For the first 20 points, un-nested MOO loops only have 10^5 iterations, and there can be at most 50 such loops (each one takes up at least 2 lines), so straightforward execution line-by-line, iteration-by-iteration is fast enough. Parsing is a pain, even though COWBASIC was designed to be friendly to parse.

For the next 20 points, we need something faster; nested MOO loops can add up to an unwieldy number of iterations. What does each line of code actually do? If you think about it, each expressions amounts to "x = ax+b", where "a" is the number of times x appears in the expression, and "b" is the sum of the constants in the expression. How about two lines of code? $x = ax + b$ followed by $x = cx + d$ just does $x = c(ax + b) + d = acx + (cb + d)$; again, it multiples $x$ by some constant and adds some constant. By combining two lines at a time, we can turn any block of code into a linear equation like this.

What about loops? How do you run "x = ax+b" n times? One way is just to expand it out. For $n = 5$, for instance, you get $x = a(a(a(a(ax + b) + b) + b) + b) + b = a^5x + b * (1 + a + a^2 + a^3 + a^4)$. The general pattern $a^nx + b * (1 + a + \ldots + a^{n-1})$ is not hard to guess. The geometric sum $1 + a + \ldots + a^{n-1}$ can be evaluated as $\frac{a^n-1}{a-1}$. "Division" by $a - 1$ mod $M = 10^9 + 7$ means multiplying by $(a - 1)^{-1}$, the unique number such that $(a - 1)(a - 1)^{-1} = 1 \bmod M$. This inverse can be computed using exponentiation by squaring as $(a - 1)^{M-2} \bmod M$.

Another, easier way is to use matrices. Matrices are a good way to represent most linear equations, especially when we want to simulate a process with "steps" like this one. The way I think about this is: we're keeping track of a state $V = [[x][1]]$ (a column vector), and we have a rule for updating to the next state, which we can represent as a matrix $M$ such that the next state is $MV$. In this case $M = [[a \quad b][0 \quad 1]]$, and $MV = [[a * x + b * 1][0 * x + 1 * 1]] = [[ax + b][1]]$. Since $M$ represents one step, $M^n$ represents $n$ steps. For example, $M^2 = [[a \quad b][0 \quad 1]]^2 = [[a * a + b * 0 \quad a * b + b * 1][0 \quad 1]] = [[a^2 \quad b + ab][0 \quad 1]]$ represents a loop with two iterations.

In general, we can compute $M^n$ quickly even for large $n$ by using exponentiation by squaring:

```
mat mat_pow(const mat& A, ll e) {
  if(e==1) {
    return A;
  } else if(e%2==0) {
    return mat_pow(mat_mul(A,A), e/2);
  } else {
    return mat_mul(A, mat_pow(A, e-1));
  }
}
```

So that's how we can run loops: represent each block of code as a $2x2$ matrix, and to run the loop, raise that matrix to the number of loop iterations. We can also represent combining lines of code as a matrix operation: we said $x = ax + b$ followed by $x = cx + d$ corresponds to $x = acx + (cb + d)$. This is the same as multiplying the corresponding matrices (which makes sense; multiplying matrices is the same as composing the linear functions they represent, which is the same as "running" the first one and then the second one, just like the COWBASIC program does): $[[c \quad d][0 \quad 1]][[a \quad b][0 \quad 1]] = [[c * a + d * 0 \quad c * b + d * 1][0 * a + 1 * 0 \quad 0 * 0 + 1 * 1]] = [[a * c \quad bc + d][0 \quad 1]]$.

The matrix approaches generalizes to multi-variable programs. Each line of code sets one variable to a linear combinations of variables (and a constant always-1 "variable"), which can be represented by an identity matrix with one row changed. For example, $a = b + c$ can be represented by $[[0 \quad 1 \quad 1 \quad 0][0 \quad 1 \quad 0 \quad 0][0 \quad 0 \quad 1 \quad 0][0 \quad 0 \quad 0 \quad 1]]$ (assuming $a, b, c$ are the only variables); the first row is doing the operation, and the rest is keeping "b", "c", and "1" where they were. Combining two lines is the same as multiplying their matrices. And running a loop i.e. running a block of code $n$ times, is the same as raising the matrix corresponding to that block of code to the $n$th power.

