CSE 431
Introduction to Theory of Computation
Homework 4
Due: Friday, April 30, 2010

W. L. Ruzzo 24 April 2010

1. Review the proof (Theorem 5.13) that $A_{TM} \leq_T ALL_{CFG}$. It does not show $A_{TM} \leq_m ALL_{CFG}$. Explain why not. It does, however, prove $A_{TM} \leq_m C$ for some language C related to ALL_{CFG}. What is C? Justify.

2. A language B is called *complete* if (a) it is Turing recognizable and (b) every Turing recognizable language A is mapping reducible to it, i.e., $A \leq_m B$. (It's not obvious that complete languages exist but you'll show below that they do. In fact, they're widespread: every Turing recognizable language we've seen is either decidable or complete, and it took 20 years for someone to prove that there are recognizable languages that are neither. One reason for interest in complete languages is that, in some sense, each embodies the "essence" of Turing recognizability, despite great superficial differences among them. For example, on the face of it, Hilbert's 10th problem, PCP and the language in part (c) have nothing to do with Turing machines, yet they are complete, too.)

   (a) Let M be a arbitrary TM. Prove that $L(M) \leq_m A_{TM}$, hence A_{TM} is complete.

   (b) Example 5.24 (and lecture notes) show that $A_{TM} \leq_m HALT_{TM}$. Use this to show that $HALT_{TM}$ is also complete. (Hint: transitivity.)

   (c) Prove that $\overline{EQ_{CFG}}$ is complete.

   An analog of "completeness" can be defined in other language classes as well:

   (d) Prove that there is a co-recognizable language to which all other co-recognizable languages are mapping reducible.

   (e) Prove that there is a decidable language D to which all decidable languages are mapping reducible.

   (f) Give two decidable languages that could *not* be D above.

3. 5.24. Additionally, show that $J \leq_T A_{TM}$, but not $J \leq_m A_{TM}$.

4. I introduced some decidability questions about "programs" in the first few slides of lecture 12. For the following, you may assume that A_{Prog} and $HALT_{Prog}$ are undecidable.

   "Useless code elimination": Many optimizing compilers identify (and remove) portions of programs that cannot be reached on any input. E.g., in

```
if (True) {
  x = 0
} else {
  y = 42
}
```

the "y" assignment is useless. Prove that the following problem is undecidable: given a program P and a location l in that program (say, a line number in the program), is statement l useless?

5. "Bounds checking": Similarly, some programming languages insist that the value of every array subscript be checked at run time to be sure it is within the declared array bounds. Obviously, programs would run faster if these bounds checks could be eliminated by compile-time verification that the subscript is safe. E.g., in:

```
sub P(x) {
   a: array[100];
   i = 42;
   a[i] = x;
   a[x] = i;
   return a[2];
}
```

the first assignment to "a" will never cause an array-ref-out-of-bounds problem, but the second one might. Prove that the following problem is undecidable: given a program P, no input to P will ever cause P to reference an out-of-bounds array element. Is the problem decidable if the input is specified? I.e., "Given P, w, will P when run on w attempt to access an out-of-bounds array element?" Would the answer change if a particular array is named, and/or a particular array reference is specified? ("Given P, w, a, l, is the reference to $a[\,]$ on line l when P runs on input w safe?")