ is defined by the formula

$$\begin{array}{l}\mathrm{Seq}(x) \wedge \mathrm{Seq}(y) \wedge \mathrm{Seq}(z) \wedge \mathrm{len}(z) = \mathrm{len}(x) + \mathrm{len}(y) \wedge \\ \forall i < \mathrm{len}(x)\, [z]_i = [x]_i \wedge \forall j < \mathrm{len}(y)\, [z]_{\mathrm{len}(x)+j} = [y]_j.\end{array}$$

The sequence restriction operation $z = x \upharpoonright y$ is defined by the formula

$$\mathrm{Seq}(x) \wedge \mathrm{Seq}(z) \wedge \mathrm{len}(z) = y \wedge \forall i < \mathrm{len}(z)\, [z]_i = [x]_i.$$

Given two sequences x and y, we shall use the notation $x \subseteq y$ if the sequence x appears as a contiguous block inside the sequence y.

Example 4.6. If $s = \langle 5, 1, 7\rangle$ and $t = \langle 17, 3, 5, 1, 7, 11\rangle$, then $s \subseteq t$.

The subsequence relation $x \subseteq y$ is defined by

$$\begin{array}{l}\mathrm{Seq}(x) \wedge \mathrm{Seq}(y) \wedge \mathrm{len}(y) \geq \mathrm{len}(x) \wedge \\ \exists i \leq \mathrm{len}(y) - \mathrm{len}(x)(\forall j < \mathrm{len}(x)([x]_j = [y]_{i+j})).\end{array}$$

An important property of the subsequence relation is that quantifiers of the form $\exists x \subseteq y$ and $\forall x \subseteq y$ do not add to the complexity of the formula following them.

Theorem 4.7. *If φ is a Σ_n-formula or a Π_n-formula, then the formulas $\exists x \subseteq y\varphi$ and $\forall x \subseteq y\varphi$ are both $\Sigma_n(\mathrm{PA})$ or $\Pi_n(\mathrm{PA})$-respectively.*

The proof is left as a homework exercise. Notice that all three sequence operations we just defined are Δ_0. We are now ready to define relations recognizing the grammar of first-order logic.

In the same way that we verify the result of an exponentiation computation by examining a sequence witnessing all its steps, we determine whether a string of L_A-symbols is a term by examining a 'term-building' sequence witnessing its construction from the basic terms. The relation termseq(x), expressing that s is a term-building sequence, is defined by the formula

$$\forall i < \mathrm{len}(s) \left(\begin{array}{l} [s]_i = \ulcorner 0 \urcorner \vee [s]_i = \ulcorner 1 \urcorner \vee \exists j \leq s([s]_i = \ulcorner x_j \urcorner) \vee \\ \exists j, k < i([s]_i = \ulcorner ([s]_j + [s]_k) \urcorner \vee [s]_i = \ulcorner ([s]_j \cdot [s]_k) \urcorner) \end{array} \right).$$

Note the expression $\ulcorner ([s]_j + [s]_k) \urcorner$ is used as a shorthand for the rather unreadable $\ulcorner (\urcorner \hat{\ } [s]_j \hat{\ } \ulcorner + \urcorner \hat{\ } [s]_k \hat{\ } \ulcorner) \urcorner$. We will continue to use like abbreviations in the future without any further notice. Clearly a string of L_A-symbols is a term if it appears on some term-building sequence. The relation Term(x), expressing that x is the Gödel-number of an L_A-term, is defined by the formula

$$\exists s(\mathrm{termseq}(s) \wedge \exists i < \mathrm{len}(s)\, [s]_i = x).$$

This gives us a Σ_1-definition for the Gödel-numbers of terms and now we would like to obtain a Π_1-definition as well. For this purpose, observe that x is the Gödel-number of a term if and only if it appears on *every* term building sequence consisting exactly of the sub-terms of x. More precisely, we define the relation seqforterm(s,x), expressing that s consists precisely of the sub-terms of x, by the formula

$$\begin{aligned}&\text{termseq}(s)\wedge\forall i<\text{len}(s)\,[s]_i\subseteq x\wedge\\&\forall y((y\subseteq x\wedge\text{termseq}(s\,\hat{}\,\langle y\rangle))\to\exists i<\text{len}(s)\,[s]_i=y).\end{aligned}$$

The relation seqforterm(s,x) allows us to define Term(x) by the Π_1-formula

$$\forall s(\text{seqforterm}(s,x)\to\exists i<\text{len}(s)\,[s]_i=x).$$

At this point, a few technical lemmas are required to show that sequences consisting of all sub-terms of a string x actually exist and that any two such sequences will agree on their elements modulo the order in which they are arranged. The lemmas do not just hold true over $\mathbb{N}$, but are provable in PA. It follows that the relation Term(x) is $\Delta_1(\text{PA})$, and, in particular, recursive. More so, PA proves that elements satisfying Term(x) behave precisely as we expect terms to behave.

Theorem 4.8. PA *proves*

$$\begin{aligned}\text{Term}(t)\leftrightarrow\quad &t=\ulcorner 0\urcorner\vee t=\ulcorner 1\urcorner\vee\exists j\leq t\,t=\ulcorner x_j\urcorner\vee\\&\exists r,s\subseteq t(\text{Term}(r)\wedge\text{Term}(s)\wedge(t=\ulcorner(r+s)\urcorner\vee t=(\ulcorner r\cdot s)\urcorner).\end{aligned}$$

Proofs of these results are straightforward argument using induction inside a model of PA and will be omitted here. At this stage the reader should believe herself expert enough to find trudging through this level of mind numbing detail to be beneath her. Readers who are fond of mind numbing details are encouraged to consult [Kay91]. If $M\models\text{PA}$ is nonstandard, then it will satisfy Term(t) of exactly those natural numbers that are Gödel-numbers of L_A-terms. But Term(t) will also hold of nonstandard elements representing 'nonstandard' terms such as $\underbrace{1+\cdots+1}_{a\text{ times}}$ for a nonstandard $a\in M$.

We determine whether string of symbols is an L_A-formula by producing a witnessing 'formula-building' sequence. The relation formseq(s), expressing that s is a formula-building sequence, is defined by the formula

$$\forall i<\text{len}(s)\left(\begin{array}{l}\exists u,v\leq s(\text{Term}(u)\wedge\text{Term}(v)\wedge([s]_i=\ulcorner(u=v)\urcorner\vee[s]_i=\ulcorner(u<v)\urcorner)\vee\\\exists j,k<i([s]_i=\ulcorner([s]_j\vee[s]_k)\urcorner\vee[s]_i=\ulcorner([s]_j\wedge[s]_k)\urcorner)\vee\\\exists j<i([s]_i=\ulcorner(\neg[s]_j)\urcorner)\vee\exists k\leq s([s]_i=\ulcorner(\exists x_k\,[s]_j)\urcorner\vee[s]_i=\ulcorner(\forall x_k\,[s]_j)\urcorner)\end{array}\right).$$

A string of L_A-symbols is a formula if it appears on some formula-building sequence. The relation Form(x), expressing that x is the Gödel-number of an L_A-formula, is defined by

$$\exists s(\text{formseq}(s)\wedge\exists i<\text{len}(s)\,[s]_i=x).$$

Next, we would like to obtain a Π_1-definition of Form(x). As, with the relation Term(x), we observe that x is the Gödel-number of a formula if and only if it appears on every formula-building sequence consisting exactly of the sub-formulas of x. More precisely, we define the relation seqforform(s,x), expressing that s consists precisely of the sub-formulas of x, by the formula

$$\begin{aligned}&\text{formseq}(s)\wedge\forall i<\text{len}(s)\,[s]_i\subseteq x\wedge\\&\forall y((y\subseteq x\wedge\text{formseq}(s\,\hat{}\,\langle y\rangle))\to\exists i<\text{len}(s)\,[s]_i=y).\end{aligned}$$

As with the relation $\mathrm{Term}(x)$, arguments justifying our definitions actually show that $\mathrm{Form}(x)$ is $\Delta_1(\mathrm{PA})$. Moreover, we have that:

Theorem 4.9. PA *proves*

$$\begin{aligned}\mathrm{Form}(x) \leftrightarrow \quad & \exists s,t(\mathrm{Term}(s) \wedge \mathrm{Term}(t) \wedge (x = s \,\hat{}\, \ulcorner{=}\urcorner \,\hat{}\, t \vee x = s \,\hat{}\, \ulcorner{<}\urcorner \,\hat{}\, t))\vee \\ & \exists y,z(\mathrm{Form}(y) \wedge \mathrm{Form}(z) \wedge (x = y \,\hat{}\, \ulcorner\vee\urcorner \,\hat{}\, z \vee x = y \,\hat{}\, \ulcorner\wedge\urcorner \,\hat{}\, z))\vee \\ & \exists y(\mathrm{Form}(y) \wedge x = \ulcorner\neg\urcorner \,\hat{}\, y \vee \exists k \leq x(x = \ulcorner\exists x_k\urcorner \,\hat{}\, y \vee x = \ulcorner\forall x_k\urcorner \,\hat{}\, y))).\end{aligned}$$

If $M \models \mathrm{PA}$ is nonstandard, then it will satisfy $\mathrm{Form}(x)$ of exactly those natural numbers that are Gödel-numbers of L_A-formulas. But $\mathrm{Form}(x)$ will also hold of nonstandard elements representing 'nonstandard' formulas having for instance a nonstandard number many conjunctions.

There are a few more relations and functions related to the grammar of first-order logic that we will require in future sections and we will just give brief descriptions of these, trusting that the reader will have no trouble convincing herself that they are recursive (and, in fact, $\Delta_1(\mathrm{PA})$). The function $y = \mathrm{Neg}(x)$ expresses that x is the Gödel-number of a formula φ and $y = \ulcorner\neg\varphi\urcorner$. The relation $\mathrm{Free}(x,i)$ expresses that x is the Gödel-number of a formula φ and x_i is a free variable of φ. The function $y = \mathrm{Sub}(x,z,i)$ expresses that x is the Gödel-number of a formula φ, y is the Gödel-number of a term t, and z is the Gödel-number of the formula that is the result of the substituting t for x_i in φ if the substitution is proper, and $z = x$ otherwise. Whenever we need to substitute a term of the form $\underline{n} = \underbrace{1 + \ldots + 1}_{n\text{-many}}$, we will abuse notation by simply writing $\mathrm{Sub}(x,n,i)$, instead of the correct but much less readable $\mathrm{Sub}(x,\ulcorner\underline{n}\urcorner,i)$.

We shall say that an L_A-theory T is *recursive* if the set of the Gödel-numbers of its formulas is recursive. We shall say that T is *arithmetic* if the set of the Gödel-numbers of its formulas is definable over $\mathbb{N}$. Using Church's Thesis that recursive is equivalent to algorithmically computable, it is clear that any theory composed in a human brain must be recursive. The theories we produce are either finite or follow a recognizable pattern, for otherwise we would not be able to express them. Thus, whenever we used the term 'reasonable' with respect to a first-order theory in previous sections, we can now replace it by 'recursive'.

Example 4.10. PA is recursive. Recall that PA consists of 15 axioms together with the induction scheme. Let $n_1, \ldots, n_{15}$ be the Gödel-numbers of the 15 axioms. The Gödel-number of a Peano axiom is either one of the numbers n_1 through n_{15}, or it has the form of an induction axiom. In the second case, there is a formula φ and a sequence s of all free variables of φ, whose first variable is the one we are inducting on. We then construct the induction axiom using s, $[s]_0$, φ, φ with 0 substituted for $[s]_0$, and φ with $[s]_0 + 1$ substituted for $[s]_0$. Now for the gory details. We define the relation $\mathrm{PA}(x)$, expressing that x is the Gödel-number of a Peano axiom by the formula

$$\begin{aligned}& x = n_1 \vee \cdots \vee x = n_{15}\vee \\ & \exists y,s \subseteq x \exists n \leq s \left(\begin{array}{l} \mathrm{Form}(y) \wedge \mathrm{len}(s) = n\wedge \\ \forall i < \mathrm{len}(n)\, \mathrm{Free}(y,[s]_i) \wedge \forall j \leq y(\mathrm{Free}(y,j) \to \exists k \leq s\, [s]_k = j)\wedge \\ \exists t \subseteq s \exists u,w \left(\begin{array}{l} \mathrm{len}(t) = \mathrm{len}(s) - 1 \wedge \forall i < \mathrm{len}(t)\, [t]_i = [s]_{i+1}\wedge \\ u = \mathrm{Sub}(y,[s]_0,\ulcorner 0\urcorner) \wedge w = \mathrm{Sub}(y,[s]_0,\ulcorner [s]_0 + 1\urcorner)\wedge \\ x = \ulcorner(\forall t\, (u \wedge (\forall [s]_0\, (y \to w) \to \forall [s]_0\, y)))\urcorner \end{array} \right) \end{array} \right).\end{aligned}$$

Note that since Sub is a function, we may replace the quantifiers $\exists u, w$ with $\forall u, w$. Thus, $\mathrm{PA}(x)$ is $\Delta_1(\mathrm{PA})$.

Example 4.11. LOG_{L_A} is recursive and there is a predicate $\mathrm{LOG}_{L_A}(x)$ defining it which is $\Delta_1(\mathrm{PA})$.

Suppose that T is an arithmetic theory and $T(x)$ is the formula that holds exactly of the Gödel-numbers of formulas in T. We define the relation $\mathrm{Proof}_T(x,s)$, expressing that x is the Gödel-number of a formula φ and s is a sequence of Gödel-numbers of formulas representing a proof of φ from T, by the formula

$$\forall i < \mathrm{len}(s)(T([s]_i) \vee \mathrm{LOG}_{L_A}([s]_i) \vee \exists j,k < i\, [s]_k = \ulcorner [s]_j \to [s]_i \urcorner).$$

If $T(x)$ is recursive (or more generally $\Delta_1(\mathrm{PA})$), then so is $\mathrm{Proof}_T(x,s)$. Finally, we define the relation $\mathrm{Pr}_T(x)$, expressing that x is the Gödel-number of a formula provable from T, by the formula

$$\exists s\, \mathrm{Proof}_T(x,s).$$

If T is recursive, then $\mathrm{Pr}_T(x)$ is a $\Sigma_1^{\mathbb{N}}$. But, we will see in Section 7 that $\mathrm{Pr}_{\mathrm{PA}}(x)$ is not $\Pi_1^{\mathbb{N}}$.

It should be noted that, aside from $\mathrm{Pr}_T(x)$, all relations we introduced in this section are, in fact, primitive recursive. It is not difficult (but tedious) to verify that in every case, we could bound the length of the search by a primitive recursive function.

4.4. **Truth is Undefinable.** Let us suppose that $\mathbb{N}$ has a complete arithmetic axiomatization T. We saw in the previous section that there is an L_A-formula $\mathrm{Pr}_T(x)$ such that $\mathbb{N} \models \mathrm{Pr}_T(n)$ if and only if n is the Gödel-number of an L_A-sentence φ and $T \vdash \varphi$. Since, for every L_A-sentence φ, we have that $T \vdash \varphi$ or $T \vdash \neg\varphi$, the formula $\mathrm{Pr}_T(x)$ holds precisely of those sentences φ that are true in $\mathbb{N}$. Thus, $\mathbb{N}$ can define its own truth! But, in this case, a formal version of Richard's paradox produces a contradiction in arithmetic! The argument for the undefinability of truth was given by Alfred Tarski in 1936.

Theorem 4.12 (Undefinability of Truth)**.** *There cannot be an L_A-formula* $\mathrm{Tr}(x)$ *such that* $\mathbb{N} \models \mathrm{Tr}(n)$ *if and only if n is the Gödel number of an L_A-sentence φ and* $\mathbb{N} \models \varphi$.

Proof. Suppose that there is such a formula $\mathrm{Tr}(x)$. We define the relation $\mathrm{Ord}(x)$, expressing that x is the Gödel-number of a formula $\varphi(y)$ and $\neg\varphi(\ulcorner\varphi(y)\urcorner)$ holds, by the formula

$$\begin{aligned}&\mathrm{Form}(y) \wedge \exists i \leq y\, \mathrm{Free}(y,i) \wedge \forall j \leq y\, \mathrm{Free}(y,j) \to j = i \wedge\\ &\exists u\, (u = \mathrm{Sub}(x,x,i) \wedge \neg\mathrm{Tr}(u)).\end{aligned}$$

Let $m = \ulcorner \mathrm{Ord}(x) \urcorner$. We have that

$$\begin{array}{ll}\mathrm{Ord}(m) & \leftrightarrow\\ \mathrm{Ord}(\ulcorner\mathrm{Ord}(x)\urcorner) & \leftrightarrow\\ \neg\mathrm{Tr}(\mathrm{Ord}(\ulcorner\mathrm{Ord}(x)\urcorner)) & \leftrightarrow\\ \neg\mathrm{Tr}(\mathrm{Ord}(m)) & \leftrightarrow\\ \neg\mathrm{Ord}(m). & \end{array}$$

Thus, we have reached a contradiction showing that truth is undefinable. $\square$

A version of the First Incompleteness Theorem follows as an immediate corollary of Theorem 4.4. Let us say that a theory T of L_A is *true* if $\mathbb{N} \models T$.

Corollary 4.13. *Suppose that T is a true arithmetic theory. Then T is incomplete.*

Thus, even if we improbably stretched our definition of a reasonable axiomatization to include arithmetic theories, we still could not hope to axiomatize number theory.

Another immediate consequence of Theorem 4.4 is that countable models of PA are not all elementarily equivalent, they do not share the same theory.

Corollary 4.14. *There is a consistent recursive theory extending* PA *that is not satisfied by* $\mathbb{N}$.

Proof. Let φ be a true sentence that does not follow from PA, then $T := \text{PA} \cup \{\neg\varphi\}$ is consistent. □

Because $\mathbb{N}$ is not a model of T, it does not follow from Theorem 4.13 that T is incomplete. But it will follows from the full Incompleteness Theorem due to Barkley Rosser in 1936, which states that any recursive theory (not just 'true' theories) extending PA^- must be incomplete.

4.5. **Constructing true undecidable sentences.** The proof of Theorem 4.13, which we presented in the previous section, is an example of a non-constructive existence proof. It demonstrates that there is a sentence that is not provable from, say PA, but it cannot tell us what that sentence is. In this section, we will present another version of the First Incompleteness Theorem that is closer to Gödel's original in both statement and proof. Here, we will actually construct a sentence that is not provable from the given theory T. The proof rests on Gödel's *Diagonalization Lemma*, a generalized version of the self-referentiality we exploited in the proof of the undefinability of truth.

Theorem 4.15 (Diagonalization Lemma)**.** *For every formula $\varphi(x)$, there is a sentence δ such that* $\text{PA}^- \vdash \delta \leftrightarrow \varphi(\ulcorner\delta\urcorner)$.