So how do we actually get the final answer? We know how to reduce the whole program to a matrix. If we multiply the matrix by the starting state of the variables, we'll get the final state of the variables. We need to make sure to start the constant variable as 1, but the starting values of the rest of variables actually doesn't matter; that's what it means to say the program won't use any variable before it's defined. So to get the answer, multiply the program matrix by any starting state vector you like, and print the component of that vector corresponding to the variable that was RETURNed.

How's our runtime? We have 100 lines, so at most 50 MOO loops. Evaluating each MOO loop might take $log_2(10^5) \sim 17$ 100x100 matrix multiplications, so that's $17 * 50 * 100^3 \sim 850,000,000$ total multiplications. This is on the edge of

feasible. But we've overestimated the bound; in reality, the program has to split between adding variables (increasing the size of the matrix) and doing loops (requiring exponentiation). So we're probably OK. A tighter bound taking this into account is $88,000,000$ total multiplications, which is definitely fine.

My complete code:

```
#include <iostream>
#include <vector>
#include <map>
#include <string>
#include <cassert>
#include <deque>
#include <memory>

using namespace std;

typedef long long ll;
typedef map<string, ll> Env;
typedef vector<ll> row;
typedef vector<row> mat;

constexpr ll p10(ll n) { return n==0 ? 1LL : p10(n-1)*10LL; }
constexpr ll MOD = p10(9) + 7;
ll ADD(ll x, ll y) {
  return (x+y)%MOD;
}
ll MUL(ll x, ll y) {
  return (x*y)%MOD;
}

mat I(ll n) {
  mat M(n, row(n, 0));
  for(ll i=0; i<n; i++) {
    M[i][i] = 1;
  }
  return M;
}
row row_add(const row& A, const row& B) {
  assert(A.size() == B.size());
  row C(A.size(), 0);
  for(size_t i=0; i<A.size(); i++) {
    C[i] = A[i]+B[i];
  }
  return C;
}
mat mat_mul(const mat& A, const mat& B) {
  mat C(A.size(), row(B[0].size(), 0));
  for(ll i=0; i<A.size(); i++) {
    for(size_t k=0; k<B.size(); k++) {
      for(size_t j=0; j<B[k].size(); j++) {
        C[i][j] = ADD(C[i][j], MUL(A[i][k], B[k][j]));
      }
    }
  }
  return C;
}
mat mat_pow(const mat& A, ll e) {
  if(e==1) {
    return A;
  } else if(e%2==0) {
    return mat_pow(mat_mul(A,A), e/2);
  } else {
    return mat_mul(A, mat_pow(A, e-1));
  }
}
ostream& operator<<(ostream& o, const row& R) {
  o << "[";
  for(size_t i=0; i<R.size(); i++) {
    o << R[i];
```

```
    if(i+1<R.size()) {
       o << " ";
    }
  }
  o << "]";
  return o;
}
ostream& operator<<(ostream& o, const mat& M) {
  for(auto& R : M) {
    o << R << endl;
  }
  return o;
}

void read(istream& in, string lit) {
  string s;
  in >> s;
  assert(s == lit);
}

ll lit_of_string(string s) {
  ll n = stoll(s);
  assert(1 <= n && n<=p10(5));
  return n;
}
string var_of_string(string s) {
  assert(1 <= s.size() && s.size() <= 10);
  for(ll i=0; i<s.size(); i++) {
    assert('a'<=s[i] && s[i]<='z');
  }
  return s;
}

struct Expr {
  virtual ll eval(const Env&) = 0;
  virtual row to_row(const Env&) = 0;
  virtual string str() = 0;
};
struct ExprVar : public Expr {
  string s;
  ExprVar(string s_) {
    s = s_;
  }
  virtual ll eval(const Env& E) override {
    return E.at(s);
  }
  virtual row to_row(const Env& E) override {
    assert(E.count(s) == 1);
    vector<ll> R(E.size()+1, 0);
    R[1+E.at(s)]++;
    return R;
  }
  virtual string str() override {
    return s;
  }
};
struct ExprLit : public Expr {
  ll n;
  ExprLit(ll n_) {
    n = n_;
  }
  virtual ll eval(const Env&) override {
    return n;
  }
  virtual row to_row(const Env& E) override {
    row R(E.size()+1, 0);
    R[0] = n;
    return R;
  }
  virtual string str() override {
```

```
    return to_string(n);
  }
};
struct ExprPlus : public Expr {
  Expr* left;
  Expr* right;
