This handout shortens or removes the problem statements and moves the longer answer to (E) to the end in order to highlight the answers in briefest form for reference in the next round of presentations.

(A) Any workable system of logic $F$ gives rise to a proof predicate $R_F(S, \pi)$ meaning $\pi$ is a proof of the theorem $S$ in the system $F$. The system is effective if $R_F$ is decidable. Then the set of theorems of $F$, namely $\{S : (\exists \pi)R_F(S, \pi)\}$, is always c.e. but not necessarily decidable. The system $F$ is sound if every theorem $T$ is true under a natural interpretation of what it says. An important body of theorems are all statements of the form

$$T(M, x, \vec{c}),$$

where $\vec{c}$ is an encoding of a sequence of IDs giving a valid accepting computation of the Turing machine $M$ on input $x$, that are factually true. This so-called Kleene $T$-predicate is also decidable—and in polynomial, indeed linear, time. Using it, we can define the language $E_{TM}$ along lines shown in lecture:

$$E_{TM} = \{\langle M \rangle : (\forall x)(\forall \vec{c})\neg T(M, x, \vec{c})\}.$$

A formal system $F$ is adequate for computation if it proves that all true cases of $T$ are true and that all false cases of $T$ are false, as well as handling logical quantifiers and basic numerical and string operations. Show that there must exist Turing machines $M$ such that $\langle M \rangle \in E_{TM}$ is true but not a theorem of $F$.

Answer:
Abbreviate the statement $(\forall x)(\forall \vec{c})\neg T(M, x, \vec{c})$ as $S_E$. Define

$$A = \{\langle M \rangle : (\exists \pi)R_F(S_E, \pi)\}.$$

That is, the machines in $A$ are the ones that $F$ can prove have empty language. Observe that $A$ is c.e., because it is defined by one existential quantifier on the decidable predicate $R_F$. Thus:

- Since $F$ is presumed sound, every $M$ such that $\langle M \rangle \in A$ really does have $L(M) = \emptyset$. So $A$ is a subset of $E_{TM}$.

- Since $A$ is c.e. and $E_{TM}$ is not c.e., $A$ cannot equal $E_{TM}$.

- Thus $A$ is a proper subset of $E_{TM}$, which means there exist $M$ such that $\langle M \rangle \in E_{TM} \setminus A$.

The upshot for such $M$ is that $L(M) = \emptyset$ is true but $F$ cannot prove it. This gives the cut-down version of Gödel’s Theorem: Every effective, sound, and adequate formal system can formulate true statements that it cannot prove.
(B) With reference to (a) and the Wed. 10/14 lecture (near the end), note that we can formalize “$M$ is total” by the statement

$$S_M = (\forall x)(\exists \vec{c})T(M, x, \vec{c}).$$

Define

$$D_F = \{\langle M, \pi \rangle : R_F(S_M, \pi) \land \langle M, \pi \rangle \notin L(M)\}.$$ 

Prove that assuming $F$ is sound, (a) the language $D_F$ is decidable, but (b) $F$ cannot prove it.

**Answer:**

Suppose we had a decider $Q$ for $D_F$ such that $F$ can prove the statement $S_Q$. Then there would exist a proof $\pi$ such that $R_F(S_Q, \pi)$. Now we can ask, is $\langle Q, \pi \rangle \in D_F$?

- If yes, then that would mean $\langle Q, \pi \rangle \notin L(Q)$, but $L(Q) = D_F$, so we should have $\langle Q, \pi \rangle \in L(Q)$. That’s a contradiction.
- If no, then by $L(Q) = D_F$, we would have $\langle Q, \pi \rangle \notin L(Q)$. But because we do have $R_F(S_Q, \pi)$, that is supposed to trigger $\langle Q, \pi \rangle \in D_F$. Again a contradiction.

