1. This problem is recognizable but undecidable.

**Proof of recognizability**: write a machine that, when given input , simulates on input and reads the state of for every step, seeing what the value of is. If in any of the steps, then return true.

**Proof of undecidability**: Let’s design a program that takes input :

M'(x){

M\_m = M, but there's a a new line of code in the very

beginning that initiates y and sets it to 0,

everything inside main is copied to another function f,

rewrite main to only have two things:

call f and then set y to 1

M\_m(x) //run M\_m on x  
}

Observe see that if halts, then modifies in the end and halts; and if doesn’t halt, then also doesn’t halt, and never gets to modify .

Suppose for now that there exists a program to rewrite into according to the above pseudocode specification.

Assume there exists a program that decides the Modifies a Variable problem.

Then there exists a program that decides the halting problem:

boolean haltChecker(string M, string x){

M' = rewrite(M);

return modifiesY(M',x);

}

This would be a correct program that decides the Halting Problem if actually could decide the Modifies a Variable Problem. This contradicts the undecidability of the Halting Problem. Thus, our assumption that exists is wrong.

1. This problem is recognizable but undecidable.

**Proof of recognizability**: write a machine that, when given input , simulates on input . If it halts, then return true.

**Proof of undecidability**: Let’s design a program that takes another program as input:

rewrite(M){

M' = M, but rewritten so that

all recursion is turned into iteration;

turn all non-boolean variables into boolean array

representations //we can do this with some rules,

//like turning integers into their binary representations

//then turning 1's into trues and 0's into falses and

//prepending [T,T,F] onto all representations of integers

return M'  
}

Suppose for now that there exists a program to rewrite to rewrite as specified by the above pseudocode specification.  
Assume there exists a program that decides the Limited Memory Halting Problem.  
Then there exists a program that decides the Halting Problem:  
boolean haltChecker(string M, string x){

M' = rewrite(M);

return limMemHalt(M',x);

}

Observe that if halts, then inside also halts, because they are the same program, just in different representations; and if doesn’t halt, then also doesn’t halt for the same reason.

This would be a correct program that decides the Halting Problem if actually could decide the Limited Memory Halting Problem. This contradicts the undecidability of the Halting Problem. Thus, our assumption that exists is wrong.

1. This problem is unrecognizable.

**Proof of unrecognizability**: Let’s design two programs, and , both of which take an input :

M1(x){

M(x);

accept;

}

M2(x){

M(x);

reject;

}

Observe that if doesn’t halt, then neither nor halts. If halts, then both and halt, but with different results.

Suppose for now there exists a program to rewrite into and according to the pseudocode specification above.  
Assume there exists an that decides the Program Agreement Problem.

Then there exists a neverHaltChecker:

boolean neverHaltChecker(M,x){

M1 = rewrite1(M,x);

M2 = rewrite2(M,x);

return agreeChecker(M1,M2,x);

}

This would be a correct program that decides the Co-Halting Problem if actually could decide the Program Agreement Problem. This contradicts the unrecognizability of the Halting Problem. Thus, our assumption that exists is wrong.