Assignment Statement

Given an assignment statement of form “variable ‘:’ ‘=’ expression” from the source language rules. We will first evaluate the expression and then push the address of the variable to the top of the stack. Then we can use the VM’s store instruction to store the value of the expression to the variables address.

Example:
a := 17 + 5

The machine code is:
<code to calculate value of the expression 17+5>
ADDR LL offset_a
SWAP
STORE

If Statements

There are two cases of if statements to handle.

CASE 1: “’if’ expression ’then’ statement ’fi’”
We will first generate code to evaluate the expression, then we will generate code to branch to a reserved label (named engIfLabel) if the expression evaluates to false, we then generate the code for the statements, and then we add a label endIfLabel:, so the code skips over the statements if the expression is false.

Example:
if a > b
then
    a := b + a
fi

The machine code will be:
<code to evaluate the expression ‘a > b’>
push endIfLabel
BF
<statements code>
endIfLabel
<rest of code>

CASE 2: "'if' expression 'then' statement 'else' statement 'fi'"
Again similar to the first case we will generate code to evaluate the expression Then we generate code to branch to <falseLabel> if the expression evaluates to false. After that we generate the code for the statements that will be called if the conditional expression is true. Next we have code to branch to the end of the if statement marked by the label endLabel. Then we define the falseLabel which is needed to branch to the statements which will be executed if the conditional expression were to evaluate to false. We generate code for all the statements in this section and finally define a endIfLabel which holds the address for the next line after the if statement.

Example:
```
if a > b
then
        a := b + a
else
        a:= a + 1
fi
```

The machine code will be:
```
<code to evaluate the expression 'a > b'>
push <falseLabel>
BF
<statements code>
push <endIfLabel>
BR
falseLabel
<statements code>
endIfLabel: <rest of code>
```

While and Repeat Statements

WHILE
Given a while loop of form "'while' expression 'do' statement 'end' ". We will first create a label called whileStart which will contain the expression and all its statements followed by a branch to whileStart, we will then evaluate the expression, if the expression is false we will jump to a label called endLoop, which will just be a label that occurs after the branch to whileStart statement.

Example:
```
while not a
do
        a := false
end
```

The machine code will be:
whileStart
<evaluating the expression not a>
push endLoop
BF
<statements, a := false in this case>
push whileStart
BR
endLoop

REPEAT

Given a repeat loop of the form "'repeat' statement 'until' expression". Repeat is very similar to while, except we change the order of the branches. In repeat we will have a label for the beginning called repeatStart, followed by the statements, followed by an evaluation of the expression and a branch to repeatStart if it's false, otherwise we continue execution. We also add a label to the end of the loop called "endLoop". This label holds the address of the first instruction after the loop.

Example:
repeat
a := a + 1
until
a = 10

The machine code will be:
repeatStart
<statements, in this case a := a + 1>
<evaluation of expression, in this case a = 10>
push repeatStart
BF
endLoop

Exit Statements

Simply branch to endLoop label. This label exists for both repeat and while loops.

repeat
a := a + 1
exit
until
a = 10

repeatStart
<statements;a := a+1>
<statement; exit:>
PUSH endLoop (Code for exit statement)
BR (Code for exit statement)
<evaluation of expression, in this case a = 10>
push repeatStart
BF
endLoop

Return and Result Statement

Return and result statements will behave similarly except for the one difference that result statements will need to set the return value before continuing with the rest of the return procedure.

Setting the return value:
In our activation record, the return value is located at a negative offset relative to disp[LL]. Specifically, the return value is under all the arguments, and so if there are n arguments, then we can push the address of the return value onto the top of the stack with ADDR LL -n-1.
Then we can evaluate our return expression as has been explained previously in the expression evaluation section and the value of that variable should now be on the top of the stack.
we can then do STORE to store the value into the return address.

Return procedure:
Following the storage of the return value, we need to do a clean up. This has been described previously in the document under the section on clean up of procedures and functions.

Get and Put Statements

GET

Given a get statement of form "'get' input". By source language rules there are two cases to consider for input.

CASE 1: Input is variable

If the input is a variable, we will need to get the address of the variable, read an integer from standard input, and then call a store. (Note: we will call a swap before the store due to the implementation of store)

Example:
var a : integer
get a

The machine code is:
<code to calculate address of a>
READI
SWAP
STORE

CASE 2: Input is multiple inputs

When input is multiple inputs, it follows a nice recursive structure where we can just call get on each input and have these 2 cases handle it.

PUT

Given a put statement of form "'put' output". By source language rules there are four cases to consider for output.

CASE 1: Output is an expression

If output is an expression we know by definition that it will be of type integer or type boolean. Since booleans are either a 1 or 0 we can treat them as integers for printing purposes. We will evaluate the expression and then call the VM's PRINTI instruction which prints the top element on the stack and pops it.

Example:
var a : boolean
a := true
put a

The machine code is:
<code to evaluate expression>
PRINTI

CASE 2: Output is text

If the output is text, it is stored as designed above (In the constant area with it's start address and length noted), so we simply have to push each character to the stack followed by PRINTC

Example
put "test"

The machine code is
<push the first character from "test" in the constant area to the top of the stack>
PRINTC
<push the next character from "test" in the constant area to the top of the stack>
PRINTC
<push the next character from "test" in the constant area to the top of the stack>
PRINTC
<push the next character from "test" in the constant area to the top of the stack>
PRINTC

CASE 3: Output is 'newline'

When output is newline, we simply need to print the \n character using PRINTC.

Example:
put newline

The machine code is:
PUSH 10 # Assuming the use of ASCII characters, we are pushing 10 to indicate newline
PRINTC

CASE 4: Output is 'output, output'

When output is multiple outputs, it follows a nice recursive structure where we can just call put on each output and have these 4 cases handle it.