Submission command

/accounts/classes/janikowc/submitProject/submit\_cs4280\_P4 *SubmitFileOrDirectory*

Invocation

comp [file]

where file is as before with implicit extension. **Wrong invocations will not be graded.**

Details

The program is to parse the input, generate a parse tree, perform static semantics, and then generate a target file. Any error should display detailed message, including line number if available (depending on scanner).

The target name should be

* **kb.asm** if keyboard input
* ***file*.asm** if ***file.*sp2020** input. The base name matches that of the input file, that is if input basename is **test2** then output should be **test2.asm**

The program has 2 parts to be properly generated:

1. Code generation 100. with approx. rubric
   * conditional 10
   * conditional nested 10
   * loop 10
   * loop nested 10
   * in/out 20
   * expressions 20
   * sequence of statements 10
   * others 10
2. Storage allocation global option 30
   * Global - all variables allocated globally. Note that the virtual machine uses all storage directives as global.

Also include README.txt  listing test files that pass successfully and test files that dont, along with a summary of the problem.

Temporary variables should be allocated as global variables as well.

# **P4 Runtime Semantics**

## **Runtime Semantics**

* Basic semantics as in C - program executes sequentially from the beginning to the end, one statement at a time
* Conditional statement is like the else-less if statement in C
* Loop statement is like the while loop in C
* Assignment evaluates the expression on the right and assigns to the ID on the left
* +-\* are standard arithmetical, / is integer division, unary \* is negation
* All expressions are evaluated before being used
* Relational and arithmetical operators have the standard meaning except:
  + < and > are less than and greater than
  + < < is less equal
  + > >  is greater equal
  + < > is NOT equal
  + == is equal
* IO reads/prints a 2-byte signed integer
* All data is 2-byte signed integer

## **Data**

* All data is 2-byte signed integers
* Assume no overflow in operations

## **Target Language**

* VM ACCumulator assembly language as given

**Upon success, only display the name of the target file generated and no other output. Upon error and the error message, no other display should be generated and no target should be generated.**

