16.31 – Part 7
Let \( I = \bigcup_{n=0}^{\infty} I_n \), and let \( X \) be the set inductively defined by \( \Gamma \). Since in part 6 it is shown that \( I \subseteq X \), we only need to show that \( X \subseteq I \). For this it suffices to show that \( I \) is \( \Gamma \)-closed (since \( X \) is contained in every \( \Gamma \)-closed set).

Let \( x \in \Gamma(I) \). Since \( \Gamma \) is finitely based, this means there is a finite set \( Y \subseteq I \) such that \( x \in \Gamma(Y) \). But since \( Y \) is finite, \( Y \subseteq I_n \) for some \( n \). (Each element of \( x \) must be some \( I_k \), and since \( Y \) is finite, there will be a largest such \( k \).) So \( x \in \Gamma(I_n) = I_{n+1} \subseteq I \). Since \( x \) was an arbitrary member of \( \Gamma(I) \), we have shown that \( I \) is \( \Gamma \)-closed.

16.31 – Part 8
Here is an operator on \( \mathbb{R} \) that is monotone but not finitely based. Let \( \Gamma(X) = X \cup l(X) \), where \( y \in l(X) \) if there is a sequence \( x_i \) such that each \( x_i \in X \), and \( y = \lim x_i \). Such a \( y \) is called a limit point of \( X \).

A more interesting (but not terribly useful) example is the following operator defined on \( \mathbb{N} \): Let \( \Gamma(X) = X \cup \{ 0, 10 \} \cup \alpha(X) \cup \beta(X) \), where
- \( \alpha(X) = \{ x + y : x, y \in X \} \), and
- \( \beta(X) = \{ x + 1 : \forall y[x \rightarrow y \in X] \} \).

Can you figure out what set is inductively defined by \( \Gamma \)?

Notes 4.17
Suppose \( S \) is a c.e. set. This means there is a computable function \( f \) such that \( S = \text{range}(f) \). Because \( f \) is computable, it can be computed by some Turing machine \( M_f \). Let’s build a new machine \( M \) such that
\[ x \in S \iff \exists z \ x = f(z) \iff M(x) \downarrow \]

Here is a high level description of such a machine \( M \) that works:

1. Put \( z = \) first string in \( \Sigma^* \) on the input tape.
2. Simulate \( M_f \) on input \( z \) until it halts (this is like using our universal machine). \( M(z) \) will halt because \( f \) is computable.
3. If \( M \) outputs \( x \) (i.e., \( f(z) = x \)) then accept (because \( x \in S \)).
4. If \( M_f \) does not output \( x \) (and halt), then try the next string \( z \). Return to step 2.

Now \( M(x) \downarrow \) if and only if \( x \in S \). The idea here is that either we find the right \( z \) and accept \( x \) or we go on vainly searching forever and never halt.

On the other hand, suppose there is a machine \( M \) such that \( x \in S \iff M(x) \downarrow \). We must show there is a computable function \( f \) such that \( S = \text{range}(f) \). Assume \( S \neq \emptyset \), and let \( s \in S \), then one such function is
\[ f(<x, t>) = \begin{cases} s & M(x) \text{ does not halt within } t \text{ steps} \\ x & M(x) \text{ does halt within } t \text{ steps} \end{cases} \]

The function \( f \) is computable since we can add a “counter” to our universal Turing machine and stop the simulation once the counter is used up.

Notes 4.23
Suppose \( A \) and \( \mathbb{N} - A \) are both c.e. So there are machines \( M_1 \) and \( M_2 \) such that
- \( x \in A \iff M_1(x) \downarrow \)
- \( x \in \mathbb{N} - A \iff M_2(x) \downarrow \)
Now build a new machine $M$ that simulates both $M_1$ and $M_2$ at once (this will be like a double version of our universal Turing machine). As soon as one of the simulated machines halts (and one of the two must), we will know whether $x \in A$ or $x \in N - A$, so we can halt and either accept or reject as appropriate. Since $M$ always halts and $x \in A \iff M(x)$ accepts, we see that $A$ is computable.