The conclusion is that we cannot have a total machine $Q$ such that $L(Q) = D_F$ and $F$ proves that $Q$ is total. Now we design our own total machine $Q_0$ such that $L(Q_0) = D_F:

1. On input $\langle M, \pi \rangle$, check whether $R_F(S_M, \pi)$. If not, reject.
2. If yes, then by the soundness of $F$, $M$ really is total.
3. So we can run $M$ on $\langle M, \pi \rangle$ and it will stop. If $M$ rejects $\langle M, \pi \rangle$ then accept $M, \pi$; else, reject $M, \pi$.

It is clear that $Q_0$ accepts an input of the form $\langle M, \pi \rangle$ if and only if $R_F(S_M, \pi)$ holds but $\langle M, \pi \rangle \notin L(M)$. So $L(Q_0) = D_F$. And by step 2, it is clear that step 3 halts (as well as step 1, of course). So $Q_0$ is total, which proves that $D_F$ is a decidable language (quite unlike $D_{TM}$, you see).

The upshot is that we have a decidable language, but it cannot be accepted by any verifiably total machine. The requirement of verification prevents us from solving a problem that really is decidable!

(C) Consider any and all languages $B$ that we can define in the form

$$x \in B \iff (\forall y)(\exists z)R(x, y, z)$$

where $R$ is a decidable predicate. The standard name for the class of languages $B$ that are definable this way is $\Pi_2^0$. Show that the language $TOT$ is complete for the class $\Pi_2^0$ under mapping reductions. We’ve shown it belongs to $\Pi_2^0$, so what you have to do is take a predicate $R(x, y, z)$ that defines a generic language $B \in \Pi_2^0$ in the above manner, and create a Turing machine $M_{R,x}$ that is total if and only if $(\forall y)(\exists z)R(x, y, z)$ is true for $x$. If time allows, do the same for the index set $INF = \{i : L(M_i) \text{ is infinite}\}$. 
Answer: The Turing machine $M_{R,x}$ takes any input $y$ and does the following:

For $z = 0, 1, 2, 3, \ldots$: if $R(x, y, z)$ then halt (and accept $y$, you can add if you wish).

The function $f$ mapping $x$ to the code of $M_{R,x}$ is easily computable. The analysis of the reduction is:

- $x \in A \implies (\forall y)(\exists z)R(x, y, z) \implies$ for all inputs $y$, $M_{R,x}$ eventually finds a $z$ such that $R(x, y, z)$ holds $\implies$ for all inputs $y$, $M_{R,x}(y)$ eventually halts $\implies M_{R,x}$ is total $\implies f(x) \in TOT$.
- $x \notin A \implies (\exists y)(\forall z)\neg R(x, y, z) \implies$ there exists an input $y$ such that $M_{R,x}(y)$ never finds a $z$ that allows it to halt $\implies M_{R,x}$ is not total $\implies f(x) \notin TOT$.

So $A \leq_m TOT$ via $f$, and this makes $TOT$ complete for $\Pi_2$. The same reduction—with the added point about accepting $y$—shows that $ALL_{TM}$ too is complete.

For $INF$, which is the index set of the class of all infinite languages that are c.e., we have that it is in $\Pi_2$ because:

$$\langle M \rangle \in INF \iff (\forall x_0)(\exists x, \tilde{c})[x > x_0 \land T(M, x, \tilde{c})].$$

Now make a new reduction $g$ by revising $M_{R,x}$ on input $y$ to not only run the above loop on $z$ for $y$, but also run it for each $y' < y$. If every $y$ is good then this machine is still total. But if some $y_0$ fails, then $M_{R,x}(y)$ will trip over that failure for all $y \geq y_0$ and thus won’t accept or even halt on any of them. That makes $L(M_{R,x})$ finite. Thus when $x \notin A$, we get $g(x) \notin INF$. So $A \leq_m INF$, which makes $INF$ become $\Pi_2$-complete as well.

(D) Suppose we have a single-tape DTM $M$ that decides whether a given binary string $x$ is a palindrome. Then $M$ works correctly on strings of the form $x = v10^r1w$ where $r = |v| = |w|$: it will accept them only when $w = v^R$. Call the two cells occupied by the two 1s surrounding the central 0" the two “mileposts.” The TM $M$ can be allowed to change the 1s in these cells but they are still the mileposts. A key fact is:

For any $r$ and $v \in \{0, 1\}^r$, with $M$ fixed, the crossing sequence $(q_1, \ldots, q_k)$ of the accepting computation of $M$ on input $x = v10^r1v^R$ uniquely determines $v$.