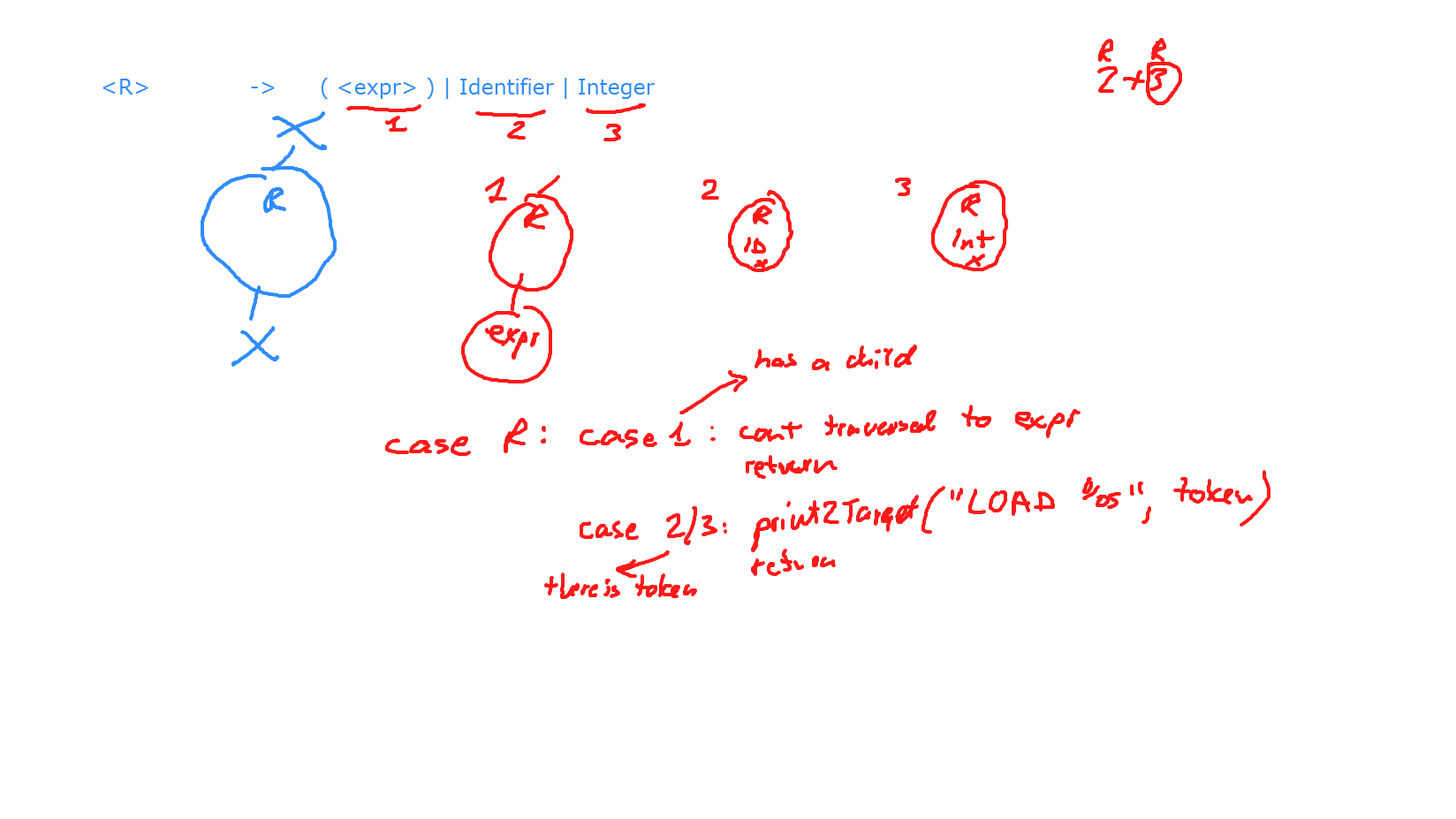
# **P4 Code Generation Support**

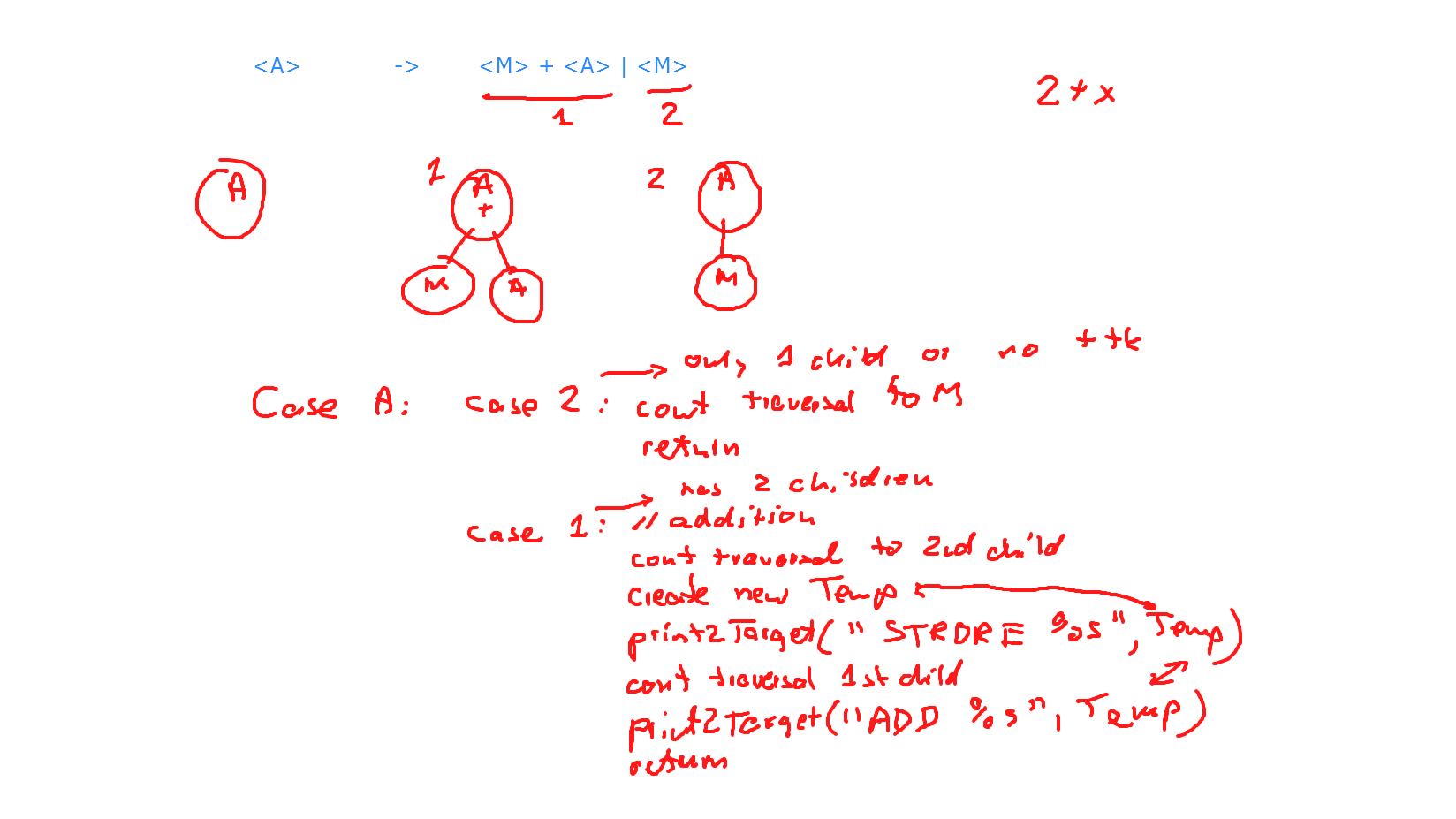
## **Code generation**

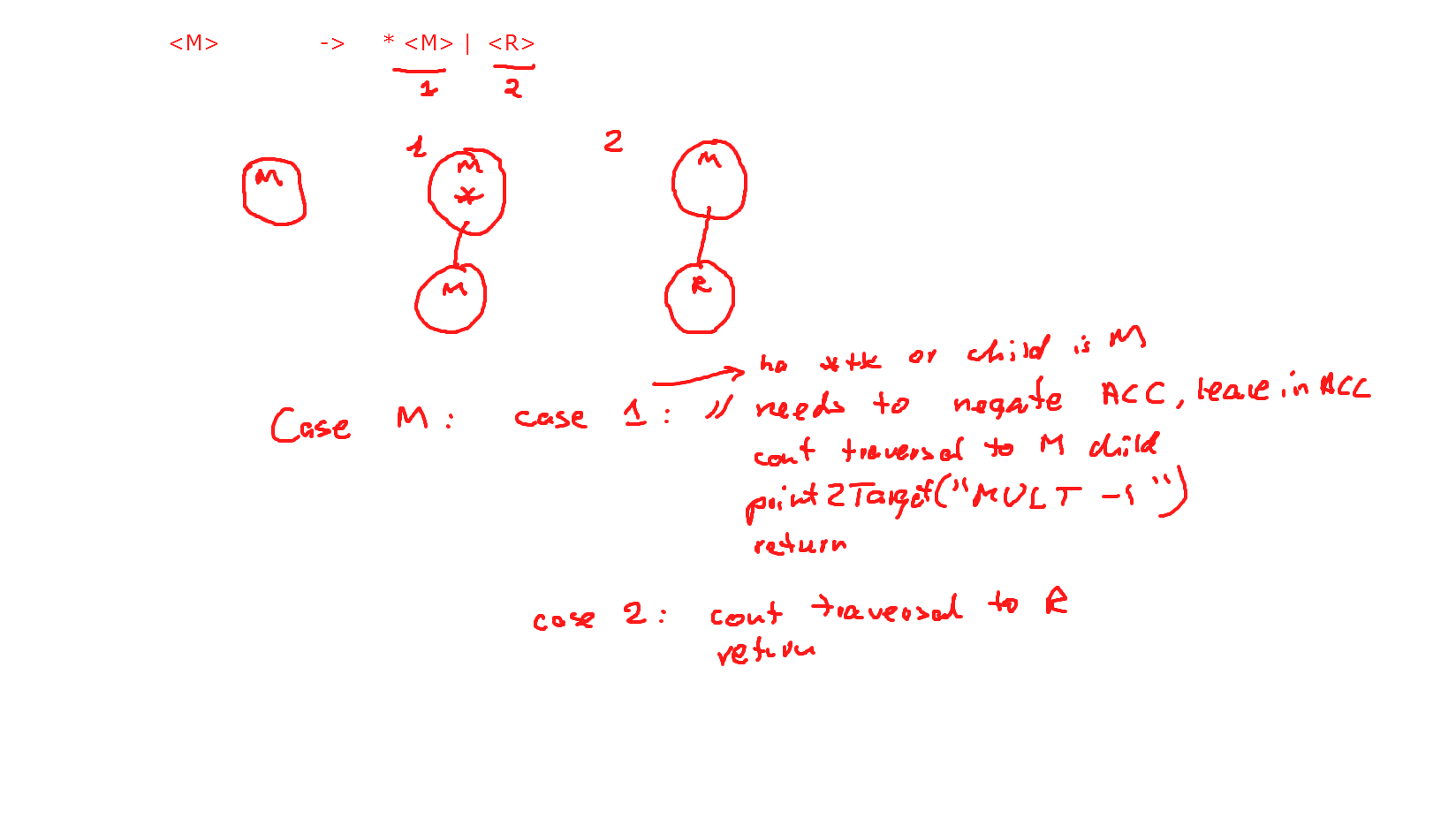
* The parse tree is equivalent to the program in left-to-right traversal (skipping syntactic tokens if not stored in the tree). Therefore, perform left to right traversal to generate the target code
  + the traversal can follow static semantics traversal or be combined
* When visiting a node - some nodes are
  + code generating - most likely only those that have a token(s)
  + not code generating - most likely without tokens but consider one at a time
* When visiting a node not generating any code
  + if no children, return
  + if children, continue traversal (likely left to right) then return
* When visiting a code generating node
  + some actions can be preorder, some inorder, some postorder
  + if value is produced, always leave the result in the ACCumulator
  + if value was produced by another node, find it in the ACCumulator
  + if value in the ACCumulator needs to be saved, store it in a new temporary variable with proper name
    - To simplify, let us assume users never use variables starting with V or T so we can use them for temporary variables without conflicts with user variables
  + each node with a given label always generates the same code
    - regardless of parents and children
    - may be one of multiple cases, one per production
    - the only difference may come from the token(s) found in the node
* At the end of the traversal, print STOP to target
  + to be followed by global variables+temporaries in storage allocation
* Variables will require
  + variable creation upon definition - see storage allocation
  + variable access upon use  - global option or temporary variable  just uses the variable name.