Proof. We start by defining the relation $\text{Diag}(x)$, expressing that x is the Gödel-number of a formula $\phi(y)$ and $\varphi(\ulcorner\phi(\ulcorner\phi(y)\urcorner)\urcorner)$ holds, by the formula

$$\begin{aligned}&\text{Form}(y) \wedge \exists i \leq x\, \text{Free}(x,i) \wedge \forall j \leq x\, \text{Free}(x,j) \to j = i \wedge \\ &u = \text{Sub}(x,x,i) \wedge \varphi(u).\end{aligned}$$

Now we let $\delta := \text{Diag}(\ulcorner\text{Diag}(y)\urcorner)$. Notice that this is exactly the construction of the proof of the Undefinability of Truth theorem, where we used $\varphi(x) = \neg\text{Tr}(x)$. Let us argue that δ has the requisite property. Suppose that $M \models \text{PA}^-$. Suppose that $M \models \delta$. Then $M \models \text{Diag}(\ulcorner\text{Diag}(x)\urcorner)$, from which it follows that $M \models \varphi(u)$ where $u = \text{Sub}(\ulcorner\text{Diag}(y)\urcorner, \ulcorner\text{Diag}(y)\urcorner, y)$. Since the function $z = \text{Sub}(x,z,i)$ is represented in PA^-, we have that $u = \ulcorner\text{Diag}(\ulcorner\text{Diag}(y)\urcorner)\urcorner$, and so $M \models \varphi(\ulcorner\text{Diag}(\ulcorner\text{Diag}(y)\urcorner)\urcorner)$. But this means that $M \models \varphi(\ulcorner\delta\urcorner)$. Now suppose that $M \models \varphi(\ulcorner\delta\urcorner)$. Since the relation $\text{Form}(x)$ is represented in PA^-, we have that $M \models \text{Form}(\ulcorner\text{Diag}(x)\urcorner)$ and since $y = \text{Sub}(x,z,i)$ is represented in PA^-, we have that $M \models \delta = \ulcorner\text{Diag}(\ulcorner\text{Diag}(y)\urcorner)\urcorner = \text{Sub}(\ulcorner\text{Diag}(y)\urcorner, \ulcorner\text{Diag}(y)\urcorner, y)$. Thus, $M \models \text{Diag}(\ulcorner\text{Diag}(y)\urcorner)$, meaning that $M \models \delta$. This completes the proof that $\text{PA}^- \vdash \delta \leftrightarrow \varphi(\ulcorner\delta\urcorner)$. □

Theorem 4.16 (First Incompleteness Theorem for true theories)**.** *Suppose that T is a true recursive theory extending* PA^-. *Then T is incomplete.*

Proof. Let us apply the Diagonalization Lemma to the formula $\neg\text{Pr}_T(x)$ to obtain a sentence σ such that

$$\text{PA}^- \vdash \sigma \leftrightarrow \neg\text{Pr}_T(\ulcorner\sigma\urcorner).$$

Intuitively, σ says 'I am not provable' from T! We will argue that T does not prove σ or its negation. Suppose towards a contradiction that $T \vdash \sigma$. It follows that $\mathbb{N} \models \sigma$ and thus $\mathbb{N} \models \neg\text{Pr}_T(\ulcorner\sigma\urcorner)$. But this means that T does not prove σ, which is the desired contradiction. Now suppose towards a contradiction that $T \vdash \neg\sigma$. It follows that $\mathbb{N} \models \neg\sigma$ and thus $\mathbb{N} \models \text{Pr}_T(\ulcorner\sigma\urcorner)$. But this means that $T \vdash \sigma$, which is the desired contradiction. Finally, since $\mathbb{N} \models \neg\text{Pr}_T(\ulcorner\sigma\urcorner)$, it follows that $\mathbb{N} \models \sigma$. □

Since every Σ_1-sentence that is true over $\mathbb{N}$ must holds in all models of PA^-, it follows that PA^- proves all true Σ_1-sentences. Since the sentence σ is $\Pi_1(\text{PA})$, it follows that a true recursive theory extending PA cannot decide all Π_1-sentences. Thus, we cannot even hope to axiomatize the Π_1-theory of the natural numbers.

4.6. **First Incompleteness Theorem.** In this section, we prove another version of the First Incompleteness Theorem due to Rosser, showing that any consistent (not necessarily true) recursive theory T extending PA^- is incomplete.

Theorem 4.17 (First Incompleteness Theorem)**.** *Suppose that T is a consistent recursive theory extending* PA^-*. Then T is incomplete.*

Proof. Let us apply the Diagonalization Lemma to the formula

$$\forall s(\text{Proof}_T(x,s) \to \exists t < s\text{Proof}_T(\text{Neg}(x),t)).$$

We obtain the sentence τ such that

$$\text{PA}^- \vdash \tau \leftrightarrow \forall s(\text{Proof}_T(\ulcorner\tau\urcorner,s) \to \exists t \leq s\,\text{Proof}_T(\text{Neg}(\ulcorner\tau\urcorner),t)).$$

Intuitively, τ says that 'whenever there is a proof of myself from T, then there is a shorter proof of my negation from T'. We will argue that T does not prove τ or its negation. Suppose towards a contradiction that $T \vdash \tau$. Let $p \in \mathbb{N}$ be the code of a proof of τ from T. Thus,

(1) $\mathbb{N} \models \text{Proof}_T(\ulcorner\tau\urcorner,p)$,
(2) for all $n < p$, $\mathbb{N} \models \neg\text{Proof}_T(\text{Neg}(\ulcorner\tau\urcorner),n)$.

Since the relation $\text{Proof}_T(x,s)$ and the function $\text{Neg}(x)$ are represented in PA^-, we have that

(1) $\text{PA}^- \vdash \text{Proof}_T(\ulcorner\tau\urcorner,p)$,
(2) for all $n < p$, $\text{PA}^- \vdash \neg\text{Proof}_T(\text{Neg}(\ulcorner\tau\urcorner),n)$.

Thus,

$$\text{PA}^- \vdash \exists s(\text{Proof}_T(\ulcorner\tau\urcorner,s) \wedge \forall t < s\,\neg\text{Proof}_T(\text{Neg}(\ulcorner\tau\urcorner),t)).$$

and so $\text{PA}^- \vdash \neg\tau$, which is the desired contradiction. Now suppose towards a contradiction that $T \vdash \neg\tau$. Let $p \in \mathbb{N}$ be the code of a proof of $\neg\tau$ from T. Thus,

(1) $\mathbb{N} \models \text{Proof}_T(\text{Neg}(\ulcorner\tau\urcorner),p)$,
(2) for all $n \in \mathbb{N}$, we have $\mathbb{N} \models \neg\text{Proof}_T(\ulcorner\tau\urcorner,n)$.

Since the relation $\text{Proof}_T(x,y)$ and the function $\text{Neg}(x)$ are represented in PA^-, we have that

(1) $\text{PA}^- \vdash \text{Proof}_T(\text{Neg}(\ulcorner\tau\urcorner),p)$,
(2) for all $n \in \mathbb{N}$ we have, $\text{PA}^- \vdash \neg\text{Proof}_T(\ulcorner\tau\urcorner,n)$.

Now suppose that $M \models \mathrm{PA}^-$. If $a \in M$ such that $M \models \mathrm{Proof}_T(\ulcorner\tau\urcorner, a)$, then it must be that a is nonstandard and hence $a > p$. Thus, $M \models \exists t < a\, \mathrm{Proof}_T(\mathrm{Neg}(\ulcorner\tau\urcorner), t)$. Thus,

$$M \models \forall s(\mathrm{Proof}_T(\ulcorner\tau\urcorner, s) \to \exists t < s\, \mathrm{Proof}_T(\mathrm{Neg}(\ulcorner\tau\urcorner), t)).$$

It follows that

$$\mathrm{PA}^- \vdash \forall s(\mathrm{Proof}_T(\ulcorner\tau\urcorner, s) \to \exists t < s\, \mathrm{Proof}_T(\mathrm{Neg}(\ulcorner\tau\urcorner), t)).$$

and hence $\mathrm{PA}^- \vdash \tau$. This completes that proof that T is incomplete. □

One can, at this point, ask whether it was really necessary for Rosser to use a sentence different from Gödel's for the proof of his theorem.

Question 4.18. Is there a consistent recursive theory T which decides Gödel's sentence σ?

In Section 5, we will see that the answer is yes, and so it was indeed necessary for Rosser to invent a different sentence.

Before the discussion of the First Incompleteness Theorem ends, it should be observed that its proof goes through for any theory T which *interprets* PA. We shall say that a theory T in a language L *interprets* PA if there are formulas $N(x)$, $a(x, y, z)$, $m(x, y, z)$, $o(x, y)$, $0(x)$, $1(x)$ satisfying

(1) $a(x, y, z)$ and $m(x, y, z)$ are binary functions on N,
(2) $o(x, y)$ is a binary relation on N,
(3) $0(x)$ and $1(x)$ hold of unique elements of N,

so that T proves that N with a interpreting $+$, m interpreting $\cdot$, o interpreting order, and 0, 1 interpreting the constants is a model of PA^-. Thus, for instance, the First Incompleteness Theorem applies to the set theoretic axioms ZFC.

4.7. **A recursive tree without a recursive branch.** In the previous section, we presented two versions of incompleteness. One version (Theorem 4.4), generalizing Gödel's argument, stated that every true arithmetic theory must be incomplete. Rosser's version (Theorem 4.17) stated that every consistent recursive theory extending PA^- must be incomplete. The two variants of incompleteness make it natural to wonder whether Rosser's result can be strengthened to show that every consistent arithmetic theory extending PA^- is incomplete. But it cannot be!

Theorem 4.19. *There is a complete consistent arithmetic theory extending* PA.

The proof we present below is much more complicated than is required, but the construction employed in it negatively answers Question **??** and will be used to prove Tennebaum's Theorem in Section 6.

Proof. First, let us define a function f on the natural numbers such that $f(0)$ is the least Gödel-number of an L_A-sentence and $f(i+1)$ is the least Gödel-number of an L_A-sentence greater than $f(i)$. The function f is a recursive enumeration of the Gödel-numbers of L_A-sentences. Let $f(i)$ be the Gödel-number of the sentence φ_i and define that $\varphi_i^1 = \varphi_i$ and $\varphi_i^0 = \neg\varphi_i$. We shall say that a proof has *size* n if its code is n. Next, let us define that a sequence $\langle \varphi_0^{i_0}, \varphi_1^{i_1}, \ldots, \varphi_{n-1}^{i_{n-1}} \rangle$, where $i_j \in \{0, 1\}$, is *n-almost consistent* if there is no proof of the sentence $0 = 1$ from $\mathrm{PA} \cup \{\varphi_0^{i_0}, \varphi_1^{i_1}, \ldots, \varphi_{n-1}^{i_{n-1}}\}$ of size $\leq n$. First, we observe that a sequence that is n-almost consistent for every $n \in \mathbb{N}$ is consistent with PA. Also, if $\langle \varphi_0^{i_0}, \varphi_1^{i_1}, \ldots, \varphi_{n-1}^{i_{n-1}} \rangle$

is n-almost consistent, then any initial segment $\langle\varphi_0^{i_0},\varphi_1^{i_1},\ldots,\varphi_{m-1}^{i_{m-1}}\rangle$ for $m<n$ is m-almost consistent. Now we let T be the collection of all binary sequences s such that $\langle\varphi_0^{s(0)},\varphi_1^{s(1)},\ldots,\varphi_{n-1}^{s(n-1)}\rangle$ is n-almost consistent for $n=\text{len}(s)$. The collection T is clearly infinite and by our previous observations, it is a binary tree. It is also easy to see that T is $\Delta_1^{\mathbb{N}}$, and hence recursive. Finally, observe that any infinite branch through T gives a consistent complete theory extending PA. So it remains to show that one such branch is definable over $\mathbb{N}$. But such a definition comes directly from the proof of König's Lemma. The branch is defined recursively so that once b_i, the element of the branch on level i, has been chosen we choose b_{i+1} above b_i on level $i+1$ as the leftmost of the two successors having infinitely many elements in T above it. □

By Theorem 4.17, the definable branch b cannot be recursive. Thus, as a corollary of the proof, we have that:

Theorem 4.20. *There is a recursive binary tree without a recursive branch.*

Corollary 4.21. *Suppose that $M\models$ PA is nonstandard. Then $SSy(M)$ contains a non-recursive set.*

Proof. Since $SSy(M)$ contains all recursive sets, it must in particular contain T. But then $SSy(M)$ must also contain some branch of T. □

In particular, every standard system contains a complete consistent L_A-theory extending PA. Corollary 4.21 will form a crucial component of the proof of Tennenbaum's Theorem in Section 6.

4.8. **On the definability of Σ_n-truth.** We showed in Theorem 4.4 that $\mathbb{N}$ (and more generally any model of PA) cannot define its own truth. But remarkably, every model $M\models$ PA can define truth for every fixed level Σ_n (or Π_n) of the arithmetic hierarchy. Suppose that $M\models$ PA. Earlier we established the convention that given $a\in M$, we consider $[a]_i=0$ for all $i\geq\text{len}(a)$. We shall argue that, for every $n\in\mathbb{N}$, there is a formula $\text{Tr}_{\Sigma_n}(x,f)$ such that if $\varphi(x_{i_0},\ldots,x_{i_m})$ is any Σ_n-formula and $f\in M$, then $M\models\text{Tr}_{\Sigma_n}(\ulcorner\varphi(x_{i_0},\ldots,x_{i_m})\urcorner,f)$ if and only if $M\models\varphi([f]_{i_0},\ldots,[f]_{i_m})$. The parameter f codes an assignment of elements of M to the free variables of φ. Similarly, there is such a formula $\text{Tr}_{\Pi_n}(x,f)$ evaluating Π_n-formulas. Indeed, the formulas $\text{Tr}_{\Sigma_n}(x,f)$ and $\text{Tr}_{\Pi_n}(x,f)$ themselves turn out to be Σ_n and Π_n respectively.

Because a Σ_n-formula consists of finitely many alterations of quantifiers followed by a Δ_0-formula, once it has been established that there is a Δ_1(PA)-definable Δ_0-truth predicate, the existence of the truth predicates $\text{Tr}_{\Sigma_n}(x,f)$ follows easily. Let us see an example of why this is the case. Suppose that $\text{Form}_{\Delta_0}(x)$ is the relation that holds exactly of the Gödel-numbers of Δ_0-formulas. It is not difficult to argue that, like the rest of the first-order logic grammar predicates, $\text{Form}_{\Delta_0}(x)$ is Δ_1(PA). Now assume that we are given $\text{Tr}_{\Delta_0}(x,f)$. We then define $\text{Tr}_{\Sigma_1}(x,f)$ by the formula

$$\exists z,i\leq x\left(\begin{array}{l}\text{Form}_{\Delta_0}(z)\wedge x=\ulcorner\exists x_i z\urcorner\wedge\\ \exists a\exists f'\left(\begin{array}{l}\text{len}(f')\geq\text{len}(f)\wedge[f]_i=a\wedge\\ \forall j<\text{len}(f)(j\neq i\to[f]_j=[f']_j)\wedge\text{Tr}_{\Delta_0}(z,f')\end{array}\right)\end{array}\right).$$

Before we can evaluate the truth of Δ_0-formulas, we need to be able to compute the value of a term. To evaluate a term for some assignment to the free variables,

we will use a term-building sequence for that term and a parallel sequence that will record our computation for all its sub-terms as we move along the term-building sequence. We define the relation valseq(s, f, e), expressing that s is a term-building sequence, f codes an assignment of elements to the free variables appearing in s, and e is the sequence consisting of the step by step evaluation according to s of the $[s]_i$, by the formula

$$\begin{array}{l}\text{termseq}(s) \wedge \text{len}(e) = \text{len}(s) \wedge \\ \forall i < \text{len}(s) \left(\begin{array}{l} ([s]_i = \ulcorner 0 \urcorner \wedge [e]_i = 0) \vee \\ ([s]_i = \ulcorner 1 \urcorner \wedge [e]_i = 1) \vee \\ \exists j \leq s([s]_i = \ulcorner x_j \urcorner \wedge [e]_i = [y]_j) \vee \\ \exists j, k < i([s]_i = \ulcorner [s]_j + [s]_k \urcorner \wedge [t]_i = [t]_j + [t]_k) \vee \\ \exists j, k < i([s]_i = \ulcorner [s]_j \cdot [s]_k \urcorner \wedge [t]_i = [t]_j \cdot [t]_k) \end{array} \right).\end{array}$$

Now we define the relation val(x, f, y), expressing that x is the Gödel-number of a term $t(x_{i_0}, \ldots, x_{i_m})$ and $y = t([f]_{i_0}, \ldots, [f]_{i_m})$, by the Σ_1-formula

$$\exists s, e \, \text{valseq}(s \mathbin{\hat{}} \langle x \rangle, f, e \mathbin{\hat{}} \langle y \rangle) \vee (\neg \text{Term}(x) \wedge z = 0)$$

Since val(x, f, y) is the graph of a function in all models of PA, it follows that it is Δ_1(PA). Moreover:

Theorem 4.22. PA *proves:*

$$\begin{array}{l} \forall f (\text{val}(\ulcorner 0 \urcorner, f, 0) \wedge \text{val}(\ulcorner 1 \urcorner, f, 1) \\ \forall f, i (i < \mathit{len}(f) \rightarrow \text{val}(\ulcorner x_i \urcorner, f, [f]_i)) \\ \forall x, z, f \exists y_1, y_2 \left((\text{val}(x, f, y_1) \wedge \text{val}(z, f, y_2) \rightarrow \text{val}(\ulcorner x + z \urcorner, f, y_1 + y_2) \right) \\ \forall x, z, f \exists y_1, y_2 \left((\text{val}(x, f, y_1) \wedge \text{val}(z, f, y_2) \rightarrow \text{val}(\ulcorner x \cdot z \urcorner, f, y_1 \cdot y_2) \right). \end{array}$$

While we were able to evaluate terms linearly, evaluating a Δ_0-formula is a branching process that is similar to computing the Ackermann function. For instance, when a formula is a conjunction of two formulas, the evaluation splits into two separate evaluations and when a formula starts with an existential quantifier bounded by some element a, the evaluation splits into a separate evaluations. Thus, similar to the case of evaluating the Ackermann function, we need to express when a sequence of evaluations witnesses that a given formula is true or false. We define the relation $\text{satseq}_{\Delta_0}(s, e)$, expressing that s is a formula-building sequence and e is a sequence witnessing the truth value of a given formula in s. Each element $[e]_l$ of e is a triple $\langle i, f, v \rangle$, where v is the truth value (1 or 0) of the formula φ coded by $[s]_i$ under the assignment f. In order to correctly witness the truth value of $[s]_i$, the sequence e should obey the following list of requirements. If $[s]_i$ is an atomic formula and $[e]_l = \langle i, f, v \rangle$, then the value of v is determined using val. If $[s]_i = \ulcorner [s]_j \wedge [s]_k \urcorner$ and $[e]_l = \langle i, f, v \rangle$, then there are elements $[e]_{l_j} = \langle j, f, v_j \rangle$ and $[e]_{l_k} = \langle k, f, v_k \rangle$, and $v = 1$ if and only if $v_j = 1$ or $v_k = 1$. Conjunctions and negations are handled similarly. If $[s]_i = \ulcorner \forall x_h < x_j [s]_k \urcorner$ and $[e]_l = \langle i, f, v \rangle$, then for all $a < [f]_j$, there are elements $[e]_{m_a} = \langle k, f_a, v_a \rangle$, where f_a agrees with f everywhere except on coordinate h and $[f_a]_h = a$. It is then the case that $v = 1$ if and only if all $v_a = 1$. The bounded existential quantifier is handled similarly. It is left as a homework project to determine the exact formula defining $\text{satseq}_{\Delta_0}(s, e)$. Finally, we define the relation $\text{Tr}_{\Delta_0}(x, f)$ by the Σ_1-formula

$$\exists s, e \left(\text{satseq}_{\Delta_0}(s \mathbin{\hat{}} \langle x \rangle, e) \wedge \exists l < \text{len}(e) \, [e]_l = \langle \text{len}(s), f, 1 \rangle \right)$$

and the Π_1-formula

$$\forall s, e\,(\exists v \leq e \exists l < \mathrm{len}(e)\,((\mathrm{satseq}_{\Delta_0}(s \mathbin{\hat{}} \langle x\rangle, e) \wedge ([e]_l = \langle \mathrm{len}(s), f, v\rangle) \to v = 1)).$$

It should be noted that it is far from obvious that truth-evaluating sequences e exist in the first place and an inductive argument must be made to verify that this is indeed the case.

Taken together the facts that truth taken as a whole is undefinable, but every Σ_n-level of truth is definable implies that the arithmetic hierarchy does not collapse.

Theorem 4.23. *Suppose that $M \models \mathrm{PA}$. For every $n \in \mathbb{N}$, there is a Σ_n-formula that is not $\Pi_n(M)$.*

4.9. **Homework.** (Source: *Models of Peano Arithmetic* by Richard Kaye [Kay91])

Question 4.1. Prove Theorem 4.7.

Question 4.2. Prove that if φ is Π_1, then the δ of the Diagonalization Lemma is $\Pi_1(\mathrm{PA})$.

Question 4.3. Provide a Δ_0-definition for the relation $\mathrm{satseq}_{\Delta_0}(x, f)$.