Use this to justify two further deductions:

- For some strings $v \in \{0, 1\}^r$, the corresponding $k$ must be proportional to $r$ (as $r$ grows—note that the code of $M$ stays fixed). Otherwise you would have a violation of the Pigeonhole principle, since there are $2^r$ distinct strings $v$ but you wouldn’t have enough crossing sequences to go around.

- It follows that the running time of $M$ must be $\Omega(r^2)$, which is quadratic in $n = |x| = 3r + 2$. 
Answer:

It was OK to take for granted that the crossing sequence \((q_1, \ldots, q_k)\) determines \(v\). It follows that any two strings \(u, v \in \{0, 1\}^r\) must give different crossing sequences, since \(u10^r1u^R\) must be accepted too. Hence there must be \(2^r\) different crossing sequences of length \(k\). (We could say “or less than \(k\),” but actually any shorter one could be uniquely counted as the length-\(k\) sequence obtained by padding out to length \(k\) with meaningless copies of \(q_{\text{rej}}\).) The important calculation to do was to realize that the number of possible sequences of \(k\) states is \(|Q|^k\), where \(|Q|\) is a constant because it depends only on \(M\). This sets up the inequality

\[|Q|^k \geq 2^r,\]

since the number of possible sequences of \(k\) states must be at least as large as the number we know we need. Taking logs of both sides gives:

\[k \log_2 |Q| \geq r, \quad \text{so} \quad k \geq \frac{r}{\log_2 |Q|}.\]

This means \(k = \Omega(r)\). And since this entails \(k\) trips across the middle \(r\) cells, we get that the total time must be \(\Omega(r^2)\), as needed to be shown.

(E) Unlike (D), this option is IMHO represented well by Debray’s notes on pages 32–33 without going to excess. They are supplemented by my own notes called “a programmer’s view of the S-m-n and Recursion Theorems” in the Optional Reading section of the course webpage. Show how to create a total program \(P\) such that \(L(P) = \{\langle P \rangle\}\). Finish with some general remarks about why this doesn’t cause any paradox.

Brief answer:

This was a more open-ended report. My answer is based on the referenced handout. My handout maximizes the distinction between an executable program and its code: substitution is expressly textual and the \(\text{Exec}(u,v)\) function is needed to run program text on a textual argument. Debray’s notes represent the opposite extreme, where exactly what is the “code of \(B(P_B)\)” is (IMHO) unclear though the action is clearly the same and the basic self-reference mechanism is put most simply. This motivated me to make my own handout more precise. The resulting long answer is after that of (F).

(F) Define a finite-state transducer (FST) \(T = (Q, \Sigma, \delta, s, F, \phi)\) to have instructions of the form \((p, c/u, q)\) where \(u\) is a string that is output during the transition. This allows \(u = \epsilon\) whereupon output is paused in that step. In addition, the rule is that if \(T\) does not end in a state in \(F\), then the entire computation is cancelled—so the function \(T(x)\) it computes can be undefined at certain \(x\). But if the final state \(f\) is in \(F\), then there is a final string \(\phi(f)\) that gets appended to the output, again possibly \(\epsilon\) to do no further change.

Now say that a language \(A\) regular reduces to a language \(B\) (written \(A \leq_{\text{reg}} B\)) if there is an FST \(T\) such that for all \(x\),

\[x \in A \iff T(x) \text{ is defined and } T(x) \in B.\]
Show that if \( A \) is not regular, then \( B \) is not regular. Use this to give some examples of non-regularity proofs by reduction rather than by Myhill-Nerode. Show, for instance, that \( \{a^n b^n : n \geq 0\} \not\leq_{reg} \{x : \#a(x) = \#b(x)\} \).