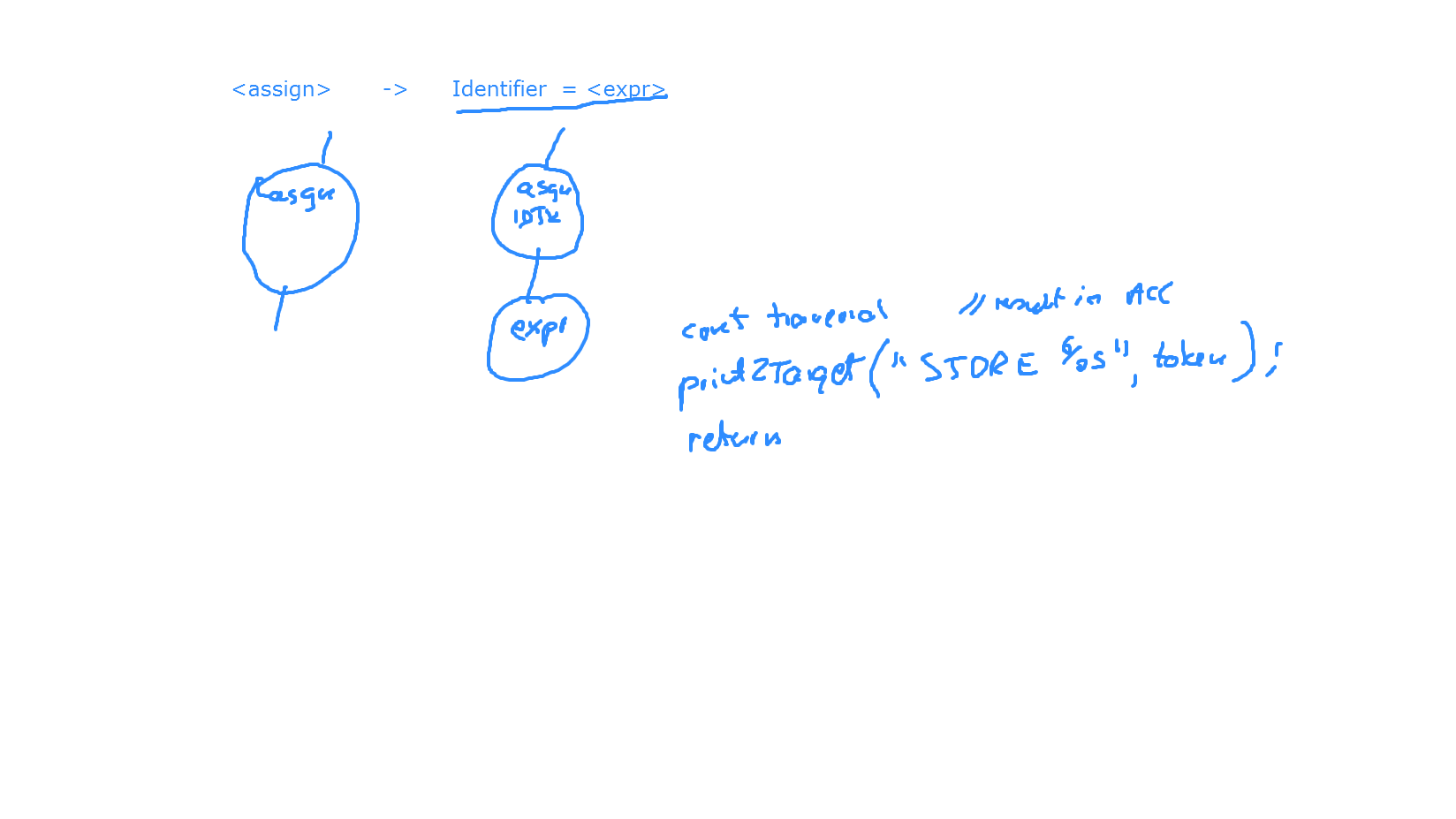
Below is code fragment that can be reused - how to create new names for temporary variables (start with T) and labels (start with L).  The traversal does different things based on the node label. If needed it generates new temporary variable or new label. Keep in mind the generation of variable or label and its use may be separated by a call to another node that may generate new variables and labels so these must be saved locally not globally.