5. The Second Incompleteness Theorem

There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable. There is another theory which states that this has already happened.
–**Douglas Adams**

5.1. **Finitary Consistency Proofs.** A subject such as set theory investigates the properties of infinite collections vastly beyond the reach of human physical experience. But its defining properties are codified by the Zermelo-Fraenkel axioms with the Axiom of Choice (ZFC), which are expressed in the language of first-order logic. Other than the countably many variables it uses, the first-order language of set theory is finite and the ZFC axioms are each a finite string of symbols in that language. A recursive procedure determines whether a string of symbols is a ZFC axiom and another recursive procedure determines whether a string of formulas in the language of set theory is a proof from ZFC. It is thus conceivable that one may study infinite collections, without ever truly leaving the finite realm, by simply manipulating the finite string of symbols representing the properties of sets via recursively defined operations. Through these manipulations, and without any reference to the infinite objects whose existence appears beyond intuitive justification, mathematicians may be nevertheless be able to justify the existence of these infinities by showing that ZFC axioms can never prove a contradiction. Such was the view of David Hilbert who sought 'finitary' consistency proofs for among other, PA and ZFC axioms. Hilbert believed that natural numbers and the recursive operations we use to manipulate them precede any understanding we might attempt to make of the physical world. They are the primitive notions out of which our scientific and mathematical thoughts are constructed. Hilbert also believed that all mathematical objects whose properties are captured by a consistent axiom system exist in the mathematical sense.

A Hilbert style finitary consistency proof would achieve its conclusion by manipulating finite strings of symbols using recursive operations. Because Peano Arithmetic codified all rules for recursive manipulation of finite objects, it followed that if there is indeed a finitary proof of the consistency of an axiomatic system such

as PA or ZFC, then it should be formalizable in PA! In 1931, Gödel presented his Second Incompleteness Theorem showing that no proof of the consistency of PA or any recursive axiom system extending it could be formalized in PA itself. This is made precise in the following way. Using the coding we introduced in Section 4, any finitary proof of the consistency of PA, or a recursive theory extending it, can be carried out inside a model of PA. Since all models of PA agree on the result of recursive operations on the natural numbers, they should recognize that a contradiction cannot be obtained from the PA axioms. For a recursive theory T of L_A, let $\mathrm{Con}(T)$ be the arithmetic sentence expressing that there is no proof of some formula and also a proof of its negation from T:

$$\mathrm{Con}(T) := \forall x(\mathrm{Form}(x) \to \neg(\mathrm{Pr}_T(x) \wedge \mathrm{Pr}_T(\mathrm{Neg}(x)))).$$

Thus, every model of PA, should recognize that Con(PA) holds. But this simply cannot be.

Theorem 5.1 (Second Incompleteness Theorem)**.** *Suppose that T is a consistent recursive theory extending* PA*. Then $T \nvdash \mathrm{Con}(T)$.*

Thus, there are models of PA which think that the PA-axioms are contradictory! How is this possible? Remember that a nonstandard model $M \models \mathrm{PA}$ has many nonstandard elements that it thinks are codes of PA-axioms, namely the induction axioms for all nonstandard formulas (as well as nonstandard instances of LOG_{L_A}-axioms). Moreover, proofs from the perspective of M are not just finite but M-finite sequences of elements of $\mathrm{Form}(x)$. This means that what M thinks is a proof possibly has an infinite descending chain of modus ponens inferences. So that M has 'nonstandard' PA-axioms as well as 'nonstandard' proofs from which to obtain the erroneous contradiction.

5.2. **Arithmetized Completeness Theorem.** Proof-theoretic proofs of the Second Incompleteness Theorem rely on the so-called *derivability conditions* capturing the fundamental properties of the provability predicate $\mathrm{Pr}_T(x)$. We will present a model-theoretic proof which uses that models of PA can internally carry out Henkin's proof of the Completeness Theorem, in the sense that if $M \models \mathrm{Con}(T)$ for some definable theory T, then M can define a model of T. This result, due to Hilbert and Paul Bernays (1939), is known as the Arithmetized Completeness Theorem. In this section, we will formally state and prove the Arithmetized Completeness Theorem.

Suppose that L is a finite first-order language. Let us enumerate as $\{r_i \mid i < m\}$ the relations of L and let m_i be the arity of r_i. Let us also enumerate as $\{f_i \mid i < n\}$ the functions of L and let n_i be arity of f_i. Finally, let us enumerate as $\{c_i \mid i < k\}$ the constants of L. We shall code the symbols of L by natural numbers as follows. A relation symbol r_i of L is coded by the number $\langle 0, \langle i, m_i\rangle\rangle$, a function symbol f_i of L is coded by the number $\langle 1, \langle i, n_i\rangle\rangle$, and a constant symbol c_i of L is coded by the number $\langle 2, i\rangle$. The logical symbols of L are coded by ordered pairs with first coordinate 3 and the variables of L are coded by ordered pairs $\langle 4, i\rangle$ for $i \in \mathbb{N}$. Once we have specified codes for the alphabet of L, we proceed to define over a model of PA, all logical notions concerning L precisely as we did in Section 4. Thus, associated to the language L are definable predicates $\mathrm{Term}_L(x)$, $\mathrm{Form}_L(x)$, $y = \mathrm{Sub}_L(x, z, i)$, and others we have previously encountered. Suppose now that M is a model of PA. We shall say that a definable subset T of M is an *L-theory*

of M if $\mathrm{Form}_L(x)$ holds of all $x \in T$. We are not insisting that all elements of T are codes of *standard*, meaning actual, formulas of L. Some elements of T might be *nonstandard* formulas, containing, for instance, nonstandard number many conjunctions. Indeed, any definable T that contains infinitely many elements coding standard formulas, must by overspill contain some nonstandard formulas as well. Every L-theory T of M comes with the definable predicate $\mathrm{Pr}_T(x)$ and the corresponding sentence $\mathrm{Con}(T)$. We shall say that T is *M-consistent* if $M \models \mathrm{Con}(T)$. If T is M-consistent, we will show M is able to carry out the Henkin construction of the Completeness Theorem to define a model N of the standard formulas of T. Since the satisfaction relation for a Henkin model is completely determined by the complete Henkin theory, the model M will be able to define a *truth predicate* $\mathrm{Tr}(x, y)$ for N. The first parameter of $\mathrm{Tr}(x, y)$ will satisfy $\mathrm{Form}_L(x)$ and the second parameter will be an x-assignment of elements of N. The predicate will behave precisely as the satisfaction relation for N by obeying Tarski's rules for the definition of truth (for standard as well as nonstandard formulas), and it will be the true satisfaction relation on standard formulas.

Theorem 5.2 (Arithmetized Completeness Theorem)**.** *Suppose that L is a finite first-order language and T is an L-theory. Suppose further that $M \models PA$ has a definable subset $\overline{T}$ containing the codes of all formulas in T such that $M \models Con(\overline{T})$. Then there is a definable subset N of M, definable subsets $R_i \subseteq N_i^m$ for $i < m$, definable subsets $F_i \subseteq N^{n_i}$ for $i < n$, and $\{C_0, \ldots, C_k\} \subseteq N$ such that N together with interpretations R_i for r_i, F_i for f_i and C_i for c_i is a model of T. Moreover, M has a truth predicate for the model $\langle N, \{R_i\}_{i<m}, \{F_i\}_{i<n}, \{C_i\}_{i<k}\rangle$.*

To prove the theorem, we will argue that all components of the Henkin construction formalize within a model of PA, and therefore we can definably build N out of the Henkin constants. We will not provide all details of the proof because doing so does not advance the cause of understanding. Indeed, even the thorough Kaye [Kay91] refuses to provide a full proof, although Adamowicz and Zbierski [AZ97] come close.

Sketch of Proof. First, we observe that all meta-theorems about provability from Section 1, used in Henkin's proof of the Completeness Theorem, are formalizable in PA, meaning that they will hold true of the predicate $\mathrm{Pr}_T(x)$ over a model M of PA. Here are some examples. Proof by 'induction on theorems' holds over M. More precisely, suppose that $S(x)$ is a definable predicate over M such that $S(x) \to \mathrm{Form}_L(x)$, $\mathrm{LOG}_{L_A}(x) \to S(x)$, $\overline{T}(x) \to S(x)$, and whenever $S(\ulcorner\psi\urcorner)$ and $S(\ulcorner\psi \to \varphi\urcorner)$, then $S(\ulcorner\varphi\urcorner)$. It follows that $\mathrm{Pr}_{\overline{T}}(x) \to S(x)$. The deduction theorem (Theorem 1.5) holds over M. More precisely, we have $\mathrm{Pr}_{\overline{T}}(\ulcorner\varphi \to \psi\urcorner)$ if and only if $\mathrm{Pr}_{\overline{T},\varphi}(\ulcorner\psi\urcorner)$. One by one, we can check that the theorems used in Henkin's proof formalize.

We can easily extend the language L to a language L^* containing infinitely many new constants (which we shall use as Henkin constants), by defining that every element of the form $\langle 2, i\rangle$ for $i \in M$ is a constant of L^*. We shall say that an L^*-theory S of M has the *Henkin property* if whenever $M \models \mathrm{Form}_{L^*}(x)$, then S contains some Henkin sentence for x. We shall say that an L^*-theory S of M is *M-complete* if for all $x \in S$ either $M \models \mathrm{Pr}_S(x)$ or $M \models \mathrm{Pr}_S(\mathrm{Neg}(x))$. So suppose that S is an M-complete M-consistent L^*-theory of M with the Henkin property. The model M can obviously define the Henkin model N via the equivalence relation on the constants of L^* and verify that N satisfies $\varphi(c_1, \ldots, c_n)$ if and only if $\varphi(c_1, \ldots, c_n)$

is proved by S. Thus, the model M can define the truth predicate $\mathrm{Tr}(x,y)$ for N from S.

It remains to argue that T can be definably extended in M to an M-complete M-consistent L^*-theory with the Henkin property. In the proof of the Completeness Theorem, we constructed the complete consistent theory with the Henkin property in ω-many stages, by adding Henkin formulas and then completing the theory at each stage. An alternative construction, which we will use here, adds all Henkin formulas in a single strep through a careful bookkeeping of the constants and then completes the resulting theory. First, we enumerate in M all pairs $\langle x,i\rangle$ such that $M\models \mathrm{Form}_{L^*}(x)$ and $M\models \mathrm{Free}_{L^*}(x,i)$. Now we assign to each pair $\langle x,i\rangle$, the Henkin constant $c_{x,i}$ chosen as follows. At stage a, we consider the ath-pair in the enumeration and assign to it the smallest Henkin constant that hasn't yet been assigned to the preceding pairs. We shall say that $a\in M$ is an L^*-*Henkin formula* if it has the form

$$\exists x_i\varphi\to\varphi(c_{x,i}/x_i),$$

with $\ulcorner\varphi\urcorner = x$. Let T^* consist of $\overline{T}$ together with all L^*-Henkin formulas. Identical arguments to those used in the proof of the Completeness Theorem verify that $M\models \mathrm{Con}(T^*)$. Next, we need to consistently complete T^* in M. Let $\{\varphi_a\mid a\in M\}$ be a definable enumeration of elements satisfying $\mathrm{Form}_{L^*}(x)$. Now we define an M-complete M-consistent extension S of T^* as the collection $\{\varphi_a^*\mid a\in M\}$, where at each stage a, we choose $\varphi_a^*=\varphi_a$ if it is consistent with T^* and all choices made before stage a, and otherwise, we choose $\varphi^*_a=\neg\varphi_a$. It should be clear that $M\models\mathrm{Con}(S)$. □

Theorem 5.3. *If $M\models \mathrm{PA}+\mathrm{Con}(\mathrm{PA})$ and $N\models\mathrm{PA}$ is defined in M using the Arithmetized Completeness Theorem, then M is isomorphic to an initial segment of N.*

Proof. We shall argue that M embeds into an initial segment of N via the map i inductively defined in M by $i(0)=0^N$ and $i(a+1)=i(a)+^N 1^N$. Note that if there is any embedding $i:M\to N$, then it must satisfy the above definition. First, we argue by induction inside M that i maps onto an initial segment of N. It is vacuously true that for all $x<^N i(0)=0^N$, there is b such that $x=i(b)$. So suppose inductively that for all $x<^N i(a)$, there is b such that $x=i(b)$. We need to show that for all $x<^N i(a+1)$, there is b such that $x=i(b)$. But if $x<^N i(a+1)=i(a)+^N 1^N$, then $x<^N i(a)$ in which case there is a b by the inductive assumption or else $x=i(a)$ in which case $b=a$. Next, we argue, again by induction inside M, that $a<b$ implies that $i(a)<^N i(b)$. It is vacuously true that $b<0$ implies $i(b)<^N i(0)$. So suppose that $b<a$ implies $i(b)<^N i(a)$. If $b<a+1$, then $b\leq a$. If $a<b$, then by inductive assumption, we have $i(a)<^N i(b)<^N i(b)+^N 1^N=i(b+1)$, and if $a=b$, then $i(a)=i(b)<^N i(b+1)$. The arguments to show that $i(a+b)=i(a)+^N i(b)$ and $i(a\cdot b)=i(a)\cdot^N i(b)$ are similar and are left as homework. □

The Arithmetized Completeness Theorem can be extended to theories in a *recursive* (potentially infinite) language once that notion is carefully defined, but we will say no more about it in these notes.

5.3. **Second Incompleteness Theorem.**

Theorem 5.4 (Second Incompleteness Theorem)**.** *Suppose that T is a consistent recursive theory extending* PA*. Then $T \nvdash \mathrm{Con}(T)$.*

Proof. Recall the $\Pi_1(\mathrm{PA})$-sentence σ, used by Gödel in the proof of the First Incompleteness Theorem, that claims to be unprovable:

$$\mathrm{PA}^- \vdash \sigma \leftrightarrow \neg\mathrm{Pr}_T(\ulcorner\sigma\urcorner).$$

First, we shall argue that $T \vdash \mathrm{Con}(T) \to \sigma$. So suppose that $M \models T \cup \{\mathrm{Con}(T)\}$ and let N be the model of T defined in M using the Arithmetized Completeness Theorem. By Theorem 5.3, we may view M as an initial segment submodel of N, from which it follows that $M \prec_{\Delta_0} N$. Now we assume towards a contradiction that $M \models \neg\sigma$, and hence $M \models \mathrm{Pr}_T(\ulcorner\sigma\urcorner)$. It immediately follows that M cannot be a model of $\mathrm{Con}(T \cup \{\neg\sigma\})$. Thus, it must be that the complete consistent extension of T with the Henkin property constructed by M in the proof of the Arithmetized Completeness Theorem chose σ, making $N \models \sigma$. But σ is equivalent to a Π_1-formula and $M \prec_{\Delta_0} N$, which means that $M \models \sigma$ as well. Thus, we have reached a contradiction showing that $M \models \sigma$.

Now we suppose towards a contradiction that $T \vdash \mathrm{Con}(T)$. Thus, $T \vdash \sigma$ and hence $T \vdash \neg\mathrm{Pr}_T(\ulcorner\sigma\urcorner)$. But this is not possible, since if $T \vdash \sigma$, we have $\mathbb{N} \models \mathrm{Pr}_T(\ulcorner\sigma\urcorner)$ and hence every $M \models T$ is a also a model of $\mathrm{Pr}_T(\ulcorner\sigma\urcorner)$. Thus, we have reached a contradiction, showing that $T \nvdash \mathrm{Con}(T)$. □

The sentence Con(PA) provides another concrete example of a statement that is not decided by PA.

Model-theoretic proofs of the Second Incompleteness Theorem came years after Gödel first published his result. Older proofs were chiefly proof-theoretic and relied on general properties of the provability predicate $\mathrm{Pr}_T(x)$. Through the work of Hilbert, Paul Bernays and Martin Löb, three key properties of $\mathrm{Pr}_T(x)$ were isolated and became known as the *Derivability Conditions*. Suppose that T is a recursive theory.

Derivability Conditions for T

(1) $T \vdash \varphi$ implies $T \vdash \mathrm{Pr}_T(\ulcorner\varphi\urcorner)$
(2) $T \vdash (\mathrm{Pr}_T(\ulcorner\psi\urcorner) \wedge \mathrm{Pr}_T(\ulcorner\psi \to \varphi\urcorner)) \to \mathrm{Pr}_T(\ulcorner\varphi\urcorner)$
(3) $T \vdash \mathrm{Pr}_T(\ulcorner\varphi\urcorner) \to \mathrm{Pr}_T(\ulcorner\mathrm{Pr}_T(\ulcorner\varphi\urcorner)\urcorner)$

We argued for and made of use of condition (1) and (2) previously. It is only condition (3) that has a tedious details-ridden proof. It is precisely to avoid proving condition (3) that we chose to argue model-theoretically. To prove condition (3) we would argue, by induction on complexity of formulas, that for every Σ_1-formula φ, we have that $\mathrm{PA} \vdash \varphi \to \mathrm{Pr}_T(\ulcorner\varphi\urcorner)$. Thus, if $M \models \mathrm{PA}$ thinks that a Σ_1-formula is true, it can prove it!

The Second Incompleteness Theorem follows almost immediately from the Derivability Conditions.

Proof 2 *of the Second Incompleteness Theorem.* We will again argue that $T \vdash \mathrm{Con}(T) \to \sigma$. By condition (1), we have

$$T \vdash \mathrm{Pr}_T(\ulcorner\mathrm{Pr}_T(\ulcorner\sigma\urcorner) \to \neg\sigma\urcorner).$$

Now we will combine conditions (2) and (3) to argue that

$$T \vdash \mathrm{Pr}_T(\ulcorner\sigma\urcorner) \to \mathrm{Pr}_T(\ulcorner\neg\sigma\urcorner).$$

By condition (3), $T \vdash \mathrm{Pr}_T(\ulcorner\sigma\urcorner) \to \mathrm{Pr}_T(\ulcorner\mathrm{Pr}_T(\ulcorner\sigma\urcorner)\urcorner)$. Thus,

$$T \vdash \mathrm{Pr}_T(\ulcorner\sigma\urcorner) \to (\mathrm{Pr}_T(\ulcorner\mathrm{Pr}_T(\ulcorner\sigma\urcorner)\urcorner) \wedge \mathrm{Pr}_T(\ulcorner\mathrm{Pr}_T(\ulcorner\sigma\urcorner) \to \neg\sigma\urcorner).$$

From which it follows by condition (2) that $T \vdash \mathrm{Pr}_T(\ulcorner\sigma\urcorner) \to \mathrm{Pr}_T(\ulcorner\neg\sigma\urcorner)$. Thus,

$$T \vdash \mathrm{Pr}_T(\ulcorner\sigma\urcorner) \to (\mathrm{Pr}_T(\ulcorner\sigma\urcorner) \wedge \mathrm{Pr}_T(\ulcorner\neg\sigma\urcorner)),$$

and hence

$$T \vdash \mathrm{Pr}_T(\ulcorner\sigma\urcorner) \to \neg\mathrm{Con}(T).$$

So, by the defining property of σ, we have

$$T \vdash \mathrm{Con}(T) \to \sigma.$$

□

Observe also that $T \vdash \sigma \to \mathrm{Con}(T)$ since if $\mathrm{Con}(T)$ fails to hold, there is a proof of every sentence including σ. It follows that $T \vdash \neg\mathrm{Con}(T) \to \neg\sigma$. Thus, any theory which proves its own inconsistency decides σ! Are there theories which prove their own inconsistency? Yes! Let T be the theory $\mathrm{PA} \cup \{\neg\mathrm{Con}(\mathrm{PA})\}$. Since T extends PA, we have that $T \vdash \neg\mathrm{Con}(T)$. So Rosser's trick sentence τ was indeed necessary.

Another important property of the provability predicate $\mathrm{Pr}_T(x)$ was discovered by Martin Löb and became known as Löb's Theorem.

Theorem 5.5 (Löb's Theorem)**.** *If $T \vdash \mathrm{Pr}_T(\ulcorner\varphi\urcorner) \to \varphi$, then $T \vdash \varphi$.*

Löb's Theorem once again demonstrates the difference between true provability and provability from the perspective of a model of PA. Intuitively, we expect that if there is a proof of φ from PA, then of course, φ should be true, making the hypothesis of Löb's Theorem a tautology. But if this were the case, then PA would prove every sentence! As Rohit Parikh pointed out: "PA could't be more modest about its own veracity."