**Answer:**

When given an implication of negatives, it is always good advice to work on the contra-positive. In this case: show that if \( B \) is regular then \( A \) is regular. That says to take a DFA \( M_B = (Q_B, \Sigma, \delta_B, s_B, F_B) \) accepting \( B \) and the FST \( T \) and combine them to make a DFA \( M_A = (Q_A, \Sigma, \delta_A, s_A, F_A) \) accepting \( A \). This suggests the Cartesian Product idea, but it’s not as simple as running \( M_B \) and \( T \) in parallel. We have to make \( T \) feed its output \( y \) to \( B \) while the whole thing reads \( x \). We still have \( Q_A = Q_T \times Q_B \) and \( s_A = (s_T, s_B) \), and the final states are the same as for Cartesian \( \cap \) because we need \( T \) as well as \( B \) to accept. There are two special elements:

- We need to use the \( \delta_B^*(q, u) \) function associated to \( M_B \), which processes whose strings \( u \), including the fact that \( \delta_B^*(q, \epsilon) = q \) for all \( q \in Q_B \).

- To handle the \( \phi \) function, we fudge by letting \( M_A \) run on input \( x\$ \) with an extra end-marker, so that in place of \( \phi(f) = u \) for a final state \( f \) of \( T \), we postulate the instruction \( (f, \$, /u, f) \) in \( \delta_T \).

The code of \( A \) is then quick to write: for every \( p \in Q_T, q \in Q_B, \) and \( c \in \Sigma \cup \{\$\} \), where \( T \) has the instruction \((p, c/u, r)\), \( \delta_A \) has the instruction

\[
((p, q), c, (r, \delta_B^*(q, u))).
\]

This advances both machines to the needed states. [The extra \$ is a fudge, but it can be eliminated as a feature of \( M_B \) by first making it nondeterministic, then converting back to a DFA—the general “quotient” mechanism may be a topic later.]

For the example, if \( A = \{a^n b^n : n \geq 0\} \) and \( B = \{x : \#a(x) = \#b(x)\} \) (note that now we’ve “un-contraposed”: both \( A \) and \( B \) are non-regular), note that \( A = B \cap a^*b^* \). We are hence fine with letting \( T \) use its own accepting states to filter through only those strings of the form \( a^*b^* \) while computing the identity function otherwise, and with \( \phi \) appending nothing. So \( T \) has states \( s, t, \text{dead} \) with \( F_T = \{s, t\} \) and instructions

\[
(s, a/a, s), (s, b/b, t), (t, b/b, t), (t, a/a, \text{dead})
\]

plus the dead-state instructions. Thus \( T(x) \) is defined only when \( x \in a^*b^* \) and for such \( x, x \in A \iff T(x) = x \in B \).

[Mind you, the language \( B \) is not any more difficult than \( A \) to prove nonregular than \( A \) by Myhill-Nerode to begin with. Both involve taking \( S = a^* \) and considering only strings of the form \( a^n b^m \) to begin with. This is true for most examples, including some that are considered annoying to prove via the Pumping Lemma. For an example where \( T \) is not the identity function, one could reduce the homework example \( \{0^n 1^n\} \) to \( B \)—or to \( A \), for that matter.]
Further discussion of (B):

But wait—didn’t we just prove that our $Q_0$ deciding $D_F$ is total? This is a course where you’re supposed to prove your answers, after all. Clearly we are adequate for computation, else we wouldn’t meet the prereqs for the course. Whether we’ve proved stuff must be decidable, else I wouldn’t be able to grade this course. (Hmmm....) We don’t even need to claim by the “Strong AI” rider on the Church-Turing thesis that each of us can be simulated by a formal system—just plug “us” in place of “$F$” in the above and the contradiction still happens. Or most to the point, we did not need any “special sauce” to design $Q_0$—any worthwhile program verification logic should be able to follow CSE491-level reasoning.