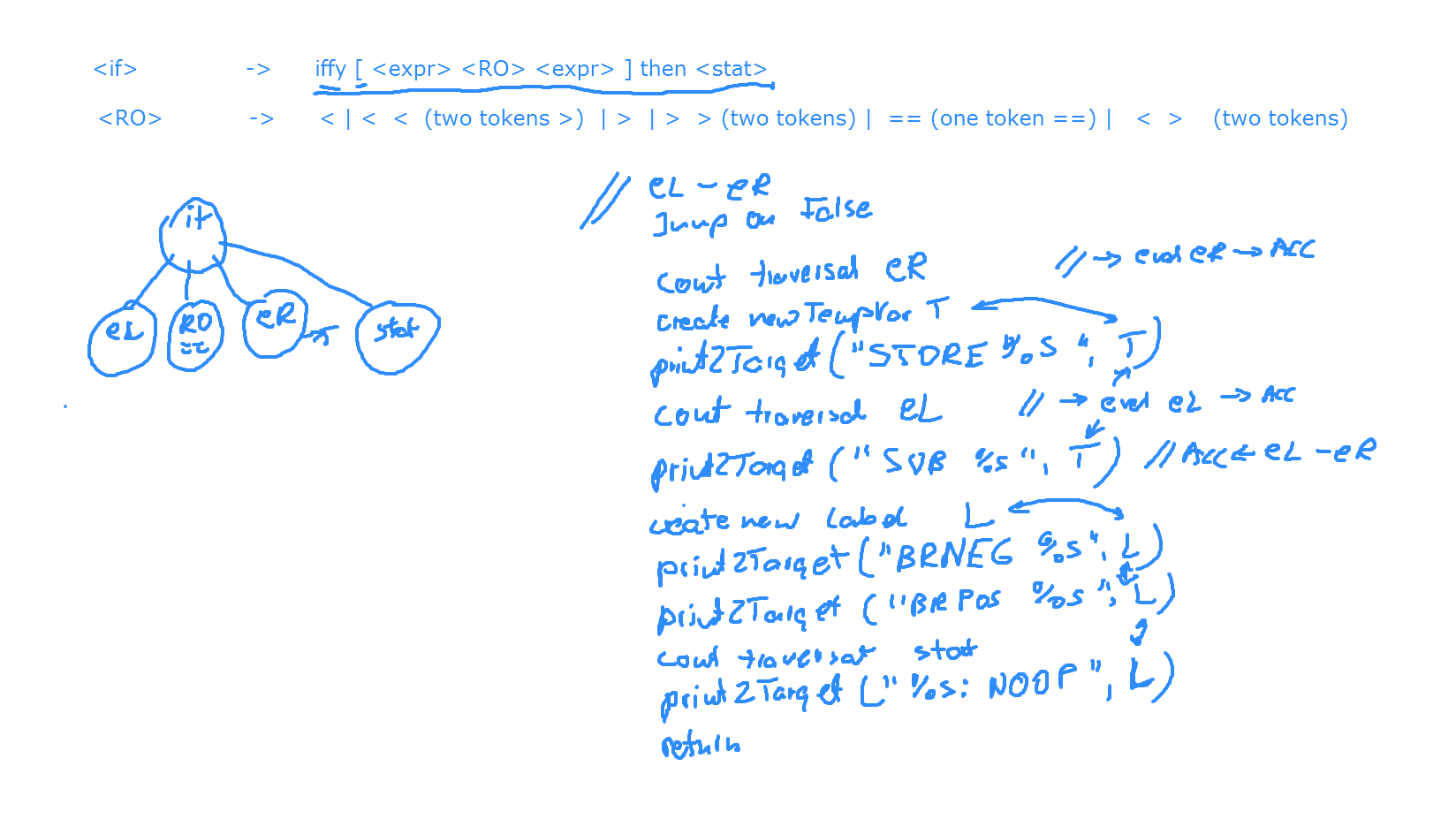
static LabelCntr=0; /\* counting unique labels generated \*/  
static VarCntr=0; /\* counting unique temporaries generated \*/  
typedef enum {VAR, LABEL} nameType;  
static char Name[20]; /\* for creation of unique names \*/  
  
static char \*newName(nameType what)  
{ if (what==VAR) // creating new temporary  
 sprintf(Name,"T%d",VarCntr++); /\* generate a new label as T0, T1, etc \*/  
 else // creating new Label  
 sprintf(Name,"L%d",LabelCntr++); /\* new lables as L0, L1, etc \*/  
 return(Name);  
}  
  
