Specification

Syntax of abstract machine code is as follows:

(see p.64)
instr::=PUSH-n | ADD | SUB | MULT | TRUE | FALSE | EQ | LE | AND | NEG | FETCH-x | STORE-x | NOOP | LOOP(c, c) | BRANCH(c, c)
c::= epsilon | instr:c

where:
instr – is an atomic instruction
c – is a sequence of instructions, i.e. (instr)*

epsilon – is an empty sequence

Instruction can have:

- value arguments: PUSH-n (number), FETCH-x (variable, cannot be number, so FETCH-1 is not valid)

- instruction arguments: LOOP, BRANCH – they contain the sequence of instructions to be executed

- no argument (most of them).

Program should accept also synonyms: MUL for MULT, NOT for NEG and EMPTYOP for NOOP.

Particular instrucitons represent elementary steps of execution. Translation functions from model language into abstract machine code is on p. 70-71. For example:

a) statement x:=x+1 is translated as PUSH-1:FETCH-x:ADD:STORE-x

- observation: arguments are taken from end, and you construct a stack

b) statement while not(x=1) do x:=x-1 is translated as:

LOOP(PUSH-1:FETCH-x:EQ:NEG, PUSH-1:FETCH-x:SUB:STORE-x)

c) the statement (Euclid algorithm)

while not(a=b) do if (a<=b) then b:=b-a else a:=a-b is translated as

LOOP(FETCH-b:FETCH-a:EQ:NEQ, FETCH-b:FETCH-a:LE:BRANCH(FETCH-a:FETCH-b:SUB:STORE-b, FETCH-b:FETCH-a:SUB:STORE-a))

So we have nested commands.

The goal is to prepare simple console application (no need to use GUI, just, if you want), which takes input source in abstract machine code (can be read from file as an argument in console) and makes source-to-source compilation to model languae.

Our model language contains only 5 statements:

S::=x:=e | skip | S;S | if b then S else S | while b do S

where e stands for any arithmetic expression, b for Boolean expression and S for statement.

Be careful, you cannot simply use split function, although the delimiter is a colon symbol (as you can see above); in any recursive call you need to consider the whole string BRANCH(..., ...) and LOOP(..., ...) as separate instructions, and then parse them recursively.

You can use for parsing the following symbolic grammar:

Prog = (Intsr)*

Instr = Atom | LOOP(Prog, Prog) | BRANCH(Prog, Prog)

Atom = PUSH-n | ADD | SUB | MULT | TRUE | FALSE | EQ | LE | AND | NEG | FETCH-x | STORE-x | NOOP

And prepare the functions:

parseProg()

parseInstr()

parseAtom()

for each rule (as usual).

One more observation: the first sequence in LOOP is an expression, so it cannot contain NOOP, STORE, LOOP nor BRANCH, but the second one, even both ones in BRANCH can!