The subtle gap is that the totality of $Q_0$ is predicated on the soundness of $F$. If $F$ could prove itself sound, then there would be a full-blown contradiction. What G"odel deduced is that no effective, sound, and adequate formal system can prove its own soundness. Actually, he proved something stronger: $F$ cannot prove its own consistency; that is, it cannot prove the statement $(\forall \pi) \neg R_F(0 = 1, \pi)$. (An inconsistent formal system can prove everything, so not being able to prove the self-contradiction $0 = 1$ entails consistency.) If we apply this to “us” then it is almost a religious principle that no person can prove his or her own soundness.

Further discussion of (D):

To see why the crossing sequence $(q_1, \ldots, q_k)$ determines $v$, consider running $M$ on inputs $u10^r v^R$ for all $u \in \{0, 1\}^r$. Of course, for $u = v$ we get the accepting computation with that crossing sequence. If we got the same sequence for any $u \neq v$, then because the $10^r v^R$ part and whatever happens to it is always the same, we would get an incorrectly accepting computation on input $u10^r v^R$. Since $M$ is correct, $v$ must be the only string in $\{0, 1\}^r$ for which that crossing sequence works.

[A similar argument shows that a streaming algorithm that uses left-to-right passes only and allows $\ell$ local memory to be carried along with random access must make $\Omega(r/\ell)$ such passes. This implies total time $\Omega(r^2/\ell)$. But if we allow duplicating, reversing, and shuffling streams (or if we allow access to two streams at once, with one reversed), then we can do it in just one final pass.]

A very long answer to (E):

In the suggested example, the program $P$ is originally coded to pretend that mycode is a second argument:

$$P = \text{"string x,mycode; cin >> x >> mycode; if (x == mycode) accept; else reject;}"$$

The eventual $P'$ will have just the one argument $x$ and will accept if and only if $x$ equals the code of $P'$, which my handout calls the string $P_{pr}$. Note that in $P$ with $\text{cin >> x >> mycode}$, two arguments come off the stream and are assigned to $x$ and to mycode in that order—it doesn’t mean that $x$ and mycode get the same input value. We write $\text{Subst}(a, \text{’u’}, P)$ with quotes around $u$ because whereas the argument $a$ is read for its string value, $u$ is a literal symbol in $P$. If $u$ is an input of $P$, say when $P$ begins with $\text{cin >> y >> u}$; then the inputting $>> u$ disappears and the new program has just one argument that is input as $\text{cin >> y}$ at the start.
To keep track of what is a string and what is executed, we nest quotes, "..." then ‘...’ and innermost ˆ...ˆ. The directions for defining the string Ppr, italicizing the parts that are assembled into the final string, are:

1. ker = "cin >> y >> u; return Exec(Execq(u,u), y);"

2. eff = "cin >> e; return Subst(e,’mycode’,P);"

3. arg = "cin >> w; return Exec(e,’Subst(w,^u^,ker)’);"

4. Ppr = "Subst(arg,’u’,ker);"

When filling in ker, eff, and arg, we up the internal nesting of quoted strings as needed:

- eff = "cin >> e; return Subst(e,’mycode’,
  ‘string x,mycode; cin >> x >> mycode; if (x == mycode) accept; else reject;’);

- arg = "cin >> w; return Exec(
  ‘cin >> e; return Subst(e,’mycode”,
  “string x,mycode; cin >> x >> mycode; if (x == mycode) accept; else reject;’);
  ‘Subst(w,”u”, “cin >> y >> u; return Exec(Execq(u,u), y);”’);

We dispense with writing angle brackets around the final code string Ppr, which admittedly is really yucky:

Subst("cin >> w; return Exec(
  ‘cin >> e; return Subst(e, 
  "mycode”,
  "string x,mycode; cin >> x >> mycode; if (x == mycode) accept; else reject;”);
  ‘Subst(w,”u”, “cin >> y >> u; return Exec(Execq(u,u), y);”’);

"u",
"cin >> y >> u; return Exec(Execq(u,u), y);”);

Now when we execute Ppr on an argument a, the outer Subst executes first and ultimately returns a program that is then run on a. The action on a is explicit if we do this by calling Exec("Subst(...)",a). The goal is that we want this to simplify at some stage to:

Exec("Subst(Ppr, ’mycode’, P)", a)