5.4. **In which we show that** $\mathrm{PA} \vdash$ **Con**(PA)**.** The sentence Con(PA), used in the statement of the Second Incompleteness Theorem, presupposes a previously agreed upon predicate $\mathrm{PA}(x)$ (not to mention the coding used to define it). To what extent does the Second Incompleteness Theorem depend on this seemingly arbitrary choice? What general properties must the predicate $\mathrm{PA}(x)$ satisfy in order for the Second Incompleteness Theorem to hold? At the very minimum, in any model of PA, the predicate $\mathrm{PA}(x)$ must hold of $n \in \mathbb{N}$ precisely when n is the Gödel-number of a PA-axiom. The predicate we defined in Section 4 had the additional property of being $\Delta_1(\mathrm{PA})$, which was used in both proofs of the Second Incompleteness Theorem. Indeed, the Second Incompleteness Theorem holds with any predicate $\mathrm{PA}^*(x)$ meeting these two conditions. This analysis makes some headway in resolving possible objections to the arbitrary choices involved in expressing Con(PA). The objections would be entirely done away with if we could remove the need for the second condition. But, alas, Solomon Fefferman showed how to define a predicate $\mathrm{PA}^*(x)$ that for $n \in \mathbb{N}$ holds precisely of the Gödel-numbers of PA-axioms and, yet, for which $\mathrm{PA} \vdash \mathrm{Con}(\mathrm{PA})$!

For $n \in \mathbb{N}$, we let the theory PA_n consist of all PA-axioms with Gödel-numbers $\leq n$. A remarkable theorem of Mostowski shows that $\mathrm{PA} \vdash \mathrm{Con}(\mathrm{PA}_n)$ for every

$n \in \mathbb{N}$.[14] Let us define the predicate $\mathrm{PA}^*(x)$ by

$$\mathrm{PA}^*(x) := \mathrm{PA}(x) \wedge \mathrm{Con}(\mathrm{PA}_x).$$

The definition seems to be saying precisely that we only take PA-axioms which do not lead to a contradiction. Now suppose that $M \models \mathrm{PA}$. Clearly $\{a \in M \mid \mathrm{PA}^*(a)\} \subseteq \{a \in M \mid \mathrm{PA}(a)\}$. Using Mostowski's theorem, it follows that for $n \in \mathbb{N}$, the predicate $\mathrm{PA}^*(x)$ holds in M exactly of the Gödel-codes of PA-axioms. Thus, the difference between $\mathrm{PA}(x)$ and $\mathrm{PA}^*(x)$ is that the later has potentially fewer nonstandard induction axioms. Now we argue that $M \models \mathrm{Con}(\mathrm{PA}^*)$. Suppose that $p \in M$ is a proof from $\mathrm{PA}^*(x)$. Let a be the largest PA^*-axiom used in p and note that p is then a proof from PA_a. Since $M \models \mathrm{PA}^*(a)$, it follows that $M \models \mathrm{Con}(\mathrm{PA}_a)$. But since p is a proof of PA_a, it cannot prove any statement of the form $\psi \wedge \neg\psi$. Thus, $M \models \mathrm{Con}(\mathrm{PA}^*)$!

An immediate consequence of Mostowski's Theorem together with the Overspill Principle is that every $M \models \mathrm{PA}$ has a nonstandard $a \in M$ such that it think PA_a is consistent! But, of course, PA_a already includes all the (standard) PA-axioms. Thus, M can build a model of PA using the Arithmetized Completeness Theorem!

5.5. **Homework.**

Question 5.1. Complete the proof of Theorem 5.3 by showing that $i(a + b) = i(a) +^N i(b)$ and $i(a \cdot b) = i(a) \cdot^N i(b)$.

Question 5.2. Prove Löb's Theorem using the Derivability Conditions.

Question 5.3. Consider a sentence ρ asserting that it is provable:

$$\mathrm{PA}^- \vdash \rho \leftrightarrow \mathrm{Pr}_{\mathrm{PA}}(\ulcorner \rho \urcorner).$$

Show that $\mathrm{PA} \vdash \rho$.

Question 5.4. Show that there is a sentence φ that is $\Delta_1(\mathbb{N})$ but not $\Delta_1(PA)$. (Hint: use Con(PA)).

6. Tennenbaum's Theorem

6.1. **A third kind of incompleteness.**

Gödel's Incompleteness Theorems established two fundamental types of incompleteness phenomena in first-order arithmetic (and stronger theories). The First Incompleteness Theorem showed that no recursive axiomatization extending PA can decide all properties of natural numbers and the Second Incompleteness Theorem showed that there cannot be a constructive proof of the consistency of PA. Both results demonstrated that inherent limitations latent in sufficiently complex formal systems will prevent us from acquiring the kind of absolute knowledge sought by Hilbert and other optimists at the start of the 20th-century. In 1959, Stanley Tennenbaum established a third type of incompleteness, when he showed that we can never construct a nonstandard model of PA. Let us say that a countable structure $\langle M, \{f_i\}_{i<n}, \{r_i\}_{i<m}, \{c_i\}_{i<k}\rangle$ of some finite first-order language is *recursive* if there is a bijection $F : M \to \mathbb{N}$ such that, identifying M with $\mathbb{N}$ via F, the functions f_i and the relations r_i are recursive. The notion of recursiveness for a first-order structure is intended to capture formally the idea that a structure is 'constructible'. Because the functions $+$, $\cdot$, and the relation

[14] The proof of Mostowski's theorem relies on advanced proof-theoretic techniques that are beyond the scope of these notes.

$<$ on $\mathbb{N}$ are recursive, the identity map witnesses that the structure $\langle \mathbb{N}, +, \cdot, <, 0, 1\rangle$ is recursive. It is also easy to see that if M is a countable nonstandard model of PA, then $\langle M, <\rangle$ is recursive. Tennenbaum's Theorem showed that there cannot be a recursive nonstandard model of PA. Essentially, there do not exist two recursive operations $+^*$ and $\cdot^*$ on $\mathbb{N}$, different from the standard $+$ and $\cdot$, obeying Peano Arithmetic. Indeed, Tennenbaum showed that if a countable $M \models \text{PA}$ is nonstandard, then, even when taken individually, neither $\langle M, +\rangle$ nor $\langle M, \cdot\rangle$ is recursive. Remarkably, (up to isomorphism) $\langle \mathbb{N}, +, \cdot, <, 0, 1\rangle$ is the unique recursive model of PA.

Theorem 6.1 (Tennenbaum's Theorem)**.** *Suppose $\langle M, +, \cdot, <, 0, 1\rangle$ is a countable nonstandard model of* PA*, then neither $\langle M, +\rangle$ nor $\langle M, \cdot\rangle$ is recursive.*

In fact, Tennenbaum's Theorem holds for number theories drastically weaker than PA. In a work published in 1985, George Wilmers showed that there are no recursive nonstandard models of PA^- together with IE_1-induction. The induction principle IE_1 states that induction holds for formulas of the form

$$\exists \overline{y} < t(\overline{x})\, t_1(\overline{x}, \overline{y}) = t_2(\overline{x}, \overline{y}),$$

where t, t_1, t_2 are some L_A-terms (polynomial functions with non-negative coefficients).

6.2. **Proof of Tennenbaum's Theorem.** We observed in Section 4 that the standard system of any nonstandard model M of PA must contain a non-recursive set (Corollary 4.21). To prove Tennenbaum's Theorem, we will argue that if either $\langle M, +^M\rangle$ or $\langle M, \cdot^M\rangle$ were recursive, then every set in $SSy(M)$ would be recursive as well.

In Section 2, we showed that $SSy(M)$ consists precisely of those subsets of $\mathbb{N}$ that are *coded* in M (Theorem 2.38). Recall that $A \subseteq \mathbb{N}$ is said to be coded in a model $M \models \text{PA}$ if there is $a \in M$ such that $n \in A$ if and only if $M \models (a)_n \neq 0$. We used the β-function coding to define the notion of a coded set, but we could have used any other coding, based say on binary expansions, just as well. For the purposes of the arguments to follow, we will use a coding based on the prime numbers sequence. Suppose that $M \models \text{PA}$ and let us argue that M can definably enumerate its prime numbers. Let $\text{Prime}(p)$ be the Δ_0-definable relation expressing that p is prime over M. The definable enumeration p_x, expressing that p is the xth-prime number of M, is given by the formula

$$\exists s\, ((s)_0 = 2 \wedge \forall i < x(\text{Prime}((s)_i) \wedge \forall z < (s)_{i+1}(z > (s)_i \to \neg\text{Prime}(z))) \wedge p = (s)_x).$$

Since the relation $\text{Prime}(p)$ is Δ_0-definable, it follows that $\mathbb{N}$ and M will agree on the enumeration of the standard prime numbers.

Lemma 6.2. *Suppose that $M \models \text{PA}$ is nonstandard. Then $A \in SSy(M)$ if and only if there is $a \in M$ such that for $n \in \mathbb{N}$, we have $p_n | a$ if and only if $n \in A$.*

Proof. Suppose that $A \in SSy(M)$. Then there is a formula $\varphi(x, \overline{y})$ and $\overline{b} \in M$ such that $A = \{n \in \mathbb{N} \mid M \models \varphi(n, \overline{b})\}$. Let $c \in M$ be nonstandard. We argue by induction on y up to c in the formula

$$\psi(y, \overline{b}) := \exists a \forall x \leq y\, p_x | a \leftrightarrow \varphi(x, \overline{b})$$

that there is $a \in M$ such that for all $x \leq c$, we have $M \models p_x | a \leftrightarrow \varphi(x, \overline{b})$. First, we verify that there is a such that $p_0 = 2 | a \leftrightarrow \varphi(0, \overline{b})$. Choose the witnessing

a so that $a = 2$ if $M \models \varphi(0, \bar{b})$, and $a = 1$ otherwise. So suppose inductively that there is $a \in M$ such that for all $x \leq y$, we have $M \models p_x | a \leftrightarrow \varphi(x, \bar{b})$. Let $a' = a \cdot p_{x+1}$ if $M \models \varphi(x+1, \bar{b})$, and $a' = a$ otherwise. Then for all $x \leq y+1$, we have $M \models p_x | a' \leftrightarrow \varphi(x, \bar{b})$. Thus, there must be a such that for all $x \leq c$, we have

$$M \models p_x | a \leftrightarrow \varphi(x, \bar{b}).$$

In particular, for $n \in \mathbb{N}$, we have $n \in A$ if and only if $M \models p_n | a$. $\square$

We are now ready to prove Tennenbaum's Theorem.

Theorem 6.3 (Tennenbaum's Theorem)**.** *Suppose that $\langle M, +, \cdot, <, 0, 1\rangle$ is a countable nonstandard model of* PA*, then neither $\langle M, +\rangle$ nor $\langle M, \cdot\rangle$ is recursive.*

Proof. We will argue that if $\langle M, +\rangle$ or $\langle M, \cdot\rangle$ was recursive, then every set in $SSy(M)$ would have to be recursive.

Suppose that $A \in SSy(M)$ and fix $a \in M$ such that $n \in A$ if and only if $M \models p_n | a$. Fix $n \in \mathbb{N}$. By the division algorithm in M, there are unique $q_n \in M$ and $r_n < p_n$ such that $a = q_n \cdot p_n + r_n$. It follows that $n \in A$ if and only if $r_n = 0$. Next, we make the crucial observation that

$$q_n \cdot p_n + r_n = \underbrace{q_n + \ldots + q_n}_{p_n\text{-many}} + \underbrace{1 + \ldots + 1}_{r_n\text{-many}}$$

because both p_n and r_n are standard. Thus, to determine whether $n \in A$, we search through all elements of M for an element q_n having the property that $a = \underbrace{q_n + \ldots + q_n}_{p_n\text{-many}} + \underbrace{1 + \ldots + 1}_{r_n\text{-many}}$ for some $r_n < p_n$ and we can verify this property using just the addition operation. Now, we need a variant property which we can verify using just the multiplication operation. Let $b = 2^a$. Observe that

$$b = 2^{q_n \cdot p_n + r_n} = (2^q)^{p_n} \cdot 2^{r_n} = \underbrace{w_n \cdot \ldots \cdot w_n}_{p_n\text{-many}} \cdot \underbrace{2 \cdot \ldots \cdot 2}_{r_n\text{-many}},$$

where $w_n = 2^{q_n}$. Thus, to determine whether $n \in A$, we search through all elements of M for an element w_n having the property that $b = \underbrace{w_n \cdot \ldots \cdot w_n}_{p_n\text{-many}} \cdot \underbrace{2 \cdot \ldots \cdot 2}_{r_n\text{-many}}$ for some $r_n < p_n$ and we can verify this property using just the multiplication operation.

Now, suppose that $\langle M, +\rangle$ is recursive and fix a bijection $F : M \to \mathbb{N}$ such that the function $+^*$ corresponding to $+$ under F is recursive. Let $F(a) = m_a$ and $F(1) = m_1$. To determine whether $n \in A$, we search for a number q such that

$$m_a = \underbrace{q +^* \ldots +^* q}_{p_n\text{-many}} +^* \underbrace{m_1 +^* \ldots +^* m_1}_{r\text{-many}}$$

for some $r < p_n$, and conclude that $n \in A$ whenever $r = 0$. It should be clear that this is a recursive procedure.

Next, suppose that $\langle M, \cdot\rangle$ is recursive and fix a bijection $F : M \to \mathbb{N}$ such that the function $\cdot^*$ corresponding to $\cdot$ under F is recursive. Let $F(b) = m_b$ and $F(2) = m_2$. To determine whether $n \in A$, we search for a number w such that

$$m_b = \underbrace{w \cdot^* \ldots \cdot^* w}_{p_n\text{-many}} \cdot^* \underbrace{m_2 \cdot^* \ldots \cdot^* m_2}_{r\text{-many}}$$

for some $r < p_n$, and conclude that $n \in A$ whenever $r = 0$. It should be clear that this is a recursive procedure. $\square$

6.3. **Homework.**

Question 6.1. Suppose that $M \models \mathrm{PA}$ is nonstandard. Show that $(M, <)$ is recursive.

7. Turing Machines and Computability

7.1. On the origin of Turing Machines.

An algorithm must be seen to be believed.

–**Donald Knuth**

In the age defined by computing, we are apt to forget that the quest to solve mathematical problems algorithmically, by a mechanical procedure terminating in finitely many steps, dates back to the beginning of human mathematical endeavors. We owe its most ambitious formulation to Leibniz, who dreamed of building a machine capable of solving any mathematical problem by manipulating the symbols of a new formal language in which all mathematics would be expressible. In the early 20th-century, with the introduction of first-order logic and the formal theory of provability, Leibniz's dream finally seemed close to realization. Hilbert had set out several goals for mechanizing mathematical reasoning and chief among them was the Entscheidungsproblem, asking for an algorithm to determine whether a first-order statement is a logical validity. Like Leibniz, Hilbert believed that no problem in mathematics was unsolvable.

> *This conviction of the solvability of every mathematical problem is a powerful incentive to the worker. We hear within us the perpetual call: There is the problem. Seek its solution. You can find it by pure reason, for in mathematics there is no ignorabimus.*

For instance, the Entscheidungsproblem algorithm for propositional logic consists of checking whether a given statement evaluates to true under all truth assignments to the propositional variables it contains. The Entscheidungsproblem was posed by Hilbert and his student Ackermann in 1928, a time when not even an agreed upon formal notion of algorithm existed. In 1935, a 22-year-old mathematics student by the name of Alan Turing heard a lecture on the Entscheidungsproblem and spent the next year searching for a solution. In the process, he elucidated a philosophical interpretation of the process of mechanical computation, implemented it with an abstract computing device, and proposed a formal definition of algorithmically computable functions based on what his devise, the Turing machine, could compute. Finally, using his definition of computability, he showed that there was no algorithm to solve the Entscheidungsproblem.[15]

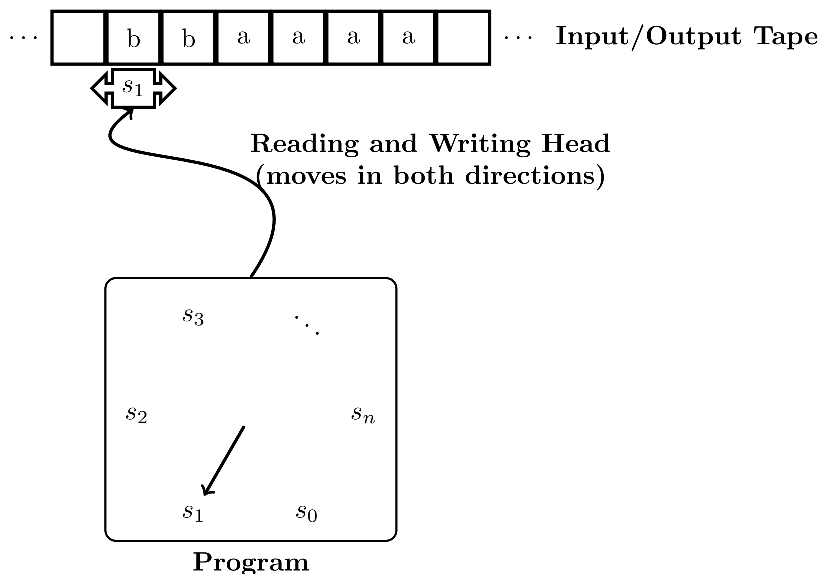
Turing decided that a computation was to be viewed as mechanical if it could be carried out by a human 'computer'[16] writing in a finite alphabet on a piece of one dimensional paper subdivided into squares, one symbol per square, in accordance with a finite list of instructions, each of which was meant to correspond to a particular 'state of mind' of the computer. The human carrying out the computation is assumed to read and modify the contents of a single square at a time, while the length of the available paper is assumed to be unlimited, so that for example when

[15] Using a notion of computability that turned out to be equivalent to Turing's, Church independently obtained the same result a few months prior.

[16] In the age before computers as we know them came along, this was a term used for a human being performing a computation.

the paper runs out, more is attached to allow the computer to continue. A 'state of mind' consists of a fixed sequence of actions to be carried out by the computer. A possible state of mind could be to move forward along the paper writing a symbol, say 'a', in each square until another symbol, say 'b', is read and once 'b' is read, to switch to a different state of mind. This new state of mind could require writing 'b' in each blank square and 'a' in each already written in square until the symbol 'c' is read (at which point, another change of states would be specified). Thus, each of the finite list of instructions provided to the computer is a description of a state of mind that specifies, based on the current contents of a square, which writing action is to be taken, whether to move forward or backward afterwards, and what new state to assume.

To formalize his notion of mechanical computation, Turing introduced an abstract computing device, the Turing machine. The hardware of a Turing machine consists of a tape, infinitely long in both directions, that is subdivided into squares, and a read/write head which moves along the tape scanning one square at a time. The actions of the head are controlled by a Turing machine program.



A Turing machine program comes with a prefixed finite alphabet, whose symbols the head is allowed to write on the tape. The program instructions define a number of states which the Turing machine can be in and for each state, how the head should respond when reading a given symbol on the tape. A given state specifies, for each possible symbol of the alphabet, what the head should write on the square being scanned upon encountering that symbol, whether it should subsequently move right or left along the tape, and whether it should at that point switch states. An individual instruction is a 5-tuple of elements consisting of the state name, the symbol being read, the symbol being written in response, the direction to move, and, finally, the new state. Specially designated 'start' and 'halt' states let the machine know how to start and end the computation. Writing Turing machine instructions is best analogous in modern computing to programming in Assembler, a language

that serves as an intermediary between machine language and the higher level programming languages programmers usually encounter. An Assembler program is subdivided into labeled regions (physically corresponding to where the instructions are stored in the memory), each of which consists of a set of simple instructions such as reading/writing to memory locations, and basic arithmetic. The 'goto (label)' instruction directs the program among the labeled regions. The labeled regions can be viewed as corresponding to machine states and the goto instruction as directing a switch to a new state.

Turing defined that a function $f : \mathbb{N}^k \to \mathbb{N}$ is computable if there is a Turing machine program, which when applied to a tape on which the n-tuple $\overline{m}$ is prewritten, reaches the halt state in finitely many steps, leaving $f(\overline{m})$ on the tape. Natural numbers are inputted and outputted on the tape using some predetermined conventions such as their binary (a finite alphabet consisting of 0 and 1) expansion.

One of the central contributions of Turing's computational model was the realization that a Turing machine program can be made to interpret (coded versions of) other Turing machine programs. Turing constructed the Universal Turing Machine program which took as input a pair consisting of a Turing Machine program and an input to that program and outputted precisely the value which that program would compute on that input. The Universal Turing Machine program essentially 'ran' the inputted program. It is generally believed that the von Neumann architecture of the stored program computer on which the modern computer is founded owes its conception to the Universal Turing Machine.