Notes 4.24

These two problems are similar to the one above. Simply simulate two machines at once. For intersection, don’t accept an input until both simulated machines have done so. For union, we can accept as soon as one of the two simulated machines accepts.
Before presenting some more solutions, let me make a general comment. An important prerequisite to solving most of these problems is a clear understanding of what must be shown. For example, to show that \( A \) is reducible to \( B \), generally speaking you must design a reduction and demonstrate that it does the task (or site some theorem that guarantees the existence of such a reduction. If you need to show that \( T \vdash S \), again you must either construct a proof or use a theorem (like completeness) to show that the proof exists. The point is that the existence of a certain kind of object is required for the argument.

**Problems Handout 28c**

Let \( Q = \{ \langle e, w, q \rangle : M_e \text{ on input } w \text{ enters state } q \} \). First we show that \( Q \) is c.e. To do this we can describe informally an algorithm that on input \( x \) will always terminate and answer correctly when \( x \in Q \), but may run forever when \( x \notin Q \). (Implicit use of Church’s Thesis here.) This is easy: just simulate \( M_e \) on input \( w \) and halt in an accepting state if \( M_e \) ever enters state \( q \). Of course, if \( M_e \) does not halt and does not enter state \( q \), this algorithm will also fail to halt.

In this case, this is the best we can do, since it can be shown that \( Q \) is not computable. This will be a proof by contradiction: We can show that if \( Q \) were computable then the halting problem would be computable. In fact, we can build a (computable) many-one reduction \( f : K \leq_m Q \), where \( K = \{ \langle e | M_e(e) \rangle \} \). The reduction works as follows: \( f(e) = \langle e', e, 1 \rangle \), where \( M_{e'} \) is a modification of the machine \( M_e \) such that

- \( M_{e'} \) has only one accepting (halting) state and it is called state 1. (To do this, simply rename states 0 and 1 if it in use, then add instructions so that on any old accepting state, \( M_{e'} \) leaves the tape unmodified and enters to state 1; on any rejecting state, \( M_{e'} \) leaves the tape unmodified and enters state 0; and when in state 0 \( M_{e'} \) leaves the tape unmodified and remains in state 0.)

- otherwise, \( M_{e'} \) is just like \( M_e \).

It is a simple matter to turn take the code \( e \) and transform it into code \( e' \) – in fact it can be done in polynomial time – so \( f \) is computable. Now \( e \in K \iff f(e) = \langle e', e, 1 \rangle \in Q \), so \( K \) is reducible to \( Q \). Since \( K \) is not computable, neither is \( Q \).

**Problems Handout 30a, 30b** Let

\[ L = \{ \langle e, w \rangle : M_e \text{ on input } w \text{ moves left at some step in the computation} \} , \]

and let

\[ S = \{ \langle e, w, k \rangle : M_e \text{ on input } w \text{ never moves more than } k \text{ tape cells away from starting position} \} . \]

Each of these problems can be shown to be c.e. in a manner similar to that used in problem 28. This time, however, there are better algorithms.

For \( L \) notice that if a machine \( M \) remains on a single cell of the tape for more than \( qk \) steps (where \( q \) is the number of states and \( k \) the size of the tape alphabet), then a state/tape combination must be repeated, so the machine is in an infinite loop and if it has not already moved left it never will. So we can tell if \( \langle e, w \rangle \in L \) by simulating \( M_e \) on input \( w \) for \( (|w| + 1)qk \) steps. At that point, if the machine has not gone left, it never will. (Do you understand why the +1 is in this formula? – think about what happens once the machine gets to the end of the inputs.) In fact, since \( |w|, q \) and \( k \) are all bounded by the length of the input, this is a polynomial time algorithm.

For \( S \), we can give an comparable algorithm that does not run in polynomial time (it is worse than exponential): Simulate \( M_e \) on input \( w \) for \( (|w|q + 1)qk \) steps. After this many steps, if the machine has not left the \( k \) cells nearest its starting point it never will, since it will have repeated an exact configuration (tape contents and state) and hence be in an infinite loop.

**Problems Handout 38** To show that \( \text{UNARY-PRIMES} \) is polynomial time computable, we can give a high level description of a 2-tape (one-way tapes) Turing machine that checks if the input is a unary code for a prime number. (The algorithm can be improved – the choices made here are for simplicity of description, not efficiency.)

