Basic format for describing reductions between programs

In order to describe a reduction from K or $\overline{K}$ into another set as the following:

$$x \mapsto p = \left[\begin{array}{l} \texttt{input } y \\ \quad \texttt{run } M_x(x) \\ \quad \texttt{accept} \end{array}\right.$$

it suffices to describe the resulting program from the reduction in a concrete programming language:

```
input y {
  runmxx;
  accept;
}
```

This language is very limited, and its syntax is described by the following grammar:

```
program: 'input' IDENTIFIER instruction_list

instruction_list: '{' instruction* '}'

instruction: instruction_list
           | IDENTIFIER '=' expr ';'
           | 'if' '(' expr ')' instruction ('else' instruction | )
           | 'accept' ';'
           | 'reject' ';'
           | 'output' expr ';'
           | 'runmxx' ';'
           | 'infiniteloop' ';'

expr: comparison (('and'|'or') comparison)*

comparison: addition (('=='|'!='|'<'|'>'|'<='|'>=') addition)*

addition: product (('+'|'-') product)*

product: unary (('*'|'/'|'%') unary)*

unary: ('not'|'-') unary
     | IDENTIFIER
     | NUMBER
     | 'mxxstopsininputsteps'
     | '(' expr ')'
```

where **IDENTIFIER** is a string over alphanumeric characters and underscore (not starting by a digit), and **NUMBER** is a natural number (an integer greater than or equal to 0).

The meaning of the constructions is the usual one. As you can see, there is no need to declare variables. We can directly assign the evaluation of an expression to them. The unique data type is integer. The input is assumed to be a natural number. The output is also assumed to be a natural number. In the case where a negative number is produced as output, it is considered that the program is rejecting the input. The instruction **output** stops the execution and produces the evaluation of the corresponding expression as result. As we said, if such an evaluation is negative, it is assumed that the input is rejected, and that the output is -1. This is equivalent to execute the instruction **reject**. The instruction

accept stops the execution and accepts. It is equivalent to execute **output** with the natural number 1. As you can see, our programming language does not have loops, but you can execute the instruction **infiniteloop**, which is a non-halting instruction. The instruction **runmxx** executes $M_x(x)$, where x is the element over which the reduction is applied. The instruction **runmxx** halts if $x \in K$, and does not halt in the opposite case. If a variable called `x` occurs in the program, then it does not affect the behaviour of **runmxx**. In other words, changes on the value of the variable `x` do not change the behaviour of **runmxx**, since it only depends on the value of the x of the domain of the reduction. The expression **mxxstopsininputsteps** runs $M_x(x)$ during as many steps as the value received as input of the program. It evaluates to true if $M_x(x)$ halts in such a number of steps or less, and evaluates to false in the opposite case. Changing the value of the input variable does not affect the behaviour of **mxxstopsininputsteps**. That is, such instruction refers always to the initial received value as input, and not to the possible changes produced on the input variable along the execution.

Let's look an example. In order to prove that $\{p \mid \exists y : \ M_p(y) \downarrow\}$ is undecidable, we can make the reduction from K shown above. We also saw how to represent such a reduction. Alternatively, we can do the following reduction for the same language:

$$x \mapsto p = \left[\begin{array}{l} \texttt{input } y \\ \quad \texttt{if } M_x(x) \texttt{ stops in } y \texttt{ steps then accept} \\ \quad \texttt{run forever} \end{array}\right.$$

Such an alternative reduction can be represented as follows:

```
input y {
  if (mxxstopsininputsteps)
    accept;
  infiniteloop;
}
```