7.2. **Turing Machines.** There is no single agreed upon definition of either the Turing machine hardware or software. Many seemingly arbitrary choices are made, but remarkably all the various definitions and assumptions produce devices that compute exactly the same functions on the natural numbers. Our Turing machine hardware will consist of a tape, infinitely long in both directions, that is subdivided into squares and a read/write head that can move left or right along the tape, one square at a time.[17] Before describing a Turing machine program, we must specify the finite alphabet whose symbols the program will direct the machine to manipulate on the tape. Every such finite alphabet Σ needs to include a special 'blank' symbol to be used by the program to refer to a blank square. A Turing machine program is a finite list of instructions, each of which directs the head, based the symbol it is currently reading on the tape, what symbol it should write on the tape, whether it should move right or left, and what new state it should transition to. Suppose that S is the set of states used by a program P. The simplest way to name states is to number them 0 through n, but for purposes of program readability, it is advisable to name states by phrases intended to convey a description of the behavior they specify. Each state set needs to include two specially designated states, the 'Start' state and the 'Halt' state. Because the instructions are not executed in order, the designated Start state is necessary as means of letting the machine must know in which state to begin operation. Although, we must have some way of specifying when a program ends, the existence of the Halt state is only one such convention. Alternatively, for instance, we could have chosen to end the program when the machine transitioned to some state for which no instruction existed. Finally, the program must be *sound*: it must have exactly one instruction

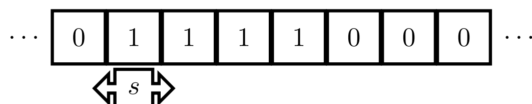
[17]For other authors, the tape only extends infinitely in the forward direction.

for every pair consisting of a state in S (other than the Halt state) and symbol in Σ.

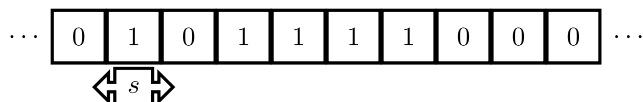
Formally, a Turing machine program with the alphabet Σ and a state set S is a finite collection P of 5-tuples $\langle s, r, w, d, s'\rangle$, where $s \in S$ is the current state, $r \in \Sigma$ is the symbol currently being read, $w \in \Sigma$ is the symbol that is supposed to replace it on the tape, $d \in \{R, L\}$ is the direction in which the head should move after writing w and s' is the state it should transition to. Moreover, for every $s \in S$ and $r \in \Sigma$, there is exactly one tuple $\langle s, r, \ldots\rangle$.

An input to a Turing machine program consists of some finitely many symbols in Σ that are pre-written on the tape before the program begins with head placed on the leftmost input symbol. The program is said to produce an output if it reaches the Halt state and the interpretation of the output is based on predetermined conventions.

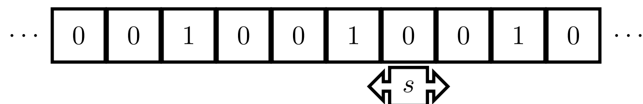
To define the class of Turing computable functions on the natural numbers, we must first choose the alphabet Σ in which the computation will take place and set up conventions for inputting and outputting natural numbers on the tape. Initially we will choose the smallest possible alphabet $\Sigma = \{0, 1\}$, consisting of just two symbols, with 0 standing in for the blank character. In the next section, we will argue that increasing the size of the alphabet, while shortening programs, does not increase computing capabilities. By convention, we will represent a natural number n on the Turing machine tape by $n + 1$-consecutive 1's. The tape below contains the input $n = 4$.



Elements of a tuple of natural numbers will be separated by a single blank square. The tape below contains the input $(0, 3)$.



We will also adapt the convention that the output of a Turing machine in the Halt state is the number n of (not necessarily consecutive) 1's left on the tape. The tape below contains the output $n = 3$.



Let us say that a program P *computes* the function $f(\overline{x})$ defined by $f(\overline{m}) = n$ if and only if P halts on input $\overline{m}$ with output n. The function f is potentially partial because P might not reach the Halt state on all inputs. With all the preliminaries complete, we finally define that a (partial) function $f : \mathbb{N}^k \to \mathbb{N}$ is (Turing) computable if there is a Turing machine program which computes it. For a computable function $f(\overline{x})$, we shall use the notation $f(\overline{x}) \downarrow$ to indicate that $f(\overline{x})$ is defined, meaning that the program computing it halts on input $\overline{x}$, and likewise we shall use the notation $f(\overline{x}) \uparrow$ to indicate that program computing it never reaches the Halt state on input $\overline{x}$. We shall say that a computable function

$f(\overline{x})$ is *total* if for all $\overline{x}$, we have $f(\overline{x})\downarrow$. We shall say that a relation is *computable* if its characteristic function is computable.

Example 7.1. The zero function $Z(x) = 0$ is computable by the program below.

$\langle \textbf{Start}, 1, 0, R, \textbf{Start}\rangle$ //Delete input

$\langle \textbf{Start}, 0, 0, R, \textbf{Halt}\rangle$ //Done!

Example 7.2. The successor function $S(x) = x + 1$ is computable by the program below.

/* For input n, we already have $n + 1$-many 1's on entered tape. Thus, the program to compute $S(x)$ needs to do absolutely nothing. */

$\langle \textbf{Start}, 1, 1, R, \textbf{Halt}\rangle$ //Do nothing

$\langle \textbf{Start}, 0, 0, R, \textbf{Halt}\rangle$ //Improper input

Example 7.3. The projection functions $P^n_i(x_1, \ldots, x_n) = x_i$ are computable. Below is a program to compute P^4_3.

/* We erase the first, second, and forth inputs by keeping track of the number of blank spaces encountered to know which input the head is currently on. */

$\langle \textbf{Start}, 1, 0, R, \textbf{DeleteFirstInput}\rangle$ //Start deleting first input

$\langle \textbf{Start}, 0, 0, R, \textbf{Halt}\rangle$ //Improper input

$\langle \textbf{DeleteFirstInput}, 1, 0, R, \textbf{DeleteFirstInput}\rangle$ //Keep deleting first input

$\langle \textbf{DeleteFirstInput}, 0, 0, R, \textbf{DeleteSecondInput}\rangle$ //Switch to deleting second input

$\langle \textbf{DeleteSecondInput}, 1, 0, R, \textbf{DeleteSecondInput}\rangle$ //Keep deleting second input

$\langle \textbf{DeleteSecondInput}, 0, 0, R, \textbf{RemoveExtra1}\rangle$ //Proceed to remove extra 1 from third input

$\langle \textbf{RemoveExtra1}, 1, 0, R, \textbf{SkipThirdInput}\rangle$ //Remove extra 1, switch to skipping third input

$\langle \textbf{RemoveExtra1}, 0, 0, R, \textbf{Halt}\rangle$ //Cannot happen

$\langle \textbf{SkipThirdInput}, 1, 1, R, \textbf{SkipThirdInput}\rangle$ //Keep reading through third input

$\langle \textbf{SkipThirdInput}, 0, 0, R, \textbf{DeleteFourthInput}\rangle$ //Switch to deleting fourth input

$\langle \textbf{DeleteFourthInput}, 1, 0, R, \textbf{DeleteFourthInput}\rangle$ //Keep deleting fourth input

$\langle \textbf{DeleteFourthInput}, 0, 0, R, \textbf{Halt}\rangle$ //Done!

We just showed that the basic primitive recursive functions are computable!

Example 7.4. Addition is computable by the program below.

/* For inputs n and m, we have $n + 1$ and $m + 1$ 1's entered on tape. To produce output $m + n$, we need to delete two extra 1's. */

$\langle \textbf{Start}, 1, 0, R, \textbf{SecondDelete}\rangle$ //First extra 1 is deleted, proceed to delete second extra 1

$\langle \textbf{Start}, 0, 0, R, \textbf{Halt}\rangle$ //Cannot happen

$\langle \textbf{SecondDelete}, 0, 0, R, \textbf{SecondDelete}\rangle$ //Space between inputs is encountered

$\langle \textbf{SecondDelete}, 1, 0, R, \textbf{Halt}\rangle$ //Done!

Example 7.5. Multiplication is computable by the program below.

/* For inputs m, n, we have $m+1$ and $n+1$ entered on tape. If m or n is 0, we delete both inputs. If $m, n \neq 0$, we make n-many copies of m-many 1's to the left of first input. Before each copy of m 1's is made, we delete a 1 from second input, which serves as counter of the number of copies to be made. We stop making copies when a single 1 is left from second input. To make a copy, we use first input as counter. Before a bit is copied, a 1 from first input is deleted. When a single 1 is left from first input, the m 1's are written back on tape. When copying is finished, we delete first and second inputs, leaving precisely n-many copies of m on tape. */

$\langle \textbf{Start}, 1, 1, R, \textbf{IsFirstZero}\rangle$ //Check if first input is 0

⟨**Start**, 0, 0, R, **Halt**⟩ //Improper input

/* First input is 0 */

⟨**IsFirstZero**, 0, 0, L, **DeleteFirst**⟩ //First input is 0, delete first input

⟨**DeleteFirst**, 1, 0, R, **FindSecondToDelete**⟩ //First input is 0, delete first input

⟨**DeleteFirst**, 0, 0, R, **Halt**⟩ //Cannot happen

⟨**FindSecondToDelete**, 0, 0, R, **FindSecondToDelete**⟩ //Find second input to delete

⟨**FindSecondToDelete**, 1, 0, R, **DeleteSecond**⟩ //Proceed to delete second input

⟨**DeleteSecond**, 1, 0, R, **DeleteSecond**⟩

⟨**DeleteSecond**, 0, 0, R, **Halt**⟩ //Done!

/* First input is not 0 */

⟨**IsFirstZero**, 1, 1, R, **FindSecondInput**⟩ //First input is not 0, proceed to second input

⟨**FindSecondInput**, 1, 1, R, **FindSecondInput**⟩ //Reading first input

⟨**FindSecondInput**, 0, 0, R, **DecreaseSecondInput**⟩ //Proceed to decrease second input counter

⟨**DecreaseSecondInput**0, 0, R, **Halt**⟩ //Cannot happen

⟨**DecreaseSecondInput**1, 0, R, **IsSecondZero**⟩ //Check if second input is 0

/* Second input is 0 */

⟨**IsSecondZero**, 0, 0, L, **FindFirstToDelete**⟩ //Go back, find first input

⟨**FindFirstToDelete**, 0, 0, L, **FindFirstToDelete**⟩

⟨**FindFirstToDelete**, 1, 0, L, **DeleteFirstInput**⟩ //First input found, proceed to delete

⟨**DeleteFirstInput**, 1, 0, L, **DeleteFirstInput**⟩

⟨**DeleteFirstInput**, 0, 0, L, **Halt**⟩ //Done!

/* Second input is not 0 */

⟨**IsSecondZero**, 1, 1, L, **FindFirstInput**⟩ //Go back, find first input

⟨**FindFirstInput**, 0, 0, L, **FindFirstInput**⟩

⟨**FindFirstInput**, 1, 1, L, **SkipFirstInput**⟩ //Proceed to skip first input

⟨**SkipFirstInput**, 1, 1, L, **SkipFirstInput**⟩ //Read through first input

/* Make copy of first input */

/* Copy first bit */

⟨**SkipFirstInput**, 0, 0, R, **StartCopy**⟩ //Prepare to copy first input

⟨**StartCopy**, 0, 0, L, **Halt**⟩ //Cannot happen

⟨**StartCopy**, 1, 0, L, **SkipSpace**⟩ //erase bit of first input, proceed to skip to second copy location

⟨**SkipSpace**, 1, 1, L, **Halt**⟩ //Cannot happen

⟨**SkipSpace**, 0, 0, L, **FindEmptySpace**⟩ //Proceed to skip through copies previously made

⟨**FindEmptySpace**, 1, 1, L, **SkipThroughCopy**⟩ //Skip through copy

⟨**SkipThroughCopy**, 1, 1, L, **SkipThroughCopy**⟩

⟨**SkipThroughCopy**, 0, 0, L, **FindEmptySpace**⟩

⟨**FindEmptySpace**, 0, 1, R, **ForwardToFirst**⟩ //Bit is copied, search for double 0 to locate first input

⟨**ForwardToFirst**, 1, 1, R, **ForwardToFirst**⟩

⟨**ForwardToFirst**, 0, 0, R, **FindDoubleZero**⟩

⟨**FindDoubleZero**, 1, 1, R, **ForwardToFirst**⟩

⟨**FindDoubleZero**, 0, 0, R, **SkipZeroes**⟩ //Skip 0's to first input

⟨**SkipZeroes**, 0, 0, R, **SkipZeroes**⟩

⟨**SkipZeroes**, 1, 1, R, **IsEndOfCopy**⟩ //First input is found, check whether at least two 1's left

/* Copy next bit */

$\langle \mathbf{IsEndofCopy}, 1, 1, L, \mathbf{DeleteBit} \rangle$ //Copy not finished, proceed to delete next bit

$\langle \mathbf{DeleteBit}, 0, 0, L, \mathbf{Halt} \rangle$ //Cannot happen

$\langle \mathbf{DeleteBit}, 1, 0, L, \mathbf{FindFirstCopy} \rangle$ //Find start of first copy

$\langle \mathbf{FindFirstCopy}, 0, 0, L, \mathbf{FindFirstCopy} \rangle$

$\langle \mathbf{FindFirstCopy}, 1, 1, L, \mathbf{SkipThroughCopy} \rangle$ //Start to loop

/* Copy finished, reconstitute first input */

$\langle \mathbf{IsEndOfCopy}, 0, 0, L, \mathbf{SkipBit} \rangle$ //Copy is finished, skip back to end of first input

$\langle \mathbf{SkipBit}, 0, 0, L, \mathbf{Halt} \rangle$ //Cannot happen

$\langle \mathbf{SkipBit}, 1, 1, L, \mathbf{ReenterInput} \rangle$ //Reenter input

$\langle \mathbf{ReenterInput}, 0, 1, L, \mathbf{ReenterInput} \rangle$

$\langle \mathbf{ReenterInput}, 1, 1, R, \mathbf{CorrectError} \rangle$ //Input reentered, go back to delete extra 1

$\langle \mathbf{CorrectError}, 0, 0, R, \mathbf{Halt} \rangle$ //Cannot happen

$\langle \mathbf{CorrectError}, 1, 0, R, \mathbf{CountThroughFirst} \rangle$ //Read through first input

/* Check second input counter, decrease if not zero */

$\langle \mathbf{CountThroughFirst}, 1, 1, R, \mathbf{CountThroughFirst} \rangle$

$\langle \mathbf{CountThroughFirst}, 0, 0, R, \mathbf{SkipZeroesBetween} \rangle$ //Skip zeroes between inputs

$\langle \mathbf{SkipZeroesBetween}, 0, 0, R, \mathbf{SkipZeroesBetween} \rangle$

$\langle \mathbf{SkipZeroesBetween}, 1, 1, R, \mathbf{IsEndOfSecond} \rangle$ //Check whether counter is zero

$\langle \mathbf{IsEndOfSecond}, 1, 1, L, \mathbf{Decrease} \rangle$ //Copying not done, proceed to decrease counter

$\langle \mathbf{Decrease}, 0, 0, L, \mathbf{Halt} \rangle$ //Cannot happen

$\langle \mathbf{Decrease}, 1, 0, L, \mathbf{FindFirstInput} \rangle$ //Now loop

/* Second input counter is zero, delete inputs to finish */

$\langle \mathbf{IsEndOfSecond}, 0, 0, L, \mathbf{DeleteRemainingBit} \rangle$ //Delete remaining bit of second input

$\langle \mathbf{DeleteRemainingBit}, 0, 0, L, \mathbf{Halt} \rangle$ //Cannot happen

$\langle \mathbf{DeleteRemainingBit}, 1, 0, L, \mathbf{SkipToFirst} \rangle$ //Skip 0's to first input

$\langle \mathbf{SkipToFirst}, 0, 0, L, \mathbf{SkipToFirst} \rangle$

$\langle \mathbf{SkipToFirst}, 1, 0, L, \mathbf{DeleteToEnd} \rangle$ //Delete first input

$\langle \mathbf{DeleteToEnd}, 1, 0, L, \mathbf{DeleteToEnd} \rangle$

$\langle \mathbf{DeleteToEnd}, 0, 0, L, \mathbf{Halt} \rangle$ //Done!

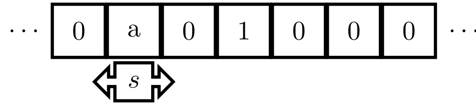
7.3. **Bigger and better Turing machines.** In the previous section, we defined that a function on the natural numbers is (Turing) computable if it can be computed by a Turing machine program using the smallest possible alphabet. Does increasing the size of the alphabet broaden the class of computable functions? It is a first testament to the robustness of Turing's definition that indeed it does not. The addition of more symbols decreases the length and improves the readability of programs, but it does not affect the computing power. Hardware embellishments also fail to increase computing power of Turing machines. Turing machines with multiple tapes, multiple heads, or even a two dimensional tape and a head that moves up and down as well as right and left, all have the same computational power. The reason is that all these improvements of software and hardware can be simulated on the bare bones Turing machine with the minimal alphabet by coding the extended alphabets and appropriately partitioning the infinite tape. Below we show how to simulate Turing machines with extended alphabets and multiple tapes.

These arguments give the general strategy of how to simulate the other varieties as well.

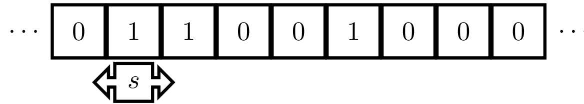
Suppose, for the sake of concreteness, that we would like to simulate a Turing machine program in the extended alphabet $\Sigma' = \{0, 1, a\}$ with a Turing machine program in Σ. First, we code each of the three symbols of Σ' by a pair of symbols in the minimal alphabet $\Sigma = \{0, 1\}$, say, as in the table below:

Symbol	Code
0	00
1	10
a	11

Now by viewing every two squares of our tape as a single square, we can simulate a program P' using Σ' by the program P with just Σ. Each time P' needs to check whether the head is reading a particular symbol on the tape, the program P would wait for the head to move through two squares, and each time P' needs the head to move left or right, the program P would send it in that direction twice. So suppose that the tape is prepared with input for program P' with some finitely many symbols from Σ', as for example, below.



First, we rewrite the input in Σ, using the above coding to obtain the tape below.



Next, we translate each instruction of program P' into several instructions of the new program P with the alphabet Σ. Suppose, for example, that P' contains an instruction $\langle \mathbf{StateA}, a, 1, R, \mathbf{StateB} \rangle$, letting the head know that when in state **StateA** and reading the symbol a, it should write the symbol 1, move right, and switch to state **StateB**. The instruction is translated into the list of instructions:

$\langle \mathbf{StateAread1}, 1, 1, R, \mathbf{StateAread2} \rangle$ //Check that the next two squares hold $a =$ '11'

$\langle \mathbf{StateAread2}, 1, 1, L, \mathbf{StateAwrite1} \rangle$

$\langle \mathbf{StateAwrite1}, 1, 1, R, \mathbf{StateAwrite2} \rangle$ //Write 1 = '10' instead of $a =$ '11'

$\langle \mathbf{StateAwrite2}, 1, 0, R, \mathbf{StateBread1} \rangle$

It should be fairly clear that the same translation strategy will work for an alphabet containing any finite number n of symbols, while coding each symbol by an m-tuple of Σ-symbols, with m chosen so that $n < 2^m$.

One practical use of an extra symbol in the alphabet is to mark the furthest locations to the right and and to the left accessed by the head. Suppose that we are given a program P in the alphabet Σ and we would like to write a program P' in the alphabet $\Sigma' = \{0, 1, m\}$, which carries out the exact same computation as P but keeps track of the furthest right and left locations on the tape accessed by the head with the symbol m. The program P' begins by writing the symbol m to the left of the beginning and to the right of the end of the input. Next, it returns the head to the beginning of the input in the state **Start***and we rewrite all instructions

of P involving the Start state using the $\mathbf{Start*}$state. Now suppose that the program P contains an instruction $\langle \mathbf{StateA}, r, w, R, \mathbf{StateB} \rangle$. We replace this instruction by the following list of instructions, which has the effect of moving the marker one cell to the right whenever necessary.