static void recGen(nodeType \*nodeP,FILE \*out) // recursive traversal   
{ char label[20], label2[20], argR[20]; // local storage for temporary or label  
 if (nodeP==NULL)  
 return; // no nodes  
 switch (nodeP->nodeId) // perform different actions based on the node label  
 { case readNode: fprintf(out,"\tREAD\t%s\n",nodeP->str);  
 break;  
 case assignNode: recGen(stats->child1,out); /\* evaluate rhs \*/  
 fprintf(out,"\tSTORE\t%s\n",nodeP->tokenP.str);  
 break;  
 case ifNode: recGen(nodeP->child3,out); /\* exprRight \*/  
 argR = newName(VAR);  
 recGen(nodeP->child1,out); /\* exprLeft \*/  
 fprintf(out,"%SUB %s\n",argR); /\* ACC <- exprLeft - exprRight \*/  
 label = newName(LABEL);  
 if (codeP->child2->token == ==Tk) { /\* False is ACC != 0 \*/  
 fprintf(out,"BRNEG %s\n",label);  
 fprintf(out,"BRPOS %s\n",label);  
 }  
 recGen(nodeP->child3,out); /\* dependent statements \*/  
 fprintf(out,"%s:\tNOOP\n",label);  
 break;  
// etc.

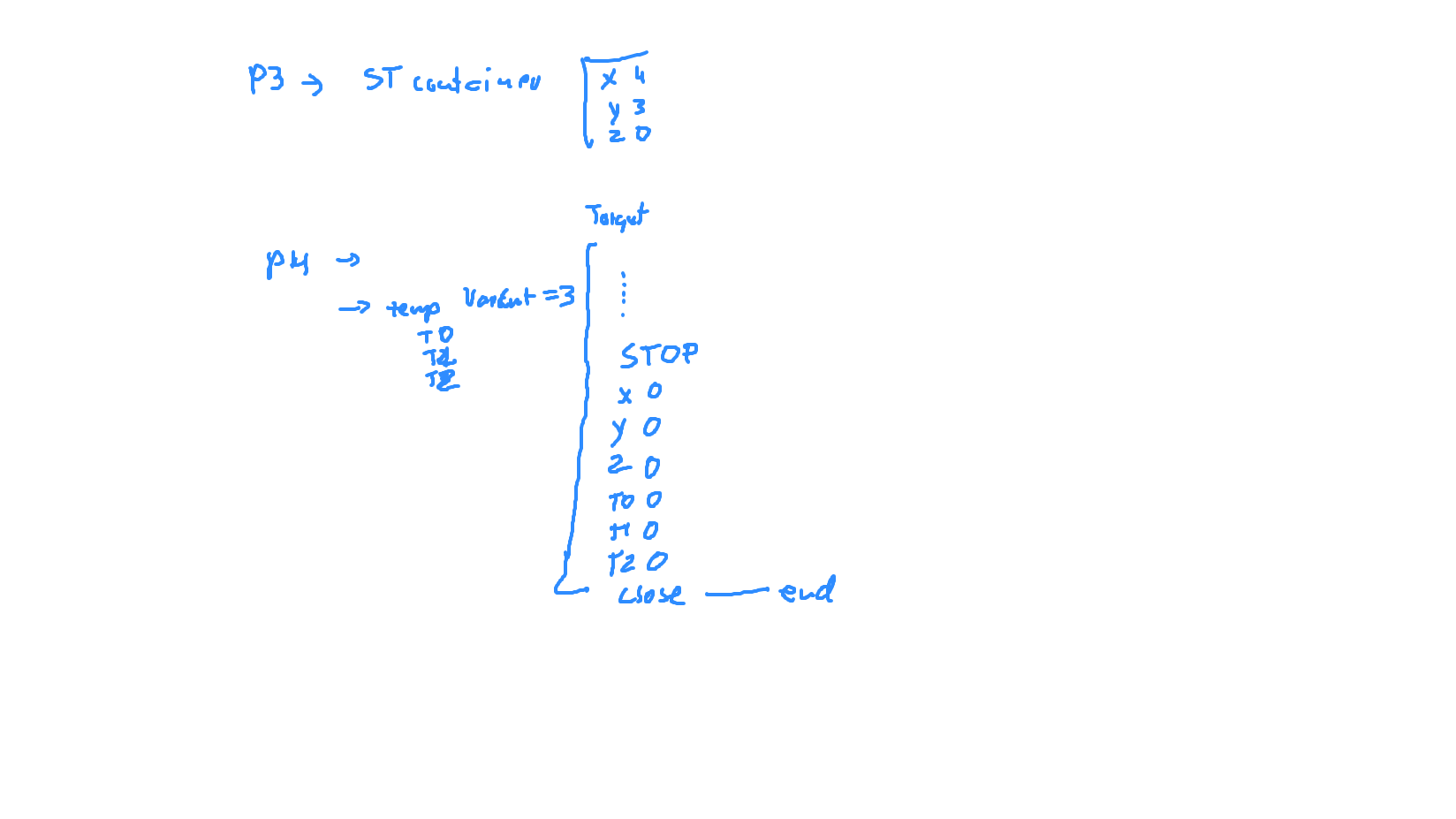












# **P4 Test Files - Code Generation**

P1 Out: Print 1

begin  
 out 1 .  
end

P2 In+Out: Echo input

data x = 1 .  
begin  
 in x .  
 out x .  
end

P3 Asgn+Out: Print 10

data x = 1 .  
begin  
 x = 10 .  
 out x .  
end

P4 ExprNegation: Print -1

begin  
 out \* \* \* 1 .  
end

P5 In+Out+Expr: Echo input

data x = 1 .  
begin  
 in x .  
 out x + 1 - 1 .  
end

P6 ExprComplex: Echo half input (floor)

data x = 1 .  
begin  
 in x .  
 out x / 5 + \* 5 + 2 \* ( 2 - 1 ) .  
end

P7 If: Echo abs val

data x = 1 .  
begin  
 in x .  
 iffy [ x < 0 ] then  
 x = \* x . .  
 out x .  
end

P8 If: Echo abs val

data x = 1 .  
begin  
 in x .  