  ExprPlus(Expr* left_, Expr* right_) : left(std::move(left_)), right(std::move(right_)) {}
  virtual ll eval(const Env& E) override {
    return ADD(left->eval(E), right->eval(E));
  }
  virtual row to_row(const Env& E) override {
    return row_add(left->to_row(E), right->to_row(E));
  }
  virtual string str() override {
    return "("+left->str()+") + (" + right->str() + ")";
  }
};
Expr* parse_expr(istream& in) {
  string s;
  cin >> s;
  if(s == "(") {
    auto left = parse_expr(in);
    read(in, ")");
    read(in, "+");
    read(in, "(");
    auto right = parse_expr(in);
    read(in, ")");
    return new ExprPlus(std::move(left), std::move(right));
  } else if(isdigit(s[0])) {
    return new ExprLit(lit_of_string(s));
  } else {
    return new ExprVar(var_of_string(s));
  }
}

// Either Block(n, [code]) or Assign(var, e)
struct Code {
  ll typ;
  ll n;
  vector<Code> block;

  string var;
  Expr* e;

  static Code loop(ll n_, vector<Code> block_) {
    Code c;
    c.typ = 0;
    c.n = n_;
    c.block = std::move(block_);
    return c;
  }
  static Code assign(string v, Expr* e_) {
    Code c;
    c.typ = 1;
    c.var = v;
    c.e = std::move(e_);
    return c;
  }
  void run_(Env& V) {
    if(typ==1) {
      V[var] = e->eval(V);
    } else {
      assert(typ == 0);
      for(ll t=0; t<n; t++) {
        for(auto& c1 : block) {
          c1.run_(V);
        }
      }
    }
  }
```

```
  ll run(string v) {
    Env V;
    run_(V);
    return V[v];
  }
  mat to_mat(const Env& V) {
    if(typ == 0) {
      mat M = I(V.size()+1);
      for(auto& c1 : block) {
        M = mat_mul(c1.to_mat(V), M);
      }
      return mat_pow(M, n);
    } else {
      assert(typ==1);
      assert(V.count(var) == 1);
      mat M = I(V.size()+1);
      M[1+V.at(var)] = e->to_row(V);
      return M;
    }
  }

  ll run_fast(string v, const Env& V) {
    assert(V.count(v) == 1);
    mat M = to_mat(V);
    mat X(V.size()+1, row(1, 0));
    X[0][0] = 1;
    mat X2 = mat_mul(M, X);
    return X2[1+V.at(v)][0];
  }
};

bool bool_of_str(string s) {
  if(s=="True") {
    return true;
  } else if(s == "False") {
    return false;
  } else {
    assert(false);
  }
}

int main(int argc, char** argv) {
  bool no_nest = false; //bool_of_str(argv[1]);
  bool one_var = false; //bool_of_str(argv[2]);
  deque<ll> MOO;
  deque<vector<Code>> CODE;
  MOO.push_back(1);
  CODE.push_back({});
  map<string, ll> VARS;

  while(true) {
    string s;
    cin >> s;
    if(s == "RETURN") {
      assert(MOO.size() == 1 && MOO.front()==1);
      assert(CODE.size()==1);
      map<string, ll> V;
      Code p = Code::loop(1, std::move(CODE.front()));
      CODE.pop_front();
      string v;
      cin >> v;
      v = var_of_string(v);
      assert(VARS.count(v) == 1);
      if(one_var) {
        assert(VARS.size() == 1);
      } else {
      }
      if(no_nest) {
        assert(p.run(v) == p.run_fast(v, VARS));
      }
```

```
      cout << p.run_fast(v, VARS) << endl;
      break;
    } else if(isdigit(s[0])) {
      ll times = lit_of_string(s);
      string moo;
      cin >> moo;
      assert(moo == "MOO");
      string brace;
      cin >> brace;
      assert(brace == "{");
      MOO.push_back(times);
      if(no_nest) {
        assert(MOO.size() <= 2);
      }
      CODE.push_back({});
    } else if(s=="}") {
      assert(MOO.size() > 0);
      assert(CODE.size() > 0);
      ll n = MOO.back(); MOO.pop_back();
      vector<Code> block = std::move(CODE.back()); CODE.pop_back();
      assert(CODE.size() > 0);
      CODE[CODE.size()-1].push_back(Code::loop(n, std::move(block)));
    } else {
      s = var_of_string(s);
      if(VARS.count(s) == 0) {
        VARS[s] = VARS.size();
      }
      string eq;
      cin >> eq;
      CODE[CODE.size()-1].push_back(Code::assign(s, parse_expr(cin)));
    }
  }
}
```