$\langle \mathbf{StateA}, r, w, R, \mathbf{StateACheckMarkerRight} \rangle$ //Check whether cell to the right contains m

$\langle \mathbf{StateACheckMarkerRight}, 0, 0, L, \mathbf{StateAMoveForward}$ //The cell doesn't contain m

$\langle \mathbf{StateAMoveForward}, 0, 0, R, \mathbf{StateB} \rangle$

$\langle \mathbf{StateAMoveForward}, 1, 1, R, \mathbf{StateB} \rangle$

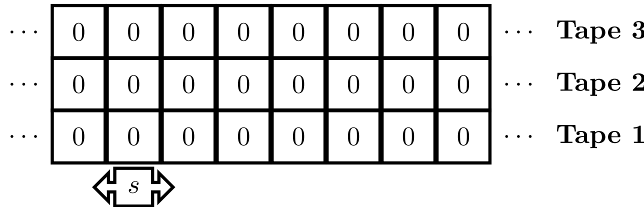
$\langle \mathbf{StateACheckMarker}, 1, 1, L, \mathbf{StateAMoveBack} \rangle$ //The cell doesn't contain m

$\langle \mathbf{StateACheckMarker}, m, 0, R, \mathbf{StateAWriteMarker} \rangle$ //Move marker to the right

$\langle \mathbf{StateAWriteMarker}, 0, m, L, \mathbf{StateB} \rangle$ //Continue computing

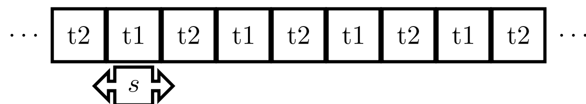
We make a similar replacement for an instruction involving a left movement. Once the we have marked the furthest locations accessed by the head, we can, for example, rewrite the output as a contiguous block of 1's. This is necessary if we want to follow our program with another program that uses its output as an input, as in the case of function composition.

The most common extension of the original Turing machine hardware is by the addition of finitely many parallel running tapes. Such a *multi-tape* Turing machine consists of n-many tapes lined up in parallel and a head that can read and write simultaneously on the vertical row of squares formed by the tapes. Here is what a 3-tape machine hardware looks like.

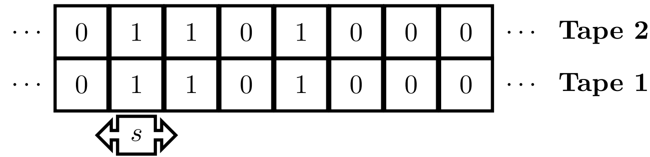


A program for an n-tape Turing machine uses n-tuples in the read and write slots of its 5-tuple instructions. For example, a program instruction for a 2-tape Turing machine, $\langle \mathbf{StateA}, \langle 0, 1 \rangle, \langle 1, 0, \rangle, R, \mathbf{StateB} \rangle$, would be letting the head know that if it is reading 0 on Tape 1 and reading 1 on the Tape 2, then it should write 1 on Tape 1 and write 0 on Tape 2.

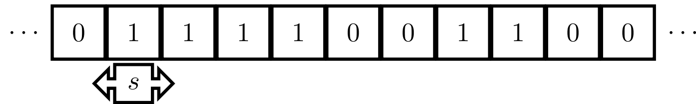
Suppose, for the sake of concreteness that we would like to simulate a program for a 2-tape Turing machine with a program for the basic Turing machine. First, we partition the single infinite tape into 2 infinite tapes, Hilbert Grand hotel style, by viewing every other square of the original tape as the second tape.



Now viewing every other square starting at the initial head position as Tape 1 and the remaining squares as Tape 2, we can simulate a program P' for a 2-tape Turing machine with the program P for the basic Turing machine. Each time P' needs to check whether the head is reading a particular tuple of symbols on the two tapes, the program P would wait for the head to move through two squares, and each time P' needs the head to move left or right, the program P would send it in the same direction twice. So suppose that the tape is prepared with input for program P', as, for example, below.



First, we transfer the input to the single tape machine, using alternate squares to simulate the two tapes.



Next, we translate each instruction of program P' into several instructions of the new program P for the basic machine. Suppose, for example, that P' contains an instruction $\langle \mathbf{StateA}, \langle 1,1\rangle, \langle 1,0\rangle, R, \mathbf{StateB}\rangle$, letting the head know that when in state **StateA** and reading 1 on Tape 1 and 1 on Tape 2, it should write 1 on Tape 1 and 0 on Tape 2, move right, and switch to state **StateB**. The instruction is translated into the list of instructions:

$\langle \mathbf{StateAread1}, 1, 1, R, \mathbf{StateAread2}\rangle$ //Check for 11

$\langle \mathbf{StateAread2}, 1, 1, L, \mathbf{StateAwrite1}\rangle$

$\langle \mathbf{StateAwrite1}, 1, 1, R, \mathbf{StateAwrite2}\rangle$ //Write 10 instead of 11

$\langle \mathbf{StateAwrite2}, 1, 0, R, \mathbf{StateBread1}\rangle$

It should be fairly clear that the Hilbert Grand Hotel strategy will work for any finite number of tapes and indeed for the two dimensional Turing machine as well. In practice, the extra tapes offered by the multi-tape Turing machines are used for scratch work and storing the results of intermediate computations.

7.4. **Arithmetizing Turing Machines.** Just as the arithmetization of first-order logic formed the basis of the Incompleteness Theorems, the seemingly mundane idea that a program can be coded by a natural number formed the basis of computability theory and the functioning of the physical computer.

Recall that a Turing machine program consists of 5-tuples $\langle s, r, w, d, s'\rangle$ where $s \in S$ (the state set) is the current state, $r \in \Sigma$ is the symbol the head is reading, $w \in \Sigma$ is the symbol the head should write, $d \in \{L, R\}$ is the direction the head should move, and s' is the state the head should transition to. Recall also that a Turing machine program always has the two special states, Start and Halt, in its state set. Let us fix the alphabet $\Sigma = \{0, 1\}$. We can assume that the state set $S = \{0, \ldots, n\}$ for some $n \geq 1$, that 0 denotes the Start state, and that 1 denotes the Halt state. We can also denote the left movement by 0 and the right movement by 1. Under these assumptions, a Turing machine program consists of 5-tuples of natural numbers. Each such 5-tuple can be coded by a single natural number using the extended Cantor's pairing function and the entire program can then be coded as a single natural number using the β-function. Since the instructions of a Turing machine program are not ordered, different ordering of the tuples will produce different codes for the program. Thus, a Turing machine program can have (finitely) many different codes.

Suppose that $f(\overline{x})$ is a computable function and P is a program that computes f. If e is some code of P, then we shall say that e is an *index* of $f(\overline{x})$. By augmenting

a program computing $f(\overline{x})$ with instructions for states that are never reached, we can obtain longer and longer programs all computing $f(\overline{x})$. Thus, a computable function $f(\overline{x})$ has infinitely many different indices. To make every natural number the index of some computable function, let us stipulate that any natural number that does not code a sound program is the index of the empty function. The n-ary function computed by the program coded by an index e will be denoted by $\varphi_e^{(n)}$[18].

Our first use of arithmetization will be to show that every Turing computable function is Σ_1-definable and therefore recursive.

7.5. **Turing machines and recursive functions.** Before Turing's introduction of his Turing machines, the two existing models of computation were Gödel's recursive functions and Church's λ-Calculus. Church had used λ-Calculus to obtain a negative solution to the Entscheidungsproblem just shortly before Turing developed his ideas. Church also showed that the class of recursive functions was identical to the class of his λ-definable functions and Turing later showed the Turing computable functions again produced precisely this class. But Turing provided such a persuasive philosophical justification for his conception of computability that it was this last equivalence that finally succeeded in convincing Gödel that every algorithmically computable function was recursive. In this section, we prove that the class of Turing computable functions is identical to the class of Gödel's recursive functions.

Theorem 7.6. *Every recursive function is Turing computable.*

Proof. We already showed that the basic functions are computable. In what follows, if P is a program in the alphabet Σ, then we will call P' the modification of this program in the alphabet $\Sigma' = \{0, 1, m\}$ which marks the furthest left and right locations accessed by the head.

Next, we show that the composition of computable functions is computable. So we suppose that $g_i(\overline{x})$, for $1 \leq i \leq n$, are computed by programs P_i and $h(x_1, \ldots, x_n)$ is computed by program P. Now we argue that the composition function $g(\overline{x}) = h(g_1(\overline{x}), \ldots, g_n(\overline{x}))$ is computable. We will work with an $n+1$-tape Turing machine using the alphabet $\Sigma = \{0, 1, m\}$.
Tapes $2, \ldots, n+1$ will be used to compute $g_i(\overline{x})$, while Tape 1 will be used to compute $h(g_1(\overline{x}), \ldots, g_n(\overline{x}))$. The extra symbol m will be used to mark the furthest left and right locations accessed by the head in the manner previously described. So suppose that Tape 1 is prepared with the input $\overline{x}$. We start by marking the square immediately to left of the initial head position on Tape 1 with m. Next, we copy over the input $\overline{x}$ to each Tape 2,..., Tape $n+1$, and return to the initial head position. Now, we run the program P_1' on Tape 2 and whenever P_1' halts, we use the markers m to rewrite all remaining 1's on the tape as a contiguous block starting at the initial head position, and add an extra 1 to the contiguous block. We repeat this routine with the remaining programs P_i. Provided that all P_i' halted, we should now have on Tape $i+1$, the value $a_i + 1$, where $g_i(\overline{x}) = a_i$. Next, we copy the a_i over to Tape 1, using the markers m to keep track of our location, and return the head to the initial head position. Finally, we run the program P on Tape 1 to compute $h(a_1, \ldots, a_n)$.

Next, we show that the function obtained by primitive recursion from computable functions is computable. So suppose that $g(\overline{x})$ is computed by program P_1 and

[18] If the superscript is omitted, its value is assumed to be 1.

$h(\overline{x}, w, z)$ is computed by program P_2. Now we argue that the function $f(\overline{x}, y)$ obtained by primitive recursion such that

$$f(\overline{x}, 0) = g(\overline{x})$$
$$f(\overline{x}, y+1) = h(\overline{x}, y, f(\overline{x}, y))$$

is computable. We will work with a 4-tape Turing machine using the alphabet $\Sigma = \{0, 1, m\}$. We will compute $f(\overline{x}, y)$ by consecutively computing $f(\overline{x}, n)$ for all $n \leq y$. Tape 4 will store the input $\overline{x}, y$, Tape 3 will store the counter n, and Tape 2 will store $f(\overline{x}, n-1)$. The consecutive computations of $f(\overline{x}, n)$ will take place on Tape 1. As before, the symbol m will be used to mark the furthest locations accessed by the head. So suppose that Tape 1 is prepared with the input $\overline{x}, y$. We start by marking the square immediately to the left of the initial head position on Tape 4 with m. Next, we copy $\overline{x}, y$ over to Tape 4 and write 1 (to indicate $n = 0$) on Tape 3 at the initial head position. Now, we erase y from Tape 1 and run the program P_1' to compute $f(\overline{x}, 0) = g(\overline{x})$. In the following iterations of the loop, we will run the program P_2' to obtain $f(\overline{x}, n)$. If $f(\overline{x}, n)$ halts, we compare the value on Tape 2 with the value on Tape 4 and halt if they are equal. Otherwise, we increment the counter on Tape 3. Next, we use the markers m to copy $f(\overline{x}, n)$ over to Tape 2 as a contiguous block of 1's starting at the initial head position, and add an extra 1 to the contiguous block. At the same time, we erase the computation of $f(\overline{x}, n)$ from Tape 1. Now, we prepare Tape 1 with $\overline{x}$ (copied from Tape 4), n (copied from Tape 3), and $f(\overline{x}, n)$ (copied from Tape 2) and loop until $n = y$.

Next, we show that the function obtained from a computable function using the μ-operator is computable. So suppose that $g(x, \overline{y})$ is computed by program P. Now we argue that the function $f(\overline{y})$ is obtained from g via the μ-operator

$$f(\overline{y}) = \mu x[g(x, \overline{y}) = 0] = \begin{cases} \min\{x \mid g(x, \overline{y}) = 0\} & \text{if exists,} \\ \text{undefined} & \text{otherwise,} \end{cases}$$

is computable. We will work with a 3-tape Turing machine using the alphabet $\Sigma = \{0, 1, m\}$. We will compute $f(\overline{y})$ by consecutively computing $g(n, \overline{y})$ starting with $n = 0$. Tape 3 will store the input $\overline{y}$ and Tape 2 will store the counter n. The consecutive computations of $g(n, \overline{y})$ will take place on Tape 1. The symbol m will be used as before. So suppose that Tape 1 is prepared with input $\overline{y}$. We start by marking the square immediately to the left of the initial head position on Tape 3 with m. Next, we copy $\overline{y}$ over to Tape 3 and write 1 (to indicate $n = 0$) on Tape 2 at the initial head position. Now, we prepare Tape 1 with the input n followed by $\overline{y}$ and run program P' to compute $g(n, \overline{y})$. If $g(n, \overline{y})$ halts, we check whether the output is equal to 0. If the output is 0, we erase the computation of $g(n, \overline{y})$ from Tape 1, copy over to Tape 1 the value on Tape 2 at the initial head position and halt. Otherwise, we increment the counter n on Tape 2 and loop. □

Theorem 7.7. *Every Turing computable function is recursive.*

Proof. We will argue that every computable function is $\Sigma_1^{\mathbb{N}}$ by defining what it means for a sequence to witness a Turing computation. Fix an index e coding some program P with n-many states. We will think of a Turing machine running the program P as carrying out a single instruction in a single instant of time. At time $t = 0$, the machine is in the Start state, and the head is located at the start of the input. At time $t = 1$, the first instruction has been carried out, and at time $t = n$, the nth-instruction has been carried out. We will think of the *snapshot* of a Turing

machine taken at time $t = n$ as a total record of the nth-step of the computation. It will consist of the portion of the tape traversed by the Turing machine since the start of the operation, the current position of the head, and the current state. The sequence of consecutive snapshots starting at time $t = 0$ and obeying the instructions of P will serve as the witness to a Turing machine computation. A Turing machine running the program P on input $\overline{x}$ halts with input y if and only if there is a witnessing sequence of snapshots, starting with input $\overline{x}$ on the tape in state 0 and halting with y on the tape in state 1. The Σ_1-definition of $\varphi_e^{(m)}(\overline{x}) = y$ will express that there exists a sequence of snapshots witnessing that P computes y on input $\overline{x}$. Now let us make all these notions precise.

We shall say that s is a *sequence of snapshots* if all its elements $[s]_i$ are triples $\langle T_i, P_i, S_i\rangle$, where T_i codes a binary sequence and $P_i < \text{len}(T_i)$. The element T_i is intended to represent the used portion of the tape, the element P_i is intended represent the head position, and the element S_i is intended to represent the current state. Suppose that an index e codes the program P with n-many states. We shall say that a sequence of snapshots s *obeys* e if the following list of requirements is satisfied. If $S_i = 1$ (Halt state), then $[s]_{i+1} = [s]_i$ (once the machine reaches the Halt state, it remains frozen for all future time instances). Suppose that $[T_i]_{P_i} = r$ and P contains the instruction $\langle S_i, r, w, d, S\rangle$.

(1) $\forall i < \text{len}(s)\, S_i < n$ (P has n-states)
(2) P_0 is the position of the leftmost 1 in T_0 and $S_0 = 0$ (the Start state)
(3) $S_{i+1} = S$
(4) if $d = 1$: (right movement)
  (a) $P_{i+1} = P_i + 1$ (head moves right)
  (b) if $P_i < \text{len}(T_i - 1)$, then $\text{len}(T_{i+1}) = \text{len}(T_i)$, $[T_{i+1}]_p = [T_i]_p$ for all $p \neq P_i$, and $[T_{i+1}]_{P_i} = w$
  (c) if $P_i = \text{len}(T_i - 1)$, then $\text{len}(T_{i+1}) = \text{len}(T_i) + 1$, $[T_{i+1}]_p = [T_i]_p$ for all $p < \text{len}(T_i - 1)$, $[T_{i+1}]_{P_i} = w$, $[T_{i+1}]_{P_i+1} = 0$ (extend the tape by adding a 0-square on the right)
(5) if $d = 0$: (left movement)
  (a) if $P_i > 0$, then $\text{len}(T_{i+1}) = \text{len}(T_i)$, $[T_{i+1}]_p = [T_i]_p$ for all $p \neq P_i$, $[T_{i+1}]_{P_i} = w$, and $P_{i+1} = P_i - 1$
  (b) if $P_i = 0$, then $\text{len}(T_{i+1}) = \text{len}(T_i) + 1$, $[T_{i+1}]_{p+1} = [T_i]_p$ for all $1 < p < \text{len}(T_i)$, $[T_{i+1}]_1 = w$, $[T_{i+1}]_0 = 0$, and $P_{i+1} = 0$ (extend the tape by adding a 0-square on the left)

Thus, $\varphi_e^{(m)}(\overline{x}) = y$ if and only if there exists a sequence of snapshots obeying e of some length n with $\overline{x}$ on T_0, y on T_{n-1}, and $S_{n-1} = 1$. Clearly this is a Σ_1-definition. □

7.6. **Universal Turing Machine.** An inherent flaw of any physical computing devise modeled on the Turing machine is that it would have to be, in some sense, 'rewired' each time it runs a different program. This devise wouldn't even be able to function as a calculator simultaneously for both addition and multiplication. That same flaw had probably, since the time of Leibniz, prevented human beings from building a computing devise. Thus, it was one of Turing's most remarkable realizations that a single program, the Universal Turing Machine (UTM), could simulate any other program, meaning that a Turing machine wired to run the UTM would be a universal computer for all other computable functions! The

modern computer is modeled on the von Neumann stored program architecture which owes its conception to the UTM. The UTM takes an input (e, a) and outputs $\varphi_e^{(n)}([a]_0, \ldots, [a]_{n-1})$, where $n = \text{len}(a)$. The existence of the UTM follows almost immediately from the witnessing sequence of snapshots concept developed in the previous section.

Theorem 7.8. *There exists a Universal Turing Machine.*

Proof. By Theorem 7.6, it suffices to show that the UTM is $\Sigma_1^{\mathbb{N}}$. Given an index e and a sequence of inputs a, the program P coded by e returns y on the inputs coded by a if and only if there exists a sequence of snapshots of length n obeying e with $\overline{x}$ on T_0, y on T_{n-1}, and $S_{n-1} = 1$. □

8. The Recursion Theorem

With the arithmetization of Turing machine programs, we get a computability theoretic analogue of Gödel's Diagonalization Lemma, known as the Recursion Theorem. The Recursion Theorem, due to Stephen Kleene from 1938, states that for any total computable function $f(x)$, there is an index e such that the program coded by e and and the program coded by $f(e)$ compute the same function.

Theorem 8.1 (Recursion Theorem)**.** *Suppose that $f(x)$ is a total computable function. Then there is $e \in \mathbb{N}$ such that $\varphi_e = \varphi_{f(e)}$.*

Proof. The proof of the Recursion Theorem is identical to the proof of the Diagonalization Lemma, using indices in place of Gödel-numbers and computable functions in place of first-order formulas. Let us define a function $h(x)$ which on input x, computes the code of the following program P. The program P on input z first computes $\varphi_x(x)$. If $\varphi_x(x)\downarrow= a$, then P runs the program coded by a on input z. Thus, whenever $\varphi_x(x)$ is defined, we have that

$$\varphi_{h(x)} = \varphi_{\varphi_x(x)}.$$

The function h is clearly total computable. Now consider the composition $f(h(x))$, which is a computable function and must therefore have an index e'. Let $e = h(e')$. Since e' is the index of a total computable function, we have that

$$\varphi_e = \varphi_{h(e')} = \varphi_{\varphi_{e'}(e')} = \varphi_{f(h(e'))} = \varphi_{f(e)}.$$

□

Example 8.2. There are index sets e such that:

(1) $\varphi_e = \varphi_{2e}$
(2) $\varphi_e = \varphi_{e+1}$
(3) $\varphi_e = \varphi_{2^e}$

A generalized version of the Recursion Theorem will appear in the homework.