 iffy [ x < 9 / 2 - 4 ] then  
 x = \* x . .  
 out x .  
end

P9 If nesteed: Echo if [1-10)

data x = 1 .  
begin  
 in x .  
 iffy [ x < 10 ] then  
 iffy [ x > > 0 ] then   
 out x . . .  
end

P10 Loop: Print input then input-1 etc down to 1

data x = 1 .  
begin  
in x .  
loop [ x > 0 ]  
 begin  
 out x .  
 x = x - 1 .  
 end .  
end

P11 Loop nested: Print input input-2 down to 1, then input -1 input-3 down to 1 etc

data x = 1 .  
data y = 2 .  
begin  
 in x .  
 loop [ x > 0 ]  
 begin  
 y = x .  
 loop [ y > 0 ]  
 begin  
 out y .  
 y = y - 2 .  
 end .  
 x = x - 1 .  
 end .  
end

# **P4 Storage Allocation Support**

## **Storage allocation**

* All storage is 2-byte signed
* Storage needed for
  + program variables in the input
  + temporaries (e.g., if accumulator needs to be saved for later use)
    - We can assume input programs do not use variables named T# or V# in the source, reserving such names for temporary variables.
    - there is no need to optimize reducing the number of temporaries
* **Global option**
  + Storage allocation should follow static semantics and code generation
  + Issue target storage directive for every variable and then every temporary, using the global storage directive in the virtual machine after the STOP instruction

# **P4 Test Files - Storage Allocation Global**

S1 Out: Echo 3 inputs in reverse

data x = 1 .  
data y = 2 .  
data z = 3 .  
begin  
 in x .  
 in y .  
 in z .  
 out z .  
 out y .  
 out x .  
end

S2 Out: Echo 3 inputs in reverse

data x = 1 .  
data y = 2 .  
begin  
 data z = 3 .  
 in x .  
 begin   
 in y .  
 in z .  
 out z .  
 end   
 out y .  
 out x .  
end