Code Generation - Stage 2

Add new alternatives to the **case** statement for `Code` for the new arguments with which it can legitimately be called.

Begin with the revisions to `Code` for the **end** statement since these are quite trivial. In revised production number one, if the punctuation following the keyword `'end'` is a period, then the entire program has terminated; otherwise, the punctuation is a semicolon, and a nested **begin-end** block has just been terminated, but the program should continue. Recall that all programs must end with a period and that the only legal way in which a statement may end with a period is if it is the final `'end'` of the program. No special action is required for a **begin-end** statement terminating with a semicolon.

In the previous stages, *operandStk* has held only operands of the various operators found in Pascallite source code such as `':='`, `'='`, `'div'`, or `'and'`, among many others. We now extend the possible elements of *operandStk* to include RAMM *labels* as well. Labels which stage2 generates will have the form `'Ln'`, where *n* is a non-negative integer. Initially, *n* is `-1` and is incremented by one with each call for a new label, so that the first label is `'L0'`. The need for labels is illustrated back in the figures for the **if**, **repeat**, and **while** structures, which show effectively how the RAMM implementations of these three statements appear. Branch instructions are issued either to skip over code that is to be conditionally executed or to return to the beginning of a loop from its end. In either case, the operand of the branch in RAMM is a label of some other RAMM statement that must be pushed onto *operandStk*.

EmitThenCode()

```
void EmitThenCode(string operand)  //emit code that follows 'then' and statement predicate
{
  string tempLabel;
  assign next label to tempLabel;
  emit instruction to set the condition code depending on the value of operand;
  emit instruction to branch to tempLabel if the condition code indicates operand is zero
    (false);
  push tempLabel onto operandStk so that it can be referenced when EmitElseCode() or
    EmitPostIfCode() is called;
  if operand is a temp then
    free operand's name for reuse;
  deassign operands from all registers
}
```

EmitElseCode()

```
void EmitElseCode(string operand)  //emit code that follows else clause of if statement
{
  string tempLabel;
  assign next label to tempLabel;
  emit instruction to branch unconditionally to tempLabel;
  emit instruction to label this point of object code with the argument operand;
  push tempLabel onto operandStk;
  deassign operands from all registers
}
```

emit_post_if_code

```
void EmitPostIfCode(string operand)  //emit code that follows end of if statement
{
  emit instruction to label this point of object code with the argument operand;
  deassign operands from all registers
}
```

EmitWhileCode()

```
void EmitWhileCode()  //emit code that follows while
{
  string tempLabel;
  assign next label to tempLabel;
  emit instruction to label this point of object code as tempLabel;
  push tempLabel onto operandStk;
  deassign operands from all registers
}
```

EmitDoCode()

```
void EmitDoCode(string operand)  //emit code that follows do
{
  string tempLabel;
  assign next label to tempLabel;
  emit instruction to set the condition code depending on the value of operand;
  emit instruction to branch to tempLabel if the condition code indicates operand is zero
    (false);
  push tempLabel onto operandStk;
  if operand is a temp then
    free operand's name for reuse;
  deassign operands from all registers
}
```

EmitPostWhileCode()

```
void EmitPostWhileCode(string operand1,string operand2)
    //emit code at end of while loop, operand2 is the label of the beginning of the loop,
    //operand1 is the label which should follow the end of the loop
{
  emit instruction which branches unconditionally to the beginning of the loop, i.e., to the
    value of operand2;
  emit instruction which labels this point of the object code with the argument operand1;
  deassign operands from all registers
}
```

EmitRepeatCode()

```
void EmitRepeatCode()  //emit code that follows repeat
{
  var string tempLabel;
  assign next label to tempLabel;
  emit instruction to label this point in the object code with the value of tempLabel;
  push tempLabel onto operandStk;
  deassign operands from all registers
}
```

EmitUntilCode()

```
void EmitUntilCode(string operand1, string operand2)
    //emit code that follows until and the predicate of loop.  operand1 is the value of the
    //predicate.  operand2 is the label that points to the beginning of the loop
{
  emit instruction to set the condition code depending on the value of operand1;
  emit instruction to branch to the value of operand2 if the condition code indicates operand is
    zero (false);
  if operand1 is a temp then
    free operand1's name for reuse;
  deassign operands from all registers
}
```