Arguably, one of the most used consequences of the Recursion Theorem is that any 2-ary computable function $f(x, y)$ can guess an index for one of its projections.

Lemma 8.3. *If $f(x, y)$ is a computable function, then there is a total computable function $g(x)$ such that $f(x, y) = \varphi_{g(x)}(y)$.*

Proof. The function $g(x)$ should, on input a, compute an index of the projection $f(a,y)$. Suppose that $f(x,y)=\varphi_e^{(2)}(x,y)$ and the index e codes the program P. The function $g(x)$ on input a computes the code of a program which on input y writes a followed by y on the tape, places the head at the start of the input, and runs program P. □

A generalized version of Lemma 8.3 is known as the s-m-n-Theorem and will appear in the homework.

Theorem 8.4 (Guessing Theorem)**.** *If $f(x,y)$ is a computable function, then there is an index e such that $\varphi_e(y)=f(e,y)$.*

Proof. Let $g(x)$ be the total computable function from the proof of Lemma 8.3 for $f(x,y)$. Then by the Recursion Theorem, there is an index e such that $\varphi_e=\varphi_{g(e)}$. It now follows that $f(e,y)=\varphi_{g(e)}(y)=\varphi_e(y)$. □

We will soon encounter some remarkable applications of the Guessing Theorem.

8.1. **Examples of non-computable sets.** In the same way that arithmetization of first-order logic gives formulas the ability to reason about properties of formulas, the arithmetization of Turing machines makes it possible for algorithms to analyze properties of algorithms. When presented with a property of computable functions, we can inquire whether it is computable to determine that a function with a given index has this property. Unfortunately, for most interesting properties of computable functions, the set of all indices of functions having that property is not itself computable. The non-computability of a large class of such properties follows from Rice's Theorem, due to Henry Rice from 1951.

Suppose that $\mathcal{A}$ is a set of unary computable functions. We shall say that the *index set* of $\mathcal{A}$ is the set $A=\{e\mid\varphi_e\in\mathcal{A}\}$ of all indices of members of $\mathcal{A}$.

Example 8.5. Here are some examples of index sets:

(1) $\text{Tot}=\{e\mid\varphi_e\text{ is total}\}$
The set Tot is at most $\Pi_2^{\mathbb{N}}$ because it is defined by the formula
$$\delta(e):=\forall n\exists s\text{ sequence of snapshots witnessing }\varphi_e(n)\downarrow.$$
(2) $K_1=\{e\mid\exists n\,\varphi_e(n)\downarrow\}$
The set K_1 is at most $\Sigma_1^{\mathbb{N}}$ because it is defined by the formula
$$\delta(e):=\exists n\exists s\text{ sequence of snapshots witnessing }\varphi_e(n)\downarrow.$$
(3) $\text{Fin}=\{e\mid\varphi_e\text{ has a finite domain}\}$
The set Fin is at most $\Sigma_2^{\mathbb{N}}$ because it is defined by the formula
$$\delta(e):=\exists n\forall s\forall m>n\;s\text{ does not witness that }\varphi_e(m)\downarrow.$$
(4) $Z=\{e\mid\varphi_e(x)=0\}$
The set Z is at most $\Pi_2^{\mathbb{N}}$ because it is defined by the formula
$$\delta(e):=\forall n\exists s\text{ sequence of snapshots witnessing }\varphi_e(n)=0.$$
(5) $S=\{e\mid\exists x\,\varphi_e(x)=0\}$
The set S is at most $\Sigma_1^{\mathbb{N}}$ because it is defined by the formula
$$\delta(e):=\exists n\exists s\text{ sequence o snapshots witnessing }\varphi_e(n)=0.$$

Observe that not every subset of $\mathbb{N}$ is an index set because an index set I must have the property that whenever $e\in I$ and $\varphi_e=\varphi_{e'}$, then $e'\in I$.

Theorem 8.6 (Rice's Theorem)**.** *If A is an index set such that $A \neq \emptyset$ and $A \neq \mathbb{N}$, then A is not computable.*

By Rice's Theorem, none of the sets from Example 8.5 are computable!

Proof. Suppose that A is an index set such that $e \in A$ and $\overline{e} \notin A$. Suppose towards a contradiction that A is computable. Consider the function $h(x, y)$ which returns $\varphi_{\overline{e}}(y)$ if $x \in A$ and $\varphi_e(y)$ otherwise. Since we assumed that A is computable, the function $h(x, y)$ is computable as well. Thus, by Lemma 8.4, there is an index e' such that $\varphi_{e'}(y) = h(e', y)$. Since e' is an index, it must be that either $e' \in A$ or $e' \notin A$. If $e' \in A$, then $\varphi_{e'}(y) = h(e', y) = \varphi_{\overline{e}}(y)$ and $\overline{e} \notin A$. If $e' \notin A$, then $\varphi_{e'}(y) = h(e', y) = \varphi_e(y)$ and $\overline{e} \in A$. Thus, we have reached a contradiction, showing that A is not computable. □

Rice's Theorem still leaves open the possibility that interesting properties of indices that are not translatable into properties of index sets are computable. For instance, we could ask whether we can compute the domain of a given function. More generally, we can ask whether there is a uniform algorithm for computing the domain a function from its index. In the parlance of computer science, this translates into asking whether there is an algorithm capable of analyzing the structure of an arbitrary program to determine if it enters into an infinite loop on a given input. Consider the *Halting Problem* set $H = \{\langle e, y\rangle \mid \varphi_e(y)\downarrow\}$. Observe that set H is clearly $\Sigma_1^{\mathbb{N}}$ since $\langle e, y\rangle$ is in H if and only if there is a sequence of snapshots s of length n obeying e with y on T_0 and $S_{n-1} = 1$. We can also consider the diagonal of H, which is the set $K = \{e \mid \varphi_e(e)\downarrow\}$. Are the sets H and K computable? First, we observe that K is not an index set and so its computability is not immediately ruled out by Rice's Theorem.

Theorem 8.7. *The set K is not an index set.*

Proof. Recall that the defining property of index sets is that whenever an index e is in it and $\varphi_e = \varphi_{e'}$, then e' must be in it as well. Thus, it suffices to produce a function $\varphi_e = \varphi_{e'}$ such that $\varphi_e(e)\downarrow$ halts, but $\varphi_{e'}(e')\uparrow$. First, observe that there is a computable function $f(x)$ which on input x outputs an index of a computable function with domain $\{x\}$. By the Recursion Theorem, there is an index e such that $\varphi_e = \varphi_{f(e)}$. Thus, the domain of φ_e is the same as the domain of $\varphi_{f(e)}$, which is by definition $\{e\}$, and so $\varphi_e(e)\downarrow$ halts. But for any other index e' such that $\varphi_e = \varphi_{e'}$, we have that $\varphi_{e'}(e')\uparrow$. □

Theorem 8.8. *The sets H and K are not computable.*

Proof. It suffices to show that K is not computable. Suppose towards a contradiction that K is computable. Consider the program P which on input (x, y), first checks whether $x \in K$ holds. If $x \in K$, then P goes into an infinite loop and if $x \notin K$, then P outputs 1. Let $f(x, y)$ be the function computed by P. By the Guessing Theorem, there is an index e such that $\varphi_e(y) = f(e, y)$. Thus, we have that $\varphi_e(e) = f(e, e)$. First, suppose that $\varphi_e(e)\downarrow$. Then, by definition, $f(e, e)\uparrow$. Next, suppose that $\varphi_e(e)\uparrow$. Then, by definition, $f(e, e)\downarrow$. Thus, we have reached a contradiction showing that K cannot be computable. □

In particular, it follows that the sets H and K are not $\Pi_1^{\mathbb{N}}$.

Even though there is no uniform algorithm to determine the domain of a computable function from its index, it is still possible that there are such algorithms for

specific computable functions. We will denote the domain of a computable function φ_e by W_e and if W_e is computable, we shall say that the *Halting problem* for φ_e is *decidable*.

Corollary 8.9. *The Halting problem for the UTM is not decidable.*

Proof. Let $D = \{\langle e, a\rangle \mid \text{UTM halts on } (e, a)\}$. Then we have that $\langle e, x\rangle \in H$ if and only if $\langle e, \langle y\rangle\rangle \in D$. Thus, if D was computable, then H would be computable as well. □

In Section 3, we introduced the μ-operator on functions. For a function $f(x, \overline{y})$, we defined that $\mu x[f(x, \overline{y}) = 0](\overline{y})$ is the least x, if it exists, such that $f(x, \overline{y}) = 0$, and is undefined otherwise. The subtle requirement on the μ-operator is that we consider x to be least only provided that $f(z, \overline{y})$ exists for all $z < x$. It turns out that this requirement is indeed necessary because otherwise computable functions are not closed under the μ-operator. For a counterexample, consider the function

$$f(x, y) = \begin{cases} 0 & \text{if } x \neq 0 \vee (x = 0 \wedge y \in K), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Let us argue that $f(x, y)$ is computable. Suppose we are given an input (x, y). If $x \neq 0$, we output 0. If $x = 0$, we run φ_y on input y, and if it halts, we output 0. Now consider the function $g(y)$ defined to be the least x such that $f(x, y) = 0$, where we allow that $f(z, y)$ is undefined for $z < x$. We shall argue that $g(y)$ is the characteristic function of the complement of K. Suppose that $n \in K$. Then $f(0, n) = 0$, and so $g(n) = 0$. Next, suppose that $n \notin K$. Then $f(0, n)$ is undefined, but $f(1, n) = 0$. So $g(y) = 1$. Thus, the alternative version of the μ-operator takes us outside the realm of computable functions!

8.2. **On the Entscheidungsproblem.** In this section, we use the fact that the domain of the UTM is not computable to show that the Entscheidungsproblem has a negative solution. Let us say that a theory T is *decidable* if the set of Gödel-numbers of its theorems is computable. It will be an easy corollary of our argument for the negative solution to the Entscheidungsproblem that PA is not decidable.

Theorem 8.10. *The set of Gödel-numbers of logical validities in the language L_A is not computable.*

Proof. Let e' be an index of the UTM, meaning that it is computed by the function $\varphi^{(2)}_{e'}$. Given $e, a \in \mathbb{N}$, consider the sentence $\psi_{e,a}$ expressing that there exists a sequence of snapshots s of length n obeying e' with $[a]_0, \ldots, [a]_{\text{len}(a)-1}$ on T_0 and $S_{n-1} = 1$. Let ψ_{PA^-} be the sentence that is the conjunction of the PA^- axioms. We will argue that $\varphi^{(2)}_{e'}(e, a)\downarrow$ if and only if $\psi_{\text{PA}^-} \to \psi_{e,a}$ is a logical validity. First, suppose that $\psi_{\text{PA}^-} \to \psi_{e,a}$ is a logical validity. It follows, in particular, that $\mathbb{N} \models \psi_{\text{PA}^-} \to \psi_{e,a}$, and so $\mathbb{N} \models \psi_{e,a}$. Thus, $\varphi^{(2)}_{e'}(e, a)\downarrow$. Next, suppose that $\varphi^{(2)}_{e'}(e, a)\downarrow$ and let s be a witnessing sequence of snapshots. Since it is Δ_0-expressible that s is such a witnessing sequence, every model $M \models \text{PA}^-$ agrees that s witnesses $\psi_{e,a}$. Thus, $\psi_{\text{PA}^-} \to \psi_{e,a}$ is a logical validity. It follows that if we could compute the set of Gödel-numbers of logical validities, we could compute the domain of the UTM, which is impossible. A crucial fact used in the argument was that there is a true finite theory, namely PA^-, all of whose models have $\mathbb{N}$ as a Δ_0-elementary submodel. □

Theorem 8.11. *Peano Arithmetic is not decidable.*

Proof. Using the notation of the proof of Theorem 8.10, we have that $\varphi^{(2)}_{e'}(e,a)\downarrow$ if and only if $\psi_{e,a}$ is a theorem of PA. □

It follows in particular, that the set Thm_{PA} consisting of the Gödel-numbers of PA-theorems is $\Sigma^{\mathbb{N}}_1$ but not $\Pi^{\mathbb{N}}_1$, and so the relation $\text{Pr}_{\text{PA}}(x)$ is not $\Pi^{\mathbb{N}}_1$.

8.3. **Computably enumerable sets.** In the previous sections, we encountered sets K, H, and Thm_{PA} that are $\Sigma^{\mathbb{N}}_1$ but not $\Pi^{\mathbb{N}}_1$. Such sets and their generalizations are fundamental to the theory of computability. In this section, we will explore some of their general properties.

We define that $A\subseteq\mathbb{N}$ is *computably enumerable* if A is the range of some computable function $f(x)$. Indeed, a set is computably enumerable if and only if there is a Turing machine program which prints out the members of the set on the Turing machine tape (starting at the initial head position) in no particular order. Suppose that A is the range of a function $f(x)$ computed by program P. Naively, we would like to print out members of A by first computing $f(0)$ and printing the result, next computing $f(1)$ and printing the result, etc. But since the function $f(x)$ is potentially partial, such a program could fail to list out A because it might get caught up in an infinite loop in the process of computing one of the $f(n)$. The solution is to have a printing program Q which runs the following loop controlled by the counter n. During each iteration of the loop, the program Q takes the counter value n and decodes it into $n=\langle m,l\rangle$. Then it runs the program P on input m for l many steps. If the computation halts, then the program Q prints out the result and otherwise it prints nothing. The program Q then increments the counter and loops. Now suppose that a program P prints out the set A on the Turing machine tape. Let us define $f(x)$ to be the xth-element printed by P. Then $f(x)$ is clearly a computable function and it is moreover total. Thus, we can assume without loss of generality that a set is computably enumerable if it is the range of a *total* computable function.

Theorem 8.12. *A set is computably enumerable if it is the range of a total computable function.*

In fact, if a computably enumerable set is infinite, then it has a computable enumeration!

Theorem 8.13. *An infinite set is computably enumerable if and only if it is the range of a total one-to-one computable function.*

Proof. Suppose that A is computably enumerable and fix a total computable $f(x)$ such that A is the range of f. Let us inductively define a function $g(x)$ such that $g(n)=f(m)$ where m is least such that $f(m)\neq g(k)$ for all $k<n$. The function $g(x)$ is clearly computable. It is one-to-one by construction and must have the same range as $f(x)$. □

Observe that if a set is computably enumerable, then the longer the Turing machine printing it runs, the more members of the set become known, but we can never rule out that a given element is a member of the set because it might always appear at some later time. Computably enumerable sets have many equivalent characterizations, in particular, they are precisely the Halting problem sets of computable functions, W_e, and precisely the $\Sigma^{\mathbb{N}}_1$-sets.

Theorem 8.14. *The following are equivalent for a set $A \subseteq \mathbb{N}$:*

(1) *A is computably enumerable,*
(2) *A is $\Sigma_1^{\mathbb{N}}$,*
(3) *$A = W_e$ for some $e \in \mathbb{N}$.*

Proof.
$(1) \to (2)$: Suppose that A is computably enumerable and fix a computable function $f(x)$ such that A is the range of f. Let f be defined by the Σ_1-formula $\psi(x, y)$. Then A is defined by the formula

$$\varphi(x) := \exists z \varphi(z, x).$$

$(2) \to (3)$: Suppose that A is defined by the Σ_1-formula $\varphi(x) := \exists z \psi(x, z)$ with $\psi(x, z)$ a Δ_0-formula. Define $f(x) := \mu_z[\psi(x, z)]$ and let e be an index of f. Then $A = W_e$.
$(3) \to (1)$: Suppose that $A = W_e$ for some index e and that e codes the program Q. Consider the program P, which on input n decodes $n = \langle k, l \rangle$ and runs the program Q for l-many steps on input k. If Q halts in l-many steps, the program P outputs k. Clearly the program P computes a function whose range is A. □

Corollary 8.15. *A subsets of the natural numbers is computable if and only if it and its complement are both computably enumerable.*

Proof. We showed earlier that computable sets are precisely the $\Delta_1^{\mathbb{N}}$-sets. Thus, every computable set is computably enumerable. Now suppose that both a set and its complement are computably enumerable. Then both are $\Sigma_1^{\mathbb{N}}$ and hence the set itself is $\Delta_1^{\mathbb{N}}$. □

Thus, the computably enumerable sets that are not computable are precisely the $\Sigma_1^{\mathbb{N}}$-sets that are not $\Pi_1^{\mathbb{N}}$.

8.4. **Oracle Turing Machines and Relative Computability.** There are continuum many subsets of the natural numbers, but only countably many of these are computable. This is because there are countably many Turing machine programs and each such program can compute the characteristic function of at most one set. Similarly there are only countably many computably enumerable sets. More generally, since there are countably many formulas, there are only countably many sets that are definable over the natural numbers. Thus, almost all subsets of $\mathbb{N}$ are not even definable, let alone computable. Although it may not appear to be the case at the outset, the subject of computability theory is concerned mainly with all these other subsets of $\mathbb{N}$. Computability theory studies the information content of sets and the resulting structure that such considerations create on the powerset of the natural numbers. Informally, the information content of a set consists of a description of what can be computed from it. Let us suppose that a computer is given unrestricted access to a magical oracle for some non-computable set A, for instance, the Gödel-numbers of all PA-theorems. In the course of running a program, the computer can query the oracle about whether a given number is an element of A and use the resulting answer in the computation. The collection of all sets computable by the oracle-endowed computer describes the informational content of A. This notion of relative computability was invented by Turing and he modeled it with his oracle Turing machines.

An oracle Turing machine possess all the hardware of a standard Turing machine along with the ability to, on request, count the number of 1's currently on the tape and query the oracle whether that number is in it or not. A program for an oracle Turing machine has a new 4-tuple instruction type $\langle s, r, s_{\text{yes}}, s_{\text{no}}\rangle$, called the *query instruction*. The query instruction dictates that a machine in state s and reading the symbol r should count the number of 1's already on the tape. If that number n is in the oracle set, then the machine should transition to the state s_{yes} and otherwise to the state s_{no}. An oracle Turing machine program is independent of any particular oracle, but will mostly likely produce different results when run on a machine with access to different oracles. Suppose that $A \subseteq \mathbb{N}$. We shall say that an oracle program P using oracle A *computes* a function $f(\overline{x})$ if for every $\overline{m}$ in $\mathbb{N}$, we have $f(\overline{m}) = n$ if and only if P halts on input $\overline{m}$ with output n. We shall say that a function $f(\overline{x})$ is A-(Turing) *computable* if there is an oracle program which computes it using A. We shall say that a set is A-*computable* if its characteristic function is A-computable and we shall say that it is A-*computably enumerable* if it is the range of an A-computable function. Natural number codes of oracle programs can be computed similarly as for standard Turing programs. If e codes an oracle program and $A \subseteq \mathbb{N}$, then we shall denote by $\Psi_e^A(x)$ the A-computable function computed by P. If A is not computable, what are some examples of non-computable functions that are A-computable?

Example 8.16. Suppose that $A \subseteq \mathbb{N}$.

(1) The characteristic function χ_A of A is A-computable.
The program P computing χ_A on input n queries the oracle as to whether $n \in A$. If the answer is yes, it outputs 1 and 0 otherwise.

(2) The characteristic function $\chi_{\overline{A}}$ of the complement $\overline{A}$ of A is A-computable.
The program P computing χ_A on input n queries the oracle as to whether $n \in A$. If the answer is yes, it outputs 0 and 1 otherwise.

Example 8.17. The characteristic function of the Halting Problem set H is Thm_{PA}-computable.