1. Mark the leftmost cell of the input tape with \#.
2. Copy the input tape to the work tape.
3. Repeatedly do the following:
(a) Erase the last non-blank symbol on the work tape.

(b) Position the heads on the # symbols at the left of each tape.

(c) If the work tape contains only ‘#0’, ACCEPT.

(d) Move back and forth on the work tape while moving left to right across the input tape. The idea here is to match 0's on the work tape with 0's on the input tape. (This will require an extra step on the work tape when we turn around.)

(e) If # is encountered on the input tape at the same time that either # or a blank is encountered on the work tape, then REJECT (since the work tape “evenly divides” the input tape).

This algorithm is roughly quadratic: \( n \) repetitions of the main loop, each one requires moving the head back and forth across the input tape.

**BINARY-PRIMES** is in \( \text{coNP} \) because **BINARY-COMPOSITES** is in \( \text{NP} \): Let the certificate consist of two integer greater than 1. The verifier checks that the product of the two integers is indeed the number being verified. Only composites can be factored, so only composites have certificates. Multiplication can be done in polynomial time, so the verifier is a poly time verifier.

**LPL 18.11**  
[Consult problem statement for some of the notation, I’m using \( h \) in place of \( h^M \).]

We need to show that

\[ M \models S \iff h(S) = 1 \]  \((*)\)

We do this by induction on \( S \).

1. If \( S \) is atomic, then \((*)\) holds by definition.

2. If \( S = \lnot \alpha \) and \((*)\) holds for \( \alpha \), then

\[ M \models S \iff M \not\models \alpha \iff h(\alpha) = 0 \iff h(S) = 1, \]

so \((*)\) hold for \( S \).

3. If \( S = \alpha \land \beta \) and \((*)\) holds for \( \alpha \) and \( \beta \), then

\[ M \models S \iff M \models \alpha \text{ and } M \models \beta \iff h(\alpha) = 1 \text{ and } h(\beta) = 1 \iff h(S) = 1, \]

4. The cases for the other logical connectives are similar. (You could put \( \land \) in for \( \land \) in the argument above.)

5. If \( S = Qx \alpha \) for some quantifier \( Q \), then \((*)\) holds by definition.

**LPL 18.12 – part 1**  
[Consult problem statement for some of the notation.]

We need to show that

\[ M_h \models S \iff h(S) = 1 \]  \((\dagger)\)

We do this by induction on \( wffs \) *without = or quantifiers*. Note that the same sort of recursive definition can be given for these sentences as for \( wffs \) in general by simply eliminating \( = \), quantifiers and variables from the definitions.

1. If \( S \) is atomic, then \( S = R(t_1, \ldots, t_k) \) for some terms \( t_i \). Since there are no variables, each term is a constant. By the definition of \( M_h \) and of satisfaction for models, \( M_h \models S \) if and only if \( \langle t_1^M, \ldots, t_k^M \rangle \in R^M \) if and only if \( h(R(t_1, \ldots, t_k)) = 1 \). So \((\dagger)\) holds for atomic \( wffs \) without variables or \( = \).

2. If \( S = \lnot \alpha \) and \((\dagger)\) holds for \( \alpha \), then

\[ M_h \models S \iff M_h \not\models \alpha \iff h(\alpha) = 0 \iff h(S) = 1, \]

so \((\dagger)\) hold for \( S \).

3. If \( S = \alpha \land \beta \) and \((\dagger)\) holds for \( \alpha \) and \( \beta \), then

\[ M_h \models S \iff M_h \models \alpha \text{ and } M_h \models \beta \iff h(\alpha) = 1 \text{ and } h(\beta) = 1 \iff h(S) = 1, \]
4. The cases for the other logical connectives are similar. (You could put \( \wedge \) in for \( \land \) in the argument above.)

5. The quantifier case does not arise in this instance.

This proof and the one in problem 18.11 look very similar because the two statements are almost identical. Thus the logical connective parts are the same. The atomic and quantifier parts change because of the different definitions involved.