This substitutes the string Ppr for occurrences of mycode as a read-from variable inside P, in a way that erases the >> mycode part as well as the declaration of the variable mycode. Note that if mycode is part of a literal string it does not get substituted—this will be important with the quoted names u and y below. The execution is then the same as

Exec("string x; cin >> x; if (x == Ppr) accept; else reject;", a)

which upon inputting a then gives if (a== Ppr) accept; else reject;—and since this is the execution of the program coded by Ppr on a, it accepts only its own code.
How you get there is messy, and the packing and unpacking of strings is a level of analysis I have not seen in any other source. From the top: executing the outer Subst leaves the following string, which we put in square brackets:

```
[cin >> y; return Exec(Exec("cin >> w; return Exec(
  'cin >> e; return Subst(e,
   "mycode",
   "string x,mycode; cin >> x >> mycode; if (x == mycode) accept; else reject;"));',
  'Subst(w,^u^,
   "cin >> y >> u; return Exec(Execq(u,u), y);"));"), y)];
```

Now this is what we execute on a, which gets input as y but only in the top level after the square brackets are taken off. This activates the first return statement, and from now on we have execution steps to trace, in which a is the value a, not a literal:

```
Exec(Execq("cin >> w; return Exec(
  'cin >> e; return Subst(e,
   "mycode",
   "string x,mycode; cin >> x >> mycode; if (x == mycode) accept; else reject;"));',
  'Subst(w,^u^,
   "cin >> y >> u; return Exec(Execq(u,u), y);"));"), y);
```

The next rule is that the inner Exec gets executed before the outer one does, and in fact, the outer execution stays put until we reach the above goal. The next action is that the second double-quoted string gets inputted as w in the first quoted string, which places it as the argument to the second Subst in that string. We get:

```
Exec(Execq(Exec(
  'cin >> e; return Subst(e,
   "mycode",
   "string x,mycode; cin >> x >> mycode; if (x == mycode) accept; else reject;"));',
  [Subst("cin >> w; return Exec(
    'cin >> e; return Subst(e,
     "mycode",
     "string x,mycode; cin >> x >> mycode; if (x == mycode) accept; else reject;"));',
      'Subst(w,^u^,
       "cin >> y >> u; return Exec(Execq(u,u), y);"));"],^u^,
  "cin >> y >> u; return Exec(Execq(u,u), y);"));"), y);
```
Yes, we have three **Exec** calls stacked up, though the third one is executing the program `eff` as described in my older handout. The substituting brought a literal double-quote inside a single quote, so I've changed the outer single quotes in the second argument to that **Exec** into square brackets. The square-bracket string now gets inputted as `e`, and the trickiest thing to note is that the `);’`, after the first `reject;` is consumed in this call, except it becomes the second reject after we input the latter string:

```
Exec(Execq(Subst([Subst("cin >> w; return Exec(
  'cin >> e; return Subst(e,
    "mycode",
    "string x,mycode; cin >> x >> mycode; if (x == mycode) accept; else reject;"");',
    'Subst(w,"u",
    "cin >> y >> u; return Exec(Execq(u,u), y);")");'"),
    "u",
    "cin >> y >> u; return Exec(Execq(u,u), y);")],
    "mycode",
    "string x,mycode; cin >> x >> mycode; if (x == mycode) accept; else reject;" ),
  a);
```

Here I have put a `"u"` on a line by itself to witness that the part in square brackets really is exactly the string `Ppr` which we had before, which is now being substituted for `mycode`. Since it is not going to be activated any more, let’s just call it `Ppr`. So we have:

```
Exec(Execq(Subst(Ppr,
    "mycode",
    "string x,mycode; cin >> x >> mycode; if (x == mycode) accept; else reject;" ),
  a);
```

Finally executing the `Execq`, which effects the substitution and also re-quotes the resulting program string for the outermost `Exec`, achieves the goal:

```
Exec("string x; cin >> x; if (x == Ppr) accept; else reject;",a);
```

This is where `Ppr` is the code of the program as originally executed.