In the previous sections we saw that the class of computable functions was identical to the class of recursive functions and the class of $\Sigma_1^{\mathbb{N}}$-functions. We also showed that the computable sets were precisely the $\Delta_1^{\mathbb{N}}$-sets and the computably enumerable sets were precisely the $\Sigma_1^{\mathbb{N}}$-sets. Such equivalent characterizations exist for the A-computable functions as well. We define the class of A-*recursive* functions as the smallest class containing the basic functions together with the characteristic function of A and closed under the operations of composition, primitive recursion, and the μ-operator. Let L_A^A be the language of arithmetic expanded by a single unary relation A and consider the L_A^A-structure $\langle \mathbb{N}, +, \cdot, <, A, 0, 1\rangle$ where the additional relation A is interpreted by the set A. Let us say that a function $f(\overline{x})$ is $\Sigma_n^{\langle \mathbb{N}, A\rangle}$, $\Pi_n^{\langle \mathbb{N}, A\rangle}$, or $\Delta_n^{\langle \mathbb{N}, A\rangle}$, if it is definable by a Σ_n, Π_n, or a $\Delta_n(\mathbb{N})$-formula over the expanded structure $\langle \mathbb{N}, +, \cdot, <, A, 0, 1\rangle$. It turns out that the class of A-computable functions is precisely the class of A-recursive functions and precisely the class of $\Sigma_1^{\langle \mathbb{N}, A\rangle}$-functions. The A-computable sets are again precisely the $\Delta_1^{\langle \mathbb{N}, A\rangle}$-sets and the A-computably enumerable sets are precisely the $\Sigma_1^{\langle \mathbb{N}, A\rangle}$-sets. More generally, all results we encountered in the previous sections extend seamlessly to relative computability.

8.5. **The Turing Universe.** Suppose that $A, B \subseteq \mathbb{N}$. Let us define that:

(1) $A \leq_T B$ if A is B-computable,
(2) $A \equiv_T B$ if $A \leq_T$ and $B \leq_T A$,
(3) $A <_T B$ if $A \leq_T B$, but $B \not\leq_T A$.

The relation $A \leq_T B$ is known as the *Turing reducibility* relation because we imagine that we are reducing the problem of computing A to the problem of computing B. We interpret the statement $A \equiv_T B$ to mean that A and B have the same informational content and we interpret the statement $A <_T B$ to mean that B has strictly more informational content than A.

Example 8.18. If $A \subseteq \mathbb{N}$ is computable and $B \subseteq \mathbb{N}$, then $A \leq_T B$.

Example 8.19. If $A \subseteq \mathbb{N}$ is $\Sigma_1^{\mathbb{N}}$ or $\Pi_1^{\mathbb{N}}$, then $A \leq_T H$.
Suppose that A is $\Sigma_1^{\mathbb{N}}$. Then $A = W_e$ for some function $\varphi_e(x)$. Thus, $n \in A$ if and only if $\langle e, n\rangle \in H$. Now suppose that A is $\Pi_1^{\mathbb{N}}$, then the complement of A is $\Sigma_1^{\mathbb{N}}$ and so is H-computable. But then A is H computable as well.

Example 8.20. If $A \subseteq \mathbb{N}$ is computable, then $A <_T H$.

Example 8.21. We have that $\text{Thm}_{\text{PA}} \equiv_T H$.
Our previous arguments show that $H \leq_T \text{Thm}_{\text{PA}}$. Also, since Thm_{PA} is computably enumerable, we have that $\text{Thm}_{\text{PA}} \leq_T H$.

Example 8.22. We have that $K \equiv_T H$.
Clearly $K \leq_T H$. So it remains to argue that $H \leq_T K$. Let $f(x)$ be a function which on input $x = \langle e, n\rangle$ outputs an index e' such that if $\varphi_e(n) = a$, then $\varphi_{e'}$ is the constant function a and if $\varphi_e(n)\uparrow$, then $\varphi_{e'}(x)\uparrow$ for all x. Clearly $f(x)$ is total computable. Now to determine whether $\langle e, n\rangle \in H$, we check whether $f(e) \in K$.

Example 8.23. We have that $K_1 \equiv H$.
Since K_1 is computably enumerable, we have $K_1 \leq_T H$. So it remains to argue that $H \leq_T K_1$. Let $f(x)$ be as in the example above. To determine whether $\langle e, n\rangle \in H$, we check whether $f(e) \in K_1$.

Example 8.24. We have that $Z \equiv_T \text{Tot}$.
First, we show that $\text{Tot} \leq_T Z$. Let $f(x)$ be a function which on input e outputs an index e' such that if $\varphi_e(n)\downarrow$, then $\varphi_{e'}(n) = 0$ and if $\varphi_e(n)\uparrow$, then $\varphi_{e'}(n)\uparrow$. Now to determine whether $e \in \text{Tot}$, we check whether $f(e) \in Z$. Next, we show that $Z \leq_T \text{Tot}$. Let $g(x)$ be a function which on input e outputs an index e' such that if $\varphi_e(n) = 0$, then $\varphi_{e'}(n) = 0$, if $\varphi_e(n) \neq 0$, then $\varphi_{e'}(n)\uparrow$, and if $\varphi_e(n)\uparrow$, then $\varphi_{e'}(n)\uparrow$. Now to determine whether $e \in Z$, we check whether $f(e) \in \text{Tot}$.

Example 8.25. We have that $\text{Fin} \equiv_T \text{Tot}$.
First, we show that $\text{Fin} \leq_T \text{Tot}$. Let $f(x)$ be the function which on input e outputs the code e' of the following program P. On input 0, the program P searches for the least $n = \langle k, l\rangle$, where $\varphi_e(k)\downarrow$ in l-many steps. If it finds such n, it outputs k. On input 1, the program P searches for the least $n_0 = \langle k_0, l_0\rangle$, where $\varphi_e(k_0)\downarrow$ in l_0-many steps, and then for the least $n = \langle k, l\rangle > n_0$ such that $k \neq k_0$ and $\varphi_e(k)\downarrow$ in l-many steps. If it finds such n, it outputs k. It should now be clear what P does on an arbitrary input m. It remains to observe that the domain of φ_e is infinite if and only if $\varphi_{f(e)}$ is total. Next, we show that $\text{Tot} \leq_T \text{Fin}$. Let $f(x)$ be the function which on input e outputs the index e' such that $\varphi_{e'}(n) = 1$ if all

$\varphi_e(0), \ldots, \varphi_e(n)$ halt and $\varphi_e(n) \uparrow$ otherwise. It remans to observe that $e \in \text{Tot}$ if and only if $f(e) \notin \text{Fin}$.

Lemma 8.26. *The Turing reducibility relation $\leq_T$ is transitive.*

Proof. Suppose that $A \leq_T B$ and $B \leq_T C$. Then there is a program P which computes the characteristic function χ_A of A using oracle B and a program Q which computes the characteristic function χ_B of B using oracle C. The program P' to compute χ_A using oracle C runs program P until it comes time to query the oracle. At this point instead of querying the oracle about whether $n \in B$, the program P' runs the program Q to determine whether $n \in B$. $\square$

Corollary 8.27. *The relation $\equiv_T$ is an equivalence relation on the subsets of $\mathbb{N}$.*

Proof. The relation $\equiv_T$ is reflexive since we already argued that $A \leq_T A$ for every $A \subseteq \mathbb{N}$. The relation $\equiv_T$ is obviously symmetric, and it is transitive by Lemma 8.26. $\square$

Suppose that $A \subseteq \mathbb{N}$. We shall call the equivalence class $[A]_T$ of A, the *Turing degree* of A and also denote it by $\mathbf{a} = \deg(A)$. Let $\mathcal{D}$ be the collection of all Turing degrees. This is the Turing Universe! Since each degree is countable, the collection $\mathcal{D}$ has size continuum. The collection $\mathcal{D}$ comes with the obvious partial order $\leq$ induced on the equivalence classes by $\leq_T$. The degree $\mathbf{0}$ of all computable sets is the least Turing degree and any other degree has at most countably many other degrees below it. Thus, in particular, there is no greatest Turing degree. Are there incomparable degrees? Yes! A theorem of Post and Kleene says $\mathcal{D}$ is not linearly ordered. In fact, every countable partial order can be embedded in $\mathcal{D}$! There are even incomparable degrees of computably enumerable sets!

Lemma 8.28. *Every two degrees $\{\boldsymbol{a}, \boldsymbol{b}\}$ have a least upper bound degree.*

Proof. Suppose that $A \in \mathbf{a}$ and $B \in \mathbf{b}$. Define the set $A \oplus B$ so that $n \in A$ if and only if $2n \in A \oplus B$ and $n \in B$ if and only if $2n+1 \in A \oplus B$. Clearly A, B are computable from $A \oplus B$, but also if $A \leq_T C$ and $B \leq_T C$, then $A \oplus B \leq_T C$. Thus, $\mathbf{a} \oplus \mathbf{b} = \deg(A \oplus B)$ is the least upper bound for $\{\mathbf{a}, \mathbf{b}\}$. $\square$

Suppose that $A \subseteq \mathbb{N}$. We define the *jump* of A to be $A' = \{\langle n, e\rangle \mid \psi_e^A(n) \downarrow\}$, the Halting Problem set for sets computable from A. Inductively, we define the $n+1^{\text{th}}$-jump of A to be $A^{(n+1)} = (A^{(n)})'$. Identical arguments to those made in the computable case show that A' is A-computably enumerable, every A-computably enumerable set is Turing reducible to A', and $A <_T A'$. Returning back to the Turing Universe $\mathcal{D}$, we note that the jump operation respects equivalence classes.

Lemma 8.29. *Suppose that $A, B \subseteq \mathbb{N}$. If $A \equiv_T B$, then $A' \equiv_T B'$.*

Proof. Suppose that $A \leq_T B$. Then there is a program P which computes the characteristic function χ_A of A using oracle B. We describe the program Q which performs the following action on input $\langle e, n\rangle$. Suppose that e codes the program p for an oracle Turing machine. First, Q computes the index e' coding the program p' which instead of querying the oracle, runs program P and acts on its output. The result is that $\psi_e^A = \psi_{e'}^B$. Now Q computes whether $\langle n, e\rangle \in A'$ by checking whether $\langle n, e'\rangle \in B'$. $\square$

Thus, it makes sense to talk about the jump $\mathbf{a}'$ of a Turing degree $\mathbf{a}$, and more generally about the nth-jumpt $\mathbf{a}^{(n)}$.

Corollary 8.30. *The collection $\mathcal{D}$ has a strictly increasing ω-chain.*

Proof. Consider the increasing chain: $\mathbf{0} < \mathbf{0}' < \mathbf{0}'' < \cdots < \mathbf{0}^{(n)} < \cdots$. □

We end with the connection of the jump hierarchy to the arithmetic hierarchy. We showed in the previous sections that a set $A \subseteq \mathbb{N}$ is computable if and only if it is $\Delta_1^{\mathbb{N}}$ and it is computably enumerable if and only if it is $\Sigma_1^{\mathbb{N}}$. More generally, it holds that:

Theorem 8.31.

(1) *A set $A \subseteq \mathbb{N}$ is computable in $\mathbf{0}^{(n)}$ if and only if it is $\Delta_{n+1}^{\mathbb{N}}$.*
(2) *A set $A \subseteq \mathbb{N}$ is $\Sigma_{n+1}^{\mathbb{N}}$ if and only if it is computably enumerable in $\mathbf{0}^{(n)}$.*

Proof. It will suffice to verify item (1) for $n = 1$. So suppose first that A is computable in $\mathbf{0}'$. Then by our previous observation A is $\Delta_1^{\langle \mathbb{N}, \mathbf{0}' \rangle}$. Thus, there is a Σ_1-formula in the language with the predicate for $\mathbf{0}'$ which defines A over $\mathbb{N}$. But since $\mathbf{0}'$ is itself $\Sigma_1^{\mathbb{N}}$, it follows that we can translate this definition of A into at worst a Σ_2-definition in L_A. Thus, A is $\Sigma_2^{\mathbb{N}}$, and the argument that A is $\Pi_2^{\mathbb{N}}$ is similar. Now suppose that A is $\Delta_2^{\mathbb{N}}$. Let us fix a Σ_2-formula $\delta(x) := \exists y \varphi(x, y)$, where $\varphi(x, y)$ is Π_1, defining A. Since $\varphi(x, y)$ is Π_1, the set it defines is computable from $\mathbf{0}'$, and therefore this definition translates into a Σ_1-definition in the language expanded by a predicate for $\mathbf{0}'$. Using a similar argument for the Π_2-definition of A, we conclude that A is computable from $\mathbf{0}'$. □

Corollary 8.32. *A subset of $\mathbb{N}$ is definable if and only if it is computable in some $\mathbf{0}^{(n)}$.*

8.6. **Scott sets.** In 1962, several years before the concept of a standard system was introduced into models of PA, Dana Scott considered collections of subsets of $\mathbb{N}$ represented in a given model of PA and arrived at the notion of what later became known as a Scott set. A *Scott set* $\mathfrak{X}$ is a non-empty collection of subsets of $\mathbb{N}$ having the following three properties:

(1) $\mathfrak{X}$ is a Boolean algebra.
(2) If $T \in \mathfrak{X}$ codes an infinite binary tree (via the β-function coding), then there is $B \in \mathfrak{X}$ coding an infinite branch through T.
(3) If $A \in \mathfrak{X}$ and $B \leq_T A$, then $B \in \mathfrak{X}$.

Put succinctly, Scott sets are Boolean algebras of subsets of $\mathbb{N}$ that are closed under branches through binary trees and under relative computability.

Once the concept of a standard system was introduced, Scott's work was reinterpreted to show that every standard system is a Scott set and that quite remarkably every countable Scott set is the standard system of a model of PA. Thus, Scott sets turned out to be precisely the standard systems of countable models of PA!

Theorem 8.33. *Suppose that $M \models \mathrm{PA}$ is nonstandard. Then $SSy(M)$ is a Scott set.*

Proof. We already showed in Section 2 that $SSy(M)$ is a Boolean algebra and that whenever it contains a binary tree, it must contain a branch through that tree. Thus, it remains to show that standard systems are closed under relative computability. So suppose that $A \in SSy(M)$ and $B \leq_T A$. Let ψ_e^A compute the

characteristic function of B using A as an oracle. Let $a \in M$ such that $n \in A$ if and only if $(a)_n \neq 0$. Consider the following formula

$$\delta(x, a) := \exists s \text{ sequence of snapshots witnessing that } \psi_e^a(x) = 1.$$

By the notation ψ_e^a, we mean that the query instruction is carried out by checking whether $(a)_n \neq 0$. We claim that the intersection of the subset of M defined by $\delta(x, a)$ with $\mathbb{N}$ is precisely B. First, suppose that $M \models \delta(n, a)$. We know that □

Before we show the converse of Theorem 8.33 for countable Scott sets, we will prove the following lemma. Suppose that L is a first-order language. We shall say that the grammar of L is *computable* if all relations and functions involved in the arithmetization of L are computable.

Lemma 8.34. *Suppose that $\mathfrak{X}$ is a Scott set, L is some first-order language, and T is an L-theory. If the set $\overline{T}$ of Gödel-numbers of sentences of T is in $\mathfrak{X}$ and the grammar of L is computable, then $\mathfrak{X}$ contains a set of Gödel-numbers of some consistent completion of T.*

Proof. Just as in the proof of Theorem 4.20, we argue that there is a $\overline{T}$-computable binary tree whose branches code consistent completions of T. □

Theorem 8.35. *Every countable Scott set is the standard system of a model of* PA.

Proof. Suppose that $\mathfrak{X}$ is a countable Scott set and enumerate $\mathfrak{X} = \{A_n \mid n \in \mathbb{N}\}$. Let L_A' be the first-order language consisting of L_A together with countably many constants $\{a_n \mid n \in \mathbb{N}\}$ and let L be the first-order language consisting of L_A' together with countably many constants $\{c_n^{(i)} \mid n, i \in \mathbb{N}\}$. Let us code L as follows. The logical symbols of L_A will be coded by pairs $\langle 0, i\rangle$, the variables will be coded by pairs $\langle 1, i\rangle$, the constants $\{a_n \mid n \in \mathbb{N}\}$ will be coded by pairs $\langle 2, i\rangle$, and finally, the constants $c_n^{(i)}$ will be coded by pairs $\langle 4+n, i\rangle$. Let S be the theory consisting of PA together with the countably many sentences

$$\bigcup_{n\in\mathbb{N}} \{(a_n)_j \neq 0 \mid j \in A_n\} \cup \{(a_n)_j = 0 \mid j \notin A_n\}.$$

We will use the constants $c_n^{(i)}$ as Henkin constants to build a complete consistent L-theory with the Henkin property extending S whose Henkin model will have precisely $\mathfrak{X}$ as the standard system.

Let us make the convention that whenever T is a theory, we shall call $\overline{T}$ the set of Gödel-numbers of sentences of T. Let T_0 be the theory PA and observe that since PA is computable, we have that $\overline{T}_0 \in \mathfrak{X}$. Let T_0' be the theory T_0 together with sentences $\{(a_0)_j \neq 0 \mid j \in A_0\} \cup \{(a_0)_j = 0 \mid j \notin A_0\}$ and observe that $\overline{T}_0' \in \mathfrak{X}$ since it is A_0-computable. Next, viewing T_0' as a theory in the language $L_A \cup \{a_0\}$, we let T_0'' be T_0' together with all the Henkin sentences, using the constants $\{c_n^{(0)} \mid n \in \mathbb{N}\}$. Clearly $\overline{T}_0''$ is $\overline{T}_0'$-computable and hence in $\mathfrak{X}$. Finally, we let S_0 be some consistent completion of T_0'' in the language $L_A \cup \{a_0\} \cup \{c_n^{(0)} \mid n \in \mathbb{N}\}$ such that $\overline{S}_0 \in \mathfrak{X}$. Following this pattern, we define S_n for $n \in \mathbb{N}$ and let $S = \bigcup_{n\in\mathbb{N}} S_n$. Since S is by construction a consistent complete Henkin theory, we let $M \models$ PA be the Henkin model constructed from S. By construction, we have that $\mathfrak{X} \subseteq SSy(M)$. So it remains to argue that $SSy(M) \subseteq \mathfrak{X}$. Fix $A \in SSy(M)$ and $a \in M$ such that

$(a)_n \neq 0$ if and only if $n \in A$. Since M is a Henkin model, we have that a is some constant $c_n^{(i)}$. But then the set of sentences $\{(a)_n \neq 0 \mid n \in \mathbb{N}\}$ is already decided by the theory S_i. Now, since $\overline{S}_i \in \mathfrak{X}$, it follows that $\{n \mid (a)_n \neq 0\} \in \mathfrak{X}$ as well. This completes the proof that $SSy(M) = \mathfrak{X}$. □

The proof method of Theorem 8.35 cannot be extended in an obvious way to uncountable Scott sets because the theory S obtained after the first countably many stages need not be an element of $\mathfrak{X}$.

Question 8.36 (Scott's Problem)**.** Is every Scott set the standard system of a model of PA?

Using a construction that involves unioning elementary chains of models of PA, Julia Knight and Mark Nadel extended Scott's Theorem to Scotts set of size ω_1, the first uncountable cardinal.

Theorem 8.37 (Knight, Nadel, 1982)**.** *Every Scott set of size ω_1 is the standard system of a model of* PA*. Thus, if the Continuum Hypothesis holds, then every Scott set is the standard system of a model of* PA*.*

Scott's problem remains one of the most important and fascinating open questions in the subject of models of PA.

8.7. **Homework.**

Question 8.1. Write a Turing machine program to compute truncated subtraction.

Question 8.2. Write a Turing machine program to compute the characteristic function of the set of even numbers.

Question 8.3. Show that the sets represented in PA are precisely the computable sets.

Question 8.4. (Generalized Recursion Theorem) Show that if $f(x, \overline{y})$ is total computable, then there exists a total computable function $k(\overline{y})$ such that $\varphi_{f(k(\overline{y}),\overline{y})} = \varphi_{k(\overline{y})}$.

Question 8.5. Let us say that a function $f : \mathbb{N} \to \mathbb{N}$ is *strictly increasing* if $n < m \to f(m) < f(n)$. Show that a set is computable if and only if it is the range of an increasing computable function.

Question 8.6. Show that every infinite computably enumerable set contains an infinite computable subset.

References

[AZ97] Zofia Adamowicz and Paweł Zbierski. *Logic of mathematics*. Pure and Applied Mathematics (New York). John Wiley & Sons Inc., New York, 1997. A modern course of classical logic, A Wiley-Interscience Publication.

[Coo04] S. Barry Cooper. *Computability theory*. Chapman & Hall/CRC, Boca Raton, FL, 2004.

[Kay91] Richard Kaye. *Models of Peano arithmetic*, volume 15 of *Oxford Logic Guides*. The Clarendon Press Oxford University Press, New York, 1991. Oxford Science Publications.

[Mil] Arnold Miller. Introduction to mathematical logic. `http://www.math.wisc.edu/~miller/old/m771-98/logintro.pdf`.